

**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"**

МИХАЙЛО ЛУЧКЕВИЧ, ІРИНА ШАКЛЕІНА, ОЛЕКСА ГУДЬ

ІНТЕГРАЦІЯ ІНФОРМАЦІЙНИХ СИСТЕМ

НАВЧАЛЬНИЙ ПОСІБНИК

*Рекомендувала Науково-методична рада
Національного університету "Львівська політехніка"*

2025

ББК 32.973.202

Рецензенти: Досин Д.Г., д-р техн. наук, с.н.с., завідувач кафедри інформаційних систем та мереж Національного університету «Львівська політехніка»

Оленич І.Б., д-р фіз.-мат. наук, проф., завідувач кафедри радіоелектронних і комп'ютерних систем Львівського національного університету ім. І. Франка

Сікора О.В., канд. техн. наук, доц., доцент кафедри фізики та інформаційних систем Дрогобицького державного педагогічного університету імені Івана Франка

Рекомендувала Науково-методична рада Національного університету «Львівська політехніка» як навчальний посібник для студентів спеціальності 126 (F 6) «Інформаційні системи та технології» (Протокол № 89 від 20 червня 2025 р.)

Лучкевич М.М., Шаклеїна І.О., Гудь О.О.

Інтеграція інформаційних систем: навчальний посібник – Дрогобич : Коло, 2025. – 400 с.

ISBN 978-617-642-710-0

Посібник є комплексним і структурованим введенням до сучасних практик DevOps та неперервної інтеграції програмних рішень, має на меті формування розуміння сучасних підходів до розробки програмного забезпечення та практичних навичок роботи з системами неперервної інтеграції.

Призначається для студентів, що навчаються за освітньо-професійними програмами першого (бакалаврського) та другого (магістерського) рівня вищої освіти спеціальності 126 (F6) «Інформаційні системи і технології» галузі знань Інформаційні технології та споріднених галузей знань, пов'язаних з вивченням комп'ютерних інформаційних технологій. Може бути використаний викладачами в якості дидактичного матеріалу, а також для самостійного вивчення і підвищення кваліфікації.

ISBN 978-617-642-710-0

© М.М. Лучкевич, І.О. Шаклеїна,

О.О. Гудь, 2025

© КОЛО, 2025

Передмова	6
Розділ 1. Вступ до DevOps	8
1.1. Devops як комбінація Development та Operation	8
1.2. Базові процеси в DevOps.....	9
1.3. Переваги DevOps	24
1.4. Виклики у впровадженні DevOps	26
1.5. Інструменти DevOps.....	28
1.6. Agile Methodology.....	31
1.7. Взаємовідносини Agile та DevOps.....	36
Контрольні питання до розділу 1	38
Розділ 2. Основи неперервної інтеграції.....	42
2.1. Поняття безперервної інтеграції	42
2.2. Безперервна доставка та розгортання коду.....	43
2.3. Порівняння кращих сучасних сервісів неперервної інтеграції..	44
2.4. Переваги неперервної інтеграції	59
Контрольні питання до розділу 2	61
Розділ 3. системи контролю версій	65
3.1. Централізовані та розподілені системи збереження даних	65
3.2. Розподілені системи контролю версій (DVCS)	76
3.3. Поняття Git Flow та GitHub Flow.....	95
Контрольні питання до розділу 3	102
Розділ 4. Платформи для роботи з системами контролю версій .	106
4.1. GitLab	106
4.2. GitHub	118
4.3. Bitbucket.....	124
4.4. Порівняння інструментів для роботи з системами контролю версій.....	129
Контрольні питання до розділу 4.....	135
Розділ 5. Оркестрація неперервної інтеграції	139
5.1. Завдання та переваги оркестрації CI	139
5.2. Переваги оркестрації CI.....	142
5.3. Виклики в оркестрації CI.....	144
5.4. Інструменти для оркестрації CI.....	146
5.5. Порівняльний аналіз основних інструментів оркестрації CI ...	161

Контрольні питання до розділу 5	162
Розділ 6. Інструмент для безперервної інтеграції Jenkins	166
6.1. Типи робіт та застосування	167
6.2. Роль інтеграцій (plug-in) в системі Jenkins	184
6.3. Приклад організації роботи. Автоматизація збірки проєкту	196
6.4. Модель (master-slave) в контексті системи Jenkins	198
6.5. Pipeline: приклади роботи	200
6.6. Підключення оркестрації до системи контролю версіями	203
Контрольні питання до розділу 6	205
Розділ 7. GitHub Actions.....	209
7.1. Історія виникнення GitHub Actions	209
7.2. Розбір інструменту GitHub Actions	212
7.3. Відмінності GitHub Actions від аналогічних систем	220
7.4. Приклади застосування GitHub Actions.....	223
Контрольні питання до розділу 7	232
Розділ 8. Автоматичне тестування.....	236
8.1. Основи автоматичного тестування	236
8.2. Типи автоматичних тестів	240
8.3. Автоматичні тести в CI/CD	250
Контрольні питання до розділу 8	255
РОЗДІЛ 9. Основи код аналізу	258
9.1. Завдання код-аналізу	258
9.2. Типи код-аналізу	263
9.3. Інтеграція код-аналізаторів з CI/CD.....	270
Контрольні питання до розділу 9	273
Розділ 10. SonarQube	277
10.1. Призначення інструменту та особливості застосування.....	277
10.2. Архітектура побудови SonarQube	283
10.3. SonarScanner.....	286
10.4. Quality Gates (Ворота якості)	288
10.5. Приклади застосування SonarQube	291
Контрольні питання до розділу 10	295
Розділ 11. Огляд мінімально укомплектованої CI-системи.....	299
11.1. Вимоги до CI-системи	299
11.2. Використання Jenkins для побудови CI-системи.....	300
11.3. Використання GitHub Action для побудови CI-системи.....	314

Контрольні питання до розділу 11	318
Розділ 12. Практичні завдання, пов'язані з інтеграцією ІС	322
12.1. Вибір проєкту та налаштування робочого середовища.....	322
12.2. Робота з Github репозиторієм	327
12.3. Робота з репозиторіями у GitLab	332
12.4. Робота з Jenkins. Налаштування Pipeline сценарію.....	334
12.5. Робота з інструментом GitHub Actions.....	340
12.6. Робота з інструментом SonarQube	344
12.7. Робота з агентом у Jenkins	352
12.8. Використання інтеграцій SonarQube у Jenkins	359
Відповіді на контрольні питання	369
Глосарій термінів	384
Список використаної та рекомендованої літератури.....	396

ПЕРЕДМОВА

У сучасному світі цифрових технологій інформаційні системи відіграють ключову роль у трансформації бізнес-процесів, управлінських практик і щоденного життя. Стрімкий розвиток технологій штучного інтелекту, хмарних обчислень, мікросервісної архітектури та контейнеризації кардинально змінює підходи до розробки, тестування, розгортання та підтримки програмного забезпечення. Водночас зростаюча складність ІТ-інфраструктур вимагає від фахівців не лише глибоких знань у сфері програмування, адміністрування та безпеки, а й уміння ефективно інтегрувати різноманітні системи, сервіси та компоненти в єдиний, цілісний інформаційний простір.

Сучасні організації потребують швидкої адаптації до мінливих ринкових умов, що зумовлює необхідність скорочення циклів розробки та впровадження нових функціональностей. Традиційні методології розробки програмного забезпечення, засновані на послідовних фазах планування, реалізації, тестування та розгортання, вже не відповідають вимогам сучасного бізнесу. Натомість, на перший план виходять гнучкі методології (Agile), практики неперервної інтеграції (Continuous Integration), неперервної доставки (Continuous Delivery) та неперервного розгортання (Continuous Deployment), які об'єднуються під загальною назвою DevOps.

DevOps-культура передбачає тісну співпрацю між командами розробки (Development) та експлуатації (Operations), спрямовану на автоматизацію процесів, підвищення якості програмного продукту та прискорення його виходу на ринок. Цей підхід не обмежується лише технічними аспектами, а охоплює організаційні зміни, культурні трансформації та впровадження нових процесів управління проектами.

Цей навчальний посібник підготовлено для студентів спеціальності 126 (F6) «Інформаційні системи і технології та споріднених галузей знань, пов'язаних з вивченням комп'ютерних інформаційних технологій. Метою посібника є формування комплексного розуміння сучасних підходів до розробки програмного забезпечення, ознайомлення з базовими принципами інтеграції інформаційних систем, актуальними технологіями автоматизації, засобами DevOps, CI/CD, системами контролю версій, інструментами тестування, аналізу коду та моніторингу. Матеріали цього видання можуть бути корисними не лише студентам, а й викладачам, розробникам програмного забезпечення, інженерам DevOps та всім, хто прагне опанувати сучасні підходи до інтеграції інформаційних систем (IS) у контексті швидкозмінного технологічного середовища.

Структура посібника відображає логіку побудови сучасного середовища розробки та експлуатації програмного забезпечення. Починаючи з

фундаментальних концепцій DevOps та неперервної інтеграції, матеріал поступово розкриває специфіку роботи з системами контролю версій, платформами для управління кодом та інструментами оркестрації CI/CD процесів. Особливу увагу приділено практичним аспектам застосування таких інструментів, як Git, GitHub, GitLab, Bitbucket, Jenkins, GitHub Actions, SonarQube, а також організації процесів автоматичного тестування, статичного аналізу коду, розгортання й контролю якості програмного продукту.

Навчальний посібник має чітку методичну структуру, яка сприяє поступовому засвоєнню матеріалу. Кожен розділ містить теоретичні основи, практичні приклади, детальні інструкції з налаштування та використання інструментів а також контрольні питання для самоперевірки. Практична частина посібника включає комплексні завдання, які дозволяють студентам отримати реальний досвід роботи з професійними інструментами та технологіями.

Особлива увага приділена актуальним тенденціям у сфері інформаційних технологій, зокрема використанню контейнерних технологій, хмарних платформ, мікросервісної архітектури та сучасних практик забезпечення якості програмного забезпечення.

Посібник розрахований на студентів, які вже мають базові знання з програмування, алгоритмів та структур даних, основ комп'ютерних мереж та операційних систем. Водночас матеріал викладено доступною мовою з поступовим нарощуванням складності, що дозволяє ефективно засвоювати нові концепції та технології.

Практична спрямованість посібника забезпечується через використання реальних кейсів та проєктів, які відображають типові завдання, з якими стикаються фахівці в галузі інформаційних технологій. Студенти отримають можливість не лише теоретично вивчити принципи роботи різних інструментів, а й набути практичних навичок їх налаштування, конфігурування та використання в реальних умовах розробки програмного забезпечення.

Розділи посібника супроводжуються численними прикладами, практичними завданнями та питаннями для контролю, що дозволяє ефективно поєднати теоретичні знання з практичними навичками. Глосарій термінів та список рекомендованої літератури допоможуть поглибити свої знання та продовжити самостійне вивчення матеріалу.

Щиро сподіваємося, що цей посібник стане надійним інструментом у процесі професійного становлення здобувачів освіти, сприятиме формуванню сучасного мислення щодо розробки програмного забезпечення та допоможе підготувати висококваліфікованих фахівців у галузі інформаційних технологій, здатних ефективно працювати в умовах цифрової трансформації економіки та суспільства.

РОЗДІЛ 1. ВСТУП ДО DEVOPS

1.1. Devops як комбінація Development та Operation

У сучасному світі інформаційних технологій, де швидкість впровадження інновацій є критичною для конкурентоспроможності, виникла необхідність у нових підходах до розробки та підтримки програмного забезпечення. Постійний розвиток цифрових технологій вимагає швидкої адаптації до змін на ринку, забезпечення стабільності ІТ-систем і безперервного вдосконалення програмних рішень. Традиційні моделі розробки та експлуатації, які працюють окремо одна від одної, часто стають причиною затримок, технічних помилок і непорозумінь між командами.

Для подолання цих викликів у 2009-му році була розроблена та вперше представлена методологія DevOps, яка мала на меті інтеграцію процесів розробки програмного забезпечення із процесами його підтримки та експлуатації.

Цей підхід включає не лише технічну оптимізацію процесів, але й зміну культурного середовища в організаціях. DevOps стає ключовим елементом трансформації ІТ-індустрії, сприяючи швидшому випуску оновлень, ефективнішому використанню ресурсів і, в кінцевому підсумку, підвищенню задоволеності кінцевих користувачів. Зазначений підхід не лише прискорює впровадження нових функцій, але й дозволяє підвищити стабільність систем, знизити ризики помилок і забезпечити якісну взаємодію між усіма учасниками процесу розробки.

DevOps – це сучасна методологія, яка виникла як відповідь на зростаючі виклики в галузі інформаційних технологій, де важливими факторами є швидкість, якість і гнучкість. Назва «DevOps» походить від поєднання слів *Development* (розробка) та *Operations* (експлуатація), що символізує інтеграцію цих двох важливих складових життєвого циклу програмного забезпечення.

Також можна зустріти варіації цієї методології такі як «DevSecOps», що походить від додавання слова «Security» та вказує на безпековий фокус, або «DevNetOps», орієнтований на інтеграцію мережових операцій (Network Operations).

DevOps не є лише набором технологій або інструментів – це передусім культурна трансформація в організаціях, яка змінює підходи до взаємодії між командами розробників, тестувальників, операційних інженерів та інших учасників процесу. Традиційно розробка програмного забезпечення та його експлуатація виконувалась ізольованими учасниками, в різних середовищах, що зумовлювало надскладну процедуру злиття змін та перетворювало етап випуску продукту на тривалий процес пошуку й усунення проблем.

DevOps спрямований на усунення цих бар'єрів шляхом створення спільної культури, де всі працюють над однією метою – швидкою та стабільною доставкою програмного забезпечення, що відповідає вимогам користувачів.

Фокусом DevOps є автоматизація, стандартизація та оптимізація всіх наступних етапів життєвого циклу розробки програмного забезпечення (ПЗ):

- *створення коду*: забезпечення прозорості та інтеграції коду через системи управління версіями, такими як Git;
- *тестування*: впровадження автоматизованого тестування для виявлення помилок на ранніх етапах розробки;
- *збірка та розгортання*: автоматизація процесу інтеграції (Continuous Integration) та доставки оновлень (Continuous Delivery);
- *експлуатація*: підтримка високої доступності та стабільності систем за допомогою моніторингу і автоматичного масштабування.

Основною метою методології DevOps є отримання інформації про якість змін в коді чи їх відповідність існуючій системі настільки швидко, наскільки це можливо.

Дана методологія також ґрунтується на принципі безперервного вдосконалення, який передбачає регулярний аналіз процесів, отримання зворотного зв'язку від користувачів та адаптацію до нових викликів. Завдяки цим підходам DevOps дозволяє скоротити час між створенням нової функції або виправленням помилки та її впровадженням у робоче середовище.

Крім того, DevOps інтегрує використання спеціалізованих інструментів, таких, наприклад, як Docker, Jenkins, Kubernetes тощо, які сприяють автоматизації й стандартизації процесів. Це робить його одним із найпотужніших підходів до управління програмними проєктами в сучасній ІТ-галузі.

Таким чином, DevOps можна визначити як симбіоз культурних змін, технічних інновацій та інструментів, які разом забезпечують ефективну розробку та експлуатацію програмного забезпечення.

1.2. Базові процеси в DevOps

Процес DevOps охоплює низку етапів, які забезпечують безперервну інтеграцію, доставку, розгортання та моніторинг програмного забезпечення. Кожен етап має на меті оптимізацію співпраці між командами розробки та операцій, забезпечуючи швидке, якісне та безпечне впровадження змін.

В таблиці 1.1 наведено основні етапи процесу DevOps, успішне виконання яких допомагає створювати ефективне середовище для реалізації життєвого циклу програмного забезпечення.

DevOps ґрунтується на впровадженні п'яти ключових процесів, які забезпечують безперервний розвиток, тестування, доставку, розгортання та моніторинг програмного забезпечення. Ці процеси сприяють швидкості, якості та стабільності розробки, створюючи основу для ефективної роботи DevOps-команди.

Таблиця 1.1. Основні етапи процесу DevOps

Етап	Опис
Планування	Визначення вимог до продукту, планування функцій і завдань для реалізації.
Розробка	Написання коду, створення нових функцій та інтерфейсів.
Тестування	Виконання автоматизованих тестів для перевірки якості та стабільності продукту.
Розгортання	Автоматичне розгортання коду в продуктивне середовище після успішного тестування.
Моніторинг	Безперервний моніторинг продуктивності та стабільності системи після розгортання.
Зворотний зв'язок	Збір даних від користувачів та адаптація продукту згідно з отриманими результатами.

1.2.1. Безперервна інтеграція

Безперервна інтеграція (Continuous Integration, CI) є важливою частиною DevOps-практики, яка забезпечує систематичний підхід до інтеграції змін у базовий код застосунку. Вона спрямована на підтримання стабільності проєкту, мінімізацію ризиків помилок і забезпечення постійної готовності до подальших етапів розробки та доставки програмного забезпечення.

Основні принципи безперервної інтеграції:

- *часте об'єднання змін*: розробники інтегрують свої зміни до центрального репозиторію кілька разів на день. Це дозволяє уникнути проблем, пов'язаних із великими, рідкими оновленнями коду, які можуть спричинити конфлікти;
- *автоматичне тестування*: з кожним комітом система автоматично запускає набір тестів (юніт-, інтеграційних тощо), що допомагає виявляти помилки на ранньому етапі;
- *системи контролю версій*: використання таких інструментів, як Git, забезпечує організоване управління змінами, дозволяє відстежувати історію змін і легко повертатися до попередніх версій коду, якщо виникають проблеми;

- *автоматизація процесу збірки*: інструменти CI забезпечують автоматичну збірку програми після кожного коміту, що дозволяє виявляти помилки ще до етапу тестування.

Інструменти для безперервної інтеграції є невід’ємною частиною сучасного процесу розробки програмного забезпечення в рамках DevOps-практик. Вони дозволяють автоматизувати багато етапів розробки, тестування та доставки коду, що дає змогу значно зменшити час на випуск нових релізів і підвищити їхню якість. Завдяки таким інструментам, як Jenkins, GitLab CI/CD, CircleCI, GitHub Actions чи Travis CI команди можуть швидше інтегрувати зміни, тестувати нові функції та забезпечувати доставку коду в робочі середовища.

Оскільки кожен інструмент має свої унікальні можливості та налаштування, вибір відповідного сервісу залежить від специфіки проекту, вимог до швидкості, гнучкості та масштабованості процесів.

Серед інструментів для CI можна виділити наступні:

- *Jenkins* – відкритий інструмент для автоматизації, який підтримує широкий спектр плагінів для налаштування процесів CI. Його можна інтегрувати з більшістю систем контролю версій і хмарних платформ;
- *GitLab CI/CD* – вбудований у GitLab інструмент, який дозволяє автоматизувати процес інтеграції, тестування та доставки, спрощуючи управління DevOps-процесами;
- *CircleCI* – хмарна CI-платформа, яка відома своєю швидкістю та гнучкістю. Підтримує інтеграцію з популярними платформами, такими як GitHub і Bitbucket;
- *Travis CI* – простий у використанні інструмент для CI, який інтегрується з GitHub і надає можливості для автоматизації тестування та збірки додатків.

Безперервна інтеграція приносить численні переваги для команд, що займаються розробкою програмного забезпечення. Вона не лише допомагає автоматизувати процеси тестування та інтеграції коду, але й суттєво підвищує ефективність і якість розробки. Завдяки CI команди можуть виявляти помилки на ранніх етапах, забезпечувати прозорість у роботі та значно прискорювати цикл розробки. Крім того, регулярне тестування та автоматизація допомагають підтримувати високу якість коду і знижують ризик виникнення конфліктів. У результаті, CI стає потужним інструментом, який сприяє ефективності роботи та покращенню результатів команди.

Автоматизоване тестування кожного коміту дозволяє виявляти помилки на ранніх етапах внесення змін в код, що мінімізує ризик їх поширення на наступні етапи і знижує загальний час на виправлення дефектів.

Важливою перевагою є прозорість процесів, оскільки вся інформація про зміни в коді доступна для кожного члена команди в реальному часі. Це сприяє кращому розумінню прогресу проекту та зменшує ймовірність виникнення непорозумінь.

Часті коміти та автоматизація інтеграційного процесу значно прискорюють загальний цикл розробки, зменшуючи час на злиття змін і дозволяючи швидше інтегрувати нові функціональності.

Також, CI сприяє покращенню якості коду завдяки постійному тестуванню і швидкому виявленню потенційних проблем, що допомагає командам підтримувати високу якість продукту.

Завдяки безперервній інтеграції зменшується час, необхідний для вирішення конфліктів у коді. Це дозволяє розробникам швидше усувати технічні проблеми, які виникають через інтеграцію змін, і тим самим підвищує ефективність процесу розробки.

Типовий процес безперервної інтеграції полягає в тому, що розробники спочатку пишуть код і створюють юніт-тести. Потім зміни заливаються у віддалений репозиторій, наприклад, Git. Інструмент CI, такий як Jenkins або GitLab CI/CD, автоматично виявляє нові коміти і запускає процеси, включаючи збірку програми та виконання автоматизованих тестів. Результати тестів відображаються у вигляді звіту.

Якщо в системі присутній інструмент аналізу коду, такий як SonarQube, якість коду оцінюється автоматично з огляду на такі властивості, як наявність коментарів, відсутність оптимізації чи наявність поширених помилок та вразливостей. Якщо тести проходять успішно, а якість коду задовільна – програма готова до наступного етапу, такого як доставка. У разі виникнення помилок розробники отримують сповіщення для негайного виправлення проблеми.

Continuous Integration є фундаментальною практикою, яка дозволяє забезпечити стабільність і якість програмного забезпечення на кожному етапі розробки. Завдяки використанню CI команди можуть працювати швидше, ефективніше та з мінімальними ризиками, що робить цей процес ключовим компонентом DevOps-культури.

1.2.2. Безперервна доставка

Безперервна доставка (Continuous Delivery, CD) є важливим етапом у DevOps-циклі, який фокусується на автоматизації всіх процесів, пов'язаних з доставкою та впровадженням змін у робоче середовище. CD забезпечує надійне

і швидке розгортання готового коду в тестових або виробничих середовищах, знижуючи ризики і підвищуючи ефективність усієї команди.

Основні принципи безперервної доставки:

- *готовність коду до розгортання в будь-який момент*: після того як код пройшов автоматизоване тестування в рамках Continuous Integration, він автоматично стає готовим до розгортання. Кожна зміна в коді, яка проходить через усі етапи тестування, є готовою для потенційного випуску без додаткових перевірок;
- *автоматизація процесу розгортання*: після того, як код успішно пройшов тестування, він автоматично розгортається у відповідному середовищі – від тестового до передвиробничого. Процес розгортання автоматизується, що мінімізує людський фактор і забезпечує швидке і безпомилкове розгортання на різних етапах. Проте останній етап (впровадження в виробниче середовище) може бути виконаний вручну, якщо є потреба в додатковій перевірці або контролі;
- *відокремленість процесів розгортання та впровадження*: хоча автоматизація дозволяє зменшити кількість помилок, рішення про остаточне впровадження в реальне середовище часто залишається за розробниками або операційною командою. Це дозволяє зберегти контроль над критичними моментами, але в той же час дозволяє максимально скоротити час на підготовку до релізу.

Безперервна доставка (CD) є важливою складовою частиною сучасних DevOps-практик, що забезпечує автоматизоване і швидке впровадження змін до програмних продуктів у різні середовища. Для досягнення цієї мети використовуються потужні інструменти, які дозволяють автоматизувати процеси упаковки, розгортання та масштабування додатків. Вони значно спрощують і прискорюють розгортання, забезпечуючи узгодженість між тестовими і виробничими середовищами, а також підвищуючи ефективність управління інфраструктурою.

В якості прикладів інструментів для CD можна виділити наступні:

- *Docker* – один з основних інструментів для створення контейнеризованих додатків, що дає можливість упаковувати додатки і всі необхідні залежності у єдині порти. Завдяки Docker контейнери є портативними, що дозволяє розгортати додатки в різних середовищах без змін у коді. Це забезпечує узгодженість між тестовими і виробничими середовищами;
- *Kubernetes* – це система для оркестрації контейнерів, яка автоматизує процес розгортання, масштабування і керування контейнеризованими додатками. Kubernetes підтримує високу доступність, автоматичне

масштабування та відновлення збоїв, що робить його ідеальним для використання в рамках безперервної доставки;

- *AWS CodePipeline* – хмарний сервіс від Amazon для автоматизації процесу розгортання. AWS CodePipeline дозволяє налаштувати етапи CI/CD, включаючи автоматичне тестування, збірку і доставку, що дозволяє швидко і надійно впроваджувати зміни в різні середовища;
- *Azure Pipelines* – хмарний сервіс від Microsoft для автоматизації CI/CD процесів, частина екосистеми *Azure DevOps*. Azure Pipelines дозволяє налаштувати етапи безперервної інтеграції та доставки, включаючи автоматичне тестування, збірку і розгортання. Завдяки інтеграції з іншими сервісами екосистеми Azure DevOps та підтримці популярних інструментів і платформ, це дозволяє швидко і надійно впроваджувати зміни в різні середовища, зменшуючи витрати часу на ручне керування процесами доставки.

Безперервна доставка відіграє важливу роль у сучасних розробках програмного забезпечення, дозволяючи швидко та надійно впроваджувати зміни у виробничі середовища. Автоматизація процесів розгортання й тестування забезпечує безпеку, стабільність та ефективність, зменшуючи ризики помилок і підвищуючи якість кінцевого продукту. Завдяки використанню CD компанії можуть оперативно реагувати на зміни, швидко додавати нові функції та покращувати досвід користувачів. Нижче наведено основні переваги безперервної доставки.

Завдяки автоматизації процесів кількість помилок, пов'язаних з людським фактором, значно зменшується. Автоматичне тестування та перевірка коду перед доставкою в середовище виробництва знижують ймовірність виникнення критичних помилок після релізу.

Безперервна доставка забезпечує швидке впровадження нових функцій і оновлень без зупинок у роботі системи. Це дозволяє компаніям швидко реагувати на зміни на ринку і задовольняти потреби користувачів, знижуючи час затримки для нових можливостей. Таким чином кінцеві користувачі отримують нові функції, виправлення помилок і покращення без значних затримок. Це значно поліпшує їхній досвід взаємодії з продуктом.

Автоматизація всіх етапів доставки знижує варіативність і неочікувані збої, що часто виникають при ручному розгортанні. Кожен реліз відбувається за однаковою перевіреною процедурою, що підвищує передбачуваність результатів.

Постійні перевірки через автоматичні тести дозволяють швидко виявляти порушення сумісності з існуючими версіями продукту, що забезпечує стабільність і знижує ризик негативного впливу на існуючих користувачів.

Типовий процес безперервної доставки (Continuous Delivery) включає кілька етапів, що забезпечують швидке та надійне впровадження змін у виробниче середовище.

Першим етапом є інтеграція змін у загальний репозиторій після завершення процесу безперервної інтеграції (CI). Система автоматично проходить етапи тестування та збірки, що дозволяє перевірити правильність інтеграції та підготувати код до наступних кроків.

Далі код перевіряється через автоматичні юніт-тести, інтеграційні тести та інші типи тестування, що гарантує високий рівень якості та працездатності продукту перед його впровадженням у тестове середовище.

Після проходження тестів код автоматично розгортається в тестове або передвиробниче середовище для проведення додаткових перевірок. Це дає змогу виявити будь-які проблеми, які можуть виникнути в умовах, наближених до реальних.

Результат побудови поточної версії проекту, або «артефакт», надсилається на збереження в спеціальне сховище, так званий «артефакторій», щоб забезпечити швидкий доступ до нього в разі потреби.

Останнім етапом є готовність коду до релізу в виробничому середовищі. Після успішного розгортання у тестовому середовищі код вважається готовим до релізу.

Безперервна доставка – це потужна стратегія, яка значно підвищує ефективність і швидкість впровадження нових функцій та оновлень. Вона мінімізує ризики, пов'язані з людським фактором, і дозволяє компаніям бути більш гнучкими в реагуванні на зміни на ринку. У поєднанні з безперервною інтеграцією та автоматизованими тестами CD робить розробку і доставку програмного забезпечення значно швидшою і надійнішою.

1.2.3. Безперервне розгортання

Безперервне розгортання (Continuous Deployment, CD) є наступним кроком після безперервної доставки у DevOps-циклі, який забезпечує автоматичне впровадження змін у виробниче середовище. Головна відмінність безперервного розгортання від безперервної доставки полягає в тому, що останній етап – впровадження в реальне середовище – також виконується автоматично, без необхідності ручного втручання. Це дозволяє скоротити час доставки нових функцій до кінцевих користувачів і підвищити загальну швидкість розробки.

Можна виділити наступні основні принципи безперервного розгортання:

- *автоматичне розгортання змін*: усі зміни, які успішно проходять етапи автоматизованого тестування, інтеграції та перевірки, автоматично впроваджуються у виробниче середовище. Це забезпечує швидке розгортання нових функцій, виправлень помилок або оновлень;
- *інтенсивне тестування*: щоб уникнути помилок у виробничому середовищі, безперервне розгортання вимагає надійного набору автоматизованих тестів, які покривають усі критичні аспекти роботи продукту;
- *безперервний моніторинг*: після розгортання змін у виробничому середовищі відбувається постійне моніторинг продуктивності та працездатності системи. Це дозволяє оперативно виявляти та вирішувати можливі проблеми.
- *резервні механізми*: безперервне розгортання часто включає механізми відкату (rollback), які дозволяють швидко повернутися до попередньої стабільної версії продукту у випадку виявлення критичних проблем.

Переваги безперервного розгортання полягають у можливості значно скоротити час впровадження змін завдяки автоматизації всього процесу – від написання коду до його розгортання у виробниче середовище. Це забезпечує швидку реакцію на потреби бізнесу та користувачів. Повна автоматизація мінімізує ризик помилок, пов'язаних із людським фактором, що робить процес розгортання більш надійним.

Крім того, безперервне розгортання дозволяє впроваджувати зміни у виробниче середовище одразу після їх готовності, що забезпечує постійний потік оновлень і підвищує зручність для користувачів. Автоматичне тестування та моніторинг гарантують стабільність і високу якість програмного забезпечення, що сприяє підвищенню загальної довіри до продукту.

В якості прикладів інструментів для безперервного розгортання можна зазначити наступні:

- *Jenkins*: потужний інструмент з відкритим кодом для автоматизації процесів CI/CD. Забезпечує підтримку широкого спектра плагінів для налаштування безперервного розгортання;
- *GitLab CI/CD*: інтегрована платформа для управління репозиторіями, автоматичного тестування та розгортання. Підтримує автоматизацію усіх етапів DevOps;
- *CircleCI*: хмарна платформа для CI/CD, яка забезпечує легке налаштування та автоматизацію процесу розгортання;
- *Argo CD*: інструмент для безперервного розгортання у Kubernetes, який підтримує автоматичне впровадження змін у кластері на основі GitOps-підходу;

- *GitHub Actions*: вбудована в GitHub платформа для автоматизації CI/CD процесів, яка дозволяє створювати workflows безпосередньо в репозиторії та забезпечує тісну інтеграцію з екосистемою GitHub;
- *Travis CI*: популярний хмарний сервіс для безперервної інтеграції, який відомий своєю простотою налаштування та хорошою підтримкою проектів з відкритим кодом.

Типовий процес безперервного розгортання починається з інтеграції змін у центральний репозиторій, що автоматично запускає процес CI/CD. Далі код проходить серію автоматизованих тестів, включаючи юніт-тести, інтеграційні тести та енд-то-енд тести, для перевірки його якості та працездатності. Після успішного завершення тестування створюється артефакт, який автоматично розгортається у виробничому середовищі. Завершальним етапом є моніторинг, що дозволяє стежити за показниками продуктивності, виявляти можливі проблеми і, за потреби, запускати механізм відкату для відновлення стабільної версії.

Безперервне розгортання дозволяє компаніям досягти максимальної швидкості впровадження змін та підвищити конкурентоспроможність на ринку. Однак його впровадження вимагає високого рівня автоматизації, ретельно налаштованих тестів та надійної інфраструктури, що є важливою передумовою успіху.

1.2.4. Безперервне тестування

Безперервне тестування (*Continuous Testing, CT*) є важливою складовою DevOps-процесів, що забезпечує автоматизовану перевірку якості коду на всіх етапах розробки. Цей процес включає тестування на кожному етапі CI/CD-конвеєра, починаючи від першої інтеграції змін у репозиторій до фінального розгортання в виробничому середовищі. Безперервне тестування допомагає виявляти помилки на ранніх етапах і забезпечує високий рівень якості коду.

Основні принципи безперервного тестування:

- *інтеграція тестування в кожен етап CI/CD-конвеєра*: тестування автоматично інтегрується на всіх етапах процесу розробки: від безперервної інтеграції (CI) до безперервної доставки (CD). Після кожного коміту коду автоматично запускаються відповідні тести, що дозволяє виявляти помилки на ранніх етапах і не допустити їх на наступні етапи розробки.
- *виконання різних типів тестів*: безперервне тестування охоплює різні типи тестів:

- юніт-тести – перевіряють окремі компоненти або функціональні одиниці програми, щоб переконатися, що кожна частина працює коректно;
- інтеграційні тести – перевіряють взаємодію між різними модулями або системами, щоб переконатися, що всі частини програми працюють разом належним чином;
- регресійні тести – перевіряють, чи не виникли нові помилки в уже протестованому функціоналі після внесення змін або додавання нових функцій;
- навантажувальні тести – перевіряють здатність програми витримувати високі навантаження, щоб виявити потенційні проблеми при високих обсягах запитів або трафіку;
- тестування на проникнення (пен-тести) – моделюють атаки на систему, щоб оцінити її вразливість до загроз безпеки та визначити можливі шляхи несанкціонованого доступу до даних.
- *автоматичний аналіз результатів тестів*: після виконання тестів результати автоматично аналізуються для виявлення потенційних проблем. Система повідомляє команду розробників про виявлені помилки або невідповідності стандартам якості, що дозволяє оперативно вирішувати проблеми і підтримувати стабільність коду.

Безперервне тестування є важливим етапом у процесах DevOps, що дозволяє забезпечити високу якість коду на всіх етапах розробки. Для автоматизованого тестування використовуються різноманітні інструменти, які допомагають ефективно перевіряти функціональність програмного забезпечення і виявляти помилки на ранніх етапах. Вони забезпечують виконання різних типів тестів, від юніт-тестів до складніших інтеграційних тестів, тим самим підтримуючи якість коду та дозволяючи швидко реагувати на зміни.

Нижче наведено приклади інструментів для СТ:

- *Selenium* – популярний інструмент для автоматизованого тестування веб-додатків. Він дозволяє автоматизувати взаємодію з веб-сторінками, перевіряти коректність відображення елементів інтерфейсу, а також тестувати функціональність веб-додатків у різних браузерах. Selenium підтримує різні мови програмування, такі як Java, Python, C#;
- *JUnit* і *TestNG* є основними інструментами для модульного тестування в Java. Вони дозволяють створювати автоматизовані тести для перевірки коректності окремих функцій і компонентів програми. Обидва інструменти інтегруються з CI/CD системами, що дозволяє виконувати тести автоматично після кожного коміту;

- *Cucumber* використовується для тестування на основі поведінки (BDD – Behavior-Driven Development). Він дозволяє розробникам, тестувальникам та бізнес-аналізаторам описувати вимоги до системи у вигляді зрозумілих для всіх учасників командних історій. *Cucumber* підтримує автоматичне виконання тестів за допомогою таких мов, як *Gherkin*, що робить його ідеальним для перевірки бізнес-логіки та сценаріїв користувачів.

Безперервне тестування дозволяє забезпечувати відповідність вимогам користувачів і бізнесу за рахунок орієнтованих на бізнес-логіку тестів, які дають змогу підтвердити, що продукт відповідає вимогам кінцевих користувачів і задовольняє їхні потреби, а також покращувати ефективність процесів тестування і випуску продукту. Завдяки автоматизації тестування, команда може швидко виявляти дефекти, виправляти їх та забезпечувати стабільність продукту.

Переваги безперервного тестування включають зменшення ризику впровадження неякісного коду, оскільки тестування на ранніх етапах допомагає виявити помилки ще до того, як код потрапить у виробниче середовище. Це дозволяє уникнути непотрібних збоїв і забезпечити високу якість продукту протягом усього циклу його життя.

Швидке виявлення дефектів і їхнє усунення стає можливим завдяки автоматизації тестів, що дозволяє оперативно реагувати на зміни в коді. Це зменшує час на виправлення помилок і підвищує ефективність роботи команди.

Автоматизація тестування сприяє підвищенню швидкості випуску нових версій продукту. Це дозволяє швидше отримувати зворотний зв'язок і коригувати проблеми ще до того, як код потрапить у виробниче середовище, що в кінцевому рахунку знижує час на випуск нових функцій.

Забезпечення зворотної сумісності завдяки регресійним тестам дозволяє уникнути порушення роботи вже існуючих функцій при додаванні нових. Це гарантує стабільність програми після кожного оновлення і запобігає виникненню проблем у вже працюючих частинах системи.

Типовий процес безперервного тестування включає кілька етапів, що забезпечують автоматизовану перевірку якості коду на кожному етапі його розробки. Спочатку, коли розробник вносить зміни до коду, ці зміни автоматично інтегруються в репозиторій через процес безперервної інтеграції (CI). Це дозволяє системі CI автоматично запустити набір тестів, включаючи юніт-тести, інтеграційні тести, регресійні тести та інші, для перевірки якості внесених змін. Важливо, що ці тести виконуються без необхідності ручного втручання, що значно підвищує ефективність процесу.

Після виконання тестів результати автоматично аналізуються для виявлення дефектів або інших проблем. Якщо тест проходить успішно, код просувається на наступні етапи CI/CD-конвеєра. У разі невдачі тесту розробник отримує сповіщення про помилку, що дозволяє оперативно виправити проблему.

На більш пізніх етапах тестування також включає регресійні та навантажувальні тести. Це забезпечує стабільність і масштабованість програми, гарантує, що нові зміни не порушують вже існуючий функціонал і що система здатна витримувати високі навантаження.

Безперервне тестування є ключовим елементом DevOps, що дозволяє забезпечити високу якість коду та мінімізувати помилки в продуктивному середовищі. Воно допомагає виявляти дефекти на ранніх етапах розробки, підвищує ефективність процесу тестування та покращує загальну стабільність і надійність програмного забезпечення. Завдяки автоматизації тестування та інтеграції з іншими етапами CI/CD, СТ допомагає зберігати високий рівень якості і прискорює процес доставки нових версій продукту.

1.2.5. Безперервний моніторинг

Безперервний моніторинг (Continuous Monitoring, CM) є ще однією важливою складовою DevOps-процесів, що дозволяє відстежувати стан додатків, інфраструктури та всієї системи в реальному часі. Це забезпечує стабільну роботу програмного забезпечення та дозволяє швидко виявляти й усувати проблеми, що виникають в експлуатаційних середовищах.

Безперервний моніторинг не лише сприяє підвищенню стабільності системи, а й допомагає виявляти ефективність роботи додатків та інфраструктури, зменшуючи час простою і покращуючи досвід користувачів.

Основні принципи безперервного моніторингу:

- *моніторинг продуктивності, навантаження, журналів помилок і доступності системи:* безперервний моніторинг включає відстеження ключових показників продуктивності (наприклад, час відгуку, використання ресурсів, пропускну здатність) і навантаження на систему. Він також включає моніторинг журналів помилок та доступності, що дозволяє швидко виявляти неполадки та реагувати на них до того, як вони вплинуть на кінцевого користувача;
- *використання сповіщень для оперативного реагування на збої:* для оперативного реагування на збої та проблеми в системі використовуються сповіщення, які спрацьовують при досягненні певних порогових значень. Наприклад, якщо система досягає

максимального порогу навантаження або виникає помилка на рівні інфраструктури, моніторингова система автоматично сповіщає відповідальних осіб або автоматично ініціює певні коригувальні дії;

- *збір даних для покращення майбутніх релізів*: безперервний моніторинг забезпечує збір та аналіз даних, що дає можливість виявляти вузькі місця в системі, а також прогнозувати потенційні проблеми у майбутньому. Ці дані використовуються для вдосконалення процесу розробки та оптимізації продуктів у наступних релізах.

Інструменти для безперервного моніторингу (Continuous Monitoring, CM) дозволяють забезпечити постійний контроль над станом систем, додатків та інфраструктури, що є критичним для підтримки їх стабільної роботи та оперативного реагування на проблеми. Вони дають змогу виявляти аномалії, визначати потенційні ризики та виконувати необхідні коригувальні дії в реальному часі, забезпечуючи надійність і ефективність роботи системи.

Розглянемо деякі інструменти для CM:

- *Prometheus* є потужним інструментом для збору метрик та створення алертів, який забезпечує гнучкий моніторинг продуктивності додатків і інфраструктури. Prometheus збирає метрики у вигляді числових значень і забезпечує інтеграцію з іншими інструментами для моніторингу. Він використовує мову запитів PromQL для гнучкого аналізу даних і створення різноманітних алертів для оперативного реагування на критичні ситуації;
- *Grafana* використовується для візуалізації даних з різних джерел, включаючи Prometheus, Elasticsearch та інші. Це дозволяє створювати інтерактивні інформаційні панелі (дашборди), які наочно відображають ключові метрики, наприклад, навантаження на сервери, час відгуку додатків, кількість запитів до бази даних, тощо. Grafana дозволяє оперативно відслідковувати стан системи і отримувати зворотний зв'язок від усіх компонентів інфраструктури;
- *ELK Stack (Elasticsearch, Logstash, Kibana)* є потужним набором інструментів для аналізу, зберігання та візуалізації логів:
 - *Elasticsearch* забезпечує зберігання та індексацію великих обсягів логів і даних;
 - *Logstash* використовується для збору, обробки та трансформації логів, перед тим як відправити їх до Elasticsearch;
 - *Kibana* надає інтерфейс для візуалізації та аналізу даних з Elasticsearch, що дозволяє швидко знаходити і виявляти аномалії або критичні проблеми в логах.

Безперервний моніторинг є важливим компонентом забезпечення стабільної роботи систем і додатків, дозволяючи не лише виявляти і реагувати на збої, але й отримувати змогу постійно оптимізувати інфраструктуру. Оперативне виявлення проблем і їхнє усунення є критичними для підтримки високої доступності та ефективності роботи сервісів, що, в свою чергу, позитивно впливає на досвід користувачів і надійність системи в цілому.

Один із основних аспектів безперервного моніторингу – це здатність швидко реагувати на збої та неполадки в системі. Коли виникають проблеми, система автоматично сповіщає команду або ініціює автоматичні коригувальні дії, що допомагає підтримувати стабільність. Також моніторинг дозволяє виявляти вузькі місця в додатках або інфраструктурі, що дає змогу ефективніше використовувати ресурси та підвищувати продуктивність.

Оперативне виявлення проблем мінімізує час простою, що безпосередньо покращує користувацький досвід. Крім того, збір історичних даних дозволяє прогнозувати потенційні проблеми, вживаючи заходів до того, як вони стануть критичними. Завдяки моніторингу збираються дані, які не тільки допомагають у реальному часі, а й використовуються для постійного вдосконалення системи, оптимізації її роботи та поліпшення користувацького досвіду з кожним оновленням.

Типовий процес безперервного моніторингу включає постійне спостереження за продуктивністю системи, збором важливих показників, таких як час відгуку, навантаження на сервери та кількість запитів до баз даних. Ці показники аналізуються для виявлення потенційних аномалій, таких як зниження продуктивності чи перевищення допустимих навантажень.

Коли виявляються відхилення, система автоматично сповіщає відповідальних осіб або може ініціювати автоматичні коригувальні дії, щоб мінімізувати вплив проблеми. Збір і аналіз логів, що здійснюється за допомогою таких інструментів, як ELK Stack, допомагає виявити додаткові проблеми та забезпечити зворотний зв'язок щодо стану системи.

На основі зібраних даних команда проводить оптимізацію інфраструктури, щоб підвищити ефективність, стабільність і доступність системи, забезпечуючи її безперервну і надійну роботу.

Безперервний моніторинг є критично важливою складовою DevOps, оскільки дозволяє підтримувати високу стабільність і ефективність роботи системи, оперативно реагувати на збої і покращувати досвід кінцевих користувачів. Завдяки використанню потужних інструментів для моніторингу продуктивності, аналізу логів та автоматизації алертів, безперервний моніторинг дає можливість швидко виявляти та усувати проблеми в реальному

часі, мінімізуючи вплив на бізнес і забезпечуючи безперебійну роботу додатків та інфраструктури.

Таким чином, розглянуті вище п'ять процесів утворюють єдиний цикл розробки, доставки та підтримки програмного забезпечення. Безперервна інтеграція забезпечує постійне оновлення коду, який автоматично проходить через безперервне тестування. Потім цей код стає доступним для доставки, а його стабільність та продуктивність контролюється через безперервний моніторинг.

Впровадження цих базових процесів у DevOps дозволяє організаціям досягти високої швидкості та якості розробки програмного забезпечення. Вони забезпечують автоматизацію, прозорість і надійність усіх етапів життєвого циклу, що є ключовим для сучасних технологічних компаній.

DevOps є потужним підходом до управління життєвим циклом програмного забезпечення, який забезпечує значну цінність як для технічних команд, так і для бізнесу. Завдяки інтеграції автоматизації, безперервної інтеграції та вдосконаленню процесів, DevOps дозволяє компаніям швидко адаптуватися до змін, покращувати якість продуктів і підвищувати ефективність.

Цей підхід прискорює розробку програмного забезпечення, зменшуючи час від написання коду до його впровадження в продуктивне середовище. Завдяки автоматизованим тестам на кожному етапі процесу, DevOps допомагає знижувати кількість помилок, що значно підвищує стабільність і якість продуктів. Команди можуть швидко реагувати на нові вимоги бізнесу та забезпечувати випуск нових функцій без затримок.

DevOps також сприяє гнучкості, дозволяючи компаніям ефективно масштабувати свої системи, інтегрувати нові технології та адаптуватися до змін. Оптимізація процесів, що включає автоматизацію тестування, розгортання та моніторинг, допомагає зменшити витрати, підвищити ефективність і зменшити час на виконання рутинних завдань. Крім того, регулярні оновлення та можливість швидкого експериментування сприяють інноваціям і покращенню досвіду користувачів.

DevOps також підтримує командну співпрацю, сприяючи розвитку культури, в якій команди працюють разом для досягнення спільних цілей, забезпечуючи більшу прозорість та задоволеність працівників завдяки автоматизації.

Таким чином, DevOps не лише прискорює розробку та доставку програмного забезпечення, але й забезпечує його високу якість, гнучкість і ефективність, сприяючи сталому розвитку організації та задоволенню кінцевих користувачів.

1.3. Переваги DevOps

DevOps є одним із найефективніших підходів для управління життєвим циклом програмного забезпечення. Його переваги охоплюють не лише технічну сферу, але й бізнес-процеси, створюючи значну цінність для організацій. Зупинимось коротко на ключових перевагах.

Швидкість

Однією з основних переваг DevOps є прискорення циклу розробки програмного забезпечення. Зокрема, можна виділити наступні моменти:

- *швидка доставка*: завдяки впровадженню автоматизації та безперервної інтеграції (CI), DevOps дозволяє зменшити час між написанням коду та його впровадженням у виробниче середовище;
- *реакція на зміни*: команди можуть швидко реагувати на нові вимоги бізнесу, впроваджуючи нові функції або виправлення за лічені години чи дні;
- *прискорений запуск продукту*: DevOps дає змогу компаніям випереджати конкурентів, запускаючи продукти швидше. Це особливо важливо в умовах швидко змінюваного ринку.

Якість

DevOps сприяє підвищенню якості програмного забезпечення завдяки впровадженню автоматизованих підходів і стандартизації процесів та дає змогу отримати наступні переваги:

- *зниження кількості помилок*: автоматизоване тестування на кожному етапі розробки допомагає виявляти помилки на ранніх стадіях, що зменшує витрати на їх виправлення;
- *безперервне вдосконалення*: регулярне впровадження оновлень дозволяє швидше реагувати на зворотний зв'язок від користувачів і усувати недоліки;
- *стабільність*: автоматизація процесів розгортання мінімізує ризики людських помилок, забезпечуючи стабільну роботу систем.

Гнучкість

DevOps надає організаціям можливість легко адаптуватися до змін і масштабувати свої системи відповідно до потреб бізнесу. Зокрема, можна виділити наступні акценти:

- *масштабування*: використання інструментів, таких як Kubernetes і Docker, дозволяє ефективно масштабувати інфраструктуру без простоїв;

- *модульність*: DevOps підтримує архітектури на основі мікросервісів, які полегшують впровадження змін і додавання нових функцій;
- *адаптивність*: організації можуть швидко інтегрувати нові технології та методології, залишаючись конкурентоспроможними.

Ефективність

Оптимізація використання ресурсів є ключовою перевагою DevOps, що досягається завдяки автоматизації та раціональному підходу до управління процесами та дає змогу отримати наступні переваги:

- *зменшення витрат*: автоматизація процесів тестування, розгортання та моніторингу знижує потребу у великій кількості ручної праці;
- *ефективне використання часу*: інженери витрачають менше часу на рутинні завдання, зосереджуючись на вирішенні стратегічних і творчих завдань;
- *зменшення простоїв*: завдяки постійному моніторингу систем, проблеми виявляються та усуваються ще до того, як вони вплинуть на користувачів.

Поліпшення командної роботи

DevOps сприяє розвитку культури співпраці в організації, де команди працюють над досягненням спільних цілей та сприяє наступним моментам:

- *задоволеність працівників*: завдяки автоматизації зменшується рівень стресу, пов'язаного з повторюваними завданнями;
- *прозорість*: спільні інструменти та процеси створюють середовище, де всі зацікавлені сторони мають доступ до актуальної інформації.

Таким чином, DevOps не лише прискорює розробку та доставку програмного забезпечення, але й забезпечує його високу якість, гнучкість і ефективність, сприяючи сталому розвитку організації та задоволенню кінцевих користувачів.

Таблиця 1.2 містить основні переваги впровадження DevOps, які охоплюють різні аспекти розробки програмного забезпечення, зокрема швидкість, якість, гнучкість, ефективність, інновації та командну роботу. Вона надає детальнішу характеристику того, як DevOps допомагає організаціям покращити процеси розробки, зменшити витрати та забезпечити стабільність систем.

Таблиця 1.2. Порівняння основних переваг DevOps

Перевага	Опис	Вплив на бізнес
Швидкість	Прискорення циклу розробки через автоматизацію та безперервну інтеграцію	Швидший вихід змін в продуктах на ринок, перевага перед конкурентами
Якість	Автоматизоване тестування і безперервне вдосконалення дозволяють знижувати кількість помилок	Покращення стабільності та зниження витрат на виправлення помилок
Гнучкість	Можливість швидко адаптуватися до змін та масштабувати системи	Підвищення конкурентоспроможності та ефективності
Ефективність	Автоматизація та оптимізація використання ресурсів	Зменшення витрат, підвищення ефективності
Інновації	Швидке створення та тестування нових ідей	Покращення досвіду користувачів, розвиток інновацій
Поліпшення командної роботи	Спільні інструменти та процеси покращують співпрацю між командами	Зростання задоволеності працівників і ефективності роботи

1.4. Виклики у впровадженні DevOps

Хоча DevOps приносить значні переваги для організацій, його впровадження може бути пов'язане з численними викликами. Ці виклики можуть бути як організаційними, так і технічними, що потребує від компаній не лише ресурсів, а й стратегічного підходу до адаптації нової методології.

Культурні зміни

DevOps – це більше ніж технологія, це культурна трансформація, яка потребує зміни мислення всієї організації. З огляду на це, при впровадженні цієї методології можуть виникати наступні ситуації:

- *розрив між командами*: в багатьох організаціях команди розробки та операцій працюють окремо, що створює бар'єри для співпраці. Інтеграція цих команд вимагає зусиль для усунення конфліктів і забезпечення спільних цілей;
- *спротив змінам*: співробітники можуть скептично ставитися до змін через страх перед новими вимогами або втрату звичних методів роботи;
- *підтримка з боку керівництва*: для успішного впровадження DevOps потрібна активна підтримка з боку керівників, яка не завжди забезпечується через нерозуміння переваг методології;

- *формування нових звичок*: зміна підходу до управління проєктами, комунікації та відповідальності займає час і потребує впровадження навчання та тренінгів.

Технічні труднощі

Інтеграція нових технологій та інструментів може стати значним викликом, особливо у великих організаціях із розгалуженою ІТ-інфраструктурою і призвести до наступних викликів:

- *складність інтеграції*: багато організацій використовують застарілі системи (legacy systems), які важко інтегрувати з сучасними DevOps-інструментами. Це може уповільнювати автоматизацію процесів та створювати додаткові технічні ризики;
- *сумісність інструментів*: інструменти DevOps потребують узгодженого використання для досягнення максимального ефекту. Невідповідність або несумісність програмного забезпечення може спричинити затримки;
- *інфраструктурні обмеження*: відсутність хмарної інфраструктури або обмеженість ресурсів для її впровадження може сповільнити процес адаптації DevOps;
- *висока вартість*: початкові витрати на закупівлю інструментів, розробку автоматизованих процесів та навчання персоналу можуть бути значними.

Потреба в кваліфікованих кадрах

DevOps вимагає від команди широкого спектра технічних і організаційних навичок, що може стати викликом для багатьох компаній та призвести до таких викликів:

- *дефіцит фахівців*: на ринку праці часто бракує кваліфікованих DevOps-інженерів, які володіють необхідними знаннями в області автоматизації, контейнеризації, CI/CD, моніторингу та інших аспектів;
- *необхідність навчання*: співробітникам, які вже працюють в організації, може знадобитися час і ресурси для опанування нових інструментів і методів роботи;
- *проблеми з адаптацією*: не всі фахівці готові швидко адаптуватися до роботи в умовах безперервного вдосконалення (Continuous Improvement);
- *мультидисциплінарність*: DevOps-інженери повинні поєднувати знання розробки, операційної підтримки, автоматизації та безпеки, що створює високі вимоги до компетенцій.

Організаційні виклики

Впровадження методології DevOps може спричинити наступні проблеми, пов'язані з особливістю бачення ситуацій організаціями, що її впроваджують:

- *відсутність стратегії*: часто організації починають впроваджувати DevOps без чіткого розуміння, як саме ця методологія має інтегруватися в їхні бізнес-процеси. Це може призвести до хаотичного використання інструментів та відсутності бажаних результатів;
- *розподілена інфраструктура*: у великих організаціях із розподіленими командами може виникнути проблема стандартизації процесів і синхронізації роботи;
- *обмежений бюджет*: витрати на впровадження DevOps, включаючи закупівлю інструментів, розробку автоматизованих процесів та навчання персоналу, можуть перевищувати доступні ресурси.

Безпека та відповідність нормативам

DevOps вимагає інтеграції процесів безпеки (DevSecOps), що додає складності до його впровадження в наступних напрямках:

- *забезпечення безпеки*: часте оновлення та автоматизація процесів можуть створити нові ризики, якщо безпека не є інтегрованою частиною конвеєра CI/CD;
- *відповідність регуляторним вимогам*: у певних галузях, таких як фінанси чи охорона здоров'я, впровадження DevOps може бути ускладнене через строгі вимоги до відповідності стандартам.
- *відсутність метрик для вимірювання ефективності*: визначення чітких метрик для оцінки успішності впровадження DevOps може бути складним завданням, особливо на ранніх етапах трансформації.

Можна зробити висновок, що впровадження DevOps – це складний і багатогранний процес, який потребує не лише технологічних рішень, а й серйозних культурних змін. Організації, які готові інвестувати у навчання персоналу, інтеграцію нових інструментів і побудову командної співпраці, можуть подолати ці виклики та досягти значного успіху.

1.5. Інструменти DevOps

Для ефективного впровадження принципів DevOps використовується широкий набір інструментів, кожен із яких оптимізує певний етап життєвого циклу програмного забезпечення. Ці інструменти автоматизують рутинні процеси, підвищують прозорість і забезпечують стабільність систем.

На рис. 1.1 показано безперервний цикл DevOps, який використовується для управління життєвим циклом програмного забезпечення. Зокрема,

зображено основні етапи DevOps та інструменти, які згруповані по функціональних ролях на кожному етапі.

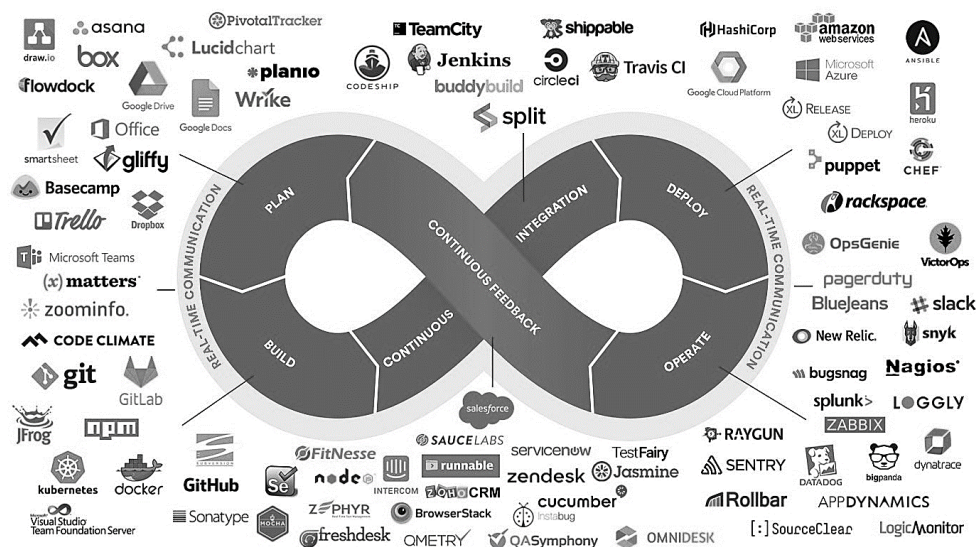


Рис. 1.1. Безперервний цикл DevOps

Розглянемо детальніше деякі з наведених інструментів, відповідно до їх призначення.

Управління конфігураціями

Управління конфігураціями – це ключовий аспект DevOps, який дозволяє забезпечити стандартизоване та автоматизоване налаштування інфраструктури. Можна виділити наступні інструменти для управління конфігураціями:

Ansible: легкий у використанні інструмент, що дозволяє автоматизувати розгортання, управління конфігураціями та оркестрацію. Його простий синтаксис YAML спрощує створення скриптів навіть для новачків;

Puppet: інструмент для автоматизації управління інфраструктурою, який дозволяє визначати конфігурації через декларативну мову. Він забезпечує контроль над великими системами завдяки централізованому управлінню;

Chef: потужний інструмент, який використовує мову програмування Ruby для опису конфігурацій, забезпечуючи гнучкість у налаштуванні складних систем.

Автоматизація CI/CD

Безперервна інтеграція (CI) і безперервна доставка (CD) є основними компонентами DevOps, які забезпечують швидке тестування та впровадження змін. Серед інструментів для автоматизації процесів CI/CD варто відзначити наступні:

Jenkins: популярна платформа з відкритим кодом, яка підтримує дуже велику кількість плагінів для автоматизації процесів CI/CD. Її гнучкість дозволяє налаштувати складні конвеєри доставки програмного забезпечення;

GitLab CI/CD: інтегрована платформа, яка забезпечує повний цикл розробки – від управління кодом до автоматизації тестування та доставки;

CircleCI: Хмарний сервіс для автоматизації CI/CD, який підтримує гнучкі сценарії розгортання та інтегрується з популярними системами контролю версій, такими як GitHub.

Контейнеризація

Контейнеризація стала невід’ємною частиною DevOps, оскільки вона дозволяє ізолювати додатки та їхні залежності, забезпечуючи портативність і легкість у масштабуванні.

Docker: інструмент для створення контейнерів, який дозволяє упакувати додаток із усіма необхідними залежностями в стандартизоване середовище, незалежне від платформи;

Kubernetes: система оркестрації контейнерів, яка дозволяє автоматизувати розгортання, масштабування та управління контейнеризованими додатками в кластерному середовищі. Kubernetes забезпечує балансування навантаження та самовідновлення систем.

Моніторинг

Моніторинг є критичним для забезпечення стабільності та продуктивності програмного забезпечення. DevOps використовує певні спеціалізовані інструменти, які дозволяють відстежувати стан системи в реальному часі та оперативно реагувати на збої:

Prometheus: інструмент для збору метрик і моніторингу системи, який підтримує створення алертів і інтеграцію з іншими інструментами, такими як Grafana;

Datadog: платформа для моніторингу і аналітики, яка збирає метрики, логи та відстеження з різних систем і додатків. Підтримує інтеграцію з багатьма іншими інструментами та надає потужні можливості для створення алертів і візуалізації даних;

Grafana: графічна платформа для візуалізації даних і створення інтерактивних дашбордів. Використовується для аналізу продуктивності системи;

ELK Stack (Elasticsearch, Logstash, Kibana): комплексний набір інструментів для збору, аналізу та візуалізації логів, що дозволяє ефективно діагностувати проблеми та відстежувати аномалії.

Керування версіями

Управління версіями коду є важливим компонентом DevOps для забезпечення прозорості, колективної роботи та історії змін. Найпопулярніша система контролю версій – система *Git*, яка забезпечує зручну співпрацю між командами через децентралізовану модель *GitHub/GitLab/Bitbucket* – найбільш популярні сервіси, які дозволяють зберігати репозиторії, керувати кодом, інтегрувати CI/CD та організовувати спільну роботу над проєктами.

Тестування

Автоматизоване тестування є ключовим для забезпечення якості коду в DevOps. В даному блоці можна виділити наступні інструменти:

Selenium: інструмент для автоматизованого тестування веб-додатків, що підтримує різні браузерери та мови програмування;

JUnit: бібліотека для модульного тестування Java-додатків, яка дозволяє створювати та виконувати тести;

TestNG: потужний інструмент для автоматизації тестування з підтримкою паралельного виконання тестів і інтеграції з іншими DevOps-інструментами.

Управління інфраструктурою як кодом (IaC)

Інфраструктура як код дозволяє створювати, конфігурувати та управляти інфраструктурою через програмні скрипти. Найбільш відомим інструментом для опису інфраструктури як коду, який підтримує різні платформи (AWS, Azure, Google Cloud), є *Terraform*. В якості інших засобів можна навести *CloudFormation* – інструмент від AWS, що дозволяє управляти ресурсами хмарної інфраструктури через декларативні шаблони.

Інструменти DevOps працюють у тісній взаємодії, забезпечуючи автоматизацію, інтеграцію та моніторинг на всіх етапах життєвого циклу програмного забезпечення. Вибір інструментів залежить від потреб організації, масштабу проєкту та технічної інфраструктури, але їхнє впровадження забезпечує підвищення продуктивності, якості та швидкості розробки.

1.6. Agile Methodology

Agile – це сучасний підхід до розробки програмного забезпечення, який спрямований на досягнення гнучкості, ефективної співпраці, швидкої адаптації до змін і постійного вдосконалення продукту. Він з'явився як реакція на недоліки традиційних методів розробки, зокрема «водоспадної» моделі (Waterfall), яка передбачала жорстку послідовність етапів проєкту – від визначення вимог до тестування й розгортання. Основні виклики водоспадного підходу, такі як повільна реакція на змінювані потреби, ризикованість фінальної

стадії проекту та відсутність тісної взаємодії між замовниками і командою, створили потребу у впровадженні більш динамічних і адаптивних рішень.

Agile орієнтований на часті ітерації, під час яких команда розробників створює й вдосконалює робочий продукт. Його основою став Маніфест гнучкої розробки програмного забезпечення (Agile Manifesto), який було опубліковано в 2001 році групою провідних інженерів і консультантів. У цьому документі визначено ключові цінності та принципи Agile, які формують його ідеологію.

Agile став відповіддю на зростаючу складність розробки програмного забезпечення в умовах динамічного розвитку технологій та бізнесу. Традиційні методи розробки часто виявлялися неефективними в ситуаціях, коли вимоги замовника змінювалися в процесі реалізації проекту. Крім того, такі підходи створювали значні ризики, такі як:

- тривалий час до випуску першої версії продукту;
- висока вартість виправлення помилок, виявлених на пізніх етапах;
- відсутність можливості отримати швидкий зворотний зв'язок від клієнтів або користувачів.

Agile вирішує ці проблеми, роблячи акцент на гнучкості процесів, взаємодії між командою та замовником і постійній доставці цінності для кінцевого користувача.

У Маніфесті Agile визначено чотири ключові цінності, які відрізняють цей підхід від традиційних моделей:

- *люди та взаємодія важливіші за процеси та інструменти*: головна увага приділяється командній роботі, комунікації та взаємодії, які сприяють досягненню поставлених цілей.
- *робоче програмне забезпечення важливіше за детальну документацію*: пріоритетом є створення продукту, який працює і відповідає реальним потребам клієнта, а не складання детальних описів або інструкцій.
- *співпраця з клієнтами важливіша за контрактні переговори*: Agile передбачає тісну взаємодію з клієнтами на всіх етапах проекту, що дозволяє забезпечити їхню залученість і задоволення результатом.
- *готовність до змін важливіша за дотримання початкового плану*: команди повинні бути здатними швидко адаптуватися до нових вимог або умов ринку, що гарантує актуальність продукту.

Окрім ключових цінностей, Agile базується на дванадцяти принципах, які конкретизують підхід до організації роботи, управління проектами, адаптації до змін і забезпечення якості продукту (рис. 1.2). Ці принципи є основою для впровадження Agile і допомагають командам ефективно працювати навіть у складних умовах. Нижче наведено розгорнутий опис кожного з принципів:

- *задоволення клієнта через швидку та безперервну доставку цінного програмного забезпечення* – сновна мета Agile – зробити клієнта задоволеним за рахунок швидкого надання продукту, який відповідає його вимогам. Команди забезпечують регулярне оновлення програмного забезпечення, щоб максимально швидко реалізувати необхідні функціональні можливості;
- *готовність до змін навіть на пізніх етапах розробки* – Agile підтримує гнучкість і здатність реагувати на зміну вимог у будь-який момент. Зміни сприймаються не як загроза, а як можливість покращити продукт, зробивши його актуальним і корисним;
- *часте постачання робочого продукту* – продукт доставляється невеликими, але робочими частинами через короткі проміжки часу (ітерації або спринти). Це дозволяє швидко отримувати зворотний зв'язок і вчасно впроваджувати коригування;
- *тісна співпраця між замовником і командою* – Agile передбачає постійну взаємодію із замовником для уточнення вимог, контролю прогресу та врахування змін. Це забезпечує високий рівень прозорості та довіри;
- *мотивація і підтримка команди* – успіх проєкту залежить від мотивованої, кваліфікованої та автономної команди. Agile сприяє створенню умов, у яких члени команди можуть самостійно приймати рішення, зберігаючи мотивацію та відчуття відповідальності;
- *фокус на живій комунікації* – основним засобом передачі інформації є особисте спілкування (очне або віртуальне). Це дозволяє швидко вирішувати проблеми, уникати непорозумінь і забезпечувати ефективну координацію між усіма учасниками;
- *робочий продукт як основний показник прогресу* – в Agile акцент робиться на наявність робочого програмного забезпечення як головного критерію успішності, а не на документації чи звітах;
- *стійкий темп розробки* – Agile підтримує стабільний, збалансований робочий ритм, який допомагає уникати вигорання команди та забезпечує довгострокову ефективність;
- *увага до технічної досконалості та якісного дизайну* – постійний фокус на технічному вдосконаленні та підтримці чистоти коду дозволяє створювати програмне забезпечення, яке легко підтримувати, масштабувати й розвивати.
- *простота* – Agile орієнтується на максимальну ефективність, уникаючи зайвих дій і витрат. Принцип «зробити тільки необхідне» дозволяє зосередитися на найважливіших аспектах проєкту;

- *самоорганізовані команди* – команди, які працюють за Agile, самостійно визначають, як найкраще досягти поставлених цілей. Це сприяє інноваціям, високій залученості та відповідальності;
- *постійний аналіз і вдосконалення* – регулярні ретроспективи та аналіз процесів дозволяють команді визначати слабкі місця, запроваджувати покращення й досягати вищих результатів.



Рис. 1.2. Основні принципи Agile: 1-6– принципи взаємодії з клієнтом та доставки продукту, 7-12 – принципи організації команди та процесів розробки

Ці 12 принципів створюють чіткі орієнтири для роботи команд, допомагають будувати довірчі відносини із замовниками та забезпечують адаптивність до мінливих умов. Вони є основою для популярних фреймворків, таких як Scrum, Kanban, SAFe та інших, і дозволяють командам досягати високих результатів навіть у складних і швидкозмінних проектах.

На рисунку 1.3 наведено порівняння циклів Agile і DevOps. Agile підкреслює гнучкість і циклічність, включаючи етапи планування, проектування, розробки, тестування, випуску та отримання зворотного зв'язку.

DevOps об'єднує розробку та операції через безперервний процес створення, планування, перевірки, упаковки, випуску, конфігурації та моніторингу, забезпечуючи інтеграцію та автоматизацію.

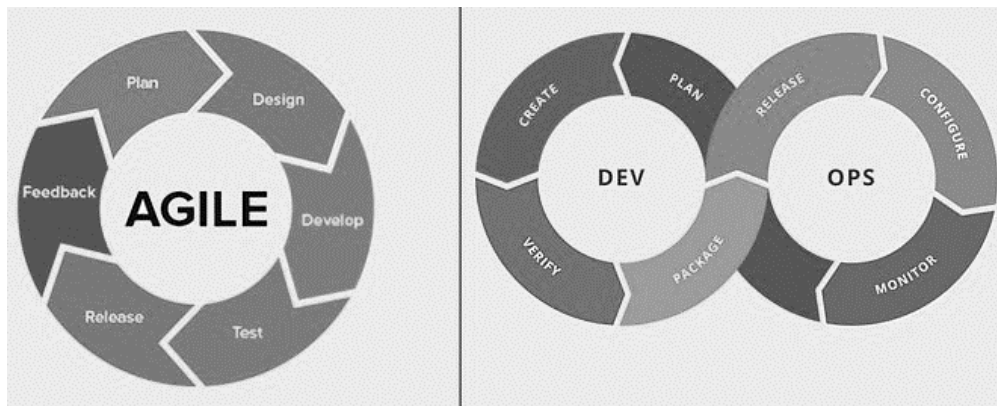


Рис. 1.3. Порівняння циклів Agile і DevOps

Обидва цикли ілюструють безперервний процес, що сприяє ефективній розробці та підтримці програмного забезпечення.

DevOps можна розглядати як логічне продовження та розширення методології Agile, яке відповідає на сучасні виклики, пов'язані з масштабуванням, автоматизацією та безперервним вдосконаленням систем. Обидві концепції мають на меті оптимізацію процесів розробки та доставки програмного забезпечення, забезпечуючи швидку реакцію на зміни й орієнтацію на потреби клієнтів.

Методологія Agile пропонує інструменти для поліпшення взаємодії між розробниками, замовниками й іншими зацікавленими сторонами та зосереджена на організації командної роботи, створенні гнучких ітеративних процесів розробки та швидкому наданні функціонального продукту. Вона орієнтована на отримання швидкого зворотного зв'язку, що дає змогу оперативно коригувати продукт у відповідь на зміни у вимогах чи умовах ринку.

DevOps додає до цих принципів новий вимір, акцентуючи увагу на інтеграції операційних процесів і створенні умов для безперервного вдосконалення системи на всіх етапах її життєвого циклу. DevOps об'єднує розробників (Dev) і фахівців з експлуатації (Ops), усуваючи бар'єри між цими командами для досягнення спільної мети – швидкого та стабільного постачання програмного забезпечення.

Якщо Agile в основному охоплює життєвий цикл розробки програмного забезпечення, то DevOps включає весь життєвий цикл продукту: від планування, розробки та тестування до розгортання й моніторингу в продуктивному

середовищі. Це забезпечує повну інтеграцію всіх процесів, що дозволяє не лише створювати якісний продукт, а й гарантувати його стабільність та ефективність у роботі.

DevOps також розширює можливості Agile через:

- *автоматизацію*: використання інструментів для автоматичного тестування, інтеграції, розгортання та моніторингу значно пришвидшує робочі процеси, зменшує кількість помилок і забезпечує стабільність системи.
- *моніторинг та аналітику*: постійний моніторинг продуктивності й аналітика зібраних даних дають змогу оперативно виявляти й усувати проблеми, а також прогнозувати можливі ризики.
- *безперервне вдосконалення*: DevOps підтримує ідею ітеративного вдосконалення, дозволяючи оновлювати та масштабувати систему з урахуванням реальних умов експлуатації.

Таким чином, DevOps не лише підтримує принципи Agile, а й розширює їхній вплив на всі аспекти розробки та експлуатації продукту. Це робить DevOps універсальним підходом для організацій, які прагнуть забезпечити високу якість, швидкість і надійність своїх програмних рішень. У поєднанні з Agile DevOps стає потужним інструментом для забезпечення інновацій, ефективності та конкурентоспроможності на сучасному ринку.

1.7. Взаємовідносини Agile та DevOps

Agile та DevOps мають схожі цілі, але підходять до їх досягнення з різних перспектив, створюючи синергію, яка значно підвищує ефективність процесів розробки та експлуатації програмного забезпечення.

Таблиця 1.3 дозволяє чітко побачити, як Agile і DevOps, маючи різні сфери фокусування, доповнюють одна одну, забезпечуючи ефективний підхід до розробки й підтримки програмного забезпечення.

Таблиця 1.3. Порівняння Agile та DevOps

Параметр	Agile	DevOps
Мета	Забезпечення гнучкої розробки та швидкого впровадження функціональних продуктів	Інтеграція процесів розробки та операцій для забезпечення якісного безперервного циклу доставки
Сфера застосування	Зосереджено на процесах розробки програмного забезпечення	Охоплює весь життєвий цикл продукту: розробку, тестування, розгортання та підтримку

Фокус	Гнучкість у процесах розробки, швидкий зворотний зв'язок із замовником	Автоматизація, моніторинг і безперервна інтеграція для стабільної роботи продукту
Командна робота	Зосереджено на співпраці в межах команди розробників	Об'єднує команди розробки, тестування та експлуатації в єдиний цикл
Основні принципи	Ітеративний підхід, короткі спринти, постійне вдосконалення	Автоматизація, безперервна інтеграція, розгортання та моніторинг
Ключові інструменти	Jira, Trello, Asana (для планування та управління проектами)	Jenkins, Docker, Kubernetes, Prometheus, Grafana (для автоматизації та моніторингу)
Швидкість впровадження змін	Забезпечується через ітерації та регулярний зворотний зв'язок із клієнтом	Автоматизовані процеси дозволяють швидше доставляти зміни у продуктивне середовище
Основна перевага	Швидка адаптація до змін і створення цінності для клієнта	Безперервність процесів і стабільність роботи системи
Область використання	Управління проектами та оптимізація процесів розробки	Технічна реалізація та автоматизація всіх процесів, пов'язаних із розробкою та підтримкою

Спільні принципи

Обидві методології спрямовані на підвищення продуктивності команд, скорочення часу на розробку та випуск продукту, а також забезпечення його якості через тісну співпрацю між усіма учасниками процесу. Agile та DevOps ставлять в основу своєї філософії швидку адаптацію до змін, орієнтацію на потреби клієнтів і постійне вдосконалення процесів. Це досягається за рахунок ітеративного підходу, прозорості в роботі команд і застосування сучасних технологічних рішень.

Сфери застосування

Agile концентрується на гнучкості команд і управлінні змінами на рівні проекту. Це включає розбиття великих завдань на невеликі, чітко визначені ітерації, що дозволяє командам отримувати регулярний зворотний зв'язок і швидко реагувати на нові вимоги. DevOps, у свою чергу, розширює цей підхід на технічні аспекти, забезпечуючи автоматизацію процесів тестування, інтеграції, розгортання та моніторингу. DevOps охоплює повний життєвий цикл продукту, включаючи його підтримку та масштабування в реальних умовах експлуатації.

Командна інтеграція

Agile робить акцент на тісній співпраці між членами команди розробки, бізнес-аналітиками та клієнтами, забезпечуючи спільне розуміння цілей і пріоритетів проєкту. DevOps йде далі, об'єднуючи розробників, тестувальників і операційних інженерів у єдиний робочий цикл. Це дозволяє уникнути розриву між командами, зменшити час на передачу завдань між ними й підвищити якість кінцевого продукту за рахунок постійного зворотного зв'язку на всіх етапах.

Взаємне доповнення

Agile забезпечує основу для створення функціонального продукту через короткі ітерації, швидке планування та пріоритезацію завдань. DevOps доповнює цю методологію, забезпечуючи безперервну інтеграцію, тестування, розгортання та моніторинг. Наприклад, результат роботи Agile-команди — це готовий до впровадження функціонал, а DevOps-команда автоматизує процес його розгортання, забезпечує стабільну роботу та швидке виявлення потенційних проблем.

Синергія в роботі

Разом ці методології утворюють комплексний підхід до розробки й підтримки програмного забезпечення. Вони дозволяють:

- швидко реагувати на зміни в умовах динамічного ринку;
- підвищувати якість продукту завдяки тісній співпраці між усіма учасниками процесу;
- мінімізувати ризики завдяки автоматизації та безперервному моніторингу;
- забезпечувати масштабованість і стабільність роботи програмного забезпечення.

Таким чином, Agile та DevOps, працюючи в тандемі, створюють міцну основу для організацій, які прагнуть розробляти інноваційні продукти, підтримувати їхню стабільну роботу та залишатися конкурентоспроможними в умовах сучасного ринку. Цей інтегрований підхід дозволяє оптимізувати весь життєвий цикл продукту – від концепції до експлуатації, забезпечуючи при цьому максимально можливу цінність для клієнтів і бізнесу.

Контрольні питання до розділу 1

- 1) Що означає термін DevOps?
- 2) Яка основна мета DevOps?
- 3) Як DevOps сприяє співпраці між розробниками та операційними інженерами?
- 4) Які основні процеси охоплює DevOps?
- 5) Що таке безперервна інтеграція (CI) в контексті DevOps?

- 6) Яке завдання безперервного розгортання (CD) в DevOps?
- 7) Як моніторинг сприяє успіху процесу DevOps?
- 8) Яким чином автоматизація тестування впливає на процес DevOps?
- 9) Чому DevOps вважається розширенням Agile?
- 10) Яка основна різниця між Agile і DevOps?
- 11) Що найкраще описує концепцію DevOps?
 - a) Тільки набір інструментів для автоматизації
 - b) Методологія тестування програмного забезпечення
 - c) Культура співпраці між командами розробки та операцій
 - d) Виключно фаза тестування у життєвому циклі розробки
- 12) Який процес у DevOps забезпечує автоматичне збирання та тестування коду після його інтеграції?
 - a) Continuous Delivery
 - b) Continuous Integration
 - c) Continuous Monitoring
 - d) Continuous Testing
- 13) Що є основною метою Continuous Delivery?
 - a) Автоматизація процесу релізу до продакшн середовища
 - b) Виявлення помилок у коді
 - c) Моніторинг продуктивності додатків
 - d) Створення резервних копій
- 14) Який з наведених процесів DevOps передбачає безперервну перевірку функціональності та продуктивності системи?
 - a) Continuous Integration
 - b) Continuous Testing
 - c) Continuous Delivery
 - d) Continuous Monitoring
- 15) Чим DevOps відрізняється від Agile?
 - a) DevOps орієнтований лише на операційну частину
 - b) Agile зосереджується на методах планування, DevOps – на автоматизації та інтеграції
 - c) Agile повністю замінює DevOps
 - d) DevOps базується виключно на Waterfall методології
- 16) Яка з наступних характеристик належить до DevOps, а не до Agile?
 - a) Гнучке управління командами
 - b) Інтеграція операційної діяльності з розробкою
 - c) Робота спринтами для виконання завдань
 - d) Використання Канбан-дошок

- 17) Що забезпечує Continuous Monitoring у DevOps?
- Тестування коду на наявність помилок
 - Постійне спостереження за продуктивністю системи
 - Автоматичне оновлення коду
 - Виконання інтеграційних тестів
- 18) Як DevOps підтримує розширення Agile методології?
- Забезпечує більшу структуру, ніж Agile
 - Виключає необхідність операційної діяльності
 - Доповнює Agile автоматизацією та безперервністю процесів
 - Замінює Agile своїми принципами
- 19) Що є одним із ключових результатів Continuous Integration?
- Менше тестових середовищ
 - Зменшення кількості випусків
 - Швидше виявлення помилок у коді
 - Автоматичне створення документації
- 20) Яка з наведених пар відображає правильний зв'язок між Agile та DevOps?
- Agile – це концепція, DevOps – це набір інструментів
 - Agile відповідає за планування, DevOps – за автоматизацію та розгортання
 - Agile зосереджується на релізах, DevOps – лише на тестуванні
 - DevOps є альтернативою Agile
- 21) Встановіть відповідність між процесами DevOps та їх описами:

A. Continuous Integration	1. Забезпечує автоматичне виявлення та виправлення помилок у реальному часі.
B. Continuous Delivery	2. Автоматичне тестування та інтеграція коду для швидкого виявлення помилок.
C. Continuous Testing	3. Автоматизація процесу релізу для готовності до продакшн середовища.
D. Continuous Monitoring	4. Постійна перевірка функціональності та продуктивності системи.

22) Встановіть відповідність між термінами та їх роллю в DevOps:

A. DevOps	1) Підхід, який інтегрує розробку та операційні процеси.
B. Agile	2) Методологія, що акцентує увагу на гнучкому управлінні проектами.
C. Continuous Integration	3) Безперервний процес інтеграції та тестування коду.
D. Continuous Monitoring	4) Спостереження за продуктивністю та стабільністю системи.

23) Встановіть відповідність між етапами DevOps та їх завданнями:

A. Create	1) Перевірка коду шляхом тестування.
B. Plan	2) Випуск змін до продуктивного середовища.
C. Verify	3) Розробка функціоналу додатка.
D. Release	4) Визначення завдань та їх пріоритетів.

24) Встановіть відповідність між аспектами Agile та DevOps:

A. Agile	1) Зосередженість на автоматизації процесів.
B. DevOps	2) Підхід до гнучкого управління проектами.
C. Обидва	3) Спрямованість на співпрацю між командами.
D. Жоден	4) Використання лише традиційних методів розробки.

25) Встановіть відповідність між перевагами DevOps та їх описами:

A. Скорочення часу розгортання	1) Забезпечення узгодженості між розробниками та операційною командою.
B. Підвищення якості програмного забезпечення	2) Автоматичне тестування, інтеграція та розгортання.
C. Покращення співпраці між командами	3) Менший час між створенням функціоналу та його релізом.
D. Автоматизація процесів	4) Постійне тестування та перевірка коду.

РОЗДІЛ 2. ОСНОВИ НЕПЕРЕРВНОЇ ІНТЕГРАЦІЇ

2.1. Поняття безперервної інтеграції

Неперервна інтеграція (Continuous Integration, CI) – це фундаментальний підхід у розробці програмного забезпечення, який дозволяє командам забезпечувати високу якість продукту, автоматизуючи процес інтеграції змін у код. Основна ідея CI полягає в регулярному об'єднанні змін у код, які виконуються кожним розробником, у спільний репозиторій. Це забезпечує можливість автоматичного тестування, виявлення помилок та конфліктів на ранніх етапах розробки, що сприяє зменшенню ризиків у процесі впровадження нових функціональностей.

Практика CI стала невід'ємною частиною сучасної розробки програмного забезпечення завдяки її здатності покращувати продуктивність роботи команд, скорочувати час випуску оновлень та забезпечувати стабільність продукту. Вона займає ключову позицію в екосистемі DevOps, яка об'єднує розробників та операційні команди для досягнення спільної мети – створення якісного програмного забезпечення.

Однією з основних переваг CI є її здатність забезпечувати автоматичне тестування змін після кожного коміту. Це дозволяє уникнути накопичення технічного боргу та зберігати високу якість коду протягом усього життєвого циклу проєкту. Крім того, інтеграція сучасних CI-систем із системами контролю версій, такими як Git, створює зручний і безперервний робочий процес для команд будь-якого масштабу.

В умовах постійного зростання вимог до якості та швидкості випуску програмного забезпечення ця методологія забезпечує конкурентоспроможність організацій, які прагнуть задовольнити потреби своїх користувачів. Завдяки автоматизації рутинних процесів та впровадженню системи регулярного тестування, CI допомагає скоротити час розробки, мінімізувати помилки та забезпечити безперервне вдосконалення продукту.

Неперервна інтеграція включає в себе процеси автоматизації збирання та тестування коду після кожного внесення змін до спільного репозиторію. Цей підхід дає змогу розробникам швидко інтегрувати зміни, уникати конфліктів у коді та забезпечувати стабільність системи. Основні етапи CI включають:

- *зміни у коді*: розробники регулярно вносять зміни до спільного репозиторію. Для цього зазвичай використовуються системи контролю версій, такі як Git, що забезпечують прозорість та зручність роботи;
- *автоматичне збирання*: як тільки зміни потрапляють до репозиторію, CI-сервер автоматично ініціює процес збирання проєкту. Цей процес

перевіряє, чи всі частини коду працюють разом коректно, виявляючи потенційні конфлікти або помилки на ранніх етапах;

- **тестування:** виконуються різні види автоматизованих тестів, зокрема:
 - юніт-тести для перевірки окремих модулів або функцій;
 - інтеграційні тести, які оцінюють взаємодію між компонентами;
 - функціональні тести для перевірки відповідності функцій системи очікуваним результатам;
 - перформанс-тести, що визначають продуктивність та масштабованість системи.

Додатково можуть виконуватись статичний аналіз коду та перевірка відповідності стандартам якості. Ці інструменти допомагають виявити можливі проблеми ще до етапу тестування.

Мета CI – швидке виявлення та виправлення помилок, що мінімізує ризики інтеграції та забезпечує стабільність системи. Завдяки регулярній інтеграції та автоматизованому тестуванню розробники отримують можливість працювати ефективніше, а користувачі – користуватися більш якісним продуктом. У поєднанні з безперервною доставкою та розгортанням CI забезпечує швидкість і стабільність розробки, що є критично важливим у сучасних умовах конкуренції.

2.2. Безперервна доставка та розгортання коду

CI тісно пов'язана з поняттями безперервної доставки (Continuous Delivery, CD) та безперервного розгортання (Continuous Deployment), які є наступними етапами автоматизації в життєвому циклі програмного забезпечення.

Безперервна доставка: це автоматизація доставки змін у кодї до середовищ тестування чи виробництва. Після проходження усіх тестів код стає готовим до розгортання у виробничому середовищі, проте цей процес виконується вручну за потреби. CD забезпечує високу гнучкість та дозволяє командам розгортати продукт у будь-який момент часу без ризику порушення стабільності системи.

Безперервне розгортання: це наступний рівень автоматизації, коли кожна успішна зміна у кодї, яка проходить усі тести, автоматично розгортається у виробничому середовищі. Цей підхід усуває потребу в ручному схваленні, що значно пришвидшує процес доставки нового функціоналу користувачам.

Можна виділити наступні переваги CD та Continuous Deployment:

- **зменшення часу виходу на ринок:** завдяки автоматизації усуваються затримки між завершенням роботи над функцією та її доступністю для кінцевих користувачів;

- *зниження людського фактора у розгортанні*: автоматизація мінімізує ризики помилок, що виникають через ручні дії під час розгортання;
- *постійна доступність нових функцій*: користувачі отримують доступ до оновлень та покращень системи у найкоротші терміни;
- *підвищення стабільності*: завдяки автоматизованому тестуванню та поступовому впровадженню змін знижується ризик порушення роботи системи;
- *економія ресурсів*: менше часу витрачається на розгортання, що дозволяє командам зосередитись на розробці нових функцій.

Для ефективної реалізації CD та Continuous Deployment команди часто використовують спеціалізовані інструменти, такі, наприклад, як Jenkins, GitLab CI/CD, GitHub Actions або AWS CodePipeline. Вони забезпечують інтеграцію із системами контролю версій, автоматизацію тестування, моніторинг та безпечне розгортання оновлень у виробничому середовищі.

Успішне впровадження цих підходів вимагає не лише технічних інструментів, але й культурних змін у командах, зокрема прийняття принципів DevOps, постійного вдосконалення процесів та активної співпраці між усіма учасниками розробки та експлуатації програмного забезпечення.

2.3. Порівняння кращих сучасних сервісів неперервної інтеграції

На даний момент можна виділити досить багато інструментів для реалізації неперервної інтеграції, кожен із яких має свої особливості, переваги та недоліки.

В таблиці 2.1 наведено основні переваги та недоліки деяких відомих сервісів неперервної інтеграції.

Таблиця 2.1. Переваги та недоліки деяких сервісів неперервної інтеграції

Сервіс	Переваги	Недоліки
Jenkins	Відкрите ПЗ, велика кількість плагінів	Високі вимоги до налаштувань
GitHub Actions	Інтеграція з GitHub, зручність	Може бути складним для великих проєктів
GitLab CI/CD	Інтеграція з GitLab, масштабованість	Обмеження безкоштовного тарифу
CircleCI	Простота використання, швидкість	Вартість для великих команд
Travis CI	Простота налаштування, кросплатформеність	Обмеження безкоштовного тарифу

Jenkins – це відкрита система автоматизації, яка використовується для організації процесів розробки програмного забезпечення, таких як безперервна інтеграція (CI) і безперервне постачання (CD). Вона дозволяє командам автоматизувати виконання завдань, що забезпечує прискорення процесів розробки, тестування та розгортання додатків. Jenkins є потужним інструментом для автоматизації процесів розробки програмного забезпечення. Його гнучкість, підтримка плагінів і масштабованість роблять його одним з найпопулярніших інструментів у сфері CI/CD. Проте для максимально ефективного використання можуть знадобитися додаткові налаштування та інтеграції з іншими DevOps-інструментами. Більш детально Jenkins розглядається в Розділі 6.

GitHub Actions – це вбудований інструмент для автоматизації робочих процесів (CI/CD) безпосередньо в GitHub. Він дозволяє створювати, налаштовувати та запускати завдання автоматизації в репозиторіях GitHub. Він ідеально підходить для інтеграції з іншими сервісами GitHub і забезпечує простоту налаштування та масштабованість для сучасних робочих процесів. Більш детально GitHub Actions розглядається в Розділі 7.

GitLab CI/CD – це потужна система для безперервної інтеграції і безперервного розгортання, яка є частиною платформи GitLab. Вона дозволяє автоматизувати процеси розробки, тестування та доставки програмного забезпечення. GitLab CI/CD – це відмінний вибір для команд, які вже використовують GitLab для управління кодом. Він забезпечує просту інтеграцію, масштабованість і ефективність для процесів CI/CD. Більш детально GitLab CI/CD розглядається в Розділі 4.

CircleCI

CircleCI – це популярна платформа для безперервної інтеграції та доставки, яка допомагає командам автоматизувати процеси розробки, тестування та розгортання програмного забезпечення.

На рисунку 2.1 показано логотип CircleCI.



Рис. 2.1. Логотип CircleCI

Основні можливості CircleCI

- автоматизація збірки та тестування:

- CircleCI автоматично запускає процеси компіляції, тестування та перевірки коду, що мінімізує людський фактор та знижує ризик помилок;
- підтримує різні мови програмування та фреймворки, включаючи Python, Java, JavaScript, Ruby, Go та інші;
- забезпечує використання кешування залежностей для скорочення часу на виконання завдань, зменшуючи тривалість збірок;
- *інтеграція з репозиторіями коду:*
 - CircleCI дозволяє легко підключати репозиторії, що зберігаються на популярних платформах, таких як GitHub, GitLab та Bitbucket;
 - можливість автоматичного запуску завдань при кожному пуші або pull request;
 - підтримка фільтрації гілок, що дозволяє запускати завдання тільки для вибраних гілок або тегів;
- *гнучкі конфігурації:*
 - конфігурація через файл .circleci/config.yml дозволяє детально налаштовувати етапи CI/CD;
 - підтримка повторного використання коду через команди та екземпляри, що спрощує управління складними пайплайнами;
 - можливість створення окремих завдань для специфічних середовищ або сценаріїв, таких як тестування, статичний аналіз коду, безпековий аудит тощо;
- *хмарні та локальні рішення:*
 - хмарна версія дозволяє швидкий старт без потреби у власних серверах чи складному налаштуванні;
 - локальні рішення (CircleCI Server) підходять для компаній з високими вимогами до безпеки, де дані повинні залишатися у власній інфраструктурі;
 - легке масштабування відповідно до потреб бізнесу;
- *паралельне виконання процесів:*
 - CircleCI підтримує розподіл тестів між кількома машинами, що суттєво прискорює час виконання великих тестових наборів;
 - можливість розділення завдань між декількома контейнерами або віртуальними машинами;
 - розширені механізми управління залежностями для забезпечення послідовності між завданнями;
- *моніторинг та логування:*
 - пропонує зручний інтерфейс для перегляду стану всіх виконаних завдань;

- журнали надають детальну інформацію про кожен етап виконання, включаючи використані команди, середовища та помилки;
- інтеграція з аналітичними інструментами для моніторингу продуктивності пайплайнів;
- *інтеграція з іншими інструментами:*
 - підтримка Docker забезпечує можливість створення ізольованих середовищ для тестування та збірки;
 - інтеграція з платформами хмарних обчислень (AWS, Google Cloud Platform, Microsoft Azure) дозволяє автоматизувати розгортання;
 - інтеграція зі Slack, Microsoft Teams та іншими месенджерами для оперативного сповіщення команди про стан завдань;
 - можливість роботи з базами даних, API або іншими сервісами для побудови складних CI/CD процесів.

Серед додаткових, але вагомих можливостей варто відмітити:

- *організацію робочих процесів (Workflows):* дозволяє створювати багатоступеневі пайплайни з умовною логікою, що залежить від результатів попередніх завдань;
- *підтримку Open Source:* CircleCI безкоштовний для невеликих команд і проєктів з відкритим кодом;
- *безпеку:* використання ключів SSH, токенів і змінних середовища для захисту конфіденційної інформації.

На рисунку 2.2 наведено інтерфейс системи керування CircleCI, де представлені доступні пайплайни та їх статуси.

Зокрема, на лівій панелі навігації представлені натупні пункти:

- *Dashboard* – головна панель для перегляду загальної інформації про пайплайни.
- *Projects* – секція для перегляду проєктів.
- *Insights* – розділ для аналітики щодо виконання пайплайнів.
- *Self-Hosted Runners* – опція для налаштування власних серверів для виконання завдань.
- *Organization Settings* – налаштування організації.
- *Plan* – інформація про план обслуговування або підписку.
- *"Can't find an organization?"* – посилання на допомогу, якщо користувач не може знайти свою організацію.
- *Notifications, Docs, Orbs, Support* – додаткові розділи для сповіщень, документації, інтеграцій (Orbs) і підтримки.

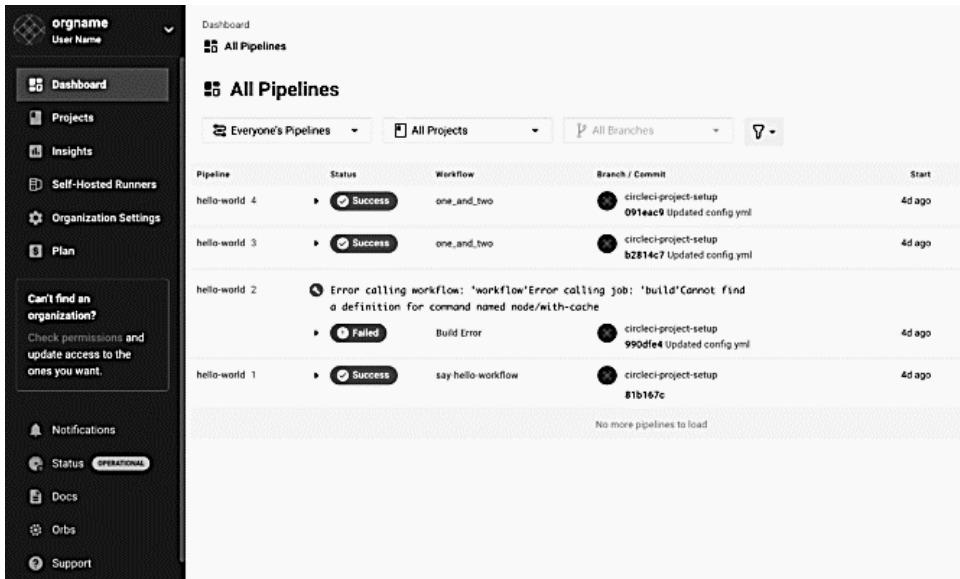


Рис. 2.2. Інтерфейс системи керування CircleCI

CircleCI – це потужний інструмент для автоматизації DevOps-процесів, який дозволяє командам ефективно інтегрувати зміни в код, тестувати їх та розгортати в різних середовищах. Його головною перевагою є здатність прискорювати процеси розробки завдяки паралельному виконанню завдань, кешуванню залежностей та інтеграції з популярними інструментами й хмарними платформами.

Розглянемо, як саме працює CircleCI – від моменту внесення змін у код до їхнього успішного розгортання в продакшн-середовищі:

- розробник відправляє код до репозиторію (наприклад, *GitHub*):
 - код може бути доданий через push до основної гілки або через pull request;
 - CircleCI інтегрується з репозиторієм завдяки вебхукам або API;
 - будь-які зміни в коді автоматично тригерять новий процес CI/CD, без необхідності додаткових дій від розробника;
- CircleCI автоматично підхоплює зміни та починає виконувати визначені завдання з конфігураційного файлу:
 - система зчитує файл конфігурації `.circleci/config.yml` для визначення інструкцій щодо виконання збірки, тестів і розгортання;
 - на основі конфігурації запускаються ізольовані середовища (контейнери Docker або віртуальні машини);

- підтримуються різні типи середовищ, зокрема Linux, Windows, macOS або спеціалізовані образи Docker;
- *проводиться збірка коду, запуск тестів і перевірка якості:*
 - виконується завантаження залежностей, компіляція та запуск статичного аналізу коду;
 - тестування включає: юніт-тести (Unit Tests), інтеграційні тести (Integration Tests), енд-то-енд тести (End-to-End Tests);
 - підтримується паралельне тестування (Parallelism), що значно скорочує час виконання;
 - аналізуються вразливості та порушення стандартів коду;
 - виконується перевірка якості збірки, яка може включати перевірку продуктивності або сканування безпеки (Security Scanning);
- *у разі успішного проходження всіх етапів код автоматично розгортається на сервери чи у хмару:*
 - CircleCI може інтегруватися з хмарними провайдерами, такими як AWS, Google Cloud Platform або Azure, для автоматизованого розгортання;
 - підтримується деплоймент на Kubernetes-кластери, Docker-образи та інші середовища;
 - можливий деплой на сервери через SSH або з використанням інструментів управління конфігураціями (Ansible, Terraform);
 - якщо тестування не пройдено, розгортання блокується до усунення проблем;
- *усі результати виконання надсилаються розробникам через email або месенджери (Slack):*
 - сповіщення можуть включати статус збірки (успішний або помилковий), деталі помилок та рекомендації щодо їх виправлення;
 - підтримується інтеграція з інструментами управління завданнями (Jira, Trello) для створення тикетів у разі виникнення проблем;
 - журнали виконання доступні для аналізу через вебінтерфейс CircleCI або API.

Варто відміти такі додаткові функції в процесі роботи:

- *кешування та повторне використання артефактів:* кешуються залежності та результати тестів для скорочення часу на повторні збірки; є можливість збереження артефактів, таких як журнали або результати тестів, для подальшого аналізу;
- *умовна логіка (Conditional Logic):* є можливість налаштувати різні сценарії виконання для окремих гілок або середовищ (development,

- staging, production); підтримка ручного підтвердження перед виконанням критичних етапів (approval jobs);
- *розгалуження процесів (Workflows)*: розробники можуть налаштовувати паралельне виконання кількох завдань, таких як тестування для різних середовищ або версій бібліотек; є підтримка відкладених завдань для кращої оптимізації ресурсів;
- *відкат (Rollback)*: CircleCI дозволяє легко відкотитися до попередньої версії в разі виявлення критичних помилок після розгортання;
- *аналітика та продуктивність*: аналітичні звіти допомагають оптимізувати процеси розробки, виявляючи вузькі місця та можливості для покращення продуктивності.

Таким чином, CircleCI забезпечує повний цикл безперервної інтеграції та доставки, автоматизуючи більшість завдань, пов'язаних із розробкою, тестуванням і розгортанням програмного забезпечення.

До переваг використання CircleCI можна віднести наступні:

- *швидкість*: завдяки кешуванню залежностей та паралельному виконанню;
- *гнучкість*: підтримка різних мов програмування та середовищ;
- *масштабованість*: легке налаштування як для малих стартапів, так і для великих корпорацій;
- *безпека*: можливість конфігурувати VPN та ключі шифрування для безпечної роботи.

Приклад конфігурації в CircleCI (.circleci/config.yml):

```
version: 2.1

jobs:
  build:
    docker:
      - image: circleci/node:14
    steps:
      - checkout
      - run: npm install
      - run: npm test

workflows:
  version: 2
  build_and_test:
    jobs:
      - build
```

Розглянемо детальніше цей сценарій.

version: 2.1 – вказує, що використовується версія 2.1 конфігураційного формату CircleCI. Ця версія надає доступ до більш гнучких функцій і можливостей, таких як повторне використання завдань (jobs).

Опис завдання (job):

```
jobs:
  build:
    docker:
      - image: circleci/node:14
```

jobs – описує завдання, які виконуються на CI/CD.

build – назва завдання.

docker – вказує, що завдання виконується в контейнері Docker.

image: circleci/node:14 – Docker-образ із встановленою Node.js версії 14.

Кроки виконання (steps)

```
steps:
  - checkout
  - run: npm install
  - run: npm test
```

steps – описує послідовність дій для виконання завдання:

checkout – клонує репозиторій проєкту з Git для подальшої роботи;

run: npm install – виконує команду npm install, щоб встановити всі залежності проєкту;

run: npm test – запускає тести проєкту за допомогою npm test.

Опис робочого процесу (workflow)

```
workflows:
  version: 2
  build_and_test:
    jobs:
      - build
```

workflows – визначає робочий процес, який складається із одного або кількох завдань.

version: 2 – вказує версію формату для робочих процесів.

build_and_test – назва робочого процесу.

jobs – перелік завдань, що входять до робочого процесу.

build – вказує на завдання, описане раніше в секції jobs.

Загальний процес:

1. *Контейнер Docker*: Використовується образ із Node.js 14 для виконання;
2. *Основні етапи*:
 - клонування репозиторію;
 - встановлення залежностей;
 - запуск тестів;

Робочий процес: автоматично виконує завдання build після запуску CI/CD.

Наведений сценарій підходить для Node.js-проектів і дозволяє автоматизувати перевірку якості коду під час кожного внесення змін до репозиторію.

CircleCI є потужним інструментом для автоматизації DevOps-процесів, що дозволяє значно прискорити розробку, підвищити якість коду та забезпечити безперервне розгортання оновлень.

Travis CI

Travis CI – це хмарна система безперервної інтеграції, яка допомагає автоматизувати процеси розробки програмного забезпечення, включаючи тестування, збірку, розгортання та моніторинг змін у коді. Ця платформа дозволяє розробникам скоротити час на виконання рутинних завдань, забезпечуючи швидке виявлення помилок і підвищення якості програмного забезпечення.

Рисунок 2.3 містить логотип Travis CI.



Рис. 2.3 Логотип Travis CI.

Travis CI особливо популярна серед команд, які використовують GitHub, завдяки її легкій інтеграції та автоматичній роботі «з коробки». Після додавання файлу конфігурації `.travis.yml` до репозиторія, система починає автоматично виконувати тести та збірки при кожному коміті або пул-реквесті. Це дозволяє командам зосередитися на створенні нових функцій і виправленні багів, а не на налаштуванні інфраструктури для CI/CD.

Окрім інтеграції з GitHub, Travis CI підтримує й інші платформи контролю версій, такі як Bitbucket і GitLab, що робить її універсальним інструментом для багатьох команд. Завдяки широкому спектру підтримуваних мов програмування (наприклад, Python, Ruby, Java, JavaScript, Go, Rust та інших), Travis CI підходить як для невеликих стартапів, так і для великих підприємств із багатокомпонентними системами.

Ключовою перевагою Travis CI є хмарна архітектура, яка дозволяє уникнути складного налаштування локальної інфраструктури. Усі збірки виконуються на віртуальних машинах або контейнерах, що забезпечує масштабованість і стабільність роботи. Сервіс також підтримує автоматичне розгортання на популярних хмарних платформах, таких як AWS, Google Cloud Platform, Heroku або Docker Hub, що спрощує випуск нових версій програмного забезпечення.

Travis CI надає різні моделі використання: безкоштовні плани для проєктів з відкритим кодом та платні підписки для приватних репозиторіїв і корпоративних клієнтів. Для великих організацій доступна версія Travis CI Enterprise, яка дозволяє запускати систему у приватних хмарах або локальних середовищах, забезпечуючи додатковий рівень безпеки та контролю над даними.

На рисунку 2.4 відображено інтерфейс Travis CI.

Таким чином, Travis CI – це універсальний і гнучкий інструмент, який значно спрощує процеси DevOps, підвищує продуктивність команд і гарантує стабільність програмних продуктів.

Основні характеристики Travis CI:

- *автоматизація:*
 - Travis CI автоматично запускає тести після кожного коміту або пул-реквесту у ваш репозиторій;
 - підтримує паралельний запуск тестів, що дозволяє пришвидшити перевірку коду;
- *підтримка різних мов програмування:*
 - Travis CI підтримує численні мови, включаючи Python, Java, Ruby, JavaScript, PHP, Go, Node.js, Rust та інші;

- для кожної мови можна налаштувати відповідне середовище виконання;
- *інтеграція з GitHub*: Travis CI легко інтегрується з GitHub. Після налаштування у веб-інтерфейсі або за допомогою файлу `.travis.yml`, система починає слідкувати за змінами у вашому репозиторії;
- *хмарна та локальна версії*: Travis CI надає хмарну версію (SaaS), а також підтримує локальне використання для приватних проєктів через Travis CI Enterprise;

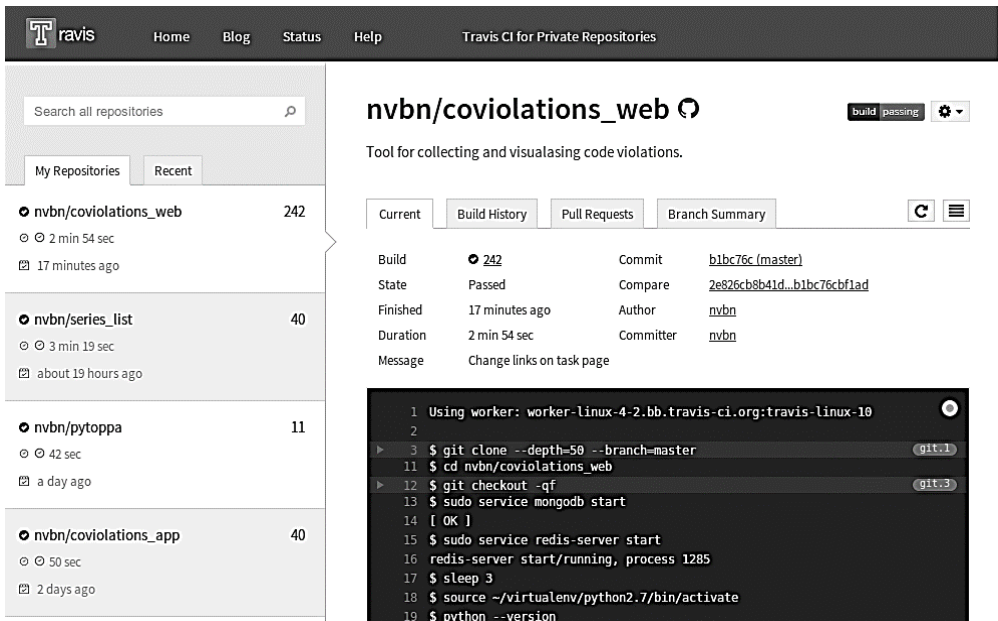


Рис. 2.4. Інтерфейс Travis CI

- *підтримка хмарних платформ*: Travis CI може розгортати додатки на популярних хмарних платформах, таких як AWS, Heroku, Google Cloud, Docker Hub тощо;
- *масштабованість та гнучкість*:
 - сервіс підтримує запуск тестів у різних середовищах і операційних системах (Linux, macOS, Windows);
 - легко адаптується до потреб як невеликих команд, так і великих організацій;
- *простота використання*: інтуїтивний інтерфейс і зручна документація роблять Travis CI легким у впровадженні, навіть для новачків.

Усі налаштування зберігаються у текстовому файлі `.travis.yml`, який знаходиться у корені репозиторія. Зазначений файл включає інформацію про

мову програмування, середовище, команди для встановлення залежностей, запуску тестів тощо;

Типовий робочий процес Travis CI має такі етапи:

- *Ініціалізація змін у репозиторії:*
 - розробник вносить зміни до вихідного коду та завантажує їх у репозиторій (наприклад, у гілку main, через pull request або у функціональну гілку);
 - зміни можуть включати оновлення коду, додавання нових тестів, зміну залежностей чи оновлення конфігураційних файлів;
- *сповіщення Travis CI:*
 - завдяки вебхукам GitHub, Travis CI автоматично отримує сповіщення про оновлення у репозиторії;
 - Travis CI виконує клонування відповідної гілки репозиторію та переходить до виконання завдань;
- *читання конфігурації з файлу .travis.yml:* Travis CI завантажує файл .travis.yml, у якому зазначено:
 - середовище виконання (наприклад, версії Python, Java або Node.js);
 - залежності, які потрібно встановити перед тестуванням;
 - скрипти, що необхідно виконати (наприклад, запуск тестів, лінтерів чи збірка проєкту);
 - умови розгортання додатку або генерації артефактів;
- *підготовка середовища:*
 - Travis CI створює ізольоване середовище (наприклад, у віртуальній машині чи контейнері) для виконання збірки;
 - встановлюються зазначені залежності, бібліотеки та інструменти;
- *запуск збірки та тестів:*
 - система запускає процеси збірки, тестування або перевірки якості коду;
 - усі етапи виконуються у визначеному порядку, і результати записуються у лог;
- *обробка результатів:*
 - у разі успішного виконання збірки Travis CI може виконати автоматичне:
 - розгортання додатку (наприклад, на сервер або у хмару);
 - створення артефактів збірки (наприклад, бінарних файлів або пакетів);
 - якщо виникають помилки:
 - виконання процесу припиняється;

- у логах детально вказується причина збою (наприклад, помилка тесту, невірна конфігурація або проблема із залежностями);
- *сповіщення розробників:*
 - у разі помилок чи успішного завершення завдань система надсилає повідомлення через email, Slack або інші інтегровані канали;
 - сповіщення включає деталі збірки, результати тестів і логи;
- *перегляд результатів:*
 - розробники можуть переглядати результати виконання завдань у вебінтерфейсі Travis CI;
 - за потреби вони виправляють помилки та повторюють процес;
- *циклічний характер процесу:* після внесення нових змін у репозиторій процес автоматично повторюється, забезпечуючи постійний контроль за якістю коду.

Такий підхід є основою CI/CD (Continuous Integration/Continuous Deployment), який сприяє швидкому виявленню помилок, забезпечує стабільність системи та пришвидшує розробку.

До основних переваг Travis CI варто віднести наступні:

- *простота у використанні:* Travis CI розроблена для швидкого і простого впровадження. Користувачам не потрібно мати глибокі технічні знання для налаштування, оскільки більшість операцій автоматизовані. Конфігурація виконується через файл `.travis.yml`, який легко читається та модифікується навіть для складних проєктів;
- *велика кількість інтеграцій:* Travis CI підтримує інтеграцію з численними інструментами для розробки та DevOps, такими як GitHub, GitLab, Slack, AWS, Heroku, Docker, і багатьма іншими. Це дозволяє автоматизувати всі аспекти процесу розробки, від тестування коду до його розгортання;
- *безкоштовна версія для проєктів з відкритим кодом:* для розробників, які працюють над проєктами з відкритим кодом, Travis CI пропонує безкоштовний план із повним набором функцій. Це робить її популярною серед ентузіастів, невеликих команд та open-source спільноти, оскільки дозволяє отримати всі переваги CI/CD без додаткових витрат;
- *підтримка розподіленого тестування:* Travis CI дозволяє запускати тести паралельно в кількох середовищах і з різними конфігураціями. Це скорочує час на перевірку коду і дозволяє виявляти помилки, специфічні для певних платформ або версій залежностей. Наприклад, можна одночасно тестувати додаток на різних версіях Python або Node.js;

- *мультиплатформеність*: Travis CI підтримує роботу на різних операційних системах, таких як Linux, macOS і Windows. Це дозволяє командам перевіряти сумісність програмного забезпечення з кількома платформами;
- *автоматичне розгортання*: платформа спрощує процес автоматичного розгортання додатків на серверах або хмарних платформах після успішного проходження тестів. Це зменшує ймовірність людських помилок і пришвидшує доставку нових функцій користувачам;
- *підтримка контейнеризації*: Travis CI підтримує Docker, що дозволяє створювати і тестувати додатки в ізольованих контейнерах. Це забезпечує більш точні результати тестування та спрощує управління середовищами;
- *масштабованість*: сервіс здатен обробляти проекти будь-якого розміру, від невеликих стартапів до великих організацій із тисячами репозиторіїв. Завдяки хмарній архітектурі, кількість одночасно виконуваних завдань можна легко масштабувати відповідно до потреб команди;
- *зручна аналітика та сповіщення*: Travis CI надає детальну інформацію про результати тестування, статус збірок і можливі помилки. Сповіщення можуть надсилатися через email, Slack або інші інтегровані сервіси, забезпечуючи оперативний зворотний зв'язок для розробників;
- *активна спільнота та підтримка*: Travis CI має активну спільноту користувачів і розробників, що допомагає швидко знаходити рішення до проблем. Крім того, офіційна документація детальна і покриває всі аспекти використання платформи.

До недоліків цього сервісу можна віднести такі аспекти, як:

- для приватних репозиторіїв Travis CI часто вимагає підписки;
- швидкість виконання може бути нижчою порівняно з іншими CI/CD інструментами, особливо на безкоштовних планах;
- обмеження у кількості паралельних завдань на безкоштовному тарифі.

Travis CI є гарним вибором для невеликих і середніх команд, які потребують простого способу автоматизувати процеси розробки та тестування коду.

Нижче наведено приклад конфігурації в Travis CI:

```

language: node_js # Вказуємо, що проект використовує Node.js
node_js:
  - "14" # Вибираємо версію Node.js

# Вказуємо гілки, для яких буде виконуватися CI
branches:
  only:
    - main
    - develop

# Стадії (скрипти), які виконуються перед основним процесом
before_script:
  - npm install # Встановлення залежностей

# Основні скрипти для виконання
script:
  - npm test # Запуск тестів

# Деплой, якщо всі тести пройшли успішно
deploy:
  provider: heroku # Вказуємо платформу розгортання (наприклад, Heroku)
  api_key:          # Секретний API-ключ для доступу
  secure: "ВАШ_ЗАШИФРОВАНИЙ_API_КЛЮЧ"
  app: your-app-name # Назва додатку на Heroku
  on:
    branch: main # Виконувати деплой тільки з гілки main

# Налаштування сповіщень
notifications:
  email:
    recipients:
      - developer@example.com # Email для сповіщень
    on_success: always # Повідомляти про успішні збірки
    on_failure: always # Повідомляти про невдалі збірки

```

Роз'яснення секцій:

language: node_js – вказує, що проект використовує середовище Node.js.

node_js – перелік версій Node.js, які Travis CI використовуватиме для тестування.

branches – гілки, для яких Travis CI виконуватиме побудову та тестування.

before_script – команди, які виконуються до запуску основного скрипту (наприклад, встановлення залежностей).

script – основні скрипти, що виконуються в рамках збірки. У прикладі це запуск тестів через `npm test`;

deploy – налаштування автоматичного розгортання:

- **provider** – платформа для деплою (у прикладі використовується Heroku);
- **api_key** – секретний ключ доступу (він зашифрований для безпеки);
- **app** – назва додатку, який розгортається;
- **on** – умови, за яких деплой виконується (наприклад, гілка `main`).

notifications – налаштування сповіщень про результати збірки:

- Email, на який будуть надсилатися повідомлення;
- Умови для сповіщення (`on_success`, `on_failure`).

Ця конфігурація автоматизує процес тестування та деплою Node.js-додатків, забезпечуючи стабільність і швидкість розробки.

2.4. Переваги неперервної інтеграції

Зменшення ризиків

Швидке виявлення помилок: CI автоматично запускає тести після кожної зміни в коді, що дозволяє оперативно виявляти баги на ранніх етапах розробки. Це зменшує ризик появи серйозних проблем у виробничому середовищі, які можуть призвести до простоїв або втрати даних.

Мінімізація конфліктів між змінами: Регулярна інтеграція коду зменшує кількість конфліктів між гілками, оскільки зміни об'єднуються частіше і в менших обсягах. Це робить процес розробки більш стабільним і передбачуваним.

Прискорення розробки

Автоматизація рутинних завдань: CI/CD бере на себе завдання, пов'язані зі збіркою, тестуванням та деплойментом, що значно економить час розробників. Наприклад, перевірка залежностей або запуск великих наборів тестів виконується автоматично без втручання людини.

Паралельні процеси: CI дозволяє виконувати тестування і збірку одночасно для різних середовищ або конфігурацій. Це прискорює цикл зворотного зв'язку і дозволяє швидше отримувати готовий до використання продукт.

Поліпшення якості коду

Регулярне тестування: CI забезпечує запуск автоматичних тестів після кожної зміни в репозиторії. Це дозволяє перевіряти, чи не порушує новий код існуючої функціональності, і допомагає підтримувати високу якість продукту.

Стандартизація: CI інструменти можуть включати перевірку стилю коду (linting) і аналіз статичного коду, що сприяє дотриманню єдиних стандартів у команді.

Зменшення технічного боргу: Постійне тестування й інтеграція допомагають уникати накопичення технічного боргу, який може ускладнити майбутній розвиток продукту.

Ефективна командна робота

Прозорість процесів: CI надає чітке уявлення про статус змін у проєкті. Кожен учасник команди знає, що саме працює, а що потребує уваги, завдяки звітам про успішність збірок і тестів.

Зменшення залежності від окремих розробників: Автоматизація процесів означає, що менше кроків залежить від ручної роботи. Це полегшує передачу обов'язків у команді та знижує ризик простоїв через відсутність когось із учасників.

Швидкий зворотний зв'язок: Коли система CI виявляє помилку, розробники одразу отримують повідомлення через інтегровані сервіси (email, Slack тощо), що дозволяє оперативно виправляти проблеми.

Зменшення витрат

Економія часу: Завдяки автоматизації, команди витрачають менше часу на ручну перевірку коду та повторювані завдання, що зменшує загальні витрати на розробку.

Зменшення вартості виправлення багів: Виявлення та усунення помилок на ранніх етапах коштує значно дешевше, ніж у пізніх фазах розробки або вже у виробничому середовищі.

Прискорення доставки продукту:

CI/CD спрощує та пришвидшує процес випуску нових версій програмного забезпечення. Регулярні малі релізи дають змогу користувачам отримувати нові функції швидше, а командам – тестувати та перевіряти відгуки в реальних умовах.

Таким чином, неперервна інтеграція дозволяє покращити ефективність, якість і надійність процесу розробки, знижуючи ризики та витрати, що робить її ключовою практикою для сучасних розробницьких команд.

Неперервна інтеграція є ключовим компонентом сучасного DevOps. Її впровадження дозволяє досягти високої швидкості розробки, зменшити кількість помилок та забезпечити якісний продукт для кінцевих користувачів. У

поєднанні з безперервною доставкою та розгортанням CI забезпечує стабільний розвиток програмного забезпечення, що відповідає сучасним вимогам.

Контрольні питання до розділу 2

- 1) Яка основна мета CI?
- 2) Якими є основні етапи процесу безперервної інтеграції?
- 3) Які переваги забезпечує використання CI для команд розробників?
- 4) Які види автоматизованих тестів використовуються в процесі безперервної інтеграції?
- 5) Які переваги використання Jenkins як інструменту CI можна виділити?
- 6) Яке призначення конфігураційного файлу `.circleci/config.yml` та які основні компоненти він містить?
- 7) Якою є послідовність етапів у процесі роботи Travis CI від внесення змін у код до розгортання?
- 8) Яке значення має інтеграція CI/CD з системами контролю версій?
- 9) В чому полягає головна особливість GitHub Actions порівняно з іншими CI/CD інструментами?
- 10) Яким чином безперервна інтеграція допомагає зменшити технічний борг у проєктах?
- 11) Який процес у DevOps забезпечує автоматичне збирання та тестування коду після його інтеграції?
 - a) Continuous Delivery
 - b) Continuous Integration
 - c) Continuous Monitoring
 - d) Continuous Testing
- 12) Що з перерахованого є основною метою безперервної інтеграції (CI)?
 - a) Зменшення витрат на розробку
 - b) Автоматизація процесу інтеграції змін у коді та раннє виявлення помилок
 - c) Збільшення кількості розробників у команді
 - d) Прискорення процесу написання коду
 - e) Зменшення кількості тестів
- 13) Який з наведених процесів НЕ є етапом безперервної інтеграції?
 - a) Автоматичне збирання
 - b) Тестування
 - c) Внесення змін у код
 - d) Маркетингові дослідження
 - e) Статичний аналіз коду

- 14) Які з наведених тестів перевіряють окремі модулі або функції коду?
- a) Інтеграційні тести
 - b) Юніт-тести
 - c) Перформанс-тести
 - d) Функціональне тестування
 - e) Аудит коду
- 15) Який з перерахованих інструментів найкраще інтегрується з GitHub для CI/CD?
- a) Travis CI
 - b) GitHub Actions
 - c) Jenkins
 - d) CircleCI
 - e) AWS CodePipeline
- 16) Який з наступних інструментів не належить до інструментів CI/CD?
- a) Travis CI
 - b) GitHub Actions
 - c) Jenkins
 - d) CircleCI
 - e) Docker
- 17) Що з переліченого можна віднести до недоліку Jenkins як інструменту безперервної інтеграції?
- a) Відсутність плагінів
 - b) Висока вартість
 - c) Високі вимоги до налаштувань
 - d) Обмеження безкоштовного тарифу
 - e) Низька швидкість роботи?
- 18) Що є основною перевагою GitLab CI/CD?
- a) Найнижча вартість
 - b) Інтеграція з GitLab та масштабованість
 - c) Найпростіше налаштування
 - d) Відсутність обмежень у безкоштовному тарифі
 - e) Найбільша кількість плагінів?
- 19) Який компонент в CircleCI відповідає за визначення послідовності дій для виконання завдання?
- a) workflows
 - b) jobs
 - c) steps
 - d) docker
 - e) version

20) Що є типовим результатом успішного проходження всіх етапів роботи інструментів безперервної інтеграції (CI)?

- a) Видалення репозиторію.
- b) Автоматичне розгортання системи.
- c) Створення конфігураційного файлу.
- d) Відправка звіту про помилки.
- e) Архівування всіх даних.

21) Встановіть відповідність між видами тестів та їх призначенням:

A. Юніт-тести	1) Оцінюють взаємодію між компонентами
B. Інтеграційні тести	2) Перевіряють окремі модулі або функції
C. Функціональні тести	3) Визначають продуктивність та масштабованість системи
D. Перформанс-тести	4) Перевіряють відповідність функцій системи очікуваним результатам

22) Встановіть відповідність між CI/CD сервісами та їх характерними особливостями:

A. Jenkins	1) Зручна інтеграція з GitHub, зручність використання
B. GitHub Actions	2) Відкрите ПЗ, велика кількість плагінів
C. GitLab CI/CD	3) Простота використання, висока швидкість, обмежений функціонал у безкоштовному тарифі
D. CircleCI	4) Зручна інтеграція з GitLab, масштабованість

23) Встановіть відповідність між перевагами CI та їх обґрунтуванням:

A. Зменшення ризиків	1) Автоматизація рутинних завдань, таких як тестування та збірка.
B. Прискорення розробки	2) Швидке виявлення помилок і мінімізація конфліктів у коді.
C. Поліпшення якості коду	3) Регулярне тестування та дотримання стандартів коду.
D. Зменшення витрат	4) Економія часу і зменшення вартості усунення помилок.

24) Встановіть відповідність між етапами роботи Travis CI та їх описом:

A. Ініціалізація змін B. Читання конфігурації C. Запуск тестів D. Обробка результатів	1) Виконуються збірка коду, перевірка залежностей та тестування. 2) Розробник додає зміни у репозиторій, що тригерить процес CI. 3) Система завантажує .travis.yml для налаштування середовища виконання. 4) Успішний результат може призводити до автоматичного розгортання.
--	--

25) Встановіть відповідність між компонентами конфігураційного файлу CircleCI та їх призначенням:

A. version: B. jobs C. steps D. workflows	1) Визначає робочий процес, що складається з одного або кількох завдань 2) Вказує версію конфігураційного формату з доступом до розширених функцій 3) Описує завдання, які виконуються на CI/CD 4) Описує послідовність дій для виконання завдання
--	---

РОЗДІЛ 3. СИСТЕМИ КОНТРОЛЮ ВЕРСІЙ

У процесі неперервної інтеграції система контролю версій виконує кілька важливих функцій, які сприяють автоматизації, ефективності та узгодженості роботи команди розробників. Однією з таких функцій є централізація змін. Код зберігається в єдиному репозиторії, який є доступним для всіх учасників проєкту. Це забезпечує кожного розробника актуальною версією коду, запобігаючи дублюванню та втраті даних.

Відстеження історії змін є ще однією важливою складовою. Система дозволяє зберігати всі внесені коміти з інформацією про те, хто та коли вніс певні зміни. Це полегшує аналіз еволюції проєкту та сприяє швидкому виявленню причин потенційних проблем у коді.

Система контролю версій також є основою для автоматизації перевірок. Будь-які зміни в коді запускають процеси автоматизованого тестування, збірки та перевірки якості коду. Це дозволяє вчасно виявляти помилки та забезпечувати стабільність розробки. Така інтеграція з CI/CD-пайплайнами підвищує надійність програмного продукту.

Підтримка колаборації є невід’ємною перевагою системи контролю версій. Завдяки використанню гілок (branches) розробники можуть паралельно працювати над різними функціями, не створюючи при цьому конфліктів у коді. Цей підхід дозволяє гнучко розподіляти задачі між членами команди та впроваджувати нові функціональні можливості без ризику порушення основного функціоналу.

Останнім елементом є інтеграція, яка забезпечує можливість швидкого злиття змін із різних гілок у головну гілку репозиторію. Це гарантує актуальність основної версії коду та дозволяє швидко підготувати його для релізу. Також така інтеграція сприяє мінімізації конфліктів під час об’єднання змін.

Система контролю версій є важливим інструментом у процесі неперервної інтеграції, оскільки вона забезпечує ефективне управління змінами, сприяє автоматизації та підтримує колективну роботу розробників.

3.1. Централізовані та розподілені системи збереження даних

Системи збереження даних є невід’ємною частиною сучасних інформаційних технологій, оскільки вони забезпечують ефективне управління, доступ і захист даних у різних контекстах, включаючи розробку програмного забезпечення. Одним із ключових аспектів управління даними є організація систем контролю версій (СКВ), які дозволяють відслідковувати зміни в коді, підтримувати його стабільність та координувати роботу команд розробників. У

цьому контексті важливо розуміти різницю між централізованими та розподіленими системами збереження даних, оскільки кожен з цих підходів має свої переваги та недоліки, що впливають на вибір інструментів для конкретного проєкту.

Централізовані системи контролю версій, такі як Subversion (SVN) чи Perforce, базуються на єдиному центральному сервері, на якому зберігаються всі дані. У свою чергу, розподілені системи, зокрема Git, дозволяють кожному розробнику мати повну копію репозиторія на своєму пристрої, що дає більше гнучкості та можливостей для автономної роботи. Ці два підходи суттєво відрізняються за архітектурою, функціональністю, масштабованістю та можливістю інтеграції з іншими інструментами, такими як системи безперервної інтеграції (CI).

Розуміння переваг та обмежень кожної з систем є важливим для оптимізації робочих процесів у команді та вибору правильного інструменту для ефективного управління змінами в проєкті. У цьому розділі розглядатимемо основні принципи функціонування централізованих і розподілених систем збереження даних, їхні сильні та слабкі сторони, а також їх застосування в контексті сучасних методів розробки програмного забезпечення.

Таблиця 3.1 надає порівняння двох основних підходів до управління версіями в розробці програмного забезпечення – централізованих (CVCS) та розподілених (DVCS) систем. Вона допомагає зрозуміти, як ці системи відрізняються за архітектурою, зручністю використання та можливостями інтеграції, а також показує основні переваги та недоліки кожної з них.

Таблиця 3.1. Порівняння різних типів систем контролю версій

Характеристика	Централізовані системи (CVCS)	Розподілені системи (DVCS)
Архітектура	Єдиний центральний сервер, копії на локальних пристроях	Кожен розробник має повну копію репозиторія
Приклади систем	Subversion (SVN), Perforce	Git, Mercurial
Доступ до історії змін	Лише через центральний сервер	Локальний доступ до повної історії змін
Робота без підключення до сервера	Неможлива	Можлива (локальні операції)
Швидкість операцій	Залежить від швидкості з'єднання	Швидші локальні операції
Можливість роботи з гілками	Підтримка є, але обмежена	Потужна підтримка гілок і злиттів

Масштабованість	Знижується з розміром команди	Підходить для великих команд і проєктів
Приклади інструментів для CI	Jenkins, TeamCity (без прямої інтеграції)	GitHub Actions, GitLab CI/CD, Jenkins (повна інтеграція)

Централізовані системи, такі як Subversion (SVN) та Perforce, дають змогу спрощувати адміністрування та контроль доступу, однак потребують постійного з'єднання з сервером для доступу до історії змін та виконання комітів.

Розподілені системи контролю версій, як Git і Mercurial, зберігають повні копії репозиторіїв на локальних пристроях розробників, що дозволяє працювати в автономному режимі, зменшуючи залежність від центрального сервера. Вони також підтримують потужні механізми для роботи з гілками та злиттям змін, що є критично важливим для великих команд та проєктів. Проте такі системи можуть вимагати більше місця на диску та бути складнішими для новачків через велику кількість доступних функцій.

Наведені в таблиці 3.1. дані дозволяють наочно порівняти ці два підходи за кількома важливими критеріями, що допомагає вибрати найбільш підходящий тип системи контролю версій в залежності від потреб і масштабу проєкту.

Централізовані системи контролю версій (CVCS)

Централізовані системи контролю версій (Centralized Version Control Systems, CVCS) є одним із традиційних підходів до управління змінами в програмному забезпеченні, який широко використовувався до появи розподілених систем. У цій архітектурі всі дані проєкту, включаючи файли, історію змін та метадані, зберігаються на центральному сервері. Розробники отримують локальні копії потрібних файлів із сервера, вносять зміни та завантажують їх назад до центрального репозиторію. Центральний сервер виконує функцію єдиної точки синхронізації, де всі члени команди можуть взаємодіяти з актуальними даними.

Особливістю централізованих систем є їх організованість і централізація управління. Завдяки цьому розробники завжди працюють з однією версією репозиторію, що дозволяє уникати дублювання змін і підтримувати єдину структуру проєкту. Також централізована модель добре підходить для організацій із чітко визначеною ієрархією доступу та адміністративними вимогами.

Серед популярних прикладів централізованих систем можна виділити Subversion (SVN) та Perforce. Ці інструменти забезпечують базовий функціонал для управління версіями, включаючи можливість відслідковувати зміни,

створювати резервні копії та відновлювати попередні версії файлів. Вони стали популярними завдяки простоті впровадження і зрозумілості процесу управління даними.

Subversion (SVN)

Subversion (SVN) – це система контролю версій з відкритим кодом, яка використовується для відстеження змін у файлах і каталогах з плином часу. Вона широко застосовується в розробці програмного забезпечення та інших сферах для управління вихідним кодом, документами та цифровими ресурсами.

Рисунок 3.1 демонструє інтерфейс програми Syncro SVN Client, яка використовується для керування версіями проєктів за допомогою системи контролю версій Subversion.

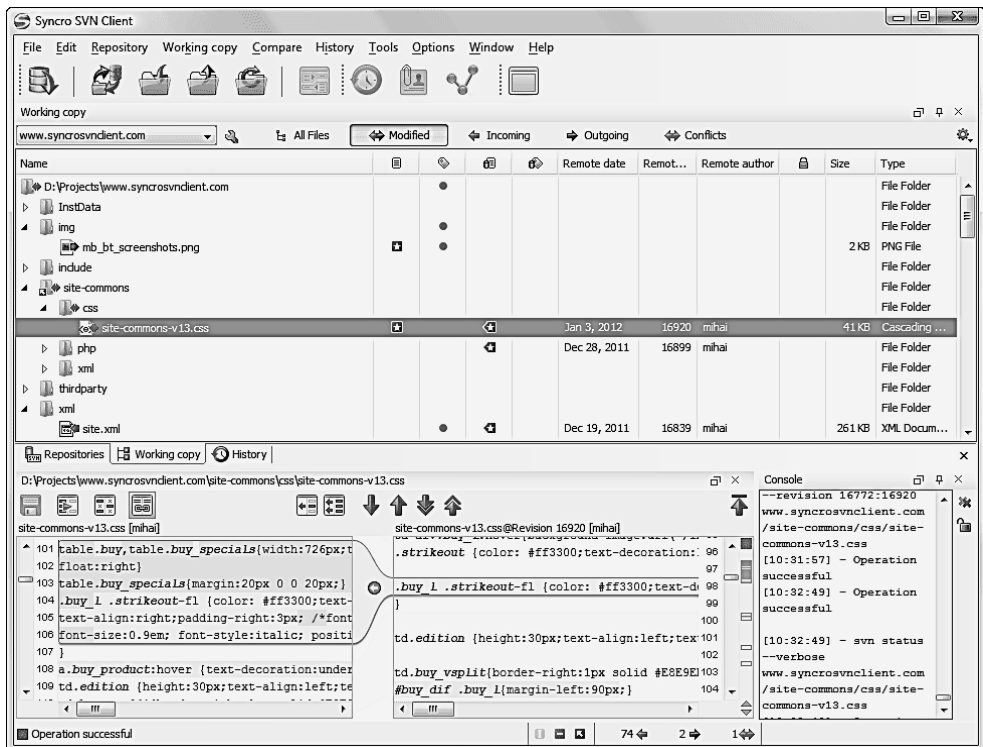


Рис. 3.1. Інтерфейс Syncro SVN Client

Цей інтерфейс допомагає програмістам і розробникам керувати версіями коду, відстежувати зміни, виявляти конфлікти та забезпечувати контроль над процесом розробки.

Розглянемо основні функціональні можливості SVN:

- *контроль версій:*

- зберігає повну історію змін, дозволяючи переглядати попередні версії файлів та відновлювати їх;
- відстежує модифікації, додавання та видалення файлів і папок, забезпечуючи повну прозорість змін;
- підтримує журнал змін із детальною інформацією про автора, час і опис кожного коміту;
- можливість перегляду різниць між версіями (diff) для аналізу змін;
- *централізоване сховище:*
 - використовує централізований сервер для зберігання всіх версій файлів, забезпечуючи єдину точку доступу до даних;
 - клієнти завантажують робочі копії (checkout) та надсилають зміни назад до сховища (commit);
 - підтримує реплікацію сховища для резервного копіювання та підвищення надійності;
- *атомарні коміти:*
 - гарантує, що всі зміни в одному коміті застосовуються як єдина транзакція;
 - запобігає ситуаціям з частковими оновленнями, що може призвести до втрати даних або несумісності;
 - забезпечує цілісність даних навіть у разі збою під час коміту;
- *робота з гілками та злиттям:*
 - дозволяє створювати гілки для паралельної розробки без впливу на основну гілку (trunk);
 - підтримує злиття змін між гілками, полегшуючи інтеграцію нових функцій або виправлень помилок;
 - допомагає відстежувати відмінності між гілками та об'єднувати конфліктуючі зміни;
- *контроль доступу:*
 - надає механізми автентифікації користувачів для захисту від несанкціонованого доступу;
 - підтримує гнучкі налаштування авторизації з можливістю обмеження доступу на рівні файлів і папок;
 - дозволяє ведення журналу дій користувачів для аудиту безпеки;
- *тегування (Tagging):*
 - дозволяє створювати мітки для позначення конкретних версій як релізів або контрольних точок;
 - полегшує повернення до важливих версій для аналізу або випуску оновлень;

- мітки можуть використовуватися для створення знімків проєкту на певний момент часу;
- *блокування файлів:*
 - запобігає конфліктам у файлах, які важко об'єднувати, наприклад графічних або бінарних файлах;
 - дозволяє користувачам блокувати файли для ексклюзивного редагування, забезпечуючи синхронізацію командної роботи;
 - підтримує зняття блокування іншими користувачами у разі виникнення потреби;
- *кросплатформність:*
 - підтримує різні операційні системи, включаючи Windows, Linux, macOS та інші;
 - доступні як командний рядок, так і графічні інтерфейси для роботи з SVN (наприклад, TortoiseSVN для Windows);
 - легка інтеграція з IDE та іншими інструментами розробки для підвищення продуктивності;
- *інтеграція з іншими інструментами:*
 - підтримує інтеграцію з інструментами безперервної інтеграції (CI/CD) для автоматизації процесів розробки;
 - забезпечує сумісність із системами відстеження помилок (наприклад, Jira, Redmine);
 - легко інтегрується з IDE (Eclipse, Visual Studio) через спеціальні плагіни;
- *підтримка офлайн-режиму:*
 - дозволяє працювати з локальною копією репозиторію без підключення до мережі;
 - зміни можна зберігати локально та надсилати в сховище після відновлення підключення;
- *резервне копіювання та відновлення:*
 - підтримує створення резервних копій сховищ для захисту даних від втрат;
 - можливість швидкого відновлення репозиторію у разі збою або пошкодження даних.

SVN залишається популярною системою контролю версій завдяки своїй простоті, надійності та широкому набору функцій. Система керування версіями SVN широко використовується в різних сферах діяльності завдяки своїй надійності та функціональності та підходить як для малих команд, так і для великих організацій, де необхідно зберігати централізовану структуру управління версіями. Вона є ефективним інструментом для розробки

програмного забезпечення, оскільки дозволяє відстежувати зміни в коді, працювати з різними версіями проєктів і полегшує координацію між членами команди розробників.

SVN також застосовується для управління документацією. Вона забезпечує зручне зберігання текстових, графічних та інших типів файлів, дозволяючи вести контроль над їх змінами, зберігати історію редагувань і підтримувати порядок у документації.

У сфері спільної роботи над проєктами SVN надає можливість централізованого доступу до даних, що значно спрощує взаємодію між учасниками команди. Кожен користувач може отримати актуальну версію файлів, вносити правки та синхронізувати зміни з іншими учасниками.

Крім того, SVN часто використовується для архівування та резервного копіювання файлів. Це дозволяє зберігати важливі дані у безпечному середовищі, запобігаючи їх втраті або пошкодженню.

SVN є оптимальним рішенням для організацій, які потребують централізованого підходу до контролю версій, безпечного зберігання інформації та зручних інструментів для спільної роботи над файлами й проєктами.

Perforce

Perforce – це система управління версіями, яка використовується для зберігання, відстеження та керування змінами в коді, документах або інших цифрових ресурсах. Вона особливо популярна серед великих команд розробників завдяки своїй швидкості, масштабованості та здатності працювати з великими файлами.

Рисунок 3.2 демонструє інтерфейс програми Perforce P4V.

Інтерфейс програми побудований таким чином, що зліва розташована панель, яка відображає структуру директорій проєкту з усіма підкаталогами та файлами. У верхній частині знаходиться стандартне меню з такими опціями, як File, Edit, View, Actions, Connection, Tools, Window і Help. Окрема панель з назвою «Pending» (Очікування) призначена для відображення змін, які ще не були надіслані, наприклад, changelist 2293 і 2309.

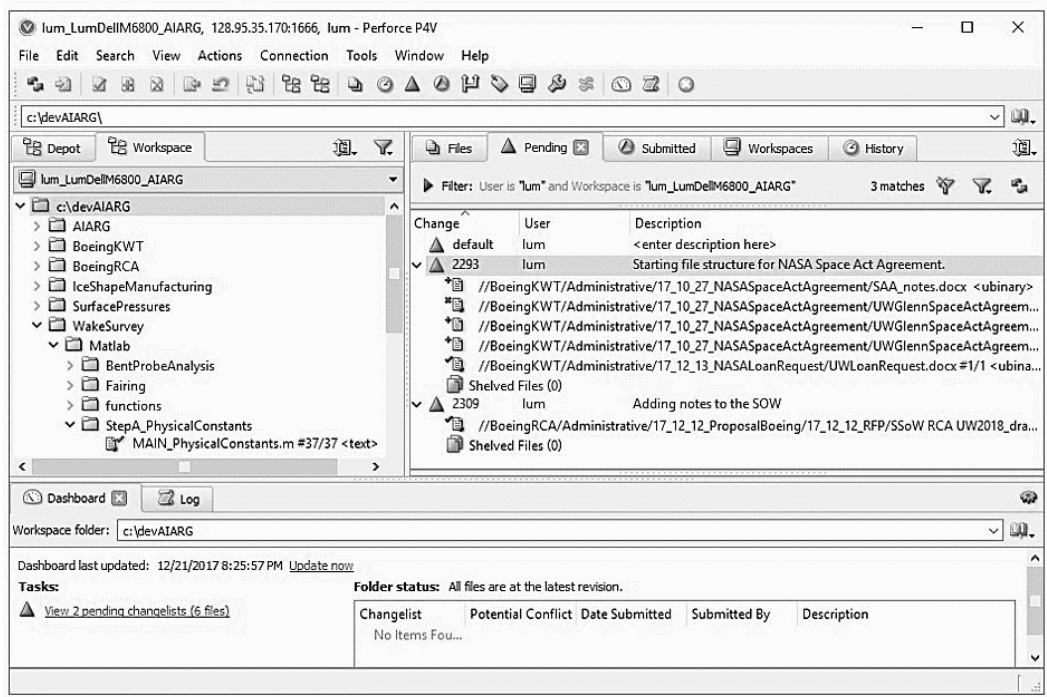


Рис.3.2. Інтерфейс програми Perforce P4V

Нижче наведені основні характеристики Perforce:

- **централізована архітектура:** усі зміни в коді або інших цифрових файлах зберігаються у центральному репозиторії, який слугує єдиним джерелом істини для всієї команди. Це забезпечує чітке відстеження історії змін, полегшує відновлення попередніх версій і спрощує управління конфліктами. Завдяки централізованому підходу знижується ризик втрати даних, а також підвищується прозорість роботи над проектами;
- **висока продуктивність:** Perforce створений для ефективної роботи з великими файлами та проектами, які можуть включати мільйони рядків коду або важкі мультимедійні ресурси. Завдяки потужному серверу Perforce, система обробляє запити з мінімальними затримками, навіть у середовищах з великою кількістю користувачів. Це робить її ідеальним вибором для індустрій, де потрібна швидкість і масштабованість, таких як ігровий дизайн, анімація чи машинобудування;
- **сумісність:** Perforce підтримує інтеграцію з широким спектром інструментів і середовищ розробки, включаючи Jenkins, Git, Visual Studio, Unreal Engine, Unity та інші. Крім того, система дозволяє використовувати її одночасно з Git завдяки функції Helix4Git, що надає

можливість розробникам працювати у звичному середовищі без необхідності змінювати свій робочий процес;

- *інструменти для співпраці*: Perforce дозволяє декільком розробникам працювати над одним проєктом без ризику конфліктів. Система автоматично відстежує, які файли змінюються, і попереджає про можливі конфлікти, надаючи засоби для їх швидкого вирішення. Крім того, Perforce забезпечує потужну функцію злиття змін (merge), яка спрощує інтеграцію різних гілок розробки. Завдяки інструментам для аналізу історії змін команда може бачити, хто, коли і що змінив у проєкті;
- *безпека*: Perforce пропонує високий рівень безпеки завдяки гнучким налаштуванням доступу. Керівники проєктів можуть встановлювати права доступу для кожного користувача, групи або окремих файлів. Це дозволяє захищати чутливі дані та зменшити ризики несанкціонованого доступу. Крім того, Perforce підтримує аутентифікацію через LDAP, Active Directory і багатофакторну автентифікацію, що додає ще один рівень захисту;
- *масштабованість*: система легко масштабується для роботи з командами будь-якого розміру – від невеликих груп до глобальних організацій із тисячами співробітників. Сервер Perforce може обробляти великі обсяги даних і одночасно підтримувати сотні тисяч транзакцій;
- *інтеграція DevOps*: Perforce має інструменти, які добре вписуються в DevOps-підхід, дозволяючи автоматизувати розгортання, тестування та доставку програмного забезпечення. Це допомагає зменшити час розробки та збільшити якість кінцевого продукту;
- *підтримка великих бінарних файлів*: окрім коду, Perforce чудово працює з великими файлами, такими як 3D-моделі, відео або текстури. Це робить її популярною серед студій, які працюють з ігровим контентом або іншими ресурсами великого розміру;
- *кросплатформеність*: Perforce підтримує різні операційні системи, включаючи Windows, macOS і Linux, а також забезпечує доступ через веб-інтерфейс і командний рядок. Це дозволяє розробникам працювати з системою на зручній для них платформі.

Таким чином, Perforce – це потужний інструмент, який забезпечує надійне управління версіями, швидкість і продуктивність, необхідні для масштабних і складних проєктів. Завдяки своїм характеристикам, ця система є одним із найкращих виборів для команд, що працюють над інноваційними продуктами в технологічній, медіа- або ігровій індустріях.

Perforce часто використовується в галузях, де потрібна обробка великої кількості даних або високий рівень надійності та контролю, наприклад, у розробці програмного забезпечення, ігор або 3D-графіки.

Переваги та обмеження централізовані системи контролю версій

Основною перевагою централізованих систем є їх простота у використанні та адмініструванні. Єдиний сервер забезпечує централізований контроль за проектом, що полегшує управління правами доступу та моніторинг активності команди. Адміністрування такої системи є менш ресурсозатратним, оскільки всі операції відбуваються через одну точку доступу. Це дозволяє швидко впроваджувати такі системи в невеликих командах чи організаціях із чіткими робочими процесами.

Ще однією перевагою є забезпечення безпеки даних. Центральний сервер можна налаштувати для автоматичного створення резервних копій, що мінімізує ризик втрати важливої інформації. Крім того, централізовані системи дозволяють ефективно керувати користувачами, розмежовуючи їхні права доступу залежно від ролей у проєкті.

Попри свої переваги, централізовані системи мають і певні обмеження. Головним недоліком є їх залежність від центрального сервера. У разі його виходу з ладу, втрати зв'язку або перевантаження розробники не зможуть отримати доступ до репозиторію чи виконувати коміти. Це може призвести до затримок у розробці, особливо якщо команда працює в умовах віддаленого доступу. Крім того, централізована архітектура менш ефективно масштабується, оскільки зростання кількості розробників чи обсягу даних може створювати значне навантаження на сервер.

Однак навіть із такими обмеженнями централізовані системи контролю версій залишаються актуальними в низці сценаріїв. Вони добре підходять для проєктів із невеликими командами, чіткою ієрархією доступу та мінімальною потребою в автономній роботі. У поєднанні з надійною інфраструктурою та регулярними резервними копіями централізовані системи можуть стати стабільною основою для управління версіями в багатьох організаціях.

Однак централізовані системи контролю версій мають певні обмеження, які можуть створювати виклики для команд, особливо в умовах складних або масштабованих проєктів, де вимоги до продуктивності, автономності та надійності значно зростають. Одним із найбільших недоліків є повна залежність від центрального сервера. Якщо сервер стає недоступним через технічні несправності, перебої в електропостачанні або проблеми з мережею, вся команда втрачає доступ до репозиторію. У таких випадках розробники не можуть переглядати історію змін, синхронізувати нові коміти чи отримувати

оновлення від інших членів команди. Це може призводити до значних затримок у роботі над проєктом, особливо якщо відновлення роботи сервера займає тривалий час.

Ще одним суттєвим обмеженням централізованих систем є їхня обмежена здатність до масштабування. У великих командах із високою частотою змін навантаження на центральний сервер може значно зростати, що призводить до зниження швидкості роботи системи. Це може проявлятися у вигляді тривалого часу завантаження чи оновлення файлів, зниження швидкості синхронізації або навіть перевантаження сервера. Такі проблеми роблять централізовані системи менш придатними для великих розподілених команд, які працюють над масштабними проєктами з великою кількістю змін.

Іншим викликом є відсутність автономності для розробників. У централізованих системах кожна дія, пов'язана з історією змін або управлінням версіями, залежить від доступу до сервера. Це означає, що розробники, які працюють у віддалених місцях із ненадійним інтернет-з'єднанням, можуть стикатися з труднощами. Наприклад, вони не зможуть комітити зміни чи переглядати попередні версії файлів без активного підключення до сервера.

Попри ці недоліки, централізовані системи контролю версій залишаються популярними в певних сценаріях, де їх переваги переважають обмеження. Вони є особливо зручними для невеликих команд, які працюють в одній локації, мають стабільну інфраструктуру та не потребують складних робочих процесів. У таких випадках стабільний централізований підхід стає важливішим за гнучкість, забезпечуючи простоту в адмініструванні, зрозумілість процесів і мінімальну кількість технічних залежностей.

Крім того, централізовані системи добре підходять для організацій, які мають чіткі вимоги до безпеки та контролю. Адміністрування таких систем є більш прямолінійним, оскільки всі операції зосереджені на одному сервері, що дозволяє легко управляти правами доступу, стежити за активністю користувачів і забезпечувати регулярне створення резервних копій. Це робить централізовані системи привабливим вибором для організацій, які віддають перевагу стабільній і передбачуваній інфраструктурі.

У перспективі централізовані системи можуть залишатися актуальними для спеціалізованих застосувань, наприклад, у середовищах із чітко регламентованими вимогами, таких як банківська справа, державні установи чи проєкти з високими вимогами до конфіденційності. У таких умовах стабільність і централізований контроль часто є важливішими за автономність та гнучкість, які пропонують розподілені системи.

3.2. Розподілені системи контролю версій (DVCS)

Розподілені системи контролю версій (Distributed Version Control Systems, DVCS) являють собою сучасний підхід до управління змінами в програмному забезпеченні, який вирізняється високою гнучкістю, автономністю та продуктивністю. Основною особливістю таких систем є те, що кожен розробник має повну копію репозиторія, яка включає не лише актуальну версію файлів, а й всю історію змін, метадані та гілки. Завдяки цьому архітектурному підходу розробники можуть працювати автономно, не залежачи від доступу до центрального сервера. Вони можуть переглядати попередні версії файлів, створювати коміти, зливати гілки та вирішувати конфлікти, навіть перебуваючи в умовах відсутності інтернет-з'єднання.

Одними з найбільш популярних прикладів розподілених систем є Git і Mercurial. Git, створений Лінусом Торвальдсом у 2005 році для розробки ядра Linux, став стандартом у багатьох галузях завдяки своїй швидкості, масштабованості та багатому набору функцій. Він є основою для таких популярних платформ, як GitHub, GitLab і Bitbucket, які забезпечують додаткові можливості для командної співпраці, включаючи управління репозиторіями, автоматизацію процесів CI/CD та інструменти для код-рев'ю.

Функціональність розподілених систем дозволяє організувати ефективну роботу як у невеликих командах, так і у великих міжнародних організаціях. Завдяки підтримці складних робочих процесів, таких як паралельна розробка різних функціональностей у гілках, вони забезпечують ефективність навіть у випадках, коли над одним проєктом працює декілька сотень чи тисяч розробників.

Важливою перевагою DVCS є здатність забезпечувати швидкодію завдяки локальним операціям. Наприклад, операції з історією комітів, створення гілок або пошуку змін виконуються локально на пристрої розробника, що дозволяє уникати затримок, пов'язаних із мережею. Це не лише підвищує продуктивність роботи, але й сприяє кращій організації складних процесів розробки, таких як інтеграція та тестування змін.

Окрім цього, DVCS пропонують підвищену стійкість до збоїв. Якщо центральний сервер тимчасово недоступний, команда може продовжувати працювати автономно, а всі зміни будуть синхронізовані з сервером, як тільки доступ буде відновлено. Це значно знижує ризик простоїв і втрати даних, що є критично важливим для великих команд і проєктів із високими вимогами до безперервної розробки.

Попри свої численні переваги, DVCS мають і певні особливості, які варто враховувати. Зокрема, кожен розробник має зберігати повну копію репозиторія, що може створити підвищені вимоги до дискового простору, особливо у великих

проектах із довгою історією змін. Наприклад, великі репозиторії з десятками гігабайтів даних можуть вимагати значного місця на пристроях розробників.

Ще одним викликом є більш крута крива навчання для новачків. Інструменти на основі DVCS, такі як Git, мають широкий функціонал і підтримують складні робочі сценарії, але це може спантеличувати нових користувачів. Наприклад, процеси вирішення конфліктів при злитті гілок або робота з віддаленими репозиторіями можуть здаватися складними без належної підготовки. Однак ця проблема вирішується завдяки розвиненій документації, навчальним матеріалам і підтримці великої спільноти користувачів.

Mercurial

Mercurial – це одна з популярних розподілених систем контролю версій з для управління змінами в програмному забезпеченні. Вона була розроблена з метою забезпечення високої продуктивності, гнучкості та простоти використання, з огляду на що є зручним інструментом для індивідуальних розробників та великих команд.

Mercurial був створений у квітні 2005 року австралійським розробником Меттом Макаллом (Matt Mackall). Основною мотивацією для створення цієї системи контролю версій було раптове припинення розробки проекту BitKeeper, який до цього часу використовувався як система контролю версій ядра Linux.

BitKeeper був комерційним продуктом, який надавав безкоштовні ліцензії для відкритих проектів, але через конфлікти між спільнотою розробників Linux та власниками BitKeeper, підтримка безкоштовної версії була припинена. Це створило потребу у створенні альтернативи з відкритим кодом, яка б була незалежною від комерційних обмежень.

Рисунок 3.3 показує графічний інтерфейс користувача системи контролю версій Mercurial.

Інтерфейс містить кілька важливих елементів для перегляду історії змін. Область під назвою *Graph (Граф)* візуалізує історію комітів у вигляді графа, де вузли та лінії демонструють структуру гілок і їх розгалуження. Поруч знаходиться стовпець *Rev (Ревізія)*, який відображає нумерацію комітів у послідовному порядку – від 0 до 11. Стовпець *Branch (Гілка)* показує назви гілок, наприклад, *default* або *v1.0 maintenance*. У полі *Description (Опис)* подано короткий опис змін, що були внесені в кожному коміті, зокрема повідомлення на кшталт «merged in v1.0 bugfixes». Останній стовпець, *Author (Автор)*, вказує, хто саме здійснив коміт.

Graph	Rev	Branch	Description	Author
	1...	default	★ Working Directory ★	Mark Heath
	11	default	default tip merged in v1.0 bugfixes	Mark Heath
	10	v1.0 maintenance	v1.0 maintenance merged in a bugfix	Mark Heath
	9	v1.0 maintenance	contributed bugfix for v1.0	Someone
	8	default	merged in the new feature	Mark Heath
	7	default	contributing a new feature in v2	A.N.Other
	6	v1.0 maintenance	another v1.0 bugfix after the fork	Mark Heath
	5	default	another change on v2 after the fork	Mark Heath
	4	default	more work on v2	Mark Heath
	3	v1.0 maintenance	bugfix for v1.0	Mark Heath
	2	default	beginning work on v2	Mark Heath
	1	default	v1.0 release	Mark Heath
	0	default	first version	Mark Heath

Рис. 3.3 Графічний інтерфейс користувача системи контролю версій Mercurial

Основні принципи розробки Mercurial є наступними:

- *швидкість та ефективність*: Mercurial створювався з акцентом на продуктивність, щоб обробка великих проектів займала мінімальний час;
- *простота у використанні*: розробник прагнув забезпечити інтуїтивно зрозумілий інтерфейс для користувачів;
- *кросплатформеність*: Mercurial мав працювати на всіх основних операційних системах;
- *мінімалізм*: код бази Mercurial був написаний компактно та зрозуміло, що спрощувало його підтримку й розширення;
- *розподілена модель*: кожен користувач міг мати локальну копію репозиторію, що дозволяло працювати офлайн і робити систему більш надійною.

Mercurial став однією з перших систем контролю версій з відкритим кодом, розроблених як альтернатива BitKeeper. Він отримав швидку популярність серед розробників, особливо завдяки своїй легкості, швидкості та простоті. Хоча з часом інші системи, такі як Git, також здобули популярність, Mercurial залишився важливим гравцем у сфері управління версіями, особливо в середовищах, де потрібна висока продуктивність і стабільність.

Нижче наведені основні характеристики Mercurial:

- *розподілена архітектура*: Mercurial є розподіленою системою контролю версій, що означає, що кожен розробник має повну локальну

копію репозиторію, включаючи всю історію змін. Така архітектура забезпечує кілька переваг:

- робота в автономному режимі: можливість виконувати більшість операцій, як-от створення комітів, аналіз історії чи робота з гілками, без підключення до центрального сервера;
- надійність: локальні копії репозиторію дозволяють уникнути втрати даних у разі збою центрального сервера;
- гнучкість: кожен користувач може експериментувати з кодом у своїй копії, не впливаючи на інших розробників;
- *простота використання*: Mercurial спроектована так, щоб бути інтуїтивно зрозумілою та зручною у використанні навіть для новачків у системах контролю версій.
 - зручний інтерфейс командного рядка: команди мають логічну структуру, що робить їх легкими для запам'ятовування;
 - документація: вичерпна документація та навчальні матеріали допомагають швидко освоїти основи роботи з системою;
 - простий розгортання: інсталяція Mercurial займає мінімум часу, і її можна налаштувати для різних середовищ розробки;
- *швидкодія*: висока швидкість роботи є однією з ключових характеристик Mercurial.
 - локальні операції: оскільки більшість операцій виконується локально, це забезпечує швидкий доступ до історії змін, створення нових гілок та комітів;
 - оптимізація для великих проектів: Mercurial ефективно працює навіть з великими репозиторіями, що включають тисячі файлів і гігабайти даних;
- *гілкування та злиття*: Mercurial підтримує потужні механізми для роботи з гілками, що особливо корисно в командній розробці.
 - легка робота з гілками: гілки дозволяють працювати над різними функціями або виправленнями ізольовано, не впливаючи на основну гілку;
 - безконфліктне злиття: інструменти злиття в Mercurial зводять до мінімуму конфлікти під час інтеграції змін;
- *безпека та надійність*: Mercurial гарантує, що всі зміни в репозиторії захищені та доступні для відновлення.
 - криптографічний контроль змін: кожен коміт має унікальний хеш, що дозволяє ідентифікувати та перевіряти кожну зміну;
 - захист історії: історія змін ніколи не втрачається, що робить відкат до попереднього стану репозиторія швидким і надійним;

- *інтеграція з іншими інструментами*: Mercurial легко інтегрується з популярними інструментами розробки та платформами;
 - платформи для хостингу: підтримує інтеграцію з сервісами, такими як Bitbucket, які спочатку розроблялися для роботи з Mercurial;
 - CI/CD системи: добре інтегрується з інструментами для автоматизації, такими як Jenkins або Bamboo, що дозволяє будувати, тестувати та розгортати проекти автоматично;
 - розширюваність: система підтримує плагіни, які можна налаштувати для специфічних потреб команди чи проекту.
- *кросплатформеність*: Mercurial працює на різних операційних системах, таких як Windows, macOS та Linux, що робить її універсальним рішенням для команд із різними середовищами розробки.

Завдяки своїм характеристикам, Mercurial забезпечує продуктивність, зручність і надійність, що робить його підходящим як для індивідуальних розробників, так і для великих команд, які працюють над складними проектами.

До переваг користування Mercurial можна віднести наступні:

- *легка у навчанні*:
 - інтерфейс командного рядка Mercurial розроблений для максимальної зручності. Команди мають зрозумілий синтаксис, що дозволяє новачкам швидко освоїти основні операції, такі як commit, merge, log, і update.
 - Mercurial надає добре структуровану документацію та навчальні посібники, які пояснюють як базові, так і розширені функції.
 - доступні сторонні інструменти, такі як TortoiseHg, які додають графічний інтерфейс для роботи з Mercurial, полегшуючи його використання для тих, хто не звик до роботи в терміналі.
- *продуктивність*:
 - більшість операцій, включаючи створення комітів і перегляд історії, виконуються локально, що забезпечує миттєву відповідь системи.
 - Mercurial добре справляється з репозиторіями, які містять тисячі файлів, завдяки оптимізованому зберіганню та швидкій обробці даних, тому є ефективним для великих проектів.
 - у випадках, коли використовується центральний сервер, його навантаження залишається мінімальним, оскільки більшість обчислень виконується локально.

- *гнучкість*:
 - підтримка різних робочих процесів – Mercurial дозволяє легко адаптуватися до різних стилів роботи команд, будь то централізовані процеси або розподілена розробка.
 - розробники можуть створювати відокремлені гілки для нових функцій або експериментів, зберігаючи при цьому стабільність основної гілки.
 - злиття змін з різних гілок проходить гладко завдяки потужним інструментам інтеграції.
- *надійність*
 - усі зміни зберігаються в історії репозиторію, що дозволяє відновлювати будь-який попередній стан проекту.
 - криптографічний контроль: Mercurial використовує хешування для перевірки змін, що забезпечує захист від випадкових або зловмисних змін історії.
 - відсутність «руйнування» даних з огляду на те, що історія змін ніколи не видаляється автоматично і це гарантує безпеку навіть у випадку помилкових операцій.
- *Кросплатформність* – Mercurial працює на різних операційних системах, включаючи Windows, macOS і Linux, що забезпечує його універсальність у гетерогенних середовищах розробки.
- *інтеграція з екосистемою розробки*
 - підтримує інтеграцію з популярними платформами, такими як Bitbucket, а також власні сервери.
 - легко інтегрується з інструментами для автоматизації процесів розробки та розгортання, такими як Jenkins або GitLab CI.
 - Mercurial можна розширювати за допомогою плагінів, що дозволяє адаптувати систему до специфічних потреб проекту.
- *командна робота*
 - Mercurial дозволяє багатьом розробникам працювати одночасно, зберігаючи незалежність кожного в локальній копії репозиторію.
 - Легкість інтеграції змін зменшує ризик конфліктів і сприяє безперебійній співпраці між членами команди.

Завдяки цим перевагам Mercurial залишається надійним і продуктивним інструментом для управління версіями, який підходить як для невеликих проектів, так і для масштабних корпоративних розробок.

Mercurial має значно меншу популярність у спільноті розробників порівняно з Git, що обмежує доступ до ресурсів, таких як навчальні матеріали, приклади використання та технічна підтримка. Це також впливає на розмір

спільноти користувачів, ускладнюючи пошук відповідей на питання чи вирішення проблем. Менша кількість активних розробників у проєкті Mercurial уповільнює розвиток системи та її адаптацію до сучасних вимог.

Ще одним недоліком є обмежена кількість інтеграцій з іншими інструментами та платформами. Хоча Mercurial підтримує інтеграцію з популярними сервісами, такими як Bitbucket, вона не має такого рівня підтримки, як Git, особливо серед новітніх інструментів для безперервної інтеграції (CI/CD) та DevOps. Наприклад, після припинення підтримки Mercurial на GitHub у 2020 році, розробникам стало складніше інтегрувати його у свої робочі процеси.

Багато інструментів і платформ розробки за замовчуванням орієнтовані на Git, що створює додаткові труднощі для команд, які бажають використовувати Mercurial. Інтерфейси для інтеграції можуть бути менш зручними або зовсім відсутні для специфічних інструментів, що збільшує час налаштування та ускладнює робочі процеси.

Загалом, ці обмеження роблять Mercurial менш привабливим вибором для нових користувачів і команд, які прагнуть інтегруватися у сучасну екосистему розробки. Однак для проєктів, які вже використовують Mercurial, ці недоліки можуть бути не настільки суттєвими за умови стабільного і добре організованого процесу роботи.

Таблиця 3.2 допомагає побачити ключові переваги та недоліки Mercurial, що дозволяє зробити зважений вибір залежно від потреб конкретного проєкту чи команди.

Таблиця 3.2. Переваги та недоліки Mercurial

Категорія	Переваги	Недоліки
Простота використання	Простий інтерфейс, легкий у навчанні	Обмежена кількість навчальних матеріалів
Швидкодія	Локальні операції виконуються дуже швидко	Може програвати в продуктивності при роботі з великими командами
Гнучкість	Підтримує різні робочі процеси	Менше інструментів для адаптації під специфічні потреби
Інтеграція	Добре працює з Bitbucket та іншими платформами	Відсутність підтримки GitHub і обмежена кількість інтеграцій
Спільнота	Надійність і стабільність підтверджені користувачами	Менш активна спільнота
Гілкування та злиття	Зручне і стабільне злиття	Вужчий набір функцій у порівнянні з Git
Безпека	Криптографічне збереження змін	Менш популярна для корпоративних потреб

Таким чином, Mercurial – це потужна, стабільна та ефективна система контролю версій, яка демонструє високу продуктивність та гнучкість у роботі з проєктами будь-якого масштабу. Її простота, інтуїтивний інтерфейс, можливість працювати в автономному режимі та надійне зберігання історії змін роблять її привабливим вибором для індивідуальних розробників і команд. Завдяки розподіленій архітектурі та легкості у використанні, Mercurial дозволяє швидко налаштовувати робочі процеси та забезпечує стабільну роботу навіть у великих проєктах.

Проте, її популярність поступається Git, який має ширшу інтеграцію з сучасними інструментами та платформами, такими як GitHub, GitLab і CI/CD сервіси. Ця різниця може стати вирішальним фактором для нових проєктів, де важливим є активне використання інструментів для співпраці, автоматизації та розгортання.

Водночас, Mercurial залишається надійним варіантом для проєктів, де критичною є швидкодія локальних операцій, збереження цілісності даних та можливість адаптувати робочі процеси до потреб команди. Завдяки своїй стабільності та зручності, вона залишається цінним інструментом для тих розробників і організацій, які шукають баланс між простотою, надійністю та функціональністю.

У підсумку, вибір між Mercurial і іншими системами контролю версій, такими як Git, залежить від специфіки проєкту, вимог команди та інтеграцій, які потрібні для ефективної роботи. Для тих, хто цінує простоту, швидкість та стабільність, Mercurial може стати ідеальним рішенням.

Git

Git був створений у 2005 році Лінусом Торвальдсом, розробником ядра операційної системи Linux. Основна мотивація створення полягала в необхідності створити власну систему контролю версій після завершення використання комерційної системи BitKeeper, яку Linux-спільнота використовувала раніше.

Лінус Торвальдс розробив Git із кількома ключовими принципами:

- *швидкість*: програма мала бути надзвичайно продуктивною, навіть для великих проєктів;
- *простота дизайну*: Git мав забезпечувати простий і зрозумілий спосіб керування кодом;
- *розподіленість*: система повинна працювати без необхідності постійного доступу до центрального сервера;
- *цілісність даних*: код і історія змін повинні бути захищені від втрат чи модифікацій.

Перша версія Git була випущена у квітні 2005 року, і з того часу він став основною системою контролю версій для більшості програмних проєктів у світі. Головна мета Git – забезпечити ефективний і надійний інструмент для керування версіями програмного забезпечення.

На рисунку 3.4 зображений інтерфейс програми GitKraken для роботи з системою контролю версій Git.

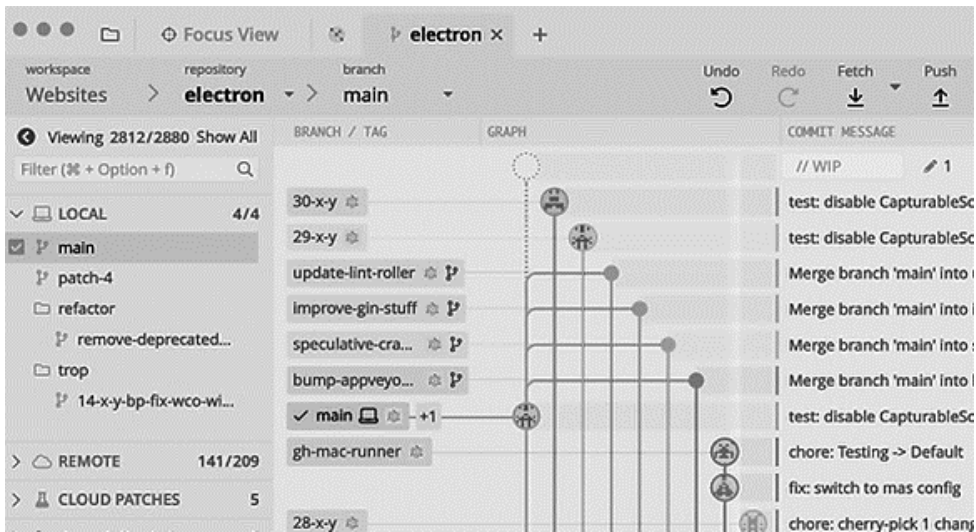


Рис. 3.4. Інтерфейс програми GitKraken для роботи з системою контролю версій Git

Ліва панель відображає список локальних та віддалених гілок репозиторію. Гілка `main` є основною та активною (виділена зеленим). Також показані інші гілки, такі як `patch-4`, `refactor`, і `remove-deprecated`.

Центральна область відображає графічне представлення дерева комітів. Лінії різних кольорів вказують на окремі гілки та їхню історію. Кружечки на лініях відображають коміти, з зазначенням опису змін у правій частині (наприклад, «Merge branch 'main' into ...»).

Верхній рядок меню показує опції управління репозиторієм, такі як «Undo», «Redo», «Fetch», «Push».

Права область містить список комітів із повідомленнями до кожного з них (наприклад, «test: disable Capturable...»).

Цей інтерфейс допомагає візуально аналізувати історію проєкту, переглядати гілки, коміти та злиття, а також зручно управляти ними без використання командного рядка.

Git розроблений як універсальний інструмент, що дозволяє працювати над будь-якими типами проєктів, від малих до великих, і забезпечує ефективність, прозорість і безпеку у процесі розробки програмного забезпечення.

Основні характеристики Git:

- *розподіленість*:
 - кожен розробник зберігає локальну копію репозиторію, яка включає всю історію змін, гілки та метадані;
 - це дозволяє працювати автономно без постійного доступу до віддаленого сервера, що є особливо корисним при слабкому або відсутньому інтернет-з'єднанні;
 - локальна копія забезпечує швидкий доступ до історії змін і дозволяє виконувати більшість операцій, таких як створення комітів чи злиття, без необхідності звертатися до сервера;
- *контроль версій*:
 - відстежує всі зміни, внесені до файлів, включаючи додавання, видалення або модифікацію;
 - дозволяє розробникам повернутися до попереднього стану репозиторію в будь-який момент, зберігаючи хронологію змін;
 - механізм створення «знімків» (snapshots) файлів забезпечує ефективне зберігання даних і дозволяє аналізувати зміни між версіями;
- *гілки та злиття*:
 - легкі гілки: створення та видалення гілок у Git є дуже швидким і займає мало ресурсів завдяки ефективній структурі даних;
 - паралельна розробка: гілки дозволяють одночасно працювати над кількома завданнями (наприклад, новими функціями, виправленнями багів або експериментами) без ризику пошкодження основного коду;
 - механізм злиття: дозволяє об'єднувати зміни з різних гілок у єдину, підтримуючи синхронізацію між учасниками команди. Git також має інструменти для автоматичного вирішення конфліктів під час злиття;
- *швидкість*:
 - більшість операцій, таких як створення комітів, перегляд історії, створення гілок і злиття, виконуються локально, що значно прискорює робочі процеси;
 - Git розроблений для обробки великих проєктів і зберігання тисяч файлів, забезпечуючи стабільну продуктивність навіть у масштабних репозиторіях;

- *цілісність даних:*
 - використовує алгоритм SHA-1 для створення хешів, що забезпечує криптографічну цілісність кожного файлу, коміту, гілки та інших даних у репозиторії;
 - будь-яка спроба змінити дані в репозиторії призводить до зміни хешу, що робить подібні дії легко помітними;
 - цілісність даних гарантує надійність історії змін і запобігає втраті або пошкодженню інформації;
- *співпраця:*
 - полегшує командну розробку, дозволяючи розробникам працювати над одним і тим самим проектом у різних гілках без конфліктів;
 - інтегрується з різними платформами для управління репозиторіями, такими як GitHub, GitLab, Bitbucket, що забезпечує додаткові інструменти для командної роботи (рев'ю коду, трекери задач тощо);
 - механізми pull, push і fetch забезпечують обмін змінами між локальними та віддаленими репозиторіями;
- *кросплатформність:*
 - Git працює на різних операційних системах, таких як Windows, macOS, Linux, та підтримує інтеграцію з популярними текстовими редакторами та IDE;
 - забезпечує однакову функціональність незалежно від платформи, що робить його універсальним для команд із різними робочими середовищами;
- *гнучкість:*
 - підтримує різні моделі робочого процесу, включаючи Git Flow, GitHub Flow, Trunk-Based Development, що дозволяє адаптувати Git до конкретних потреб проекту;
 - дозволяє створювати кастомні сценарії роботи з репозиторіями завдяки розширеним можливостям командного рядка;
- *масштабованість:*
 - оптимізований для роботи з великими проектами з тисячами файлів і розгалуженою історією змін;
 - добре підходить для як невеликих проектів, так і складних корпоративних систем із великою кількістю розробників;
- *прозорість історії:*
 - забезпечує повну прозорість історії змін із можливістю перегляду, аналізу та фільтрації комітів;

- можливість створювати анотації (git blame) для аналізу того, хто і коли вніс певні зміни до коду.

Git – це потужний інструмент для контролю версій, що забезпечує розподілену роботу, гнучкість, високу швидкість та надійність. Він ідеально підходить для командної розробки, а також для індивідуальних проєктів, дозволяючи ефективно керувати історією змін і адаптуватися до різних моделей розробки.

Основний робочий процес у Git складається з наступних етапів:

- 1) ініціалізація репозиторію:

git init

- 2) клонування репозиторію:

git clone <url>

- 3) перевірка статусу змін:

git status

- 4) додавання файлів до індексу:

git add <назва_файлу> # або git add . для всіх файлів

- 5) фіксація змін (коміт):

git commit -m "Опис змін"

- 6) відправка змін у віддалений репозиторій:

git push

- 7) отримання змін із віддаленого репозиторію:

git pull

Основні команди для роботи з гілками:

- 1) створення нової гілки:

git branch <назва_гілки>

- 2) перемикання на іншу гілку:

git checkout <назва_гілки>

- 3) об'єднання змін із гілки:

git merge <назва_гілки>

- 4) видалення гілки:

git branch -d <назва_гілки>

Таблиця 3.3 охоплює базові та просунуті команди для роботи з Git. Вони охоплюють ініціалізацію, синхронізацію, історію комітів, управління гілками та інші операції.

Таблиця 3.3. Команди Git

Команда	Призначення	Приклад
git init	Ініціалізує новий локальний репозиторій	git init
git clone <url>	Клонує існуючий репозиторій із сервера в локальний каталог	git clone https://github.com/user/repo.git
git status	Показує поточний статус репозиторію, включаючи змінені файли та файли, які ще не додані до індексації	git status
git add <file>	Додає файл до індексації (staging area) перед створенням коміту	git add file.txt
git commit -m "<msg>"	Створює коміт із доданими файлами з повідомленням, що описує зміни	git commit -m "Added new feature"
git log	Відображає історію комітів репозиторію	git log
git branch	Виводить список локальних гілок або створює нову гілку	git branch feature-branch
git checkout <branch>	Перемикається на зазначену гілку або відновлює файли з певної версії	git checkout feature-branch
git switch <branch>	Перемикається на іншу гілку (сучасна альтернатива git checkout)	git switch develop
git merge <branch>	Зливає зміни з однієї гілки в поточну	git merge feature-branch
git pull	Завантажує останні зміни з віддаленого репозиторію та автоматично зливає їх у поточну гілку	git pull origin main
git push	Завантажує локальні зміни у віддалений репозиторій	git push origin main
git remote -v	Показує список віддалених репозиторіїв та їх URL-адреси	git remote -v
git fetch	Завантажує останні зміни з віддаленого репозиторію, не зливаючи їх із локальними змінами	git fetch origin

git diff	Показує різницю між файлами в різних версіях (наприклад, робоча директорія vs staging area)	git diff
git stash	Зберігає незавершені зміни у тимчасове сховище, дозволяючи перемикатися між гілками без втрати змін	git stash
git stash apply	Відновлює зміни з тимчасового сховища (stash)	git stash apply
git reset <file>	Видаляє файл із staging area, не видаляючи його з робочої директорії	git reset file.txt
git reset --hard <commit>	Повертає репозиторій до певного коміту, видаляючи всі подальші зміни	git reset --hard HEAD~1
git revert <commit>	Створює новий коміт, що скасовує зміни, внесені в зазначеному коміті	git revert abc1234
git tag <name>	Додає тег до конкретного коміту (зазвичай для маркування релізів)	git tag v1.0.0
git rm <file>	Видаляє файл із робочої директорії та staging area	git rm file.txt
git blame <file>	Показує, хто і коли вніс зміни у кожен рядок файлу	git blame file.txt
git show <commit>	Відображає детальну інформацію про конкретний коміт	git show abc1234
git config	Налаштовує параметри Git, наприклад, ім'я користувача чи email	git config --global user.name "John"
git cherry-pick <commit>	Переносить зміни з одного коміту в іншу гілку	git cherry-pick abc1234
git clean -f	Видаляє незбережені файли або директорії з робочої директорії	git clean -f

Ця таблиця охоплює основні команди Git, які використовуються для керування версіями, гілками та співпраці в проєктах.

Mercurial і Git є популярними розподіленими системами контролю версій, які надають розробникам інструменти для ефективного управління змінами в коді. Обидві системи мають багато спільного, таких як швидкість локальних операцій, підтримка розподіленої архітектури та потужні можливості для роботи з гілками (таблиця 3.4).

Таблиця 3.4. Порівняння Mercurial і Git

Характеристика	Mercurial	Git
Популярність	Менш популярна у спільноті розробників	Широко використовується та підтримується
Простота використання	Простий і зрозумілий інтерфейс	Має складніший інтерфейс, але більше функцій
Швидкість локальних операцій	Висока	Висока
Гілкування	Інтуїтивне і легко інтегрується	Потужне, але вимагає більше налаштувань
Інтеграція	Обмежена кількість підтримуваних платформ	Інтегрується з більшістю сучасних інструментів
Документація	Добре організована, але менш поширена	Велика кількість матеріалів і навчальних курсів
Спільнота	Менша	Дуже активна і велика
Вимоги до сервера	Мінімальні	Мінімальні, але частіше залежить від інструментів
Підтримка графічних інтерфейсів	Підтримка через TortoiseHg та інші GUI	Велика кількість сторонніх GUI

Проте між ними є суттєві відмінності. Git домінує на ринку завдяки своїй популярності, широкій інтеграції з сучасними інструментами, такими як GitHub і GitLab, та активній спільноті, яка постійно розширює його можливості. Mercurial, натомість, відзначається простішим у використанні інтерфейсом, який часто вважається зручнішим для новачків, а також стабільністю при роботі з історією змін.

Хоча Git має більшу екосистему та інтеграцію, Mercurial може бути кращим вибором для тих, хто шукає простоту та надійність. Таблиця допомагає наочно порівняти основні характеристики обох систем, що дозволяє оцінити їх переваги та недоліки залежно від потреб вашого проєкту.

Переваги і обмеження розподілених систем контролю версій

Таблиця 3.5 містить порівняння основних переваг і обмежень розподілених систем контролю версій, таких як Git, у контексті їх використання

в процесах безперервної інтеграції. Вона висвітлює, як розподілені системи можуть сприяти покращенню ефективності розробки, прискоренню процесів та підвищенню автономності команди, водночас звертаючи увагу на потенційні труднощі, які можуть виникнути при використанні таких систем.

Таблиця 3.5. Переваги та недоліки систем контролю версій у контексті CI

Переваги	Опис	Недоліки	Опис
Робота в автономному режимі	Можливість роботи без підключення до сервера (локальна копія репозиторія)	Високі вимоги до дискового простору	Кожен розробник зберігає повну копію репозиторія на своєму пристрої
Швидкодія	Локальні операції (створення гілок, коміти, порівняння версій) виконуються швидше	Складність для новачків	Розподілені системи мають складніший інтерфейс та функціональність
Підтримка гілок та злиттів	Легке створення гілок для нових функцій, злиття змін без перешкод	Конфлікти при злитті змін	Потрібно вміти вирішувати конфлікти при інтеграції змін
Інтеграція з CI платформи	Легко інтегрується з платформами на кшталт GitHub, GitLab для автоматизації тестів та релізів	Потреба в навчанні та адаптації	Розробники повинні мати знання для ефективного використання всіх функцій Git

У підсумку, розподілені системи контролю версій є незамінним інструментом у сучасній розробці програмного забезпечення. Вони забезпечують гнучкість, швидкодію та надійність, що дозволяє командам будь-якого розміру ефективно працювати навіть у складних умовах. Завдяки своїй універсальності DVCS стали стандартом у більшості організацій, які прагнуть досягти високого рівня продуктивності, якості та масштабованості у своїх проєктах.

Серед головних переваг розподілених систем контролю версій можна виділити можливість роботи в автономному режимі, що є однією з найважливіших їхніх характеристик. Завдяки тому, що кожен розробник має локальну копію всього репозиторія, включаючи всю історію змін, вони можуть переглядати попередні версії файлів, створювати коміти, аналізувати зміни та навіть зливати гілки без необхідності постійного підключення до центрального

сервера. Це дозволяє розробникам продовжувати працювати продуктивно навіть у ситуаціях із ненадійним або відсутнім інтернет-з'єднанням. Така автономність є особливо корисною для віддалених працівників, які можуть знаходитися в різних часових зонах або регіонах із нестабільною інфраструктурою.

Ще однією значною перевагою розподілених систем є покращена швидкодія завдяки локальним операціям. Наприклад, операції, пов'язані з переглядом історії змін, створенням нових гілок або порівнянням версій файлів, виконуються безпосередньо на пристрої розробника, оскільки всі необхідні дані зберігаються локально. Це суттєво знижує затримки, пов'язані з мережею, та дозволяє уникнути перевантаження центрального сервера, яке може виникати в централізованих системах. Як наслідок, розробники отримують більш швидку і зручну взаємодію із системою контролю версій, що підвищує їхню продуктивність.

Завдяки своїй архітектурі розподілені системи чудово підходять для підтримки складних робочих процесів. Вони дозволяють розробникам створювати окремі гілки для нових функцій, виправлення помилок або експериментів, зберігаючи при цьому стабільність основної (головної) гілки. Це дає змогу команді працювати паралельно над різними аспектами проєкту без ризику взаємного перетину чи втрати змін. Коли розробники завершують свою роботу, зміни можуть бути легко злиті в головну гілку за допомогою вбудованих механізмів злиття (merge). Навіть у випадку конфліктів DVCS надають інструменти для їхнього швидкого і зручного вирішення.

Крім того, можливість паралельної роботи з гілками робить розподілені системи ідеальним вибором для великих команд, які працюють над масштабними проєктами. Розподілена архітектура дозволяє знижувати навантаження на центральний сервер, оскільки більшість операцій виконується локально. Це є особливо важливим у великих організаціях, де кількість розробників може сягати сотень або тисяч, а частота комітів та інтеграцій дуже висока. У таких умовах DVCS забезпечують плавність і стабільність робочих процесів, даючи змогу масштабувати розробку без значних затримок або технічних обмежень.

Ще однією важливою перевагою є те, що розподілені системи сприяють підвищеній надійності та безпеці даних. Оскільки кожен розробник має повну копію репозиторія, ризик втрати даних значно знижується. У разі збою або пошкодження центрального сервера дані можна легко відновити з копій, які зберігаються на локальних пристроях розробників. Це робить розподілені системи більш стійкими до збоїв порівняно з централізованими системами, де збої сервера можуть стати критичним фактором.

Таким чином, розподілені системи контролю версій, як-от Git і Mercurial, пропонують широкий спектр переваг, які роблять їх ідеальним вибором для сучасної розробки програмного забезпечення. Вони забезпечують автономність, швидкодію, гнучкість у створенні робочих процесів і високу надійність, що дозволяє ефективно організовувати розробку як у невеликих командах, так і в масштабних розподілених організаціях. Завдяки цим властивостям DVCS є невід'ємною частиною багатьох сучасних проєктів, сприяючи підвищенню їхньої продуктивності, якості та безперервності розробки.

Попри численні переваги, розподілені системи контролю версій мають і свої недоліки, які можуть створювати певні виклики для розробників та організацій. Одним із таких недоліків є підвищена потреба в дисковому просторі на локальних пристроях. Оскільки кожен розробник має повну копію репозиторія, включаючи всю історію змін, обсяг даних може значно зрости, особливо у великих проєктах із тривалою історією та великою кількістю файлів. Це може створювати проблеми для розробників, які використовують пристрої з обмеженим обсягом пам'яті або мають справу з репозиторіями, що займають десятки чи навіть сотні гігабайт.

Ще одним важливим аспектом є складність освоєння розподілених систем для новачків. Розвинений набір функцій і можливостей, притаманний таким системам, як Git, може бути надмірно складним для тих, хто тільки починає працювати з системами контролю версій. Наприклад, процеси злиття конфліктів, управління гілками чи взаємодія з віддаленими репозиторіями вимагають від розробника розуміння як концепцій, так і конкретних команд або сценаріїв. Для новачків це може стати бар'єром, що потребує додаткового навчання та практики, щоб повністю розкрити потенціал цих систем.

Крім того, розподілені системи можуть бути складнішими для організації та адміністрування в порівнянні з централізованими системами. Наприклад, контроль доступу до репозиторія та управління роботою над кількома гілками можуть потребувати чітких правил і політик, щоб уникнути хаосу в проєкті. Для великих організацій це може означати необхідність додаткових зусиль для встановлення стандартів роботи та налаштування інструментів інтеграції, таких як CI/CD системи, що базуються на розподілених репозиторіях.

Незважаючи на ці недоліки, розподілені системи контролю версій залишаються найкращим вибором для більшості сучасних проєктів, оскільки їхні переваги суттєво переважають недоліки. Архітектура таких систем надає командам можливість працювати автономно, забезпечує гнучкість у створенні робочих процесів і пропонує потужні інструменти для управління змінами. Наприклад, використання гілок у розподілених системах дозволяє командам працювати над різними функціональностями одночасно, не впливаючи на

стабільність основної версії проєкту. Це є особливо корисним у великих і розподілених командах, де кожен розробник або група може працювати незалежно.

Загалом, розподілені системи контролю версій стали стандартом у сучасній розробці програмного забезпечення завдяки своїй здатності підтримувати складні робочі процеси, забезпечувати високу надійність та гнучкість. Вони дозволяють ефективно організовувати роботу як у невеликих командах, так і в масштабних міжнародних організаціях, забезпечуючи продуктивність, стабільність і якість проєктів навіть у складних і динамічних умовах.

Роль систем контролю версій у неперервній інтеграції

Роль систем контролю версій у неперервній інтеграції (Continuous Integration, CI) є ключовою, оскільки вони забезпечують ефективне управління змінами коду, їх інтеграцію та перевірку. У системах CI найпопулярнішою є розподілена модель контролю версій, зокрема Git, завдяки її гнучкості, високій швидкодії та широким можливостям інтеграції з платформами на кшталт GitHub, GitLab чи Bitbucket.

Розподілені системи контролю версій дозволяють організувати процес розробки таким чином, щоб максимально полегшити впровадження практик CI. Однією з найважливіших функцій є можливість інтеграції змін у кодову базу з автоматизованою перевіркою якості через скрипти CI. Такі інструменти, як Jenkins, GitHub Actions, GitLab CI/CD або CircleCI, автоматично запускають тести, збірку, аналіз коду та інші перевірки при внесенні змін у репозиторій. Це забезпечує швидке виявлення помилок, підвищуючи загальну якість програмного забезпечення.

Використання гілок у розподілених системах дозволяє ефективно організувати робочий процес. Розробники можуть створювати окремі гілки для кожної нової функції, виправлення помилки чи експерименту. Це ізолює зміни, мінімізуючи ризики конфліктів у кодовій базі. Завдяки механізмам Git, таким як pull requests або merge requests, можна інтегрувати зміни в основну гілку лише після їх перевірки та схвалення. Це допомагає підтримувати стабільність основної версії проєкту.

Ще однією важливою перевагою є можливість швидкого вирішення конфліктів. У разі виникнення конфліктів у коді Git надає потужні інструменти для їх локалізації та усунення. Крім того, система дозволяє переглядати історію змін, аналізувати, які саме зміни спричинили конфлікт, і швидко приймати рішення щодо їх вирішення. Це дозволяє командам ефективно підтримувати

стабільність головної гілки (main або master), яка часто використовується для релізів.

Інтеграція розподілених систем із CI також сприяє автоматизації управління релізами. Наприклад, платформи на основі Git дозволяють створювати теговані релізи, що запускають автоматизовані процеси збірки, тестування та розгортання в продуктивне середовище. Це забезпечує швидший цикл випуску нових версій продукту та зменшує людський фактор у процесі розробки.

Таким чином, розподілені системи контролю версій, особливо Git, є незамінним елементом у реалізації CI. Вони забезпечують високу гнучкість, ефективне управління змінами та стабільність кодової бази, що дозволяє командам швидко інтегрувати, перевіряти та доставляти якісний програмний продукт.

3.3. Поняття Git Flow та GitHub Flow

Git Flow – це одна з популярних моделей робочих процесів у системі контролю версій Git, яка пропонує чітке визначення того, як має відбуватися робота з гілками у проєкті. Ця модель була розроблена Vincent Driessen у 2010 році і стала стандартом для багатьох команд завдяки своїй структурованості та здатності впорядковувати складні процеси розробки, тестування та релізу програмного забезпечення. Вона особливо ефективна для командної роботи над великими проєктами, які потребують частих оновлень і чіткої організації.

Головна мета Git Flow – це оптимізація та впорядкування таких етапів розробки як:

- *розробка нових функцій*: кожна функція створюється та розвивається окремо, без впливу на стабільну версію програми;
- *тестування і стабілізація*: спеціальні гілки дозволяють проводити повне тестування перед фінальним релізом;
- *реліз та підтримка*: забезпечується стабільність основної гілки, яка відповідає за продакшн-версію, та можливість швидко виправляти критичні помилки.

Основні типи Git Flow

1) Основні гілки:

- **master** (або **main**): Ця гілка містить стабільний код, готовий до випуску або вже випущений в продукцію. Всі зміни, які потрапляють до цієї гілки, повинні бути перевірені і протестовані.

Зазвичай використовують теги для позначення версій:

git tag -a v1.0 -m "Реліз версії 1.0"

- **develop**: гілка для інтеграції нових функцій, яка представляє собою активну розробку. Усі зміни, що додаються до цієї гілки, тестуються, і після стабілізації та завершення розробки нових функцій код зливаються в master для випуску.

2) Допоміжні гілки:

- **feature** (особливі гілки для розробки нових функцій): вони створюються від гілки develop і використовуються для роботи над новими можливостями або змінами. Кожна нова функція розробляється в окремій гілці feature, що дозволяє ізолювати її від основної розробки. Після завершення розробки функції, її зливають з гілкою develop.

Приклад створення та злиття:

```
git checkout -b feature/назва_функції develop
# Після завершення
git checkout develop
git merge feature/назва_функції
```

- **release** (гілки для підготовки до релізу): коли нові функції, що розробляються в гілці develop, готові до релізу, створюється гілка release. Вона служить для стабілізації коду перед випуском, включаючи виправлення багів, оновлення документації та проведення фінальних тестів. Після завершення роботи, зміни з гілки release зливаються з master і develop.

Приклад створення:

```
git checkout -b release/1.0 develop
```

- **hotfix** (гарячі виправлення): гілки для термінових виправлень у коді, які потрібно випустити без очікування стабільної версії. Вони створюються безпосередньо від гілки master, коли виникає потреба в терміновому виправленні помилки в продукції. Після виправлення багу, зміни з гілки hotfix зливаються в master (для публікації виправлення) і develop (для збереження актуальності основної гілки розробки).

Приклад створення:

```
git checkout -b hotfix/critical_bug main
```

Алгоритм роботи з даними у моделі Git Flow є наступним:

- *створення нової гілки feature*:
 - коли розпочинається робота над новою функцією, створюється окрема гілка feature на основі гілки develop;

- назва гілки повинна бути описовою, щоб зрозуміло, над якою функцією ведеться робота (наприклад, feature/login-system)
git checkout develop
git checkout -b feature/<назва_функції>
- усі зміни, що стосуються цієї функції, зберігаються у гілці feature.
- *злиття гілки feature із develop:*
 - після завершення роботи над функцією і перевірки її працездатності, гілка feature зливається назад у develop;
 - перед злиттям бажано оновити гілку feature, синхронізувавши її з актуальним станом develop:
git checkout feature/<назва_функції>
git pull origin develop
 - після цього виконується злиття:
git checkout develop
git merge feature/<назва_функції>
 - якщо гілка більше не потрібна, її можна видалити:
git branch -d feature/<назва_функції>
- *створення гілки release для стабілізації:*
 - коли в гілці develop є готовий набір функцій, створюється гілка release для підготовки релізу;
 - у цій гілці відбувається фінальне тестування, дрібні виправлення та підготовка до релізу (наприклад, оновлення документації, конфігурацій тощо):
git checkout develop
git checkout -b release/<номер_релізу>
 - усі виправлення додаються в гілці release;
- *злиття гілки release у master та develop:*
 - після завершення роботи над стабілізацією коду гілка release зливається в master для випуску нової версії:
git checkout master
git merge release/<номер_релізу>
 - для збереження хронології розробки ці ж зміни інтегруються в develop:
git checkout develop
git merge release/<номер_релізу>
 - гілка release більше не потрібна і може бути видалена:
git branch -d release/<номер_релізу>
 - позначається нова версія у master:
git tag -a v<номер_версії> -m "Опис релізу"

- *робота з гілками hotfix у разі критичних помилок:*
 - якщо виявлено критичну помилку у продукційній версії, створюється гілка hotfix від master для оперативного виправлення.
git checkout master
git checkout -b hotfix/<назва_виправлення>
 - після внесення змін гілка hotfix зливається в master для негайного оновлення продукційної версії:
git checkout master
git merge hotfix/<назва_виправлення>
 - зміни також інтегруються у develop, щоб вони не були втрачені в наступних релізах:
git checkout develop
git merge hotfix/<назва_виправлення>
 - гілка hotfix видаляється після завершення роботи:
git branch -d hotfix/<назва_виправлення>

На рисунку 3.5 зображена структура репозиторію Git із гілками master, develop і feature. Це графічне представлення історії комітів у різних гілках, що ілюструє робочий процес розробки.

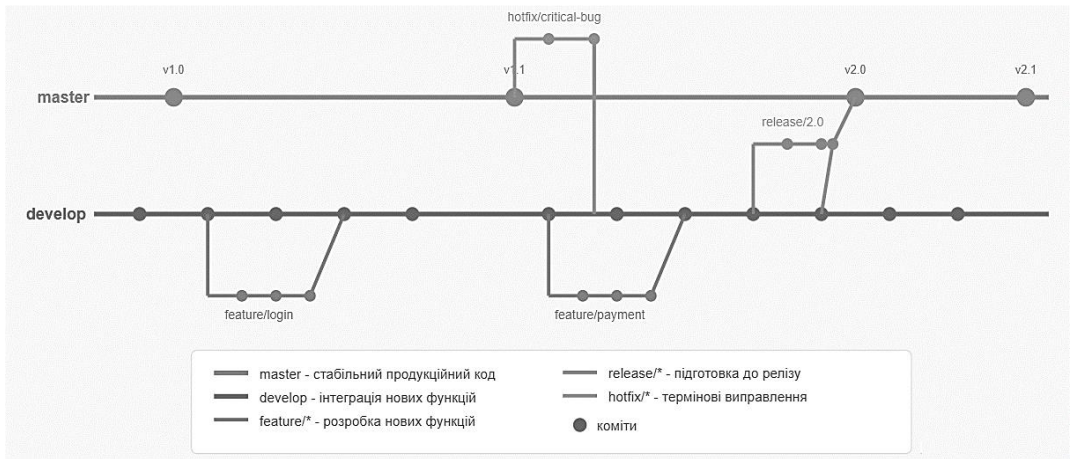


Рис. 3.5. Структура репозиторію Git

Така організація дозволяє зберігати стабільність основної гілки, а також забезпечує зручність для паралельної роботи над кількома завданнями.

Ця схема відповідає популярній моделі Git Flow, яка використовується для організації процесу розробки в команді.

Переваги Git Flow:

- *чітка організація процесів:*
 - забезпечує логічну структуру для роботи над проектом, яка легко зрозуміла навіть для нових учасників команди;
 - допомагає підтримувати порядок у репозиторії, особливо у великих проектах із багатьма учасниками;
 - полегшує планування релізів і відстеження прогресу завдяки чітко визначеним гілкам для кожного етапу роботи;
- *ізоляція функцій та виправлень:*
 - дозволяє розробляти кожну функцію чи виправлення окремо, зменшуючи ймовірність неочікуваних конфліктів або помилок;
 - забезпечує можливість тестувати функції в ізоляції перед інтеграцією в основний код;
 - полегшує відкат (revert) у разі, якщо певна функція чи зміна викликає проблеми, оскільки всі зміни зберігаються в окремих гілках;
- *паралельна робота:*
 - дозволяє одночасно працювати над декількома функціями чи виправленнями, оскільки кожен розробник або команда може використовувати власну гілку;
 - підвищує ефективність роботи, особливо у великих командах, де одночасно вирішуються різні завдання;
 - забезпечує стабільність основного коду під час активної розробки;
- *підготовка до релізу:*
 - гілка release дозволяє проводити фінальне тестування, стабілізацію та підготовку коду до випуску, не перешкоджаючи подальшій розробці в гілці develop;
 - спрощує оновлення документації та внесення змін до конфігурації перед випуском;
- *безпека продукційного коду:*
 - гілка master завжди містить лише стабільні версії, готові до використання в продакшн-середовищі;
 - у разі критичних помилок модель дозволяє швидко створити гілку hotfix для виправлення проблеми без впливу на розробку нових функцій;
- *легкість у відстеженні версій:*
 - використання тегів у гілці master дозволяє чітко позначати релізи, що спрощує управління версіями програмного забезпечення;

- історія розробки залишається прозорою, що корисно для аналізу роботи або вирішення проблем;
- *зручність у масштабованих проєктах:*
 - добре підходить для великих команд і розподіленої роботи, оскільки кожен розробник або підрозділ може працювати у своїй гілці, синхронізуючи результати з основними гілками за потреби;
 - дозволяє одночасно підтримувати поточну версію продукту, розробляти нові функції та готувати наступні релізи;
- *підтримка на всіх етапах розробки:*
 - від розробки функцій до стабілізації та випуску продукту – Git Flow покриває всі аспекти життєвого циклу розробки;
 - гілки hotfix дають можливість оперативно реагувати на критичні проблеми в продукційній версії без порушення основного процесу розробки;
- *інтеграція з CI/CD системами:* хоча Git Flow не завжди оптимальний для CI/CD, він добре інтегрується зі сценаріями, де автоматичне тестування й деплой виконуються для конкретних гілок (release або hotfix), забезпечуючи додаткову перевірку перед злиттям у master;
- *документованість процесу:* завдяки чітко визначеним гілкам та їх призначенню вся історія роботи над проєктом залишається зрозумілою, що полегшує аналіз та внесення змін у майбутньому.

Git Flow має складну структуру з великою кількістю гілок, що може бути важким для новачків і малих команд. Злиття гілок часто супроводжується конфліктами, що потребує часу і досвіду. Також новим учасникам потрібно більше часу для освоєння принципів Git Flow, що уповільнює розробку;

Таким чином, Git Flow – це потужна, але складна модель, яка підходить для проєктів середнього та великого розміру з довготривалим циклом розробки. Для менших проєктів або команд із динамічним стилем роботи вона може бути занадто складною і неефективною. Перед впровадженням Git Flow варто оцінити потреби проєкту та команди, щоб уникнути зайвих витрат часу та ресурсів.

GitHub Flow

GitHub Flow – це спрощена модель розгалуження Git, розроблена компанією GitHub для оптимізації процесу розробки програмного забезпечення. На відміну від більш складних моделей як Git Flow, GitHub Flow базується на простому принципі: основна гілка (main/master) завжди готова до продакшену, а всі нові функції розробляються в окремих гілках, які створюються від основної. Ця модель особливо підходить для команд, що практикують безперервну

інтеграцію та розгортання, оскільки дозволяє швидко та безпечно інтегрувати зміни.

Робочий процес GitHub Flow складається з кількох простих кроків. Спочатку розробник створює нову гілку від main для роботи над конкретною функцією або виправленням помилки. Після завершення роботи він створює Pull Request (PR), що дозволяє іншим членам команди переглянути код, залишити коментарі та провести обговорення змін. Після схвалення PR зміни зливаються в основну гілку, а робоча гілка видаляється (рис. 3.6). Цей підхід забезпечує прозорість процесу розробки та якість коду через обов'язкове рецензування.

Основними перевагами GitHub Flow є простота використання, швидкість інтеграції змін та відсутність складних правил управління гілками. Модель ідеально підходить для веб-додатків та сервісів, де можливе часте розгортання, проте може бути менш придатною для проектів з фіксованими релізами або складними циклами тестування. GitHub Flow сприяє культурі співпраці в команді та дозволяє підтримувати стабільність основної гілки, що є критично важливим для безперервної доставки програмного забезпечення.

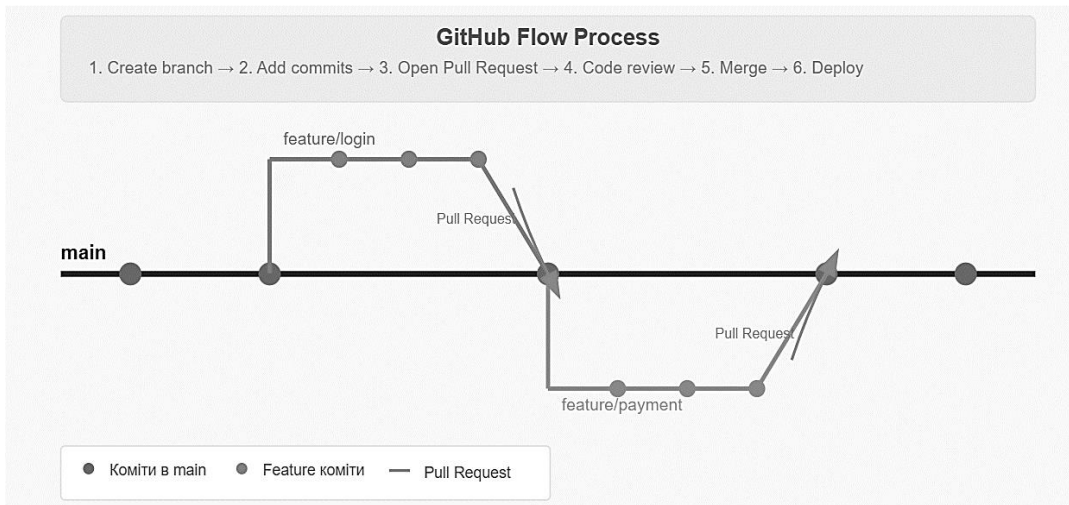


Рис. 3.6. Схема роботи з даними у моделі GitHub Flow

Важливою особливістю GitHub Flow є її адаптивність до різних розмірів команд та проектів. Невеликі команди можуть працювати з мінімальною кількістю правил, тоді як великі організації можуть додавати додаткові перевірки, автоматизоване тестування та політики захисту гілок. Модель також добре інтегрується з сучасними практиками DevOps, включаючи автоматизоване тестування, continuous integration/continuous deployment (CI/CD) пайплайни та моніторинг після розгортання. Це робить GitHub Flow не просто

інструментом для управління кодом, а частиною цілісної екосистеми розробки програмного забезпечення.

Контрольні питання до розділу 3

- 1) Що забезпечує централізація змін у системі контролю версій?
- 2) Що, на Вашу думку, можна вважати головним недоліком централізованих систем контролю версій?
- 3) В чому полягає перевага розподілених систем контролю версій над централізованими?
- 4) У яких випадках варто використовувати централізовані системи, незважаючи на їх обмеження?
- 5) Яке призначення має термін "commit" при роботі з Git?
- 6) Яким чином Git забезпечує швидкодію при роботі з великими проєктами?
- 7) Яку роль відіграє система контролю версій у процесі неперервної інтеграції??
- 8) Яке призначення команди git status?
- 9) Що таке Git Flow, і яку роль відіграють його основні гілки?
- 10) Яким чином система контролю версій підтримує автоматизацію перевірок?
- 11) Що є однією з ключових функцій будь-якої системи контролю версій?
 - a) Резервне копіювання файлів
 - b) Централізація всіх змін в проєкті
 - c) Впровадження нових мов програмування
 - d) Підтримка клієнтів
 - e) Забезпечення доступу до мережі Інтернет
- 12) Що з перерахованого не належить до функцій системи контролю версій у процесі неперервної інтеграції?
 - a) Централізація змін
 - b) Відстеження історії змін
 - c) Автоматизація перевірок
 - d) Компіляція програм
 - e) Підтримка спільної роботи
- 13) Який тип архітектури використовує Subversion (SVN)?
 - a) Розподілена
 - b) Централізована
 - c) Гібридна
 - d) Кластерна

- е) Паралельна
- 14) Яка основна перевага розподілених систем контролю версій?
 - а) Простота адміністрування
 - б) Менші вимоги до дискового простору
 - с) Можливість автономної роботи
 - д) Централізоване керування
 - е) Швидший доступ до сервера
- 15) Яка перерахованих систем контролю версій має найбільшу інтеграцію з сучасними інструментами DevOps?
 - а) Subversion (SVN)
 - б) Mercurial
 - с) Git
 - д) Perforce
 - е) BitKeeper
- 16) Для чого використовується команда `git clone` при роботі з даними засобами Git?
 - а) Видалення існуючого репозиторію
 - б) Сортуння даних у репозиторії
 - с) Завантаження даних з вказаного репозиторію до нового
 - д) Архівування репозиторію
 - е) Оновлення існуючого репозиторію
- 17) Яка команда використовується в Git для завантаження змін з віддаленого репозиторію?
 - а) `git push`
 - б) `git commit`
 - с) `git branch`
 - д) `git merge`
 - е) `git pull`
- 18) З якою метою використовується команда `git stash`?
 - а) Для видалення файлів
 - б) Для створення нової гілки
 - с) Для злиття гілок
 - д) Для тимчасового збереження незавершених змін
 - е) Для створення тегів
- 19) Який алгоритм використовується в Git для забезпечення цілісності даних?
 - а) MD5
 - б) SHA-1
 - с) AES

- d) RSA
- e) SHA -1

20) Яка основна мета моделі Git Flow?

- a) Забезпечення централізованого контролю.
- b) Оптимізація процесів CI/CD.
- c) Збереження історії змін у хмарі.
- d) Організація зручної роботи з гілками.
- e) Використання графічного інтерфейсу для керування репозиторієм.

21) Встановіть відповідність між системами контролю версій та сферами використання:

A. Git	1) Підходить для великих проєктів із мультимедійними файлами
B. Perforce	2) Використовується в документації та централізованих проєктах
C. Subversion (SVN)	3) Основний вибір для роботи з GitHub чи GitLab
D. Mercurial	4) Підходить для індивідуальних розробників та команд із потребою в автономності

22) Встановіть відповідність між системами контролю версій та їхніми особливостями:

A. Git	1) Легка у вивченні, але не забезпечена інтеграція із сучасними платформами.
B. Subversion (SVN)	2) Висока продуктивність із великими командами та мультимедійними файлами.
C. Perforce	3) Робота в автономному режимі та гнучкість у створенні гілок.
D. Mercurial	4) Централізована модель із простим адмініструванням.

23) Встановіть відповідність між командами Git та їхніми функціями (призначенням):

A. git init	1) Створення нового локального репозиторію.
B. git clone	2) Об'єднання змін із різних гілок.
C. git commit	3) Локальне збереження даних з повідомленням про зміни.
D. git merge	4) Клонування репозиторію з віддаленого сервера.

24) Встановіть відповідність між термінами та їхніми відповідностями в системі Git:

A. Розподіленість	1) Використання алгоритму SHA-1 для перевірки хешів
B. Гілки	2) Виконання більшості операцій локально
C. Цілісність даних	3) Локальна копія репозиторію на пристрої кожного розробника
D. Швидкість	4) Легке створення й злиття для паралельної розробки

25) Встановіть відповідність між типами гілок у моделі Git Flow та їх призначенням:

A. master	1) Розробка нових функцій
B. develop	2) Стабільний код для випуску
C. feature	3) Робота над основним кодом проєкту
D. hotfix	4) Термінові виправлення

РОЗДІЛ 4. ПЛАТФОРМИ ДЛЯ РОБОТИ З СИСТЕМАМИ КОНТРОЛЮ ВЕРСІЙ

4.1. GitLab

GitLab – це сучасна платформа для управління життєвим циклом розробки програмного забезпечення, створена з урахуванням потреб DevOps-підходу. Вона забезпечує централізоване середовище, де всі учасники процесу розробки – від розробників і тестувальників до DevOps-інженерів і менеджерів – можуть ефективно співпрацювати. Платформа надає повний набір інструментів, що дозволяють автоматизувати, контролювати та вдосконалювати всі етапи розробки: від створення коду до його тестування, релізу та підтримки в продакшені.

GitLab заснували Дмитро Запорожець та Сід Сібранджі в 2011 році. Ідея виникла через потребу в інструменті, який дозволяв би організовувати ефективну командну роботу над кодом із застосуванням системи контролю версій Git.

У 2011 році Дмитро Запорожець, український розробник, створив першу версію GitLab як проєкт з відкритим кодом для особистих потреб. Метою було спростити управління репозиторіями коду та взаємодію між членами команди. У 2013 році до проєкту приєднався Сід Сібранджі, який допоміг комерціалізувати GitLab. Вони заснували компанію, щоб розвивати інструмент як рішення для командної роботи. У 2015 році GitLab став платформою з інтегрованими функціями CI/CD.

У 2018 році компанія додала можливості DevSecOps, зосередившись на забезпеченні безпеки на всіх етапах розробки. З 2021 року GitLab став публічною компанією, вийшовши на біржу NASDAQ під тікером GTLB.

GitLab виділяється своєю філософією «все в одному»: замість використання багатьох різних інструментів для кожного етапу розробки, компанії отримують єдину інтегровану платформу, яка:

- спрощує управління вихідним кодом за допомогою вбудованої підтримки Git;
- забезпечує автоматизацію процесів розробки та розгортання через вбудовані пайплайни CI/CD;
- пропонує засоби для планування завдань, відстеження прогресу, управління пріоритетами та організації командної роботи;
- інтегрує інструменти безпеки для виявлення вразливостей ще до розгортання продукту;
- підтримує функції моніторингу, які дозволяють аналізувати продуктивність додатків у реальному часі.

GitLab також забезпечує масштабованість і гнучкість, що робить його придатним для використання як невеликими командами стартапів, так і великими корпораціями. Крім того, платформа може бути розгорнута як у вигляді хмарного сервісу (GitLab SaaS), так і в локальній інфраструктурі організацій (GitLab Self-Managed), що забезпечує високий рівень адаптації до різних потреб бізнесу.

Оскільки GitLab охоплює всі аспекти життєвого циклу розробки, він не лише спрощує робочі процеси, але й сприяє ефективнішій командній роботі, зменшує затрати часу на інтеграцію інструментів і допомагає досягати кращих результатів у коротші терміни.

Основні функції GitLab є наступними:

- *управління вихідним кодом*: GitLab пропонує повноцінну підтримку системи контролю версій Git, що дозволяє розробникам створювати, зберігати та керувати репозиторіями коду; використовувати розгалуження (branches) для ізоляції змін перед інтеграцією в основний код; об'єднувати зміни за допомогою Merge Requests та використовувати вбудовані інструменти для аналізу коду та перевірки якості перед інтеграцією;
- *надання різнорівневого доступу* на основі ролей (наприклад, розробник, менеджер, гість) для контролю рівня доступу;
- *безперервна інтеграція та доставка*: GitLab має вбудовані можливості для безперервної інтеграції та доставки (CI/CD), які дозволяють автоматично запускати тестування коду після кожного коміту, автоматизувати процеси збірки, тестування та розгортання додатків;
- *гарантія безпеки на всіх етапах DevOps-процесів*: надає інструменти безпеки та підтримує DevSecOps-підхід, що включає в себе: аналіз вихідного коду для виявлення вразливостей (SAST - Static Application Security Testing), перевірку під час виконання програми для виявлення потенційних ризиків (DAST - Dynamic Application Security Testing), аналіз залежностей для пошуку вразливих версій (Dependency Scanning), перевірку контейнерних образів на вразливості (Container Scanning) та надання зведеної інформації про всі знайдені проблеми безпеки (Security Dashboard);
- *інструменти планування*: GitLab надає можливості для управління завданнями та проблеми, які можна відстежувати, групувати за тегами та організовувати проектами (Issues), інструменти для управління великими задачами, які охоплюють кілька Issues (Epics), візуалізацію дорожньої карти проекту, що допомагає визначати пріоритети та

терміни виконання (Roadmaps) та канбан-дошки для організації taskів, що дозволяє командам легко управляти прогресом виконання (Boards);

- *моніторинг та аналітика*: GitLab допомагає командам контролювати стан розробки та розгорнутих програм, дає змогу аналізу логів та інтеграції з інструментами типу Prometheus для збору метрик, надає звіти про швидкість виконання пайплайнів, частоту успішних/невдалих збірок тощо;
- *інтеграція*: GitLab легко інтегрується з широким спектром зовнішніх інструментів для керування проєктами (Jira, Trello, Asana), контейнеризації (Docker, Kubernetes), моніторингу (Prometheus, Grafana), використання хмарних сервісів (AWS, Google Cloud, Microsoft Azure) та управління кодом (GitHub чи Bitbucket для міграції або інтеграції репозиторіїв).

Цей набір функцій робить GitLab універсальною платформою для командної розробки програмного забезпечення, що відповідає сучасним вимогам DevOps та DevSecOps.

Таким чином, GitLab є ефективним інструментом для організації командної роботи, надаючи потужні засоби співпраці, що включають в себе:

- *обговорення коду*: інтегровані рев'ю-коди дозволяють учасникам команди залишати коментарі, пропозиції або зауваження до змін;
- *зручний доступ*: можливість надання доступу різним членам команди з різними рівнями прав, що забезпечує безпеку та організованість;
- *прозорість*: усі зміни, запити на злиття (Merge Requests) та обговорення стають частиною історії проєкту, що сприяє кращому розумінню прогресу.

GitLab значно спрощує та прискорює цикл розробки завдяки інтегрованим інструментам безперервної інтеграції та доставки, оскільки забезпечує:

- *швидше виявлення помилок*: автоматизоване тестування після кожного коміту знижує ризик накопичення помилок;
- *гнучкість налаштувань*: пайплайни CI/CD можна адаптувати для різних середовищ – від простих веб-додатків до складних мікросервісних архітектур;
- *релізи без зупинок*: завдяки автоматичному розгортанню з мінімальним втручанням забезпечується висока доступність додатків.

GitLab пропонує інтеграцію безпеки безпосередньо в процесі розробки:

- *превентивний підхід*: інструменти, такі як статичний аналіз коду (SAST) та динамічне тестування (DAST), дозволяють виявляти вразливості до їх виходу в продакшн;
- *інтеграція з DevSecOps*: забезпечує активну співпрацю між розробниками та командами безпеки для швидкого усунення проблем;
- *аналіз залежностей*: автоматичне сканування сторонніх бібліотек та компонентів на наявність вразливих версій.

GitLab дозволяє об'єднати всі етапи розробки програмного забезпечення в одному місці та забезпечує надійне управління вихідним кодом завдяки можливості переглядати, відслідковувати та повертати попередні версії коду; використанню Git для спрощення роботи з репозиторіями. GitLab може використовуватися як у малих стартапах, так і у великих корпораціях завдяки своїй здатності адаптуватися до різного масштабу проєктів:

- *малі команди*: простота використання та швидкий старт роблять GitLab зручним вибором для стартапів;
- *великі організації*: підтримка розподілених команд, інтеграція з іншими корпоративними системами (Jira, Slack, Prometheus) та масштабованість дозволяють обслуговувати великі та складні проєкти;
- *інфраструктурна гнучкість*: можливість розгортання як у хмарі, так і на локальних серверах забезпечує високу адаптивність до вимог бізнесу.

На даний момент GitLab став універсальним інструментом, який одночасно спрощує процеси розробки, підвищує їхню якість і забезпечує інтегровану безпеку, роблячи його оптимальним вибором для багатьох компаній у світі.

Налаштування системи GitLab

GitLab може бути встановлений на локальних серверах або у хмарі.

Основні способи встановлення:

- *Omnibus-пакет*: універсальне рішення, що включає все необхідне;
- *Docker*: для контейнеризації;
- *Kubernetes*: для оркестрації контейнерів;
- *Manual Installation*: встановлення компонентів вручну.

Процес встановлення (Omnibus) може містити наступні етапи:

1. додати сховище GitLab:
`curl https://packages.gitlab.com/install/repositories/gitlab/gitlab-ee/script.deb.sh | sudo bash`
2. встановити GitLab:

```
sudo EXTERNAL_URL="http://gitlab.example.com" apt-get install gitlab-ee
```

3. налаштувати конфігурацію згідно потреб: потрібно відредагувати файл `/etc/gitlab/gitlab.rb`. та налаштувати URL, порти, SSL.
4. запустити інструмент GitLab:
`sudo gitlab-ctl reconfigure`

Після встановлення потрібно перейти за URL (наприклад, `http://gitlab.example.com`) та увійти в систему під обліковим записом адміністратора (створений автоматично, пароль знаходиться у файлі `/etc/gitlab/initial_root_password`). Наступним кроком є налаштування глобальних параметрів (наприклад, SMTP для відправки листів, інтеграції тощо).

Для керування користувачами потрібно створити їх через веб-інтерфейс або API та обрати ролі користувачів з наступних категорій Owner, Maintainer, Developer, Reporter, Guest. Для зручності керування проекти та користувачі об'єднуються в групи.

Основні інструменти GitLab

Репозиторії Git

GitLab використовує Git як основну систему контролю версій, забезпечуючи розподілену роботу з кодом. Це дозволяє кожному розробнику мати локальну копію всієї історії проекту. Користувачі можуть переглядати, редагувати та аналізувати код безпосередньо через веб-інтерфейс GitLab.

GitLab дозволяє користувачам створювати необмежену кількість гілок для розробки нових функцій, виправлення помилок або тестування. Адміністратори можуть захистити гілки, обмеживши доступ до змін (наприклад, тільки для злиття через merge requests).

За виконання запитів на злиття відповідає інструмент *Merge Requests*, що має також вбудовані інструменти для аналізу змін, порівняння різниць між гілками та додавання коментарів. Для зручності є змога автоматизувати процеси завдяки тригерам, які виконуються перед злиттям, наприклад, запуск CI/CD пайплайнів або тестів.

Вартими уваги є також наступні можливості:

- *тегування для позначення версій*: GitLab дозволяє додавати теги до певних комітів, що зручно для управління версіями програмного забезпечення;
- *управління файлами*: веб-інтерфейс дозволяє переглядати зміни у файлах з повною історією комітів та дозволяє знаходити потрібні файли чи строки коду в межах репозиторію.

- *контроль доступу*: GitLab підтримує гнучкі ролі (Owner, Maintainer, Developer, Reporter, Guest), що визначають права на читання, запис, створення гілок і злиття, розробники можуть працювати з репозиторіями через безпечні канали.
- *веб-хуки*: репозиторії GitLab підтримують налаштування веб-хуків для автоматизації дій (наприклад, надсилання повідомлень у Slack після кожного коміту);
- *інтеграція зі сторонніми інструментами*: наприклад, можна інтегрувати GitLab з Jira для відстеження завдань або з Jenkins для CI/CD.

Інструменти управління репозиторіями в GitLab пропонують низку переваг, які роблять їх зручними для командної роботи. Вони забезпечують централізацію коду, дозволяючи зберігати всі репозиторії проєкту на одній платформі, що спрощує співпрацю великих команд. Прозорість процесу розробки досягається завдяки доступу до історії комітів, відстеженню змін у коді та використанню merge requests, що створює чітке уявлення про всі етапи роботи.

Платформа забезпечує високий рівень безпеки завдяки захищеним гілкам, контролю доступу та можливостям аудиту дій, що допомагає знизити ризики небажаних змін у коді. Вона також легко інтегрується з CI/CD процесами та іншими інструментами, роблячи її універсальним рішенням для автоматизації. Крім того, GitLab підтримує ефективну роботу з великими репозиторіями та командами, що робить його придатним для масштабних проєктів. Завдяки гнучким налаштуванням, GitLab дозволяє адаптувати репозиторії під специфіку кожного проєкту.

Code Review

Code Review (перевірка коду) є ключовим етапом у розробці програмного забезпечення, метою якого є підвищення якості коду, забезпечення відповідності стандартам та виявлення помилок на ранніх етапах. У GitLab цей процес інтегрований у робочий цикл через *Merge Requests* (центральний інструмент перевірки коду) та супутні інструменти. Merge Requests дозволяють команді обговорювати та оцінювати зміни до коду перед їхнім злиттям у основну гілку.

GitLab пропонує комплексний набір інструментів для проведення перевірки коду, що включає можливості

- *локалізованого коментування коду*,
- *автоматичні перевірки через CI/CD та лінери*,
- *створення тимчасових середовищ (Review Apps) для тестування змін*.

Система підтримує гнучкі правила затвердження змін, захист важливих гілок та інтеграцію з іншими інструментами розробки, такими як системи відстеження задач та комунікаційні платформи. GitLab також надає потужні можливості для візуалізації змін через Code Diff Viewer та генерації статистики щодо процесу перевірки коду, що допомагає покращувати ефективність розробки.

Code Review у GitLab має кілька ключових переваг, що підвищують ефективність розробки. Регулярний аналіз коду допомагає виявляти помилки та забезпечувати відповідність стандартам, що підвищує якість кінцевого продукту. Прозорість процесу підтримується завдяки збереженню всіх обговорень і затверджень, що створює детальний журнал змін.

Інтеграція з CI/CD дозволяє автоматично перевіряти код, що зменшує навантаження на рецензентів і забезпечує швидший процес перевірки. Завдяки можливості поєднання з Issue Tracker, CI/CD та Review Apps, GitLab створює єдиний потік роботи для команди. Платформа також підтримує складні сценарії затвердження для великих команд, що дає гнучкість в організації процесу рецензування.

Code Review у GitLab має кілька обмежень, з якими слід рахуватися:

- крива навчання може бути довгою для новачків, оскільки для швидкого освоєння всього функціоналу потрібно витратити час.
- у безкоштовному плані доступні не всі функції, наприклад, додаткові правила затвердження, які є лише в платних версіях.
- великі Merge Requests можуть бути складними для огляду, що підвищує ризик пропуску помилок через обсяг коду.

Code Review у GitLab активно використовується в різних сферах. Для малих та середніх команд це зручний інструмент для організації простого й ефективного процесу перевірки коду. Великі корпоративні проєкти отримують переваги від розширених можливостей затвердження та автоматизації, що полегшує масштабування процесу. Крім того, GitLab є чудовим рішенням для проєктів з відкритим кодом, забезпечуючи централізований контроль якості та підтримку спільноти розробників.

Issue Tracking

Issue Tracking – це система управління завданнями та відстеження проблем, інтегрована в GitLab. Вона дозволяє створювати, організовувати та контролювати роботу над завданнями всередині проєкту. Цей інструмент є важливим компонентом Agile-розробки, забезпечуючи прозорість і контроль на кожному етапі розробки.

Основні можливості Issue Tracking у GitLab наведено нижче.

- *Створення завдань (Issues)*: розробники можуть створювати завдання для багів, нових функціоналів або технічного боргу; яким можна встановити певні пріоритети важливості (низький, середній, високий, критичний) та додати мітки;
- *Використання шаблонів завдань*: використання попередньо налаштованих шаблонів для створення завдань дає змогу спростити роботу. Наприклад, шаблони для багів можуть включати поля для опису, кроків відтворення та очікуваного результату; адміністратори можуть створювати власні шаблони, адаптовані під потреби команди.
- *Робота з панелями управління (Boards)*: використання канбан-дошок для візуалізації завдань у вигляді колонок та налаштування відображення завдань за тегами, виконавцями, пріоритетами або іншими параметрами.
- *Пріоритизація завдань*: можливість групування завдань у часові рамки або спринти для досягнення певної мети та сортування завдання за рівнем важливості, щоб зосередитися на критичних питаннях.
- *Призначення відповідальних (Assignees)*: можливість призначення завдання конкретному розробнику або команді; використання вбудованих чеклістів для детального розподілу підзадач.
- *Інтеграція з кодом*: завдання можуть бути автоматично закриті після виконання пов'язаних Merge Requests.
- *Сповіщення та обговорення*: додавання коментарів для обговорення завдання, прикріплення файлів, скріншотів або посилань; автоматичні сповіщення електронною поштою або через інтегровані сервіси, як-от Slack.
- *Аналітика та звіти*: відображення кількості відкритих і закритих завдань у Milestone та вбудовані звіти про ефективність команди, середній час вирішення завдань, кількість багів тощо;
- *Інтеграція з іншими інструментами*: GitLab Issue Tracker може інтегруватися з Jira, Trello або іншими платформами.

Issue Tracking у GitLab має кілька значних переваг, які сприяють ефективності робочого процесу. Інтеграція з кодом, CI/CD і тестуванням створює єдину екосистему, що забезпечує прозорість на всіх етапах. Система є гнучкою, пропонуючи налаштовувані теги, шаблони та правила, які можна адаптувати під потреби різних проєктів.

Для покращення прозорості використовуються канбан-дошки та звіти, що дозволяють візуалізувати прогрес виконання завдань. Автоматизація процесу здійснюється за допомогою налаштування правил переходу між статусами та інтеграції з кодовими змінами. Платформа також підтримує

командну роботу завдяки інструментам для обговорень, коментарів і сповіщень, що забезпечує ефективну співпрацю в команді.

Issue Tracking у GitLab має певні обмеження. Для налаштування певних функцій може знадобитися додатковий досвід або конфігурація. У безкоштовному плані доступні не всі можливості, і деякі розширені функції можна використовувати лише в преміум-планах. Крім того, для дуже великих команд або проєктів GitLab може бути менш масштабованим, і в таких випадках часто використовуються інші спеціалізовані системи, як-от Jira.

Таким чином, Issue Tracking у GitLab є потужним і зручним інструментом для управління завданнями на різних етапах розробки.

Security & Compliance

Security & Compliance – це набір інструментів та функцій GitLab, спрямованих на забезпечення безпеки коду, управління ризиками та відповідності нормативним стандартам. Вони інтегровані в DevSecOps-підхід, що дозволяє командам забезпечувати безпеку на кожному етапі життєвого циклу розробки (SDLC).

Основні можливості Security & Compliance у GitLab є наступними:

- *інтеграція безпеки в CI/CD;*
- *управління залежностями (Dependency Scanning);*
- *управління секретами (Secret Detection);*
- *політики відповідності (Compliance Management);*
- *ланцюжки постачання (Supply Chain Security);*
- *моніторинг та сповіщення.*

Security & Compliance у GitLab має кілька важливих переваг. Безпека інтегрована в DevOps процеси, що дозволяє виявляти вразливості ще на етапі розробки, забезпечуючи більш високий рівень захищеності на всіх етапах життєвого циклу. Вбудовані інструменти для аналізу коду, залежностей і конфігурацій значно скорочують час на виявлення проблем, автоматизуючи цей процес.

Звіти з безпеки інтегровані у GitLab UI, що забезпечує легкий доступ до них для всіх членів команди. Крім того, GitLab підтримує широку інтеграцію з іншими системами безпеки та аналітики, що дозволяє створювати комплексну інфраструктуру для забезпечення безпеки даних.

Security & Compliance у GitLab має кілька обмежень, з якими варто рахуватися. Деякі функції безпеки доступні лише в платних версіях, що обмежує можливості користувачів без преміум-плану. Крім того, для оптимальної роботи в конкретному середовищі необхідне налаштування інструментів, що може бути складним для деяких користувачів. Враховуючи

складність інструментів, для ефективного використання потрібні знання в області DevSecOps і безпеки.

GitLab Security & Compliance активно використовується в різних сферах. У фінансовому секторі GitLab допомагає забезпечити безпеку даних та відповідність нормативним вимогам, таким як PCI DSS. У медичній галузі він використовується для контролю за відповідністю HIPAA та захисту медичних даних. Для великих підприємств GitLab інтегрується в DevOps-процеси для забезпечення безпеки на рівні підприємства. Для стартапів GitLab є економічним рішенням для забезпечення базової безпеки програмного забезпечення з мінімальними витратами.

Security & Compliance у GitLab є ключовим інструментом для інтеграції безпеки у DevOps-процеси. Від статичного аналізу коду до управління залежностями та моніторингу відповідності, цей набір функцій забезпечує повний контроль за безпекою на кожному етапі розробки. Завдяки своїй гнучкості та автоматизації GitLab допомагає командам розробляти безпечно програмне забезпечення швидше й ефективніше.

Wiki

Wiki в GitLab – являє собою інтегрований інструмент для створення та зберігання документації, пов'язаної з проектом. Він дозволяє командам організовувати знання, ділитися ними та підтримувати актуальність інформації в одному централізованому місці. GitLab Wiki базується на системі контролю версій Git, що забезпечує можливість відстеження змін і повернення до попередніх версій.

Нижче наведено основні можливості GitLab Wiki:

- *створення та редагування сторінок*: зручний редактор з підтримкою Markdown і Rich Text, можливість попереднього перегляду сторінок перед збереженням, щоб уникнути помилок форматування, та використання ієрархічної організації для зручного доступу до інформації.
- *контроль версій*: всі зміни в Wiki зберігаються у вигляді комітів, що дозволяє переглядати історію правок та можливість порівнювати різні версії сторінок, щоб оцінити зміни або повернутися до попередньої версії, легке відновлення попередніх версій документації у разі потреби.
- *підтримка інтеграції з Git* : Wiki також доступний як окремий репозиторій Git, що дозволяє редагувати документацію локально за допомогою улюбленого текстового редактора або IDE; посилання на Wiki можна легко додавати в Issues, Merge Requests, чи опис проєкту.

- *пошук і навігація*: інтегрований пошуковий інструмент дозволяє швидко знаходити потрібну інформацію в документації; бічне меню автоматично генерується з наявної структури сторінок, забезпечуючи зручний доступ до розділів.
- *підтримка media*: можливість прикріплення та відображення зображень, графіків або діаграм для покращення візуального сприйняття, легке форматування та вставка блоків коду для технічної документації;
- *спільна робота*: підтримка багатокористувацького доступу: кілька членів команди можуть працювати над Wiki одночасно; можливість бачити, хто і коли вніс зміни до документації; команди можуть коментувати сторінки Wiki через Issues або Merge Requests, якщо в них необхідно внести зміни.

Wiki в GitLab є єдиною платформою для зберігання документації, що дозволяє уникнути необхідності використовувати сторонні сервіси. Механізм контролю версій забезпечує гнучке управління змінами, що дозволяє уникнути втрати важливої інформації. Крім того, доступ до Wiki є простим і зручним як через вебінтерфейс, так і через локальні репозиторії.

Wiki в GitLab має кілька обмежень. По-перше, він має обмежені можливості щодо кастомізації зовнішнього вигляду сторінок. Також для великих проєктів, де потрібна масштабна документація, краще використовувати спеціалізовані інструменти, наприклад, Confluence. При великій кількості сторінок виникає необхідність ручного налаштування структури, що може бути не дуже зручним.

Wiki використовується для різних цілей, таких як створення технічної документації, наприклад, інструкцій з налаштування середовища розробки, опису архітектури проєкту та посібників для нових членів команди. Він також підходить для написання керівництв та довідників, таких як посібники користувача, переліки API-методів і керівництва з тестування. Для управління знаннями Wiki допомагає створювати FAQ для команди, бази знань і організовувати логістику процесів. В проєктах з відкритим кодом Wiki використовується для створення інструкцій щодо участі у проєкті, опису структури коду та правил співпраці.

Таким чином, Wiki є потужним і зручним інструментом для управління документацією, який особливо ефективний для невеликих та середніх команд. Завдяки інтеграції з GitLab, цей інструмент дозволяє створювати централізовану базу знань, яка легко підтримується та залишається актуальною. Хоча його функціонал може бути недостатнім для великих

проектів, у поєднанні з іншими інструментами GitLab Wiki стає невід’ємною частиною робочого процесу сучасних команд розробки.

Моніторинг та Аналітика

Моніторинг та Аналітика у GitLab – це набір інструментів, які забезпечують аналіз продуктивності, виявлення проблем, а також збір метрик у реальному часі для оцінки стану застосунків, інфраструктури й процесів розробки. Цей функціонал допомагає командам розробників і DevOps-інженерам приймати обґрунтовані рішення на основі даних.

До основних можливостей Моніторингу та Аналітики у GitLab варто віднести наступні:

- *Моніторинг продуктивності застосунків:* інструмент Application Performance Monitoring дозволяє відслідковувати продуктивність додатків, включаючи час відповіді, навантаження на сервери, використання ресурсів тощо, вбудована інтеграція з Prometheus дає змогу збирати метрики, інтерактивні дашборди з ключовими метриками дозволяють легко візуалізувати стан інфраструктури та застосунків;
- *Інтеграція логування:* можливість інтеграції з системами логування, такими як Elasticsearch, Kibana або Grafana, для аналізу логів, що допомагає швидко ідентифікувати причини збоїв або проблем у застосунку;
- *Моніторинг розгортань (Deploy Monitoring):* відстеження впливу конкретних релізів на продуктивність системи, порівняння стану системи до і після розгортання оновлення та повернення змін (Rollback) у разі виявлення критичних проблем.
- *Моніторинг середовищ (Environments Monitoring):* відстеження стану розробницьких, тестових і продуктивних середовищ; ведення журналу дій для кожного середовища, що дозволяє простежити всі внесені зміни;
- *Аналітика роботи команди:* дозволяє вимірювання швидкості та ефективності обробки Merge Requests та оцінку частоти комітів і продуктивності команди;
- *Аналітика релізів:* збір доказів, які підтверджують стан релізу, включаючи тести, метрики та журнали, перед його затвердженням та відстеження повного циклу розробки, від створення завдання до його виконання та доставки кінцевому користувачу.
- *Моніторинг безпеки:* виявлення та відображення інформації про потенційні загрози безпеці в інфраструктурі або застосунках;

Всі метрики, аналітика та звіти доступні в одному інтерфейсі, що полегшує управління проєктом. Дані оновлюються в реальному часі, що дозволяє оперативно реагувати на проблеми. Інструменти моніторингу інтегровані в CI/CD пайплайни, забезпечуючи безперервний аналіз на кожному етапі розробки. Інтерактивні дашборди надають командам та керівникам прозорий огляд стану проєкту, а система масштабована і підходить як для малих команд, так і для великих корпоративних проєктів.

Інструменти моніторингу та аналітики мають певні обмеження. Зокрема, для повного функціоналу необхідна інтеграція з зовнішніми сервісами, такими як Prometheus чи Grafana. Деякі функції доступні тільки у преміум- або корпоративних версіях GitLab. Налаштування та ефективне використання інструментів можуть вимагати додаткового навчання. Моніторинг великих проєктів може створювати значне навантаження на систему.

Таким чином, розглянуті інструменти для моніторингу та аналітики є потужним засобом підтримки стабільності, безпеки та ефективності проєктів. Завдяки інтеграції з DevOps-процесами, реальному часу та гнучкості налаштувань ці інструменти дозволяють розробникам та командам швидко реагувати на проблеми, оптимізувати процеси й покращувати кінцевий результат. Хоча вони мають певні обмеження, їхній потенціал для підвищення прозорості та керованості проєктів робить їх невід’ємною частиною сучасного програмного забезпечення.

4.2. GitHub

GitHub – це одна з найпопулярніших платформ для зберігання, обміну та співпраці над вихідним кодом. Вона побудована на основі системи контролю версій Git і орієнтована на команди розробників, що прагнуть ефективної роботи над проєктами.

GitHub відрізняється від GitLab своєю орієнтацією на хмарне розміщення та більшим фокусом на відкриті проєкти. Платформа має потужну екосистему з GitHub Marketplace, де доступні численні інтеграції та розширення. GitHub також пропонує унікальні функції для соціальної взаємодії розробників, такі як GitHub Sponsors та GitHub Stars.

GitHub був створений у 2008 році Томом Престоном-Вернером, Крісом Ваномстратом, Пі Джей Гайтом і Скоттом Чаконом. Головною ідеєю платформи було об’єднати можливості Git із веб-інтерфейсом, що значно спрощує управління репозиторіями, обмін кодом і командну співпрацю. У 2018 році GitHub був придбаний корпорацією Microsoft, що дозволило платформі ще більше інтегруватися з екосистемою Microsoft і отримати нові можливості для розвитку.

GitHub розроблено для:

- *зберігання та управління вихідним кодом*: платформа дозволяє створювати Git-репозиторії для розробки програм, зручного відстеження змін та управління історією проєкту;
- *командної співпраці*: забезпечує інструменти для обговорення, рев'ю коду та спільної роботи над проєктами;
- *автоматизації*: використовуючи GitHub Actions, користувачі можуть автоматизувати процеси тестування, збірки та розгортання програмного забезпечення;
- *публікації відкритих проєктів*: GitHub став глобальною платформою для розміщення open-source проєктів, що дозволяє розробникам з усього світу співпрацювати.

Основні функції GitHub наведено нижче:

- *контроль версій із Git*: GitHub забезпечує зручний інтерфейс для роботи з Git, включаючи створення гілок, злиття (merge), управління комітами та відстеження змін;
- *Pull Requests*: інструмент для обговорення, перегляду та затвердження змін перед їхнім інтегруванням у основну гілку проєкту;
- *GitHub Actions*: потужна платформа автоматизації для створення пайплайнів CI/CD, управління робочими процесами та розгортання додатків;
- *Issue Tracker*: вбудований інструмент для трекінгу завдань і помилок із можливістю додавання коментарів, тегів, призначення відповідальних і створення списків справ;
- *Wiki*: кожен репозиторій на GitHub може мати власну вікі для документування функціоналу, архітектури або будь-якої іншої інформації, пов'язаної з проєктом;
- *канбан-дошки (Projects)*: інструмент для управління завданнями за допомогою візуальних дощок, подібний до Trello;
- *інтеграції*: GitHub має інтеграції з багатьма популярними сервісами, такими як Jira, Slack, Docker, Kubernetes, AWS, Azure тощо;
- *безпека*: платформа включає інструменти для сканування вразливостей у залежностях, моніторингу змін у коді та керування доступом до репозиторіїв.

GitHub надає комплексний набір інструментів для роботи з репозиторіями, що охоплює як *базові*, так і *розширені* операції з кодом. На базовому рівні платформа забезпечує всі необхідні функції для щоденної роботи з кодом. Розробники можуть клонувати існуючі репозиторії для

локальної роботи або створювати форки для власних модифікацій. Платформа надає зручний інтерфейс для створення та управління гілками, що дозволяє ізолювати розробку нових функцій або виправлення помилок. Процес внесення змін реалізується через стандартні операції Git: коміти для збереження змін та push/pull операції для синхронізації з віддаленим репозиторієм. GitHub також забезпечує детальне відстеження всіх змін та повну історію розробки проекту.

Для більш складних проєктів GitHub пропонує *розширені* можливості управління кодовою базою:

- адміністратори можуть налаштовувати захищені гілки (Protected branches) з детальними правилами, що визначають хто і як може вносити зміни.
- система Code owners дозволяє автоматично призначати відповідальних розробників для перевірки змін у конкретних частинах коду.
- branch protection rules забезпечують додатковий рівень контролю, дозволяючи встановлювати вимоги до процесу внесення змін, наприклад, обов'язкове проходження тестів або отримання схвалення від певної кількості рев'юерів.
- для керування послідовністю злиття змін використовується Merge queue, що допомагає уникнути конфліктів та забезпечити стабільність основної гілки.

Основні інструменти GitHub

Code Review

Особливу увагу GitHub приділяє процесу перевірки коду. Центральним елементом цього процесу є система Pull Requests, яка надає розширені можливості для коментування коду та обговорення змін. Рев'юери можуть використовувати функцію Suggested changes для пропозиції конкретних змін безпосередньо в інтерфейсі, що значно прискорює процес доопрацювання коду. Для організації процесу перевірки використовуються Review requests, які дозволяють призначати конкретних розробників для рев'ю коду. Коли зміни ще не готові до повноцінного розгляду, розробники можуть створювати Draft pull requests, що дозволяє отримати ранній зворотний зв'язок від команди, не запускаючи формальний процес рев'ю.

Repository Management (Управління репозиторіями)

GitHub надає потужні інструменти для управління репозиторіями коду. Основою системи є Git-репозиторії, які можна створювати як публічними, так і приватними. Розробники можуть клонувати існуючі репозиторії або

створювати форки для власних модифікацій. Платформа забезпечує зручний веб-інтерфейс для перегляду коду, історії змін та гілок розробки. Для захисту важливих гілок використовуються *branch protection rules*, які дозволяють встановлювати правила внесення змін та вимоги до процесу перевірки коду. Система також підтримує розширені можливості для управління доступом, дозволяючи налаштовувати права для різних користувачів та команд.

Issues and Project Management

Для ефективного управління проектами GitHub пропонує систему *Issues*. Цей інструмент дозволяє створювати та відстежувати завдання, помилки та побажання щодо покращень. *Issues* можна категоризувати за допомогою міток, призначати виконавців та пов'язувати з *milestone*'ами для планування релізів.

Основні можливості *Issues*:

- шаблони для різних типів задач;
- мілестоуни та лейбли;
- зв'язування з комітами та PR;
- автоматичне закриття через ключові слова.

Платформа також надає *Projects* – інтерактивні дошки для візуального управління завданнями в стилі канбан. Користувачі можуть створювати різні подання проектів, автоматизувати робочі процеси та відстежувати прогрес за допомогою діаграм та метрик.

Ключові функції *Projects*:

- створення класичних дошок з колонками;
- використання *New Projects* з таблицями та різними представленнями;
- автоматизація через *Project workflows*;
- інтеграція з *Issues* та *Pull Requests*.

Documentation and Wiki

GitHub забезпечує вбудовані інструменти для створення та підтримки документації проекту. Кожен репозиторій може мати власну Wiki-систему, яка підтримує Markdown-форматування та дозволяє створювати структуровану документацію з перехресними посиланнями.

Додатково платформа пропонує *GitHub Pages* – сервіс для хостингу статичних веб-сайтів безпосередньо з репозиторію. Це особливо корисно для створення документації API, технічних керівництв та сайтів проектів. Система також автоматично відображає README-файли репозиторіїв, що допомагає швидко зрозуміти призначення та спосіб використання проекту.

Security Tools

GitHub забезпечує комплексний підхід до безпеки, починаючи з надійної системи аутентифікації. Платформа підтримує декілька рівнів захисту облікових записів:

- Two-factor authentication для додаткового рівня безпеки;
- SSH keys для безпечного з'єднання з репозиторіями;
- Personal access tokens для API та інтеграцій;
- OAuth apps для авторизації сторонніх додатків.

GitHub пропонує користувачам комплексні інструменти для забезпечення безпеки коду. Система включає Dependabot, який автоматично сканує залежності проекту на наявність вразливостей та пропонує оновлення. Code scanning дозволяє виявляти потенційні проблеми безпеки безпосередньо в коді, використовуючи семантичний аналіз. Secret scanning захищає від випадкового потрапляння конфіденційних даних у код, автоматично виявляючи та попереджаючи про наявність токенів доступу, паролів та інших секретних даних. GitHub також підтримує security advisories – механізм для відповідального розкриття інформації про вразливості та координації їх виправлення.

Team Collaboration

Платформа пропонує різноманітні інструменти для ефективної командної роботи. Організації можуть створювати teams - групи користувачів з певними правами доступу та обов'язками. Discussions забезпечують простір для обговорень, що не пов'язані безпосередньо з кодом чи конкретними завданнями. GitHub також підтримує @mentions для швидкого залучення конкретних користувачів до обговорень та notifications для відстеження важливих подій у проекті. Інтеграція з популярними месенджерами та системами управління проектами дозволяє ефективно координувати роботу команди.

Тарифні плани та обмеження

GitHub пропонує різні плани підписки, щоб задовольнити потреби як індивідуальних розробників, так і великих підприємств. Безкоштовний план (Free) надає базовий, але потужний набір інструментів для розробки включає в себе:

- необмежену кількість публічних репозиторіїв;
- отримання базового функціоналу для приватних репозиторіїв;
- змогу використовувати GitHub Actions з обмеженням хвилин;
- базові інструменти безпеки.

План Team розроблений для професійних команд, що потребують розширених можливостей для спільної роботи. Він включає всі функції безкоштовного плану, а також додаткові переваги такі, як:

- розширені можливості для команд;
- більше хвилин для GitHub Actions;
- розширені інструменти безпеки;
- додаткові опції для рев'ю коду.

Enterprise-план призначений для великих організацій, які потребують максимального контролю та безпеки. Цей план надає повний набір корпоративних функцій:

- повний контроль над середовищем;
- розширена підтримка користувачів;
- SAML single sign-on;
- Advanced security features.

Кожен план має свої обмеження щодо кількості співробітників, обсягу сховища та доступних функцій, що дозволяє організаціям вибрати оптимальне рішення відповідно до їхніх потреб та масштабу проекту.

Таким чином, GitHub має безліч переваг, які роблять його популярною платформою серед розробників. Він є де-факто стандартом для open-source спільноти, пропонуючи доступ до мільйонів проєктів і забезпечуючи широку базу користувачів. Зручний веб-інтерфейс дозволяє легко працювати з репозиторіями навіть тим, хто не є експертом у Git. Крім того, платформа має розширену екосистему завдяки GitHub Marketplace, де доступні численні розширення та інструменти для автоматизації й інтеграції. GitHub також однаково підходить як для розробників open-source, так і для корпоративних команд, що працюють із закритими проєктами.

GitHub, на відміну від GitLab, історично більше орієнтований на хмарне розміщення, хоча компаніям із високими вимогами до безпеки доступна версія GitHub Enterprise Server для локального розгортання, однак в неї доволі висока цінова політика. Крім того, хоча безкоштовний план в хмарі має достатній функціонал, деякі функції, такі як розширена аналітика чи більша кількість одночасних пайплайнів, доступні лише в платних версіях.

GitHub широко використовується як індивідуальними розробниками, так і великими командами. Його застосовують для підтримки open-source проєктів, розробки комерційного програмного забезпечення, автоматизації робочих процесів через GitHub Actions, а також для навчання нових розробників у галузі програмування.

4.3. Bitbucket

Bitbucket – це хмарна та локальна платформа для розробки програмного забезпечення, створена для управління Git-репозиторіями, командної співпраці та автоматизації DevOps-процесів. Вона орієнтована на бізнеси будь-якого масштабу, особливо ті, що вже використовують Atlassian-екосистему, до якої належать Jira, Confluence та інші інструменти.

Bitbucket був заснований у 2008 році як платформа для роботи з репозиторіями Mercurial. У 2010 році Atlassian придбала Bitbucket, і з часом акцент змістився на підтримку Git, що стало основним стандартом для систем контролю версій. Сьогодні Bitbucket є однією з провідних платформ для корпоративної розробки програмного забезпечення, особливо серед компаній, які використовують інші продукти Atlassian.

Bitbucket розроблено для:

- *управління вихідним кодом*: робота з Git-репозиторіями, відстеження змін та управління версіями коду;
- *командної співпраці*: інструменти для рев'ю коду, обговорення змін і роботи над спільними проєктами;
- *автоматизації*: використання Bitbucket Pipelines для налаштування CI/CD процесів безпосередньо в репозиторії;
- *інтеграції з Atlassian-продуктами*: легка інтеграція з Jira, Confluence та іншими інструментами для централізованого управління проєктами.

Крім того, платформа забезпечує високий рівень корпоративної безпеки, надаючи розширені можливості для захисту коду та управління доступом.

Основні функції платформи:

- *контроль версій*: Bitbucket підтримує Git як основну систему контролю версій, що дозволяє командам працювати з гілками, відстежувати зміни та об'єднувати код через Pull Requests;
- *Bitbucket Pipelines*: вбудований інструмент для автоматизації CI/CD, який дозволяє створювати, тестувати та розгортати код без використання сторонніх сервісів;
- *Pull Requests*: інструмент для обговорення та затвердження змін перед інтеграцією в основну гілку;
- *інтеграція з Jira*: можливість створювати завдання Jira безпосередньо з репозиторію та відстежувати зв'язок між змінами в коді та бізнес-завданнями;
- *перевірка безпеки (Security Scanning)*: інтеграція із сервісами для сканування залежностей та виявлення вразливостей у коді;

- *командна співпраця*: Bitbucket дозволяє розробникам залишати коментарі до коду, брати участь у рев'ю, додавати рекомендації та обговорювати зміни;
- *масштабованість для команд*: підтримка приватних репозиторіїв, налаштування доступу на рівні гілок і інтеграція з LDAP/Active Directory у локальних версіях;
- *хмарне та локальне розгортання*: Bitbucket доступний як у хмарному середовищі (Bitbucket Cloud), так і для локального розгортання (Bitbucket Server і Bitbucket Data Center).

Bitbucket має низку переваг, які роблять його привабливим вибором для команд розробників. Одна з ключових переваг – інтеграція з Atlassian-екосистемою, що особливо зручно для тих, хто використовує Jira для управління завданнями або Confluence для документації. Платформа пропонує необмежену кількість приватних репозиторіїв навіть у безкоштовній версії, що є ідеальним рішенням для команд, які працюють над закритими проєктами.

Основні функції та інструменти

Repository Management

Bitbucket пропонує комплексний набір інструментів для ефективного управління репозиторіями коду. На базовому рівні платформа забезпечує всі необхідні операції для щоденної роботи з кодом:

- створення та клонування репозиторіїв;
- зручний інтерфейс для керування гілками та комітами;
- можливість форкінгу проєктів;
- злиття змін через механізм merge.

На розширеному рівні Bitbucket надає додаткові можливості для більш складних сценаріїв використання. Зокрема, Branch permissions дозволяють встановлювати детальні правила доступу до різних гілок, забезпечуючи додатковий рівень контролю над змінами в коді. Система merge checks автоматично перевіряє відповідність змін встановленим критеріям якості та безпеки перед їх інтеграцією в основну кодову базу. Для організації ефективного процесу розробки платформа пропонує налаштування branch workflows, що дозволяє визначити оптимальні робочі процеси для різних типів гілок. Підтримка LFS (Large File Storage) забезпечує ефективну роботу з великими файлами, такими як медіаконтент або бінарні дані. Для географічно розподілених команд особливо корисною є функція Smart Mirroring, яка оптимізує швидкість доступу до репозиторіїв.

Code Review

Система перевірки коду в Bitbucket розроблена для забезпечення якості та надійності програмного забезпечення. Центральним елементом цього процесу є Pull Requests, які надають комплексний набір інструментів для ефективного рев'ю коду. Розробники можуть створювати детальні огляди змін, використовуючи вбудовані інструменти для аналізу коду. Система підтримує inline-коментарі, що дозволяє точно вказувати на проблемні місця та пропонувати конкретні покращення. Важливою функцією є можливість блокування merge до повного вирішення всіх зауважень, що забезпечує дотримання стандартів якості.

Процес рев'ю в Bitbucket підтримується різноманітними інструментами для командної роботи. Система дозволяє призначати конкретних ревьюерів для перевірки змін, що забезпечує залучення потрібних експертів. Механізм затвердження змін допомагає відстежувати статус перевірки та гарантує, що всі необхідні схвалення отримані. Підтримка групового рев'ю дозволяє декільком розробникам одночасно працювати над перевіркою коду. Інтеграція з системами CI/CD забезпечує автоматичну перевірку якості коду та виконання тестів.

Issues and Project Management

Bitbucket надає інтегровану систему управління завданнями, яка тісно пов'язана з процесами розробки. Issue Tracking система дозволяє:

- ефективно відстежувати та управляти різними аспектами проєкту;
- кастомізацію полів та workflow, що дозволяє адаптувати систему під специфічні потреби проєкту;
- тісну інтеграцію з комітами та pull requests, що забезпечує прозорий зв'язок між завданнями та змінами в коді;
- автоматично оновлювати статуси завдань на основі дій розробників.

Для ефективного управління проєктами Bitbucket пропонує різноманітні інструменти візуалізації та планування. Agile-дошки допомагають командам візуалізувати робочий процес та відстежувати прогрес. Функціонал спринт-планування дозволяє ефективно організовувати роботу в рамках agile-методологій. Система надає різноманітні інструменти для створення звітів та відстеження метрик продуктивності. Особливо цінною є глибока інтеграція з Jira Software, що створює єдине середовище для управління проєктами розробки.

CI/CD ma DevOps

Bitbucket Pipelines представляє собою потужне рішення для автоматизації процесів розробки та розгортання. В області Continuous

Integration платформа забезпечує автоматичне виконання тестів при кожній зміні коду, що допомагає швидко виявляти потенційні проблеми. Система також проводить статичний аналіз коду для виявлення можливих помилок та порушень стандартів кодування. Постійна перевірка якості допомагає підтримувати високі стандарти розробки. Важливим аспектом є автоматичне сканування безпеки, яке допомагає виявити потенційні вразливості на ранніх етапах.

У сфері Continuous Delivery Bitbucket надає інструменти для автоматизації процесу розгортання додатків. Система підтримує автоматичне розгортання в різні середовища, що значно прискорює процес доставки програмного забезпечення. Розробники можуть налаштувати ефективне управління артефактами, забезпечуючи надійне зберігання та доставку компонентів програмного забезпечення. Платформа також пропонує глибоку інтеграцію з популярними хмарними платформами, що спрощує процес розгортання в хмарній інфраструктурі.

Security Tools

Bitbucket забезпечує комплексний підхід до безпеки, надаючи різноманітні інструменти для захисту коду та управління доступом. Система контролю доступу дозволяє встановлювати детальні налаштування прав для різних користувачів та груп. Функція IP whitelisting забезпечує додатковий рівень захисту, обмежуючи доступ до системи з визначених IP-адрес. Двофакторна автентифікація надає додатковий рівень безпеки для облікових записів користувачів. Для корпоративних клієнтів особливо важливою є можливість інтеграції з корпоративними системами автентифікації.

В області безпеки коду Bitbucket пропонує різноманітні інструменти для виявлення та запобігання потенційним загрозам. Система автоматично сканує залежності проєкту на наявність відомих вразливостей. Регулярна перевірка безпеки допомагає виявити потенційні проблеми в коді. Особлива увага приділяється захисту конфіденційних даних та секретів. Система також підтримує детальний аудит змін, що дозволяє відстежувати всі модифікації коду та конфігурацій.

Documentation

Bitbucket надає комплексні інструменти для створення та підтримки документації проєкту. Wiki-система платформи підтримує зручне форматування з використанням markdown, що дозволяє створювати добре структуровану документацію. Версіонування документації забезпечує відстеження змін та можливість повернення до попередніх версій. Система пропонує готові шаблони сторінок, що прискорює процес створення типової

документації. Особливо корисною є інтеграція з Confluence, яка дозволяє створити єдину базу знань проєкту.

У контексті документації репозиторіїв Bitbucket забезпечує зручні інструменти для створення та підтримки технічної документації. Зокрема:

- README файли автоматично відображаються на головній сторінці репозиторію, надаючи швидкий доступ до основної інформації про проєкт;
- платформа підтримує автоматичну генерацію документації на основі коментарів у коді;
- інтеграція з різними системами документування коду дозволяє створювати комплексну технічну документацію.

Тарифні плани та обмеження

Bitbucket пропонує гнучку систему тарифних планів, що задовольняє потреби різних користувачів та організацій. Free план орієнтований на невеликі команди та індивідуальних розробників, надаючи базовий набір функцій для ефективної роботи. Цей план включає підтримку до 5 користувачів, необмежену кількість приватних репозиторіїв та базові інструменти CI/CD з квотою 50 хвилин Pipelines на місяць.

Standard план розширює можливості платформи для середніх та великих команд. Він знімає обмеження на кількість користувачів та надає розширений набір інструментів для командної роботи. План включає 2,500 хвилин Pipelines на місяць та забезпечує пріоритетну підтримку для швидкого вирішення технічних питань.

Premium план призначений для великих організацій з підвищеними вимогами до функціональності та безпеки. Він включає всі можливості Standard плану та додає необмежену кількість хвилин Pipelines. План також надає доступ до розширених функцій безпеки та можливість дзеркалювання репозиторіїв для покращення продуктивності.

Data Center план розроблений спеціально для підприємств, які потребують локального розгортання системи. Цей план забезпечує високу доступність сервісу, підтримує географічну реплікацію даних та надає корпоративний рівень підтримки для критично важливих систем.

Особливості розгортання

Bitbucket пропонує три основні варіанти розгортання, кожен з яких має свої особливості та переваги. Bitbucket Cloud – це хмарне рішення, яке забезпечує максимальну простоту використання та обслуговування та автоматичні оновлення системи, безпроблемну інтеграцію з хмарними

сервісами та можливість швидкого масштабування відповідно до потреб організації.

Bitbucket Server забезпечує можливість локального розгортання системи та орієнтований на організації, які потребують повного контролю над своїм середовищем розробки. Це рішення, що особливо важливо для компаній з високими вимогами до безпеки даних. Server версія підтримує глибоку інтеграцію з локальними системами та дозволяє здійснювати детальну кастомізацію під специфічні потреби організації.

Bitbucket Data Center підтримує кластерне розгортання для забезпечення високої доступності сервісу і може бути корисним для великих підприємств. Система може горизонтально масштабуватися для обробки зростаючого навантаження. Особливо важливою функцією є підтримка географічної реплікації, що забезпечує ефективну роботу розподілених команд.

Bitbucket має кілька обмежень, які слід враховувати. Він менш популярний серед open-source спільноти та рідше використовується для розміщення відкритих проєктів порівняно з GitHub. Хмарна версія платформи має певні обмеження у продуктивності та функціоналі в порівнянні з локальними версіями. Також інтерфейс Bitbucket може здатися менш інтуїтивним для новачків у порівнянні з конкурентами, такими як GitHub.

Bitbucket найкраще підходить для комерційних команд, оскільки пропонує інтеграцію з Atlassian-екосистемою та підтримує приватні репозиторії. Він ефективний для автоматизації DevOps-процесів завдяки використанню Bitbucket Pipelines для CI/CD. Платформа добре підходить для закритих проєктів завдяки надійним засобам безпеки та розширеному управлінню доступами. Крім того, вона особливо зручна для організацій, які працюють із Jira, забезпечуючи повну синхронізацію для трекінгу завдань і управління проєктами.

Bitbucket, як частина Atlassian-екосистеми, пропонує потужні інструменти для розробки та управління проєктами, особливо для бізнесів, які цінують інтеграцію, гнучкість і можливості автоматизації.

4.4. Порівняння інструментів для роботи з системами контролю версій

GitLab, GitHub та Bitbucket використовують Git як основу для управління вихідним кодом, забезпечують прозору, надійну та організовану роботу з кодом, проте мають різні підходи до роботи з репозиторіями та DevOps-процесами. GitLab надає комплексне рішення для CI/CD, GitHub має потужну open-source спільноту, а Bitbucket тісно інтегрований з Atlassian-продуктами, такими як Jira та Confluence.

Зокрема, всі зазначені інструменти забезпечують:

- *відстеження змін*: розробники можуть легко переглядати історію змін, повертатися до попередніх версій або аналізувати коміти для розуміння еволюції проєкту;
- *робота з гілками (branches)*: кожна система дозволяє створювати окремі гілки для нових функцій, виправлення помилок чи експериментів. Це запобігає змішуванню незавершених змін із основною гілкою;
- *злиття змін (merge)*: усі ці платформи підтримують зручні інструменти для об'єднання змін, дозволяючи проводити рев'ю коду перед тим, як включити новий функціонал у основну версію програми.

Підтримка CI/CD

Усі згадані системи інтегрують або надають інструменти для безперервної інтеграції та доставки, що є основою сучасних DevOps-процесів (рис. 4.1).

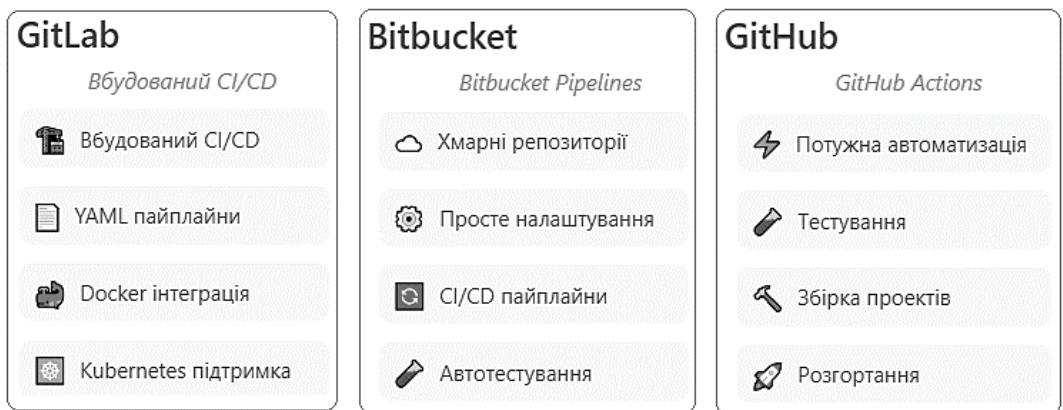


Рис.4.1. Особливості CI/CD інструментів Git-платформ

GitHub використовує GitHub Actions – потужний інструмент для автоматизації робочих процесів. Це включає налаштування тестування, збірки та розгортання програм із широкою підтримкою сторонніх інтеграцій.

Bitbucket пропонує Bitbucket Pipelines, які забезпечують CI/CD із простим налаштуванням для хмарних репозиторіїв, але з деякими обмеженнями для локальних серверів.

GitLab має вбудовану підтримку CI/CD, яка дозволяє автоматизувати тестування, збірку та розгортання без необхідності сторонніх інструментів.

GitLab пропонує гнучкі налаштування пайплайнів через YAML-файли, а також інтеграцію з Docker і Kubernetes для роботи з контейнеризованими додатками.

Спільна робота над кодом

Платформи, пропонують широкий набір інструментів для командної роботи, що полегшує процес колаборації розробників, тестувальників і DevOps-фахівців:

- *Merge Requests (аналог Pull Requests)*: інструмент для перегляду і об'єднання змін у коді. У GitLab Merge Requests містять усі необхідні компоненти, включаючи рев'ю, історію обговорення та статус тестування CI/CD. У GitHub: Pull Requests підтримують коментарі, рев'ю, інтеграцію з GitHub Actions для запуску тестів і перевірок. Вони включають шаблони для стандартизації запитів і інтеграцію з GitHub Discussions для тривалих обговорень. У Bitbucket: Pull Requests дозволяють створювати обговорення навколо змін у коді, з підтримкою коментарів, зазначення рев'юерів та автоматичних перевірок через Pipelines.
- *рев'ю коду (Code Review)*: розробники можуть переглядати код своїх колег, залишати зауваження, пропонувати зміни та затверджувати готові коміти. Цей процес допомагає покращити якість коду, зменшити кількість помилок і підвищити продуктивність команди;
- *обговорення та коментарі*: є змога коментувати конкретні рядки коду, обговорювати правки безпосередньо в Merge Requests, створювати тредовані дискусії та додавати роздільні коментарі для кращої організації роботи;
- *гнучкі правила рев'ю*: можна налаштувати обов'язковий список рев'юерів, що гарантує проходження змін кількома членами команди перед їхнім впровадженням.

Ці можливості роблять спільну роботу над кодом ефективною, особливо для розподілених команд.

Інтеграція з іншими інструментами

Всі наведені інструменти в певній мірі підтримують інтеграції з великою кількістю популярних сервісів і платформ, що дозволяє організаціям налаштовувати свої робочі процеси під конкретні потреби:

- *Jira*: GitLab має пряму інтеграцію, що дозволяє автоматично створювати посилання на Jira-завдання з комітів, Merge Requests та Issues. Інтеграція через GitHub Marketplace (наприклад, плагіни), дозволяє автоматично GitHub додавати посилання на Jira-завдання в коміти та Pull Requests. Bitbucket має найкращу інтеграцію серед

- інших платформ, оскільки Bitbucket є частиною Atlassian Suite. Прямий зв'язок із Jira-завданнями дозволяє переглядати статуси завдань і робити автоматичні оновлення через коміти та Pull Requests;
- *Slack*: нативна інтеграція GitLab через Slack App дозволяє надсилати повідомлення про статус пайплайнів, Merge Requests, Issues та інші події у Slack-канали. GitHub використовуючи інтеграцію через GitHub Slack App може надсилати повідомлення про Pull Requests, Issues, CI/CD-статуси та коміти у Slack-канали. Інтеграція Bitbucket через Bitbucket Slack App для сповіщень про Pull Requests, Issues та CI/CD статуси у Slack-канали;
 - *Microsoft Teams*: GitLab підтримується через вебхуки або нативні конектори, які надсилають оновлення про зміни в коді, пайплайни та рев'ю. Нативна інтеграція GitHub через GitHub App для Microsoft Teams дозволяє надсилати сповіщення та обговорювати зміни в коді. Інтеграція Bitbucket через вебхуки або сторонні конектори дає можливість отримувати повідомлення про активність у репозиторіях;
 - *Docker*: GitLab має глибоку інтеграцію, включаючи автоматичну побудову Docker-образів у CI/CD та публікацію їх у GitLab Container Registry. GitHub має підтримку створення Docker-образів через GitHub Actions. Образи можна зберігати в GitHub Container Registry. Bitbucket підтримує створення Docker-образів через Bitbucket Pipelines. Можна зберігати образи в Docker Hub або інших реєстрах;
 - *Kubernetes*: нативна інтеграція дає можливість розгортати застосунки, здійснювати моніторинг та управляти кластерами Kubernetes безпосередньо через GitLab. Інтеграція GitHub через GitHub Actions дає можливість автоматичного розгортання на Kubernetes-кластери. Інтеграція Bitbucket через Pipelines розгортає контейнери у кластери Kubernetes;
 - *Prometheus і Grafana*: GitLab має вбудовані функції моніторингу через Prometheus. Дані з Prometheus можна інтегрувати з Grafana для візуалізації. GitLab дозволяє відстежувати метрики CI/CD і застосунків. GitHub підтримує через сторонні інтеграції або GitHub Actions збір метрик і моніторинг CI/CD. Bitbucket можна інтегрувати через сторонні інструменти або вебхуки, але нативної підтримки немає;
 - *інші сервіси*: інтеграція GitLab з Terraform для інфраструктури як коду, підтримка AWS, Google Cloud і Azure для розгортання, а також Webhooks та REST API для кастомних інтеграцій. GitHub має багатий Marketplace, що включає інтеграції з AWS, Google Cloud, Azure

DevOps, Jenkins, Terraform, і численні Webhooks для кастомізації. Bitbucket підтримує глибоку інтеграцію з іншими Atlassian-продуктами, такими як Confluence, Trello, Bamboo. Підтримка AWS, Google Cloud, Terraform та інших сервісів через Pipelines.

Завдяки цим інтеграціям команди можуть об'єднувати розрізнені інструменти в єдину екосистему, що значно підвищує продуктивність.

Управління проектами

Всі розглянуті платформи мають вбудовані функції для управління проектами, що дозволяють командам не лише писати код, але й планувати та організовувати свою роботу:

- *трекінг завдань*: у GitLab та GitHub є інструменти для створення та керування завданнями, коментувати їх, призначати відповідальних і відстежувати прогрес;
- *епіки та дорожні карти (Roadmaps)*: Функціонал дозволяє групувати завдання в більші епіки, створювати загальну картину проекту та встановлювати терміни виконання.
- *Kanban-дошки*: дозволяють візуально відображати завдання на різних етапах виконання. Це спрощує управління робочими процесами та допомагає командам слідкувати за прогресом;
- *Milestones*: інструмент для визначення ключових етапів проекту. Команди можуть використовувати його для фіксації дедлайнів та пріоритизації завдань;
- *інструменти звітності*: сервіси надають аналітичні дані про виконання завдань, трекінг часу та ефективність команди.

Використання даних інструментів дає змогу поєднувати розробку коду та управління проектами в одній платформі, що спрощує процеси планування й виконання завдань, а також забезпечує прозорість для всіх учасників.

Вибір системи контролю версій залежить від потреб команди, її робочого процесу та технічних вимог. GitLab, GitHub та Bitbucket є найпопулярнішими платформами для управління репозиторіями, співпраці над кодом та інтеграції з DevOps-практиками. Кожна з цих платформ має свої переваги та обмеження, які роблять їх більш або менш придатними для різних сценаріїв використання.

У таблиці 4.1 наведено порівняння ключових відмінностей між GitLab, GitHub та Bitbucket за такими критеріями, як модель використання, підтримка CI/CD, можливості DevSecOps, інтеграції з інструментами моніторингу, управління репозиторіями, функції управління проектами та вартість.

Таблиця 4.1 Відмінності між системами контролю версій

Критерій	GitLab	GitHub	Bitbucket
Модель використання	SaaS (хмара) і Self-Managed (локальний сервер)	Переважно SaaS, обмежені можливості корпоративного локального розгортання (GitHub Enterprise Server)	SaaS і Self-Managed (Bitbucket Server)
CI/CD	Вбудований CI/CD без додаткових інструментів	GitHub Actions інтегрований, але вимагає окремого налаштування	Bitbucket Pipelines доступний, але обмежений для локальних серверів
DevSecOps	Вбудовані інструменти для безпеки: SAST, DAST, Dependency Scanning, Container Scanning	Обмежена підтримка, переважно через сторонні інтеграції	Обмежена підтримка, інтеграції через сторонні інструменти
Інструменти моніторингу	Вбудована інтеграція з Prometheus, Grafana для збору метрик і аналізу продуктивності	Немає власних інструментів моніторингу, доступ через сторонні рішення	Немає вбудованих інструментів моніторингу
Підтримка репозиторіїв	Необмежена кількість приватних і публічних репозиторіїв	Безкоштовні приватні репозиторії, з можливістю їх масштабування	Переважно підтримує приватні репозиторії (обмежені публічні)
Управління проєктами	Підтримка епік, дорожніх карт, Kanban-дошок, завдань і milestone	Основний фокус на Issues, Projects і Kanban-дошках (простий функціонал)	Інтеграція з Jira для розширених функцій управління проєктами

Вартість	Безкоштовний рівень із розширеними можливостями, платні тарифи з додатковими функціями	Безкоштовний рівень, але функції Enterprise доступні тільки у платних тарифах	Доступні гнучкі ціни, але платформи Atlassian (наприклад, Jira) часто вимагають додаткових витрат
-----------------	--	---	---

Таким чином, GitLab є ефективним локальним рішенням, більше підходить для комплексного управління DevOps-процесами завдяки вбудованому CI/CD, DevSecOps функціям і широкій інтеграції з DevOps інструментами.

GitHub – зазвичай надає послуги за моделлю SaaS і має обмежені можливості локального розгортання; ідеально підходить для відкритих проєктів і команд, що шукають потужний інструмент із широким вибором інтеграцій і GitHub Actions для CI/CD.

Bitbucket найкраще інтегрується з продуктами Atlassian, такими як Jira, що робить його зручним вибором для команд, які використовують Atlassian Suite.

Контрольні питання до розділу 4

- 1) Яке основне призначення системи GitLab?
- 2) Які основні функції виконує GitLab CI/CD?
- 3) Які основні ролі користувачів існують у GitLab?
- 4) Як GitHub та GitLab забезпечують керування доступом до репозиторіїв?
- 5) Яке призначення має GitLab Runners?
- 6) З якою метою використовуються Protected Branches?
- 7) Що являють собою теги (tags) у GitLab та для чого вони використовуються?
- 8) Що таке DevSecOps у контексті GitLab?
- 9) Яким є призначення GitHub?
- 10) Чим Bitbucket відрізняється від GitHub та GitLab?
- 11) Що з перерахованого можна віднести до головного призначення GitLab?
 - а) Резервне копіювання файлів
 - б) Організація зустрічей розробників
 - в) Управління життєвим циклом розробки ПЗ
 - г) Підтримка клієнтів
 - д) Розробка мобільних додатків

- 12) Яка з наведених платформ першою запропонувала інтеграцію з DevSecOps?
 - a) GitHub
 - b) GitLab
 - c) Bitbucket
 - d) Azure DevOps
 - e) SourceForge
- 13) Який файл використовується для налаштування CI/CD в GitLab?
 - a) .gitlab-config.yml
 - b) .gitlab-ci.yml
 - c) .gitlab-build.yml
 - d) .gitlab-runner.yml
 - e) .gitlab-pipeline.yml
- 14) Яка команда використовується для реєстрації GitLab Runner?
 - a) gitlab-runner start
 - b) gitlab-runner create
 - c) gitlab-runner setup
 - d) gitlab-runner init
 - e) gitlab-runner register
- 15) Який сервіс дозволяє хостити репозиторії Git і Mercurial?
 - a) GitHub
 - b) GitLab
 - c) Bitbucket
 - d) Jenkins
 - e) Google Drive
- 16) Що являє собою інструмент GitLab Runner?
 - a) Компонент для інтеграції GitLab із Jira
 - b) Сервер для хостингу репозиторіїв
 - c) Система моніторингу змін в команді
 - d) Інструмент для виконання завдань CI/CD
 - e) Інструмент для аналізу залежностей
- 17) Яку роль виконує Merge Request у GitLab?
 - a) Автоматизація пайплайнів
 - b) Аналіз продуктивності роботи над проектом
 - c) Керування репозиторіями
 - d) Захист визначених гілок
 - e) Злиття змін із рев'ю коду
- 18) Що являє собою Collaborator у GitHub?
 - a) Сервіс для моніторингу стану репозиторіїв.

- b) Автоматизований бот для перевірки коду.
 - c) Інструмент для створення pull request.
 - d) Користувач, який має доступ до роботи з репозиторіями.
 - e) Роль для видалення репозиторіїв.
- 19) Яка функція GitLab допомагає перевіряти вразливості у контейнерних образах?
- a) SAST
 - b) Container Scanning
 - c) Dependency Scanning
 - d) Review Apps
 - e) Container Monitor
- 20) Який інструмент GitHub дозволяє створювати документацію для проекту?
- a) GitHub Pages
 - b) Code Review
 - c) GitHub Actions
 - d) GitHub Wiki
 - e) GitHub Runner
- 21) Встановіть відповідність між ролями користувачів GitLab та їх рівнем доступу:

A. Owner	1) Базовий доступ для перегляду проекту
B. Maintainer	2) Повний контроль над проектом та його налаштуваннями
C. Developer	3) Можливість працювати з кодом та створювати merge requests
D. Guest	4) Управління проектом без адміністративних прав

- 22) Встановіть відповідність між функціями безпеки GitLab та їх описом:

A. SAST (Static Application Security Testing)	1) Аналізує залежності для виявлення вразливих версій бібліотек.
B. DAST (Dynamic Application Security Testing)	2) Аналізує статичний код для виявлення вразливостей.
C. Dependency Scanning	3) Перевіряє працюючий додаток на можливі ризики.
D. Container Scanning	4) Сканує контейнерні образи на наявність вразливостей.

23) Встановіть відповідність між інструментами GitLab та їх функціями:

A. Boards B. Epics C. Milestones D. Roadmaps	1) Інструмент для групування завдань у часові рамки або спринти. 2) Візуалізація завдань у вигляді колонок для управління прогресом. 3) Інструмент для управління великими задачами, які охоплюють кілька завдань (Issues). 4) Візуалізація загальної дорожньої карти проєкту.
---	---

24) Встановіть відповідність між етапами налаштування GitLab та відповідними командами:

A. Перезапуск служб GitLab B. Встановлення GitLab C. Реєстрація Runner D. Застосування конфігурації	1) <code>sudo gitlab-ctl restart</code> 2) <code>sudo EXTERNAL_URL="http://gitlab.example.com" apt-get install gitlab-ee</code> 3) <code>sudo gitlab-ctl reconfigure</code> 4) <code>sudo gitlab-runner register</code>
--	--

25) Встановіть відповідність між основними компонентами GitHub та їх призначенням:

A GitHub Actions B Code Review C. Issue D. Projects	1) Автоматизація тестування, збірки та розгортання. 2) Перевірка та обговорення коду. 3) Створення та відстеження завдань. 4) Візуальне керування завданнями в стилі канбан.
--	---

РОЗДІЛ 5. ОРКЕСТРАЦІЯ НЕПЕРЕРВНОЇ ІНТЕГРАЦІЇ

У сучасному світі розробка програмного забезпечення стала складним процесом, який вимагає ефективного управління, автоматизації та інтеграції. Одним із ключових підходів, який допомагає забезпечити безперервність і високу якість у створенні ПЗ, є оркестрація неперервної інтеграції. Ця технологія дозволяє розробникам автоматизувати тестування, збірку і доставку програмного забезпечення, забезпечуючи його надійність і зменшуючи ризики.

Неперервна інтеграція – це практика в розробці програмного забезпечення, яка полягає у частому злитті змін до основної кодової бази. Цей підхід включає:

- *автоматизоване тестування*: після кожного злиття код перевіряється за допомогою набору тестів;
- *побудова артефактів*: код збирається у виконувану форму або інші необхідні артефакти;
- *моніторинг процесу*: розробники отримують швидкий зворотний зв'язок про успішність інтеграції.

5.1. Завдання та переваги оркестрації CI

Оркестрація неперервної інтеграції – це процес управління і координації автоматизованих етапів розробки, тестування та збірки програмного забезпечення. Завдяки оркестрації CI забезпечується злагоджена робота всіх компонентів і інструментів, що беруть участь у життєвому циклі розробки, що дозволяє автоматизувати рутинні завдання, прискорювати випуск продукту і зменшувати ризик людських помилок.

Оркестрація неперервної інтеграції є невід'ємною частиною сучасного процесу розробки програмного забезпечення. Вона забезпечує автоматизацію ключових етапів життєвого циклу проєкту, сприяючи підвищенню ефективності роботи команд, мінімізації людських помилок та прискоренню релізів.

Таблиця 5.1 ілюструє основні завдання оркестрації CI, які забезпечують стабільність, продуктивність та якість процесів розробки.

Автоматизація пайплайнів розробки забезпечує управління послідовністю виконання процесів, таких як компіляція коду, тестування, аналіз якості та розгортання. Інтеграція з репозиторіями коду дозволяє автоматично запускати процеси CI після внесення змін, що забезпечує ефективність і оперативність. Моніторинг і звітність надають розробникам інформацію в реальному часі про статус збірок, результати тестів і виявлені помилки, сприяючи швидкому вирішенню проблем. Оптимізація роботи з залежностями допомагає управляти

зовнішніми бібліотеками та середовищами виконання, мінімізуючи ризик конфліктів.

Таблиця 5.1. Основні завдання оркестрації CI

Завдання	Опис
Автоматизація пайплайнів	Управління послідовністю виконання процесів, включаючи компіляцію коду, тестування, аналіз якості та розгортання
Інтеграція з репозиторіями	Автоматичний запуск процесів після внесення змін до коду
Моніторинг і звітність	Надає в реальному часі статус збірок, результати тестів і виявлені помилки
Оптимізація роботи з залежностями	Управління бібліотеками для уникнення конфліктів

На рисунку 5.1 наведено основні етапи оркестрації CI.

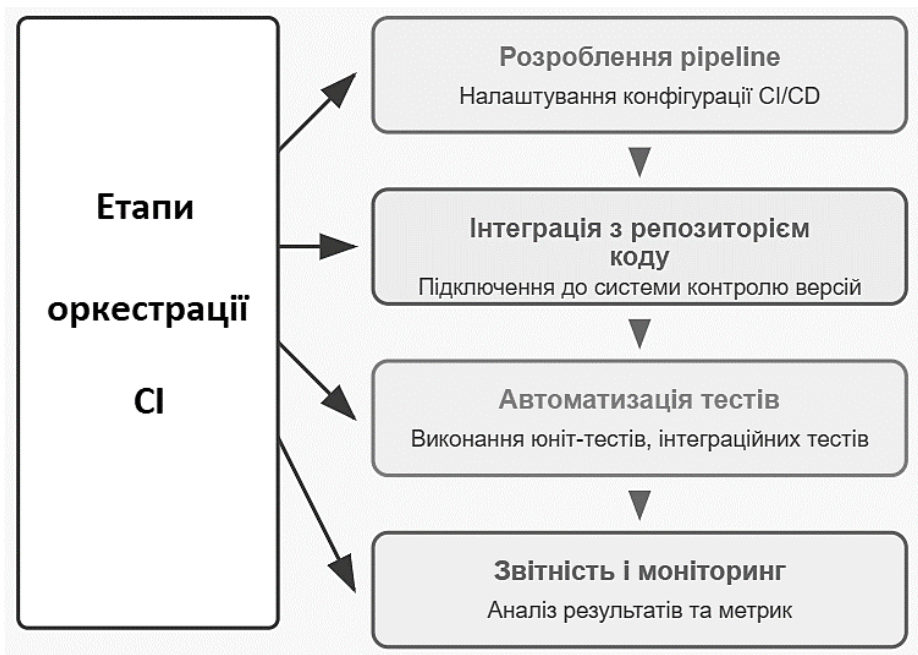


Рис. 5.1. Основні етапи оркестрації CI

1. Розробка pipeline

Pipeline – це набір етапів або процесів, які визначають послідовність дій для інтеграції, тестування та доставки програмного забезпечення. Розробка pipeline включає:

- *опис конфігурації*: використання спеціалізованих файлів, таких, наприклад, як `.gitlab-ci.yml` для GitLab або `Jenkinsfile` для Jenkins, де прописуються всі етапи інтеграції;
- *розподіл на етапи*: поділ pipeline на логічні частини, наприклад, збірка, тестування, розгортання;
- *визначення залежностей*: опис того, які завдання мають виконуватися послідовно, а які можуть виконуватись паралельно;
- *оптимізація ресурсів*: вибір середовищ для виконання (контейнерів Docker, віртуальних машин або хмарних обчислень).

2. Інтеграція з репозиторієм коду

Цей етап передбачає налаштування тригерів, які автоматично запускають pipeline при внесенні змін у код. Важливі аспекти інтеграції:

- *вебхуки (webhook)*: механізми для миттєвого оповіщення CI-системи про зміни в репозиторії;
- *гілки та pull-реквести*: окрема конфігурація для основної гілки (main) і для гілок розробки, щоб уникнути впровадження помилок у стабільну версію;
- *кешування*: збереження результатів попередніх інтеграцій для пришвидшення повторних збірок.

3. Автоматизація тестів

Інтеграція тестів є невід'ємною частиною CI, що забезпечує якість і стабільність програмного забезпечення. Етапи автоматизації тестів включають:

- *модульне тестування*: перевірка окремих компонентів або модулів програми;
- *інтеграційне тестування*: тестування взаємодії між модулями;
- *системне тестування*: перевірка всієї системи в цілому;
- *тестування безпеки*: виявлення вразливостей у коді за допомогою спеціалізованих інструментів, таких як SonarQube;
- *паралелізація тестів*: виконання кількох тестів одночасно для зменшення часу їх виконання.

4. Звітність і моніторинг

Автоматичне створення звітів і моніторинг виконання pipeline допомагають розробникам оперативно реагувати на проблеми. Цей етап включає:

- *генерація звітів*: створення документів із детальним описом результатів збірки, тестування та інтеграції;

- *візуалізація метрик*: відображення стану збірок, часу виконання тестів і успішності етапів у вигляді графіків або дашбордів;
- *сповіщення*: автоматичне інформування розробників через email, Slack або інші канали про результати виконання pipeline;
- *логи*: збереження журналів виконання для аналізу помилок і відстеження можливих вузьких місць у процесі інтеграції.

5.2. Переваги оркестрації CI

Оркестрація CI забезпечує численні переваги, які позитивно впливають на процес розробки програмного забезпечення. У таблиці 5.2 наведено ключові аспекти, що визначають переваги впровадження CI в сучасних командах розробки.

Таблиця 5.2. Переваги оркестрації CI

Перевага	Опис
Швидке виявлення помилок	Автоматичний запуск тестів дозволяє швидко знаходити та виправляти помилки, спрощуючи їх локалізацію до конкретного коміту.
Підвищення ефективності	Автоматизація рутинних завдань звільняє час для роботи над важливими аспектами та сприяє паралельній роботі команди.
Підвищення якості продукту	Постійне тестування забезпечує відповідність стандартам, високий рівень тестового покриття та стабільність програмного забезпечення.
Прискорення циклу розробки	Автоматизація інтеграції та доставки дозволяє швидше випускати релізи та адаптуватися до змін бізнес-вимог.
Прозорість і контроль	Центральне управління та звіти CI-системи забезпечують повну видимість статусу проекту та відповідність стандартам.
Зниження витрат	Автоматизація зменшує витрати на виправлення помилок і оптимізує використання ресурсів.

Швидке виявлення помилок є ключовою перевагою оркестрації CI. Завдяки автоматичному запуску тестів після кожної зміни коду розробники одразу отримують зворотний зв'язок про проблеми, які виникли. Це дозволяє виявляти та виправляти помилки на ранніх етапах, коли їхнє усунення є менш витратним. Локалізація проблем також стає значно простішою, оскільки помилки можна відстежити до конкретного коміту, що спрощує пошук їхніх причин. Регулярне тестування допомагає уникнути ризику регресій, забезпечуючи стабільність і функціональність уже перевіреного коду.

Підвищення ефективності команди є одним із важливих результатів впровадження оркестрації СІ. Автоматизація рутинних завдань, таких як збірка проєкту, запуск тестів чи перевірка стилів коду, дозволяє зменшити навантаження на команду, звільняючи час для роботи над ключовими аспектами розробки. Оркестрація також сприяє паралельній роботі: різні члени команди можуть виконувати свої завдання одночасно, не залежачи від завершення інших етапів. Регулярна інтеграція коду допомагає уникати масштабних і складних об'єднань змін, які часто спричиняють конфлікти, забезпечуючи злагоджену і безперебійну роботу.

Підвищення якості продукту є одним із головних досягнень завдяки оркестрації СІ. Постійна перевірка коду забезпечує вчасне виявлення невідповідностей вимогам, що дозволяє швидко реагувати на відхилення. Автоматизовані інструменти для перевірки стилю та статичного аналізу коду підтримують високий рівень якості, допомагаючи дотримуватися встановлених стандартів. Завдяки процесу оркестрації підвищується тестове покриття, що зменшує ймовірність прихованих помилок у програмному забезпеченні. Продукт, який проходить через СІ-процеси, демонструє високу стабільність і надійність, адже всі ключові аспекти перевіряються автоматично.

Прискорення циклу розробки є ще однією важливою перевагою оркестрації СІ. Завдяки інтеграції з системами Continuous Deployment (CD), зміни автоматично доставляються на тестові або продакшн-середовища, що значно скорочує час розгортання. Автоматизовані процеси дозволяють частіше випускати релізи, забезпечуючи швидку доставку нових функцій кінцевим користувачам. Гнучкість, яку забезпечує оркестрація СІ, дає змогу командам оперативно реагувати на зміни вимог або бізнес-пріоритетів, підтримуючи ефективність розробки на кожному етапі.

Прозорість і контроль є важливими аспектами оркестрації СІ. Централізоване управління через єдину СІ-систему забезпечує команді повну видимість статусу проєкту, виконаних етапів і наявних проблем. Інструменти СІ генерують докладні звіти, які дозволяють аналізувати продуктивність команди, якість коду та ефективність інтеграції. Автоматизовані перевірки на відповідність стандартам безпеки та галузевим нормам допомагають дотримуватися регуляторних вимог, забезпечуючи надійність і відповідність продукту встановленим стандартам.

Зниження витрат є одним із ключових результатів впровадження оркестрації СІ. Автоматизація процесів суттєво скорочує час, витрачений на рутинні перевірки та виправлення помилок, звільняючи ресурси для більш продуктивної роботи. Менша кількість помилок і стабільність інтеграції допомагають уникнути значних витрат, пов'язаних із усуненням дефектів на

пізніх етапах розробки. Інструменти CI також забезпечують ефективне використання обчислювальних ресурсів, оптимізуючи інфраструктуру та зменшуючи непотрібні витрати.

Таким чином, оркестрація CI стає фундаментальною складовою розробки програмного забезпечення, допомагаючи командам досягати високої продуктивності, якості та конкурентоспроможності на ринку.

5.3. Виклики в оркестрації CI

Складність налаштування включає кілька аспектів. По-перше, різноманітність інструментів: існує багато платформ та інструментів для CI, кожен із яких має свої особливості. Вибір найбільш оптимального рішення вимагає глибокого аналізу потреб проєкту. По-друге, конфігурація середовищ: налаштування середовищ для виконання pipeline (локальних, хмарних або контейнеризованих) може бути складним завданням, особливо для великих команд. Також значним викликом є інтеграція з наявною інфраструктурою, адже поєднання CI-систем із існуючими інструментами розробки, такими як репозиторії, системи управління проєктами та середовища розгортання, вимагає значних зусиль. Нарешті, важливим аспектом є документація та навчання: командам потрібно освоювати нові підходи та стандарти, що може уповільнити початкову інтеграцію.

Проекти часто залежать від множинних зовнішніх бібліотек і сторонніх API, що ускладнює підтримку стабільності pipeline. Зміна версій залежностей може викликати конфлікти або помилки в роботі, особливо при масштабних змінах у коді. Крім того, час завантаження великих пакетів або залежностей під час кожного виконання pipeline може значно збільшити тривалість процесу інтеграції. Забезпечення однаковості середовищ для тестування та розгортання (наприклад, через Docker або віртуальні машини) є ще одним важливим викликом для команд.

Зі збільшенням обсягу коду, кількості розробників та частоти інтеграцій зростає потреба в обчислювальних ресурсах, а їх нестача може уповільнити процес. Також із ростом кількості тестів і складності проєкту тривалість виконання pipeline може значно збільшитися, що негативно впливає на швидкість розробки. Організація ефективного виконання паралельних завдань стає дедалі складнішою зі збільшенням кількості етапів та залежностей між ними. Крім того, масштабовані CI-системи потребують постійного моніторингу, оптимізації та оперативного вирішення проблем.

Використання хмарних сервісів для виконання pipeline може виявитися досить вартісним рішенням, особливо для великих проєктів із високою частотою

інтеграцій. Організації, які обирають локальні сервери, повинні забезпечити їх надійність, безпеку та регулярне оновлення інфраструктури, що також потребує значних ресурсів.

У складних проєктах pipeline можуть містити десятки або навіть сотні етапів, що значно ускладнює їх координацію та управління. Пошук вузьких місць і оптимізація тривалих етапів потребують додаткових зусиль для забезпечення ефективності. Грамотне керування залежностями між завданнями є критично важливим для уникнення збоїв і забезпечення стабільної роботи.

Ще одним фактором може виступати те, що члени команди можуть не мати достатнього досвіду роботи з CI, що нерідко призводить до помилок у налаштуванні та експлуатації. У деяких випадках розробники проявляють опір змінам через звичку до ручного виконання завдань і небажання впроваджувати нові процеси. Забезпечення дотримання стандартів кодування та тестування всією командою також може стати викликом для успішної інтеграції CI.

Забезпечення правильного розподілу прав доступу до CI/CD-системи є важливим для уникнення витоків даних або несанкціонованого доступу. Сторонні бібліотеки, що використовуються в проєкті, можуть містити вразливості, які впливають на загальну безпеку. Крім того, критично важливо налаштувати безпечне зберігання секретів, таких як ключі API, паролі та токени, щоб уникнути їх витоків і забезпечити захист конфіденційних даних.

CI/CD-інструменти постійно оновлюються, тому команди мають швидко адаптуватися до нових можливостей і вимог. Водночас відсутність регулярної оптимізації pipeline може призводити до застарілості процесів, що негативно впливає на продуктивність і конкурентоспроможність проєкту.

Попри ці виклики, грамотний підхід до оркестрації CI дозволяє мінімізувати їхній вплив, забезпечуючи ефективну автоматизацію процесів і стабільну роботу програмного забезпечення.

Оркестрація безперервної інтеграції є фундаментальним елементом сучасного процесу розробки програмного забезпечення. Вона спрямована на оптимізацію робочих процесів, підвищення ефективності команд та забезпечення стабільності програмних рішень. Завдяки автоматизації рутинних завдань, таких як збірка, тестування та доставка коду, CI сприяє зменшенню людського фактора та знижує ризик виникнення помилок.

Сучасні інструменти CI, як-от Jenkins, GitLab CI/CD, GitHub Action, CircleCI чи Travis CI, пропонують командам різні варіанти для досягнення автоматизації залежно від потреб і розмірів проєкту. Ефективне впровадження цих інструментів дозволяє компаніям адаптуватися до швидких змін на ринку, підвищувати конкурентоспроможність і відповідати високим стандартам індустрії.

Із розвитком DevOps-практик та автоматизації в цілому, оркестрація CI продовжує еволюціонувати. Зростання ролі контейнеризації, хмарних обчислень та інструментів штучного інтелекту відкриває нові можливості для підвищення ефективності CI-систем.

Таким чином, оркестрація неперервної інтеграції є не лише технічним процесом, а й стратегічним підходом до розробки програмного забезпечення. Її правильне використання дозволяє командам створювати інноваційні, якісні й конкурентоспроможні продукти, забезпечуючи успіх у динамічному світі ІТ.

5.4. Інструменти для оркестрації CI

Вибір інструмента залежить від вимог проєкту, масштабів, бюджету та рівня автоматизації. Наприклад, Jenkins підходить для великих проєктів з високими вимогами до кастомізації, тоді як Travis CI чи CircleCI є зручними для невеликих команд або проєктів з відкритим кодом. GitHub Actions пропонує зручну інтеграцію з GitHub-репозиторіями та має простий процес налаштування з великою кількістю готових шаблонів для типових задач CI/CD. GitLab CI/CD забезпечує інтеграцію з репозиторіями GitLab та є зручним для команд, які використовують цю платформу для управління кодом. TeamCity є потужним вибором для організацій, які потребують гнучкого налаштування build-процесів, а Bamboo надає переваги тісної інтеграції з Atlassian-екосистемою, включно з Jira та Bitbucket.

Хмарні сервіси, такі як Azure DevOps, AWS CodePipeline чи Google Cloud Build пропонують комплексні рішення для управління проєктами, CI/CD та моніторингу, що робить їх ідеальним для великих корпоративних середовищ.

GitLab CI/CD

GitLab CI/CD – це інтегрований в GitLab інструмент для автоматизації процесів CI/CD, що забезпечує зручний і інтуїтивно зрозумілий спосіб налаштування. Основні характеристики:

- *автоматичне розпізнавання pipeline*: система автоматично знаходить файл .gitlab-ci.yml у репозиторії та запускає процеси;
- *гнучке управління середовищами*: підтримка середовищ розробки, тестування і продакшну з чітким відображенням стадій розгортання;
- *вбудована аналітика*: моніторинг виконання pipeline, аналіз продуктивності і споживання ресурсів;
- *контейнеризація*: підтримка Docker-контейнерів для виконання завдань, що спрощує ізоляцію і масштабування;
- *безкоштовні можливості для відкритих проєктів*: доступ до CI/CD навіть для проєктів з обмеженим бюджетом.

GitLab CI/CD дозволяє командам ефективно керувати збіркою, тестуванням і розгортанням коду завдяки використанню *pipelines* – це послідовність завдань (tasks), які виконуються автоматично після змін у коді і можуть містити такі етапи (stages), як:

- Збірка (Build);
- Тестування (Test);
- Розгортання (Deploy).

Для опису процесів CI/CD потрібно створити файл `.gitlab-ci.yml` у репозиторії. Рисунок 5.2 представляє просту конфігурацію GitLab CI/CD пайплайну, написану мовою YAML.

```
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Building..."

test_job:
  stage: test
  script:
    - echo "Testing..."

deploy_job:
  stage: deploy
  script:
    - echo "Deploying..."
```

Рис. 5.2. Конфігурація GitLab CI/CD пайплайну

Розглянемо опис структури пайплайну (рис. 5.2):

- Визначення етапів (stages):
 - stages: визначає послідовність етапів у пайплайні: build (збірка), test (тестування) і deploy (розгортання).
- Опис завдань (jobs):
 - build_job:

- ✓ Належить до етапу build.
- ✓ Виконує скрипт: `echo "Building..."`.
- `test_job`:
 - ✓ Належить до етапу test.
 - ✓ Виконує скрипт: `echo "Testing..."`.
- `deploy_job`:
 - ✓ Належить до етапу deploy.
 - ✓ Виконує скрипт: `echo "Deploying..."`.

Ця конфігурація демонструє базовий процес CI/CD із трьома послідовними етапами: збірка, тестування і розгортання. Потрібно пояснити детальніше чи доповнити приклад?

GitLab Runner – це інструмент, який виконує завдання з пайплайнів. Виділяють наступні типи Runner: *Shared Runners* (доступні для всіх проєктів у вашій GitLab-інстанції) та *Specific Runners* (прив'язані до певного проєкту або групи).

Для реєстрації Runner використовується команда:

gitlab-runner register

Завдяки інтеграції з Git пайплайни запускаються автоматично після комітів або змін у гілках, що забезпечує безперервний процес розробки. За запуск пайплайнів відповідають наступні тригери:

- Manual Triggers: запуск завдань вручну;
- Scheduled Pipelines: автоматичний запуск пайплайнів за розкладом (наприклад, для тестів);
- API Triggers: запуск пайплайнів через API для інтеграції з іншими системами.

Платформа вирізняється гнучкістю, підтримуючи широкий спектр сценаріїв – від найпростіших до надзвичайно складних.

GitLab CI/CD дозволяє працювати з різними середовищами, включаючи Docker, Kubernetes, хмарні провайдери та локальні сервери. Він також пропонує вбудовані інструменти для аналізу коду й перевірки безпеки, що підвищує рівень захищеності проєктів. Масштабованість платформи забезпечується можливістю додавати нові Runners для розподілення завдань, що робить її ефективною навіть для великих команд і проєктів.

CI/CD у GitLab має певні обмеження, які варто враховувати. Налаштування пайплайнів і Runners може бути складним для новачків, що потребує додаткового часу на освоєння. Для великих проєктів можуть виникати обмеження продуктивності, що вимагає потужніших серверів або оптимізації процесів. Крім того, деякі функції, зокрема Advanced Security, доступні лише в платних планах, що може стати перешкодою для команд із обмеженим бюджетом.

GitLab CI/CD активно використовується для автоматизації розробки, забезпечуючи автоматичне тестування та збірку після кожного коміту. Зокрема, в пайплайн можна запланувати перевірку коду в рамках merge requests або провести інтеграцію з правилами злиття (наприклад, дозволити злиття тільки після успішного виконання пайплайна).

Платформа підходить для розгортання додатків у хмарних середовищах, таких як AWS, Azure або Google Cloud. Вона ефективно підтримує складні DevOps-сценарії завдяки інтеграції з Kubernetes для оркестрації контейнерів. Крім того, GitLab сприяє швидкому циклу розробки, тестування та доставки, що ідеально відповідає принципам Agile і DevOps.

Інструменти CI/CD у GitLab є ключовим компонентом платформи, який дозволяє значно прискорити процес розробки, підвищити якість програмного забезпечення та автоматизувати всі етапи життєвого циклу продукту. Завдяки гнучким налаштуванням, широкій інтеграції та підтримці сучасних технологій, GitLab CI/CD підходить для різних сценаріїв – від невеликих проєктів до масштабних корпоративних рішень.

Більш детально функції та інструменти GitLab було розглянуто в Розділі 4.

Jenkins

Jenkins – це потужна відкрита платформа для автоматизації процесів CI/CD, яка відзначається високою гнучкістю та масштабованістю. Основні переваги Jenkins:

- *широкий набір плагінів*: понад 1800 плагінів дозволяють інтегруватися з різними інструментами, такими як Git, Docker, Kubernetes, SonarQube тощо;
- *гнучка конфігурація*: підтримує налаштування як через графічний інтерфейс, так і через Jenkinsfile (файл декларативної конфігурації pipeline);
- *підтримка різних середовищ*: Jenkins може працювати на локальних серверах, віртуальних машинах або в хмарі;
- *масштабованість*: можливість створення кластерів з майстра і агентів для паралельного виконання завдань;
- *активна спільнота*: постійна підтримка і оновлення завдяки великій спільноті користувачів.

Більш детально Jenkins розглядається в Розділі 6.

GitHub Action

GitHub Actions – це інтегрований інструмент автоматизації CI/CD від GitHub, що дає змогу створювати робочі процеси безпосередньо у GitHub-

репозиторіях та забезпечує автоматизацію всього життєвого циклу розробки програмного забезпечення. Основні переваги сервісу:

- *проста інтеграція з GitHub*: забезпечує безперебійну роботу з кодом та автоматичне відстеження змін; можливість автоматичного запуску workflow при різних подіях в репозиторії;
- *інтеграція з хмарними провайдерами*: дає змогу розгорнути ресурси, використовуючи секрети, що зберігаються в GitHub, та команди CLI або дії, специфічні для хмарного постачальника;
- *marketplace з готовими діями*: велика бібліотека готових actions для типових завдань CI/CD; можливість створення та публікації власних actions для специфічних потреб проекту;
- *матричне тестування*: можливість паралельного тестування на різних операційних системах та версіях мов програмування;
- *гнучка конфігурація workflow*: можливість створення складних pipeline з послідовним та паралельним виконанням завдань;
- *безкоштовні хвилини виконання*: можливість лімітованого безкоштовного запуску та використання self-hosted runners для власних серверів; гнучка система тарифікації для масштабування згідно з потребами проекту.

Більш детально GitHub Actions розглядається в Розділі 7.

CircleCI

CircleCI – це хмарний або локальний сервіс, який забезпечує високу продуктивність у побудові та тестуванні програмного забезпечення. Основні особливості:

- *автоматична інтеграція*: просте налаштування через конфігураційний файл .circleci/config.yml;
- *підтримка різних мов програмування*: сумісність із популярними мовами, такими як Python, JavaScript, Ruby, Go, і багатьма іншими;
- *хмарна інфраструктура*: масштабування завдань у хмарі без потреби в управлінні серверами;
- *кешування і паралельне виконання*: можливість зменшити час виконання pipeline завдяки збереженню результатів і запуску кількох завдань одночасно;
- *інтеграція з популярними інструментами*: підтримка роботи з GitHub, Bitbucket та Docker Hub.

Більш детально CircleCI розглянуто в Розділі 2.

Travis CI

Travis CI – це інструмент CI/CD, який особливо популярний серед проектів з відкритим кодом завдяки своїй простоті та доступності. Основні характеристики:

- *простота налаштування*: pipeline конфігурується через `.travis.yml`, що легко інтегрується з GitHub;
- *підтримка хмарних середовищ*: Travis CI забезпечує виконання завдань у хмарі, що звільняє розробників від необхідності налаштовувати локальні сервери;
- *вбудоване тестування*: підтримка автоматизованого тестування для різних мов програмування;
- *інтеграція з GitHub*: автоматичний запуск pipeline при комітах, pull-реквестах або створенні нових гілок;
- *безкоштовний доступ для open-source проектів*: привабливий вибір для невеликих проектів або стартапів.

Більш детально Travis CI розглянуто в Розділі 2.

Azure DevOps

Azure DevOps – це потужна платформа від Microsoft, яка поєднує інструменти для управління проектами, неперервної інтеграції та доставки, а також DevOps-практики:

- інтеграція з іншими сервісами Microsoft, такими як Azure Cloud і GitHub;
- можливість налаштування пайплайнів через YAML-файли або графічний інтерфейс;
- інструменти для планування роботи, такі як Kanban-дошки, та управління беклогом;
- вбудована підтримка для управління артефактами, тестами та середовищами розгортання.

Глобальна масштабованість досягається завдяки використанню Azure Cloud. Платформа забезпечує можливість інтеграції з іншими інструментами, такими як Jenkins або Terraform. Додатково підтримується високий рівень безпеки завдяки функціям управління доступом і секретами.

Основні функції Azure DevOps:

- *планування*: управління задачами через Azure Boards; беклоги, історії користувачів, Kanban-дошки, діаграми згорання;
- *контроль версій*: сховища Git та TFVC для спільної роботи над кодом;
- *автоматизація CI/CD*: Azure Pipelines для автоматизації зборки, тестування та доставки;

- *тестування*: Azure Test Plans для створення сценаріїв тестування (ручного і автоматизованого);
- *управління артефактами*: Azure Artifacts для зберігання залежностей проєкту (npm, Maven, NuGet);
- *інтеграції*: підтримка інтеграцій із GitHub, Visual Studio, Docker, Kubernetes, Jenkins, Jira тощо;
- *аналітика*: вбудовані інструменти для аналітики продуктивності та моніторингу.
- *оркестрація CI*: можливість створення складних потоків виконання з умовними етапами, паралельним виконанням та автоматичним масштабуванням агентів; підтримка шаблонів для повторного використання конфігурацій та кастомних розширень.

Ця платформа має низку переваг, що роблять її універсальним рішенням для розробників. Завдяки інтеграції з екосистемою Microsoft, вона забезпечує безперешкодну роботу з Azure Cloud, Visual Studio та GitHub. Платформа підтримує багатоплатформні рішення, працюючи на Windows, Linux і macOS, а також підтримуючи будь-які мови програмування.

Хмарна масштабованість дозволяє використовувати ресурси для реалізації великих проєктів, забезпечуючи ефективність і продуктивність. Однак, платформа має кілька обмежень, які слід враховувати. Для малих команд або стартапів вартість може виявитися високою, особливо при використанні розширених функцій. Хоча Azure DevOps підтримує багато інтеграцій, у деяких випадках інтеграція зі сторонніми інструментами може бути ускладненою.

Azure DevOps найкраще підходить для команд, які активно використовують екосистему Microsoft та потребують комплексного рішення для управління життєвим циклом розробки. Платформа особливо ефективна для проєктів із складними вимогами до тестування та автоматизації, а також для організацій, які потребують глибокої інтеграції з Azure Cloud та масштабованої інфраструктури.

AWS CodePipeline

AWS CodePipeline – це повністю керована служба CI/CD від Amazon Web Services, яка автоматизує побудову, тестування та розгортання програмних додатків. Вона інтегрується з іншими сервісами Amazon Web Services (AWS), такими як CodeBuild, CodeDeploy, що дозволяє організовувати безперервну інтеграцію та робочих процесів безперервної доставки (CI/CD) шляхом автоматизації етапів збірки, тестування та розгортання.

До ключових особливості AWS CodePipeline можна віднести:

- *високу гнучкість*: можливість створення складних конвеєрів з різними етапами та паралельним виконанням завдань за допомогою YAML-описів;
- *інтеграція з іншими сервісами AWS*: зручна робота з CodeCommit, CodeBuild, CodeDeploy S3, Lambda, CloudFormation та іншими сервісами AWS для повного циклу розробки;
- *автоматизація*: пайплайн автоматично запускається при внесенні змін у кодову базу, що забезпечує швидку інтеграцію та доставку;
- *масштабованість*: автоматичне масштабування ресурсів для обробки великих навантажень;
- *безпека*: високий рівень захисту даних завдяки інтеграції з AWS Identity and Access Management (IAM);
- *автоматизоване тестування*: підтримка тестування через CodeBuild або сторонні інструменти, такі як Selenium або JUnit;
- *підтримка різних технологій*: широкий спектр технологій, включаючи різні мови програмування, фреймворки та інструменти.

До переваг сервісу AWS CodePipeline варто віднести те, що функціонал сервісу дає змогу охопити весь цикл розробки від коміту до розгортання та він легко інтегрується з іншими сервісами екосистеми AWS, надає можливість обробки великих навантажень.

Основні функції AWS CodePipeline:

- **Планування**: Автоматизація процесів CI/CD з можливістю створення складних пайплайнів.
- **Тестування**: Інтеграція з CodeBuild для автоматизації тестів і перевірки якості коду.
- **Розгортання**: Забезпечує інтеграцію з AWS CodeDeploy для швидкого та безпечного впровадження змін.
- **Моніторинг**: Підтримує вбудовані аналітичні інструменти для відстеження ефективності пайплайнів.

На рис 5.3. наведено етапи конвеєра в AWS CodePipeline.

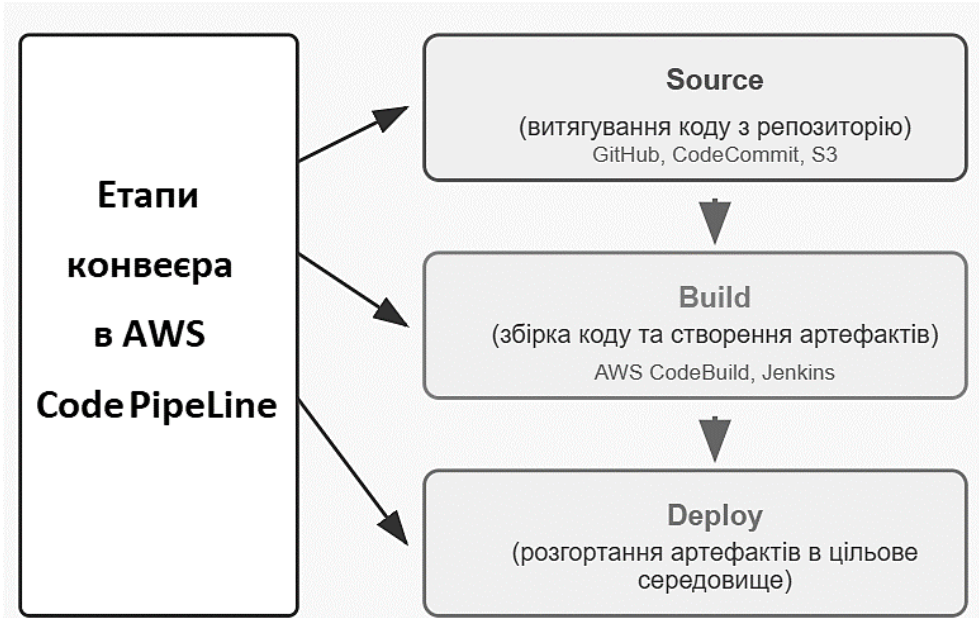


Рис. 5.3. Основні етапи конвеєра в AWS CodePipeline

Плануючи збірку за допомогою AWS CodeBuild і AWS CodePipeline, важливо розробити робочий процес, який автоматизує інтеграцію коду, тестування та розгортання. Корисним може бути дотримання наступного алгоритму:

1. Визначення, які саме процеси запускать збірку (може бути Git push до певної гілки в AWS CodeCommit, GitHub, Bitbucket або іншому підтримуваному джерелі);
2. Вибір обчислювальних ресурсів: AWS CodeBuild дозволяє вибрати тип обчислювальних ресурсів, необхідних для збірки, залежно від розміру та складності процесу створення;
3. Налаштування середовища збірки (можна обрати готовий образ AWS CodeBuild, або створити власний образ згідно потрібної операційної системи, мови програмування та необхідних для збірки інструментів);
4. Визначення команд збірки та порядку їх виконання у файлі `buildspec.yml`;
5. Інтеграція з VPC за умови, що процес збірки потребує доступу до ресурсів у межах VPC, таких як бази даних або внутрішні служби тощо.

AWS CodeBuild підтримує кілька мов програмування та надає готові середовища для Go, Python, Java, Ruby та Rust. Планування збірки за допомогою AWS CodeBuild і AWS CodePipeline вимагає розуміння конкретних потреб

проекту, від середовища збірки та обчислювальних ресурсів до інтеграції VPC і розгортання служб AWS.

Ретельно налаштувавши свої проєкти збірки та використовуючи потужні можливості інтеграції AWS, розробники можуть повністю автоматизувати процес збірки, тестування та розгортання, що призведе до більш ефективних і надійних робочих процесів розробки програмного забезпечення.

Отже, AWS CodePipeline підходить для компаній, що активно використовують екосистему AWS і потребують високої інтеграції з хмарними сервісами. Основними перевагами є гнучкість, масштабованість і тісна інтеграція з іншими продуктами AWS. Однак, для невеликих проєктів вартість може бути значною, а навчальна крива – крутою для початківців.

Google Cloud Build

Google Cloud Build – це хмарний інструмент від Google для автоматизації процесів збірки, тестування та розгортання, який працює на інфраструктурі Google Cloud Platform (GCP) і пропонує швидке та масштабоване виконання завдань CI/CD.

Cloud Build можна інтегрувати з різними службами GCP, що забезпечує безперервний робочий процес від розроблення до розгортання. Сервіс підтримує широкий спектр мов програмування та фреймворків, що робить його універсальним інструментом для розробників, які працюють над різними типами проєктів.

До основних функцій Cloud Build можна віднести наступні:

- *тестування*: підтримка автоматизоване тестування через інтеграцію з популярними фреймворками, такими як JUnit або Selenium.
- *розгортання*: забезпечує доставку програмного забезпечення в GKE, Cloud Run або App Engine.
- *керування артефактами*: інтеграція з Google Artifact Registry для зберігання залежностей і контейнерів.
- *моніторинг*: Вбудовані інструменти для аналізу продуктивності пайплайнів і відстеження результатів.

Google Cloud Build він пропонує високу масштабованість, швидкість виконання завдань і є оптимальним вибором для команд, які прагнуть автоматизувати свої процеси розробки та підвищити ефективність роботи, з огляду на наступні особливості.

- *гнучкість конфігурації*: підтримка налаштування пайплайнів через YAML- або JSON-файли, що дозволяє легко адаптувати процеси до потреб проєкту.

- *хмарна продуктивність*: завдання виконуються в хмарі без необхідності налаштування власних серверів, що забезпечує високу швидкість і масштабованість.
- *підтримка мультимовності*: Cloud Build працює з популярними мовами програмування, такими як Python, Java, Go, Node.js, .NET.
- *глибока інтеграція в екосистему GCP*: пов'язаний із сервісами Google, такими як Google Kubernetes Engine (GKE), Cloud Functions і Cloud Storage.
- *контейнеризація*: підтримка Docker та інтеграція з Google Container Registry для роботи з контейнерами.
- *паралельне виконання*: можливість одночасної обробки кількох завдань, що пришвидшує розробку великих проєктів.
- *інтеграція з системами контролю версій*: підтримка GitHub, Bitbucket, а також Google Cloud Source Repositories.
- *автоматизація*: запуск збірок після змін у репозиторії.

Створення конвеєра CI/CD за допомогою Google Cloud Build передбачає визначення кроків, тригерів (запуск конвеєра за певними подіями) і виходів збірки у файлі конфігурації YAML.

Google Cloud Build ідеально підходить для команд, які активно працюють із GCP, потребують масштабованих хмарних рішень і використовують контейнеризацію. Основними перевагами є продуктивність, глибока інтеграція з GCP та простота налаштування, проте він може бути дорогим для невеликих проєктів і складним для інтеграції з іншими платформами. Доступний безкоштовний рівень для невеликих проєктів.

TeamCity

TeamCity – це комерційний інструмент для неперервної інтеграції та доставки, розроблений компанією JetBrains. Він допомагає автоматизувати процеси збірки, тестування та розгортання програмного забезпечення, забезпечуючи високу продуктивність і зручність у використанні.

Основні особливості TeamCity:

- *гнучкість конфігурації*: TeamCity дозволяє створювати складні пайплайни з використанням як графічного інтерфейсу, так і сценаріїв. Підтримує налаштування для різних типів проєктів і мов програмування;
- *широка інтеграція*:
- підтримує роботу з популярними системами контролю версій, такими як Git, Subversion, Mercurial, Perforce та інші;

- інтегрується з багатьма інструментами JetBrains (наприклад, IntelliJ IDEA), що полегшує розробку та тестування;
- можливість інтеграції з іншими системами CI/CD, такими як Jenkins або Azure DevOps;
- *підтримка Docker і Kubernetes*: TeamCity дозволяє створювати та тестувати контейнери Docker, а також працювати з Kubernetes для оркестрації контейнерів у масштабованих середовищах;
- *паралельне виконання завдань*: забезпечує виконання декількох завдань одночасно, що прискорює процеси розробки і тестування, особливо у великих командах;
- *потужна система кешування*: використовує кешування для збереження попередніх результатів збірки, що дозволяє зменшити час виконання повторюваних операцій;
- *широкі можливості звітності*:
- автоматичне формування звітів про стан збірки, результати тестів і покриття коду;
- графічне відображення метрик для аналізу продуктивності та якості коду;
- *підтримка розподіленої інфраструктури*: TeamCity дозволяє налаштовувати агентів для виконання завдань на різних серверах, забезпечуючи масштабованість і продуктивність;
- *робота з залежностями*: автоматично аналізує залежності між проектами, що дозволяє уникнути конфліктів під час збірки та тестування;
- *безпека*: TeamCity пропонує засоби для управління доступом і інтеграцію з механізмами автентифікації, такими як LDAP або Active Directory;
- *плагіни та розширення*: завдяки плагінам TeamCity можна адаптувати до специфічних потреб проекту. Існує велика кількість готових розширень, доступних для завантаження.

На рисунку 5.4 наведено панель JetBrains TeamCity, яка використовується для роботи з проектами та конфігураціями збірки.

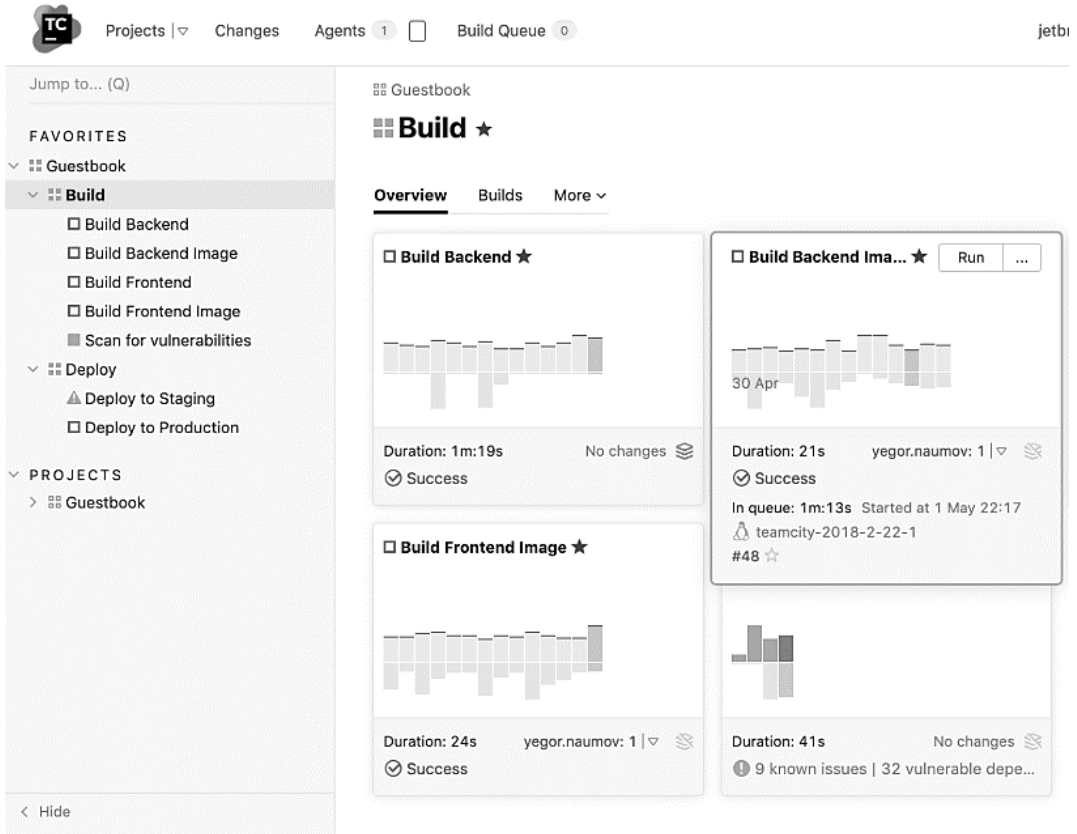


Рис. 5.4. Панель управління проектами JetBrains TeamCity

TeamCity має низку важливих переваг. Його інтуїтивний інтерфейс є зручним і зрозумілим для користувачів із різним рівнем технічної підготовки. Інструмент підтримує роботу з більшістю популярних мов програмування, що робить його універсальним рішенням для різноманітних проектів. Крім того, TeamCity забезпечує швидке налаштування, дозволяючи легко розпочати роботу навіть новим командам, а також пропонує високий рівень кастомізації для адаптації під конкретні потреби.

TeamCity підходить для команд різного розміру – від невеликих стартапів до великих корпорацій. Його часто використовують у поєднанні з іншими DevOps-інструментами для створення безперервних пайплайнів розробки та доставки програмного забезпечення.

TeamCity пропонує безкоштовну версію для невеликих команд, що дозволяє спробувати основний функціонал, а також комерційні плани з розширеними можливостями для великих проектів.

Bamboo

Bamboo – це комерційний інструмент для неперервної інтеграції та доставки, розроблений компанією Atlassian. Bamboo надає розробникам і командам DevOps потужні можливості для автоматизації процесів збирання, тестування, та розгортання програмного забезпечення. Завдяки тісній інтеграції з іншими продуктами Atlassian, такими як Jira і Bitbucket, Bamboo створює єдину екосистему для управління всім життєвим циклом розробки.

Основні особливості Bamboo:

- *інтеграція з Atlassian-екосистемою:*
 - тісно інтегрується з Jira, що дозволяє легко відстежувати виконання завдань і статус розробки;
 - забезпечує автоматичне оновлення статусу комітів у Bitbucket, включаючи відображення результатів тестів;
 - інтеграція з Confluence для створення документів і звітів безпосередньо з результатів збірок;
- *гнучке налаштування пайплайнів:* Bamboo підтримує налаштування складних пайплайнів з можливістю створення декількох етапів (Stages), завдань (Tasks) і тригерів (Triggers). Це забезпечує максимальну кастомізацію процесів CI/CD під конкретні потреби команди;
- *автоматизація процесів CI/CD:* Bamboo автоматично виконує збірку, тестування, аналіз коду та розгортання, що дозволяє мінімізувати ручну роботу та уникнути людських помилок;
- *підтримка контейнеризації та хмарних сервісів:*
 - вбудована підтримка Docker дозволяє працювати з контейнерами та створювати ізольовані середовища;
 - Bamboo інтегрується з хмарними платформами, такими як AWS, Google Cloud, і Microsoft Azure, для розгортання в масштабованих середовищах;
- *масштабованість і розподілене виконання:* Bamboo дозволяє налаштовувати розподілені агенти, які можуть виконувати завдання на різних серверах. Це забезпечує високу продуктивність і можливість роботи над великими проектами;
- *інструменти для тестування та звітності:*
 - Bamboo інтегрується з системами тестування, такими як Selenium, JUnit і NUnit, забезпечуючи автоматичний запуск тестів;
 - формує детальні звіти про стан збірок, результати тестів і покриття коду;

- *контроль залежностей*: Bamboo може автоматично визначати зміни у залежностях проекту, щоб гарантувати стабільність та зручність при роботі з великими кодовими базами;
- *безпека та контроль доступу*: Bamboo дозволяє налаштовувати права доступу для команд і користувачів, забезпечуючи конфіденційність даних та безпеку проекту.

Рисунок 5.5 демонструє інтерфейс Atlassian Bamboo, зокрема розділ Marketplace для вибору й встановлення плагінів.

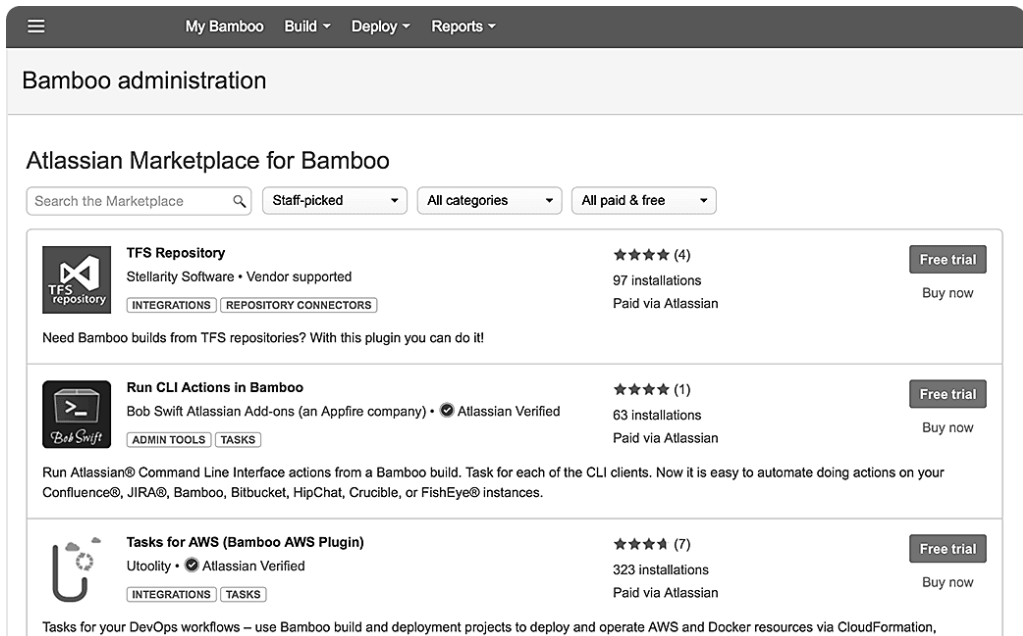


Рис. 5.5. Розділ для вибору й встановлення плагінів Bamboo

Bamboo має кілька важливих переваг. Інструмент забезпечує повну інтеграцію в екосистему Atlassian, охоплюючи всі етапи розробки – від планування в Jira до зберігання коду в Bitbucket і автоматизації через Bamboo. Простота використання проявляється у зручному інтерфейсі та готових шаблонах для швидкого налаштування пайплайнів. Bamboo відрізняється гнучкістю, підтримуючи різні мови програмування та платформи, такі як Java, Python, .NET, PHP, Node.js тощо. Крім того, його масштабованість дозволяє застосовувати інструмент як у невеликих командах, так і у великих підприємствах із багатокомпонентними проектами.

Bamboo також чудово підходить для автоматизації ключових процесів. Наприклад, він використовується для запуску модульних, інтеграційних і енд-то-енд тестів після кожної зміни в коді, розгортання в продакшн середовище з автоматичним оновленням серверів після проходження всіх перевірок, а також для відстеження прогресу за допомогою інтеграції з Jira. Інструмент ефективно підтримує DevOps-процеси, забезпечуючи автоматизацію CI/CD у хмарних або локальних середовищах.

Однак Bamboo має деякі недоліки. Його комерційна ліцензія може стати перешкодою для невеликих команд або проєктів із обмеженим бюджетом. Крім того, для проєктів з відкритим кодом Bamboo є менш популярним вибором, ніж безкоштовні аналоги, такі як Travis CI.

Bamboo є відмінним вибором для компаній, які вже використовують Atlassian-екосистему, таких як Jira і Bitbucket. Він пропонує потужні можливості для автоматизації CI/CD і забезпечує прозорість та зручність управління проектами. Bamboo чудово підходить для команд, які прагнуть створити ефективні процеси розробки та доставки програмного забезпечення.

5.5. Порівняльний аналіз основних інструментів оркестрації CI

Вибір інструменту для оркестрації неперервної інтеграції є важливим етапом у налаштуванні ефективного процесу розробки.

Аналіз основних переваг, недоліків та сфер застосування основних популярних CI-інструментів, наведених в таблиці 5.2, допоможе командам обрати найкраще рішення для забезпечення автоматизації, продуктивності та якості програмного забезпечення.

Таблиця 5.2. Порівняння інструментів для CI

Інструмент	Переваги	Недоліки	Кращий для...
Jenkins	Гнучкість, масштабованість, велика кількість плагінів	Складність налаштування	Великі команди, проєкти з високими вимогами
GitLab CI/CD	Інтеграція з GitLab, зручність, контейнеризація	Залежність від платформи	Малі та середні команди
Travis CI	Простота налаштування, безкоштовність для open-source	Обмеження функціоналу у великих проєктах	Open-source проєкти
CircleCI	Автоматична інтеграція, підтримка різних мов, хмарна інфраструктура	Вартість для великих проєктів	Хмарні проєкти, гнучкі пайплайни

GitHub Action	Інтеграція з GitHub, безкоштовність для open-source проєктів, широкий набір готових actions	Обмеження на безкоштовному плані, залежність від GitHub	Проєкти на GitHub, невеликі та середні команди
TeamCity	Широка інтеграція, гнучкість конфігурації, підтримка Docker	Комерційна ліцензія	Великі проєкти з багатьма залежностями
Bamboo	Інтеграція з Atlassian (Jira, Bitbucket), потужні інструменти тестування	Комерційна ліцензія, менш популярний у open-source	Команди, що працюють в Atlassian-екосистемі
Azure DevOps	Інтеграція з Azure, масштабованість, комплексний підхід	Вартість, ускладнена інтеграція зі сторонніми сервісами	Microsoft-орієнтовані проєкти, enterprise-рішення, .NET розробка
AWS CodePipeline	Інтеграція з AWS, масштабованість, безсерверна архітектура, гнучке налаштування тригерів	Вартість, складність налаштування для початківців	AWS-орієнтовані проєкти та команди
Google Cloud Build	Інтеграція з GCP, serverless архітектура, масштабованість, підтримка контейнеризації	Вартість, обмежена інтеграція зі сторонніми сервісами	Проєкти на базі Google Cloud Platform

Таким чином, незважаючи на дещо спільну кінцеву мету та призначення, різні CI-інструменти пропонують унікальні функції, адаптовані до специфічних потреб команд, проєктів і організацій.

Контрольні питання до розділу 5

- 1) Що являє собою оркестрація неперервної інтеграції?
- 2) Які основні етапи оркестрації CI можна виділити?
- 3) Що таке pipeline в контексті CI?
- 4) Яким чином можна підвищити швидкість виконання пайплайнів?
- 5) З яких компонентів може складатись пайплайн?
- 6) Чому важливою частиною CI є інтеграція з репозиторіями?
- 7) Які інструменти дають змогу забезпечити окрестрацію?

- 8) Яке значення мають контейнери в оркестрації CI?
- 9) Яка роль DevOps інженера у налаштуванні оркестрації CI?
- 10) Яка мета кешування в контексті CI?
- 11) Який з факторів є наслідком автоматизації пайплайнів?
 - a) Полешення рутинних завдань та їх надійність.
 - b) Розгортання проєктів без тестування.
 - c) Зниження якості коду та його об'єму.
 - d) Зменшення рівня кваліфікації розробників.
 - e) Збільшення потреби у ручному тестуванні.
- 12) Що з перерахованого є завданням оркестрації CI?
 - a) Тестування програмного забезпечення
 - b) Оперативне керування базами даних
 - c) Моніторинг з метою аналізу продуктивності серверів
 - d) Автоматизація тестування, збірки та доставки коду
 - e) Створення звітної документації
- 13) Який з етапів CI забезпечує виявлення вразливостей у коді?
 - a) Інтеграційне тестування.
 - b) Тестування безпеки.
 - c) Модульне тестування.
 - d) Тестування продуктивності.
 - e) Регресійне тестування.
- 14) Яка мова для опису пайплайнів використовується у CircleCI та GitLab CI/CD?
 - a) Python
 - b) TypeScript
 - c) YAML
 - d) Groovy
 - e) Ruby
- 15) Який з елементів пайплайна відповідає за виконання окремих завдань?
 - a) Triggers
 - b) Steps
 - c) Stages
 - d) Jobs
 - e) Artifacts
- 16) Який з наведених інструментів є безкоштовним для проєктів з відкритим кодом?
 - a) AWS CodePipeline
 - b) Travis CI
 - c) Bamboo

- d) Azure DevOps
- e) CircleCI
- 17) Який інструмент від Google Cloud Platform використовується для оркестрації CI?
 - a) CodePipeline
 - b) Google Cloud Build
 - c) Google Drive
 - d) Jenkins
 - e) Google Storage Bucket
- 18) Що забезпечують контейнери в контексті оркестрації CI?
 - a) Моніторинг продуктивності системи.
 - b) Мануальне виконання інтеграційних тестів.
 - c) Ізоляцію середовища для стабільного виконання завдань.
 - d) Автоматичне створення документації.
 - e) Уникнення кешування даних.
- 19) Який з інструмент CI є найбільш популярним та має понад 1800 плагінів?
 - a) GitLab CI/CD.
 - b) Bamboo.
 - c) Travis CI.
 - d) AWS CodePipeline
 - e) Jenkins.
- 20) Який протокол найбільш часто використовується для автоматизації розгортання в CI/CD?
 - a) HTTPS
 - b) SSH
 - c) FTP
 - d) SMTP
 - e) WebSocket
- 21) Встановіть відповідність між етапами CI та їхнім описом:

A. Створення pipeline	1) Налаштування webhook для запуску процесів.
B. Інтеграція з репозиторієм	2) Модульне та інтеграційне тестування проєкту.
C. Автоматизація тестів	3) Опис конфігурації та етапів, визначення залежностей.
D. Звітність	4) Генерація звітів та візуалізація метрик.

22) Встановіть відповідність між інструментами CI та їхніми особливостями:

A. Jenkins	1) Тісна інтеграція з Atlassian-екосистемою.
B. GitLab CI/CD	2) Простота налаштування для проєктів з відкритим кодом.
C. Travis	3) Велика кількість плагінів і гнучкість.
D. Bamboo	4) Інтеграція з GitLab, зручність використання.

23) Встановіть відповідність між наступними елементами компонентами пайплайна у Jenkins та їхніми функціями:

A. Triggers	1) Події, які запускають виконання пайплайна.
B. Stages	2) Ізольовані середовища, де виконуються завдання пайплайна.
C. Artifacts	3) Групи завдань, що виконуються послідовно.
D. Runners	4) Результати виконання завдань, які передаються між етапами.

24) Встановіть відповідність між інструментами CI та їхніми конфігураційними файлами:

A. GitLab CI/CD	1) .gitlab-ci.yml
B. Jenkins	2) .travis.yml
C. Travis	3) .github/workflows/<workflow-name>.yaml
D. GitHub Actions	4) Jenkinsfile

25) Встановіть відповідність між перевагами оркестрації CI та їхнім описом:

A. Швидке виявлення помилок.	1) Централізоване управління, детальні звіти.
B. Прискорення циклу розробки.	2) Автоматичне тестування та локалізація проблем.
C. Прозорість і контроль.	3) Автоматизація інтеграції та доставки.
D. Зниження витрат.	4) Оптимізація ресурсів, зменшення дефектів.

РОЗДІЛ 6. ІНСТРУМЕНТ ДЛЯ БЕЗПЕРЕРВНОЇ ІНТЕГРАЦІЇ JENKINS

Jenkins – це популярний інструмент з відкритим вихідним кодом, який використовується для автоматизації процесів розробки програмного забезпечення. Основним завданням Jenkins є автоматизація процесів побудови, тестування та інтеграції програмного забезпечення. Його гнучкість дозволяє налаштовувати процеси Continuous Integration (CI) та частково підтримувати Continuous Delivery (CD), інтегруючись з іншими DevOps-інструментами.

Jenkins спрощує координацію роботи між різними командами розробників, зменшуючи кількість ручної роботи та мінімізуючи ризик помилок, які можуть виникнути під час складних циклів розробки. Його масштабованість дозволяє інтегруватися з практично будь-якою інфраструктурою, починаючи від локальних серверів і закінчуючи хмарними платформами.

На рисунку 6.1 зображено логотип Jenkins – іконку дворецького у смокінгу з метеликом. Він символізує автоматизацію рутинних процесів у розробці програмного забезпечення. Косуке Кавагучі, розробник інструменту, вибрав назву «Дженкінс» (а також колишню назву «Хадсон»), оскільки, на його думку, це була загальна назва для британських дворецьких, що слугує аналогією щодо функціональності інструменту, який він розробив.



Рис. 6.1. Логотип Jenkins

Jenkins працює за принципом "master-agent", що дозволяє масштабувати CI/CD-процеси на різних серверах. Він може запускати завдання на різних операційних системах (Windows, Linux, macOS) та інтегруватися з мовами програмування, такими як Java, Python, JavaScript, Ruby, Go тощо. Завдяки потужній системі плагінів, Jenkins може інтегруватися з популярними інструментами розробки, такими як Git, Docker, Kubernetes, Maven, Gradle, Selenium та іншими.

Гнучкість Jenkins дозволяє створювати як прості задачі, так і складні пайплайни, що включають безліч етапів і умов. Крім того, система дозволяє

налаштовувати оркестрацію процесів розробки за допомогою сценаріїв на мові Groovy, яка використовується для створення Jenkins Pipeline.

Jenkins також є важливим інструментом для підтримки сучасних DevOps-підходів, забезпечуючи автоматизацію розгортання додатків, тестування та моніторингу. Це робить його незамінним елементом в екосистемі інструментів, які використовуються в сучасній розробці програмного забезпечення.

6.1. Типи робіт та застосування

Jenkins пропонує гнучкість у виконанні різноманітних завдань завдяки підтримці кількох типів робіт, кожен із яких адаптований до певних сценаріїв використання. Це дозволяє ефективно організувати автоматизацію процесів розробки та розгортання програмного забезпечення.

Рисунок 6.2 демонструє вікно вибору робіт у системі управління проектами.

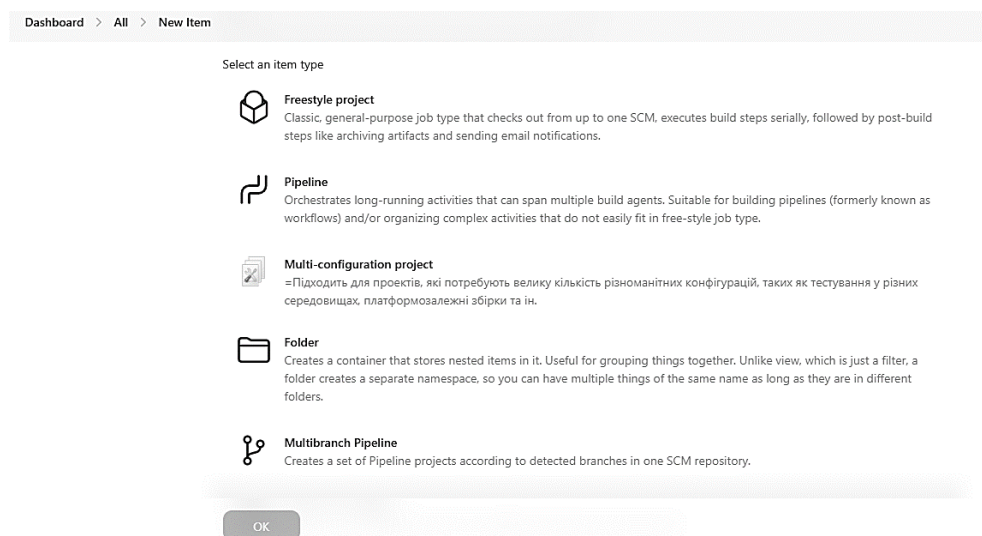


Рис. 6.2. Вікно вибору типу робіт у Jenkins

Основні опції включають:

- *Freestyle project* – класичний універсальний тип задачі, який працює з одним SCM (системою контролю версій), виконує послідовно кроки складання, а також пост-білд дії, наприклад, архівацію артефактів та відправлення електронних листів;
- *Pipeline* – використовується для організації складних завдань, які можуть включати кілька агентів побудови, підходить для створення складних конвеєрів;

- *Multi-configuration project* – підходить для проєктів із великою кількістю різноманітних конфігурацій, таких як тестування у різних середовищах, платформозалежні збірки тощо;
- *Folder* – створює контейнер для організації вкладених елементів, що зручно для групування схожих задач;
- *Multibranch Pipeline* – створює набір конвеєрів на основі виявлених гілок в одному SCM репозиторії.

Freestyle Project

Freestyle Project – це базовий тип роботи в Jenkins, який призначений для виконання простих завдань. Він забезпечує легке налаштування через графічний інтерфейс і не вимагає глибоких знань програмування чи написання складних сценаріїв.

Завдяки своїй простоті та універсальності, Freestyle Project підходить для виконання широкого спектра задач, таких як:

- *компіляція коду*: запуск компіляторів для різних мов програмування;
- *виконання тестів*: інтеграція з інструментами для тестування, наприклад JUnit, для перевірки якості програмного забезпечення;
- *запуск скриптів*: виконання *shell*- або *batch*-скриптів для автоматизації окремих кроків;
- інтеграція з системами контролю версій: завантаження останніх змін із Git, GitHub, SVN або інших систем;
- *генерація звітів*: створення звітів про збірку, тестування чи аналіз коду.

Основні можливості Freestyle Project є наступними:

- *завантаження коду з репозиторію*:
 - Jenkins може підключитися до систем контролю версій, таких як Git або Subversion, для автоматичного отримання останніх змін коду;
 - це налаштовується у розділі Source Code Management, де вказується URL репозиторію та гілка, з якою працюватиме Jenkins;
- *виконання команд у терміналі*: у розділі Build можна налаштувати запуск команд або скриптів, які виконуватимуться під час роботи. Наприклад:

mvn clean install

./run-tests.sh

- *інтеграція з інструментами тестування*: Freestyle Project дозволяє автоматично запускати тести та аналізувати результати. Наприклад, за допомогою плагіна JUnit можна зібрати звіти про успішність тестів і відобразити їх у вигляді діаграм або таблиць;

- *повідомлення про результати*: Jenkins може надсилати повідомлення про результати роботи (успішність або помилки) через електронну пошту чи інші сервіси, такі як Slack. Це налаштовується в секції Post-build Actions.

Freestyle Project має наступні сценарії використання:

- *початкові інтеграції*: використовується для створення базових CI-процесів, таких як отримання коду з репозиторію, збірка проєкту та запуск простих тестів;
- *одноразові завдання*: підходить для виконання одноразових задач, наприклад, запуску міграції бази даних чи оновлення документації;
- *малі проєкти*: використовується для невеликих проєктів, які не потребують складних залежностей чи інтеграцій.

Рисунок 6.3 демонструє інтерфейс налаштувань у Jenkins для конфігурації певного завдання.

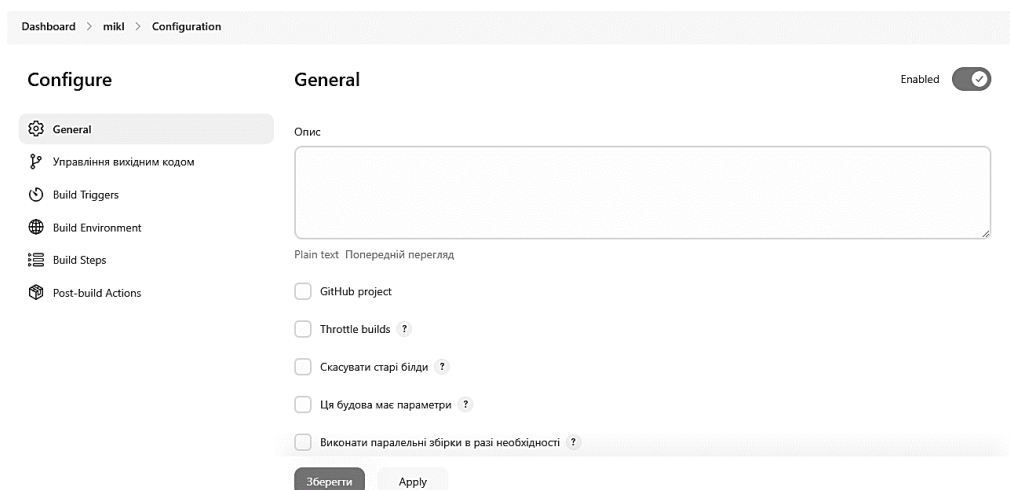


Рис. 6.3. Інтерфейс налаштувань Freestyle Project

Нижче наведено призначення основних пунктів лівої панелі меню:

- *General* – основні налаштування.
- *Управління вихідним кодом* – налаштування джерела вихідного коду.
- *Build Triggers* – тригери для запуску збірки.
- *Build Environment* – параметри середовища для збірки.
- *Build Steps* – кроки процесу збірки.
- *Post-build Actions* – дії після завершення збірки.

Переваги Freestyle Project полягають у його простоті налаштування, оскільки всі конфігурації виконуються через графічний інтерфейс без необхідності писати код. Цей тип роботи є надзвичайно гнучким, дозволяючи вирішувати широкий спектр базових завдань. Інтуїтивний інтерфейс робить Freestyle Project ідеальним вибором для початківців або тих, хто прагне швидко налаштувати Jenkins для виконання простих задач. Крім того, він легко інтегрується з іншими інструментами завдяки підтримці численних плагінів, які значно розширюють його функціональні можливості.

Freestyle Project має певні обмеження, зокрема низьку масштабованість, що робить його непридатним для складних процесів автоматизації. У таких випадках доцільніше використовувати Pipeline, який забезпечує більшу гнучкість завдяки сценаріям на Groovy. Ще одним недоліком є відсутність вбудованої підтримки кодування процесів, що ускладнює перенесення налаштувань на інші Jenkins-сервери.

Розглянемо приклад використання Freestyle Project

Завдання: автоматизувати збірку та тестування Java-проєкту.

1. Кроки налаштування:

- створіть нову роботу в Jenkins і виберіть тип Freestyle Project;
- у секції Source Code Management додайте URL репозиторію Git;
- у секції Build додайте команди для виконання збірки:

mvn clean compile

mvn test

- у секції Post-build Actions додайте звіт про успішність тестів за допомогою плагіна JUnit.

Freestyle Project залишається важливим елементом Jenkins завдяки своїй простоті, універсальності та легкості використання. Він ідеально підходить для швидкого налаштування базових задач або невеликих проєктів. Хоча для складних сценаріїв і складної інтеграції краще використовувати Pipeline, Freestyle Project залишається чудовим інструментом для виконання простих завдань у Jenkins.

Pipeline

Pipeline є потужним інструментом для складної та гнучкої організації процесів CI/CD. Його головною особливістю є використання сценаріїв, написаних мовою Groovy, що дозволяє детально описати багатоступеневі процеси, такі як збірка, тестування, розгортання та моніторинг. Завдяки Pipeline, Jenkins надає можливість централізованого керування складними робочими процесами.

Pipeline дозволяє описувати процеси CI/CD в одному файлі, який називається Jenkinsfile. Цей файл зберігається разом із кодом у репозиторії, що робить процес автоматизації прозорим і контрольованим. Pipeline підтримує:

- о умови та цикли, що дозволяють виконувати динамічні дії залежно від параметрів або стану системи;
- о паралельне виконання етапів, яке скорочує час виконання складних процесів, розподіляючи завдання між різними агентами;
- о етапи та кроки, що дозволяють структурувати процеси у зрозумілі блоки для кращого контролю та налагодження.

Рисунок 6.4 демонструє інтерфейс налаштувань конфігурації завдання у Jenkins.

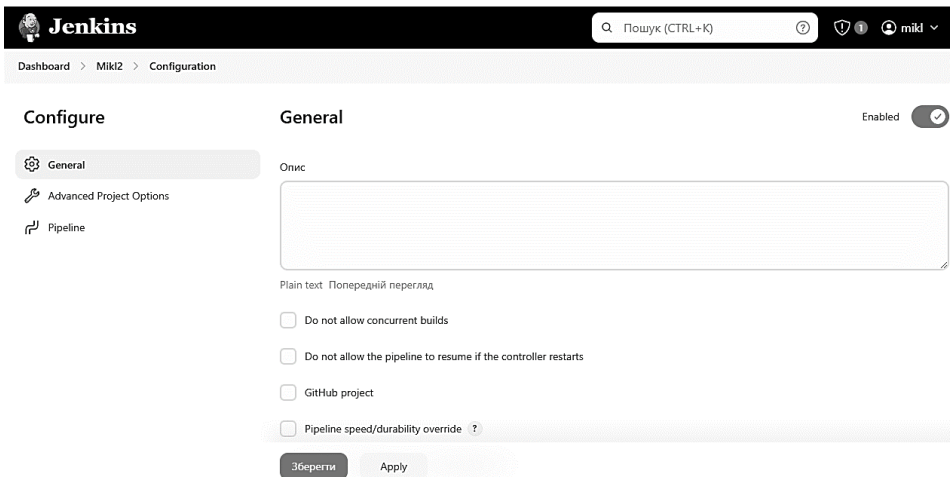


Рис. 6.4. Інтерфейс налаштувань конфігурації Pipeline

У лівій панелі меню розміщені розділи, які відповідають за конфігурацію проєкту. Розділ *General* відкриває доступ до загальних налаштувань завдання, тоді як *Advanced Project Options* містить розширені параметри, що дозволяють детальніше налаштувати поведінку проєкту. Розділ *Pipeline* відповідає за конфігурацію конвеєра, де задаються етапи та умови його виконання.

Pipeline володіє значними перевагами, що роблять його ідеальним вибором для складних проєктів. Уся логіка автоматизації може бути описана в одному файлі – Jenkinsfile, що спрощує обслуговування, зберігання і спільну роботу над проєктом. Завдяки підтримці паралельного виконання та динамічного розподілу завдань між агентами, Pipeline здатен ефективно обробляти великі навантаження.

Використання мови Groovy забезпечує високу гнучкість, дозволяючи створювати умовні конструкції, цикли та складні сценарії, адаптовані до конкретних потреб. Візуалізація етапів у веб-інтерфейсі Jenkins сприяє прозорості процесів і спрощує виявлення та виправлення помилок. Зберігання Jenkinsfile у системах контролю версій, таких як Git, дозволяє вести історію змін і легко повертатися до попередніх конфігурацій, що додає зручності та контролю в управлінні процесами автоматизації.

Pipeline ідеально підходить для складних проєктів із численними залежностями, наприклад:

- побудова та тестування багатокомпонентних систем;
- паралельне тестування на різних платформах і середовищах;
- розгортання в контейнеризованих середовищах за допомогою Docker або Kubernetes;
- моніторинг та інтеграція зі сторонніми сервісами, такими як Slack, Jira або Prometheus.

Pipeline є ключовим компонентом Jenkins для реалізації повного циклу CI/CD. Його гнучкість і потужність роблять його ідеальним інструментом для команд, які прагнуть автоматизувати складні процеси, забезпечити високу якість програмного забезпечення та ефективно керувати життєвим циклом своїх проєктів.

Multi-configuration Project

Multi-configuration Project – це потужний тип роботи в Jenkins, який дозволяє запускати задачі з різноманітними конфігураціями. Він особливо корисний для проєктів, де необхідно тестувати та виконувати завдання в різних середовищах або з різними параметрами.

Цей тип роботи забезпечує гнучке налаштування для виконання таких задач, як:

- тестування на різних операційних системах (Windows, Linux, macOS);
- використання різних версій бібліотек, залежностей або інструментів;
- запуск програм на різних версіях Java, Python чи інших мов програмування;
- перевірка сумісності з різними конфігураціями обладнання або програмного забезпечення.

На рисунку 6.5 наведено інтерфейс конфігурації Jenkins для Multi-configuration Project.

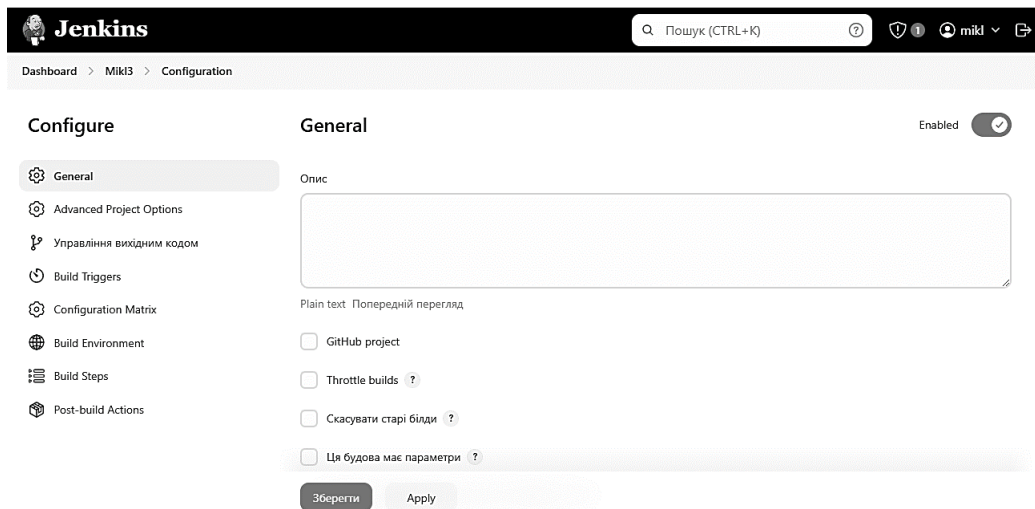


Рис. 6.5. Інтерфейс конфігурації Multi-configuration Project

Ліва панель інтерфейсу містить наступний перелік вкладок для налаштування проєкту в Jenkins:

General – загальні налаштування проєкту, такі як опис, інтеграція з GitHub, параметри збірок тощо.

Advanced Project Options – додаткові опції для налаштування проєкту (зазвичай приховані, доки не розгорнути).

Управління вихідним кодом – налаштування системи контролю версій (наприклад, Git, Subversion) для отримання вихідного коду.

Build Triggers – тригери для запуску збірок, наприклад, за розкладом або при змінах у репозиторії.

Configuration Matrix – налаштування матричних збірок для багатоплатформенних чи параметризованих збірок (опція доступна для мультиконфігураційних проєктів).

Build Environment – налаштування середовища збірки (наприклад, змінні середовища, підготовка перед виконанням).

Build Steps – етапи збірки, що описують, як виконувати завдання (наприклад, викликати скрипти чи компілятори).

Post-build Actions – дії, що виконуються після завершення збірки (наприклад, надсилання повідомлень, архівація артефактів, запуск тестів).

Multi-configuration Project дозволяє створювати матриці конфігурацій, які автоматично генерують і виконують комбінації завдань. Наприклад, для тестування програми можна створити матрицю, яка включає:

- операційні системи: Windows, Linux;

- версії Java: 8, 11;
- типи баз даних: MySQL, PostgreSQL.

Це призводить до автоматичного створення та виконання тестів для всіх можливих комбінацій, наприклад:

- Windows + Java 8 + MySQL.
- Linux + Java 11 + PostgreSQL.

Отже, Multi-configuration Project є ефективним рішенням для середовищ, які потребують високої адаптивності та масштабності. Він забезпечує гнучкість конфігурацій, дозволяючи легко налаштовувати матриці з різними параметрами, що спрощує адаптацію до змін у проєкті. Автоматичне тестування програм у різних середовищах допомагає виявляти специфічні помилки або проблеми сумісності, оптимізуючи процес перевірки.

Автоматизація створення конфігурацій значно економить час, роблячи процес тестування та виконання завдань швидшим і менш трудомістким. Завдяки можливості інтеграції з моделлю Master-Slave (Controller-Agent), Multi-configuration Project ефективно розподіляє навантаження між вузлами, підвищуючи ефективність виконання завдань навіть у складних і масштабних проєктах.

Multi-configuration Project використовується у випадках, коли проєкт має широкий спектр залежностей або середовищ виконання. Наприклад:

- *мультиплатформне тестування*. Перевірка роботи програмного забезпечення на різних операційних системах;
- *перевірка сумісності*. Тестування програми з різними версіями залежностей, такими як бібліотеки, API чи драйвери;
- *паралельне тестування*. Запуск тестів із різними конфігураціями для швидкого отримання результатів;
- *CI/CD для великих команд*. Використання різних конфігурацій залежно від компонентів або підсистем у великих проєктах.

Приклад налаштування Multi-configuration Project

Для тестування вебзастосунку можна налаштувати матрицю, яка перевіряє:

- операційні системи: Windows, Linux.
- браузері: Chrome, Firefox, Safari.
- версії JavaScript-двигунів: ES5, ES6.

Це дає змогу переконатися, що вебзастосунок коректно працює у всіх можливих конфігураціях, що є критичним для забезпечення якісного кінцевого продукту.

Multi-configuration Project забезпечує ефективність і надійність у роботі з проєктами, які потребують складних та багатограних процесів автоматизації, роблячи його важливим інструментом для розробників і тестувальників.

Folder

Тип роботи Folder забезпечує ефективну організацію проєктів у вигляді ієрархічної структури, що особливо корисно для великих команд або компаній із великою кількістю робіт. Цей інструмент використовується для групування робіт, які належать до одного модуля, продукту, підрозділу або команди, створюючи чітку структуру та впорядкованість у середовищі Jenkins.

Folder дозволяє об'єднувати різні типи завдань, такі як Freestyle Project, Pipeline чи Multi-configuration Project, в одне логічне дерево, полегшуючи навігацію та керування.

Основні переваги Folder полягають у його зручності та розширеній функціональності. Цей інструмент забезпечує комфортну навігацію навіть при роботі з великою кількістю проєктів, що дозволяє швидко знаходити потрібні завдання та ефективно структурувати їх. Folder також надає можливість спільного налаштування параметрів для всіх проєктів у папці, таких як глобальні змінні середовища або права доступу для певної групи користувачів.

На рисунку 6.6 показано інтерфейс налаштувань Folder.

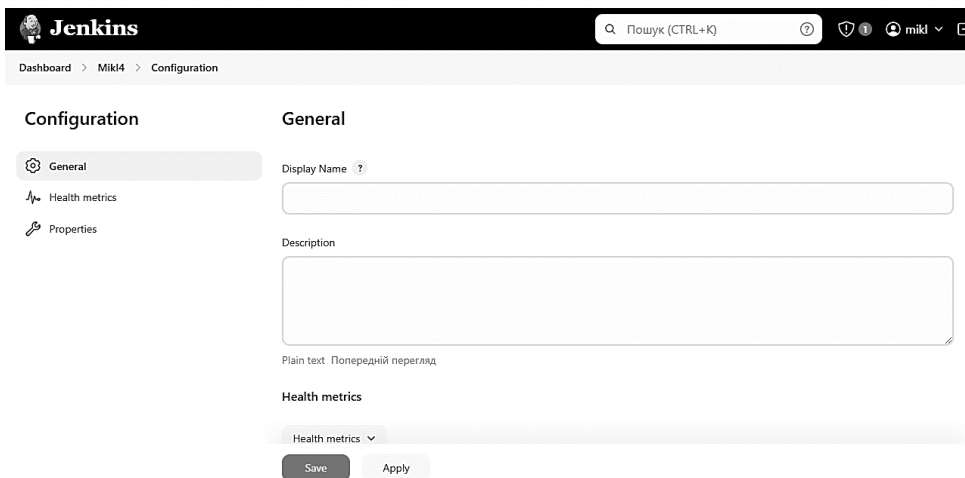


Рис. 6.6. Інтерфейс налаштувань Folder

Ліва панель має наступні пункти:

General – загальні налаштування проєкту (відображення імені, опису, основних параметрів).

Health metrics – налаштування метрик, що визначають стан або продуктивність проєкту.

Properties – вкладка для визначення додаткових властивостей проєкту, таких як параметри доступу, змінні середовища тощо.

Інтеграція зі стратегіями безпеки дозволяє налаштовувати права доступу до папок окремо, забезпечуючи контроль на рівні команди або продукту. Крім того, Folder спрощує масштабування проєктів: додавання нових робіт у вже створену папку не потребує додаткових налаштувань, якщо папка має попередньо задані глобальні параметри.

Folder є незамінним для великих організацій або команд, які працюють над кількома модулями чи продуктами. Наприклад:

- *розділення за командами.* Усі проєкти конкретної команди зберігаються в окремій папці з відповідними правами доступу.
- *розподіл за модулями.* Проєкти, пов'язані з певним модулем, згруповані разом, що спрощує керування залежностями.
- *робота з великими продуктами.* Якщо продукт складається з кількох компонентів, кожен із них можна організувати в окрему папку з власними параметрами.

Folder також підтримує інтеграцію з плагінами, які розширюють можливості Jenkins. Наприклад, плагіни дозволяють автоматично створювати підпапки для Multibranch Pipeline, що зручно для роботи з великою кількістю гілок у репозиторії.

Крім того, папки можуть містити власні налаштування, наприклад, планувальники завдань або скрипти для автоматичного створення нових робіт, що значно спрощує адміністрування великих Jenkins-середовищ.

Folder забезпечує простоту, впорядкованість та ефективність управління у складних CI/CD-процесах, надаючи необхідні інструменти для організації роботи навіть у масштабних командах.

Multibranch Pipeline

Multibranch Pipeline є потужним інструментом Jenkins, який дозволяє автоматизувати створення та управління пайплайнами для кожної гілки репозиторію. Це ідеальний вибір для сучасних методологій розробки, таких як GitFlow, оскільки забезпечує автоматичне налаштування процесів для функціональних або релізних гілок.

Multibranch Pipeline інтегрується з системами контролю версій, такими як Git, GitHub або Bitbucket, і автоматично створює пайплайни для гілок, у яких є файл Jenkinsfile. Завдяки цьому інструменту кожна гілка має можливість

виконувати ізольовані завдання, зберігаючи індивідуальні налаштування, визначені в Jenkinsfile.

На рисунку 6.7 показано сторінку конфігурації Multibranch Pipeline в Jenkins.

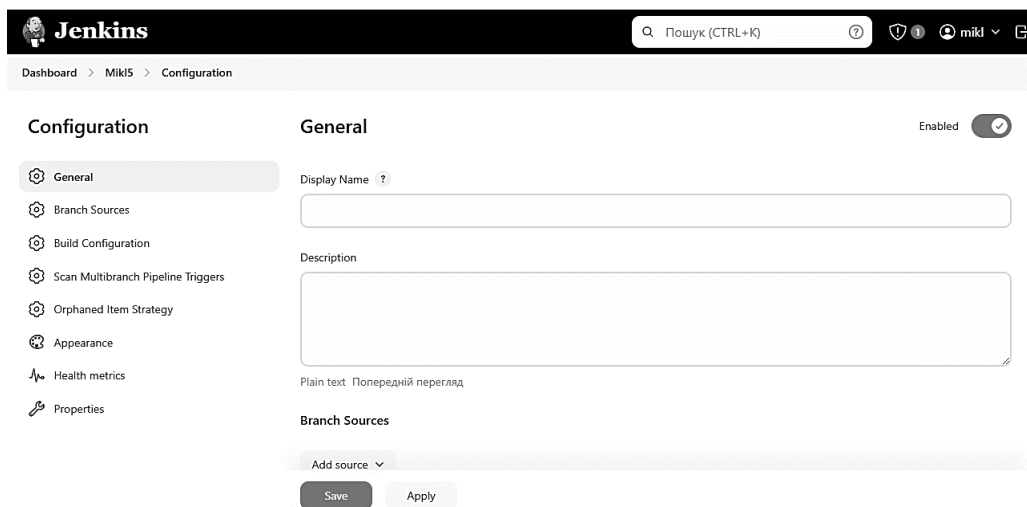


Рис.6.7. Сторінка конфігурації Multibranch Pipeline

Ліва панель містить такі пункти:

General – основні налаштування проєкту (ім'я, опис, базові параметри).

Branch Sources – налаштування джерел гілок, що використовуються для мультигілкових збірок.

Build Configuration – параметри, пов'язані з налаштуванням процесу збірки.

Scan Multibranch Pipeline Triggers – тригери, які ініціюють сканування гілок для мультигілкових пайплайнів.

Orphaned Item Strategy – стратегія для роботи з покинутими елементами (наприклад, видалення старих гілок).

Appearance – налаштування зовнішнього вигляду та інтерфейсу проєкту.

Health metrics – метрики здоров'я для оцінки стану проєкту.

Properties – додаткові властивості та параметри для кастомізації проєкту.

Multibranch Pipeline пропонує ряд важливих переваг, що роблять його ефективним у складних проєктах. Однією з головних переваг є автоматичне виявлення гілок, при якому Jenkins самостійно сканує репозиторій для виявлення нових або змінених гілок, створюючи пайплайни, якщо в гілці є Jenkinsfile. Це значно зменшує потребу в ручному налаштуванні робіт.

Ще однією важливою перевагою є ізольоване виконання завдань. Завдяки цьому кожна гілка виконує свої завдання незалежно від інших, що допомагає уникати конфліктів та зберігати чистоту середовищ.

Динамічні налаштування дозволяють кожній гілці мати свій власний Jenkinsfile, що дає змогу налаштувати специфічні пайплайни в залежності від потреб або стадій розробки. Це забезпечує високу гнучкість і адаптивність процесів.

Multibranch Pipeline також ідеально підходить для інтеграції та доставки змін із різних гілок, що дозволяє спростити процеси тестування та розгортання нових функцій, відповідно до стандартів CI/CD.

Ще однією суттєвою перевагою є підтримка паралельних робіт, що дозволяє одночасно виконувати завдання для кількох гілок. Це значно прискорює процеси розробки та тестування, що в свою чергу підвищує продуктивність команди.

Multibranch Pipeline особливо корисний у великих командах і масштабних проєктах, де активно використовуються гілки для розробки, тестування або підготовки релізів. Наприклад:

- *розробка нових функцій*. Кожна гілка для розробки функції автоматично отримує пайплайн для тестування змін;
- *тестування стабільності*. Гілки, такі як staging або release, мають індивідуальні пайплайни для тестування та перевірки готовності до релізу;
- *груповою робота*. Для кожного члена команди або команди-розробників можна створювати гілки, забезпечуючи незалежність їхніх процесів.

Multibranch Pipeline підтримує інтеграцію з вебхуками систем контролю версій, що дозволяє автоматично запускати пайплайни після змін у репозиторії. Крім того, завдяки можливості налаштування сканування репозиторію за розкладом, Jenkins автоматично виявляє нові гілки, що зменшує час на ручне управління процесами.

Multibranch Pipeline – це незамінний інструмент для організації CI/CD у проєктах із великою кількістю гілок, забезпечуючи автоматизацію, ізоляцію та гнучкість у розробці.

Кожен тип роботи в Jenkins має своє унікальне застосування, залежно від вимог проєкту:

- *Freestyle Project* – для невеликих і простих завдань;
- *Pipeline* – для побудови складних процесів CI/CD;
- *Multi-configuration Project* – для тестування різних сценаріїв і конфігурацій;
- *Folder* – для управління великою кількістю проєктів;

- *Multibranch Pipeline* – для автоматизації роботи з гілками репозиторіїв.

Jenkins є багатофункціональним інструментом, який охоплює широкий спектр завдань, пов'язаних із розробкою, тестуванням, автоматизацією та розгортанням програмного забезпечення.

Автоматична збірка проєктів

Автоматична збірка проєктів є ключовою функцією Jenkins, яка значно спрощує процес розробки та підтримки програмного забезпечення. Завдяки цьому інструменту можна виконувати такі дії:

- *збірка коду після кожного коміту в репозиторій*: Jenkins автоматично відслідковує зміни в репозиторії та запускає процес збірки, що дозволяє швидко виявляти проблеми на ранніх етапах розробки;
- *підтримка різних інструментів збірки*: Jenkins інтегрується з популярними інструментами, такими як Maven, Gradle, Ant або Make, що дозволяє адаптувати його до потреб конкретного проєкту;
- *паралельна збірка*: Jenkins підтримує паралельне виконання задач, що знижує затримки та підвищує ефективність, особливо для великих або складних проєктів, але збільшує використання ресурсів;
- *автоматизація тестування*: разом зі збіркою можна автоматично запускати тести, щоб переконатися у працездатності коду;
- *гнучке планування*: Jenkins дозволяє налаштовувати тригери для запуску збірки, включно з таймерами, ручними запусками або подіями, такими як коміти чи відкриття pull request;
- *моніторинг результатів*: Всі результати збірок доступні в реальному часі через веб-інтерфейс, де також можна переглянути логи, звіти про помилки та статистику виконання.

Завдяки цим можливостям Jenkins сприяє впровадженню безперервної інтеграції та безперервного розгортання, що забезпечує команді швидке постачання якісного продукту.

Запуск тестів

Jenkins забезпечує широкі можливості для інтеграції та автоматизації різних типів тестування, що є важливим етапом у забезпеченні якості програмного забезпечення. Платформа дозволяє налаштовувати виконання тестів на різних етапах розробки, гарантуючи своєчасне виявлення помилок і недоліків.

Нижче наведено основні типи тестів, що підтримуються Jenkins:

- *модульні тести*: використовуються для перевірки окремих функцій або модулів програми ізольовано від інших компонентів. Зазвичай інтегруються з фреймворками, такими як JUnit або TestNG;

- *інтеграційні тести*: перевіряють взаємодію між різними модулями системи, що особливо важливо для мікросервісної архітектури або складних систем із численними залежностями;
- *функціональні тести*: оцінюють відповідність поведінки програми бізнес-вимогам і користувацьким сценаріям. Для цього часто використовуються Selenium, Cucumber та інші інструменти;
- *тести продуктивності та навантаження*: аналізують стабільність і продуктивність системи під високим навантаженням. Інтегруються з такими інструментами, як JMeter, Gatling або BlazeMeter;
- *регресійні тести*: перевіряють, чи не вплинули зміни в коді на існуючий функціонал, допомагаючи уникнути повторного виникнення старих помилок;
- *безпекові тести*: оцінюють вразливості системи за допомогою інструментів, таких як OWASP ZAP або SonarQube.

Jenkins надає потужні можливості для аналізу результатів тестів завдяки інтеграції з різноманітними плагінами. Для відображення результатів модульних та інтеграційних тестів широко використовуються JUnit і TestNG. Тестування інтерфейсів і користувацьких сценаріїв можна автоматизувати за допомогою Selenium. Аналіз продуктивності та навантаження забезпечується через Performance Plugin. Для створення читабельних BDD-звітів зручно використовувати Cucumber Reports. Глибокий аналіз якості коду та виявлення потенційних проблем виконується за допомогою SonarQube. Завдяки цим інструментам Jenkins спрощує контроль якості програмного забезпечення та забезпечує наочність результатів тестування.

Також, користувачам пропонуються додаткові функції для тестування:

- *генерація звітів* – детальні звіти з результатами тестів, графіками та діаграмами допомагають легко аналізувати продуктивність і виявляти проблеми;
- *нотифікації* – Jenkins може надсилати сповіщення про результати тестів через електронну пошту, Slack або інші месенджери;
- *паралельне виконання тестів* – дозволяє скорочувати час перевірок за рахунок одночасного запуску тестових наборів на різних середовищах;
- *інтеграція з контейнерами та віртуальними машинами* – дає змогу виконувати тести в ізольованих середовищах, що забезпечує точність і повторюваність результатів.

Автоматизоване тестування в Jenkins забезпечує своєчасне виявлення дефектів і збоїв, що дозволяє оперативно реагувати на проблеми. Інструмент підтримує безперервну інтеграцію (CI) та безперервне розгортання (CD), спрощуючи процес розробки та впровадження змін. Завдяки автоматизації

тестування суттєво прискорюється перевірка програмного забезпечення, що підвищує ефективність роботи команди. Систематичний підхід до тестування сприяє покращенню якості продукту, а зручні інструменти для моніторингу й аналізу результатів допомагають швидко приймати обґрунтовані рішення.

Таким чином, Jenkins дозволяє не лише автоматизувати запуск тестів, а й інтегрувати їх у єдиний процес розробки, забезпечуючи стабільність і надійність продукту.

Аналіз якості коду

Jenkins забезпечує інтеграцію з різними інструментами для аналізу якості коду, що дозволяє автоматизувати перевірку його відповідності стандартам і найкращим практикам розробки. Інструмент SonarQube використовується для оцінки якості коду, пошуку потенційних дефектів, вразливостей і повторів, а також для виявлення проблем із технічним боргом. Він підтримує широкий спектр мов програмування та надає інтерактивні звіти з докладною інформацією про проблеми й рекомендаціями щодо їх усунення.

Для перевірки стилю коду та виявлення порушень стандартів можна використовувати інструменти, такі як Checkstyle, PMD і FindBugs (або його сучасний аналог SpotBugs). Checkstyle контролює дотримання кодових стилів, забезпечуючи узгодженість форматування. PMD аналізує код на наявність поширених помилок і потенційних вразливостей. SpotBugs фокусується на статичному аналізі коду, знаходячи логічні помилки та можливі збої.

Jenkins дозволяє інтегрувати ці інструменти в конвеєр розробки, автоматизуючи процес перевірки коду на кожному етапі. Після кожної збірки результати аналізу можна переглядати через інтерактивні звіти або графічні інтерфейси, що значно спрощує ідентифікацію й виправлення проблем.

Такий підхід допомагає підтримувати високі стандарти якості коду, що особливо важливо для великих команд розробників і довготривалих проєктів. Регулярний аналіз коду не тільки знижує ризик виникнення помилок, але й покращує підтримуваність та масштабованість системи, забезпечуючи стабільність і надійність продукту.

Розгортання на сервери

Jenkins забезпечує гнучку та надійну автоматизацію процесів розгортання програмного забезпечення, що дозволяє ефективно керувати життєвим циклом додатків – від розробки до впровадження у виробниче середовище.

Завдяки вбудованим можливостям і підтримці численних плагінів Jenkins може автоматично розгортати додатки на тестових і продакшн-серверах одразу після успішного виконання збірки та тестів. Це гарантує, що в середовищі розгортання завжди використовується стабільна та протестована версія коду.

Інтеграція з інструментами для управління контейнерами та оркестрації, такими як Docker і Kubernetes, дозволяє створювати ізольовані середовища для мікросервісів. Це спрощує управління залежностями, підвищує портативність додатків і забезпечує масштабованість. Jenkins підтримує автоматичне створення та розгортання контейнерів, їх масштабування, а також оновлення кластерів Kubernetes без простоїв.

Для роботи з хмарними платформами, такими як AWS, Azure або Google Cloud, Jenkins надає можливість інтеграції з відповідними сервісами для масштабування, резервного копіювання та управління інфраструктурою. Наприклад, можна автоматично створювати й налаштовувати віртуальні машини, розгортати сервери або оновлювати функції без використання серверів (Serverless).

Процеси розгортання можуть бути налаштовані для підтримки безперервної доставки. Це дозволяє мінімізувати затримки між розробкою й впровадженням нових функцій у продуктивне середовище, забезпечуючи постійне оновлення продукту для кінцевих користувачів.

Jenkins також підтримує блакитно-зелене розгортання і канарейковий реліз, що знижує ризики при впровадженні нових версій додатків. Блакитно-зелене розгортання дозволяє підготувати нову версію в окремому середовищі, а потім миттєво перемикає трафік на оновлений додаток. Канарейковий реліз, у свою чергу, дає змогу поступово впроваджувати зміни для невеликої групи користувачів, оцінюючи стабільність і продуктивність перед повним розгортанням.

Додатково Jenkins дозволяє автоматизувати процес відкату (rollback) у разі виявлення критичних проблем після розгортання, що забезпечує високу надійність та безпеку розгорнутих додатків.

Таким чином, Jenkins є потужним інструментом для автоматизації розгортання, забезпечуючи ефективність, масштабованість і стабільність процесів впровадження змін у сучасних IT-інфраструктурах.

Розширене застосування

Jenkins пропонує широкі можливості для автоматизації та оптимізації процесів розробки програмного забезпечення, виходячи за межі стандартних завдань збірки, тестування та розгортання. Його гнучкість і масштабованість роблять його незамінним інструментом у сучасних DevOps-процесах.

Організація випусків (release management) дозволяє автоматизувати процеси підготовки та випуску нових версій програмного забезпечення. Jenkins підтримує контроль версій, управління залежностями та інтеграцію з

репозиторіями артефактів, такими як Nexus або Artifactory. Це забезпечує централізоване зберігання збірок і спрощує їх розповсюдження.

Моніторинг і збір метрик інтегрується з такими інструментами, як Grafana, Prometheus або New Relic, що дозволяє відстежувати стан системи після розгортання, аналізувати продуктивність і оперативно реагувати на проблеми. Автоматизований моніторинг сприяє підвищенню стабільності та надійності програмного забезпечення.

Оркестрація складних процесів в Jenkins реалізується через створення багатоступінчастих пайплайнів із взаємозалежними етапами. Це дозволяє керувати складними процесами, такими як побудова, тестування, розгортання та валідація в різних середовищах. Завдяки функціоналу декларативних та скриптових пайплайнів можна налаштувати як прості, так і складні сценарії виконання завдань, що робить Jenkins ефективним інструментом для керування великими проектами.

Інтеграція з іншими DevOps-інструментами, такими як Ansible, Terraform та Puppet, розширює можливості Jenkins для автоматизації управління інфраструктурою. Ansible дозволяє автоматизувати розгортання та налаштування серверів, Terraform – управління інфраструктурою як кодом (IaC), а Puppet – централізоване керування конфігураціями. Це забезпечує узгодженість середовищ та швидке масштабування інфраструктури.

Підтримка GitOps-підходу дозволяє використовувати Jenkins для автоматизації процесів на основі змін у репозиторіях, що спрощує управління конфігураціями та забезпечує прозорість у розробці.

Обробка великих обсягів даних реалізується через інтеграцію з платформами Big Data, такими як Apache Spark або Hadoop. Це відкриває можливості для аналізу великих даних і автоматизації обчислювальних процесів.

Побудова мобільних додатків підтримується інтеграцією з платформами, такими як Fastlane та Firebase, що дозволяє автоматизувати тестування, збірку та публікацію мобільних застосунків.

Робота з контейнерами та хмарними середовищами дає можливість створювати та масштабувати додатки за допомогою Docker і Kubernetes, а також взаємодіяти з хмарними платформами AWS, Azure та Google Cloud для налаштування CI/CD-конвєсів.

Таким чином, Jenkins не лише автоматизує повторювані завдання, а й оптимізує всі етапи життєвого циклу програмного забезпечення. Його гнучкість і здатність інтегруватися з різними інструментами та технологіями дозволяють командам швидше й ефективніше досягати поставлених цілей, забезпечуючи високу якість і надійність продукту.

6.2. Роль інтеграцій (plug-in) в системі Jenkins

Система плагінів є одним із ключових аспектів, які роблять Jenkins універсальним та масштабованим інструментом для автоматизації. Плагіни дозволяють інтегрувати Jenkins із багатьма сторонніми інструментами та платформами, розширюючи його базову функціональність і адаптуючи під різні потреби проєктів.

Основні переваги використання плагінів у Jenkins полягають у можливості інтеграції з різноманітними сторонніми інструментами, що дозволяє Jenkins взаємодіяти з системами керування версіями, такими як Git, Subversion та Mercurial, а також з інструментами для управління завданнями, як-от JIRA, Redmine і Trello. Крім того, плагіни дозволяють інтегрувати Jenkins з хмарними платформами (AWS, Azure, Google Cloud Platform) та інструментами для статичного аналізу коду, наприклад, SonarQube та Checkstyle.

Плагіни також сприяють автоматизації процесів CI/CD, надаючи гнучкість у налаштуванні пайплайнів для автоматичного тестування, збірки, доставки та деплою програмного забезпечення. Це включає інтеграцію з плагінами для контейнеризації, такими як Docker і Kubernetes, а також з системами моніторингу, як Nagios і Grafana.

Іншою важливою перевагою є здатність плагінів розширювати функціональність Jenkins без необхідності змінювати основний код системи. Це включає можливість створення динамічних звітів (HTML Publisher) та підтримку різних мов програмування і середовищ.

Завдяки великій кількості плагінів Jenkins можна налаштувати для специфічних завдань і бізнес-процесів, що дозволяє ефективно застосовувати систему в різних галузях, таких як фінансовий сектор чи Інтернет речей (IoT).

Основні плагіни та їх роль

GitHub/Git

Сучасна розробка програмного забезпечення вимагає високого рівня автоматизації для забезпечення ефективності, надійності та швидкості доставки продуктів. Системи безперервної інтеграції та доставки (CI/CD) стали невідомою частиною цього процесу, забезпечуючи автоматизацію збірки, тестування, та деплою програмного забезпечення. Успішна реалізація CI/CD значною мірою залежить від використання відповідних інструментів та плагінів, які інтегруються з репозиторіями коду, системами контролю версій і хмарними платформами.

Одними з ключових елементів цього процесу є плагіни для роботи з системами контролю версій, такими як Git та GitHub. Вони забезпечують автоматизацію взаємодії з кодовою базою, дозволяють ефективно управляти

гілками, автоматизують запуск збірок після комітів і полегшують підтримку багатогалузевих пайплайнів.

На рисунку 6.8 представлена сторінка документації плагіна Git для Jenkins (<https://plugins.jenkins.io/git/>).

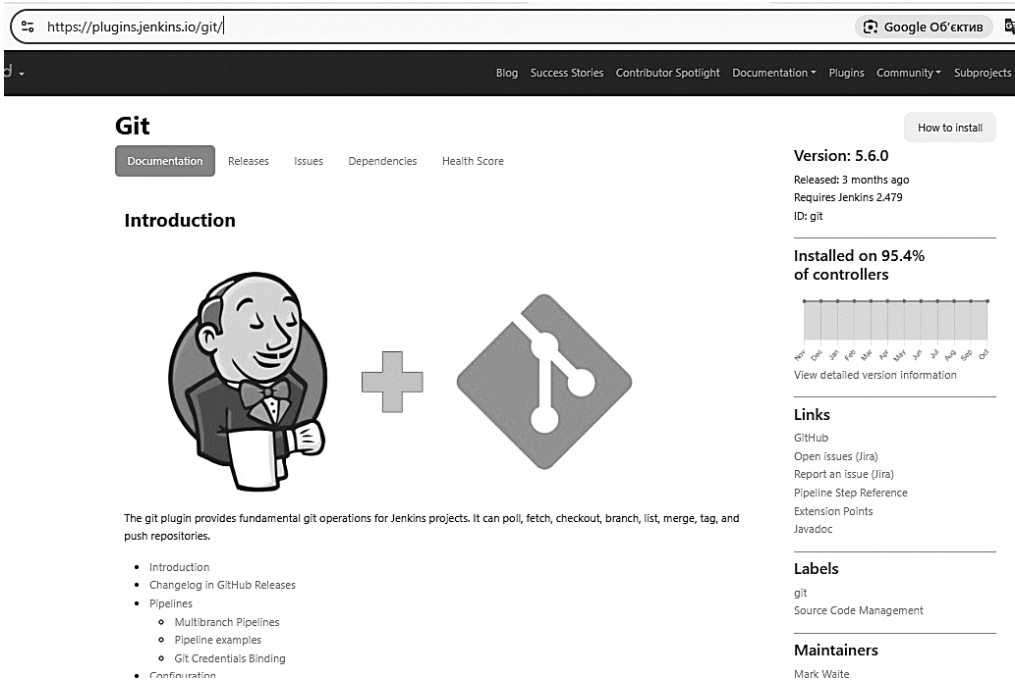


Рис. 6.8. Сторінка документації плагіна Git для Jenkins

Плагін Git забезпечує виконання основних операцій Git у проектах Jenkins. До них належать: опитування (poll), завантаження (fetch), перевірка (checkout), створення гілок (branch), перегляд (list), злиття (merge), створення міток (tag) та відправка (push) репозиторіїв.

Зокрема, плагіни GitHub/Git:

- забезпечують взаємодію між CI/CD інструментами (такими як Jenkins, GitLab CI/CD, або інші) та репозиторіями коду;
- дозволяють автоматично завантажувати останні версії коду з репозиторіїв для подальшого тестування, збірки та деплою;
- підтримують налаштування вебхуків для тригерів різних подій у репозиторії (наприклад, push, merge, pull request). Це спрощує запуск автоматичних збірок після змін у коді;
- інтегруються з пайплайнами CI, дозволяючи виконувати перевірку, збірку, тестування і деплой коду автоматично;

- спрощують управління кількома гілками, дозволяючи автоматично виявляти й запускати пайплайни для кожної гілки репозиторію, що полегшує управління великою кількістю функціональних або релізних гілок.

Pipeline

Pipeline є одним із найважливіших плагінів для організації CI/CD процесів у сучасних системах автоматизації. Його можливості дозволяють створювати складні, багатоступеневі пайплайни, налаштовуючи повний життєвий цикл розробки програмного забезпечення:

- *можливість створення складних CI/CD пайплайнів за допомогою сценаріїв на мові Groovy*: Pipeline базується на сценаріях, написаних мовою Groovy, що забезпечує високу гнучкість і кастомізацію. Це дозволяє реалізувати як прості пайплайни для невеликих проєктів, так і складні, багатоступеневі процеси для масштабних рішень. Завдяки цьому, пайплайни можуть включати специфічні правила, умови виконання, інтеграцію з іншими сервісами та динамічну адаптацію до змін.
- *підтримка декларативного і скриптового підходу для налаштування процесів*:
 - *декларативний підхід*: використовується для простих і структурованих описів процесів. Завдяки зрозумілому синтаксису, підходить для розробників будь-якого рівня досвіду;
 - *скриптовий підхід*: забезпечує більше можливостей для реалізації складної логіки та кастомізації. Використовується в проєктах, які потребують високого рівня контролю за процесами;
- *управління всім циклом життєвого циклу розробки*: Pipeline охоплює всі етапи життєвого циклу програмного забезпечення:
 - *збірка (Build)*: включає компіляцію коду, створення пакетів і артефактів;
 - *тестування (Test)*: автоматичний запуск різних типів тестів, таких як юніт-тести, інтеграційні, функціональні або performance-тести;
 - *розгортання (Deploy)*: автоматизація доставки програмного забезпечення на різні середовища, включаючи staging та production;
 - *моніторинг (Monitor)*: інтеграція з системами моніторингу для відстеження стабільності, продуктивності та виявлення проблем після релізу;
- *гнучкість та масштабованість*: завдяки структурі Pipeline, процеси можна легко розширювати та адаптувати до нових вимог, включаючи

роботу з мікросервісами, хмарними середовищами або контейнеризацією (Docker, Kubernetes);

- *інтеграція з іншими плагінами та сервісами*: Pipeline підтримує інтеграцію з численними плагінами, такими як GitHub, GitLab, Slack, Jenkins Artifactory, AWS, Azure DevOps, JUnit, Selenium та інші. Це дозволяє створювати комплексні процеси CI/CD з єдиним управлінням;
- *можливість паралельного виконання*: Pipeline дозволяє виконувати етапи паралельно, що суттєво зменшує час виконання складних процесів і підвищує продуктивність;
- *розширені можливості налагодження та логування*: плагін забезпечує детальний журнал виконання, який допомагає швидко виявляти та виправляти помилки.

На рисунку 6.9 показано сторінку документації плагіна Pipeline для Jenkins (<https://plugins.jenkins.io/workflow-aggregator/>).

Pipeline є основою для побудови надійних і ефективних процесів автоматизації, що дозволяє командам розробників зосередитися на створенні якісного програмного забезпечення, зменшуючи рутинні завдання та забезпечуючи швидкість і стабільність доставки продуктів.

The screenshot displays the Jenkins Plugin Catalog page for the 'Pipeline' plugin. The page is titled 'Pipeline' and includes tabs for 'Documentation', 'Releases', 'Issues', 'Dependencies', and 'Health Score'. The current version is '600.vb_57cdd26fdd7' with 246k installs. A bar chart shows the installation status across various Jenkins controllers, with a note that it is installed on 88.7% of controllers. The page also provides links to the plugin's documentation, GitHub repository, and other resources.

Рис. 6.9. Сторінка документації плагіна Pipeline для Jenkins

Docker

Docker-плагіни відіграють ключову роль у побудові сучасних процесів CI/CD, забезпечуючи інтеграцію Jenkins із Docker для роботи з контейнерами та спрощуючи управління контейнеризованими середовищами.

- *інтеграція Jenkins із Docker*: Docker-плагіни дозволяють використовувати контейнеризацію як основний підхід до організації середовища виконання. Це забезпечує можливість інтеграції з Docker Engine для запуску контейнерів безпосередньо із Jenkins, що значно полегшує управління залежностями, середовищами та артефактами;
- *запуск завдань у ізольованих контейнерах*:
 - завдяки контейнеризації, кожне завдання (job) виконується в ізольованому середовищі, яке має всі необхідні залежності. Це забезпечує стабільність виконання та уникнення конфліктів між середовищами, які можуть виникнути під час роботи з різними проєктами;
 - ізоляція також допомагає зберігати чистоту середовища, уникаючи впливу попередніх завдань на поточне;
- *відтворюваність середовища виконання*: використання контейнерів дозволяє створювати однакові середовища для розробки, тестування та продакшену, що мінімізує ризики, пов'язані з розбіжностями між середовищами;
- *створення Docker-образів*: Docker-плагіни дозволяють автоматизувати процес створення Docker-образів безпосередньо в Jenkins. Це включає:
 - написання та використання Dockerfile;
 - створення образів із вихідного коду;
 - тегування образів для зручного управління версіями;
- *тестування в контейнерах*: контейнеризація дозволяє запускати автоматичні тести в контрольованих середовищах. Це забезпечує узгодженість результатів і дозволяє швидко перевіряти код у різних конфігураціях;
- *розгортання мікросервісів*: Docker активно використовується для роботи з мікросервісною архітектурою. Завдяки плагінам Jenkins можна:
 - автоматично розгорнути мікросервіси у вигляді контейнерів;
 - інтегрувати процес із оркестраторами контейнерів, такими як Kubernetes або Docker Swarm, для масштабування та управління сервісами;

- *інтеграція з Docker Hub і приватними реєстрами:*
 - Jenkins дозволяє підключатися до Docker Hub або інших реєстрів контейнерів для завантаження образів, створених у процесі CI/CD, або отримання базових образів для створення нових контейнерів;
 - підтримується автентифікація і управління доступами до приватних реєстрів;
- *підтримка динамічних агентів:* використовуючи Docker, Jenkins може створювати динамічні агенти (slaves), які запускаються в контейнерах. Це забезпечує масштабованість і оптимізацію ресурсів, дозволяючи запускати агенти тільки тоді, коли вони необхідні;
- *економія ресурсів і масштабованість:* контейнеризація забезпечує ефективне використання ресурсів, оскільки контейнери легші за віртуальні машини. Це дозволяє запускати більше процесів на тому ж апаратному забезпеченні, а також легко масштабувати інфраструктуру.

У Jenkins існує декілька плагінів для інтеграції з Docker, які допомагають автоматизувати процеси побудови, тестування та розгортання застосунків. Основні з них: Docker Plugin, Docker Pipeline Plugin, Kubernetes Plugin, Docker Build Step Plugin, CloudBees Docker Hub Notification Plugin.

На рисунку 6.10 показана сторінка плагіна Docker для Jenkins (<https://plugins.jenkins.io/docker-plugin/>).

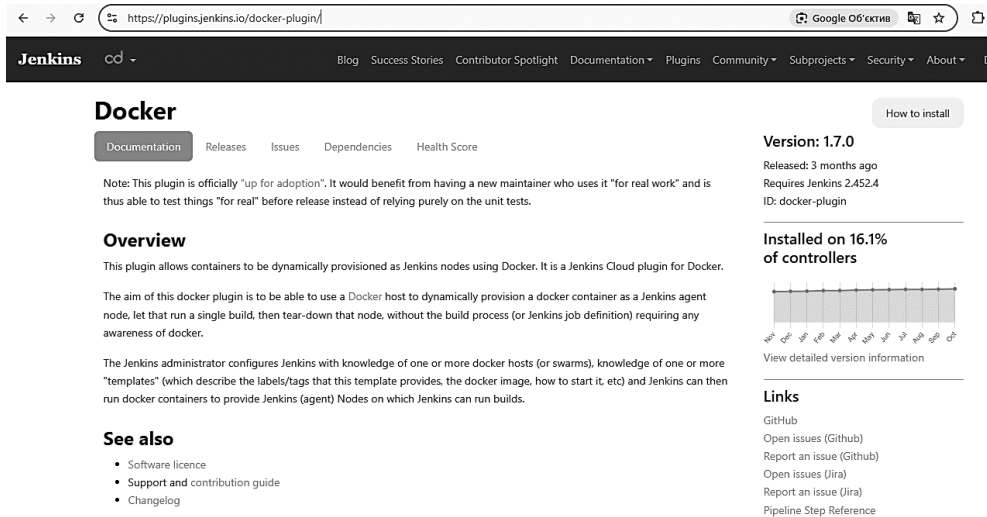


Рис. 6.10. Сторінка плагіна Docker для Jenkins

Плагін Docker дозволяє динамічно надавати контейнери як вузли (агенти) Jenkins із використанням Docker.

Адміністратор Jenkins може налаштувати використання одного чи кількох Docker-хостів (або swarm-кластерів), а також шаблонів, які описують, як запускати Docker-контейнери.

Завдання Jenkins можуть запускатися у цих контейнерах безпосередньо, що спрощує управління середовищем виконання.

Docker-плагіни є незамінним інструментом для команд, які працюють з контейнеризацією. Вони сприяють автоматизації, підвищують стабільність і повторюваність процесів, а також спрощують роботу з мікросервісною архітектурою.

JUnit

JUnit-плагін відіграє ключову роль у забезпеченні якості та стабільності програмного забезпечення, інтегруючи результати модульного тестування в процес CI/CD. Він надає широкі можливості для аналізу тестів, генерації звітів і моніторингу стану коду протягом усього життєвого циклу розробки:

- *інтеграція аналізу результатів модульних тестів:*
 - плагін автоматично обробляє результати тестів, які виконуються за допомогою JUnit, і додає їх до звітів Jenkins;
 - це дозволяє командам розробників бачити, які тести пройшли успішно, а які зазнали невдач, безпосередньо у веб-інтерфейсі Jenkins;
 - підтримується робота з XML-звітами JUnit, що робить інтеграцію легкою та універсальною;
- *генерація звітів про проходження тестів:*
 - звіти включають детальну інформацію про кожен тест: час виконання, успішність, помилки або винятки;
 - графічні інтерфейси Jenkins показують зведення, тренди проходження тестів і дозволяють аналізувати прогрес у їхній стабільності;
 - ці звіти допомагають визначати критичні точки, які потребують додаткової уваги розробників;
- *відслідковування успішності або невдач тестів:*
 - Jenkins автоматично позначає збірки як «успішні» чи «провальні» на основі результатів тестів, що дозволяє оперативно реагувати на проблеми;
 - плагін також може сигналізувати про збільшення кількості провалених тестів, що є індикатором можливих регресій у коді;
- *широке використання для перевірки якості коду:*

- JUnit-плагін дозволяє забезпечувати високий рівень тестування, охоплюючи ключові компоненти додатків;
- використовується як для модульного тестування, так і для інтеграційного тестування, забезпечуючи впевненість у стабільності системи;
- *підтримка автоматизації тестування в CI/CD процесах:*
 - плагін автоматично інтегрується в пайплайни Jenkins, дозволяючи запускати тести як окремий етап у процесі CI/CD;
 - забезпечує видимість впливу змін у коді на стабільність, дозволяючи розробникам отримувати швидкий зворотний зв'язок;
- *інтеграція з іншими інструментами:*
 - JUnit-плагін може працювати разом із іншими плагінами, такими як Cobertura (для перевірки покриття коду), SonarQube (для аналізу якості) або Allure (для детальних звітів);
 - це створює повноцінну екосистему для забезпечення високої якості програмного забезпечення;
- *моніторинг стабільності коду з часом:*
 - плагін дозволяє будувати тренди проходження тестів, що допомагає відстежувати прогрес або погіршення стабільності в часі;
 - такі тренди допомагають приймати рішення щодо пріоритизації задач і фокусування на критичних проблемах;
- *простота налаштування та використання:*
 - JUnit-плагін легко інтегрується у проекти, які вже використовують JUnit для тестування;
 - Налаштування займає мінімум часу, а результати тестів доступні відразу після виконання.

У Jenkins існує декілька плагінів, які інтегруються з JUnit для обробки та відображення результатів тестування. Основні з них: JUnit Plugin, JUnit Attachments Plugin.

На рисунку 6.11 показана сторінка плагіна JUnit Plugin для Jenkins (<https://plugins.jenkins.io/junit/>).

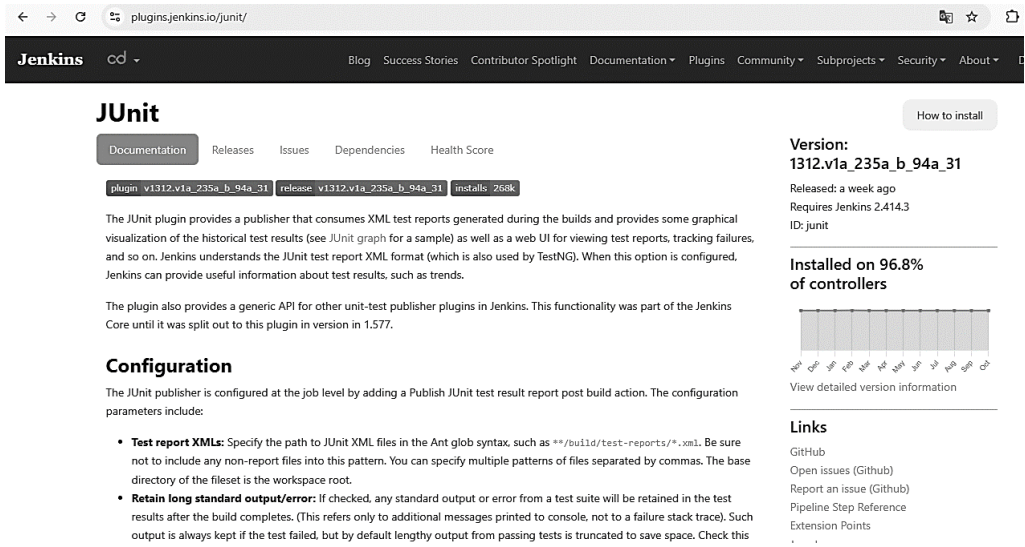


Рис. 6.11. Сторінка плагіна JUnit для Jenkins

Плагін JUnit забезпечує обробку XML-звітів тестів, згенерованих під час збірок, та надає графічну візуалізацію історичних результатів тестування. Цей плагін також надає веб-інтерфейс для перегляду звітів тестів та відстеження помилок.

JUnit-плагін є незамінним інструментом для команд розробників, які прагнуть забезпечити високу якість програмного забезпечення, покращити процеси тестування та мінімізувати ризики, пов'язані з помилками або регресіями.

Blue Ocean

Blue Ocean – це плагін, створений для значного покращення користувацького досвіду в Jenkins. Його завданням є забезпечення сучасного, інтуїтивно зрозумілого інтерфейсу, який робить роботу з пайплайнами і CI/CD процесами більш візуально привабливою, зрозумілою і продуктивною:

- сучасний та зручний інтерфейс користувача:
 - Blue Ocean надає новий погляд на традиційний інтерфейс Jenkins, орієнтований на простоту та зручність;
 - його дизайн базується на сучасних принципах UX/UI, що дозволяє розробникам і DevOps-інженерам легко знаходити необхідну інформацію;
 - ідеально підходить як для новачків, так і для досвідчених користувачів завдяки мінімалістичному підходу до відображення інформації;

- *покращена візуалізація пайплайнів і процесів CI/CD:*
 - плагін пропонує графічне представлення пайплайнів, що дозволяє бачити всі етапи виконання у вигляді блок-схеми;
 - кожен етап відображається в реальному часі з кольоровими індикаторами успішності (зелений для успішних етапів, червоний для невдалих);
 - це дозволяє швидко оцінити стан пайплайну без необхідності глибокого аналізу журналів виконання;
- *інтерактивна взаємодія з пайплайнами:*
 - користувачі можуть клікати на будь-який етап пайплайну для отримання детальної інформації про виконання, включаючи час виконання, лог виконання, а також виявлення помилок;
 - Blue Ocean дозволяє зупиняти, перезапускати або пропускати окремі етапи пайплайну безпосередньо з інтерфейсу;
- *підтримка автоматизації з багатьма гілками (Multibranch Pipelines):*
 - Blue Ocean чудово працює з багатогілковими пайплайнами, автоматично відображаючи стан кожної гілки та її тестів;
 - це спрощує управління складними проектами, де кожна гілка має власні завдання CI/CD;
- *інтеграція з іншими плагінами та інструментами:*
 - Blue Ocean підтримує інтеграцію з популярними інструментами, такими як GitHub, Bitbucket і GitLab, забезпечуючи зручну навігацію між репозиторіями та пайплайнами;
 - він також дозволяє отримувати сповіщення про стан збірок через інтеграцію зі Slack або іншими месенджерами;
- *зручний аналіз і діагностика проблем:*
 - користувачі можуть швидко аналізувати помилки, які виникли на будь-якому етапі пайплайну, завдяки інтерактивним логам і підсвічуванню проблемних місць;
 - плагін також надає інформацію про відповідальні коміти та зміни, пов'язані з невдалими збірками, що прискорює пошук винуватців і виправлення проблем;
- *полегшення співпраці в командах:*
 - Blue Ocean пропонує інтуїтивно зрозумілу візуалізацію, яка сприяє кращому розумінню процесів CI/CD для всіх учасників команди, незалежно від їхньої технічної підготовки;
 - завдяки цьому навіть нетехнічні учасники команди, такі як менеджери проєктів, можуть легко відстежувати прогрес розробки;
- *простота налаштування:*

- Blue Ocean дозволяє швидко налаштувати та інтегрувати пайплайни через зручні майстри налаштувань;
- навіть для складних процесів плагін зменшує потребу в детальній документації, завдяки чому користувачі можуть легко зрозуміти структуру та логіку роботи пайплайнів.

На рисунку 6.12 показана сторінка плагіна Blue Ocean для Jenkins (<https://plugins.jenkins.io/blueocean/>).

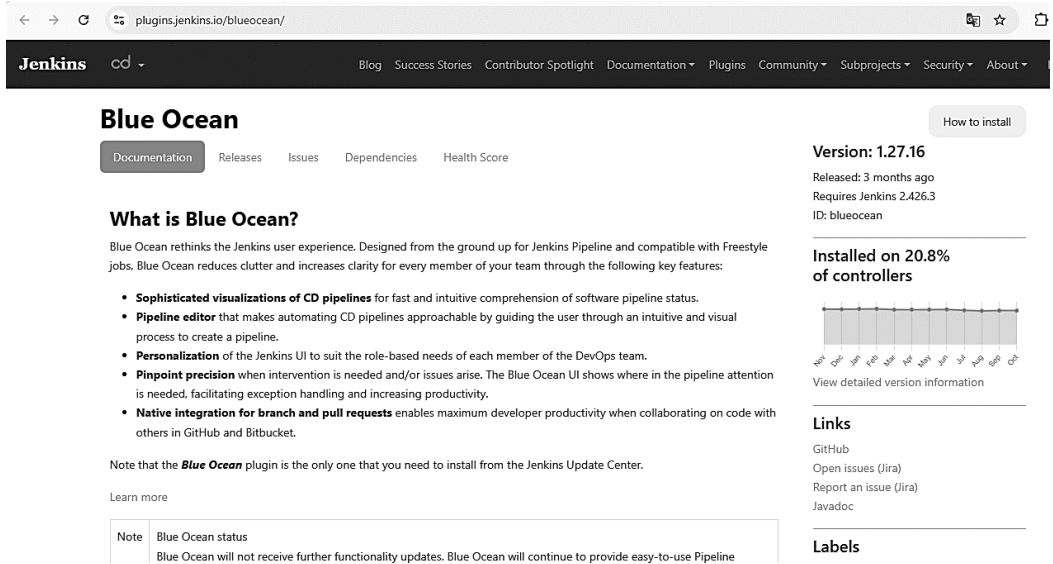


Рис. 6.12. Сторінка плагіна Blue Ocean для Jenkins

Blue Ocean є революційним рішенням для покращення зручності використання Jenkins. Він значно підвищує продуктивність команд, скорочує час аналізу проблем і робить роботу з пайплайнами більш доступною для широкого кола користувачів.

Інші корисні плагіни

SonarQube Scanner

Використовується для статичного аналізу коду з метою виявлення помилок, технічного боргу, вразливостей безпеки та порушень стандартів кодування.

Забезпечує детальний звіт про якість коду, включаючи показники покриття тестами, складність функцій та дублювання коду.

Інтеграція з Jenkins дозволяє автоматично запускати аналіз після збірки і відображати результати безпосередньо в інтерфейсі.

Дозволяє визначати порогові значення для якості, після перевищення яких збірка вважається невдалою.

Slack

Забезпечує інтеграцію з месенджером Slack, дозволяючи відправляти повідомлення про стан збірок, виконання пайплайнів чи помилки.

Полегшує комунікацію в командах, забезпечуючи оперативне інформування про ключові події в процесі CI/CD.

Підтримує налаштування різних типів повідомлень: успіх збірки, провал, початок виконання завдання тощо.

Повідомлення можуть включати посилання на логи, звіти чи артефакти, що прискорює аналіз і виправлення проблем.

Email Extension

Дозволяє відправляти електронні листи з детальною інформацією про результати виконання завдань або пайплайнів.

Надає можливість налаштування шаблонів повідомлень: можна вказати, які саме дані включати (логи, помилки, час виконання).

Підтримує гнучке налаштування умов для відправки (наприклад, тільки у випадку провалу збірки).

Полегшує інформування зацікавлених осіб безпосередньо з Jenkins.

Kubernetes

Використовується для інтеграції Jenkins із Kubernetes для управління контейнеризованими середовищами.

Дозволяє запускати завдання Jenkins у динамічних агентах, створених у кластерах Kubernetes.

Полегшує автоматизацію розгортання додатків у кластері Kubernetes, забезпечуючи масштабованість і гнучкість.

Підтримує використання Helm-чартів, управління секретами та моніторинг розгортання через Jenkins.

Artifact Repository (Nexus, Artifactory)

Використовується для збереження та управління артефактами збірок, такими як бібліотеки, Docker-образи або ZIP-архіви.

Плагіни для Nexus або Artifactory дозволяють інтегрувати Jenkins із репозиторіями артефактів, забезпечуючи їхню доступність для подальшого використання.

Автоматизує процес публікації артефактів після успішної збірки.

Підтримує контроль версій артефактів, забезпечуючи відстежуваність і можливість повернення до попередніх версій.

Ці плагіни дозволяють зробити Jenkins більш потужним, адаптуючи його до потреб різних команд і проєктів, покращуючи якість, продуктивність і зручність роботи з CI/CD процесами.

AWS EC2

Забезпечує інтеграцію з Amazon EC2, дозволяючи автоматично створювати, масштабувати та керувати Jenkins-агентами у хмарі AWS.

Полегшує управління ресурсами, динамічно додаючи чи видаляючи build-агенти залежно від навантаження, що оптимізує витрати на інфраструктуру.

Підтримує запуск Jenkins-агентів на різних типах EC2-інстансів, використання кастомних AMI, а також автоматичне завершення невикористаних інстансів.

Дозволяє інтегруватися з AWS IAM для керування доступом, а також з іншими сервісами AWS, такими як S3 та CloudWatch, для моніторингу та збереження артефактів.

Плагіни роблять Jenkins універсальним інструментом, який можна налаштувати для будь-яких потреб у DevOps, розробці та тестуванні. Вони забезпечують просту інтеграцію з популярними інструментами й платформами, а також дозволяють створювати масштабовані та адаптивні процеси автоматизації. Завдяки цьому Jenkins є вибором номер один для багатьох команд розробників та DevOps-інженерів.

6.3. Приклад організації роботи. Автоматизація збірки проєкту

Для автоматизації процесу збірки, тестування та аналізу Java-проєкту в Jenkins можна реалізувати наступний сценарій.

1. Налаштування середовища

Перед створенням роботи необхідно налаштувати Jenkins і підготувати середовище:

- *встановлення Jenkins*: потрібно завантажити й встановити Jenkins на сервер (локальний чи хмарний);
- *встановлення плагінів*:
 - Git: для інтеграції з репозиторієм;
 - Maven Integration: для виконання команд Maven;
 - JUnit: для збору й аналізу результатів тестування;
- *підключення репозиторію Git*: слід додати URL репозиторію Java-проєкту (наприклад, GitHub чи GitLab) та налаштувати аутентифікацію (ключ SSH або токен).

- *встановлення Java та Maven*: варто переконатися, що Jenkins має доступ до коректних версій Java Development Kit (JDK) та Apache Maven.

2. Створення роботи

- *тип роботи*: потрібно обрати Freestyle Project, що підходить для базових задач автоматизації;
- *параметри роботи*:
 - у секції Source Code Management вказати репозиторій Git і налаштувати гілку, яку необхідно збирати (наприклад, main);
 - у секції Build потрібно додати виконання команд Maven:

clean install

Ця команда очищає попередні артефакти та компілює проєкт, збираючи його у виконуваний JAR-файл.

- *тригери запуску*:
 - потрібно налаштувати автоматичний запуск збірки після кожного коміту за допомогою вебхуків із репозиторію;
 - можна додати тригер на основі розкладу, наприклад ***H/5 * * **** запускає роботу кожні 5 хвилин.

3. Запуск тестів

Після успішної збірки можна виконати автоматичне тестування:

- у розділі *Post-build Actions* налаштувати виконання тестів за допомогою JUnit;
- вказати шлях до звітів тестування, наприклад:
target/surefire-reports/*.xml
- Jenkins автоматично збере результати тестів, запущених Maven, і згенерує детальний звіт.

4. Результати

- *публікація звітів*:
 - Jenkins автоматично збере звіти про тести, збірку та успішність виконання; користувач може переглянути історію збірок, включаючи тривалість, успішність і збої;
- *графічне представлення*: можна використати плагіни, такі як Test Results Analyzer, для візуалізації результатів тестів у вигляді графіків;
- *повідомлення*: для сповіщення команди про успішні чи невдалі збірки можна інтегрувати Jenkins із Slack чи електронною поштою.

Якщо проєкт має складнішу структуру, можна:

- використовувати Pipeline для організації багатоступеневих процесів (збірка, тести, аналіз якості коду, розгортання);
- інтегрувати аналіз якості коду через плагін SonarQube;
- використовувати Docker для запуску ізольованого середовища збірки.

Цей процес автоматизації допомагає мінімізувати помилки, прискорює цикл розробки та забезпечує стабільну інтеграцію й доставку програмного забезпечення.

6.4. Модель (master-slave) в контексті системи Jenkins

Архітектура Master-Slave (нині відома як Controller-Agent) є ключовою для забезпечення масштабованості та ефективності роботи Jenkins у великих проєктах. Вона дозволяє розподіляти завдання між кількома вузлами, підтримуючи баланс навантаження та гнучкість у використанні різних середовищ.

Основні компоненти архітектури:

- Master (Controller):
 - центральний сервер, який виконує такі функції:
 - ✓ керує загальним процесом роботи Jenkins;
 - ✓ координує виконання завдань між агентами;
 - ✓ зберігає конфігурації проєктів, плагінів і журналів;
 - ✓ надає вебінтерфейс для взаємодії з користувачами;
 - Master не завжди виконує завдання самостійно, оскільки основна мета – делегувати обчислення агентам;
- Slave (Agent):
 - вузли, що виконують завдання, передані Master;
 - забезпечують виконання збірок, тестів, аналізу коду та інших операцій;
 - можуть бути встановлені на різних платформах (Linux, Windows, Mac) і навіть у хмарних середовищах;
 - взаємодіють із Master через протоколи, такі як SSH, JNLP або WebSocket.

Модель Master-Slave забезпечує високу масштабованість завдяки можливості додавання нових агентів, що дозволяє одночасно виконувати більшу кількість завдань. Це рішення ефективно адаптується до обробки великих обсягів задач, роблячи систему зручною для великих проєктів.

Розподіл навантаження між агентами допомагає зменшити тиск на центральний сервер (Master), забезпечуючи стабільну роботу навіть у проєктах

із сотнями щоденних збірок. Завдяки цьому досягається рівномірний розподіл ресурсів і мінімізується ризик перевантаження.

Однією з ключових переваг є підтримка різних середовищ. Агенти можуть працювати в різних операційних системах із унікальними конфігураціями, що дозволяє виконувати тести на таких платформах, як Linux, Windows і macOS, паралельно.

Модель також сприяє підвищенню ефективності завдяки розподілу завдань між агентами. Це значно прискорює виконання складних багатоступеневих процесів, зменшуючи час розробки й тестування.

Ізольованість забезпечується завдяки виконанню завдань на окремих вузлах. Такий підхід знижує ризик впливу одного завдання на інше, сприяючи стабільній роботі системи в цілому.

Модель має значну залежність від центрального сервера (Master). У разі його перевантаження або збою вся система може зупинитися, що створює критичну точку відмови.

Налаштування системи також є досить складним процесом. Підключення агентів і забезпечення їхньої безперебійної роботи вимагає додаткових зусиль із налаштування, а також постійного моніторингу для підтримки стабільності.

Ще одним обмеженням є високий мережевий трафік між Master і Slave. У великих інфраструктурах це може стати серйозною проблемою, впливаючи на продуктивність і швидкість передачі даних.

Реалізація моделі Master-Slave

1) Налаштування Master (Controller):

- потрібно встановити Jenkins на сервері, який виконуватиме роль Master;
- налаштувати плагіни для взаємодії з агентами (наприклад, SSH Build Agents Plugin);

2) Додавання агентів (Slave):

- обрати метод підключення агентів:
 - SSH: підключення до віддалених серверів через Secure Shell;
 - JNLP: використовується для запуску агентів у середовищах, де немає прямого доступу до Master;
 - Cloud Agents: автоматичне створення й підключення агентів у хмарі (наприклад, через AWS, Azure чи Kubernetes);
- встановити Jenkins Agent на вузол і вкажіть шлях для підключення до Master.

3) Розподіл завдань:

- налаштувати виконавців (executors) на кожному агенті – це кількість одночасно виконуваних завдань;

- використовувати мітки (labels) для призначення завдань агентам із певними конфігураціями (наприклад, «Linux», «Windows», «Docker»).

Модель Master-Slave забезпечує Jenkins високу масштабованість, гнучкість і продуктивність, дозволяючи ефективно виконувати великі обсяги завдань і підтримувати різноманітні середовища. Вона є ідеальним рішенням для великих проєктів і DevOps-команд, що працюють у розподілених системах.

6.5. Pipeline: приклади роботи

Pipeline в Jenkins – це спосіб автоматизації всього процесу розробки програмного забезпечення за допомогою сценаріїв на мові Groovy. Використовуючи Pipeline, можна описати весь процес CI/CD у вигляді коду, що зберігається в репозиторії разом із вихідним кодом проєкту.

У Jenkins Pipeline існує в двох основних формах: декларативний (Declarative) і скриптовий (Scripted). Обидва підходи забезпечують автоматизацію процесів CI/CD, проте кожен із них має свої переваги та призначення.

Декларативний Pipeline

Декларативний підхід є більш структурованим і зручним для використання. Завдяки простій та зрозумілій синтаксичній структурі, він ідеально підходить для стандартних процесів автоматизації.

Основні характеристики:

- чітка структура з обов'язковими секціями (pipeline, agent, stages);
- менше коду, що зменшує ризик помилок;
- підходить для новачків і команд, які дотримуються стандартних практик;
- містить вбудовані механізми обробки помилок і постпроцесів (post).

Розглянемо приклад простого Declarative Pipeline та призначення його елементів:

```
groovy

pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                echo 'Building the project...'
                sh 'mvn clean install'
            }
        }
    }
}
```

pipeline: оголошує основний конвеєр Jenkins.

agent any: виконує завдання на будь-якому доступному агенті Jenkins.

stages: визначає етапи роботи конвеєра.

stage('Build'): це етап збірки проєкту.

steps: містить команди, які виконуються під час етапу.

echo 'Building the project...': виводить повідомлення "Building the project..." у консоль Jenkins для інформування.

sh 'mvn clean install': виконує команду mvn clean install, яка очищає попередню збірку проєкту та створює нову збірку (залежності, компіляція, пакування).

Цей простий приклад корисний для Java-проєктів, що використовують Maven для автоматизації процесу збірки.

Скриптовий Pipeline

Скриптовий підхід базується на Groovy, що робить його більш гнучким та універсальним. Він дозволяє створювати складні, динамічні сценарії, які можуть адаптуватися до різних умов.

Основні характеристики є наступними:

- використовує Groovy для написання коду, що надає повну свободу дій;
- дозволяє створювати динамічні процеси з умовами, циклами та складною логікою;
- підходить для досвідчених розробників, які мають потребу у складних, нестандартних сценаріях.

Нижче наведено приклад простого Scripted Pipeline:

```

groovy

node {
    stage('Checkout') {
        checkout scm
    }
    stage('Build') {
        sh 'mvn clean install'
    }
    if (env.BRANCH_NAME == 'main') {
        stage('Deploy') {
            echo 'Deploying to production...'
        }
    }
}
}

```

node: визначає середовище (агент) Jenkins, на якому виконуватимуться завдання;

stage('Checkout'):

- етап отримання коду з системи керування версіями (SCM, наприклад, Git);
- використовується команда `checkout scm`, яка автоматично отримує останню версію коду з репозиторію, налаштованого для цього завдання.

stage('Build'): етап збірки проєкту;

sh 'mvn clean install' очищує попередні артефакти та виконує процес збірки за допомогою Maven.

if (env.BRANCH_NAME == 'main'): умова, яка перевіряє, чи активна гілка називається `main`. Якщо умова виконується, відбувається етап деплою.

stage('Deploy'): етап деплою в продакшн. Виводить повідомлення "Deploying to production..." у консоль Jenkins для підтвердження.

У таблиці 6.1 наведено порівняння ключових характеристик декларативного та скриптового підходів.

Декларативний Pipeline підходить для автоматизації стандартних CI/CD процесів, таких як збірка, тестування та розгортання. Наприклад, розробка вебсайтів або мобільних додатків із типовими середовищами.

Скриптовий Pipeline доцільний для складних сценаріїв, де необхідно обробляти динамічні параметри, виконувати умови або взаємодіяти з кількома інструментами та середовищами. Наприклад, автоматизація для великих розподілених систем чи багатокомпонентних додатків.

Таблиця 6.1. Порівняння декларативного та скриптового підходів

Характеристика	Декларативний Pipeline	Скриптовий Pipeline
Простота	Простий та зручний у використанні	Потребує знань Groovy
Гнучкість	Обмежена чіткою структурою	Максимальна гнучкість
Придатність	Для стандартних та простих процесів	Для складних і динамічних задач
Обробка помилок	Вбудована в секцію post	Реалізується вручну через try-catch
Навчання	Легко вивчити навіть новачкам	Вимагає більшого досвіду

Обидва типи Pipeline є ефективними інструментами для автоматизації в Jenkins. Вибір між ними залежить від складності завдання, рівня підготовки команди та вимог до гнучкості. У більшості випадків декларативний підхід є оптимальним завдяки простоті, але скриптовий підхід залишається незамінним для складних процесів.

6.6. Підключення оркестрації до системи контролю версіями

Підключення оркестрації до системи контролю версіями (наприклад, Git або GitHub) дозволяє Jenkins автоматично реагувати на зміни в коді, що дає можливість виконувати процеси CI/CD (безперервної інтеграції та доставки) на кожен коміт або пул-реквест. Це є важливим аспектом автоматизації процесів розробки, оскільки дозволяє підтримувати актуальність тестів, збірки та розгортання без участі розробників у кожному кроці.

Підключення Jenkins до Git дозволяє реалізувати безперервну інтеграцію та автоматизацію процесів розробки. Однією з основних переваг є можливість автоматичної збірки після кожного коміту, що забезпечує оперативну перевірку коду на помилки та відповідність вимогам.

Інтеграція з гілками дозволяє створювати окремі пайплайни для різних частин проекту, наприклад, для основної гілки чи тестових гілок. Це дає змогу виконувати тести або інші операції лише для вибраних гілок, оптимізуючи процес розробки.

Jenkins також може автоматично запускати тести після кожного коміту, що допомагає швидко виявляти проблеми та підтримувати високу якість коду.

Ще однією важливою можливістю є автоматичне розгортання. Завдяки інтеграції з Git, нові зміни можуть автоматично доставлятися на сервери для

тестування або у продуктивне середовище, що робить процес безперервної доставки ефективним та швидким.

Кроки підключення Jenkins до Git/GitHub:

1) встановлення плагіна Git: щоб підключити Jenkins до системи контролю версіями, необхідно спершу встановити плагін Git. Для цього:

- перейдіть у Jenkins → Manage Jenkins → Manage Plugins;
- у вкладці Available знайдіть Git Plugin і встановіть його;
- після установки плагін Git дозволяє Jenkins виконувати операції з репозиторіями Git, такі як клонування, перевірка комітів і гілок.

2) Додавання URL-адреси репозиторію в налаштуваннях Jenkins: після установки плагіна необхідно вказати URL репозиторію Git або GitHub у конфігурації Jenkins:

- створіть нову задачу в Jenkins або відкрийте вже існуючу;
- у налаштуваннях проекту знайдіть розділ Source Code Management;
- виберіть Git і вкажіть URL вашого репозиторію. Це може бути HTTPS або SSH шлях до репозиторію. Для GitHub зазвичай використовують HTTPS-адресу репозиторію, наприклад:

<https://github.com/username/repository.git>

- додайте параметри доступу, якщо це необхідно (наприклад, SSH-ключ або облікові дані для доступу).

3) Налаштування вебхуків у GitHub для автоматичного запуску робіт: для автоматичного запуску задач Jenkins при зміні в репозиторії, необхідно налаштувати вебхук в GitHub:

- перейдіть у ваш репозиторій на GitHub;
- виберіть Settings → Webhooks → Add webhook;
- в полі Payload URL введіть адресу вашого Jenkins сервера з додаванням шляху до вебхука, наприклад:

<http://your-jenkins-url/github-webhook/>

- у полі Content type виберіть application/json;
- у секції Which events would you like to trigger this webhook? виберіть Just the push event або Let me select individual events, щоб вибрати конкретні події, які повинні запускати пайплайн;
- після цього GitHub почне надсилати запити на ваш Jenkins сервер кожного разу, коли буде здійснено коміт в репозиторії, що спричиняє автоматичний запуск відповідної роботи.

Підключення Jenkins до системи контролю версіями є необхідним етапом для реалізації CI/CD процесів. Це дозволяє автоматизувати весь цикл розробки: від коміту до розгортання. Завдяки налаштуванню вебхуків і інтеграції з

Git/GitHub можна забезпечити безперервне тестування, збірку та доставку, що значно підвищує ефективність та якість розробки програмного забезпечення.

Можна зробити висновок, що Jenkins є потужним і надзвичайно гнучким інструментом, який забезпечує повний цикл автоматизації від розробки до розгортання. Завдяки своїм можливостям автоматизувати безліч етапів життєвого циклу програмного забезпечення, Jenkins став незамінним інструментом для реалізації практик безперервної інтеграції та доставки (CI/CD). Його основні переваги включають:

- *гнучкість*: Jenkins підтримує різноманітні типи робіт, таких як freestyle project, pipeline, а також можливість налаштування процесів за допомогою мови Groovy, що дозволяє адаптувати інструмент до специфічних потреб команди чи проєкту;
- *розширюваність*: завдяки багатому набору плагінів, Jenkins може інтегруватися з численними інструментами та технологіями, включаючи Git, Docker, Kubernetes, різні системи контролю версій і багато інших. Це дозволяє створювати висококонфігуровані і масштабовані рішення, що підходять для різних типів проєктів і середовищ;
- *масштабованість*: Завдяки архітектурі Master-Slave (або Controller-Agent), Jenkins здатний ефективно розподіляти навантаження і забезпечувати паралельну обробку задач, що дозволяє обробляти великий обсяг збірок, тестів і розгортань без перевантаження серверів;
- *підтримка сучасних DevOps практик*: Jenkins є серцем DevOps-інфраструктури, сприяючи автоматизації всіх етапів розробки програмного забезпечення. Інтеграція з системами контролю версій, автоматичний запуск тестів та збірки після кожного коміту, а також розгортання на різних середовищах робить його ключовим елементом сучасних CI/CD процесів.

У результаті, Jenkins забезпечує швидку і ефективну реалізацію повного циклу розробки, тестування та доставки програмного забезпечення. Цей інструмент є важливим елементом для команд, які працюють у рамках методології DevOps, і значно підвищує продуктивність та ефективність розробки програмних продуктів. Його підтримка різноманітних плагінів та можливість масштабування дозволяють застосовувати Jenkins в різних сценаріях і на різних етапах життєвого циклу програмного забезпечення.

Контрольні питання до розділу 6

- 1) Яка основне призначення інструменту Jenkins?
- 2) Що являю собою Pipeline в Jenkins?

- 3) Яким чином користувач отримує результати тестування в Jenkins?
- 4) Які основні переваги архітектури Master-Slave в Jenkins?
- 5) Для чого потрібен Jenkinsfile і де він зберігається?
- 6) Яку функцію виконує Multibranch Pipeline в Jenkins?
- 7) Які основні плагіни потрібні для налаштування середовища Java-проєкту в Jenkins?
- 8) Які кроки потрібні для підключення Jenkins до репозиторію в GitHub?
- 9) Що таке Slave (Agent) в архітектурі Jenkins?
- 10) Яке призначення мають плагіни Jenkins?
- 11) Що з перерахованого є перевагою проєкту Freestyle Project у Jenkins?
 - a) Автоматизація роботи з контейнерами
 - b) Швидке виконання складних пайплайнів
 - c) Управління багатогалузовими репозиторіями
 - d) Просте налаштування через графічний інтерфейс
 - e) Паралельне виконання завдань
- 12) Яка основна функція інструменту Jenkins?
 - a) Тестування програмного забезпечення
 - b) Керування базами даних
 - c) Автоматизація процесів розгортання програмного забезпечення
 - d) Розробка програмного забезпечення
 - e) Створення звітної документації
- 13) Який тип розгортання підтримується Jenkins з метою мінімізації ризиків?
 - a) Повне розгортання
 - b) Часткове розгортання
 - c) Перехресне розгортання
 - d) Резервне розгортання
 - e) Блакитно-зелене розгортання
- 14) Яка мова програмування використовується для створення Jenkins Pipeline?
 - a) Python
 - b) TypeScript
 - c) JavaScript
 - d) Groovy
 - e) Ruby
- 15) Який протокол НЕ використовується для підключення агентів до Master?
 - a) SSH
 - b) JNLP

- c) HTTP
 - d) FTP
 - e) WebSocket
- 16) Який інструмент використовується для статичного аналізу коду в Jenkins?
- a) Maven
 - b) Gradle
 - c) SpotBugs
 - d) Jenkins Pipeline
 - e) Ant
- 17) Що означає директива "agent any" в декларативному Pipeline?
- a) Видалення всіх агентів
 - b) Створення нового агента
 - c) Перезапуск агентів
 - d) Виконання завдання на будь-якому доступному агенті
 - e) Блокування всіх агентів
- 18) Який тип проєкту в Jenkins дозволяє автоматично створювати пайплайни для кожної гілки репозиторію?
- a) Freestyle Project
 - b) Multi-configuration Project
 - c) Folder
 - d) Pipeline
 - e) Multibranch Pipeline
- 19) Що з вказаного є основною перевагою використання Pipeline у порівнянні з Freestyle Project?
- a) Простота налаштування
 - b) Надійніша інтеграція з Git
 - c) Швидкість виконання
 - d) Підтримка скриптів на Groovy
 - e) Менше споживання ресурсів
- 20) Яка функція плагіна Blue Ocean у Jenkins?
- a) Інтеграція з Docker
 - b) Візуалізація пайплайнів і покращення UX
 - c) Створення звітної документації
 - d) Автоматизація моніторингу серверів
 - e) Керування правами доступу

21) Встановіть відповідність між наступними елементами Jenkins та їх поясненнями:

A. Jenkinsfile	1) Структурні блоки пайплайну
B. Groovy	2) Одночасне виконання кількох завдань для скорочення часу
C. Stage	3) Файл для опису процесів CI/CD
D. Parallel	4) Мова написання сценаріїв

22) Встановіть відповідність між компонентами Jenkins та їх функціями:

A. Master	1) Виконує збірки та тести на різних платформах.
B. Slave	2) Керує загальним процесом роботи та координує завдання.
C. Pipeline	3) Забезпечує автоматичний запуск завдань після змін у репозиторії.
D. Webhook	4) Автоматизує процес розробки через сценарії Groovy.

23) Встановіть відповідність між типами робіт у Jenkins та їх призначенням:

A. Freestyle Project	1) Автоматизація багатоступеневих CI/CD процесів із використанням Groovy-сценаріїв.
B. Pipeline	2) Автоматичне створення пайплайнів для гілок репозиторію.
C. Multibranch Pipeline	3) Організація задач у вигляді папок із спільними налаштуваннями.
D. Folder	4) Для базових завдань із простим графічним налаштуванням.

24) Встановіть відповідність між етапами налаштування Jenkins та їх компонентами:

A. Source Code Management	1) Налаштування автоматичного запуску збірки
B. Build Triggers	2) Вказування URL репозиторію
C. Build Environment	3) Налаштування середовища виконання
D. Post-build Actions	4) Публікація результатів тестування

25) Встановіть відповідність між плагінами Jenkins та їх призначенням:

A. Git Plugin	1) Автоматизація процесів CI/CD через сценарії.
B. Slack Plugin	2) Відправлення повідомлень про стан завдань у месенджери.
C. Pipeline Plugin	3) Інтеграція Jenkins із системою контролю версій Git.
D. Docker Plugin	4) Інтеграція з контейнеризаційними середовищами.

РОЗДІЛ 7. GITHUB ACTIONS

7.1. Історія виникнення GitHub Actions

GitHub Actions був вперше анонсований у жовтні 2018 року на конференції GitHub Universe. Цей анонс викликав значний інтерес у спільноті розробників завдяки новій можливості автоматизувати робочі процеси безпосередньо в екосистемі GitHub. Після періоду бета-тестування інструмент був офіційно запусканий у листопаді 2019 року.

Головною метою створення GitHub Actions було спрощення процесу автоматизації для розробників, які вже активно використовують GitHub для управління репозиторіями. На той час багато компаній і команд використовували сторонні CI/CD сервіси, такі як Jenkins, Travis CI чи CircleCI, які вимагали додаткового налаштування та інтеграції. GitHub Actions став логічним рішенням, що об'єднало функції CI/CD із можливістю створення універсальних автоматизованих сценаріїв.

З розвитком DevOps та практик CI/CD дедалі більше команд потребували інструментів, які б допомагали знижувати кількість рутинних завдань і прискорювали процеси розробки та доставки програмного забезпечення. Автоматизація стала критично важливою, особливо в умовах сучасного динамічного ринку програмних продуктів.

GitHub, як лідер серед платформ для управління кодом, стратегічно вирішив інтегрувати автоматизацію безпосередньо у свою екосистему. Це дозволило розробникам уникати використання сторонніх сервісів для CI/CD і суттєво спростило процес інтеграції робочих процесів із репозиторіями.

GitHub Actions створювався з акцентом на зручність і простоту, зменшуючи поріг входу для автоматизації навіть для команд, які не мали значного досвіду роботи з CI/CD інструментами. Використання декларативного формату YAML для опису робочих процесів зробило налаштування інтуїтивно зрозумілим і доступним.

Ще одним важливим фактором стала гнучкість інструменту. GitHub Actions запропонував широку підтримку для різних операційних систем (Windows, MacOS, Linux) та мов програмування, що зробило його універсальним рішенням для багатьох проєктів. Він став інструментом, який можна адаптувати під різноманітні сценарії, від простих автоматизаційних задач до складних багатоступеневих робочих процесів.

Загалом, GitHub Actions був створений як зручний, потужний і інтегрований інструмент автоматизації, що дозволяє командам будь-якого рівня підготовки швидко налаштовувати процеси CI/CD та збільшувати ефективність розробки в цілому.

З моменту свого запуску GitHub Actions став основним інструментом для автоматизації робочих процесів у рамках GitHub. Завдяки постійному розвитку і впровадженню нових можливостей, він активно використовується як малими командами, так і великими корпораціями, здобувши популярність серед спільнот DevOps, CI/CD та open-source розробників. В таблиці 7.1 наведено зано основні етапи розвитку GitHub Actions.

Таблиця 7.1. Етапи розвитку GitHub Actions

Рік	Опис етапу	Додаткові деталі
2018 рік	Перша презентація GitHub Actions. Фокус на гнучких робочих процесах для автоматизації всередині GitHub	У цей період GitHub анонсував інструмент, орієнтуючись на інтеграцію з існуючою екосистемою. Початковий акцент був на можливостях автоматизації для малих і середніх команд
2019 рік	Випуск стабільної версії з акцентом на функціональність CI/CD	GitHub Actions отримав повну стабільність, забезпечивши підтримку для CI/CD процесів. З'явилися перші основні фічі, такі як підтримка YAML для опису робочих процесів та інтеграція з GitHub репозиторіями
2020-2021 роки	Активний розвиток і додавання нових функцій: підтримка матричних тестів, інтеграція з хмарними платформами (AWS, Azure, Google Cloud) і покращення продуктивності	Підтримка багатьох середовищ у рамках одного робочого процесу (матричні тести) та розширена інтеграція з хмарними платформами дозволили GitHub Actions стати потужним інструментом для різних типів розробки
2022 рік і далі	Впровадження нових засобів моніторингу, масштабованості та підтримка великих команд і організацій. GitHub Actions стає частиною стандартного інструментарію DevOps	GitHub Actions став невід'ємною частиною сучасних DevOps практик, з підтримкою масштабованості для великих проєктів і можливістю моніторингу в реальному часі для ефективнішого управління процесами CI/CD

GitHub Actions став вагомим фактором у трансформації автоматизації процесів у сфері розробки програмного забезпечення. Інструмент не лише

спростив впровадження CI/CD для розробників, але й створив нові можливості для вдосконалення робочих процесів.

GitHub Actions зробив автоматизацію доступною навіть для невеликих команд і незалежних розробників. Завдяки інтеграції з екосистемою GitHub, командам більше не потрібно витрачати ресурси на налаштування складної інфраструктури для CI/CD. Відсутність необхідності у використанні додаткового програмного забезпечення дозволила багатьом проектам швидко перейти до автоматизації. Безкоштовний доступ для публічних репозиторіїв зробив інструмент ідеальним вибором для open-source спільноти. Розробники отримали платформу, яка дозволяє легко налаштовувати тестування та релізи для публічних проєктів без витрат. Багато популярних open-source проєктів перейшли на використання GitHub Actions, що сприяло подальшому вдосконаленню цього інструменту.

GitHub Actions сприяв стандартизації автоматизації за допомогою опису робочих процесів у форматі YAML. Ця декларативна структура забезпечує прозорість і легкість у підтримці. Вона також знизила бар'єр входу для новачків, підвищивши сумісність і сприяючи обміну знаннями між командами.

Інтеграція GitHub Actions із GitHub Marketplace стала ключовим моментом у розвитку екосистеми. Спільнота розробників створила тисячі готових дій (actions), які значно прискорюють налаштування автоматизації. Marketplace став платформою для поширення рішень, що забезпечує легке впровадження популярних інструментів та сервісів, таких як Docker, AWS та Kubernetes.

Інструмент активно сприяє поширенню DevOps-підходу, об'єднуючи розробників та операційні команди. Автоматизація тестування, розгортання та моніторингу дозволила командам зосередитися на вдосконаленні продуктів, залишаючи рутинні завдання автоматизації. Навіть малі команди, які не мають доступу до складних рішень, отримали можливість впроваджувати DevOps-практики завдяки GitHub Actions.

З появою цього інструменту ринок CI/CD систем зазнав значних змін. Конкуренти, такі як Jenkins, GitLab CI/CD і CircleCI, почали адаптувати свої рішення, щоб відповідати новим вимогам спрощення налаштувань та інтеграції. Зростання популярності GitHub Actions стимулювало здорову конкуренцію, що позитивно вплинуло на розвиток індустрії загалом.

Крім того, GitHub Actions став платформою для інновацій у робочих процесах. Інтеграція з інструментами машинного навчання та штучного інтелекту відкрила нові можливості для автоматизації аналізу даних та оптимізації процесів. Кастомні дії дозволяють створювати унікальні рішення під конкретні потреби бізнесу, сприяючи гнучкості та адаптивності.

GitHub Actions змінив правила гри на ринку автоматизації, зробивши CI/CD доступним для всіх. Його вплив поширюється від малих команд до великих підприємств, стимулюючи розвиток DevOps-практик, стандартизацію процесів і впровадження інновацій у розробці програмного забезпечення.

7.2. Розбір інструменту GitHub Actions

Можна підсумувати, що GitHub Actions – це потужний інструмент автоматизації, який забезпечує розробників широкими можливостями для створення, тестування, розгортання додатків і виконання інших завдань безпосередньо у межах екосистеми GitHub. Він побудований на ідеї інтеграції та простоти, надаючи як базові, так і розширені функції для автоматизації робочих процесів.

Основні можливості GitHub Actions

Автоматизація процесів

GitHub Actions дозволяє автоматизувати широкий спектр завдань, включаючи:

- *запуск тестів*: автоматичне тестування змін у коді, що забезпечує стабільність проєкту;
- *розгортання додатків*: інструмент дозволяє безперервно розгортати додатки на хмарних сервісах, серверах або контейнерах;
- *управління релізами*: генерація релізів на основі певних подій, таких як створення нової гілки або тегу;
- *моніторинг і повідомлення*: надсилання сповіщень у Slack, Telegram або інші сервіси про успішне виконання чи помилки;
- *інтеграція сторонніх сервісів*: легка інтеграція з такими сервісами, як Docker, Kubernetes, AWS, Azure, Google Cloud та іншими.

Декларативний підхід

GitHub Actions використовує декларативний підхід для створення робочих процесів, що дозволяє чітко та зрозуміло описати автоматизовані завдання. Основна ідея цього підходу полягає в тому, що замість визначення процедур для кожного кроку, користувач описує самі результати і поведінку робочих процесів. Це знижує складність і робить налаштування автоматизації більш доступним для розробників, навіть без глибоких технічних знань.

Нижче наведено основні елементи декларативного підходу в GitHub Actions:

- *формат YAML*: GitHub Actions використовує формат YAML для опису робочих процесів. YAML є простим та інтуїтивно зрозумілим, що

дозволяє навіть новачкам легко писати та підтримувати сценарії. Завдяки цьому формату можна ефективно описувати навіть складні робочі процеси без необхідності в складному програмуванні;

- *логічна структура*: робочий процес (Workflow) складається з кількох основних компонентів:
 - Events (події): це тригери, що запускають робочий процес, наприклад, push, pull request, issue тощо;
 - Jobs (завдання): завдання – це набір кроків (steps), які виконуються на певному середовищі (наприклад, Ubuntu, Windows чи MacOS);
 - Steps (кроки): це окремі операції в рамках завдання, такі як виконання скриптів або запуск дій.

Ця чітка і зрозуміла структура допомагає організувати робочі процеси, знижує ймовірність помилок і спрощує подальше обслуговування автоматизованих сценаріїв. Користувачі можуть легко управляти та розширювати свої процеси, додаючи нові події, завдання або кроки, що робить систему надзвичайно гнучкою.

Інтеграція з репозиторієм

GitHub Actions забезпечує тісну інтеграцію з репозиторіями GitHub, що дозволяє автоматично виконувати завдання на основі подій, які відбуваються в репозиторії. Це дає змогу створювати гнучкі та потужні робочі процеси, які активуються безпосередньо від подій, що відбуваються в репозиторії, без необхідності в зовнішніх тригерах чи інструментах.

Розглянемо основні події, які активують робочий процес (workflow):

- *push*: ця подія спрацьовує, коли код змінюється в певній гілці репозиторію. Це дозволяє запускати автоматичні тести або розгортання при кожній зміні в коді;
- *pull_request*: спрацьовує, коли створюється або оновлюється pull request, що дозволяє виконувати перевірки, тестування або інші дії до злиття змін у головну гілку;
- *schedule*: подія, яка активує робочий процес на основі cron-розкладу. Це корисно для виконання періодичних завдань, таких як оновлення залежностей, регулярне тестування чи резервне копіювання;
- *issue*: відкриття або закриття завдання (issue) може запустити робочий процес, наприклад, для автоматизації коментарів або керування проектними задачами;
- *release*: спрацьовує, коли створюється новий реліз, що дозволяє автоматично виконувати завдання, такі як створення changelog або оновлення документації.

GitHub Actions має доступ до коду, історії комітів та інших метаданих репозиторіїв GitHub. Крім того, інструмент дозволяє працювати з GitHub API для автоматизації різних дій. Наприклад, можна автоматично створювати коментарі до pull request, оновлювати статуси завдань чи виконувати інші інтеракції з репозиторієм, що значно розширює можливості для автоматизації та покращення взаємодії в команді.

Гнучкість і кастомізація

GitHub Actions підтримує як використання готових дій, так і створення кастомних рішень, які також можна використовувати повторно:

- *готові дії (Reusable Actions):*

- у GitHub Marketplace доступні тисячі готових дій для різних завдань, таких як запуск тестів, розгортання контейнерів або інтеграція зі сторонніми сервісами;
- приклад використання дії для налаштування Node.js:

```
yaml
- uses: actions/setup-node@v3
  with:
    node-version: '16'
```

- *кастомні дії:* розробники можуть створювати власні дії на основі скриптів або Docker-контейнерів:
 - скриптові дії: написані на будь-якій мові програмування (наприклад, JavaScript);
 - дії на основі контейнерів: забезпечують незалежність середовища виконання;
 - матричні тести: можливість паралельного виконання завдань у різних середовищах або з різними параметрами.

```
yaml
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest]
    node: [14, 16]
```

Додаткові можливості GitHub Actions

До додаткових функцій GitHub Actions можна віднести такі, як (рис. 7.1):

- *середовища виконання (Runners)*: GitHub надає хости для виконання завдань у різних середовищах, зокрема Linux, Windows і macOS. Це дає можливість запускати робочі процеси на різних платформах, забезпечуючи сумісність та тестування на різних операційних системах. Окрім наданих GitHub середовищ, існує можливість налаштувати self-hosted runners – власні середовища для виконання робочих процесів. Це дозволяє більш гнучко керувати інфраструктурою, наприклад, для виконання завдань, які потребують спеціального програмного забезпечення або конфігурацій;
- *секрети (Secrets)*: GitHub Actions підтримує зберігання секретів, таких як ключі API, токени доступу або інші конфіденційні дані, які використовуються під час виконання workflow. Секрети зберігаються у зашифрованому вигляді, і доступ до них можливий тільки під час виконання робочих процесів. Це забезпечує безпеку при використанні чутливих даних в автоматизованих завданнях, наприклад, для розгортання додатків на серверах або доступу до зовнішніх API;
- *аналітика та моніторинг*: GitHub Actions надає інструменти для перегляду статусу робочих процесів, логів виконання та звітів про помилки. Це дозволяє відслідковувати прогрес виконання завдань, швидко ідентифікувати проблеми, а також здійснювати детальну перевірку кожного кроку в процесі. Моніторинг забезпечує високий рівень контролю за автоматизацією і дає змогу оперативно реагувати на неполадки;
- *сумісність*: GitHub Actions забезпечує сумісність з різними мовами програмування, фреймворками та платформами. Інструмент підтримує роботу з більшістю сучасних технологій, що дозволяє інтегрувати автоматизацію для різноманітних проєктів – від Python, JavaScript, Ruby до Java, Go, .NET і інших. Це робить GitHub Actions універсальним рішенням для команд, які працюють з різними стеками технологій.

Отже, GitHub Actions поєднує в собі простоту налаштування з потужними можливостями, що робить його ідеальним вибором як для новачків, так і для досвідчених DevOps-інженерів.

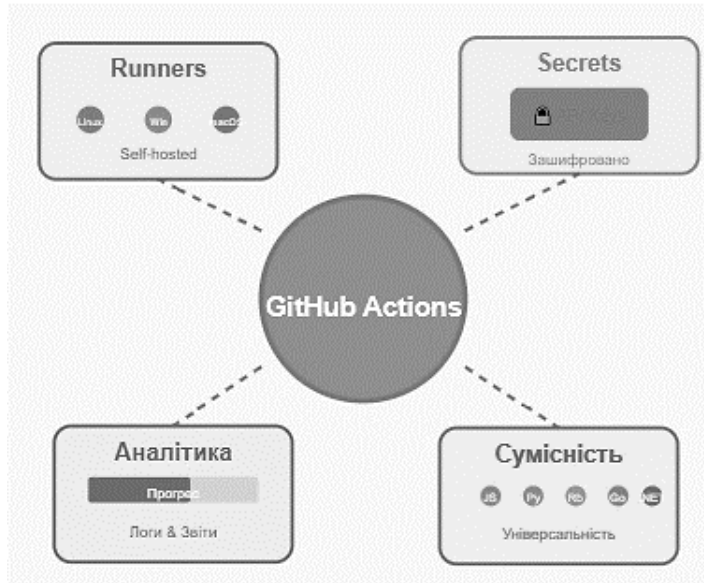


Рис. 7.1. Деякі додаткові функції GitHub Actions

Ключові компоненти GitHub Actions

GitHub Actions складається з кількох основних елементів, які дозволяють гнучко налаштовувати робочі процеси автоматизації. Завдяки цим компонентам розробники можуть легко адаптувати інструмент для виконання різноманітних завдань.

Workflow (робочий процес)

Workflow – це набір інструкцій, які визначають, як і коли виконується автоматизація. Вони описуються у файлах формату YAML, що зберігаються в каталозі `.github/workflows` репозиторія. Розглянемо ключові особливості:

- один репозиторій може містити кілька workflow, кожен із яких відповідає за виконання певних завдань (наприклад, CI, деплоймент або моніторинг);
- Workflow можуть бути простими (один Job) або складними (кілька взаємозалежних Job);
- вони активуються певними подіями (Events).

Нижче наведено приклад мінімального workflow:

```
yaml

name: CI Workflow
on: push
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: echo "Hello, world!"
```

Event (подія)

Event – це тригер, що запускає виконання workflow. Події визначаються у розділі `on` у YAML-файлі.

Типи подій є наступними:

- *події GitHub*:
 - *push*: коли зміни надсилаються до певної гілки;
 - *pull_request*: коли створюється або оновлюється pull request;
 - *release*: коли створюється новий реліз;
 - *issue*: коли створюється або оновлюється issue.
- *заплановані події*: виконання завдань за розкладом за допомогою синтаксису Cron (`schedule`);
- *ручний запуск*: використання події `workflow_dispatch` для запуску workflow вручну.

Приклад запланованого запуску наведено нижче:

```
yaml

on:
  schedule:
    - cron: "0 0 * * *" # Щодня опівночі
```

Job (завдання)

Job – це набір кроків (Steps), які виконуються у певному середовищі (Runner). Завдання у workflow можуть бути виконані паралельно або послідовно.

Нижче наведено особливості завдань:

- завдання визначаються у розділі jobs;
- кожне завдання має свою платформу виконання (runs-on), наприклад, ubuntu-latest, windows-latest або macos-latest;
- завдання можуть залежати одне від одного через параметр needs.

Для прикладу нижче наведено сценарій, що містить завдання із залежностями:

```
yaml

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Build process"
  test:
    runs-on: ubuntu-latest
    needs: build
    steps:
      - run: echo "Testing process"
```

Step (крок)

Step – це окремий крок у межах Job, який виконує конкретну дію, наприклад:

- завантаження коду репозиторія;
- запуск скрипта;
- виклик дії (Action);
- типи кроків:
 - дії (Actions): використання готових або кастомних модулів;
 - команди оболонки (Shell Commands): виконання команд у середовищі виконання.

Нижче наведено приклад кроків:

```
yaml

steps:
  - name: Checkout code
    uses: actions/checkout@v3
  - name: Install dependencies
    run: npm install
  - name: Run tests
    run: npm test
```

Actions (дії)

Actions – це окремі виконувані модулі, які дозволяють виконувати повторювані завдання. Дії можуть бути створені розробниками або використані з GitHub Marketplace.

Виділяють наступні типи дій:

- *готові дії*: наприклад, actions/checkout для завантаження коду репозиторія або actions/setup-node для налаштування середовища Node.js;
- *кастомні дії*: створюються розробниками для вирішення специфічних завдань.

Розглянемо приклад використання готової дії:

```
yaml

- name: Checkout code
  uses: actions/checkout@v3
```

Нижче проаналізовано приклад створення кастомної дії (на основі JavaScript). Код дії:

```
javascript

const core = require('@actions/core');
const github = require('@actions/github');

try {
  const name = core.getInput('name');
  console.log(`Hello, ${name}!`);
} catch (error) {
  core.setFailed(error.message);
}
```

Використання кастомної дії може бути наступним:

```
yaml

steps:
  - uses: ./my-action
    with:
      name: 'World'
```

Можна зробити висновок, що GitHub Actions надає гнучку архітектуру, яка дозволяє легко адаптувати її до потреб конкретного проєкту. Кожен із компонентів доповнює інші, забезпечуючи можливість створення простих або складних автоматизованих сценаріїв.

7.3. Відмінності GitHub Actions від аналогічних систем

GitHub Actions – це сучасний інструмент автоматизації, який має свої переваги й обмеження порівняно з іншими популярними CI/CD системами, такими як Jenkins, GitLab CI/CD, CircleCI та Travis CI. Розглянемо ключові відмінності.

Інтеграція

GitHub Actions є нативним рішенням для GitHub, що забезпечує повну інтеграцію з екосистемою цієї платформи. Зокрема, GitHub Actions:

- не потребує додаткового налаштування для роботи з репозиторіями GitHub;

- дозволяє легко використовувати події GitHub (push, pull_request, issue, release) як тригери для робочих процесів;
- має прямий доступ до GitHub API, токенів і метаданих репозиторіїв.

Інші системи:

- Jenkins, Travis CI, CircleCI і GitLab CI/CD можуть працювати з GitHub, але потребують інтеграції через вебхуки, плагіни або додаткові конфігурації;
- наприклад, для використання Jenkins із GitHub потрібен плагін GitHub Integration.

GitHub Actions є більш зручним для користувачів GitHub завдяки тісній інтеграції без додаткових налаштувань.

Простота у використанні

GitHub Actions відрізняється простим налаштуванням і декларативним підходом, що робить його зручним для розробників усіх рівнів.

Так, наприклад, GitHub Actions:

- використовує YAML-файли для опису робочих процесів;
- мінімальний поріг входу для новачків завдяки зрозумілій документації та готовим прикладам;
- не потребує складного налаштування серверів або хостів для виконання завдань.

Для порівняння, інші системи:

- Jenkins: надзвичайно гнучкий, але має круту криву навчання через складність конфігурації;
- GitLab CI/CD: також використовує YAML, але вимоги до хостингу (self-hosted GitLab) можуть ускладнити процес для новачків.

GitHub Actions виграє в простоті завдяки мінімальному обсягу налаштувань і декларативному підходу.

Маркетплейс готових дій (GitHub Marketplace)

GitHub Actions пропонує великий вибір готових дій у GitHub Marketplace, що суттєво прискорює автоматизацію.

GitHub Actions забезпечує доступ до тисяч готових дій (наприклад, для налаштування середовищ, запуску тестів, деплойменту) та підтримку кастомних дій, які можна легко розповсюджувати через Marketplace.

Для порівняння, у Travis CI, CircleCI чи Jenkins немає централізованого маркетплейсу, хоча існують окремі спільноти та сховища готових скриптів чи плагінів. Jenkins має плагіни, але їх налаштування може бути складнішим.

Наявність GitHub Marketplace робить налаштування та розширення функціоналу GitHub Actions швидким і зручним.

Цінова політика

Ціна є важливим фактором вибору CI/CD інструменту, і GitHub Actions має вигідні умови для невеликих проєктів.

GitHub Actions надає змогу використовувати:

- безкоштовний рівень:
 - для публічних репозиторіїв – необмежене використання;
 - для приватних репозиторіїв – обмеження за кількістю хвилин виконання та місцем на зберіганні артефактів (2,000 хвилин на місяць для безкоштовного тарифу).
- можливість використовувати self-hosted runners для уникнення обмежень.

Для прикладу, інші системи мають такі особливості:

- Jenkins: безкоштовний, але потребує хостингу (власний сервер або хмарний сервіс), що може бути дорогим для великих проєктів;
- GitLab CI/CD: має безкоштовний рівень, але з подібними обмеженнями для приватних проєктів;
- Travis CI: обмеження безкоштовного використання для приватних репозиторіїв (споживання віртуальних хвилин).

GitHub Actions є більш економічним для команд, що працюють із публічними репозиторіями, хоча використання self-hosted runners може вимагати додаткових витрат на інфраструктуру, але для великих приватних проєктів витрати будуть значно більші.

Гнучкість

Гнучкість – це ключовий параметр для складних CI/CD сценаріїв.

GitHub Actions забезпечує високу гнучкість для більшості стандартних задач та підтримує кастомні дії, але для надзвичайно складних сценаріїв може знадобитися більше зусиль.

Для порівняння, інші системи:

- Jenkins: дуже гнучкий і потужний завдяки плагінам, але потребує більше часу для налаштування;
- CircleCI: забезпечує просте налаштування для типових сценаріїв, але менш гнучкий для кастомних задач порівняно з Jenkins;
- GitLab CI/CD: схожий за функціоналом на GitHub Actions, але кращий для інтеграції у власну екосистему GitLab.

GitHub Actions є ідеальним вибором для простих і середньої складності задач. Для складних кастомних сценаріїв Jenkins залишається кращим.

У таблиці 7.2 представлено порівняння GitHub Actions з іншими популярними інструментами для автоматизації CI/CD, такими як Jenkins, GitLab

CI/CD, CircleCI та Travis CI. Це порівняння допоможе оцінити сильні та слабкі сторони кожної платформи, а також зрозуміти, які рішення можуть найкраще відповідати потребам вашого проєкту.

Таблиця охоплює ключові характеристики таких інструментів, як зручність інтеграції, підтримка різних середовищ, ціна, гнучкість у налаштуваннях і підтримка масштабованості, що дозволяє вибрати оптимальний інструмент для автоматизації процесів у вашій команді.

Таблиця 7.2 Порівняння GitHub Actions з інструментами для CI/CD

Параметр	GitHub Actions	Jenkins	GitLab CI/CD	CircleCI	Travis CI
Інтеграція	Глибока з GitHub	Плагіни потрібні	Глибока з GitLab	Вимагає налаштувань	Вимагає налаштувань
Простота	Висока	Низька	Середня	Висока	Висока
Готові рішення	GitHub Marketplace	Плагіни	Обмежені	Обмежені	Обмежені
Цінова політика	Безкоштовно для публічних	Безкоштовний (але хостинг)	Безкоштовно для публічних	Безкоштовний рівень	Обмежено безкоштовний
Гнучкість	Висока для більшості задач	Дуже висока	Висока	Середня	Середня

7.4. Приклади застосування GitHub Actions

GitHub Actions широко використовується для автоматизації процесів безперервної інтеграції (CI) та доставки (CD). Розглянемо деякі приклади практичного використання сервісу.

Тестування коду при кожному pull request

GitHub Actions дозволяє автоматизувати запуск тестів при створенні або оновленні **pull request**. Це важливий крок у процесі безперервної інтеграції, який забезпечує перевірку коректності коду перед його об'єднанням із основною гілкою репозиторію. Запуск тестів на кожному етапі дає змогу:

- *виявити помилки та баги* на ранніх етапах, що знижує ймовірність потрапляння неякісного коду в основний репозиторій;
- *покращити командну співпрацю*, адже всі зміни перевіряються перед їх злиттям, що дозволяє уникнути конфліктів;
- *автоматизувати процес перевірки*, що економить час і знижує навантаження на розробників, дозволяючи зосередитись на важливих задачах.

Нижче наведено приклад сценарію, що включає в себе виконання тестів.

```
yaml

name: Test Code on PR
on: pull_request
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm install
      - run: npm test
```

Розглянемо детальніше його складові:

name: Test Code on PR – назва робочого процесу (workflow) – у цьому випадку "Test Code on PR", що означає «Тестування коду при створенні pull request».

on: pull_request – визначає, що цей робочий процес буде запускатися, коли створюється або оновлюється pull request у репозиторії.

Jobs – визначає завдання (jobs), які будуть виконуватися в рамках цього робочого процесу.

Test – ім'я одного з завдань. У цьому випадку це завдання називається "test".

runs-on: ubuntu-latest – вказує, що завдання буде виконуватися на віртуальній машині з операційною системою Ubuntu (остання стабільна версія).

Steps – перелік кроків, які виконуються в рамках завдання.

- **- uses: actions/checkout@v3**

Перший крок використовує офіційну дію actions/checkout (версія 3), щоб клонувати код репозиторія в робоче середовище.

- **- run: npm install**

Другий крок виконує команду npm install, щоб встановити всі залежності проекту, визначені у файлі package.json.

- **- run: npm test**

Третій крок виконує тести за допомогою команди npm test.

Цей робочий процес автоматизує перевірку коду при створенні pull request. Він забезпечує:

- клонування коду з репозиторія;
- встановлення залежностей;

- виконання тестів для перевірки коду.

Розгортання додатка на сервері чи хмарній платформі

GitHub Actions дозволяє автоматизувати процес розгортання додатка після успішного проходження тестування, що є важливою частиною CI/CD процесу. Це дає змогу значно скоротити час між написанням коду та його реальним впровадженням у продуктивне середовище. Під час налаштування автоматичного розгортання можна врахувати наступні аспекти:

- автоматизація розгортання: після того як код пройшов тестування, GitHub Actions автоматично виконує розгортання додатка на обрану платформу. Це може включати завантаження контейнерів, налаштування серверів, або деплой на хмарні платформи;
- підтримка різних платформ: GitHub Actions підтримує інтеграцію з багатьма популярними хмарними платформами і сервісами, що дає змогу розгорнути додатки на таких платформах, як:
 - AWS (Amazon Web Services);
 - Azure (Microsoft Cloud);
 - Google Cloud;
 - Heroku;
 - Docker Hub;
 - інші хмарні сервіси та сервіси контейнеризації.

Це дозволяє забезпечити гнучкість у виборі хмарної інфраструктури та швидко здійснювати деплой на будь-яку платформу, що використовується командою.

Розглянемо приклад фрагменту сценарію, що включає автоматичне розгортання на AWS:

```
yaml
- name: Deploy to AWS
  uses: aws-actions/configure-aws-credentials@v3
  with:
    aws-access-key-id: ${ secrets.AWS_ACCESS_KEY_ID }
    aws-secret-access-key: ${ secrets.AWS_SECRET_ACCESS_KEY }
    aws-region: us-east-1
```

Цей фрагмент конфігурації YAML для GitHub Actions, використовується для налаштування облікових даних AWS з метою деплоя (впровадження) додатків або ресурсів. Розглянемо детальніше елементи:

name: Deploy to AWS – назва цього кроку в робочому процесі. Цей крок позначає впровадження ресурсу чи застосунку до AWS.

uses: aws-actions/configure-aws-credentials@v3 – використовується дія з репозиторію GitHub Actions – aws-actions/configure-aws-credentials (версія 3). Ця дія дозволяє налаштувати облікові дані для доступу до AWS.

with – далі вказано перелік параметрів, які передаються до цієї дії.

- **aws-access-key-id: \${ secrets.AWS_ACCESS_KEY_ID }**
Вказує ідентифікатор ключа доступу AWS (AWS Access Key ID), що зберігається у секретех GitHub (GitHub Secrets). Це забезпечує безпечне використання чутливої інформації;
- **aws-secret-access-key: \${ secrets.AWS_SECRET_ACCESS_KEY }**
Вказує секретний ключ доступу AWS (AWS Secret Access Key), також збережений у GitHub Secrets;
- **aws-region: us-east-1**
Визначає регіон AWS, у якому буде здійснюватися впровадження. У цьому випадку це us-east-1 (Один з регіонів AWS, наприклад, Північна Вірджинія).

Цей код може бути частиною CI/CD процесу, де застосунок автоматично впроваджується на AWS, наприклад, у службу Amazon S3, ECS або Lambda.

Моніторинг

GitHub Actions дозволяє автоматизувати процеси моніторингу, забезпечуючи зручний механізм відслідковування виконання робочих процесів та сповіщення команди про їхній статус. Це допомагає оперативно реагувати на помилки та швидко реагувати на завершення важливих завдань у робочому процесі.

GitHub Actions дозволяє надсилати сповіщення в популярні сервіси, такі як Slack, Telegram та інші платформи, після завершення workflow, незалежно від його результату (успіх чи помилка). Це корисно для того, щоб вся команда була в курсі подій і могла вчасно реагувати на будь-які помилки чи проблеми, що виникають під час виконання робочих процесів.

Сповіщення у Slack можуть бути налаштовані для відправки повідомлення після виконання кожного кроку workflow, що дозволяє команді миттєво дізнатись про результат тестування, деплою або інших завдань.

Для налаштування таких сповіщень можна використовувати інтеграцію GitHub Actions із відповідними діями, що дозволяють автоматично надсилати повідомлення через вебхуки або спеціальні API, що підтримують ці сервіси.

Наведений нижче фрагмент коду представляє крок у конфігурації GitHub Actions, який відправляє повідомлення в Slack про статус виконання робочого процесу.

```
yaml
- name: Notify Slack
  uses: slackapi/slack-github-action@v1
  with:
    slack-message: "Workflow завершено: ${ github.workflow } зі статусом ${ job.status }"
    slack-webhook-url: ${ secrets.SLACK_WEBHOOK }
```

Розглянемо детальніше його складові елементи:

- **name: Notify Slack** – назва кроку – "Notify Slack". Він позначає, що цей крок відповідає за надсилання сповіщення в Slack.

uses: slackapi/slack-github-action@v1 – крок використовує дію slackapi/slack-github-action (версія 1), яка інтегрує Slack з GitHub Actions для відправлення повідомлень.

with – визначає вхідні параметри для дії.

- **slack-message: "Workflow завершено: \${ github.workflow } зі статусом \${ job.status }"** – повідомлення, яке буде надіслано до Slack.
 - **github.workflow** – змінна GitHub, яка підставляє ім'я робочого процесу;
 - **job.status** – змінна GitHub, яка підставляє статус виконання завдання (наприклад, "success", "failure" тощо). Приклад повідомлення: *"Workflow завершено: Build and Test зі статусом success"*.
- **slack-webhook-url: \${ secrets.SLACK_WEBHOOK }** – URL вебхука Slack, через який надсилаються повідомлення.

secrets.SLACK_WEBHOOK – це змінна середовища, що містить URL вебхука. Вона зберігається в GitHub Secrets для безпеки (чутлива інформація не зберігається у відкритому вигляді).

Цей крок може бути частиною CI/CD процесу, де команда отримує повідомлення в Slack про результат виконання, наприклад, тестування, деплою чи інших операцій. Це допомагає вчасно отримувати статуси та реагувати на помилки.

Автоматизація релізів

GitHub Actions значно спрощує процес створення релізів, автоматизуючи багато етапів, що зазвичай виконуються вручну. Це дозволяє зменшити час і зусилля, необхідні для підтримки релізного циклу, а також знижує ймовірність помилок. Основні функції автоматизації релізів включають генерацію changelog і створення нових релізів.

GitHub Actions може інтегруватися з інструментами, такими як actions/create-release, для автоматичного створення нових релізів та генерації changelog. Це забезпечує безперервний процес випуску нових версій продукту, з детальним описом змін, що відбулися. Автоматично створені changelog файли містять список змін, виправлених помилок, нових функцій та інших важливих оновлень, що допомагає користувачам та розробникам бути в курсі змін у нових релізах.

Використання actions/create-release у GitHub Actions дозволяє автоматично створювати релізи після кожного коміту або за розкладом, включаючи оновлений changelog. Це забезпечує ефективний та автоматизований процес для команди, зменшуючи кількість рутинних завдань і дозволяючи зосередитися на розробці нових функцій.

Інтеграція цих процесів в один робочий потік за допомогою GitHub Actions дає можливість безперервно оновлювати продукт і ефективно керувати релізами.

Роглянемо наступний приклад сценарію:

```
yaml
- name: Generate Changelog
  uses: actions/github-script@v6
  with:
    script: |
      const { generateChangelog } = require('changelog-generator');
      generateChangelog();
- name: Create Release
  uses: actions/create-release@v1
  with:
    tag_name: ${ github.sha }
    release_name: "Release ${ github.sha }"
```

На першому кроці генерується журнал змін, використовуючи JavaScript-код.

На другому кроці створюється новий реліз із автоматично створеним тегом і назвою, які базуються на SHA комміту.

Перша секція: `Generate Changelog` має таку структуру:

Name – назва цього кроку. У цьому випадку це "Generate Changelog" (згенерувати журнал змін). Вона буде відображена в логи виконання GitHub Actions.

Uses – вказує, що використовується офіційний GitHub Action під назвою `github-script`, версії `v6`. Цей Action дозволяє виконувати JavaScript-код безпосередньо в рамках CI/CD-процесу.

with: – блок передає вхідні параметри для `github-script`.

script: Поле `script` містить JavaScript-код, який буде виконаний у контексті GitHub. Символ `|` означає, що наступні рядки є частиною цього скрипта.

const { generateChangelog }: Із пакета `changelog-generator` імпортується функція `generateChangelog`. Ця функція відповідає за генерацію журналу змін.

require('changelog-generator'); Використовується для підключення зовнішньої бібліотеки або модуля, який генерує журнал змін.

generateChangelog(); Викликається імпортована функція `generateChangelog` для створення журналу змін. Її поведінка залежить від реалізації в бібліотеці `changelog-generator`.

Розглянемо другу секцію `Create Release`:

name – назва другого кроку "Create Release" (створити реліз). Цей етап створює новий реліз у репозиторії.

uses – використовується інший GitHub Action – `create-release`, версії `v1`. Він призначений для створення релізів у GitHub.

with – блок параметрів, які передаються екшену `create-release`.

tag_name: – тег релізу, що відповідає SHA (унікальному ідентифікатору) останнього комміту.

{{{ github.sha }}}: Спеціальна змінна GitHub, що динамічно підставляє SHA комміту.

release_name: Назва релізу. У цьому випадку вона задається у вигляді `Release <SHA>`, де `<SHA>` – це той самий ідентифікатор комміту.

Сканування безпеки

GitHub Actions надає можливість інтегрувати інструменти безпеки для автоматичного сканування та аналізу вразливостей у коді та залежностях. Це допомагає підтримувати високий рівень безпеки проєкту та знижує ризики, пов'язані з використанням небезпечних бібліотек або конфігурацій.

Dependabot є інструментом, який дозволяє автоматично перевіряти залежності проєкту на наявність вразливостей. Він регулярно аналізує

бібліотеки, які використовуються в репозиторії, і повідомляє про потенційно небезпечні пакети або застарілі версії.

У разі виявлення вразливості Dependabot може автоматично створювати pull request із оновленням відповідної залежності до безпечної версії.

Нижче наведено фрагмент конфігурації:

```
yaml

- name: Dependency Review
  uses: actions/dependency-review-action@v2
```

Цей код належить до конфігурації GitHub Actions і використовується для автоматизованого аналізу залежностей проєкту.

Name: Dependency Review – назва кроку в GitHub Actions (аналіз залежностей). Назва з'явиться у журналах виконання.

uses – використовується готовий GitHub Action – dependency-review-action, версії v2.

Цей екшен автоматично аналізує залежності проєкту (наприклад, бібліотеки, модулі), щоб знайти потенційні уразливості чи небезпечні оновлення. Код перевіряє залежності проєкту, аналізуючи зміни в файлах залежностей, таких як package.json чи requirements.txt. Він виявляє нові або оновлені залежності та попереджає про відомі вразливості у їхніх версіях.

Це корисно тим, що захищає кодову базу від небезпечних залежностей, автоматизує аналіз, який зазвичай потребує ручної перевірки, і допомагає розробникам виявляти потенційні ризики на етапі pull request або перед релізом.

Перевірка стилю коду

GitHub Actions дозволяє автоматизувати процес перевірки стилю та якості коду, забезпечуючи дотримання командних стандартів і підвищення загальної читабельності та підтримуваності проєкту.

Для перевірки стилістичних помилок рекомендується використання статичних аналізаторів коду, таких як ESLint (для JavaScript/TypeScript), Prettier (для форматування коду) або інших інструментів залежно від мови програмування. Перевірка може виконуватися автоматично при кожному push або pull request, щоб уникнути введення стилістичних помилок у основну гілку.

Розглянемо наступний приклад:

```
yaml

- name: Lint Code
  uses: actions/setup-node@v3
  with:
    node-version: '16'
- run: npm install eslint
- run: npx eslint . --fix
```

Цей код є фрагментом YAML-конфігурації для GitHub Actions, яка виконує автоматичне лінтингування (перевірку та виправлення стилю коду) в репозиторії.

name: Lint Code – назва кроку, вказує, що на цьому етапі виконується лінтинг коду.

Uses – використовується офіційний GitHub Action `setup-node`, версії `v3`. Він налаштовує середовище `Node.js` для подальшої роботи.

With – передаються параметри для налаштування `Node.js`.

node-version: – встановлюється версія `Node.js` – в даному випадку `16`.

Run – команда, яка виконується в терміналі.

npm install eslint: – встановлює `ESLint` (пакет для перевірки стилю та якості коду) за допомогою менеджера пакетів `npm`.

Run – виконується друга команда в терміналі.

npx eslint . --fix: – запускає `ESLint` для перевірки коду у всьому проєкті `(.)` та автоматично виправляє знайдені помилки `(--fix)`.

Наведений код налаштовує середовище `Node.js` версії `16`, встановлює `ESLint` як залежність і перевіряє весь код у репозиторії на відповідність стилістичним і якісним правилам, автоматично виправляючи помилки, де це можливо.

Це потрібно для забезпечення узгодженості стилю коду в проєкті, допомагає знаходити та виправляти потенційні проблеми ще до злиття змін у репозиторій і автоматизує процес лінтингу, що значно економить час розробників.

Інші приклади застосування

- *автоматизація документування:* автоматичне оновлення документації в репозиторії після внесення змін у код. Інтеграція з генераторами документації, такими як `JSDoc` або `Sphinx`;

- *перевірка pull request*: автоматична перевірка опису pull request для відповідності шаблонам чи правилам;
- *очищення артефактів*: видалення застарілих артефактів для економії місця.

GitHub Actions дозволяє вирішувати широкий спектр завдань, від простих перевірок до складних сценаріїв автоматизації.

Контрольні питання до розділу 7

- 1) Яке призначення сервісу GitHub Actions?
- 2) Де зберігаються файли Workflow у репозиторії?
- 3) У чому полягає декларативний підхід у GitHub Actions, і який формат використовується для опису робочих процесів?
- 4) Які завдання можна автоматизувати за допомогою GitHub Actions?
- 5) Яке призначення має GitHub Marketplace?
- 6) Що означає параметр needs у налаштуванні завдань?
- 7) Які типи середовищ виконання (Runners) надає GitHub?
- 8) Що таке кастомні дії (Custom Actions) у GitHub Actions?
- 9) Яким чином GitHub Actions забезпечує безпеку при роботі з даними?
- 10) Яким чином можна інтегрувати GitHub Actions з хмарними платформами, такими як AWS чи Google Cloud?
- 11) Що з наведеного є основним компонентом GitHub Actions?
 - a) Container
 - b) Pipeline
 - c) Workflow
 - d) Script
 - e) Branch
- 12) Яка подія використовується для ручного запуску workflow?
 - a) manual_trigger
 - b) workflow_dispatch
 - c) manual_start
 - d) workflow_trigger
 - e) start_manually
- 13) Який основний формат використовується для конфігурації GitHub Actions?
 - a) JSON
 - b) HTML

- c) YAML
 - d) TOML
 - e) XML
- 14) Який з наступних сервісів має найкращу інтеграцію з GitHub?
- a) CircleCI
 - b) Travis CI
 - c) Jenkins
 - d) GitHub Actions
 - e) GitLab CI/CD
- 15) Який з наступних файлів може зберігати опис Workflow у GitHub Actions?
- a) .github/workflows.yaml
 - b) .github/github.tf
 - c) .github/workflows/test.yaml
 - d) .github-actions/workflows.yaml
 - e) github/workflow-test.txt
- 16) Який з наступних компонентів відповідає за виконання завдань у Workflow?
- a) Action
 - b) Event
 - c) Secret
 - d) Runner
 - e) Webhook
- 17) Що являє собою Secrets у GitHub Actions?
- a) Набір дій для інтеграції з Docker
 - b) Опис всіх YAML-файлів
 - c) Налаштування Workflow
 - d) Зашифровані конфіденційні дані
 - e) Сповіщення у Slack
- 18) Яка з вказаних подій не є стандартним тригером для GitHub Actions?
- a) push
 - b) pull_request
 - c) issue
 - d) deploy
 - e) release

19) Яка мінімальна кількість jobs може бути у workflow?

- a) Одна
- b) Жодної
- c) Дві
- d) П'ять
- e) Три

20) Яка основна перевага GitHub Actions для open-source проєктів?

- a) Інтеграція з Jenkins
- b) Можливість створення секретів
- c) Автоматичне створення репозиторіїв
- d) Автоматизація моніторингу продуктивності
- e) Безкоштовний доступ для публічних репозиторіїв

21) Встановіть відповідність між подіями GitHub Actions та їхнім описом:

A. push	1) Створення нової версії
B. pull_request	2) Надсилання змін до гілки
C. release	3) Створення завдання
D. issue	4) Запит на злиття змін

22) Встановіть відповідність між компонентами, що відповідають за безпеку, та їх призначенням:

A. Secrets	1) Зберігання конфіденційних даних.
B. Permissions	2) Перевірка прав доступу.
C. Environment	3) Виконання завдань.
D. Runner	4) Ізольоване середовище виконання.

23) Встановіть відповідність між типами сповіщень та їхнім застосуванням:

A. Slack	1) Системні повідомлення.
B. GitHub	2) Командні комунікації.
C. Email	3) Формальні сповіщення.
D. Webhook	4) Інтеграція з іншими сервісами.

24) Встановіть відповідність між типами секретів та їхнім використанням:

A. API-ключі	1) Доступ до віддалених серверів.
B. Токени доступу	2) Взаємодія з зовнішніми API.
C. Паролі сервісів	3) Авторизація у зовнішніх системах.
D. SSH-ключі	4) Захищений доступ до репозиторіїв.

25) Встановіть відповідність між наступними діями та їх функціями:

A. actions/checkout	1) Інтеграція з сервісом AWS.
B. actions/setup-node	2) Налаштування Node.js середовища.
C. aws-actions/configure-aws-credentials	3) Надсилання сповіщень у Slack.
D. slackapi/slack-github-action	4) Завантаження коду репозиторію.

РОЗДІЛ 8. АВТОМАТИЧНЕ ТЕСТУВАННЯ

Автоматичне тестування – це процес перевірки програмного забезпечення з використанням спеціальних інструментів та скриптів для виконання тестів без втручання людини. Воно є невід’ємною частиною сучасної розробки програмного забезпечення та спрямоване на оптимізацію процесу тестування, підвищення ефективності та мінімізацію людського фактора. Завдяки автоматизації тестування можна швидше знаходити помилки, забезпечувати стабільність коду та спрощувати інтеграцію нових функціональностей. Це дозволяє зменшити час виходу продукту на ринок, скоротити витрати на підтримку, а також підвищити загальну якість програмного забезпечення.

8.1. Основи автоматичного тестування

Автоматичне тестування включає в себе використання спеціалізованих інструментів і скриптів для автоматизації виконання тестових сценаріїв, які перевіряють різні аспекти програмного забезпечення. Воно дозволяє проводити регулярні, повторювані та масштабовані перевірки з можливістю налаштування умов виконання, зменшуючи потребу в ручному втручанні.

Автоматизація виконує завдання, які займають багато часу під час ручного тестування, наприклад, повторювані перевірки або тестування великих наборів даних. Проте, у деяких випадках ручне тестування залишається необхідним, наприклад, для оцінки UX або перевірки нестандартних сценаріїв.

Тестові скрипти – це автоматизовані сценарії, які виконують перевірки функціональності, продуктивності, безпеки, сумісності тощо, використовуючи декларативний підхід для опису умов та результатів виконання. Ці скрипти можуть бути написані мовами програмування (наприклад, Python, Java) або створені у спеціалізованих середовищах для автоматичного тестування (наприклад, Selenium або Cypress).

Завдяки автоматизації тестування можна швидко виявляти помилки та забезпечувати високу якість програмного забезпечення. Це особливо важливо в умовах сучасного Agile-розроблення та CI/CD процесів, де релізи відбуваються часто й швидко.

Автоматизація дозволяє розробникам одразу дізнаватися про помилки в нових змінах завдяки швидкому запуску тестів. Тести можна запускати багаторазово в однакових умовах, що забезпечує стабільність і передбачуваність результатів. Автоматичне тестування легко адаптується до великих проєктів, де тестування вручну може бути занадто складним або тривалим. Незважаючи на

початкові витрати на розробку тестів, автоматизація значно зменшує витрати на тестування під час тривалого циклу життя проєкту.

На рисунку 8.1 наведено основні типи завдань, які вирішуються за допомогою автоматизації.



Рис. 8.1. Основні завдання, що вирішуються завдяки автоматичним тестам

Можна виділити такі переваги автоматичного тестування

Швидкість виконання тестів

- автоматизовані тести виконуються набагато швидше, ніж ручні, оскільки вони не залежать від людського фактора. Це особливо важливо для великих проєктів, де кількість тестів може обчислюватися сотнями або тисячами;
- завдяки швидкості автоматизації тестування розробники можуть отримувати результати одразу після внесення змін у код, що значно прискорює цикл розробки.

Можливість повторного використання тестів

- один раз створені автоматизовані тести можуть використовуватися багаторазово, що робить їх ефективними для регресійного тестування;
- тести можна налаштувати для виконання на різних платформах, пристроях або середовищах, що забезпечує гнучкість і масштабованість;
- при оновленні програмного забезпечення можна легко адаптувати існуючі тести, що економить час і ресурси.

Раннє виявлення помилок

- автоматичне тестування дозволяє швидко виявляти помилки на ранніх етапах розробки, зменшуючи витрати на їх виправлення;
- інтеграція автоматичних тестів у процеси CI/CD забезпечує постійну перевірку кожного нового коміту, що дозволяє уникати накопичення дефектів у коді;
- це зменшує ризики, пов'язані з релізом, і забезпечує високу якість продукту.

Підвищення стабільності релізів:

- постійне автоматичне тестування допомагає уникнути ситуацій, коли неочікувані помилки потрапляють у продуктивне середовище;
- завдяки ретельному тестуванню всієї системи забезпечується стабільність релізів, навіть якщо продукт активно розвивається;
- регресійне тестування дозволяє перевірити, що нові зміни не порушують існуючу функціональність, що робить релізи більш передбачуваними.

Також, в якості додаткових переваг варто відмітити:

- *зниження навантаження на команду тестувальників*: автоматизація зменшує обсяг рутинної роботи, дозволяючи тестувальникам зосередитися на складніших і творчих завданнях, наприклад, тестуванні користувацького досвіду або нестандартних сценаріїв;
- *підвищення точності*: виконання тестів автоматичними інструментами усуває ризики, пов'язані з людськими помилками під час ручного тестування;
- *економія часу та ресурсів*: хоча розробка автоматичних тестів вимагає початкових інвестицій, у довгостроковій перспективі це знижує витрати на підтримку та тестування;
- *масштабованість*: автоматизовані тести легко адаптуються до великих проєктів або систем із багатокomпонентною архітектурою.

Інструменти для автоматичного тестування

Сучасні інструменти автоматичного тестування пропонують широкий спектр можливостей для перевірки функціональності, продуктивності, безпеки, а також інтеграції та розгортання. Нижче наведено деякі з найбільш популярних інструментів, їх призначення та особливості.

Selenium – це відкрита платформа для автоматизації тестування веб-додатків, яка широко використовується завдяки своїй гнучкості та підтримці різних мов програмування, таких як Java, Python, C#, Ruby, JavaScript. Інструмент дозволяє тестувати веб-додатки на різних браузерах, включаючи

Chrome, Firefox, Edge та Safari, що забезпечує кросбраузерну сумісність. Для розподіленого тестування використовується Selenium Grid, що дає можливість виконувати тести паралельно на різних машинах. Selenium найчастіше застосовується для перевірки інтерфейсу користувача (UI), забезпечуючи автоматизацію взаємодії з веб-сторінками.

JUnit є інструментом для автоматизації юніт-тестів у Java. Він дозволяє ефективно писати, організовувати та запускати юніт-тести. Серед його можливостей – інтеграція з багатьма інструментами CI/CD, такими як Jenkins, а також підтримка анотацій, що спрощують процес написання тестів. JUnit є одним із найпоширеніших фреймворків серед Java-розробників.

TestNG є розширеним фреймворком для автоматичного тестування Java-додатків. Він забезпечує підтримку паралельного запуску тестів, інтеграцію з інструментами звітності та гнучку структуру, яка дозволяє виконувати тести залежно від їх пріоритету. Цей фреймворк часто використовується разом із Selenium для розширення можливостей автоматизації.

PyTest – це фреймворк для тестування Python-додатків, що відзначається простотою використання завдяки зрозумілому синтаксису. Він дозволяє створювати як юніт-, так і інтеграційні тести, а також підтримує плагіни для розширення функціональності. PyTest особливо зручний для автоматизації тестування веб-додатків і API.

Cypress – це сучасний інструмент для тестування веб-додатків, який особливо популярний у фронтенд-розробці. Він забезпечує просте налаштування та швидку інтеграцію з проектом, підтримує інтерактивне тестування в реальному часі та дозволяє тестувати як інтерфейс користувача, так і API. Cypress чудово підходить для роботи з сучасними JavaScript/TypeScript фреймворками, такими як React, Angular і Vue.js.

Jenkins – це інструмент автоматизації, широко використовуваний у процесах CI/CD. Він дозволяє автоматизувати запуск тестів після кожного коміту, інтегрується з більшістю інструментів тестування та розгортання, а також підтримує плагіни для розширення функціональності. Jenkins часто застосовується для створення конвеєрів, які включають автоматичне тестування, збірку та розгортання додатків.

Apache JMeter – це інструмент для тестування продуктивності та навантаження, який дозволяє перевіряти продуктивність веб-додатків, баз даних і серверів. Він підтримує багатопотокове тестування та створення сценаріїв із великими обсягами даних. JMeter широко використовується для тестування систем під високим навантаженням.

Postman – це інструмент для тестування API, який забезпечує автоматизацію тестування RESTful API, підтримує колекції запитів і створення

автоматичних тестових сценаріїв. Він інтегрується з CI/CD, дозволяючи перевіряти API на кожному етапі розробки. Postman є простим у використанні та підходить як для початківців, так і для професіоналів.

Robot Framework – це гнучкий інструмент для автоматичного тестування, заснований на ключових словах. Він підтримує різні типи тестів, зокрема юніт-, інтеграційні та регресійні, і може інтегруватися з бібліотеками Python. Завдяки використанню ключових слів забезпечується створення читабельних тестів. Robot Framework зручний для великих команд, учасники яких мають різний рівень досвіду.

Allure – це інструмент для створення звітів про виконання автоматичних тестів. Він генерує зрозумілі та привабливі звіти, інтегрується з більшістю інструментів тестування, таких як JUnit, TestNG, PyTest, і відображає результати тестування у вигляді діаграм і графіків. Allure забезпечує зручну візуалізацію результатів, що сприяє аналітиці та прийняттю рішень.

8.2. Типи автоматичних тестів

На рис. 8.2 наведено основні типи тестів, що використовуються під час розроблення та розгортання програмних продуктів.

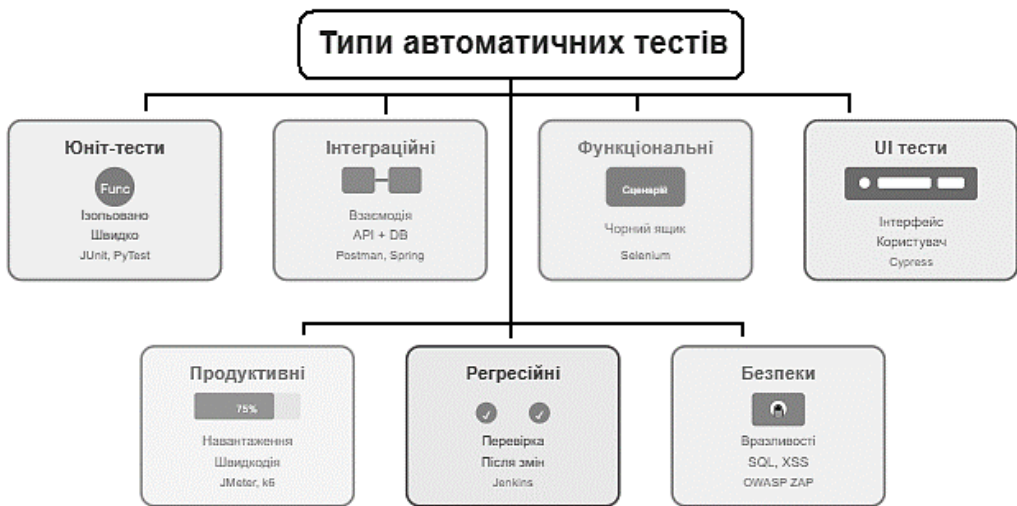


Рис. 8.2. Основні типи автоматичних тестів

Розглянемо коротко їх призначення та особливості проведення.

Юніт-тести (Unit Tests)

Юніт-тести перевіряють роботу окремих модулів або функцій програми в ізольованому середовищі, використовуючи макети (mocking) для заміщення

залежностей. Вони мають на меті виявлення помилок у конкретних частинах коду та забезпечення їхньої коректної роботи.

Так, наприклад, якщо є функція для підрахунку суми двох чисел, юніт-тест перевірить, чи повертається правильний результат для різних комбінацій введених значень.

До переваг юніт-тестів можна віднести відносну легкість і швидкість виявлення помилок на ранніх етапах розробки. Швидке виконання забезпечується тестуванням невеликих частин коду. Також спрощується рефакторинг, адже тести підтверджують, що зміни не порушили основний функціонал.

Серед інструментів для юніт-тестування можна виділити кілька популярних рішень. JUnit є найпоширенішим інструментом для написання та виконання тестів у Java-додатках. NUnit використовується для юніт-тестування у середовищі .NET. PyTest є універсальним фреймворком для написання тестів у Python, який також підтримує юніт-тестування. Для тестування коду на C++ часто використовують Google Test, який є популярним і ефективним інструментом.

Особливості юніт-тестів є наступними:

- виконуються ізольовано, без взаємодії з базами даних, API або зовнішніми сервісами;
- використовуються макети (mocking) для заміщення залежностей та перевірки тільки того коду, що тестується.

Нижче наведено приклад тесту для функції, яка визначає, чи є деяке число парним:

```
python

def is_even(num):
    return num % 2 == 0

def test_is_even():
    assert is_even(4) == True
    assert is_even(5) == False
```

Хоча юніт-тести є базовим типом тестування, вони часто поєднуються з іншими видами для забезпечення комплексного підходу.

Інтеграційні тести (Integration Tests)

Інтеграційні тести перевіряють взаємодію між окремими модулями або компонентами системи, щоб переконатися, що вони працюють разом правильно, включаючи передачу даних, логіку взаємодії та обробку помилок. Цей тип тестування є важливим для виявлення проблем, які можуть виникнути на стиках між модулями.

Інтеграційні тести перевіряють, як один модуль передає дані іншому, чи правильно виконується логіка взаємодії між компонентами, а також сумісність компонентів, які залежать один від одного.

Розглянемо деякі приклади використання тестів такого типу.

1) *Веб-додаток*: перевірка взаємодії між фронтендом (React, Angular, Vue.js) і бекендом (Node.js, Django, Spring). Наприклад, тест перевіряє, чи API повертає правильні дані, а фронтенд відображає їх коректно.

```
python
```

```
def test_api_integration():  
    response = requests.get("https://example.com/api/data")  
    assert response.status_code == 200  
    assert "name" in response.json()
```

2) *мобільний додаток*: перевірка роботи між мобільним клієнтом і сервером через REST або GraphQL API.

3) *бази даних*: тестування взаємодії між бекендом і базою даних, наприклад, перевірка, чи правильно записуються або читаються дані.

Виділяють такі основні підходи до інтеграційного тестування:

- **Big Bang Testing**: усі модулі інтегруються одночасно, і система тестується як є;
- **Incremental Testing**: модулі інтегруються поступово й тестуються на кожному кроці;
- **Top-Down Testing**: тестування починається з високорівневих модулів і поступово переходить до нижчого рівня;
- **Bottom-Up Testing**: спочатку тестуються модулі нижнього рівня, а потім додаються модулі верхнього рівня.

Переваги інтеграційних тестів полягають у можливості виявляти помилки в логіці взаємодії модулів. Вони забезпечують перевірку реальної поведінки системи, а не лише її окремих частин, а також дозволяють переконатися, що система відповідає вимогам замовника.

Розглянемо веб-додаток, який передбачає заповнення користувачем форми на фронтенді, а бекенд обробляє отримані дані й записує їх у базу даних. Інтеграційний тест може виглядати так (Python):

```
def test_form_submission_integration():  
    # Імітація заповнення форми на фронтенді  
    form_data = {"username": "test_user", "email": "test@example.com"}  
    response = requests.post("https://example.com/api/submit", json=form_data)  
  
    # Перевірка відповіді сервера  
    assert response.status_code == 201  
  
    # Перевірка, чи дані записані в базу  
    db_response = db.query("SELECT * FROM users WHERE email = 'test@example.com'")  
    assert db_response["username"] == "test_user"
```

Серед інструментів для інтеграційного тестування можна виділити кілька ефективних рішень. Так, наприклад, Postman використовується для тестування API, перевірки запитів та їх взаємодії з бекендом. SoapUI підходить для тестування SOAP і REST API, особливо для складних інтеграцій. Cucumber дозволяє писати тести у вигляді зрозумілих сценаріїв, що часто використовується в підході BDD. Spring Test є фреймворком для інтеграційного тестування Java-додатків із використанням Spring. Комбінація JUnit із Mock Server підходить для перевірки інтеграції з базами даних і зовнішніми API у Java-проектах. У Python для написання інтеграційних тестів можна використовувати PyTest разом із бібліотеками mock або requests.

Інтеграційні тести часто потребують налаштування тестового середовища, яке імітує реальні умови роботи системи. Це може включати використання локальних серверів, макетів API або тестових баз даних. Для складних систем із великою кількістю компонентів такі тести дозволяють виявляти проблеми, які важко передбачити на рівні юніт-тестування.

Функціональні тести (Functional Tests)

Функціональні тести перевіряють, чи працює програмне забезпечення відповідно до специфікацій і вимог. Основна мета — перевірити функціональність системи з точки зору користувача, забезпечуючи правильну роботу окремих функцій і сценаріїв.

Функціональні тести зосереджені на перевірці того, що система виконує відповідно до вимог користувача, не беручи до уваги внутрішню реалізацію

коду. Вони здійснюються за методом тестування чорного ящика, без необхідності глибокого розуміння внутрішньої реалізації коду. Основою таких тестів зазвичай є сценарії, що імітують реальну поведінку користувача.

Розглянемо деякі приклади функціональних тестів:

1) **Форма реєстрації користувача.** Користувач може зареєструватися, ввівши коректні дані (ім'я, email, пароль).

Сценарії тестування:

- заповнення всіх полів правильно – реєстрація має бути успішною;
- введення некоректного email – система має показати відповідне повідомлення про помилку;
- введення вже існуючого email – користувач отримує сповіщення, що такий email вже зареєстрований.

Нижче наведено приклад коду тесту (Python):

```
def test_user_registration():
    data = {"username": "test_user", "email": "test@example.com",
"password": "password123"}
    response = requests.post("https://example.com/api/register", json=data)
    assert response.status_code == 201
    assert response.json()["message"] == "User registered successfully."
```

2) **Авторизація.** Перевіряють: введення коректного логіна і пароля дозволяє отримати доступ. Неправильний пароль або логін – показується повідомлення про помилку.

3) **Додавання товару до кошика.** Додавання товару до кошика перевіряє кілька важливих аспектів. Перш за все, чи товар додається до кошика і чи його кількість відображається коректно. Також тестується правильність обчислення загальної суми. Окрім цього, перевіряється, що видалення товару з кошика працює без помилок.

4) **Обробка платежів.** Обробка платежів включає перевірку коректності введення платіжних даних і тестування завершення транзакції. Сценарії охоплюють як успішне, так і неуспішне виконання платіжних операцій.

Перевагами функціональних тестів є те, що вони забезпечують перевірку виконання всіх заявлених функцій системи. Такі тести легко зрозумілі замовникам, оскільки фокусуються на перевірці конкретних сценаріїв. Вони також допомагають переконатися, що продукт відповідає бізнес-вимогам.

До недоліків функціональних тестів можна віднести той факт, що вони охоплюють лише зовнішню функціональність, залишаючи без уваги внутрішню реалізацію. Крім того, для великих проєктів такі тести можуть бути часозатратними, якщо не використовується автоматизація.

Інструменти для функціонального тестування включають Selenium, який застосовується для автоматизації тестів веб-додатків та створення скриптів для перевірки інтерфейсу користувача. Cypress підходить для роботи з сучасними JavaScript-фреймворками та створення інтерактивних тестів. TestComplete є універсальним інструментом для тестування веб, мобільних і десктопних додатків. Postman використовується для перевірки функціональності REST API, а QTP/UFT від Micro Focus є популярним рішенням для тестування додатків у різних середовищах.

Тести інтерфейсу користувача (UI Tests)

Тести інтерфейсу користувача (UI Tests) спрямовані на перевірку вигляду, взаємодії та функціональності елементів користувацького інтерфейсу на різних пристроях і роздільних здатностях екранів. Їх основна мета – переконатися, що програма виглядає та працює відповідно до очікувань користувача.

Такі тести оцінюють коректність відображення елементів інтерфейсу, як-от кнопки, поля вводу чи таблиці, та перевіряють їхню реакцію на дії користувача, зокрема натискання, введення тексту чи навігацію. Вони також забезпечують відповідність дизайну до специфікацій, включаючи кольори, шрифти та розташування елементів.

Окрім цього, увага приділяється сумісності інтерфейсу з різними роздільними здатностями екранів, перевірки адаптивності та інтерактивних елементів, таких як випадаючі меню, чекбокси чи модальні вікна. Важливою частиною тестів є перевірка зворотного зв'язку, наприклад, відображення повідомлень про помилки при некоректному введенні даних.

Приклади використання:

1) **Форма входу**: перевірка, чи відображається поле для логіна та пароля. Тестування коректності кнопки «Увійти»: вона має стати активною лише після заповнення полів. Перевірка, чи з'являється відповідне повідомлення, якщо логін або пароль некоректні.

3) **Кошик у веб-магазині**: перевірка, чи можна додати товар до кошика. Перевірка, чи правильно оновлюється кількість товарів. Включають у себе тестування кнопок «Оформити замовлення» та «Видалити товар».

4) **Мобільний додаток**: перевірка коректного відображення елементів на різних пристроях (телефони, планшети). Передбачають тестування жестів: свайпи, довгі натискання тощо.

Інструменти для тестування інтерфейсу включають кілька популярних рішень. Selenium є одним із найпоширеніших фреймворків для автоматизації тестування веб-додатків. Він підтримує різні мови програмування, такі як Python, Java, та C#, і дозволяє автоматизувати взаємодію з елементами

інтерфейсу, наприклад, натискання кнопок, введення тексту або навігацію між сторінками.

Cypress – сучасний інструмент для автоматизації тестів веб-додатків. Він простий у налаштуванні, працює безпосередньо в браузері та підтримує інтеграцію з CI/CD платформами, такими як Jenkins чи GitHub Actions.

TestComplete є універсальним інструментом для UI-тестування десктопних, веб і мобільних додатків. Він дозволяє створювати тести як за допомогою запису дій користувача, так і через написання скриптів.

Appium спеціалізується на тестуванні інтерфейсу мобільних додатків для платформ Android та iOS. Він підтримує автоматизацію жестів та роботу з різними елементами мобільного інтерфейсу.

Playwright – новий інструмент для автоматизації браузерів, який підходить для роботи з сучасними веб-додатками, що активно використовують JavaScript. Він підтримує такі мови програмування, як JavaScript, Python і C#.

Переваги тестів інтерфейсу полягають у тому, що вони забезпечують перевірку системи з точки зору кінцевого користувача. Такі тести дозволяють виявляти проблеми, які можуть залишатися непоміченими на рівні юніт- або інтеграційних тестів, і охоплюють увесь користувацький сценарій, наприклад, від авторизації до оформлення замовлення.

Проте вони мають і певні недоліки. Тести інтерфейсу можуть бути досить часозатратними, особливо для великих проєктів. Вони потребують регулярного оновлення у разі змін у дизайні або логіці інтерфейсу, а також є складними для підтримки, якщо інтерфейс часто змінюється.

Нижче наведено приклад UI тесту: Перевірка кнопки «Додати в кошик»:

```
python

from selenium import webdriver

driver = webdriver.Chrome()
driver.get("https://example.com/product-page")

# Натискання кнопки "Додати в кошик"
add_to_cart_button = driver.find_element("id", "add-to-cart")
add_to_cart_button.click()

# Перевірка, чи товар додано до кошика
cart = driver.find_element("id", "cart-items")
assert "Product Name" in cart.text
driver.quit()
```

Продуктивні тести (Performance Tests)

Продуктивні тести призначені для оцінки ключових характеристик системи, таких як швидкодія, стабільність, масштабованість і надійність під різними навантаженнями.

Такі тести перевіряють швидкодію, зокрема час завантаження сторінок, відповіді сервера та виконання запитів до бази даних. Вони оцінюють масштабованість, визначаючи, як система реагує на збільшення кількості користувачів або обсягу даних. Тести також перевіряють стабільність, забезпечуючи виявлення збоїв при тривалих навантаженнях. Крім того, оцінюється пропускна здатність, тобто кількість запитів або операцій, які система здатна обробити за одиницю часу.

Виділяють такі основні типи продуктивних тестів:

1) *тести навантаження (Load Testing):*

- визначають, як система працює під нормальним і піковим навантаженням;
- приклад: 500 одночасних користувачів заходять на вебсайт і завантажують сторінку продукту.

2) *стрес-тестування (Stress Testing):*

- визначає, як система поводить себе за межами своїх можливостей (коли ресурси перевантажені);
- приклад: спроба обробити 10 000 одночасних запитів, коли сервер розрахований на 5 000.

3) *тестування стабільності (Stability/Soak Testing):*

- перевіряє, чи може система працювати тривалий час під постійним навантаженням без деградації продуктивності;
- приклад: сервер обробляє 1 000 запитів щогодини протягом 48 годин.

4) *тестування пропускної здатності (Capacity Testing):* визначає максимальну кількість користувачів або операцій, які система може обробити без деградації продуктивності.

Розглянемо детальніше деякі приклади продуктивних тестів:

1) **Тестування веб-додатка.**

Мета: Виміряти час завантаження головної сторінки для 1 000 користувачів, які одночасно відвідують сайт.

Інструмент: JMeter

```
bash

jmeter -n -t test-plan.jmx -l results.jtl
```

Результати аналізують у вигляді графіків часу відповіді для кожного запиту.

2) Тестування бази даних.

Мета: Перевірити, чи здатна база даних обробляти 10 000 запитів на секунду.

Інструмент: Gatling

```
scala

val scn = scenario("Database Load Test")
  .exec(http("Query DB")
    .get("/database/query"))
  setUp(scn.inject(atOnceUsers(10000))).protocols(httpProtocol)
```

Інструменти для продуктивного тестування включають кілька ефективних рішень. JMeter є одним із найпопулярніших інструментів для навантажувального тестування, підтримуючи протоколи HTTP, FTP, JDBC, SOAP, REST та інші. Він дозволяє створювати складні сценарії навантаження.

Gatling – потужний інструмент на основі мови Scala, який підходить для інтеграції в CI/CD процеси. Він забезпечує високу масштабованість і детальне логування результатів.

k6 – це простий у використанні інструмент для тестування продуктивності API, написаний на JavaScript, що робить його зручним для веб-розробників.

LoadRunner є корпоративним рішенням для стресового та навантажувального тестування з широкими можливостями аналізу та звітності.

Apache Bench (ab) пропонує легке рішення для швидкого тестування HTTP-запитів, а Locust, створений на основі Python, підтримує розробку сценаріїв користувацьких потоків для ефективного тестування навантаження.

Переваги тестів продуктивності полягають у тому, що вони допомагають виявити вузькі місця системи ще до її реального використання, підвищуючи надійність і стабільність продукту. Такі тести дозволяють прогнозувати поведінку системи за різних сценаріїв, забезпечуючи краще розуміння її можливостей.

Разом із тим, продуктивні тести мають і недоліки. Вони потребують значних ресурсів для моделювання реального навантаження, вимагають ретельного налаштування і точного вимірювання параметрів. Крім того, результати тестів можуть бути складними для інтерпретації.

Регресійні тести (Regression Tests)

Регресійні тести призначені для перевірки того, чи впровадження нових змін або функціональностей не порушило вже існуючий функціонал. Вони є одним із ключових елементів забезпечення якості, особливо в проєктах із частими оновленнями, де ризик виникнення побічних ефектів досить високий.

Мета таких тестів – гарантувати, що програмний продукт працює так само добре після внесення модифікацій у код, як і раніше. Це дозволяє зберігати стабільність системи та уникати неочікуваних помилок у функціональності.

Нижче наведено деякі приклади використання регресійних тестів:

- після виправлення помилки проводяться регресійні тести, щоб переконатися, що це виправлення не спричинило проблем в інших частинах програми;
- перед випуском нової версії програми регресійні тести допомагають перевірити, чи всі основні функції працюють належним чином.

Інструменти для регресійного тестування включають кілька ефективних рішень. Selenium забезпечує автоматизоване тестування веб-додатків і використовується для перевірки регресії. TestComplete є універсальним інструментом для автоматизації як функціональних, так і регресійних тестів. PyTest спеціалізується на регресійному тестуванні Python-додатків. Jenkins дозволяє інтегрувати регресійні тести в процеси CI/CD, забезпечуючи постійний моніторинг стабільності системи.

Регресійні тести можуть бути повними (всі частини системи перевіряються) або вибічковими (тестуються лише частини, які могли постраждати від змін). Їх автоматизація є надзвичайно важливою в сучасних підходах до розробки, таких як Agile або DevOps, оскільки вона знижує витрати часу та людських ресурсів, забезпечуючи водночас якість та стабільність.

Тести безпеки (Security Tests)

Тести безпеки є важливим інструментом для забезпечення надійності та захисту програмного забезпечення від потенційних загроз, атак чи зловживань. Їх головна мета – виявлення вразливостей, перевірка рівня безпеки та забезпечення відповідності додатка сучасним стандартам кібербезпеки.

Такі тести допомагають ідентифікувати можливі точки входу для атак, зокрема SQL-ін'єкції, XSS чи CSRF. Вони перевіряють коректність реалізації механізмів автентифікації та авторизації, оцінюють рівень захищеності конфіденційних даних, таких як паролі, фінансова інформація або персональні дані користувачів. Крім того, тести безпеки забезпечують перевірку правильного використання шифрування та інших протоколів безпеки, а також визначають здатність системи протистояти атакам типу DoS/DDoS.

Нижче наведено деякі приклади тестування:

- перевірка, чи можуть неавторизовані користувачі отримати доступ до захищених ресурсів;
- моделювання атак, наприклад, SQL-ін'єкцій або міжсайтового виконання скриптів (XSS);
- оцінка того, як система поводить себе в умовах підвищеного навантаження або при спробах зломисників зламати шифрування даних.

Можна виділити наступні інструменти для тестування безпеки:

- OWASP ZAP (Zed Attack Proxy) – це інструмент з відкритим кодом для пошуку вразливостей у веб-додатках. Він дозволяє виконувати сканування, аналізувати безпеку та виявляти слабкі місця.
- Burp Suite – потужний комплекс інструментів для тестування безпеки, який підтримує моделювання атак, перевірку вразливостей і тестування взаємодії між клієнтом та сервером.
- Nessus – інструмент для сканування мережі та виявлення вразливостей у конфігураціях.
- Metasploit – платформа для моделювання атак та перевірки захисту системи.
- SQLMap – автоматизований інструмент для виявлення SQL-ін'єкцій.
- Arachni – інструмент для тестування безпеки веб-додатків, який виявляє загальні типи вразливостей.

Переваги тестів безпеки включають забезпечення захищеності системи від зломисників, виявлення потенційних вразливостей до їх експлуатації, дотримання стандартів і норм безпеки, таких як GDPR або PCI DSS. Крім того, вони сприяють зниженню репутаційних і фінансових ризиків, пов'язаних із витоком даних або кібератаками.

Регулярне тестування безпеки особливо важливе для фінансових, медичних і державних систем, де захист даних має критичне значення. Автоматизація тестів безпеки у поєднанні з регулярним ручним аудитом допомагає забезпечити найвищий рівень захисту додатків.

8.3. Автоматичні тести в CI/CD

Автоматичне тестування є ключовим елементом у процесах CI/CD, оскільки забезпечує ефективність і надійність розробки програмного забезпечення. Його використання дозволяє командам швидко і безпечно впроваджувати зміни, зберігаючи високу якість коду.

Стабільність коду є одним із основних досягнень завдяки автоматичному тестуванню. Тести дозволяють оперативно виявляти дефекти та помилки до

того, як код потрапить у продуктивне середовище. Це знижує ризик збоїв системи, підвищує довіру користувачів і зменшує витрати на виправлення помилок на пізніх етапах.

Швидка перевірка змін допомагає розробникам зосередитися на створенні нової функціональності, не витрачаючи багато часу на ручне тестування. Автоматизація дозволяє виконувати сотні або навіть тисячі тестів одночасно, прискорюючи процес валідації функціоналу. Завдяки цьому кожна зміна може бути перевірена відразу після її внесення.

Запобігання помилкам на ранніх етапах розробки є критичним завданням автоматичного тестування. Завдяки детальному покриттю коду тестами вдається уникати серйозних проблем у продуктивному середовищі. Автоматичні тести діють як запобіжний механізм, який блокує впровадження коду з помилками, що забезпечує надійність усієї системи.

Крім основних функцій, автоматичне тестування в CI/CD сприяє більшій узгодженості командної роботи. Усі члени команди мають доступ до результатів тестів і можуть вчасно реагувати на виявлені проблеми. Інтеграція тестів із CI/CD інструментами створює єдиний робочий процес, що об'єднує розробку, тестування та розгортання.

Автоматичне тестування також дозволяє створювати стабільні релізи навіть у проєктах із високою частотою змін. Використання автоматизації зменшує людський фактор і забезпечує постійний моніторинг стану коду, що особливо важливо для Agile- та DevOps-команд. У підсумку, автоматичне тестування є невід'ємною частиною CI/CD процесів, яка забезпечує їхню ефективність і стабільність.

Таким чином, автоматичні тести є ключовою частиною процесів CI/CD, забезпечуючи контроль якості коду на кожному етапі його впровадження. Робочий процес автоматичного тестування починається з внесення змін у код і завершується розгортанням стабільного програмного забезпечення в продуктивному середовищі.

Коли розробник створює або змінює код, він комітить його в репозиторій, наприклад, GitHub, GitLab або іншу систему контролю версій. Після кожного коміту CI/CD система автоматично реагує, активуючи визначені конвеєри тестування.

Система CI/CD (наприклад, Jenkins, GitLab CI/CD, CircleCI або Travis CI) запускає серію автоматичних тестів. Це можуть бути різні типи тестів, як-от:

- юніт-тести, які перевіряють окремі функції або модулі на базовому рівні;
- інтеграційні тести, що перевіряють взаємодію між різними модулями чи компонентами;

- тести продуктивності, які аналізують швидкість системи;
- тести безпеки, спрямовані на виявлення вразливостей у системі тощо.

Якщо всі тести проходять успішно, CI/CD система розгортає зміни в тестовому середовищі для додаткової перевірки або для роботи QA-команди. На цьому етапі також можуть запускатися додаткові автоматизовані сценарії тестування, такі як регресійні або функціональні тести.

У разі успішного проходження тестування в тестовому середовищі система автоматично або з мінімальним втручанням команди розробки деплоїть код у продуктивне середовище. Якщо ж один або кілька тестів завершуються невдачею, процес розгортання блокується, а розробник отримує звіт із деталями про знайдені помилки. Це дозволяє оперативно виправити проблему і повторити процес.

Такий підхід до автоматичного тестування в CI/CD допомагає не лише виявляти помилки на ранніх етапах, але й забезпечувати безперервну інтеграцію та стабільну доставку програмного забезпечення. Він є невід'ємною частиною сучасного DevOps-підходу, де швидкість і якість розробки мають критичне значення.

Інструменти для автоматизації в CI/CD

Інструменти для автоматизації CI/CD відіграють важливу роль у забезпеченні безперервної інтеграції, тестування та розгортання коду. Вони автоматизують рутинні процеси, прискорюючи доставку програмного забезпечення та знижуючи ризик людських помилок. Нижче наведено деякі приклади таких інструментів.

Jenkins – один із найпопулярніших відкритих інструментів для CI/CD. Він забезпечує інтеграцію з різними мовами програмування, системами контролю версій та сторонніми плагінами. Jenkins дозволяє створювати складні конвеєри з багатьма етапами тестування, збірки та розгортання.

GitLab CI/CD – це вбудований інструмент у GitLab, який дозволяє розробникам легко налаштовувати CI/CD конвеєри. GitLab підтримує інтеграцію з репозиторіями, автоматичне тестування, побудову та деплой на будь-які середовища. Завдяки зрозумілому YAML-файлу конфігурації, цей інструмент є ідеальним вибором для команд, які працюють безпосередньо з GitLab.

CircleCI – хмарне рішення для CI/CD, яке підтримує різні мови програмування та платформи. CircleCI вирізняється високою швидкістю виконання конвеєрів і гнучкістю у налаштуванні. Він ідеально підходить для невеликих і середніх команд, які потребують потужного інструменту для автоматизації, але не хочуть розгортати власну інфраструктуру.

Travis CI – простий у використанні сервіс для автоматизації тестування та розгортання. Travis CI інтегрується з GitHub і дозволяє швидко налаштувати автоматичні процеси. Це зручний вибір для проєктів з відкритим кодом завдяки безкоштовній підтримці для таких репозиторіїв.

Azure Pipelines – хмарний сервіс CI/CD, який є частиною екосистеми Azure DevOps від Microsoft. Azure Pipelines дозволяє створювати автоматизовані конвеєри для збірки, тестування та розгортання коду на різних платформах. Він пропонує гнучкі можливості для налаштування CI/CD процесів та інтеграцію з Azure Cloud сервісами, а також підтримує роботу з різними мовами програмування та системами контролю версій.

Кожен із цих інструментів має свої переваги та підходить для різних потреб команд. Вибір інструменту залежить від розміру команди, складності проєкту та інфраструктури, з якою працює команда.

Приклад CI/CD конвеєра

CI/CD конвеєр автоматизує процеси інтеграції, тестування та розгортання, забезпечуючи безперервну доставку якісного коду. Нижче наведено детальний приклад етапів такого конвеєра:

Розробник комітить код у репозиторій (наприклад, GitHub, GitLab або Bitbucket). При цьому нові зміни автоматично відстежуються CI/CD системою.

На першому етапі автоматично запускаються юніт-тести, які перевіряють базову функціональність окремих модулів або компонентів. Це забезпечує швидке виявлення помилок у коді на ранньому етапі.

Далі система переходить до інтеграційних тестів, які перевіряють взаємодію між різними модулями або сервісами. Це гарантує, що новий код не порушує роботу залежностей або інтеграцій між компонентами.

На наступному етапі виконуються продуктивні тести, які перевіряють продуктивність, стабільність і витривалість системи під навантаженням. Наприклад, тестується час відгуку сервісів або обробка великого обсягу даних.

У разі успішного проходження всіх тестів система автоматично розгортає зміни в тестовому середовищі. Це середовище використовується для додаткової перевірки вручну, а також для кінцевих автоматичних тестів, таких як регресійні або тести безпеки.

Після підтвердження стабільності в тестовому середовищі код автоматично деплоїться в продуктивне середовище. Це дозволяє користувачам миттєво отримати доступ до нових функцій або виправлених помилок.

Цей процес може також включати додаткові етапи, такі як:

- автоматичне створення повідомлень про результати тестування;
- перевірка якості коду за допомогою статичного аналізу;

- створення резервних копій перед розгортанням;
- сповіщення команди про статус конвеєра через Slack, електронну пошту або інші інструменти.

Такий підхід дозволяє швидко доставляти якісний код, скорочуючи час між написанням змін і їхнім використанням кінцевими користувачами.

Переваги автоматичного тестування в CI/CD:

Автоматичне тестування є ключовим елементом процесу CI/CD, надаючи низку значних переваг для команд розробки:

- *швидкий зворотний зв'язок*: автоматичне тестування забезпечує розробникам миттєве повідомлення про стан змін у коді, включаючи інформацію про виконання умов та залежностей. Це дозволяє оперативно виявляти та виправляти помилки, зберігаючи високий темп розробки;
- *підтримка високої частоти випусків*: завдяки автоматизації перевірок код можна швидко доставляти у вигляді нових функцій чи оновлень, не витрачаючи час на ручне тестування. Це особливо актуально для проєктів, що працюють у рамках Agile або DevOps підходів;
- *мінімізація людських помилок*: ручне тестування часто супроводжується ризиком пропущених дефектів або помилок через людський фактор. Автоматизація дозволяє уникнути таких ситуацій, забезпечуючи повторюваність і точність процесу;
- *економія часу та ресурсів*: виконання тестів автоматично значно скорочує час перевірки коду та дозволяє командам зосередитися на створенні нових функцій. Це також знижує потребу у великих командах тестувальників;
- *покращення надійності розгортань*: правильно налаштована інтеграція тестування в конвеєр CI/CD гарантує, що тільки якісний і перевірений код потрапляє у продуктивне середовище. Це мінімізує ризик дефектів у релізах;
- *прозорість процесу розробки*: автоматичне тестування вбудовується в CI/CD конвеєр, що дозволяє всім учасникам команди стежити за статусом тестів, розгортанням і потенційними проблемами в реальному часі;
- *масштабованість*: автоматизовані тестові системи легко адаптуються до зростання проєкту, дозволяючи ефективно обробляти збільшення кількості тестів, функціональності та інтеграцій.

Інтеграція автоматичних тестів у CI/CD значно підвищує ефективність, надійність та швидкість розробки. Це робить процес більш керованим і

прозорим, що є особливо важливим для великих команд або проєктів із високими вимогами до якості та швидкості.

Контрольні питання до розділу 8

- 1) Що таке автоматичне тестування і які переваги воно має?
- 2) Що собою являють тестові скрипти, і які типи тестів вони можуть виконувати?
- 3) З якою метою проводять функціональні тести ?
- 4) Яким чином автоматизація тестування інтегрується в процеси CI/CD?
- 5) Яке призначення інструменту Selenium?
- 6) Які основні типи автоматичних тестів використовуються в розробці програмного забезпечення?
- 7) Що перевіряють UI тести?
- 8) Для чого зазвичай використовується Apache JMeter?
- 9) З якою метою проводяться інтеграційні тести?
- 10) Як регресійні тести допомагають зберігати стабільність системи?
- 11) Що є ключовою перевагою автоматичного тестування?
 - a) Тестування лише в ізольованих середовищах
 - b) Тестування великих проєктів більш вартісне
 - c) Зменшення впливу людського фактора
 - d) Використання мінімальної кількості ресурсів
 - e) Неможливість повторного запуску скриптів
- 12) Що таке автоматичне тестування?
 - a) Тестування за допомогою систем ШІ
 - b) Автоматизація тестів за допомогою спеціальних інструментів
 - c) Тестування тільки інтерфейсу користувача
 - d) Тестування лише під час розгортання продукту
 - e) Процес тестування ІС
- 13) Який етап CI/CD зазвичай включає виконання інтеграційних тестів?
 - a) Злиття коду (Merge).
 - b) Створення документації (Documentation).
 - c) Тестування (Test Stage)
 - d) Моніторинг (Monitoring)
 - e) Розгортання (Deployment Stage)
- 14) Який з інструментів використовується для створення звітів про виконання тестів?
 - a) CircleCI
 - b) Cypress
 - c) Jenkins

- d) Allure
 - e) TestNG
- 15) Який підхід до тестування починається з високорівневих модулів?
- a) Bottom-Up Testing
 - b) Big Bang Testing
 - c) Top-Down Testing
 - d) Incremental Testing
 - e) Unit Testing
- 16) Який тип тестів перевіряє роботу окремих модулів програми в ізоляції?
- a) Інтеграційні тести
 - b) Unit -тести
 - c) UI тести
 - d) Функціональні тести
 - e) Системні тести
- 17) Який тип тестів перевіряє стабільність системи під час навантаження?
- a) Інтеграційні тести
 - b) Unit -тести
 - c) UI тести
 - d) Функціональні тести
 - e) Performance тести
- 18) Що перевіряють тести безпеки?
- a) Автоматизацію процесів CI/CD
 - b) Вразливості системи, механізми автентифікації та авторизації
 - c) Швидкодію та стабільність системи під навантаженням
 - d) Вплив нових змін на існуючий функціонал
 - e) Масштабованість системи
- 19) Яке тестування може бути першим етапом у CI/CD конвеєрі після коміту коду?
- a) Інтеграційні тести
 - b) Unit -тести
 - c) UI тести
 - d) Тести безпеки
 - e) Регресійні тести
- 20) Який з вказаних інструмент використовується для автоматизації процесів CI/CD?
- a) Jenkins
 - b) Selenium
 - c) Apache JMeter
 - d) Postman

21) Встановіть відповідність між етапами CI/CD та типами тестування, які зазвичай виконуються на кожному з них:

A. Етап інтеграції (Integration Stage)	1) Виконання продуктивних тестів для оцінки стабільності системи.
B. Етап тестування (Test Stage)	2) Виконання інтеграційних тестів для перевірки взаємодії між модулями.
C. Етап розгортання (Deployment Stage)	3) Виконання регресійних тестів для перевірки, чи нові зміни не порушили існуючий функціонал.
D. Етап моніторингу (Monitoring Stage)	4) Тестування середовища розгортання та перевірка відповідності вимогам.

22) Встановіть відповідність між підходом до інтеграційного тестування та його характеристикою:

A. Big Bang Testing	1) Поступова інтеграція модулів.
B. Top-Down Testing	2) Одночасна інтеграція всіх модулів.
C. Bottom-Up Testing	3) Початок з нижчого рівня.
D. Incremental Testing	4) Початок з вищого рівня.

23) Встановіть відповідність між типами тестів і їхнім призначенням:

A. Unit -тести	1) Виявляють вразливості системи.
B. Інтеграційні тести	2) Перевіряють взаємодію між модулями.
C. Регресійні тести	3) Перевіряють, чи нові зміни не порушують функціонал.
D. Тести безпеки	4) Оцінюють окремі функції програми.

24) Встановіть відповідність між вказаними тестами продуктивності з їх призначенням:

A. Load Testing	1) Перевірка роботи системи під тривалим навантаженням.
B. Stress Testing	2) Тестування за межами можливостей системи.
C. Stability Testing	3) Перевірка роботи під нормальним навантаженням.
D. Capacity Testing	4) Визначення максимальної кількості користувачів.

25) Встановіть відповідність між типом тестування та його характеристикою:

A. Автоматичне тестування	1) Перевірка функціональності системи з точки зору користувача.
B. Функціональне тестування	2) Використання спеціальних скриптів.
C. Ручне тестування	3) Тестування інтерфейсу застосунку.
D. UI тестування	4) Потребує втручання людини у проведення тестів.

РОЗДІЛ 9. ОСНОВИ КОД АНАЛІЗУ

Код-аналіз – це процес автоматизованої перевірки вихідного коду на відповідність заданим стандартам, правилам та можливим дефектам. Він є важливим компонентом сучасного циклу розробки програмного забезпечення, особливо в умовах швидких темпів змін, які забезпечує неперервна інтеграція.

У контексті неперервної інтеграції, код-аналіз виконується автоматично кожного разу, коли розробник додає або змінює код у репозиторії. CI – це практика, яка дозволяє автоматично збирати, тестувати та перевіряти програмний продукт після внесення змін у його вихідний код. Код-аналіз забезпечує декілька ключових функцій:

- *раннє виявлення помилок*: завдяки автоматизації перевірки вдається виявляти потенційні помилки ще до того, як код буде інтегрований у основну гілку проєкту;
- *підтримка відповідності стандартів*: автоматизовані правила стилю та якості коду дозволяють команді дотримуватися узгоджених стандартів, зменшуючи кількість конфліктів;
- *підвищення якості продукту*: аналізатори допомагають знайти «мертвий» код, проблеми продуктивності та потенційні уразливості безпеки, покращуючи загальний стан системи;
- *прискорення розробки*: розробники отримують швидкий зворотний зв'язок про свої зміни, що зменшує час на пошук і виправлення дефектів на пізніх стадіях розробки;
- *захист від технічного боргу*: регулярний код-аналіз дозволяє контролювати накопичення неякісного коду, що ускладнює підтримку системи.

9.1. Завдання код-аналізу

Аналіз коду дозволяє виявляти потенційні проблеми на ранніх етапах, що значно зменшує витрати на їх виправлення та мінімізує ризики. З поміж таких проблем можна виділити наступні:

1. *синтаксичні помилки* – автоматичне виявлення помилок у структурі коду, які можуть спричинити його некоректну роботу;
2. *порушення стандартів* – забезпечення відповідності стилю та структури коду встановленим правилам та гайдам, що покращує читабельність і підтримку;
3. *проблеми продуктивності* – ідентифікація частин коду, які можуть уповільнювати виконання програми;

4. *уразливості безпеки* – пошук потенційних точок для атак, таких як SQL-ін'єкції чи міжсайтові сценарії (XSS);
5. *інші потенційні проблеми* – аналіз дублювання коду, некоректних залежностей чи надмірної складності алгоритмів тощо.

Синтаксичні помилки

Синтаксичні помилки – це помилки у структурі коду, які зазвичай виникають через неправильно написаний код (наприклад, пропущені дужки, неправильне використання ключових слів чи операторів). Такі помилки призводять до того, що програма не компілюється або аварійно завершується на ранніх етапах виконання.

В якрсті прикладу можна навести наступні ситуації:

- використання змінної, яка не була оголошена;
- пропущений символ крапки з комою (;) у мовах, які цього вимагають.

Автоматизовані інструменти, такі як SonarQube або ESLint, можуть виявляти синтаксичні помилки під час перевірки коду, вказуючи точне місце помилки та навіть пропонуючи виправлення.

Порушення стандартів

Порушення стандартів стосуються відхилень від загальноприйнятих або командних правил написання коду. Це може бути відсутність форматування, неправильна структура функцій або змінні, що не відповідають угодам про іменування.

Чітка структура коду має велике значення, оскільки погано організований код складніше читати, розуміти та підтримувати. У командних проєктах важливо дотримуватись єдиного підходу до написання коду, щоб уникнути плутанини та забезпечити ефективну співпрацю.

В якості прикладів існуючих стандартів можна навести дотримання PEP 8 у Python та стандарти Airbnb або Google у JavaScript.

Такі інструменти, як Checkstyle (Java), Pylint (Python) або Prettier (JavaScript), автоматично перевіряють стиль коду та надають рекомендації щодо його покращення, а також проводять контроль за дотриманням угод про іменування та організацію імпортів.

Проблеми продуктивності

Проблеми продуктивності виникають, коли код працює повільніше, ніж очікується внаслідок неефективних алгоритмів, зайвих обчислень або нераціонального використання ресурсів.

Типовими проблемами є використання застарілих або неефективних методів для роботи з великими обсягами даних, невиправдане застосування циклів чи рекурсій, а також недостатня оптимізація запитів до бази даних.

Прикладом такої проблеми може бути отримання даних з бази в одному циклі, за умови, що це можна зробити одним запитом.

Модулі статичного аналізу (SonarQube) та такі інструменти, як Dynatrace, AppDynamics дозволяють виявити вузькі місця (bottlenecks) та пропонують шляхи оптимізації.

Уразливості безпеки

Уразливості безпеки виникають, коли в коді присутні недоліки, які можуть бути використані зловмисниками для атак на систему. Вони є одним із найважливіших аспектів аналізу, оскільки безпосередньо впливають на захищеність продукту.

В якості найпоширеніших ситуацій можна навести:

- SQL-ін'єкції: зловмисники маніпулюють запитами до бази даних;
- міжсайтовий скриптинг (XSS): ін'єкція шкідливих скриптів у веб-додаток;
- уразливості в аутентифікації: наприклад, слабке хешування паролів;
- помилки у доступі до пам'яті: наприклад, переповнення буфера (Buffer Overflow).

Так, наприклад, наведений нижче запит може дозволити зловмиснику обійти авторизацію:

```
sql

SELECT * FROM users WHERE username = 'admin' OR '1'='1';
```

Спеціальні інструменти, такі як OWASP Dependency-Check, Bandit (Python) та модулі в SonarQube, сканують код на наявність уразливостей безпеки, перевіряють залежності та пропонують оновлення.

Інші потенційні проблеми

Крім вищезазначених проблем, код-аналіз може виявляти:

- дублювання коду: один і той самий код, написаний кілька разів, що ускладнює його підтримку;
- «мертвий код»: фрагменти коду, які більше не використовуються, але залишаються в репозиторії;
- логічні помилки: наприклад, неправильне порівняння або помилкова умова в if-конструкції.

Інтеграція аналізу коду у процеси розробки дозволяє не лише покращити якість продукту, але й підвищити продуктивність розробників, знизити

технічний борг та забезпечити безпеку програмного забезпечення. Це робить код-аналіз одним із ключових етапів сучасного CI/CD.

Переваги код-аналізу в CI

Код-аналіз є важливим етапом у сучасних підходах до розробки програмного забезпечення, зокрема в контексті неперервної інтеграції (CI). Інструменти код-аналізу забезпечують численні переваги, які покращують якість коду, оптимізують робочий процес і знижують витрати. Нижче наведено опис переваг:

Автоматизація перевірок підвищує швидкість і точність

Автоматизований код-аналіз інтегрується у процеси CI/CD і працює без втручання людини. Це дозволяє:

- *зменшити людський фактор*: інструменти перевірки не схильні до втоми чи неуважності, тому аналіз є більш надійним порівняно з ручним;
- *прискорити роботу*: автоматизація дозволяє ініціювати перевірки одразу, як тільки розробник додає зміни до репозиторію та проводити їх досить швидко. Наприклад, перевірка коду в масштабному проєкті може тривати кілька хвилин, тоді як вручну це займе години;
- *масштабованість*: інструменти можуть обробляти великі обсяги коду незалежно від його складності, не впливаючи на час розробки.

Так, наприклад, інтеграція SonarQube з Jenkins дозволяє запускати аналіз після кожного коміту, формувати звіти та відразу ж повідомляти розробника про помилки.

Раннє виявлення дефектів знижує витрати на їхнє усунення

Згідно з принципом, відомим як «вартість виправлення помилок», чим раніше дефект виявлено, тим дешевше його усунути. Код-аналіз, інтегрований у CI, допомагає:

- виявляти проблеми ще до етапу тестування або релізу, включаючи залежності та потенційні конфлікти з іншими змінами;
- запобігати накопиченню технічного боргу, який може сповільнити розвиток продукту;
- зменшувати кількість дефектів, які досягають середовища користувача, що покращує репутацію продукту.

Якщо, наприклад, статичний аналізатор виявить уразливість SQL-ін'єкції на ранньому етапі, її виправлення може коштувати лише кілька хвилин роботи розробника. Проте якщо така помилка досягне продакшну, її виправлення може

вимагати великих витрат, включаючи витрати на підтримку, компенсацію клієнтам або навіть штрафи.

Забезпечення єдиного стандарту коду для всієї команди

У великих командах, де над проєктом працюють десятки або сотні розробників, важливо дотримуватися єдиних стандартів написання коду, враховуючи специфічні вимоги проєкту. Код-аналіз забезпечує:

- *контроль за стилем*: перевірка відповідності коду стилістичним вимогам, наприклад, PEP 8 для Python або правил Airbnb для JavaScript;
- *покращення читабельності*: єдиний стиль коду полегшує розуміння та підтримку проєкту всією командою;
- *легкість у рев'ю*: коли код стандартизований, його легше аналізувати під час code review, що економить час.

Спеціалізовані інструменти код-аналізу, такі як Prettier або ESLint, автоматично форматують код перед комітом, забезпечуючи єдиний стиль для всієї команди.

Підвищення безпеки та надійності продукту

Код-аналіз дозволяє виявляти уразливості, які можуть бути використані зловмисниками. Завдяки інтеграції з CI:

- уразливості, такі як SQL-ін'єкції або міжсайтовий скриптинг (XSS), виявляються на ранніх етапах;
- інструменти перевіряють залежності та бібліотеки на наявність відомих проблем.

Спеціалізовані інструменти код-аналізу, такі як, наприклад, OWASP Dependency-Check автоматично сканують залежності на наявність уразливостей і попереджають команду про необхідність оновлення.

Зниження ризиків пов'язаних із якістю коду

Інструменти код-аналізу зменшують ризик інтеграції поганого коду в основну гілку проєкту:

- забезпечують зупинку CI/CD-пайплайну, якщо виявлено критичні помилки;
- дозволяють командам фокусуватися на вирішенні задач замість ручного пошуку дефектів.

Так, наприклад, якщо система CI виявляє дублювання коду чи «мертвий код», це дозволяє оптимізувати продукт ще до його виходу в продакшн.

Підвищення мотивації та ефективності команди

Код-аналіз спрощує роботу розробників, оскільки:

- забезпечує зворотний зв'язок у реальному часі;
- зменшує кількість «нудної» ручної роботи;

- полегшує адаптацію нових розробників, які одразу можуть слідувати стандартам команди.

Автоматизований код-аналіз у контексті СІ не лише спрощує розробку програмного забезпечення, але й забезпечує високий рівень якості продукту. Завдяки ранньому виявленню дефектів, підтримці єдиного стандарту коду та підвищенню швидкості розробки, компанії можуть випускати продукти швидше, безпечніше та з меншими витратами.

9.2. Типи код-аналізу

Код-аналіз поділяється на два основних типи: статичний і динамічний. Кожен із них має свої особливості, переваги й застосування, які допомагають виявляти різні аспекти якості коду.

Статичний аналіз коду (Static Code Analysis)

Статичний аналіз проводиться *без запуску програми*. Його основна мета – перевірка вихідного коду на відповідність стандартам, виявлення потенційних помилок, які можуть виникнути під час виконання, та забезпечення структурної цілісності коду.

Основні особливості такого аналізу:

- виконується автоматично на основі аналізу синтаксису та структури коду;
- застосовується ще на етапі написання коду або під час коміту змін у репозиторій;
- дозволяє швидко виявляти потенційні помилки, такі як неправильне використання змінних, відсутність перевірок тощо.

Завдання статичного аналізу:

- *перевірка стилю коду*: дотримання стандартів, наприклад, PEP 8 для Python або Google Java Style Guide;
- *виявлення «мертвого» коду*: фрагментів, які більше не використовуються і можуть бути видалені;
- *попередження потенційних помилок*: наприклад, можливі NullPointerException у Java або використання неініціалізованих змінних;
- *перевірка на відповідність безпековим стандартам*: виявлення потенційних загроз безпеці, наприклад, неправильного оброблення вхідних даних.

В якості ефективних інструментів статичного аналізу можна виділити:

- SonarQube: сканує код на наявність помилок, проблем продуктивності, дублювання та інших недоліків;

- ESLint: аналізує JavaScript-код на відповідність стилістичним стандартам і потенційним помилкам;
- Pylint: автоматично перевіряє Python-код на стиль, помилки та дублювання.

Статичний аналіз має свої переваги та недоліки. Серед переваг можна виділити швидкий аналіз із миттєвим зворотним зв'язком, зменшення кількості дефектів на ранніх етапах розробки та сприяння узгодженості стилю коду в команді. Проте статичний аналіз не здатен виявляти помилки, що проявляються лише під час виконання, і його ефективність залежить від якості та налаштувань правил для аналізу.

Динамічний аналіз коду (Dynamic Code Analysis)

Динамічний аналіз проводиться *під час виконання програми*. Він дозволяє оцінити поведінку програми в реальних умовах, виявити помилки, які проявляються лише в процесі виконання, та протестувати продуктивність системи, а також перевірити відповідність вимогам до безпеки та масштабованості.

Основні особливості:

- вимагає запуску програми або її частин для аналізу;
- використовується для оцінки взаємодії компонентів програми, виявлення помилок у виконанні, продуктивності та відповідності вимогам;
- часто поєднується з тестуванням, наприклад, юніт-тестами, інтеграційними тестами чи тестуванням навантаження.

Завдання динамічного аналізу є наступними:

- *аналіз продуктивності*: виявлення «вузьких місць» (bottlenecks), що сповільнюють роботу системи;
- *перевірка на витoki пам'яті*: наприклад, у мовах C++ або Java, де можливе некоректне використання ресурсів;
- *тестування з реальними даними*: наприклад, перевірка, як програма обробляє великі обсяги даних чи специфічні сценарії;
- *виявлення помилок виконання*: помилки, які виникають лише за певних умов, наприклад, переповнення буфера або вихід за межі масиву.

Можна відзначити наступні інструменти динамічного аналізу:

- JProfiler: аналізує продуктивність Java-додатків і витрати пам'яті;
- Valgrind: інструмент для динамічного аналізу програм, написаних на C/C++, зокрема для пошуку витоків пам'яті;

- AppDynamics: дозволяє моніторити продуктивність додатків у реальному часі, а також аналізувати взаємодію між компонентами та виявляти вузькі місця.

Основними перевагами динамічного аналізу є здатність виявляти помилки, які неможливо знайти за допомогою статичного аналізу, оцінка продуктивності та масштабованості програми, а також підвищення надійності продукту завдяки тестуванню в реальних умовах. Водночас динамічний аналіз зазвичай займає більше часу, потребує виконання програми або тестових сценаріїв і може вимагати значних обчислювальних ресурсів для роботи зі складними системами.

Отже, аналіз коду є важливою складовою процесу забезпечення якості програмного забезпечення. Залежно від мети та етапу розробки, використовується два основних типи аналізу: статичний та динамічний. Кожен із них має свої переваги, обмеження та специфічні випадки застосування.

Статичний аналіз коду виконується без запуску програми, дозволяючи ідентифікувати потенційні проблеми на етапі написання коду. Він зосереджується на синтаксисі, стандартах та структурі коду.

Динамічний аналіз коду здійснюється під час виконання програми, що дозволяє виявити проблеми, які проявляються лише в реальному середовищі, наприклад, уразливості безпеки або збої через неправильне використання ресурсів.

В таблиці 9.1 наведено порівняння цих підходів, яке допоможе зрозуміти їх ключові особливості та сфери застосування.

Статичний і динамічний аналізи доповнюють один одного, забезпечуючи всебічну перевірку коду. Використання обох підходів у процесах CI/CD дозволяє виявляти помилки на різних етапах розробки, підвищувати якість продукту та забезпечувати його стабільну роботу.

Таблиця 9.1. Порівняння статичного та динамічного аналізу

Характеристика	Статичний аналіз	Динамічний аналіз
Виконання	Без запуску програми	Під час виконання програми
Виявлення помилок	Структурні помилки, стиль, безпека	Проблеми виконання, продуктивність
Приклади задач	Дотримання стандартів, «мертвий» код	Витоки пам'яті, навантажувальні тести
Швидкість	Висока	Низька (залежить від складності тестів)
Інструменти	SonarQube, ESLint, Pylint	JProfiler, Valgrind, AppDynamics

Розбір аналізаторів коду

Аналізатори коду – це спеціалізовані інструменти, які автоматизують процес перевірки якості вихідного коду. Вони працюють за заданими правилами та стандартами, допомагаючи розробникам і командам швидко знаходити помилки, недоліки, потенційні уразливості та відхилення від стандартів. Ці інструменти інтегруються в CI/CD-пайплайни, значно спрощуючи процес забезпечення якості коду.

Основні функції аналізаторів коду:

- *перевірка синтаксису та стилю*: аналізатори забезпечують відповідність коду заданим стандартам і стилістичним рекомендаціям, полегшуючи його читання та підтримку;
- *виявлення помилок і дефектів*: вони дозволяють автоматично знаходити помилки, які можуть спричинити збої під час виконання програми;
- *аналіз продуктивності*: деякі інструменти здатні визначати, які частини коду знижують продуктивність системи, наприклад, ідентифікувати вузькі місця;
- *пошук уразливостей безпеки*: багато сучасних аналізаторів зосереджуються на забезпеченні безпеки, виявляючи потенційно небезпечні ділянки коду, такі як SQL-ін'єкції або XSS-атаки;
- *контроль технічного боргу*: інструменти допомагають відслідковувати та мінімізувати накопичення технічного боргу шляхом аналізу складності, дублювання або залежностей у коді.

Класифікація аналізаторів коду

Всі аналізатори коду можна умовно поділити на групи (класифікувати) за наведеними нижче ознаками.

За типом аналізу:

- *статичні аналізатори коду*: працюють без виконання програми (SonarQube, ESLint, Pylint);
- *динамічні аналізатори коду*: аналізують програму під час виконання (Valgrind, JProfiler).

За мовами програмування:

- *інструменти, які підтримують одну мову* (наприклад, RuboCop для Ruby);
- *мультимовні аналізатори* (наприклад, SonarQube, що підтримує Java, Python, JavaScript, C++ тощо).

За цільовим призначенням:

- *інструменти для перевірки стилю та крапих практик*: Prettier, ESLint;
- *аналізатори безпеки*: OWASP Dependency-Check, Snyk;

- *аналізатори продуктивності*: JProfiler, Perf;
- *інтегровані інструменти CI/CD*: інтегровані в Jenkins, GitLab CI.

Популярні аналізатори коду та їх можливості

Серед популярних аналізаторів коду виділяється **SonarQube**, який підтримує понад 25 мов програмування, включаючи Java, Python, C#, JavaScript. Він дозволяє виявляти помилки, дублювання коду та технічний борг, перевіряти безпеку та відповідність стандартам якості. Крім того, SonarQube інтегрується з CI/CD-інструментами, такими як Jenkins, GitLab CI та Azure DevOps, що дає змогу автоматизувати аналіз коду після кожного коміту в репозиторій. Більш детально SonarQube розглядається в Розділі 10.

ESLint є популярним інструментом для аналізу коду, який підтримує JavaScript і TypeScript. Його основними можливостями є виявлення помилок і проблем зі стилем коду, використання налаштовуваних правил перевірки, а також автоматичне виправлення деяких помилок. ESLint допомагає розробникам дотримуватися стандартів написання коду, зменшує кількість дрібних помилок і сприяє підтримці чистоти коду.

Цей інструмент широко використовується у фронтенд-проектах для забезпечення узгодженості стилю коду, покращення читабельності та спрощення співпраці в командах. Завдяки інтеграції з популярними редакторами, такими як Visual Studio Code, ESLint забезпечує миттєвий зворотний зв'язок, що дозволяє розробникам виправляти помилки ще під час написання коду. Інструмент також може бути інтегрований у CI/CD-процеси для автоматичної перевірки коду перед його злиттям у основну гілку проєкту.

Pylint – це потужний інструмент для аналізу коду, створений спеціально для Python. Він допомагає перевіряти відповідність коду стандартам PEP 8, що є основним гідом стилю для Python. Крім того, Pylint виявляє різноманітні помилки, такі як синтаксичні проблеми, зайві або непотрібні імпорти, а також дублювання коду, що сприяє підвищенню його якості та чистоти.

Цей інструмент широко застосовується для перевірки Python-коду перед його інтеграцією в основну гілку проєкту або перед розгортанням. Він допомагає розробникам зосередитися на написанні ефективного й чистого коду, мінімізуючи ризик появи дефектів.

Pylint також дозволяє налаштовувати правила аналізу відповідно до специфіки проєкту, що забезпечує гнучкість і адаптивність. Його інтеграція з популярними текстовими редакторами, такими як PyCharm, VS Code і Sublime Text, забезпечує миттєвий зворотний зв'язок під час написання коду. У контексті

CI/CD-процесів Pylint використовується для автоматичної перевірки Python-коду, що дозволяє виявляти проблеми ще до етапу розгортання.

Завдяки своїм можливостям Pylint є важливим інструментом для підтримання високих стандартів кодування та забезпечення стабільності Python-додатків.

Valgrind – це потужний інструмент для динамічного аналізу програм, розроблений для мов програмування C та C++. Він спеціалізується на виявленні проблем із використанням пам'яті, таких як витоки, переповнення буфера та доступ до неініціалізованих даних. Завдяки цьому Valgrind допомагає забезпечити стабільність і ефективність роботи програм.

Окрім аналізу пам'яті, Valgrind дозволяє тестувати продуктивність програми, надаючи детальну інформацію про споживання ресурсів, що допомагає розробникам знаходити вузькі місця та оптимізувати роботу додатків. Він також підтримує кілька модулів, таких як Memcheck для виявлення помилок із пам'яттю та Callgrind для профілювання продуктивності.

Valgrind широко використовується в проєктах, які потребують оптимізації використання пам'яті, особливо у великих системах та критично важливих застосунках, де навіть незначні помилки можуть призвести до серйозних наслідків. Інструмент ідеально підходить для аналізу серверних додатків, систем із високим навантаженням або програм, які працюють у реальному часі.

Завдяки своїй універсальності Valgrind є незамінним інструментом для розробників, які прагнуть створювати стабільні, надійні та продуктивні програми. Його інтеграція в процеси розробки дозволяє виявляти й виправляти потенційні проблеми ще на ранніх етапах, знижуючи витрати на підтримку та підвищуючи якість кінцевого продукту.

OWASP Dependency-Check – це мультимовний інструмент, створений для аналізу залежностей у програмному забезпеченні з метою виявлення відомих уразливостей. Його основне завдання — забезпечення безпеки проєктів, які використовують велику кількість зовнішніх бібліотек та компонентів. Dependency-Check аналізує метадані, такі як файли POM, файли JAR, файли package.json та інші, щоб ідентифікувати залежності, а потім перевіряє їх на відповідність базам даних відомих вразливостей, таких як NVD (Національна база даних уразливостей).

Цей інструмент особливо корисний у проєктах з великими екосистемами залежностей, наприклад, у розробці на основі Java, JavaScript, Python, .NET або інших популярних мов. Він дозволяє розробникам швидко виявляти потенційні загрози, пов'язані з використанням уразливих бібліотек, і вживати необхідних заходів для їх усунення, таких як оновлення версій або пошук альтернативних рішень.

OWASP Dependency-Check інтегрується з багатьма інструментами для автоматизації, включаючи Jenkins, GitHub Actions, GitLab CI/CD і інші, що дозволяє автоматично перевіряти безпеку залежностей під час кожного збирання або розгортання. Крім того, він надає зрозумілі звіти, які допомагають розробникам і командам безпеки оперативно реагувати на виявлені загрози.

Завдяки своїй ефективності Dependency-Check сприяє підвищенню рівня безпеки програмного забезпечення, знижує ризики, пов'язані з використанням стороннього коду, і підтримує загальну стабільність проєктів. Використання цього інструменту є невід'ємною частиною практик DevSecOps, які забезпечують інтеграцію безпеки у всі етапи розробки програмного забезпечення.

Переваги та недоліки використання аналізаторів коду

Використання аналізаторів коду має як переваги, так і недоліки, які варто враховувати в процесі розробки програмного забезпечення.

Однією з ключових переваг є автоматизація та економія часу. Аналізатори значно прискорюють процес перевірки коду, знімаючи з розробників тягар ручного пошуку помилок. Це дозволяє зосередитися на вирішенні складних завдань і створенні нових функцій.

Ще одна важлива перевага – покращення якості коду. Інструменти забезпечують дотримання стандартів написання коду, зменшують кількість дефектів, які можуть досягти кінцевого користувача, та підвищують загальну читабельність і підтримуваність проєкту.

Аналізатори також сприяють зниженню технічного боргу. Вони автоматично виявляють складні та застарілі ділянки коду, пропонуючи варіанти їх оптимізації, що дозволяє уникати накопичення проблем у майбутньому.

Безпека – ще один важливий аспект, який забезпечують аналізатори. Вони допомагають виявляти уразливості в коді та залежностях, що значно знижує ризик атак і підвищує довіру до кінцевого продукту.

Інтеграція з CI/CD-процесами робить аналізатори невід'ємною частиною сучасної розробки. Завдяки автоматичній перевірці на кожному етапі створення програмного забезпечення дефекти можна виявляти ще до того, як код потрапить у основну гілку проєкту, що зменшує витрати на виправлення помилок.

Однак, використання аналізаторів має і свої недоліки, які можуть вплинути на їх ефективність у процесі розробки. Наприклад, вони можуть створювати додаткові затримки у процесі розробки через час, потрібний для виконання аналізу. Інколи інструменти можуть видавати помилкові попередження, що відволікає розробників від важливіших завдань. Крім того,

для максимального ефекту потрібно налаштовувати правила аналізу відповідно до специфіки проєкту, що може зайняти додатковий час і ресурси.

Одним із основних недоліків є хибнопозитивні результати. Інструменти можуть генерувати помилкові попередження, що призводить до зайвого часу, витраченого на їх перевірку та виправлення. Це може знижувати ефективність роботи розробників, адже їм доводиться приділяти увагу помилкам, які насправді не є проблемами у коді, що може відволікати від реальних завдань. Проте, це можна мінімізувати шляхом налаштування правил і фільтрації менш критичних повідомлень.

Інший значний недолік — високі вимоги до налаштування. Для того щоб інструменти працювали ефективно, їх необхідно налаштувати відповідно до специфіки проєкту. Це включає в себе вибір правил перевірки, встановлення відповідних рівнів строгості та інтеграцію з іншими інструментами. Без правильного налаштування інструмент може не дати бажаних результатів або навіть створити додаткові проблеми.

Ще однією проблемою є обмежена підтримка деяких мов програмування. Багато аналізаторів мають більш обмежену підтримку для специфічних мов, що може ускладнити їх застосування в багатомовних проєктах або проєктах, де використовуються менш популярні технології. Це означає, що для ефективного аналізу коду можуть знадобитись додаткові інструменти, додаткові скрипти або ручна перевірка коду, що збільшує витрати часу та ресурсів.

Таким чином, хоча аналізатори коду можуть значно покращити процес розробки, їх недоліки, такі як хибнопозитивні результати, необхідність налаштування та обмеження в підтримці мов, вимагають уважного підходу до вибору та впровадження цих інструментів у проєкти.

Аналізатори коду є ключовими інструментами для автоматизації забезпечення якості програмного забезпечення. Їх застосування дозволяє командам покращити якість коду, оптимізувати процеси CI/CD та забезпечити стабільність і безпеку продукту. Використання таких інструментів, як SonarQube, ESLint, Pylint чи Valgrind, дозволяє зменшити витрати на розробку та підвищити задоволеність кінцевого користувача.

9.3. Інтеграція код-аналізаторів з CI/CD

Інтеграція код-аналізаторів з системами Continuous Integration / Continuous Deployment – це важливий етап у забезпеченні автоматизації перевірок якості коду та зниження ризику введення дефектів у кодову базу. Цей підхід дозволяє забезпечити постійний контроль якості та безпеки на кожному етапі розробки.

Розглянемо особливості інтеграції аналізаторів коду з CI/CD:

1) Автоматичний запуск перевірок при змінах у репозиторії

- кожного разу, коли розробник додає зміни (наприклад, коміт або пул-реквест), CI/CD-система автоматично запускає аналіз коду;
- аналізатор перевіряє нові зміни та порівнює їх з існуючими стандартами якості, забезпечуючи швидкий зворотний зв'язок.

2) Генерація звітів про якість коду

- результати перевірки інтегруються в звіти CI/CD. Ці звіти містять інформацію про:
 - виявлені помилки та їх критичність;
 - порушення стандартів стилю та безпеки;
 - технічний борг, що накопичився;
- візуалізація результатів через такі системи, як SonarQube або вбудовані інструменти CI/CD, дозволяє команді легко ідентифікувати проблеми.

3) Блокування злиття коду з помилками

- у разі виявлення критичних помилок, система блокує злиття коду в основну гілку до їх усунення. Це запобігає потраплянню дефектів у стабільну версію програми;
- правила блокування можуть бути налаштовані, наприклад:
 - блокувати, якщо знайдено помилки високого пріоритету;
 - Попереджати про менш критичні проблеми, але не зупиняти процес.

Приклади інтеграції аналізаторів коду з CI/CD-системами***Jenkins***

- аналітичні плагіни, такі як SonarQube Scanner for Jenkins, інтегруються для виконання перевірок на кожному етапі пайплайну;
- плагін генерує звіти та дозволяє переглядати помилки прямо в інтерфейсі Jenkins.

GitLab CI

- GitLab має вбудовану підтримку статичного аналізу коду через Code Quality;
- інтеграція з зовнішніми інструментами, такими як ESLint, Pylint або Snyk, дозволяє виконувати глибокий аналіз на кожному етапі пайплайну;
- виявлені помилки автоматично відображаються у звітах про злиття (merge request), що допомагає рев'юерам.

GitHub Actions

- за допомогою GitHub Actions можна створювати спеціалізовані робочі процеси для автоматизації аналізу коду;
- приклади дій (actions):
 - використання ESLint для перевірки JavaScript-коду;
 - інтеграція з Dependabot для перевірки залежностей на уразливості;
- результати аналізу автоматично додаються до Pull Request'ів.

Azure Pipelines

- інтеграція з SonarCloud або іншими інструментами дозволяє додавати кроки аналізу якості в пайплайн;
- Azure DevOps підтримує автоматичне виявлення дефектів, а також блокування релізів, якщо якість не відповідає вимогам.

Таким чином розглянуті особливості дають змогу виділити наступні **ключові переваги інтеграції з CI/CD:**

- *швидкий зворотний зв'язок:* розробники одразу отримують інформацію про помилки та недоліки, що дозволяє швидко їх виправити;
- *підвищення якості коду:* постійний контроль допомагає дотримуватися високих стандартів якості, зменшуючи ризик введення дефектів;
- *запобігання регресіям:* інтеграція аналізаторів з CI/CD дозволяє виявляти та усувати проблеми ще до того, як вони потраплять у стабільну версію програми;
- *економія часу та ресурсів:* автоматизація процесів значно зменшує витрати на ручну перевірку коду, забезпечуючи швидший випуск оновлень;
- *трансляція стандартів у команді:* завдяки автоматичним перевіркам аналізатори сприяють дотриманню єдиних стандартів якості серед усіх членів команди.

З огляду на вище зазначене, можна сформувати такі **потенційні виклики та рішення:**

- *велика кількість хибних сповіщень:*
 - *виклик:* аналізатори можуть генерувати багато хибнопозитивних попереджень, які ускладнюють роботу;
 - *рішення:* налаштування правил перевірки, щоб уникати зайвих попереджень для конкретного проєкту;
- *високі ресурси на виконання:*
 - *виклик:* виконання аналізу може сповільнювати пайплайн;

- *рішення*: використання інкрементального аналізу, який перевіряє тільки змінені ділянки коду.
- *опір з боку команди*:
 - *виклик*: розробники можуть неохоче сприймати обмеження, пов'язані з інтеграцією аналізаторів;
 - *рішення*: проведення тренінгів і пояснення переваг, таких як зниження технічного боргу та зменшення помилок.

Інтеграція аналізаторів коду з CI/CD забезпечує високий рівень автоматизації, допомагає підвищити якість та безпеку програмного забезпечення, а також сприяє економії часу. Завдяки гнучкості сучасних інструментів, таких як SonarQube, ESLint, або OWASP Dependency-Check, кожна команда може адаптувати процес під свої потреби, мінімізуючи ризики та збільшуючи ефективність розробки.

Контрольні питання до розділу 9

- 1) Що таке код-аналіз?
- 2) Які типи код-аналізу можна виділити?
- 3) Яким є основний принцип «вартості виправлення помилок»?
- 4) В чому полягають основні завдання статичного код-аналізу?
- 5) Які найпоширеніші види вразливостей безпеки можна виділити?
- 6) Якими є основні функції аналізаторів коду?
- 7) Що таке SQL-ін'єкція, і як код-аналіз допомагає її виявити?
- 8) Які переваги надає автоматизація перевірок коду?
- 9) Що таке «технічний борг» у контексті код-аналізу?
- 10) Яким чином код-аналіз інтегрується в процеси CI/CD?
- 11) Якою є основна мета динамічного аналізу коду?
 - a) Перевірка стилю коду
 - b) Оцінка продуктивності коду під час виконання
 - c) Зменшення впливу людського фактора
 - d) Виявлення синтаксичних помилок в коді
 - e) Виявлення «мертвого коду»
- 12) Що з перерахованого є основною метою статичного аналізу коду?
 - a) Аналіз продуктивності коду під час виконання
 - b) Перевірка вихідного коду без його запуску
 - c) Тестування коду з реальними даними
 - d) Аналіз мережових взаємодій
 - e) Процес моніторингу параметрів системи
- 13) Який тип аналізу коду має найвищу швидкість виконання?

- a) Статичний аналіз
 - b) Динамічний аналіз
 - c) Інтеграційний аналіз
 - d) Моніторинг
 - e) Змішаний аналіз
- 14) Що з наведеного не належить до вразливостей безпеки?
- a) Проблеми в реалізації автентифікації
 - b) Міжсайтовий скриптинг
 - c) Переповнення буфера
 - d) Форматування коду
 - e) SQL-ін'єкції
- 15) Що з наступного описує SQL-ін'єкції?
- a) Некоректне форматування коду
 - b) Помилки у використанні типів змінних
 - c) Проблеми продуктивності в базі даних
 - d) Порушення стандартів стилю
 - e) Вразливості, пов'язані з запитами до баз даних
- 16) Що є найбільш ефективним способом уникнути проблем з «мертвим кодом»?
- a) Документування всіх функцій
 - b) Автоматизований статичний аналіз коду
 - c) Щотижневе код-рев'ю
 - d) Збільшення кількості коментарів
 - e) Щоденне ручне тестування
- 17) Яка з перелічених практик є найбезпечнішою для захисту від SQL-ін'єкцій?
- a) Екранування спеціальних символів
 - b) Фільтрація вхідних даних
 - c) Перевірка довжини введеного тексту
 - d) Видалення лапок з введення
 - e) Використання параметризованих запитів
- 18) Яка характеристика є важливою для інструменту код-аналізу в контексті CI/CD?
- a) Графічний інтерфейс
 - b) Автоматизація та можливість інтеграції
 - c) Підтримка багатьох мов програмування
 - d) Детальні звіти
 - e) Безкоштовність або низька вартість ліцензії

19) Який із наведених інструментів є мультимовним аналізатором коду?

- a) Prettier
- b) SonarQube
- c) Jenkins
- d) Prometheus
- e) Apache JMeter

20) Яка перевага інтеграції код-аналізу з CI/CD?

- a) Автоматичне виправлення всіх помилок
- b) Зменшення кількості розробників у команді
- c) Повна відмова від ручного тестування
- d) Заміна інтеграційних тестів на функціональні
- e) Отримання зворотнього зв'язку про якість коду

21) Встановіть відповідність між типами код-аналізу та їхніми характеристиками:

A. Статичний аналіз	1) Виявлення вузьких місць у системі.
B. Динамічний аналіз	2) Виявлення проблем без запуску програми.
C. Аналіз продуктивності	3) Оцінка продуктивності коду під час виконання.
D. Аналіз безпеки	4) Перевірка на SQL-ін'єкції та міжсайтовий скриптинг.

22) Встановіть відповідність між перевагами аналізу коду та їх обґрунтуваннями:

A. Раннє виявлення дефектів	1) Зменшення помилок на етапі тестування.
B. Підвищення якості продукту	2) Виявлення «мертвого коду» та проблем продуктивності.
C. Прискорення розробки	3) Швидкий зворотний зв'язок для розробників.
D. Зменшення технічного боргу	4) Контроль якості коду на ранніх етапах.

23) Встановіть відповідність між виявленими проблемами коду та їх прикладами:

A. Синтаксичні помилки	1) Неоптимізовані запити до бази даних.
B. Проблеми безпеки	2) Відсутність форматування згідно PEP 8.
C. Порушення стандартів	3) Можливість SQL-ін'єкцій.
D. Проблеми продуктивності	4) Відсутність крапки з комою.

24) Встановіть відповідність між вказаними інструментами та їх призначенням:

A. SonarQube	1) Комплексний аналіз якості коду.
B. ESLint	2) Перевірка JavaScript-коду на стиль і помилки.
C. Valgrind	3) Пошук витоків пам'яті.
D. JProfiler	4) Аналіз продуктивності Java-додатків.

25) Встановіть відповідність між перевагами код-аналізу та їх конкретними проявами:

A. Економічні переваги	1) Зменшення кількості помилок за рахунок автоматизації.
B. Технічні переваги	2) Зниження витрат на виправлення помилок на пізніх етапах.
C. Організаційні переваги	3) Підвищення безпеки та надійності продукту
D. Якісні переваги	4) Забезпечення єдиного стандарту коду в команді.

РОЗДІЛ 10. SONARQUBE

10.1. Призначення інструменту та особливості застосування

SonarQube – це платформа для автоматичного аналізу якості коду, яка допомагає розробникам та командам забезпечувати високий рівень відповідності стандартам, мінімізувати технічний борг та підвищувати загальну надійність програмного забезпечення.

На рисунку 10.1 наведено логотип інструменту SonarQube.



Рис. 10.1. Логотип SonarQube

SonarQube створений як універсальний інструмент для автоматизації процесу аналізу вихідного коду з акцентом на забезпечення високої якості програмного забезпечення. Його основною метою є оптимізація розробки через постійний моніторинг коду, ідентифікацію потенційних проблем, аналіз залежностей та рекомендації щодо їх вирішення.

В якості основних цілей можна виділити наступні:

- *підвищення якості коду*: SonarQube дозволяє виявляти широкий спектр проблем у коді, серед яких:
 - *технічний борг*: недоліки коду, які знижують його читабельність, підтримуваність і ефективність (погана структура, застарілі практики та неефективний дизайн тощо);
 - *потенційні загрози безпеки*: пошук уразливостей у коді, таких як SQL-ін'єкції, XSS (міжсайтовий скриптинг) та інші;
 - *дублювання коду*: виявлення повторюваних частин коду;
 - *помилки стилю*: проводиться аналіз відповідності коду стандартам стилю та рекомендаціям найкращих практик, що сприяє узгодженості коду в команді;
- *забезпечення відповідності коду стандартам*: SonarQube перевіряє код на відповідність міжнародним стандартам програмування, таким як CWE (Common Weakness Enumeration), OWASP (Top 10) та іншим, для деяких специфічних стандартів може знадобитися додаткове

налаштування, проте це можливо завдяки гнучкості конфігурації. Це особливо корисно для проєктів, які потребують високого рівня безпеки чи сертифікації;

- *інтеграція в CI/CD процеси*: інструмент ідеально підходить для DevOps середовищ, оскільки його можна інтегрувати в пайплайни безперервної інтеграції та доставки. Під час кожного коміту чи пул-реквесту SonarQube може виконувати автоматичний аналіз, що дозволяє виявляти проблеми на ранніх етапах розробки, коли їх виправлення коштує значно менше. Розробники, своєю чергою, мають можливість отримувати зворотний зв'язок у режимі реального часу та продукувати якісний код;
- *зниження технічного боргу*: інструмент не тільки виявляє проблеми, але й надає розробникам чіткі рекомендації щодо їх усунення, що дає змогу знизити витрати на подальше обслуговування та розвиток програмного забезпечення. Наприклад:
 - заміна неефективних алгоритмів;
 - покращення читабельності коду;
 - усунення застарілих або нефункціональних частин.

Окрім основних цілей, SonarQube сприяє кращій командній співпраці, оскільки надає прозору картину стану проєкту, допомагаючи менеджерам приймати обґрунтовані рішення.

SonarQube підтримує понад 30 мов програмування, включаючи такі як Java, Python, C#, JavaScript, TypeScript, C++, Go, Kotlin тощо. Це робить його універсальним рішенням для команд, які працюють із різними технологіями.

Інструмент легко інтегрується з популярними системами безперервної інтеграції та розгортання, такими як Jenkins, GitLab CI/CD, GitHub Actions тощо. Це дозволяє автоматично проводити аналіз коду на кожному етапі розробки.

SonarQube дозволяє налаштовувати профілі якості, правила та порогові значення для різних метрик відповідно до специфічних вимог проєкту чи організації. Завдяки інтеграції з DevSecOps практиками, SonarQube забезпечує високий рівень безпеки, виявляючи вразливості в коді ще до його розгортання.

Інструмент має зручний веб-інтерфейс, який дозволяє командам швидко переглядати результати аналізу, метрики якості, тенденції змін та рекомендації щодо покращення.

Можна виділити такі ключові переваги використання SonarQube:

- *забезпечення відповідності стандартам якості коду*: завдяки чітким визначеним правилам та метрикам, SonarQube допомагає дотримуватися галузевих стандартів, таких як OWASP Top 10 для безпеки або ISO/IEC 25010 для якості програмного забезпечення;

- *рання діагностика проблем*: SonarQube дозволяє виявляти помилки, баги, вразливості, дублювання коду та проблеми з технічним боргом ще на ранніх етапах розробки, що значно знижує витрати на їх усунення;
- *підвищення продуктивності команди*: інструмент автоматизує процеси перевірки якості, звільняючи час розробників для роботи над новими функціональностями замість виправлення проблем у коді;
- *прозорість та контроль*: інструмент надає централізовану платформу для моніторингу стану коду всіх проєктів, що дозволяє менеджерам приймати обґрунтовані рішення;
- *покращення безпеки*: завдяки автоматичному аналізу SonarQube виявляє потенційно небезпечний код, що може призвести до кіберзагроз, і допомагає усунути ці вразливості.

До особливостей, що виділяють SonarQube серед інструментів з аналогічним чи подібним функціоналом, варто віднести наступні:

- Quality Gates дозволяють встановлювати пороги для ключових метрик, таких як покриття тестами, кількість багів або технічного боргу. Це допомагає командам автоматично приймати рішення щодо якості коду;
- SonarScanner – модуль аналізу, який виконує статичний аналіз коду, збирає метрики та передає результати на сервер SonarQube для подальшої обробки та візуалізації;
- SonarQube надає можливість зосередитися на перевірці змін у нових комітах або пул-реквестах, що є особливо корисним для великих проєктів;
- інструмент підтримує роботу з великими застарілими кодовими базами, дозволяючи поступово покращувати їхній стан через рефакторинг;
- розширюваність забезпечується завдяки плагінам, які дозволяють інтегрувати SonarQube з іншими інструментами чи сервісами, наприклад, SonarCloud для хмарних рішень.

SonarQube активно застосовують провідні компанії по всьому світу. ІТ-гіганти, такі як Google, Microsoft та Amazon, інтегрують його у свої процеси DevOps, щоб забезпечити стабільність та безпеку продуктів. У фінансовому секторі банки та страхові компанії використовують SonarQube для аналізу легасі-коду та дотримання нормативних вимог щодо безпеки.

У сфері мобільної розробки компанії використовують SonarQube для оптимізації продуктивності та стабільності своїх продуктів. В індустрії розваг студії, які займаються розробкою ігор, застосовують інструмент для аналізу продуктивності, безпеки та якості коду своїх проєктів.

Таким чином, SonarQube є не просто інструментом для аналізу коду, а невід’ємною частиною процесів розробки, тестування та забезпечення якості,

яка допомагає командам створювати надійне, безпечне і якісне програмне забезпечення. Його популярність зумовлена гнучкістю, потужною функціональністю та здатністю ефективно інтегруватися у сучасні DevOps-процеси.

Галузі застосування інструменту SonarQube

SonarQube є універсальним інструментом, що застосовується в багатьох галузях розробки програмного забезпечення та забезпечення його якості. Завдяки своїй функціональності, він допомагає командам розробників, інженерам DevOps, архітекторам програмного забезпечення та експертам із кібербезпеки забезпечувати надійність, безпеку та ефективність коду.

Розробка програмного забезпечення

SonarQube використовується для аналізу якості та відповідності коду на різних мовах програмування завдяки наступним аспектам:

- підтримує понад 30 мов програмування, зокрема Java, Python, C++, JavaScript, TypeScript, C#, PHP, Go, Ruby та багато інших;
- дозволяє командам розробляти великі та складні проекти, аналізуючи кожен модуль окремо;
- використовується як на етапі розробки, так і для перевірки якості коду перед релізом.

Прикладами ефективного застосування інструменту можуть слугувати:

- автоматичне тестування якості коду в Agile-процесах;
- використання у проектах, що мають високі вимоги до продуктивності та масштабованості.

DevOps

Інтеграція SonarQube в CI/CD пайплайни робить його важливим інструментом для команд DevOps завдяки наступним аспектам:

- *інтеграція з інструментами*: Jenkins, GitLab CI/CD, Azure DevOps, GitHub Actions, Bamboo, Bitbucket Pipelines тощо;
- *автоматизація процесів*: аналіз коду відбувається автоматично під час кожного коміту, пул-реквесту або створення білду;
- *запобігання проблемам*: якщо код не відповідає заданим Quality Gates, він не проходить далі у процесі розгортання.

Приклади застосування:

- у великих компаніях, які постійно оновлюють своє програмне забезпечення;
- у мікросервісних архітектурах для забезпечення стабільності кожного компонента.

Кібербезпека

SonarQube є важливим інструментом у сфері DevSecOps, оскільки допомагає виявляти та усувати вразливості в коді, включаючи перевірку безпеки залежностей та зовнішніх бібліотек, завдяки:

- аналізу безпеки для виявлення загроз, таких як SQL-ін'єкції, XSS, вразливі бібліотеки, порушення доступу до даних;
- інтеграції з OWASP Top 10 та CWE.

Приклади застосування:

- банківські системи та фінансові сервіси для забезпечення захисту даних клієнтів;
- розробка вебзастосунків із високими вимогами до безпеки.

Мобільна розробка

SonarQube забезпечує аналіз вихідного коду для мобільних платформ, таких як Android та iOS, включаючи перевірку продуктивності та сумісності з різними версіями операційних систем:

- підтримка мов Kotlin, Swift, Objective-C, Java;
- перевірка ефективності використання ресурсів мобільних пристроїв (пам'ять, процесор, батарея);
- виявлення проблем сумісності з різними версіями операційних систем.

Приклади застосування:

- розробка мобільних застосунків для e-commerce або фінансових сервісів;
- виявлення вузьких місць у продуктивності під час масштабного тестування.

Аудит існуючого коду

SonarQube використовується для аналізу великих кодових баз, що дозволяє оцінити їхній стан і знайти зони для рефакторингу. Зокрема, він надає наступні функції:

- оцінка стану коду у застарілих проєктах (legacy systems);
- виявлення дублювання, застарілих методів, помилок та вразливостей тощо;
- зниження технічного боргу та спрощення підтримки.

Приклади застосування:

- реорганізація великих монолітних систем перед переходом до мікросервісної архітектури;
- підготовка до впровадження нових функцій у проєктах зі значною історією розробки.

Освітні та наукові дослідження

SonarQube також використовується в навчальних цілях та для досліджень у галузі якості програмного забезпечення.

Приклади:

- використання на курсах програмування для навчання принципам чистого коду;
- аналіз та покращення студентських проєктів;
- проведення наукових досліджень у галузі статичного аналізу коду.

SonarQube є потужним і універсальним інструментом, який знаходить своє застосування в широкому спектрі завдань і середовищ

На рисунку 10.2 наведено інтерфейс SonarQube для проєкту, в якому виконано аналіз коду.

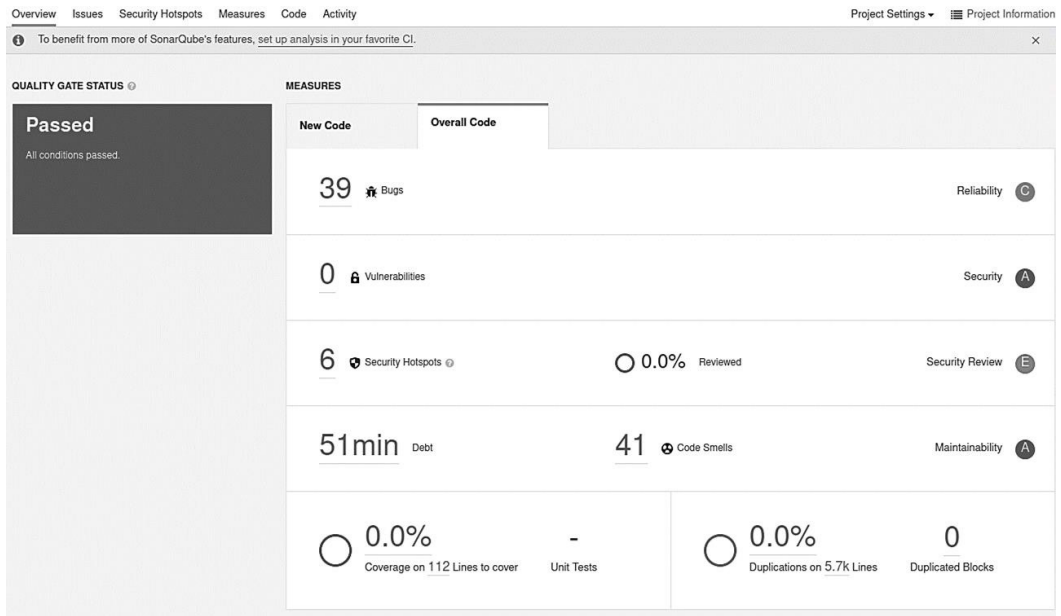


Рис. 10.2. Інтерфейс SonarQube для проєкту, в якому виконано аналіз коду

Розглянемо основні інформаційні блоки та їх значення:

1. Quality Gate Status: результат аналізу позначений зеленим блоком із статусом «Passed», що означає, що всі умови якості успішно виконані.

2. Measures (Метрики)

Метрики поділені на дві частини: New Code (новий код) і Overall Code (загальний код).

- *New Code:*

- *39 Bug*: виявлено певну кількість помилок;
- *0 Vulnerabilities*: вразливості відсутні;
- *6 Security Hotspots*: виявлено шість потенційно небезпечних ділянок коду, які потребують перевірки;
- *Overall Code*:
 - *51 min Debt*: обсяг технічного боргу складає 51 хвилину;
 - *41 Code Smells*: виявлено 41 слабких місць або потенційних проблем;
 - *0.0% Coverage*: покриття тестами відсутнє (0% з 112 рядків коду);
 - *659 Unit Tests*: є 659 модульних тестів;
 - *0.0% Duplications*: не виявлено дублікатних блоків коду;

3. Оцінки за категоріями

- *Reliability* (Надійність): оцінка «С» свідчить про середній рівень надійності коду;
- *Security* (Безпека): оцінка «А» означає високу безпеку;
- *Security Review* (Огляд безпеки): оцінка «Е» вказує на серйозні проблеми, які потребують уваги;
- *Maintainability* (Підтримуваність): оцінка «А» демонструє високу підтримуваність коду.

4. Додаткові елементи інтерфейсу

- у верхній частині екрана є вкладки для доступу до інших функцій: *Projects, Issues, Rules, Quality Profiles, Quality Gates, Administration*;
- також відображена дата та час аналізу: *April 28, 2020, 3:30 PM*.

Цей інтерфейс надає огляд стану проекту та дозволяє командам швидко ідентифікувати проблеми, фокусуючись на новому коді та ключових метриках.

10.2. Архітектура побудови SonarQube

SonarQube побудований на багаторівневій архітектурі, яка забезпечує гнучкість, масштабованість та інтеграцію з різними інструментами розробки. Кожен компонент виконує свою унікальну функцію, що дозволяє ефективно аналізувати код, обробляти дані та взаємодіяти з користувачами.

1. Database Layer (Рівень бази даних) – відповідає за збереження всіх даних, які створюються під час аналізу проекту.

- *що зберігається*:
 - метрики якості коду (кількість багів, дублікацій, рівень покриття тестами тощо);
 - логи та звіти з аналізу;

- історія змін у проектах, яка дозволяє відстежувати прогрес команди;
- *особливості:*
 - підтримує реляційні бази даних, такі як PostgreSQL, MySQL або Oracle;
 - гарантує швидкий доступ до даних навіть у великих проектах з великими обсягами коду.

В якості прикладу використання можна навести зберігання історії якості проекту для аналізу прогресу в часі або оцінки впливу рефакторингу.

2. Server Layer (Серверний рівень) – центральний компонент SonarQube, який забезпечує обробку даних, взаємодію з іншими інструментами та управління проектами.

- *основні функції:*
 - отримання даних від сканера (SonarScanner) та їх обробка;
 - виконання правил аналізу, заданих Quality Gates;
 - надання API для інтеграції з іншими системами, такими як Jenkins або GitLab;
- *особливості:*
 - використовує потужний механізм обробки даних для аналізу великих проектів;
 - масштабується для підтримки великих команд і багатьох проектів.

Цей компонент є центральною точкою для керування якістю коду у великих організаціях.

3. Scanner Layer (SonarScanner) – є важливим компонентом, який відповідає за збір даних із вихідного коду.

- *призначення:*
 - виконує статичний аналіз вихідного коду;
 - збирає інформацію про структуру коду, залежності, коментарі, метрики, баги тощо;
- *можливості:*
 - підтримує різні мови програмування та білд-системи (Maven, Gradle, Ant тощо);
 - інтегрується з CI/CD пайплайнами для автоматизації аналізу;
 - може працювати з локальним кодом або кодом із репозиторію;

Саме цей компонент забезпечує збирання коду із проекту, виконання локального аналізу та передавання результатів аналізу до серверного рівня. В якості прикладу використання можна навести процес аналізу проекту в реальному часі під час створення білду.

4. Web Interface (Веб-інтерфейс) – являє собою інтуїтивний інтерфейс для перегляду результатів аналізу, управління проектами та налаштування системи.

- *можливості:*
 - *візуалізація метрик:* веб-інтерфейс відображає звіти про стан проектів, динаміку змін, баги, уразливості та інші показники;
 - *управління проектами:* додавайте нові проекти, налаштовуйте Quality Gates, переглядайте статус проектів;
 - *історія змін:* відстежуйте, як якість коду змінювалася з часом.
- *особливості:*
 - забезпечує зручну навігацію по проектах;
 - підтримує багатокористувацький доступ із можливістю налаштування прав для різних ролей;
 - дозволяє порівнювати поточний стан проекту з попередніми версіями.

Веб інтерфейс дає змогу у зручному режимі переглядати звіти про аналіз коду для прийняття рішень щодо релізу.

5. Плагіни та розширення

Окрім основних компонентів, архітектура SonarQube підтримує систему плагінів, які розширюють його функціональність, включаючи підтримку залежностей, умов виконання та аналізу специфічних вимог безпеки.

- *приклади плагінів:*
 - плагіни для специфічних мов програмування;
 - інтеграція з ALM-системами (Application Lifecycle Management), такими як Jira або Azure DevOps;
 - розширення для аналізу специфічних вимог безпеки.
- *можливості розширення:* користувачі можуть створювати власні плагіни для вирішення унікальних завдань.

Таким чином, SonarQube забезпечує високу масштабованість, дозволяючи ефективно працювати як для невеликих команд, так і для великих організацій із сотнями проектів. Гнучкість системи досягається завдяки плагінам і API, що дозволяють адаптувати її до будь-яких потреб. Прозорість забезпечується через доступність усіх результатів аналізу у веб-інтерфейсі, що покращує взаємодію команди. Завдяки відкритим API та сумісності з багатьма інструментами DevOps, SonarQube ідеально інтегрується у сучасні процеси розробки.

10.3. SonarScanner

SonarScanner – це ключовий компонент у екосистемі SonarQube, який використовується для збору, аналізу та передачі даних із вихідного коду до сервера. Він забезпечує виконання статичного аналізу проектів незалежно від їхньої складності, обсягу та мови програмування.

Основні цілі SonarScanner:

- збір метрик: отримання показників якості коду, таких як кількість багів, вразливостей, технічного боргу, дублікацій та інші;
- аналіз файлів: виконання статичного аналізу вихідного коду відповідно до налаштувань якості (Quality Profiles);
- передача даних: надсилання зібраних результатів на сервер SonarQube для подальшої обробки та візуалізації.

На рисунку 10.3 наведено сторінку з документацією SonarQube, зокрема розділ, що стосується SonarScanner – інструмента для аналізу вихідного коду.

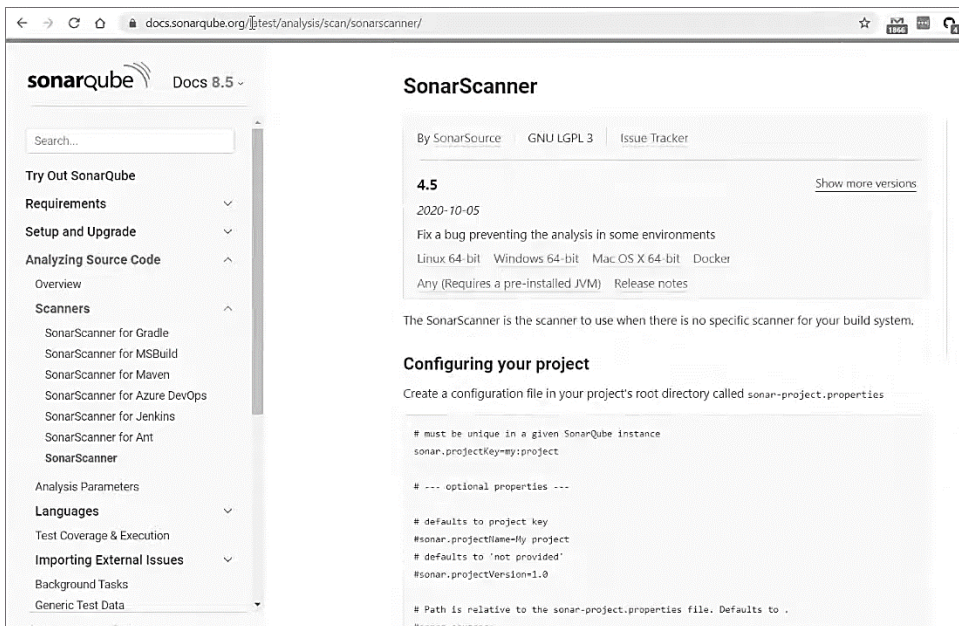


Рис. 10.3. Розділ документації, що стосується SonarScanner

SonarScanner розроблений для роботи в різних середовищах розробки та підтримує широкий спектр технологій. Він є мультиплатформним інструментом, що працює на Windows, macOS і Linux, а також може бути викликаний через командний рядок, забезпечуючи гнучкість і універсальність.

SonarScanner легко інтегрується з популярними білд-системами, такими як Maven, Gradle та Ant, і може бути налаштований для автоматичного запуску під

час створення білду. Крім того, він підтримує інтеграцію з CI/CD пайплайнами, такими як Jenkins, GitLab CI/CD, GitHub Actions, Azure DevOps, забезпечуючи автоматичний аналіз кожного коміту або пул-реквесту.

Інструмент відзначається гнучкістю у налаштуванні, дозволяючи кастомізувати профілі якості для адаптації аналізу до специфічних вимог проєкту. Він може використовуватися як для локального аналізу, так і в рамках автоматизованих процесів.

Принцип роботи SonarScanner

Процес роботи SonarScanner можна розбити на кілька етапів:

1) Збір вихідного коду: сканер збирає весь вихідний код із проєкту. Це може бути виконано локально з робочої станції розробника або з репозиторію, що зберігається у системах контролю версій, таких як Git або SVN;

2) Статичний аналіз: на основі налаштованого профілю якості (Quality Profile) виконується аналіз коду, включаючи:

- пошук багів, вразливостей, дублікацій та «поганих запахів» у коді;
- вимірювання метрик, таких як складність коду, покриття тестами, кількість рядків коду тощо;

3) Генерація звіту: після завершення аналізу SonarScanner створює звіт із результатами, який передається на сервер SonarQube.

4) Передача даних на сервер:

- сервер SonarQube отримує звіт і обробляє дані для візуалізації у веб-інтерфейсі;
- користувачі можуть переглядати результати аналізу, включаючи список проблем і рекомендації щодо їх усунення.

Варіанти використання SonarScanner

Запуск через командний рядок: прямий запуск із CLI дозволяє аналізувати код безпосередньо на робочій станції.

Приклад команди:

```
bash

sonar-scanner \
  -Dsonar.projectKey=my_project \
  -Dsonar.sources=src \
  -Dsonar.host.url=http://localhost:9000 \
  -Dsonar.login=your_token
```

Інтеграція з білд-системами: використовується у білд-скриптах Maven, Gradle або Ant.

Наприклад, для Maven це може бути:

```
mvn sonar:sonar -Dsonar.host.url=http://localhost:9000 -Dsonar.login=your_token
```

Інтеграція у CI/CD пайплайни: автоматичний аналіз кожного коміту через Jenkins, GitLab CI/CD або GitHub Actions.

Приклад конфігурації для GitLab CI/CD:

```
sonar_scanner:
  stage: analyze
  script:
    - sonar-scanner -Dsonar.projectKey=my_project -Dsonar.source
  only:
    - main
```

SonarScanner відзначається швидкістю, виконуючи аналіз безпосередньо на робочій станції або в пайплайні, що мінімізує затримки. Його простота інтеграції дозволяє легко вбудувати SonarScanner в існуючі процеси розробки. Крім того, він підтримує автоматизацію, забезпечуючи автоматичний аналіз із кожним новим комітом або зміною у проєкті.

Таким чином, SonarScanner є критично важливим інструментом у роботі з SonarQube, що забезпечує ефективний аналіз коду та покращення його якості.

10.4. Quality Gates (Ворота якості)

Quality Gates – це один із найважливіших елементів у системі SonarQube, який забезпечує об'єктивну оцінку якості коду шляхом встановлення набору критеріїв, на основі яких ухвалюється рішення про те, чи відповідає проєкт стандартам якості. Вони дозволяють розробникам та командам автоматично контролювати якість коду на різних етапах розробки.

Quality Gates спрямовані на забезпечення високої якості коду, виявляючи проблеми ще на ранніх етапах розробки, щоб мінімізувати їхній вплив на фінальний продукт. Вони дозволяють автоматизувати процес прийняття рішень,

використовуючи метрики для оцінки успішності збірки чи випуску релізу. Постійний моніторинг ключових показників допомагає зменшувати технічний борг і підтримувати його на прийнятному рівні. Крім того, Quality Gates забезпечують відповідність коду корпоративним або галузевим стандартам якості.

Розглянемо основні характеристики Quality Gates:

- *метрики, що використовуються:*

Quality Gates базуються на ключових показниках якості коду, серед яких:

- *покриття тестами:* відсоток коду, перевірений автоматизованими тестами;
 - *кількість багів:* визначення проблем у коді, що можуть викликати некоректну роботу програми;
 - *вразливості:* аналіз безпеки для виявлення можливих точок злому (наприклад, SQL-ін'єкції, XSS);
 - *технічний борг:* час, необхідний для усунення проблем у коді;
 - *дублікація коду:* виявлення повторюваних блоків коду, які можуть ускладнювати підтримку проекту;
 - *складність коду:* оцінка когнітивної або цикломатичної складності коду для забезпечення його зрозумілості.
- *автоматизація:* Quality Gates інтегруються в CI/CD пайплайни, що дозволяє автоматично приймати рішення про успішність збірки або пул-реквесту. У разі невідповідності встановленим критеріям процес збірки може бути автоматично зупинений;
 - *гнучкість:*
 - Quality Gates можуть бути налаштовані відповідно до потреб проекту або організації.
 - Дозволяють задавати різні профілі якості для різних типів проектів або команд.
 - *результати в реальному часі:* усі результати перевірки воріт якості доступні у веб-інтерфейсі SonarQube, що дозволяє командам швидко реагувати на проблеми.

Приклади правил Quality Gates

Стандартний набір правил:

- покриття тестами має бути не менше 80%;
- кількість критичних багів та вразливостей має бути 0;
- дублікація коду повинна бути нижче 5%;
- технічний борг не повинен перевищувати 10% від загального часу розробки.

Правила для нового коду (New Code):

- покриття тестами нового коду має становити не менше 90%;
- жодних критичних або блокуючих проблем;
- вразливості безпеки на новому коді повинні бути відсутніми.

Правила для релізу:

- усі блокуючі баги мають бути усунені;
- загальна кількість технічного боргу не повинна перевищувати визначений поріг (наприклад, 20 годин);
- покриття тестами має бути стабільним або зростати.

Приклад використання Quality Gates у CI/CD пайплайнах

Jenkins

Під час виконання пайплайну Jenkins запускається SonarScanner для аналізу коду, після чого перевіряються результати Quality Gates:

```
stage('Quality Gate') {
  steps {
    script {
      def qualityGate = waitForQualityGate()
      if (qualityGate.status != 'OK') {
        error "Pipeline failed due to Quality Gate failure:
${qualityGate.status}"
      }
    }
  }
}
```

GitLab CI/CD

Quality Gates можуть бути інтегровані як окремий етап у пайплайн GitLab:

```
yaml

sonar_quality_gate:
  stage: analyze
  script:
    - sonar-scanner -Dsonar.projectKey=my_project -Dsonar.sources=src
  allow_failure: false
```

GitHub Actions

Використання Quality Gates через SonarCloud:

jobs:

sonarcloud:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: SonarCloud Scan

uses: sonarsource/sonarcloud-github-action@v2

env:

SONAR_TOKEN: \${ secrets.SONAR_TOKEN }

- name: Quality Gate Check

run: |

*if ["\$(curl -s -u \$SONAR_TOKEN:
'https://sonarcloud.io/api/qualitygates/project_status?projectKey=my_project' | jq -r
'projectStatus.status')" != "OK"]; then*

exit 1

fi

Quality Gates забезпечують об'єктивну оцінку якості коду, створюючи єдиний підхід для всієї команди. Вони дозволяють діагностувати критичні проблеми ще до релізу, що знижує ризик помилок у продакшні. Завдяки автоматизації Quality Gates ефективно інтегруються у DevOps-процеси, забезпечуючи безперервний моніторинг якості. Крім того, вони підтримують безпеку, контролюючи вразливості та підвищуючи захищеність програмного забезпечення.

Отже, Quality Gates – це важливий інструмент забезпечення якості коду, що сприяє стандартизації, автоматизації та підвищенню безпеки у процесах розробки. Завдяки своїй гнучкості вони можуть бути налаштовані відповідно до потреб кожної команди або проекту.

10.5. Приклади застосування SonarQube

SonarQube є універсальним інструментом, який активно використовується у різних сценаріях розробки, тестування та забезпечення якості коду. Ось кілька розширених прикладів його застосування:

Інтеграція з Jenkins

SonarQube легко інтегрується з Jenkins, дозволяючи автоматизувати аналіз якості коду на кожному етапі CI/CD пайплайну.

Сценарій роботи може бути наступним:

- після кожного коміту в репозиторій Jenkins запускає завдання для збірки проекту;
- під час білду викликається SonarScanner для аналізу коду;
- ворота якості (Quality Gates) перевіряють результати аналізу. Якщо код не відповідає критеріям, процес збірки блокується.

Це забезпечує такі переваги:

- автоматичне виявлення критичних помилок ще до релізу;
- забезпечення відповідності коду стандартам якості.

Розглянемо приклад конфігурації:

```
groovy
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
                sh 'mvn clean install'
            }
        }
        stage('Code Analysis') {
            steps {
                withSonarQubeEnv('SonarQube') {
                    sh 'mvn sonar:sonar'
                }
            }
        }
        stage('Quality Gate') {
            steps {
                script {
                    def qualityGate = waitForQualityGate()
                    if (qualityGate.status != 'OK') {
                        error "Quality Gate failed: ${qualityGate.status}"
                    }
                }
            }
        }
    }
}
```

CI/CD пайплайни

SonarQube інтегрується з різними системами автоматизації, такими як GitHub Actions, GitLab CI/CD, Bitbucket Pipelines, Azure DevOps, забезпечуючи безперервний аналіз коду.

GitHub Actions:

- під час створення пул-реквесту SonarQube виконує аналіз змін;
- якщо новий код порушує встановлені стандарти якості, об'єднання блокується.

GitLab CI/CD:

- SonarQube автоматично аналізує код перед злиттям змін в основну гілку (main branch).

Приклад використання GitHub Actions:

```
yaml

name: SonarQube Analysis
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
jobs:
  analyze:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v3
      - name: SonarQube Scanner
        uses: sonarsource/sonarcloud-github-action@v2
        env:
          SONAR_TOKEN: ${ secrets.SONAR_TOKEN }
```

Аналіз легасі-коду

SonarQube є потужним інструментом для роботи з існуючими (легасі) проектами, які накопичили значну кількість технічного боргу.

Сценарій роботи може мати такі етапи:

- виконується аналіз великої кодової бази, щоб отримати звіт про її стан;
- визначаються найпроблемніші модулі або файли, які потребують рефакторингу;
- пріоритетність виправлень встановлюється на основі технічного боргу та критичності проблем.

Переваги:

- допомагає командам сфокусуватися на найважливіших частинах коду;
- знижує ризик проблем, пов'язаних із підтримкою та масштабуванням.

Так, наприклад, компанія може використовувати SonarQube для створення звіту, який покаже, що, наприклад, 20% файлів містять 80% проблем. Це дозволить команді зосередитися на цих 20% для максимального ефекту.

Кібербезпека та DevSecOps

SonarQube використовується в DevSecOps для статичного аналізу коду (SAST), допомагаючи виявляти потенційні вразливості ще на етапі розробки. Він не замінює динамічний аналіз (DAST) або тестування на проникнення, але суттєво покращує безпеку коду.

Сценарій роботи:

- виконується аналіз коду для виявлення потенційно небезпечних елементів, таких як SQL-ін'єкції, XSS (міжсайтовий скриптинг), неправильна обробка автентифікації тощо;
- вразливості класифікуються за рівнем критичності, і розробники отримують рекомендації щодо їх усунення.

Переваги:

- знижує ризик атак на програмне забезпечення;
- додає безпеку в процеси розробки без значного збільшення часу.

В якості прикладу можна навести використання SonarQube у банківському програмному забезпеченні для пошуку SQL-ін'єкцій у модулях, пов'язаних із транзакціями.

Підтримка мобільної розробки

SonarQube також застосовується для аналізу якості коду мобільних додатків для платформ Android та iOS та кросплатформних рішень, таких як React Native, Flutter та Ionic.

Сценарій роботи:

- аналіз Java/Kotlin коду для Android-додатків;
- аналіз Swift/Objective-C для iOS-додатків;
- виявлення проблем, які можуть вплинути на продуктивність або стабільність мобільного додатку, зокрема витоків пам'яті, помилок потоків (concurrency issues) та некоректного використання ресурсів.

Переваги:

- дозволяє виявити проблеми ще до того, як додаток потрапить у руки користувачів;
- підвищує якість додатків, що особливо важливо для конкурентного мобільного ринку.

SonarQube є незамінним інструментом у процесах забезпечення якості коду, кібербезпеки та автоматизації розробки. Його широкий спектр застосування робить його важливим елементом як для нових проєктів, так і для роботи з легасі-кодом.

Контрольні питання до розділу 10

- 1) Яким є призначення інструменту SonarQube?
- 2) Які основні рівні має архітектура SonarQube?
- 3) Які основні оцінки присвоює проєктам SonarQube?
- 4) Для чого використовується Quality Gates у SonarQube?
- 5) Які основні метрики використовуються в Quality Gates?
- 6) Яке призначення має SonarScanner?
- 7) Що таке Code Smells у SonarQube?
- 8) Яким чином SonarQube інтегрується з процесами CI/CD?
- 9) Як відбувається інтеграція SonarQube з Jenkins у процесі CI/CD?
- 10) Якою є роль SonarQube в процесах DevSecOps?
- 11) Що означає рейтинг «А» у секції Security в SonarQube?
 - a) Максимальне покриття коду автоматичними тестами
 - b) Виявлення синтаксичних помилок в коді
 - c) Зменшення технічного багу порівняно з минулою перевіркою
 - d) Відсутність вразливостей в коді
 - e) Виявлення «мертвого коду»
- 12) Який компонент SonarQube відповідає за збір даних з вихідного коду?
 - a) Database Layer
 - b) Server Layer
 - c) Web Interface
 - d) SonarScanner
 - e) Quality Gate
- 13) Що з перерахованого відображає основну функцію SonarQube?
 - a) Розгортання ІТ проєктів
 - b) Аналіз якості коду проєкту
 - c) Управління версіями проєкту
 - d) Моніторинг серверів

- е) Тестування продуктивності ІС
- 14) Який мінімальний відсоток покриття тестами рекомендується для нового коду згідно зі стандартними правилами Quality Gates?
 - а) 10%
 - б) 30%
 - в) 90%
 - г) 100%
 - д) 50%
- 15) Які платформи мобільної розробки підтримує SonarQube?
 - а) Лише iOS
 - б) Android та iOS
 - в) Лише Android
 - г) Android, iOS та Windows Phone
 - д) Лише Windows Phone
- 16) Що відбувається при невідповідності коду критеріям Quality Gates в CI/CD пайплайні?
 - а) Код автоматично виправляється
 - б) Процес збірки блокується
 - в) Надсилається повідомлення розробнику, але процес не зупиняється
 - г) Створюється новий репозиторій
 - д) Код автоматично відправляється на додаткове тестування
- 17) Який компонент SonarQube відповідає за збереження метрик, історії змін та звітів?
 - а) Database Layer
 - б) Server Layer
 - в) Web Interface
 - г) SonarScanner
 - д) Quality Gate
- 18) Яке з наступних тверджень відповідає 100% покриттю коду тестами?
 - а) Встановлено відсутність в коді технічного боргу
 - б) Усі ділянки коду перевірені автоматизованими тестами
 - в) Код відповідає стандартам якості
 - г) Виявлено SQL-ін'єкції та міжсайтовий скриптинг
 - д) Аналіз показав відсутність багів в коді
- 19) Яке обмеження на дублювання коду встановлюється у стандартному наборі правил Quality Gates?
 - а) Нижче 3%
 - б) Нижче 5%
 - в) Нижче 7%

- d) Нижче 10%
- e) Нижче 15%
- 20) Який компонент виконує безпосередній аналіз коду при інтеграції з CI/CD системами?
- a) Quality Gates
- b) SonarCloud
- c) SonarScanner
- d) SonarAnalyzer
- e) Web Interface
- 21) Встановіть відповідність між елементами Quality Gates та відповідними критеріями:

A. Покриття тестами	1) Має бути нижче 5%.
B. Кількість критичних помилок	2) Час, необхідний для виправлення проблем, не перевищує порог.
C. Дублікація коду	3) Не менше 80% перевіреного коду.
D. Технічний борг	4) Рівна нулю.

- 22) Встановіть відповідність між галузями застосування SonarQube та їх прикладами:

A. Кібербезпека	1) Пошук SQL-ін'єкцій.
B. Мобільна розробка	2) Рефакторинг старих проєктів.
C. DevOps	3) Інтеграція з Jenkins.
D. Аудит легасі-коду	4) Аналіз Swift/Objective-C.

- 23) Встановіть відповідність між компонентами SonarQube з їхніми функціями:

A. Database Layer	1) Обробка даних та управління проєктами.
B. Server Layer	2) Збереження метрик та історії змін.
C. SonarScanner	3) Візуалізація результатів аналізу.
D. Web Interface	4) Виконання статичного аналізу коду.

24) Встановіть відповідність між метриками та їх описами:

A. Покриття тестами	1) Час необхідний для виправлення проблем.
B. Технічний борг	2) Відсоток протестованого коду.
C. Дублікація коду	3) Повторювані фрагменти коду.
D. Вразливості	4) Потенційні загрози безпеці.

25) Встановіть відповідність між етапами роботи SonarScanner та їхніми завданнями:

A. Збір вихідного коду	1) Виявлення багів і вразливостей.
B. Статичний аналіз	2) Надсилання результатів на сервер.
C. Генерація звіту	3) Отримання файлів проєкту.
D. Передача даних	4) Формування результатів аналізу.

РОЗДІЛ 11. ОГЛЯД МІНІМАЛЬНО УКОМПЛЕКТОВАНОЇ СІ-СИСТЕМИ

11.1. Вимоги до СІ-системи

СІ-система має на меті забезпечення автоматизацію ключових процесів розробки та тестування програмного забезпечення і має на меті швидке виявлення помилок і забезпечення якості коду після регулярного інтегрування змін.

Така система може складатись з наступних базових компонентів:

- *системи контролю версій*: дозволяють відстежувати зміни в кодовій базі та керувати різними версіями програмного забезпечення. Розподілені VCS (наприклад, Git) також забезпечують ефективну співпрацю в команді;
- *сховища коду (GitHub, GitLab або Bitbucket)*: надають веб-інтерфейс для роботи з репозиторіями та додаткові інструменти для спільної розробки. Ці платформи забезпечують функціонал для code review, керування проектами, відстеження помилок та інтеграції з іншими інструментами розробки;
- *СІ-інструмент*: дозволяє автоматизувати робочі процеси, включаючи збірку проекту, запуск тестів та розгортання;
- *тестове середовище*: використовуються рішення, які можуть бути реальними (фізичні сервери або локальні машини) чи віртуалізованими (контейнери Docker);
- *інструмент для аналізу коду*: перевіряє код на відповідність стандартам, виявляючи потенційні вразливості, code smells та дублювання коду;
- *система оповіщень (Slack, електронна пошта або звіти в СІ-платформі)* : інформує команду про статус збірки, тести та результати аналізу коду.

СІ-сервери, такі як Jenkins, CircleCI чи GitLab CI/CD виконують автоматизовані завдання, такі як збірка додатків, запуск тестів і перевірка якості коду. GitHub Actions, як вбудований інструмент CI/CD в GitHub, дозволяє автоматизувати робочі процеси безпосередньо в репозиторії, включаючи збірку, тестування та розгортання додатків.

Інструменти для аналізу коду (наприклад SonarQube) інтегруються в СІ-процеси для виявлення потенційних проблем у коді, таких як code smells, вразливості та дублювання коду, які можуть призвести до технічного боргу, вразливостей і відповідність стандартам написання. Вони також відстежують покриття коду тестами та допомагають контролювати технічний борг.

Для централізованого зберігання та управління збірками проекту, бібліотеками і залежностями можна використати артефакт-репозиторії, такі як Nexus або JFrog Artifactory

Автоматичні нотифікації є важливою складовою CI/CD-процесів, оскільки вони дозволяють тримати команду в курсі подій. Автоматичні нотифікації забезпечують оперативність, дозволяючи членам команди миттєво отримувати інформацію про стан пайплайнів і швидко реагувати на помилки. Вони сприяють прозорості, тримаючи всю команду в курсі прогресу CI/CD-процесу, включаючи інформацію про успішні та провальні білди.

Інтеграція зі звичними інструментами дозволяє надсилати сповіщення через Email, Slack, Telegram, GitHub або інші системи, які команда використовує у своїй щоденній роботі. Нотифікації можна налаштувати для відображення лише певних подій, наприклад, лише для невдалих білдів або результатів проходження Quality Gate у SonarQube.

Сповіщення можуть містити детальні звіти, такі як лінки на результати білду, логи, статус Quality Gate чи метрики тестів, що дозволяє отримувати всю необхідну інформацію в одному місці.

Для зручності сприйняття нотифікації в Slack або Telegram можуть використовувати кольорове кодування, щоб позначати статус завдань: зелений для успіху та червоний для помилок.

Таким чином, CI-система є критичним інструментом сучасної розробки програмного забезпечення, що забезпечує автоматизацію, контроль якості та ефективну співпрацю команди. Вона дозволяє швидко виявляти та виправляти помилки, підтримувати високу якість коду та прискорювати процес розробки.

Правильно налаштована CI-система значно знижує ризики при розробці, покращує якість продукту та підвищує ефективність команди через автоматизацію рутинних завдань та забезпечення швидкого зворотного зв'язку.

11.2. Використання Jenkins для побудови CI-системи

Розглянемо використання інструментів GitHub, Jenkins та SonarQube для побудови мінімально укомплектованої CI-системи.

Jenkins – це потужний інструмент для автоматизації процесів розробки, який часто використовується в тісній інтеграції з GitHub. Для підключення GitHub репозиторію до Jenkins існують кілька способів, кожен із яких має свої переваги та підходить для певних сценаріїв.

Для з'єднання з репозиторієм можна використати наступні способи:

- плагін Git Plugin
- Webhooks

- Polling SCM

Git Plugin

Jenkins має вбудовану підтримку Git через плагін Git Plugin, який дозволяє з'єднувати репозиторій із сервером Jenkins.

Для встановлення з'єднання можна виконати наступні кроки:

- 1) Встановити Git Plugin через «Manage Jenkins» → «Manage Plugins».
- 2) Додати репозиторій до Jenkins job: для цього у полі «Source Code Management» слід обрати Git та вказати URL репозиторію (HTTPS або SSH);
- 3) Налаштувати аутентифікацію:
 - **HTTPS:** використовуйте особистий токен доступу (Personal Access Token) для GitHub, який можна згенерувати у налаштуваннях GitHub;
 - **SSH:** використовуйте SSH-ключ, який додається до Jenkins у розділі «Credentials».

Приклад використання SSH:

```
git@github.com:username/repository.git
```

Webhooks

Webhooks – це механізм зворотного виклику, який дозволяє миттєво реагувати на події, що відбуваються в репозиторії (наприклад, push, створення pull request, merge). Ця технологія базується на передачі HTTP-запитів від GitHub (чи іншого сервісу) до сервера Jenkins, завдяки чому інтеграція стає більш ефективною та динамічною. В даному випадку Webhooks дає змогу автоматично запускати Jenkins job при змінах у репозиторії.

Для налаштування потрібно проробити наступні кроки:

- 1) У налаштуваннях GitHub репозиторію потрібно перейти у розділ «Settings» → «Webhooks» та додати новий webhook:
 - URL: `http://<Jenkins_URL>/github-webhook/`
 - Content type: `application/json`
- 2) Слід переконатись, що Jenkins має плагін "GitHub Integration Plugin" та активувати опцію «GitHub hook trigger for GITScm polling» у налаштуваннях Jenkins job.
- 3) Далі слід налаштувати webhook, вказавши події, на які буде реагувати webhook (наприклад, push, pull request, tag). Коли зазначена подія відбувається, GitHub надсилає HTTP POST-запит на вказаний URL. Jenkins приймає запит і запускає відповідний pipeline або job.

Нижче наведено приклад у Jenkins Pipeline:

```
groovy

pipeline {
    agent any
    triggers {
        githubPush() // Використання тригера GitHub Push через webhook
    }
    stages {
        stage('Build') {
            steps {
                echo 'Building project...'
            }
        }
    }
}
```

Перевагою цього способу є висока швидкість виконання, адже завдання запускаються майже миттєво після події у репозиторії. Це також допомагає знизити навантаження на сервер Jenkins, оскільки відсутня необхідність у постійних перевірках репозиторію.

Серед недоліків можна відзначити вимогу забезпечення публічного доступу до Jenkins, що потребує налаштування NAT, проксі-сервера або використання VPN для безпечного з'єднання.

Polling SCM

Polling SCM – це метод, за якого Jenkins періодично порівнює поточну версію коду в репозиторії з останньою зібраною версією, щоб виявити нові зміни. Це підходить для випадків, коли налаштування Webhooks неможливе (наприклад, через обмеження мережі).

Для налаштування можна скористатись наступним алгоритмом:

- 1) У налаштуваннях Jenkins job увімкнути опцію "Poll SCM".
- 2) У полі розкладу (Schedule) використати cron-подібний синтаксис,

наприклад:

```
H/5 * * * *
```

Цей запис означає перевірку репозиторію кожні 5 хвилин.

Перевагою цього підходу є його простота налаштування, особливо для середовищ із закритими мережами, де публічний доступ обмежений.

До недоліків належать додаткові витрати ресурсів на періодичні перевірки репозиторію, навіть якщо змін немає, а також можливі затримки між моментом внесення змін (push) і запуском збірок.

Таблиця 11.1 містить порівняння основних способів інтеграції між GitHub та Jenkins, враховуючи такі фактори, як швидкість реакції, складність налаштування, навантаження на ресурси Jenkins та рекомендоване використання в залежності від умов проєкту.

Таблиця 11.1. Порівняння способів інтеграції між GitHub та Jenkins

Спосіб	Швидкість реакції	Налаштування	Ресурси Jenkins	Рекомендоване використання
Git Plugin	Середня	Легке	Помірні	Базова інтеграція
Webhooks	Висока	Складніше	Мінімальні	Часті зміни у коді
Polling SCM	Низька	Легке	Високі	Закриті мережі

Для більшості сучасних сценаріїв інтеграції Webhooks є найкращим вибором, оскільки вони забезпечують миттєву реакцію на зміни в репозиторії без перевантаження сервера. Якщо є специфічні вимоги, такі, наприклад, як обмежений доступ до інтернету, можна використовувати Polling SCM.

Використання cron-стилю в Jenkins Jobs

Jenkins підтримує різні підходи для автоматизації запуску завдань залежно від потреб проєкту. Окрім Webhooks, налаштувати реакцію системи на зміни в кодовій базі або планувати регулярне виконання завдань можна використовуючи cron-стиль

Cron-стиль використовується в Jenkins для завдань, які повинні виконуватися на регулярній основі, наприклад, для планових бекапів або періодичних перевірок безпеки. Планувальник Jenkins базується на синтаксисі cron – текстових виразах, які визначають періодичність виконання.

Синтаксис cron у Jenkins:

```
sql

MINUTE HOUR DOM MONTH DOW
```

- MINUTE: хвилина (0-59);
- HOUR: година (0-23);

- DOM (Day of Month): день місяця (1-31);
- MONTH: Місяць (1-12);
- DOW (Day of Week): день тижня (0-7), де 0 і 7 – це неділя;

Приклади cron-розкладу:

- H/5 * * * *: завдання виконується кожні 5 хвилин;
- 0 12 * * * *: завдання виконується щодня о 12:00;
- 30 2 * * 1-5: завдання виконується о 2:30 ночі кожен будній день (з понеділка по п'ятницю).

Особливості cron у Jenkins включають використання H (hashed), що дозволяє рівномірно розподіляти запуск завдань у зазначений інтервал. Наприклад, ви можете налаштувати запуск кожні 10 хвилин за допомогою виразу H/10 * * * *, що забезпечить унікальний початок для кожного job. Також cron-стиль дає змогу здійснювати перевірки за розкладом, що особливо корисно у середовищах із закритими мережами за умови належного налаштування мережевої інфраструктури, де Jenkins може перевіряти репозиторій за розкладом. Крім того, цей метод має високу гнучкість у налаштуванні періодичності.

До недоліків cron-стилю можна віднести затримки між подією та запуском завдання, які можуть тривати до кількох хвилин. Також наявність періодичних перевірок, навіть коли в репозиторії немає змін, може призводити до зайвих витрат ресурсів і збільшення навантаження на сервер.

Приклад використання у Jenkins Pipeline:

```

groovy

pipeline {
    agent any
    triggers {
        pollSCM('H/5 * * * *') // Перевірка змін у репозиторії кожні 5 хвилин
    }
    stages {
        stage('Build') {
            steps {
                echo 'Running periodic build...'
            }
        }
    }
}

```

В таблиці 11.2 наведено порівняння webhooks та cron-стилю за основними критеріями, такими як швидкість реакції, навантаження на ресурси сервера, гнучкість налаштування та оптимальні варіанти використання кожного методу в залежності від специфіки проєкту.

Таблиця 11.2. Порівняння webhooks та cron-стилю:

Метод	Швидкість реакції	Ресурси сервера Jenkins	Гнучкість	Рекомендоване використання
Webhooks	Миттєва	Мінімальні	Висока	Для проєктів із частими змінами
Cron-стиль	Затримка (хвилини)	Помірні	Висока	Для періодичних завдань або обмежень мережі

Обидва методи можна комбінувати для створення гнучкої СІ-системи. Наприклад, використовувати webhooks для швидкого запуску pipeline після push, а cron-стиль для періодичного виконання завдань, як-от оновлення залежностей чи перевірки безпеки.

Доступність Jenkins

Безперервна інтеграція залежить від належно налаштованого та стабільного мережевого середовища, оскільки різні компоненти СІ-системи взаємодіють через мережу. У цьому контексті важливо врахувати такі аспекти, як доступність серверів, забезпечення безпеки з'єднань, управління мережевими фаєрволами та налаштування взаємодії між різними службами.

Для коректної роботи Jenkins, особливо в інтеграції з GitHub або іншими зовнішніми сервісами, сервер Jenkins має бути доступним для зовнішніх запитів. Це особливо важливо для використання webhooks, які надсилають запити на Jenkins для автоматичного запуску процесів СІ.

Основні вимоги до доступності:

- публічна IP-адреса або DNS: сервер Jenkins повинен мати публічну IP-адресу або доменне ім'я (наприклад, jenkins.example.com), щоб приймати запити від зовнішніх сервісів, таких як GitHub. У випадках, коли Jenkins працює в локальній мережі, можна використовувати NAT (Network Address Translation) або проксі-сервер для забезпечення доступу;

- *використання тунелювання*: якщо Jenkins розміщений у локальній мережі без прямого доступу, можна використовувати інструменти тунелювання, такі як ngrok, щоб забезпечити тимчасовий доступ через публічний URL;
- *обмеження доступу*: щоб уникнути небажаного доступу, слід обмежити коло IP-адрес, які можуть звертатися до Jenkins. Наприклад, у фаєрволі можна дозволити лише запити з IP-адрес сервісів GitHub.

Приклад:

Якщо Jenkins розміщений за NAT, потрібно:

- прокинути порт (наприклад, 8080) із зовнішньої мережі до внутрішнього сервера Jenkins;
- використовувати DNS-ім'я для зручного доступу (jenkins.company.com → публічний IP).

Безпека

Безпека є критично важливим аспектом при інтеграції Jenkins у мережеве середовище, особливо якщо сервер Jenkins доступний із зовнішньої мережі.

Рекомендації щодо дотримання безпеки є наступними:

- *HTTPS-з'єднання*: усі з'єднання з Jenkins мають бути зашифрованими. Для цього використовується SSL-сертифікат. Можна отримати безкоштовний SSL-сертифікат від Let's Encrypt або придбати комерційний сертифікат;
- *аутентифікація та авторизація*: потрібно налаштувати багаторівневу аутентифікацію для доступу до Jenkins. Наприклад, використати плагіни Jenkins для інтеграції з LDAP або OAuth (GitHub, Google). Також варто обмежити доступ до ресурсів Jenkins за ролями. Наприклад, адміністратори повинні мати доступ до налаштувань, а розробники – лише до своїх проєктів.
- *захищені токени для Webhooks*: у налаштуваннях GitHub webhook потрібно додати секретний токен (secret), який Jenkins перевірятиме під час кожного запиту, та увімкнути перевірку цього токена в Jenkins.
- *оновлення та патчі*: варто регулярно оновлювати Jenkins та плагіни для усунення вразливостей та використовувати LTS-версії Jenkins.

Приклад налаштування HTTPS для Jenkins:

- встановити сертифікат на сервер Jenkins (через Nginx або Apache як проксі-сервер);
- увімкнути переадресацію з HTTP на HTTPS.

```
nginx
server {
    listen 80;
    server_name jenkins.example.com;
    return 301 https://$host$request_uri;
}

server {
    listen 443 ssl;
    server_name jenkins.example.com;

    ssl_certificate /path/to/cert.pem;
    ssl_certificate_key /path/to/key.pem;

    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

Проблеми з мережевими фаєрволами

У мережевому середовищі часто виникають проблеми через обмеження фаєрволів або інших систем безпеки, які блокують з'єднання між компонентами CI/CD.

Нижче наведені основні виклики, з якими може стикнутися користувач:

- *закриті порти*: Jenkins за замовчуванням використовує порт 8080, але цей порт може бути закритим фаєрволом. Для роботи webhooks із GitHub потрібно відкрити порт 80 або 443 (для HTTPS);
- *з'єднання між Jenkins і сторонніми сервісами*: Jenkins може взаємодіяти з іншими сервісами, такими як SonarQube, Docker Registry, чи Kubernetes. Для цього потрібно переконатися, що всі необхідні порти відкриті;
- *налаштування NAT*: якщо Jenkins працює у внутрішній мережі, а SonarQube або інші сервіси розташовані зовні, можна налаштувати NAT, VPN, тунелювання (наприклад, через ngrok) або використати проксі-сервер для забезпечення доступу;

- *доступ до інтернету*: Jenkins потребує доступу до мережі Інтернет для завантаження залежностей, оновлення плагінів і взаємодії з хмарними сервісами. У деяких компаніях доступ обмежується через проксі – у таких випадках можна налаштувати проксі у Jenkins або використовувати локальні дзеркала для завантаження залежностей.

Рекомендується використовувати інструменти моніторингу, такі як Wireshark чи tcpdump, для діагностики проблем мережевого з'єднання. У налаштуваннях фаєрвола варто дозволити лише необхідні порти, зокрема HTTP/HTTPS (80/443), порт для Jenkins (8080, якщо він використовується) та порт для SonarQube (9000).

Розглянемо приклад типового CI-середовища:

- *Jenkins*: доступний через HTTPS (443). Взаємодіє з GitHub через webhooks;
- *SonarQube*: працює на порті 9000. Отримує результати сканування від Jenkins через Sonar Scanner.
- *GitHub*: надсилає запити на сервер Jenkins через webhooks;
- *мережевий фаєрвол*: відкриті лише потрібні порти, а всі інші блокуються для забезпечення безпеки.

Коректна робота CI/CD системи залежить від налаштування мережевої доступності та безпеки. Забезпечення доступності Jenkins, шифрування з'єднань через HTTPS, а також оптимальна конфігурація фаєрволів і NAT допоможуть уникнути більшості потенційних проблем і підвищити надійність інтеграції.

Інтеграція SonarQube в інфраструктуру за допомогою Sonar-Scanner

SonarQube – це платформа для аналізу якості та безпеки коду, яка дозволяє знаходити баги, уразливості, кодову заборгованість і забезпечувати відповідність стандартам програмування. Його інтеграція в CI/CD-процес через Sonar-Scanner допомагає підтримувати високу якість коду, автоматизуючи перевірки.

Інтеграція SonarQube у CI/CD-процес складається з кількох основних кроків:

1) Інсталяція SonarQube:

- *локально*: слід встановити SonarQube на виділеному сервері або віртуальній машині (контейнерна інсталяція потребує зміну подальших конфігурацій, тому в даному сценарії – не розглядається). За замовчуванням SonarQube працює на порті 9000;
- *у хмарі*: можна використати хмарні сервіси, наприклад, SonarCloud – SaaS-версію SonarQube.

- забезпечити доступність SonarQube для Jenkins та розробників через URL (наприклад, <http://sonarqube.example.com>).
- 2) Налаштування Jenkins для SonarQube: потрібно встановити плагін SonarQube Scanner for Jenkins через Jenkins Plugin Manager та додати сервер SonarQube у налаштуваннях Jenkins наступним чином:
 - увійти до Jenkins → Manage Jenkins → Configure System;
 - у секції SonarQube servers додати новий сервер, вказавши URL SonarQube та токен для аутентифікації.
- 3) Налаштування інструменту командного рядка для запуску аналізу коду Sonar-Scanner: для Jenkins рекомендується додати Sonar-Scanner як глобальний інструмент наступним чином:
 - увійти до Jenkins → Manage Jenkins → Global Tool Configuration;
 - у секції SonarQube Scanner додати новий інструмент, вказавши шлях до встановленого Sonar-Scanner.
- 4) Інтеграція з Jenkins Pipeline: потрібно розробити pipeline для автоматизації аналізу коду в процесі CI/CD.

Налаштування SonarQube та Sonar-Scanner

Для інсталяції SonarQube на сервер потрібно завантажити останню версію SonarQube з офіційного сайту, розпакувати архів та запустити сервер:

```
bash

./bin/linux-x86-64/sonar.sh start
```

Після чого слід відкрити SonarQube у браузері (<http://localhost:9000>) і створити токен доступу для Jenkins.

Для встановлення Sonar-Scanner потрібно завантажити Sonar-Scanner:
`wget https://binaries.sonarsource.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-<version>.zip`

Розпакувати архів і додати шлях до sonar-scanner у змінну середовища PATH.

Приклад конфігурації sonar-project.properties для аналізу проєкту:

```
properties

sonar.projectKey=my_project
sonar.projectName=My Project
sonar.projectVersion=1.0
sonar.sources=src
sonar.language=java
sonar.sourceEncoding=UTF-8
sonar.host.url=http://sonarqube.example.com
sonar.login=<TOKEN>
```

Інтеграція SonarQube із Jenkins через Sonar-Scanner є ключовим елементом у забезпеченні якості коду. Використання Quality Gates та інтеграція з системами контролю версій дозволяють створити прозорий процес перевірки якості, знижуючи ризики впровадження небезпечного чи неякісного коду.

Автоматичні нотифікації

Автоматичні нотифікації є важливою частиною CI/CD-процесу, оскільки вони забезпечують швидке інформування команди про результати виконання завдань, збої чи інші події. Jenkins пропонує широкий спектр можливостей для інтеграції нотифікацій у різні канали.

Email-нотифікації

Jenkins підтримує надсилання звітів і сповіщень через електронну пошту.

Для цього потрібно налаштувати SMTP-сервер наступним чином:

- увійти до Jenkins → Manage Jenkins → Configure System;
- у секції Email Notification вказати SMTP-сервер (наприклад, smtp.gmail.com), порт (587), ім'я користувача та пароль.

Slack/Discord-нотифікації

Для інтеграції Slack або Discord використовується плагін Slack Notification Plugin.

Налаштування Slack є наступними:

- у Slack потрібно створити новий Incoming Webhook для робочого простору та отримати URL для інтеграції;
- у Jenkins → Manage Jenkins → Configure System додати налаштування Slack.

GitHub-нотифікації

Jenkins може автоматично відправляти статус виконання завдань (наприклад, Success, Failure) до pull request або commit в GitHub.

Для цього використовується плагін GitHub Integration Plugin. Нижче наведено приклад конфігурації GitHub-нотифікації:

```
pipeline {
  agent any
  stages {
    stage('Build') {
      steps {
        echo 'Building project...'
      }
    }
  }
  post {
    success {
      githubNotify context: 'Build Status',
                  status: 'SUCCESS',
                  description: 'Build completed successfully.'
    }
    failure {
      githubNotify context: 'Build Status',
                  status: 'FAILURE',
                  description: 'Build failed.'
    }
  }
}
```

Нотифікації через Telegram

Jenkins також може інтегруватися з Telegram для надсилання сповіщень у групи або особисті чати.

Так, наприклад, для використання плагіну Telegram Notification Plugin потрібно створити бота в Telegram через @BotFather; отримати токен та у Jenkins налаштувати токен бота і ідентифікатор чату.

Завдяки використанню різні канали, такі як Email, Slack, Telegram, або GitHub, Jenkins забезпечує гнучкість і адаптивність нотифікацій відповідно до потреб проекту. Інтеграція цих сповіщень у пайплайни сприяє підвищенню продуктивності команди та полегшує моніторинг процесів.

Послідовності роботи елементів у створеній CI

Безперервна інтеграція забезпечує автоматизацію ключових етапів розробки, включаючи тестування, перевірку коду та підготовку артефактів. Нижче наведено детальний опис етапів роботи елементів у розглянутій CI-структурі:

Етапи роботи CI-процесу:

- 1) розробник робить push у GitHub репозиторій:
 - код із новими змінами надсилається до віддаленого репозиторію (branch або mainline);
 - це є тригером для початку CI-процесу.
- 2) Webhook у GitHub відправляє запит на Jenkins:
 - після події push, pull request чи merge, Webhook у GitHub надсилає HTTP-запит до Jenkins за заздалегідь налаштованим URL (наприклад, `http://<Jenkins_URL>/github-webhook/`);
 - Webhook передає інформацію про зміни (коміти, автор, гілка тощо), що дозволяє Jenkins ініціювати відповідні завдання.
- 3) Jenkins запускає збірку (build) проекту:
 - Jenkins ініціює визначений у Pipeline процес збірки. Процес може включати:
 - завантаження вихідного коду з репозиторію;
 - компіляцію проекту (для мов, таких як Java, C++ тощо);
 - виконання статичних і динамічних перевірок.
 - конфігурація Pipeline може бути як Declarative, так і Scripted.
- 4) під час збірки виконується:
 - статичний аналіз коду (SonarQube):
 - SonarQube сканує код на наявність помилок, вразливостей, повторень і недоліків у стилі;
 - Sonar-Scanner інтегрується через Jenkins Pipeline для автоматичного запуску після збірки;
 - успішне проходження Quality Gate є критичним для подальших етапів;
 - автоматичне тестування:
 - виконуються модульні, інтеграційні чи системні тести;
 - інструменти для тестування можуть включати JUnit, TestNG, Selenium, або PyTest залежно від технологій проекту;
 - звіти про результати тестів автоматично генеруються та додаються до Jenkins Dashboard.
- 5) Jenkins повідомляє про статус збірки через нотифікації:
 - після завершення збірки Jenkins надсилає сповіщення:

- у Slack або Telegram для команди розробників, тестувальників або менеджерів залежно від потреб проєкту;
- у GitHub у вигляді статусу завдання (success/failure) у pull request;
- через Email для менеджерів або CI-адміністраторів.
- у разі помилки розробники отримують детальну інформацію про проблему (лог збірки, звіт тестів тощо).
- 6) У разі успіху створюється артефакт для подальшого використання (наприклад, у CD):
 - артефактом може бути:
 - зібраний .jar, .war, .zip, або інший виконуваний файл;
 - докер-образ для контейнеризації програми;
 - статичний вебсайт або інші ресурси для деплою;
 - артефакт автоматично завантажується у репозиторій артефактів, наприклад, Nexus, Artifactory або Docker Hub та стає основою для наступного етапу в CI/CD – Continuous Deployment.

На рис. 11.1 наведено послідовність роботи CI.

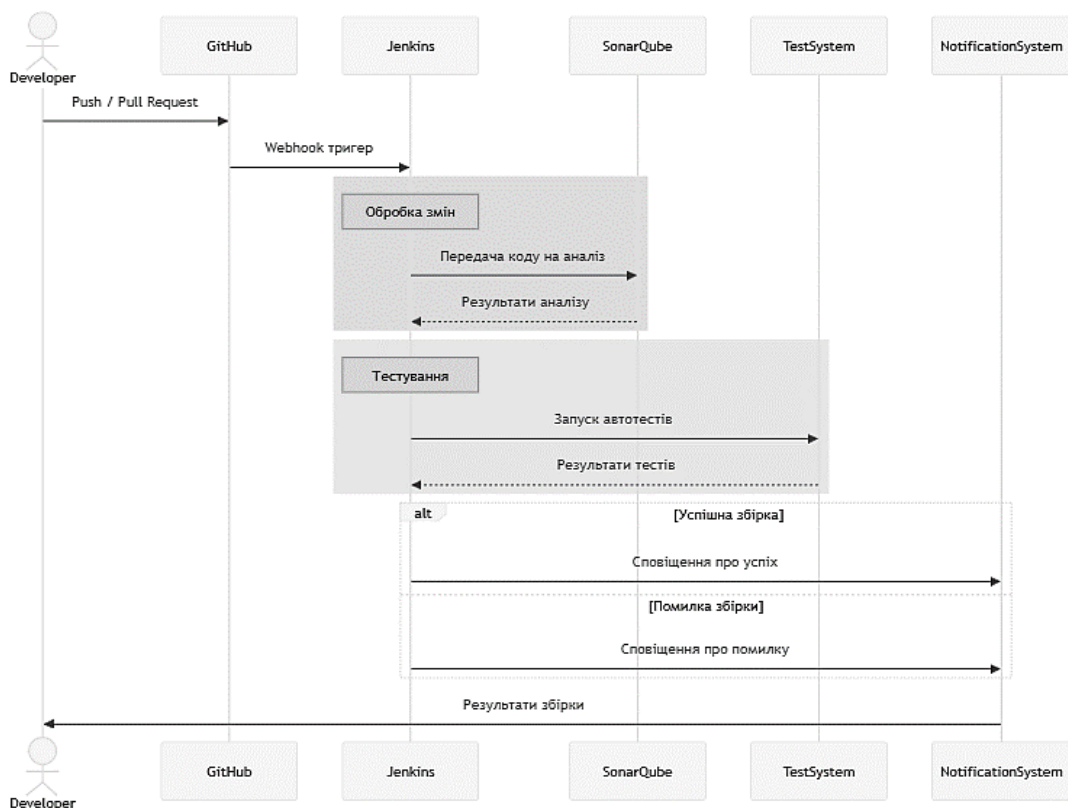


Рис. 11.1. Взаємодія між елементами в системі.

Інтеграція тестів і аналізу коду дозволяє швидко виявляти помилки на ранніх етапах розробки, що сприяє підвищенню якості продукту.

Автоматизація процесів мінімізує ручну роботу, знижує ризик людських помилок і значно пришвидшує виконання задач. Централізована звітність забезпечується через Jenkins, який надає єдине місце для моніторингу стану збірок, тестів і аналізу коду.

Готовність до деплоюменту досягається завдяки створенню збережених артефактів, які можна одразу використовувати в CD-процесі.

Послідовність роботи компонентів CI забезпечує автоматизований і структурований процес розробки, тестування та підготовки програмного забезпечення. Чітко визначені етапи та їх інтеграція через Jenkins, GitHub, SonarQube й інші інструменти створюють стабільну та надійну екосистему, яка сприяє підвищенню якості продукту та продуктивності команди.

11.3. Використання GitHub Action для побудови CI-системи

Розглянемо використання інструментів GitHub та GitHub Action для побудови мінімально укомплектованої CI-системи. З огляду на те, що наразі найкращим інструментом аналізу коду є SonarQube, як і в попередньому прикладі використаємо саме цей інструмент.

GitHub та GitHub Actions забезпечують зручну екосистему для автоматизації процесів CI/CD включаючи збірку, тестування та розгортання додатків, безпосередньо в репозиторії.

Для побудови CI-системи за допомогою GitHub Webhooks, яка інтегрується із SonarQube для аналізу якості коду, можна дотримуватись таких кроків:

1. Налаштування Webhooks в GitHub

Webhooks дозволяють GitHub автоматично повідомляти SonarQube або CI-систему про події в репозиторії (наприклад, push чи pull request). Для налаштування у репозиторії GitHub потрібно перейти до пункту Settings, обрати Webhooks та додати новий вебхук.

Наприклад, в даному випадку можуть бути наступні параметри:

- *Payload URL:* наприклад, URL сервера SonarQube `http://sonarqube.example.com/webhook`
- *Content type:* `application/json`.
- *Secret:* потрібно згенерувати секретний токен для перевірки автентичності;

- *Events*: слід обрати відповідні події push та/або pull_request залежно від потреб.

Потрібно перевірити, чи Webhook надсилає повідомлення. GitHub надає інструменти для діагностики у розділі Webhooks.

2. Налаштування GitHub Actions

Для роботи з GitHub Actions потрібно створити файл .github/workflows/sonar.yml у потрібному репозиторії для виконання автоматизованого аналізу коду з використанням SonarQube.

Нижче наведено фрагмент файлу конфігурації sonar.yml:

```
jobs:
  sonar_analysis:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Set up Java
        uses: actions/setup-java@v3
        with:
          java-version: 17

      - name: Cache SonarQube Scanner
        uses: actions/cache@v3
        with:
          path: ~/.sonar
          key: ${ runner.os }-sonar
          restore-keys: |
            ${ runner.os }-sonar

      - name: Run SonarQube Scanner
        env:
          SONAR_TOKEN: ${ secrets.SONAR_TOKEN }
        run: |
          sonar-scanner \
            -Dsonar.projectKey=<PROJECT_KEY> \
            -Dsonar.host.url=<SONARQUBE_SERVER_URL> \
            -Dsonar.login=${ secrets.SONAR_TOKEN }
```

3. Налаштування SonarQube

Для інтеграції інструменту аналізу коду в систему потрібно створити проєкт у SonarQube та згенерувати токен аутентифікації для GitHub Actions.

Відповідний токен потрібно додати до GitHub Secrets: у репозиторії GitHub потрібно перейти до налаштування (Settings) та в пункті Secrets and variables обрати Actions і додати секрет SONAR_TOKEN із токеном доступу SonarQube.

4. Інтеграція Webhooks та GitHub Actions

GitHub Webhooks працюють як тригери для запуску GitHub Actions при подіях (push, pull_request). Аналіз коду виконується автоматично через конфігурацію sonar.yml.

5. Перевірка результатів у SonarQube

Після виконання GitHub Actions результати аналізу коду з'являться у проєкті SonarQube.

6. Додавання сповіщень у CI-систему

Сповіщення дозволять оперативно інформувати команду про статус аналізу коду, результати Quality Gate та інші події. Як і попередньому прикладі, для сповіщень можна використати різні сервіси: Microsoft Teams, Telegram, Slack, Email тощо.

Так, наприклад, для інтеграції зі Slack потрібно:

- увійти у Slack і у розділі Apps обрати пункт Incoming Webhooks;
- створити новий Webhook та обрати канал, куди надсилатимуться сповіщення;
- скопіювати URL Webhook та додати до GitHub Secrets: для цього у GitHub репозиторії потрібно перейти до налаштувань (Settings) обрати пункт Secrets and variables та Actions відповідної додати секрет.

Далі потрібно оновити конфігурацію у файлі .github/workflows/sonar.yml, так , щоб включити сповіщення через Slack.

Для використання інших сервісів для сповіщень потрібно замінити Webhook відповідним API або інтеграцією та налаштувати конфігурацію у відповідному yaml-файлі.

Автоматичні сповіщення про результати аналізу в реальному часі дозволяють команді швидко реагувати на помилки або проблеми з якістю коду.

Послідовності роботи елементів у створеній CI

Розглянемо основні етапи роботи у запропонованій CI-структурі:

Процес роботи створеної CI системи являє собою комплексний автоматизований потік, який починається з дій розробника. Коли розробник вносить зміни до кодової бази через push команду або створює Pull Request у

GitHub репозиторії, це ініціює цілий ланцюг автоматизованих процесів. GitHub, отримавши такі зміни, миттєво активує налаштований webhook, який служить тригером для подальших дій. Час активації може залежати від завантаженості GitHub і мережових умов, проте зазвичай це відбувається миттєво.

Активований webhook передає керування до GitHub Actions, де починається виконання попередньо налаштованого workflow, визначеного у конфігураційному файлі sonar.yml. На цьому етапі GitHub Actions готує необхідне ізольоване середовище (контейнери або віртуальні машини) для проведення аналізу коду, завантажуючи всі потрібні залежності та інструменти.

Наступним кроком у послідовності є взаємодія з системою аналізу якості коду SonarQube, яка має наступні кроки:

- GitHub Actions передає код до SonarQube для аналізу;
- SonarQube проводить статичний аналіз коду;
- результати аналізу зберігаються в SonarQube;
- SonarQube зберігає результати аналізу у своїй базі даних;
- GitHub Actions отримує статус перевірки, надсилаючи API-запити до SonarQube.

SonarQube проводить комплексну перевірку коду, включаючи пошук потенційних помилок, оцінку якості коду, перевірку дотримання стандартів кодування та багато інших метрик. Всі результати аналізу зберігаються в базі даних SonarQube для подальшого використання та порівняльного аналізу з попередніми версіями коду.

Після завершення аналізу GitHub Actions отримує детальний звіт від SonarQube і на його основі оновлює статус перевірки безпосередньо в Pull Request або коміті. Це дозволяє іншим членам команди миттєво бачити, чи пройшов код необхідні перевірки якості. На базі цих результатів система також генерує інформативне сповіщення, яке містить ключові метрики та виявлені проблеми.

Важливим елементом процесу є система сповіщень, яка забезпечує оперативне інформування команди. У даному випадку налаштована інтеграція зі Slack, через який команда отримує миттєві повідомлення про результати аналізу. Це дозволяє розробникам швидко реагувати на виявлені проблеми та підтримувати високу якість коду.

У випадку, якщо в процесі аналізу були виявлені проблеми або код не відповідає встановленим стандартам якості, запускається цикл виправлень. Розробники отримують детальну інформацію про виявлені проблеми, вносять необхідні виправлення у код та роблять новий коміт. Цей новий коміт автоматично запускає весь процес аналізу знову, створюючи таким чином безперервний цикл покращення якості коду.

На рис. 11.2 наведена схема взаємодії між компонентами CI системи.

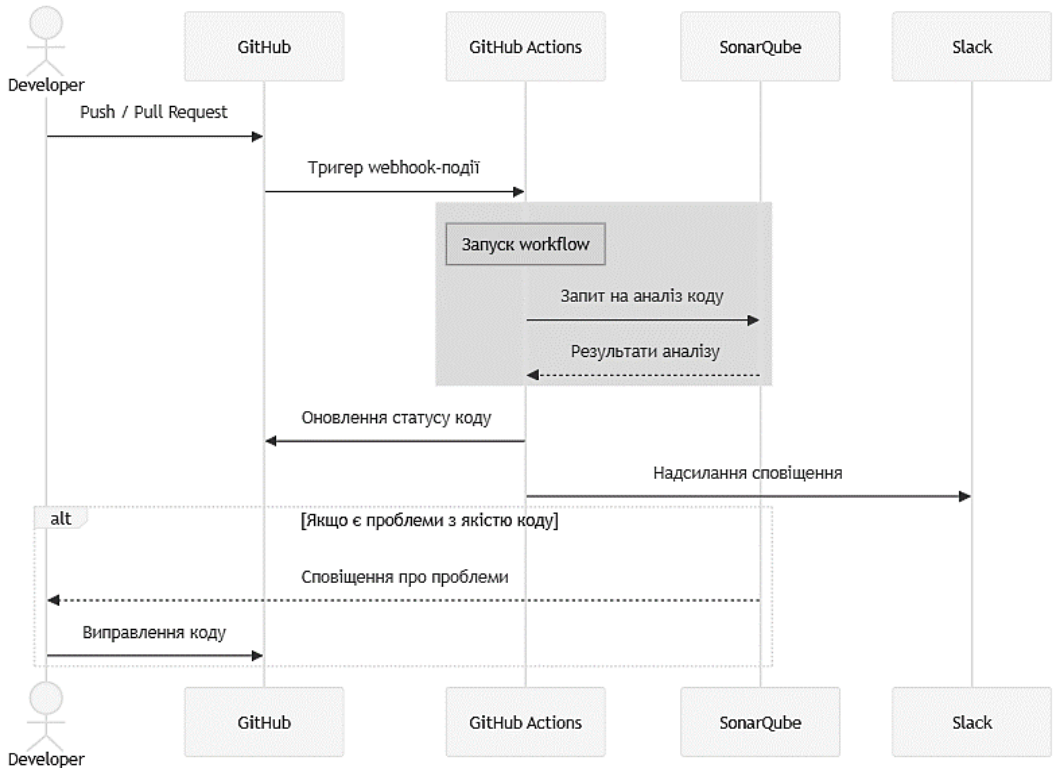


Рис. 11.2. Взаємодія між елементами CI системи.

Вся розглянута послідовність працює як єдиний автоматизований механізм, який не потребує ручного втручання після початкового налаштування. Це забезпечує стабільну та надійну перевірку якості коду, дозволяючи команді зосередитися на розробці функціоналу, в той час як система автоматично відстежує та контролює якість внесених змін. Така автоматизація значно пришвидшує процес розробки та допомагає підтримувати високі стандарти якості коду протягом всього життєвого циклу проекту.

Контрольні питання до розділу 11

- 1) Які основні компоненти може містити мінімально укомплектована CI-система?
- 2) Що таке Webhooks і яке їх призначення?
- 3) Чому використання Webhooks є ефективним?
- 4) Яким чином можна реалізувати інтеграцію Jenkins з GitHub?

- 5) Що таке артефакт в контексті СІ і яке його призначення?
- 6) З якою метою будуються СІ-системи в процесі розробки?
- 7) Які основні етапи СІ-процесу можна виділити?
- 8) Які існують засоби безпеки в СІ-системі при інтеграції з зовнішніми сервісами?
- 9) Які канали нотифікацій підтримує Jenkins?
- 10) З якою метою використовується cron-стилю в Jenkins?
- 11) Що може бути результатом успішного виконання СІ-процесу?
 - а) Максимальний аналіз вмісту репозиторію.
 - б) Виявлення синтаксичних помилок в коді.
 - с) Оновлення інструменту SonarQube.
 - д) Створення артефакту для подальшого розгортання.
 - е) Виявлення «мертвого коду».
- 12) Який порт за замовчуванням використовує Jenkins?
 - а) 8080
 - б) 3000
 - с) 9000
 - д) 443
 - е) 10101
- 13) Які події GitHub можуть викликати активацію Webhook?
 - а) Видалення репозиторію.
 - б) Push або Pull Request.
 - с) Додавання коментарів до коду
 - д) Оновлення вмісту README.md
 - е) Зміна власника репозиторію
- 14) Що означає символ "H" в cron-виразах Jenkins?
 - а) Hour
 - б) Help
 - с) Hash
 - д) High
 - е) Host
- 15) Як часто виконується завдання з cron-виразом "H/10 * * * *"?
 - а) Кожні 10 годин
 - б) Кожні 10 хвилин
 - с) Раз на 5 днів
 - д) Кожні 10 секунд
 - е) Лише 10 разів за добу

- 16) Який з перерахованих методів інтеграції забезпечує найшвидшу реакцію на зміни в репозиторії?
- a) Git Plugin
 - b) Webhooks
 - c) Polling SCM
 - d) Manual trigger
 - e) Scheduled builds
- 17) Який протокол рекомендується використовувати для безпечного з'єднання з Jenkins?
- a) FTP
 - b) POP3
 - c) SMTP
 - d) TCP
 - e) HTTPS
- 18) Який тип аутентифікації рекомендується для HTTPS з'єднання з GitHub?
- a) Basic Authentication
 - b) API Key
 - c) Personal Access Token
 - d) Username/Password
 - e) OAuth Token
- 19) Що з перерахованого забезпечує доступ Jenkins до репозиторію у закритій мережі?
- a) NAT або VPN
 - b) Slack Plugin
 - c) Quality Gate
 - d) Тунелювання через ngrok
 - e) SonarQube
- 20) Який інструмент може бути використано для аналізу коду у CI-процесі?
- a) Jenkins
 - b) GoogleCloud
 - c) SonarScanner
 - d) PostgreSQL
 - e) Docker

21) Встановіть відповідність між етапами СІ та їх очікуваними результатами:

A. Збірка	1) Побудований артефакт проєкту.
B. Тестування	2) Звіт про виявлені помилки.
C. Аналіз коду	3) Отримання сповіщення учасниками команди.
D. Нотифікація	4) Проаналізовані метрики якості коду.

22) Встановіть відповідність між термінами та їхніми значеннями:

A. Firewall	1) Механізм автоматичної передачі інформацію між сервісами на основі HTTP-запитів.
B. Артефакт	2) Результат успішної збірки проєкту.
C. DevOps	3) Механізм контролю трафіку в комп'ютерній мережі.
D. Webhook	4) Набір практик та інструментів для оптимізації розробки ІС.

23) Встановіть відповідність між компонентами СІ-системи та їхнім призначенням:

A. GitHub	1) Автоматизація процесів CI/CD.
B. Jenkins	2) Аналізу якості коду.
C. SonarQube	3) Керування вихідним кодом та спільна робота.
D. Git	4) Зберігання коду, робота з репозиторіями.

24) Встановіть відповідність між підходами до запуску завдань та їхніми характеристиками:

A. Webhooks	1) Планування завдань за заданим розкладом.
B. Cron-стиль	2) Періодична перевірка змін у репозиторії.
C. Polling SCM	3) Миттєвий запуск після події у репозиторії.
D. Manual Trigger	4) Ручний запуск завдань користувачем.

25) Встановіть відповідність між каналами нотифікацій та їхніми особливостями:

A. Email	1) Чат-повідомлення для командної співпраці.
B. Slack	2) Статус комітів і pull request.
C. GitHub	3) Формальні звіти для менеджерів або адміністраторів.
D. Discord	4) Миттєві сповіщення у канали або приватні чати.

РОЗДІЛ 12. ПРАКТИЧНІ ЗАВДАННЯ, ПОВ'ЯЗАНІ З ІНТЕГРАЦІЄЮ ІС

В даному розділі містяться практичні завдання за темами, розглянутими в попередніх розділах. Виконання запропонованих завдань має на меті ознайомлення з особливостями забезпечення супроводу програмного продукту протягом його життєвого циклу в рамках Development Operations підходу та сприятиме кращому розумінню матеріалу, розглянутому вище.

12.1. Вибір проєкту та налаштування робочого середовища

Перед початком роботи варто підготувати середовище та об'єкт опрацювання для подальших завдань.

Середовище для роботи

В якості робочого середовища зручно використовувати віртуальні машини завдяки їх безпеці та універсальності. Всі зміни у віртуальному середовищі не впливають на основну операційну систему, що дозволяє експериментувати без ризику для хост-машини. Крім того, віртуальні машини можна легко клонувати, створювати резервні копії та переносити між різними фізичними комп'ютерами.

Для створення віртуальної машини можна скористатися локальним гіпервізором або хмарними сервісами. Серед локальних рішень найпопулярнішим є Oracle VirtualBox – безкоштовний гіпервізор з відкритим кодом, що підтримує всі основні операційні системи. Для професійного використання також часто обирають VMware Workstation, який має розширені можливості для налаштування мережі та інтеграції з корпоративною інфраструктурою. Користувачі Windows можуть використовувати вбудований Hyper-V, а в Linux – нативний KVM.

Хмарні платформи такі як AWS, Google Cloud та Microsoft Azure пропонують віртуальні машини як сервіс, що дозволяє швидко розгортати середовища без витрат на власну інфраструктуру та з можливістю гнучкого масштабування. Хмарні рішення особливо корисні для виробничих середовищ, де потрібна висока доступність та можливість швидкого масштабування ресурсів.

Для вибору цього варіанту в якості робочого середовища для виконання запропонованих завдань бажано, щоб користувач мав навички роботи з зазначеними сервісами:

- *EC2* для Amazon Web Services (AWS);
- *Compute Engine* для Google Cloud Platform (GCP);
- *Virtual Machines* для Microsoft Azure.

Створену віртуальну машину потрібно підготувати для роботи. Процес налаштування віртуальної машини зазвичай включає вибір операційної системи, виділення обчислювальних ресурсів (процесор, пам'ять, диск) та базове налаштування мережі.

У хмарних платформах можна використовувати готові образи з попередньо встановленим програмним забезпеченням, що значно прискорює розгортання. Для роботи з VM на одній з зазначених платформ потрібно:

- *створити обліковий запис на обраній платформі або скористатись існуючим;*
- *перейти до створення VM за допомогою відповідного сервісу;*
- *вибрати регіон розміщення;*
- *налаштувати конфігурацію VM (vCPU, RAM, диск);*
- *вибрати образ ОС;*
- *налаштувати мережеву безпеку;*
- *згенерувати SSH-ключі для доступу та налаштувати доступ за потреби.*

Однак, варто звернути увагу, що використання хмарних платформ для навчання має кілька суттєвих ризиків.

По-перше, більшість хмарних провайдерів вимагають прив'язки банківської картки навіть для безкоштовного пробного періоду. Неуважність при налаштуванні ресурсів може призвести до неочікуваних витрат, оскільки, наприклад, забуті запущені віртуальні машини продовжують тарифікуватися.

По-друге, хмарні платформи мають складні системи безпеки та доступу - помилки в налаштуванні можуть відкрити доступ до ресурсів зловмисникам, які часто використовують скомпрометовані хмарні машини для криптомайнінгу або DDoS-атак. Це може призвести як до значних фінансових втрат, так і до блокування облікового запису. Тому для початкового навчання краще використовувати локальні гіпервізори, де ризики обмежені лише власним комп'ютером.

При використанні локальних гіпервізорів для налаштування середовища потрібно створити VM, завантажити образ операційної системи та пройти стандартний процес інсталяції.

Розглянемо алгоритм налаштування в якості робочого середовища віртуальної машини, створеної засобами VirtualBox:

1. **Завантажити *VirtualBox*** з офіційного веб-сайту Oracle (<https://www.virtualbox.org/>) та встановити, дотримуючись інструкцій для операційної системи відповідної хост-машини.
2. **Завантажити образ** ISO-файлу дистрибуції Ubuntu з офіційного веб-сайту Ubuntu (<https://ubuntu.com/download>). Можна обрати LTS (довгострокову підтримку) або останню версію Ubuntu, в залежності від

потреб. Варто звернути увагу, що всі розміщені далі в якості прикладів команди орієнтовані на Ubuntu 20.04 і потребують уточнення для інших варіантів робочого середовища.

3. **Створити нову віртуальну машину в VirtualBox:**

- відкрити VirtualBox і натиснути кнопку "New";
- ввести ім'я віртуальної машини, вибрати тип "Linux" та "Ubuntu (64-bit)" в якості версії, якщо завантажувється 64-бітна версія Ubuntu;
- вибрати обсяг оперативної пам'яті (RAM): рекомендується вибрати не менше 2 ГБ оперативної пам'яті, однак, якщо ресурси дозволяють, можна обрати більше;
- створити віртуальний жорсткий диск: потрібно обрати "Create a virtual hard disk now" (Створити віртуальний жорсткий диск зараз) та натиснути "Create". Далі слід обрати тип файлу віртуального диска (рекомендовано використовувати "VDI") та обрати "Dynamically allocated" (Динамічно виділений) для виділення місця на диску тільки за необхідності. Останнім кроком є вибір розміру віртуального диска: Рекомендований розмір – не менше 20 ГБ, однак можна вибрати більший розмір, якщо ресурси дозволяють.

4. **Налаштувати створену віртуальну машину:**

- вибрати нову віртуальну машину в списку VirtualBox та перейти до пункту "Settings" (Налаштування). В розділі "System" (Система) переконатись, що обрано CD/DVD віртуальний привід та встановлено "Boot Order" (Порядок завантаження) з підтримкою CD/DVD, що дозволить завантажити з образу ISO.
- додати ISO-файл Ubuntu до віртуальної машини: в розділі "Storage" (Сховище) обрати контролер IDE і натиснути на зображення диска CD/DVD. Далі слід обрати опцію "Choose a disk file" (Вибрати файл диска) та вибрати завантажений ISO-файл Ubuntu.

5. **Запустити віртуальну машину:** натиснути "Start" (Запустити) для запуску віртуальної машини; у вікні віртуальної машини обрати "Install Ubuntu" (Встановити Ubuntu) та слідувати інструкціям на екрані для встановлення операційної системи.

6. **Налаштувати Ubuntu:** після встановлення Ubuntu на віртуальній машині слід виконати необхідні початкові налаштування, такі як, наприклад, створення облікового запису користувача, встановлення програм та пакетів за необхідності.

Об'єкт опрацювання

Для подальшої практичної роботи потрібно підготувати програмний продукт – обрати деякий проєкт, який буде використовуватись при виконанні завдань. Він може бути створений на довільній платформі засобами не інтерпретованої мови програмування та має передбачати кінцевий результат у вигляді вебдодатку і вимагати процесу компіляції. Застосунок може бути створений власноруч або завантажений з мережі Інтернет.

Після вибору проєкту потрібно обов'язково переконатись, що програмний продукт відповідає поставленим вимогам та коректно запускається.

В якості прикладу наведено вибір проєкту «Vue Paper Dashboard», що знаходиться у вільному доступі за адресою: <https://github.com/creativetimofficial/vue-paper-dashboard> (рис. 12.1). Проєкт відповідає поставленим вимогам, оскільки розроблений на основі Bootstrap і Vue та являє собою портовану версію оригінальної паперової панелі Vue.

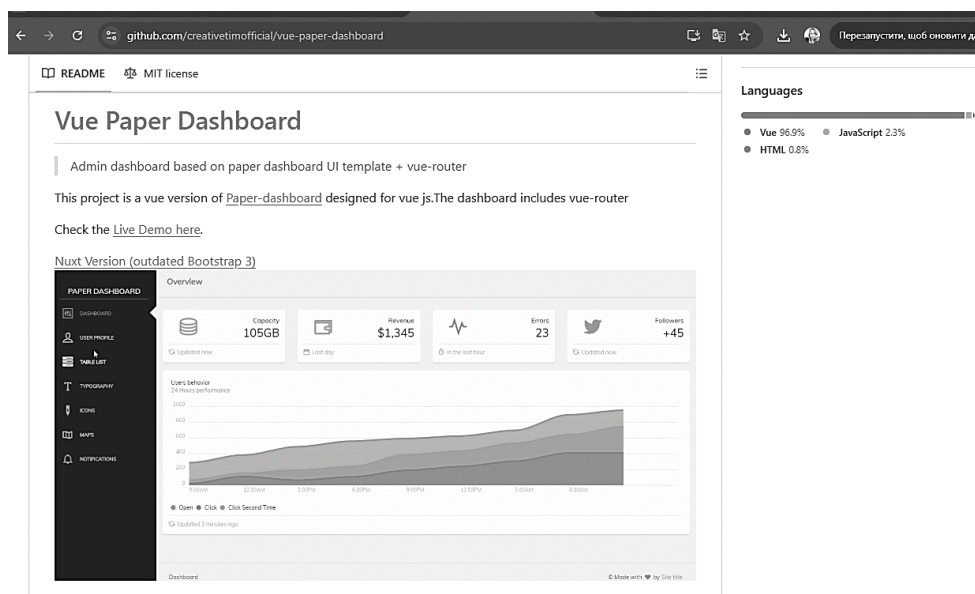


Рис. 12.1. Сторінка з описом вмісту репозиторію з проєктом

Для подальшої роботи з обраним проєктом потрібно завантажити його локально:

```
git clone https://github.com/creativetimofficial/vue-paper-dashboard
```

Після завантаження необхідно переконатись, що обраний застосунок працює коректно. Для перевірки обраного в якості прикладу проєкту «Vue Paper Dashboard» потрібно проробити наступні кроки:

- встановити *npm* за допомогою команди

apt install npm

```

cheapyd@cheapyd-VirtualBox:~$ sudo apt install npm
[sudo] password for cheapyd:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:

```

- *встановити пакети та модулі до проєкту:* за допомогою команди

npm install

```

cheapyd@cheapyd-VirtualBox:~/vue-paper-dashboard$ npm install
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: 'eslint-plugin-vue@9.9.0',
npm WARN EBADENGINE   required: { node: '^14.17.0 || >=16.0.0' },
npm WARN EBADENGINE   current: { node: 'v12.22.9', npm: '8.5.1' }
npm WARN EBADENGINE }
npm WARN EBADENGINE Unsupported engine {
npm WARN EBADENGINE   package: 'vue-eslint-parser@9.1.0',
npm WARN EBADENGINE   required: { node: '^14.17.0 || >=16.0.0' },
npm WARN EBADENGINE   current: { node: 'v12.22.9', npm: '8.5.1' }
npm WARN EBADENGINE }

```

- *запустити проєкт:* командою npm run dev

```

cheapyd@cheapyd-VirtualBox:~/vue-paper-dashboard$ npm run dev

> vue-paper-dashboard@1.0.1 dev
> npm run serve

> vue-paper-dashboard@1.0.1 serve
> vue-cli-service serve --open

INFO Starting development server...
40% building 156/179 modules 23 active .../lib/runtime/componentNormalizer.js
Deprecation Warning: Using / for division outside of calc() is deprecated and will be removed in Dart Sass 2.0.0.

```

Наведені кроки є специфічними саме для мови програмування NodeJS і мають бути модифіковані у випадку інших засобів розробки.

Після успішної побудови та запуску обраного проєкту потрібно переглянути результат (рис. 12.2):

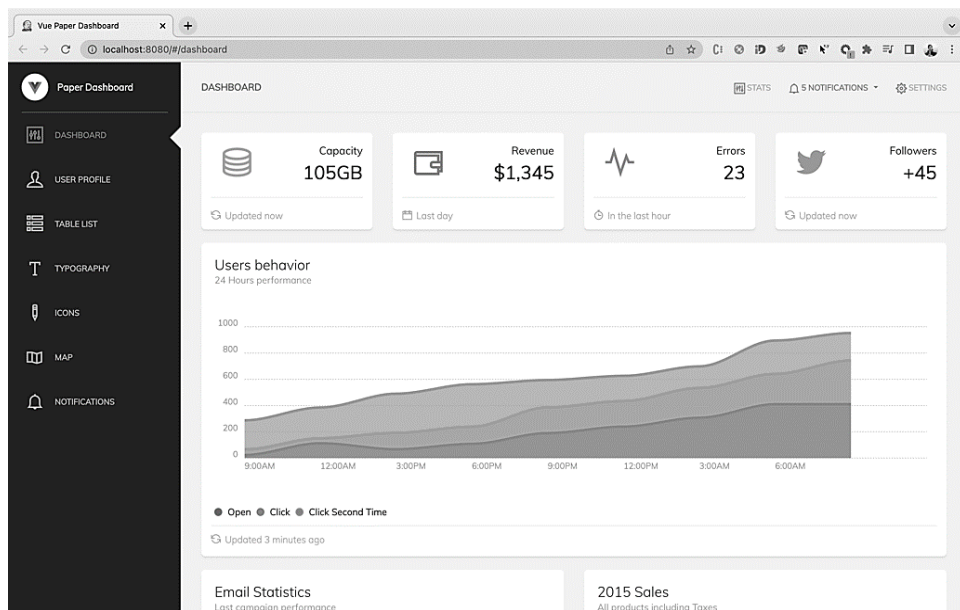


Рис. 12.2. Перегляд запущеного застосунку в браузері

12.2. Робота з Github репозиторієм

GitHub дозволяє розробникам співпрацювати над проектами, використовуючи систему контролю версій Git, і містить в собі низку інструментів, таких як інструменти звітування, систему відстеження проблем, функції інтеграції з іншими сервісами та інші. З метою узагальнення навичок роботи з Github репозиторієм варто підготувати Git репозиторій для подальшої роботи з інструментами DevOps; вдосконалити навички виконання простих операцій над ним.

Для цього можна виконати наступні завдання.

1. Зареєструвати обліковий запис на веб-сайті *GitHub.com*

Для реєстрації потрібно перейти до сервісу GitHub.com, ввести власне ім'я, адресу електронної пошти та пароль. Якщо є відповідний обліковий запис – можна використати його.

2. Створити новий репозиторій для роботи

Для створення репозиторію потрібно увійти до свого облікового запису на GitHub.com і обрати кнопку "New" на сторінці свого профілю (рис.12.3).

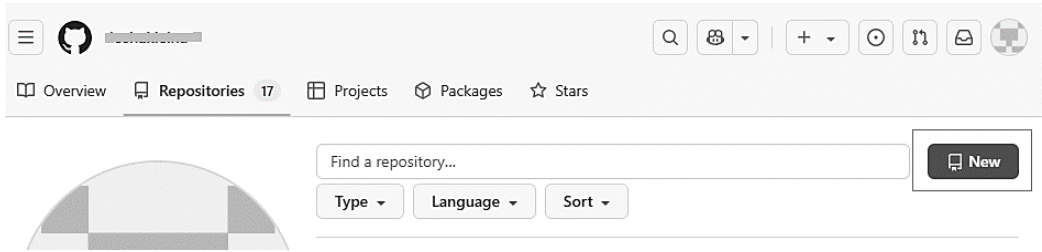


Рис. 12.3. Вибір функції створення нового репозиторію.

На сторінці "Create a new repository", що відкриється після натискання на кнопку, потрібно вказати назву свого репозиторію. Назва має бути унікальною для вашого облікового запису. Також опціонально можна ввести короткий опис свого репозиторію, щоб пояснити його призначення та зміст.

Далі слід обрати тип доступу до нового репозиторію (приватний чи публічний). Публічні репозиторії доступні для перегляду та клонування будь-яким користувачем інтернету, їх можна знайти через пошук GitHub. Вони найкраще підходять для проєктів з відкритим кодом, навчальних матеріалів та портфоліо. Якщо репозиторій приватний, то лише користувачі, які мають доступ до його URL, зможуть його переглядати та завантажувати. Їх варто використовувати для комерційних проєктів, внутрішньої документації компанії або коду, який не потрібно поки показувати публічно. В даному випадку можна обрати публічний репозиторій, щоб не виникало проблем з доступом до нього за потреби.

За потреби можна також додати опис ліцензії свого репозиторію. Популярні ліцензії включають MIT (максимально ліберальна), GPL (вимагає збереження відкритого коду) та Apache (захищає патенти).

Якщо потрібно додати файл README.md до нового репозиторію, потрібно обрати опцію «Initialize this repository with a README» (ініціалізувати цей репозиторій з README.md файлом). Цей файл є важливим елементом будь-якого репозиторію, оскільки це перше, що бачать відвідувачі. В ньому можна описати призначення проєкту, інструкції з встановлення та використання, вимоги до середовища розробки тощо. GitHub автоматично відображає вміст README.md на головній сторінці репозиторію.

Owner * / Repository name *

information_systems_integ

information_systems_integration is available.

Great repository names are short and memorable. Need inspiration? How about [potential-octo-palm-tree](#) ?

Description (optional)

DevOps Practical// working repository for studying the features of CI/CD tools

☒ Public
Anyone on the internet can see this repository. You choose who can commit.

☐ Private
You choose who can see and commit to this repository.

Initialize this repository with:

☒ Add a README file
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Choose a license

License: MIT License

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

This will set `main` as the default branch. Change the default name in your [settings](#).

i You are creating a public repository in your personal account.

Create repository

Рис. 12.4. Створення репозиторію на GitHub.

Після заповнення всіх необхідних відомостей потрібно підтвердити створення репозиторію натисканням на кнопку "Create repository" (Створити репозиторій) (рис. 12.4). Після завершення створення буде доступна URL-адреса репозиторію та надані інструкції щодо початку роботи з ним.

3. Завантажити проєкт до створеного репозиторію

Після створення репозиторію на GitHub, потрібно завантажити до нього обраний для практичної роботи проєкт. Це можна зробити безпосередньо з терміналу Ubuntu чи скористатись іншими засобами (наприклад, через вбудований термінал Visual Studio Code).

Для цього потрібно спочатку ініціалізувати Git у директорії з потрібним проєктом та встановити зв'язок з віддаленим репозиторієм. Для цього можна скористатись командами:

```
git init
```

```
git remote add origin <url-адреса-вашого-репозиторію>
```

```
git add .
```

```
git commit -m "Initial project commit"
```

```
git push -u origin main
```

Остання команда відправляє зміни у віддалений репозиторій на GitHub. Варто звернути увагу, що під час першого використанні Git потрібно налаштувати облікові дані. Також, для автентифікації на GitHub рекомендується використовувати персональний токен доступу (Personal Access Token) замість пароля. Його можна згенерувати в налаштуваннях облікового запису GitHub.

Також можна спочатку клонувати створений порожній репозиторій на локальний комп'ютер. Для цього потрібно відкрити термінал, перейти у директорию де слід розмістити проєкт та виконати команду:

```
git clone <url-адреса-вашого-репозиторію>
```

Після клонування репозиторію, потрібно скопіювати файли обраного для роботи проєкту в створену директорию та аналогічно описаному вище додати файли до Git, створити коміт та відправити зміни на GitHub.

Після повідомлення про успішне завантаження файлів потрібно відкрити відповідний репозиторій на GitHub, щоб переконатись, що потрібні файли були успішно завантажені (рис. 12.5)

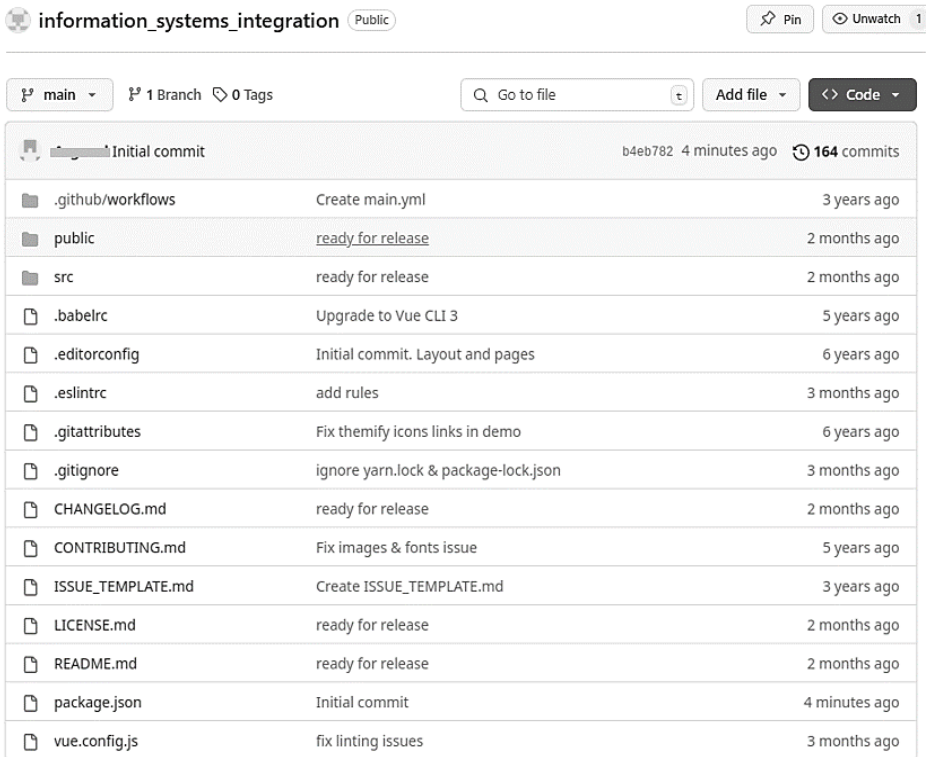


Рис. 12.5. Доданий вміст репозиторію.

Після завантаження даних можна наприклад, внести зміни у файл README.md та оновити дані у репозиторії.

4. Створити нову гілку dev, внести зміни у ній та виконати злиття

Для створення нової гілки можна скористатись командою

```
git checkout -b dev
```

Після цього потрібно внести деякі зміни в проєкт – оновити вміст README.md чи додати для прикладу деякий файл з описом. Внесені зміни потрібно закомітити:

```
git add .
```

```
git commit -m "Changes in dev branch"
```

```
git push origin dev
```

Для злиття гілок можна скористатись командами:

```
git checkout main
```

```
git pull origin main
```

```
git merge dev
```

```
git push origin main
```

Команда *git checkout main* переключає нас на головну гілку. Команда *git pull* потрібна для отримання актуального стану головної гілки перед злиттям, щоб уникнути конфліктів. Команда *git merge dev* виконує саме злиття, тобто переносить всі зміни з гілки dev у поточну гілку main. Якщо виникнуть конфлікти при злитті (коли одні й ті ж файли були змінені в обох гілках), Git позначить їх у файлах, і потрібно буде вирішити їх вручну, потім виконати commit та push.

Після успішного злиття гілку dev можна видалити, якщо вона більше не потрібна:

```
git branch -d dev
```

```
git push origin --delete dev
```

Таким чином, створення та налаштування репозиторію на GitHub, разом з базовими операціями додавання та оновлення файлів, дозволяє організувати ефективну роботу з кодом проєкту та забезпечити його надійне зберігання у хмарі. Використання системи гілок Git надає можливість безпечно експериментувати з кодом у окремих гілках та контролювано впроваджувати зміни у основну версію проєкту через механізм злиття гілок, що є важливим елементом професійної розробки програмного забезпечення.

12.3. Робота з репозиторіями у GitLab

GitHub містить в собі систему контролю версій Git, інструменти для збору, тестування та розгортання програмного забезпечення, інструменти звітування та систему відстеження проблем. GitLab дозволяє розробникам ефективно співпрацювати, зберігати та керувати версіями свого програмного забезпечення в одному місці, що зменшує час та зусилля, необхідні для розробки та випуску продукту. Наведені нижче завдання орієнтовані на роботу безпосередньо з терміналу Ubuntu (орієнтовані на Ubuntu 20.04) та мають на меті практичне ознайомлення з даним інструментом.

1. Встановлення інструменту GitLab

Перед встановленням GitLab рекомендується виконати оновлення системи командою:

```
sudo apt update && sudo apt upgrade
```

Після успішного оновлення потрібно встановити залежності GitLab:

```
sudo apt install -y curl openssh-server ca-certificates tzdata perl
```

Для завантаження і встановлення GitLab потрібно виконати наступні команди:

```
curl -sS https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/script.deb.sh | sudo bash
```

```

~/vue-paper-dashboard$ curl -sS https://packages.gitlab.com/install
Detected operating system as Ubuntu/jammy.
Checking for curl...
Detected curl...
Checking for gpg...
Detected gpg...
Running apt-get update... done.
Installing apt-transport-https... done.
Installing /etc/apt/sources.list.d/gitlab_gitlab-ce.list...done.
Importing packagecloud gpg key... done.
Running apt-get update... done.

```

```
sudo apt install gitlab-ce
```

```

~/vue-paper-dashboard$ sudo apt install gitlab-ce
Зчитування переліків пакунків... Виконано
Побудова дерева залежностей... Виконано
Зчитування інформації про стан... Виконано
НОВІ пакунки, які будуть встановлені:
gitlab-ce

```

2. Налаштування GitLab

Після встановлення GitLab потрібно буде налаштувати його для подальшої роботи. Для того, щоб відкрити файл конфігурації GitLab, потрібно скористатись командою:

```
sudo vim /etc/gitlab/gitlab.rb
```

У зазначеному файлі можна провести налаштування необхідних параметрів, таких як порт, доменні імена та інші.

Щоб отримати доступ до інтерфейсу GitLab потрібно відкрити веб-оглядач (браузер) та ввести адресу `http://<ваша_IP_адреса>` або `http://<ваше_доменне_ім'я>`. При першому вході потрібно буде створити адміністративний обліковий запис та налаштувати пароль (рис. 12.6).

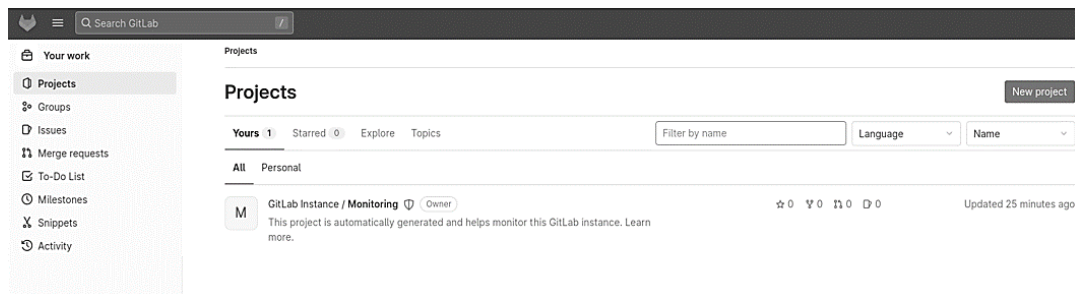


Рис. 12.6. Веб інтерфейс сервісу GitLab.

3. *Робота з репозиторіями у GitLab*

Після виконання попередніх кроків потрібно розглянути особливості роботи з даними. Для цього можна створити нову гілку, дати їй назву (наприклад, `test`) та внести деякі зміни у обраний для роботи проєкт (як і в попередньому завданні це може бути оновлення вмісту `README.md` чи додавання деякого тестового файл з описом).

Після збереження змін у локальній гілці потрібно завантажити зміни на GitLab:

```
git push origin test
```

Після успішного завантаження потрібно перевірити, що гілка з'явилася у проєкті на GitLab.

Далі потрібно виконати злиття гілок (рис. 12.7). Для цього можна скористатись наступним алгоритмом:

- ✓ увійти в веб-інтерфейс GitLab та перейти у репозиторій проєкту;
- ✓ вибрати вкладку Merge Requests у меню і натиснути кнопку New Merge Request;
- ✓ обрати Source branch (створену гілку `test`) та Target branch (основну гілку `main`) та натиснути Compare branches and continue;
- ✓ додати опис до Merge Request та натиснути Create Merge Request, щоб створити запит на злиття;

- ✓ натиснути кнопку Merge для злиття змін у гілку main, якщо всі перевірки пройдені і немає конфліктів;
- ✓ видалити гілку test у GitLab, натиснувши Delete branch у Merge Request.

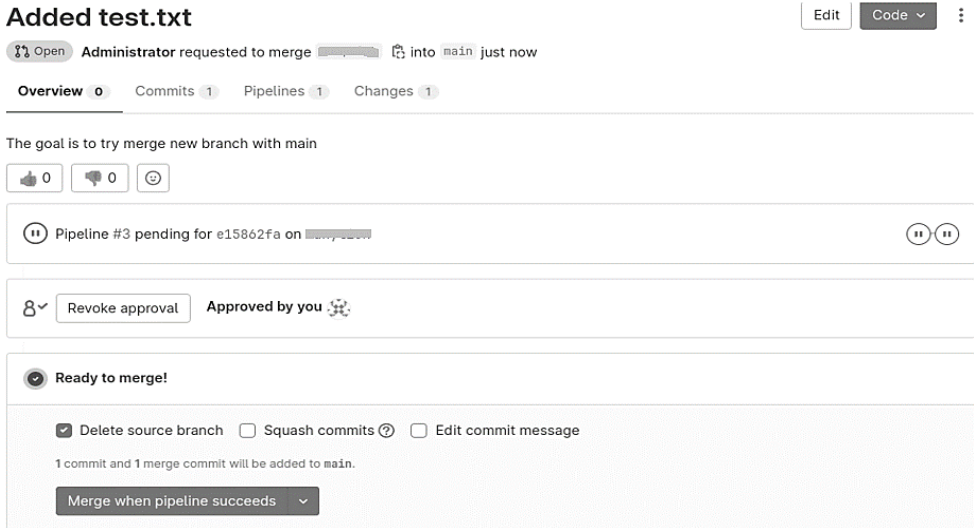


Рис. 12.7. Робота з гілками у GitLab.

Отже, процес роботи з GitLab включає встановлення та налаштування середовища, створення й управління репозиторіями та роботу з гілками, що робить його ефективним інструментом для командної розробки проєктів.

12.4. Робота з Jenkins. Налаштування Pipeline сценарію

Jenkins є одним з найпопулярніших інструментів безперервної інтеграції та доставки (CI/CD), який протягом багатьох років демонструє надійність у автоматизації рутинних процесів розробки програмного забезпечення - від збірки та тестування до розгортання. Завдяки відкритій архітектурі та великій активній спільноті розробників Jenkins може бути легко інтегрований з майже будь-яким інструментом розробки та адаптований під специфічні потреби проєкту, що робить його універсальним рішенням для команд різного розміру та спеціалізації.

Jenkins надає можливість описувати процеси автоматизації у вигляді коду (Pipeline as Code), що дозволяє не лише зберігати конфігурацію разом із кодом проєкту, але й інтегрувати її у систему контролю версій, забезпечуючи прозорість та відстежуваність змін у процесах збірки та розгортання.

Розглянемо створення та налаштування Jenkins сценарію практично.

1. Встановлення інструменту Jenkins

Перед встановленням потрібно оновити систему. Для цього в терміналі терміналу Ubuntu (в якості робочого середовища розглядається Ubuntu 20.04) потрібно виконати команду, щоб оновити систему:

```
sudo apt update && sudo apt upgrade
```

Після успішного оновлення потрібно встановити Java (якщо не встановлено заздалегідь), оскільки Jenkins вимагає Java для роботи:

```
sudo apt install -y default-jdk
```

```
cheapyd@cheapyd:~$ sudo apt install -y default-jdk
Зчитування переліків пакунків... Виконано
Побудова дерева залежностей... Виконано
Зчитування інформації про стан... Виконано
The following additional packages will be installed:
```

Наступним кроком є додавання ключа репозиторію Jenkins та джерела для встановлення:

```
wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add
```

```
sudo sh -c 'echo deb https://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
```

Після успішного виконання попередніх команд перед встановленням Jenkins на Ubuntu потрібно переконаватися, що система містить актуальні дані про доступні пакети. Це робиться за допомогою команди оновлення списку пакетів:

```
sudo apt update
```

Ця команда синхронізує локальний індекс пакетів з репозиторіями, забезпечуючи доступ до найновіших версій програмного забезпечення. Після оновлення списку пакетів можна приступити до встановлення Jenkins командою:

```
sudo apt install -y jenkins
```

```
cheapyd@cheapyd:~$ sudo apt install -y jenkins
[sudo] password for cheapyd:
Зчитування переліків пакунків... Виконано
Побудова дерева залежностей... Виконано
Зчитування інформації про стан... Виконано
НОВІ пакунки, які будуть встановлені:
jenkins
оновлено 0, встановлено 1 нових, 0 відмічено для видалення і 4 не оновлено.
Необхідно завантажити 0 B/96,1 MB архівів.
Після цієї операції об'єм зайнятого дискового простору зросте на 98,4 MB.
Selecting previously unselected package jenkins.
(Reading database ... 345939 files and directories currently installed.)
Preparing to unpack .../jenkins_2.387.1_all.deb ...
Unpacking jenkins (2.387.1) ...
Setting up jenkins (2.387.1) ...
```

Параметр `-y` автоматично підтверджує згоду на встановлення пакету та всіх необхідних залежностей, що спрощує процес інсталяції. Під час встановлення система автоматично створить системного користувача `jenkins`, налаштує його як системний сервіс та запустить його на порту 8080 за замовчуванням.

2. *Запуск Jenkins*

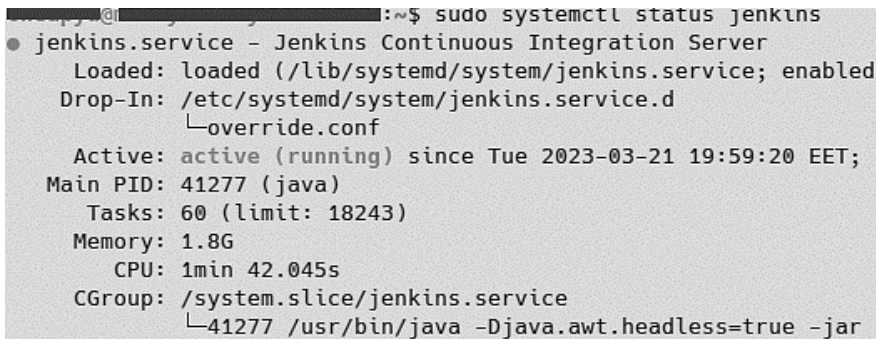
Після встановлення Jenkins потрібно запустити службу:

```
sudo systemctl start jenkins
```

В якості наступного кроку варто перевірити стан служби Jenkins:

```
sudo systemctl status jenkins
```

Ця команда покаже детальну інформацію про статус сервісу, включаючи чи активний він (`running/active`), час запуску, використання системних ресурсів та останні записи журналу. Якщо сервіс успішно запущений, буде виведено статус `"active (running)"` та зелений індикатор.



```
jenkins@jenkins:~$ sudo systemctl status jenkins
● jenkins.service - Jenkins Continuous Integration Server
   Loaded: loaded (/lib/systemd/system/jenkins.service; enabled; vendor preset: enabled)
   Drop-In: /etc/systemd/system/jenkins.service.d
            └─override.conf
   Active: active (running) since Tue 2023-03-21 19:59:20 EET; 1min 42s ago
     Main PID: 41277 (java)
        Tasks: 60 (limit: 18243)
       Memory: 1.8G
          CPU: 1min 42.045s
      CGroup: /system.slice/jenkins.service
              └─41277 /usr/bin/java -Djava.awt.headless=true -jar
```

Перезапустити сервіс Jenkins (наприклад, після зміни конфігурації) можна командою:

```
sudo systemctl restart jenkins
```

Після успішного запуску Jenkins буде доступний через веб-браузер за адресою `http://localhost:8080` або `http://ваш_IP:8080`.

4. *Налаштування Jenkins*

Для налаштування служби зручно скористатись графічним інтерфейсом. Для цього потрібно відкрити веб оглядач та ввести адресу `http://<ваш_IP_адрес>:8080` для доступу до Jenkins. При першому вході користувач повинен ввести пароль, який знаходиться у файлі `/var/lib/jenkins/secrets/initialAdminPassword` (рис. 12.8). Для того, щоб отримати доступ до цього файлу, може бути потрібно зробити його доступним для запису, зчитування наступною командою:

sudo chmod 755 /var/lib/jenkins/secrets/initialAdminPassword



Рис. 12.8. Вікно для введення паролю під час першого входу у Jenkins

На наступному кроці слід обрати інсталяцію запропонованих плагінів (рис.12.9):

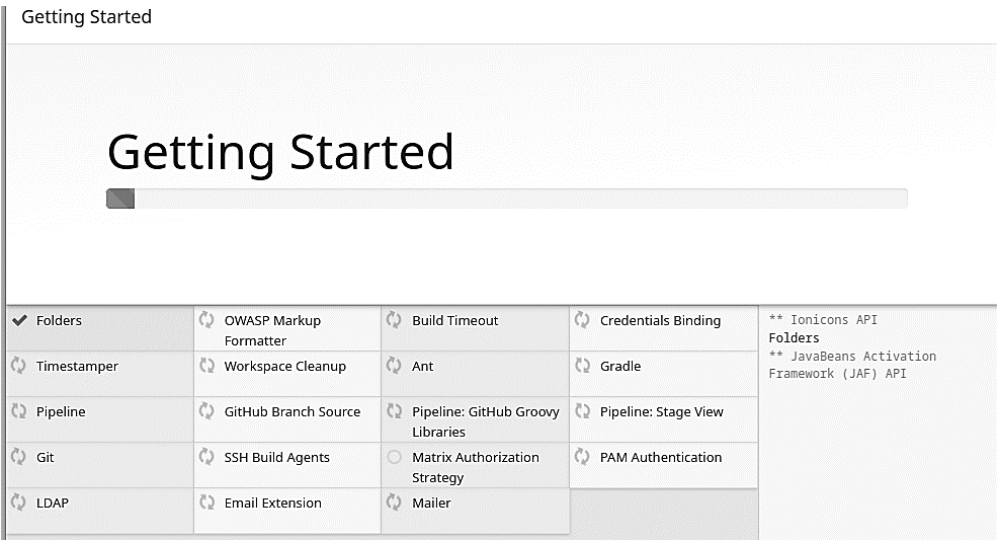


Рис. 12.8. Вікно вибору плагінів у Jenkins

Також, потрібно заповнити дані у вікні Create First Admin User та вказати Jenkins URL.

Jenkins URL:

The Jenkins URL is used to provide the root URL for absolute links to various Jenkins resources. That means this value is required for proper operation of many Jenkins features including email notifications, PR status updates, and the BUILD_URL environment variable provided to build steps.

The proposed default value shown is **not saved yet** and is generated from the current request, if possible. The best practice is to set this value to the URL that users are expected to use. This will avoid confusion when sharing or viewing links.

Not now

Save and Finish

5. Робота з пайплайном

Після проходження авторизації у Jenkins для роботи з пайплайном потрібно обрати опцію "New Item" на головній сторінці та виконати наступні кроки:

- ввести назву для нового pipeline та обрати тип "Pipeline";
- натиснути кнопку "ОК" для створення нового pipeline.
- на наступній сторінці перейти до розділу "Pipeline" та оберіть тип виконання "Pipeline script";
- у полі "Script" ввести код для pipeline. Це може бути скрипт на мові Groovy або відповідна конфігурація для обраного проекту.

В даному випадку в якості прикладу можна реалізувати скрипт на Groovy для завантаження обраного в якості прикладу проєкту «Vue Paper Dashboard» з Github та забезпечення процесу його компіляції і побудови:

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                git '<адреса відповідного репозиторію>'
            }
        }

        stage('Install dependencies') {
            steps {
                sh 'npm install'
            }
        }
    }
}
```

```
stage('Compile') {  
    steps {  
        sh 'npm run build'  
    }  
}  
}
```

Цей Jenkins Pipeline скрипт на Groovy описує автоматизований процес збірки проєкту і складається з трьох основних етапів (stages):

- 1. **stage('Checkout')** – на цьому етапі відбувається клонування коду з GitHub репозиторію за вказаною адресою. Це забезпечує отримання актуальної версії коду для подальшої роботи.
- 2. **stage('Install dependencies')** – етап встановлення залежностей проєкту за допомогою команди `npm install`. Ця команда читає файл `package.json` та завантажує всі необхідні пакети для роботи проєкту.
- 3. **stage('Compile')** – фінальний етап, де виконується збірка проєкту командою `npm run build`, яка компілює код та створює готовий для розгортання артефакт.

Директива `agent any` на початку скрипту вказує, що pipeline може виконуватися на будь-якому доступному агенті Jenkins. Опцію "Use Groovy Sandbox" потрібно увімкнути для забезпечення безпечного виконання скрипту в ізольованому середовищі.

Після того, як зміни внесені та збережені, створений новий pipeline потрібно запустити на виконання, натиснувши кнопку "Build Now" (рис. 12.9).

Stage View

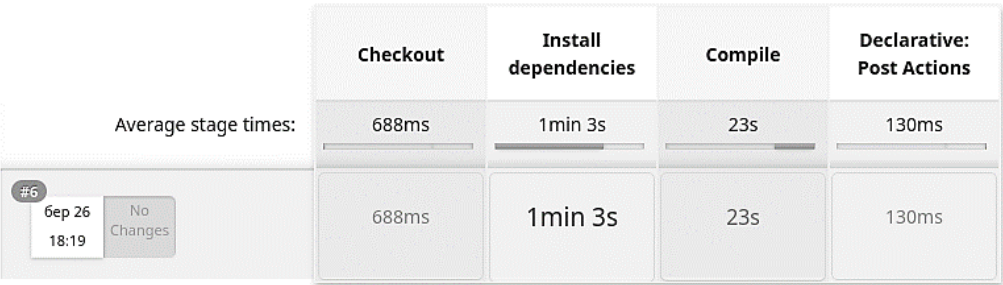


Рис. 12.9. Виконання пайплайну у Jenkins

Це базовий приклад створення pipeline в Jenkins. Зазвичай pipeline складається з кількох етапів, які виконуються послідовно, і включає в себе операції, такі як клонування репозиторію, збірка коду, тестування, розгортання та інші дії, які потрібні для роботи з проєктом. Після створення pipeline користувач може налаштувати його та додати нові етапи в процесі розробки, щоб забезпечити автоматизацію та ефективність проєкту.

Запропоновані завдання ознайомлюють з процесом встановлення та налаштування сервісу Jenkins і створення пайплайнів для автоматизації процесів збірки та розгортання проєкту, що забезпечує ефективне управління CI/CD-процесами, підвищує прозорість змін та дозволяє інтегрувати Jenkins із системами контролю версій, такими як GitHub чи GitLab.

12.5. Робота з інструментом GitHub Actions

GitHub Actions являє собою набір інструментів для автоматизації процесів розробки, які дозволяють збирати, тестувати та розгортати програмне забезпечення на платформі GitHub. За допомогою GitHub Actions можна створювати складні робочі процеси для виконання різноманітних дій на основі різних подій, таких як коміти до репозиторію, відкриття pull request, випуски програмного забезпечення та багато іншого.

Для ознайомлення з побудовою пайплайнів для автоматизації процесів спочатку потрібно створити папку `.github/workflows` у кореневій директорії свого репозиторію з обраним для роботи проєктом.

Наступним кроком є створення нового файлу з назвою `<ім'я файлу>.yaml` у папці `.github/workflows` (рис. 12.10).

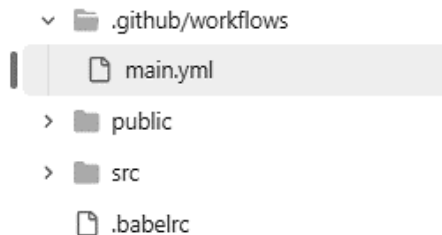


Рис. 12.10. Додавання yaml-файлу у репозиторій

У створеному файлі потрібно додати конфігурацію відповідного пайплайну на мові YAML. Конфігурація повинна включати список кроків, які необхідно виконати в пайплайні. Нижче наведено приклад простого сценарію:

name: My GitHub Actions Pipeline

on:

push:

branches: [main]

pull_request:

branches: [main]

jobs:

build:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v2

- name: Build code

run: <BUILD>

Розберемо детально кожен компонент цього GitHub Actions workflow:

- *name: My GitHub Actions Pipeline* – вказана назва workflow, яка відображатиметься в інтерфейсі GitHub Actions.
- *on:* секція, що визначає події, які запускають workflow. В даному прикладі вказано дві події:
 - push:*
 - branches: [main]* – workflow запускається при кожному push-коміті в гілку main;
 - pull_request:*
 - branches: [main]* – workflow також запускається при створенні або оновленні pull request до гілки main.
- *jobs:* – секція, що описує завдання для виконання. Розглянемо її вміст:
 - *build:* – назва завдання;
 - *runs-on: ubuntu-latest* – вказує, що завдання виконуватиметься на віртуальній машині з останньою версією Ubuntu
 - *steps:* – послідовність кроків для виконання. В даному випадку маємо два кроки:
 1. *name: Checkout code* – крок для отримання коду;
 - uses: actions/checkout@v2* – використовує офіційну GitHub Action для клонування репозиторію;
 2. *name: Build code* – крок для збірки коду;
 - run: <BUILD>* – місце для вставки реальної команди збірки (наприклад, `npm run build` для Node.js проєкту)

Цей сценарій є базовим шаблоном CI/CD pipeline, який можна налаштувати під конкретний проєкт, замінивши <BUILD> на відповідні команди збірки (потребує конкретизації команди збірки відповідно до проєкту).

Так, наприклад, на рис.12.11 наведено дещо модифікований пайплайн для Node.js проєкту, який запускається тільки при push у гілку та додатково вказує версію Node.js (16.x) і містить налаштування залежностей (адаптований для тестового проєкту «Vue Paper Dashboard»).

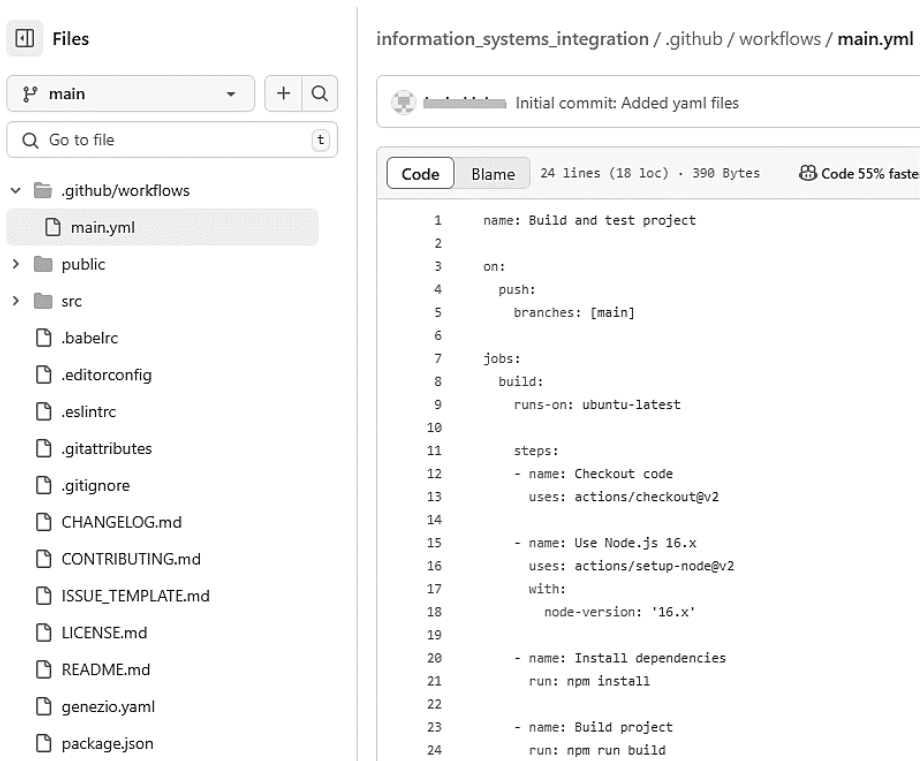


Рис. 12.11. Вміст yml-файлу у репозиторії

Після додавання відповідного коду потрібно зберегти створений файл.

Для перевірки роботи створеного пайплайну можна локально внести деякі зміни в проєкт та запустити їх до даного репозиторію на GitHub. Далі потрібно перейти до вкладки "Actions" та перевірити статус.

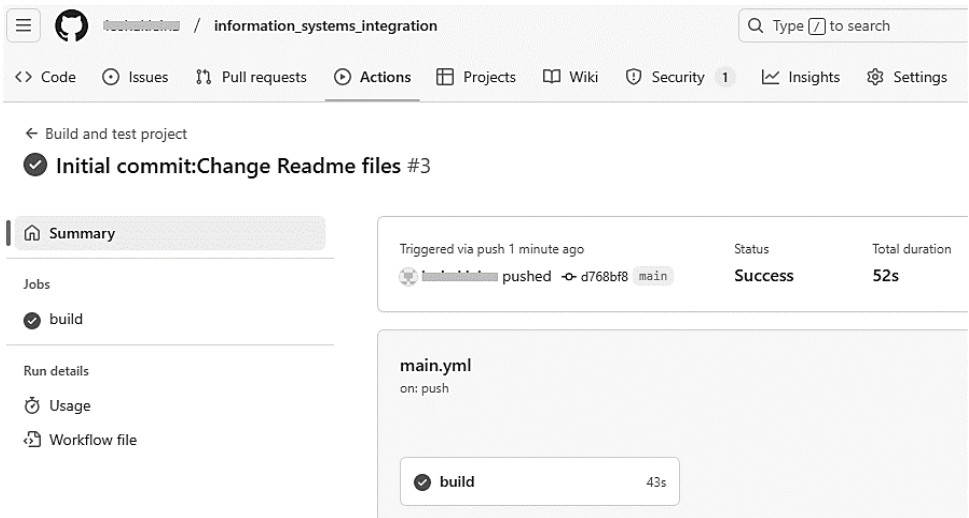


Рис. 12.11. Успішне виконання пайплайну у GitHub Actions

У випадку виникнення проблем при побудові проєкта чи для більш детального аналізу процесу побудови потрібно обрати відповідний запуск і відкрити детальний журнал. Він містить перелік всіх процесів із деталізацією кроку, при виконанні якого виникла помилка(рис. 12.12). Якщо така ситуація виникла – потрібно провести аналіз логів та визначити основну причину збою, як-от синтаксична помилка у файлі робочого процесу, невдалий тест або проблема зі сценарієм збірки чи розгортання. Після вирішення проблеми та внесення змін в код потрібно зафіксувати зміни, щоб запустити новий запуск та перевірити його статус.

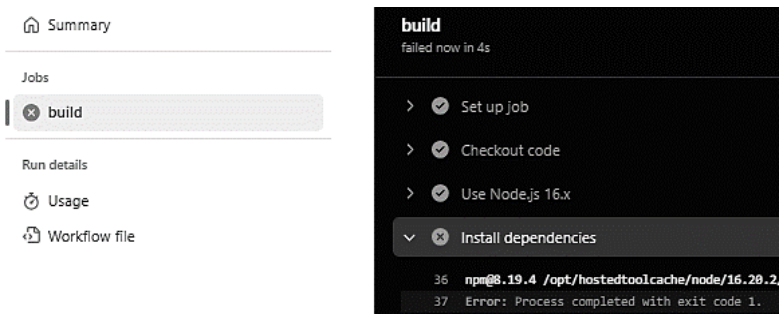


Рис. 12.12 (а). Перегляд процесів з метою вирішення проблеми

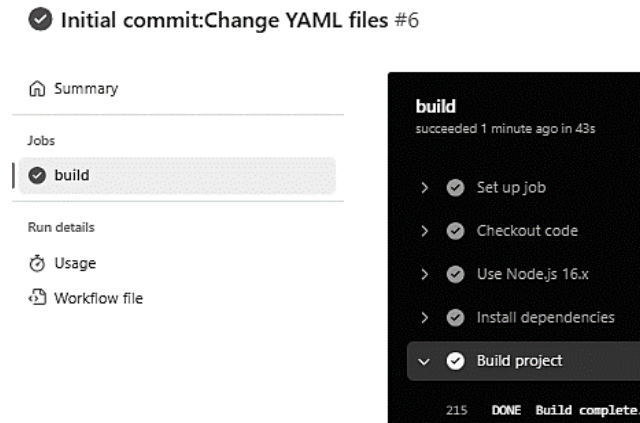


Рис. 12.12 (б). Успішне побудова проєкту після виправлення

Отже, GitHub Actions дозволяє автоматизувати процеси розробки, такі як збірка, тестування та розгортання програмного забезпечення. Налаштування пайплайну передбачає створення `.github/workflows/<ім'я файлу>.yaml`, де визначаються тригери, середовище виконання та послідовність кроків. Після збереження конфігурації можна перевірити її роботу через вкладку "Actions", що забезпечує ефективний контроль та автоматизацію процесів розробки.

12.6. Робота з інструментом SonarQube

SonarQube надає інструменти для статичного аналізу коду, оцінки ризиків безпеки та покриття коду тестами й надає корисну інформацію про стан проєкту, таку як покриття тестами, показники якості коду, виявлені помилки, проблеми з безпекою, дублювання коду тощо. Це дозволяє командам розробників зосередитися на покращенні якості коду і зниженні технічного боргу, що, в свою чергу, підвищує стабільність, безпеку і ефективність програмного забезпечення.

Розглянемо налаштування та практичне застосування вказаного інструменту аналізу кодової бази. Всі зазначені нижче процеси були протестовані на Oracle VirtualBox, Ubuntu 20.04, 2GB RAM, 10GB HDD. Для роботи на інших ОС потрібно звернутись до офіційної документації: <https://docs.sonarqube.org/9.8/setup-and-upgrade/install-the-server/>.

1. Встановлення інструменту SonarQube

Перед встановленням SonarQube потрібно встановити Java 11 з репозиторію Ubuntu. Для цього в терміналі потрібно виконати команду:

```
sudo apt install default-jdk
```

```

[username@hostname ~]$ sudo apt install default-jdk
[sudo] password for username:
Зчитування переліків пакунків... Виконано
Побудова дерева залежностей... Виконано
Зчитування інформації про стан... Виконано
default-jdk is already the newest version (2:1.11-72build2).
оновлено 0, встановлено 0 нових, 0 відмічено для видалення і 19 не оновлено.

```

Далі потрібно завантажити останню версію SonarQube з офіційного сайту:

```
wget https://binaries.sonarsource.com/Distribution/sonarqube/sonarqube-9.1.0.47736.zip
```

та розпакувати завантажений архів:

```
sudo unzip sonarqube-9.1.0.47736.zip -d /opt/
```

У випадку, якщо утиліти `unzip` немає на ОС, можна встановити її, використовуючи пакетний менеджер `apt`.

В якості складового сервісу SonarQube у фоновому режимі запускає ElasticSearch, який у свою чергу заборонено запускати від імені користувача `root`. З огляду на це потрібно встановити права на директорію `cd/opt/` своєму користувачу:

```
sudo chown -R <username>:<username> sonarqube-9.1.0.47736/
```

Після успішного виконання попередніх команд можна переглянути команди (параметри), що доступні SonarQube:

```
/opt/sonarqube-9.1.0.47736/bin/linux-x86-64/sonar.sh
```

Після цього можна запустити SonarQube:

```
/opt/sonarqube-9.1.0.47736/bin/linux-x86-64/sonar.sh start
```

та перевірити його статус.

```

[username@hostname ~]$ /opt$ /opt/sonarqube-9.1.0.47736/bin/linux-x86-64/sonar.sh start
Starting SonarQube...
Started SonarQube.
[username@hostname ~]$ /opt$ /opt/sonarqube-9.1.0.47736/bin/linux-x86-64/sonar.sh status
SonarQube is running (16720).

```

Якщо при запуску виникають проблеми, можна скористатись командою `console`, яка призначена для аналізу помилок.

2. Ознайомлення з SonarQube WebUI

Наступним кроком після успішного встановлення є відкриття веб-інтерфейсу SonarQube в браузері за адресою '`http://<your-server-IP>:9000`'.

При першому вході потрібно авторизуватись як адміністратор, використовуючи користувацьке ім'я та пароль по замовчуванню (рис. 12.13)

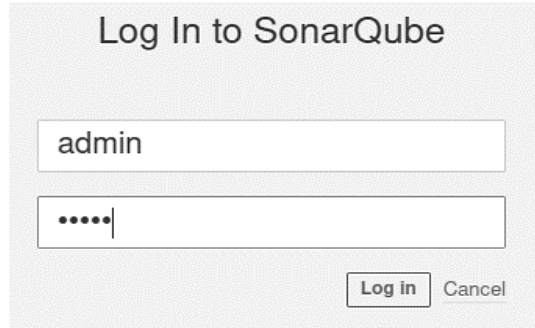


Рис. 12.13. Авторизація в ролі адміністратора

Після входу у SonarQube WebUI на вкладці адміністратор потрібно додати користувача (рис. 12.14).

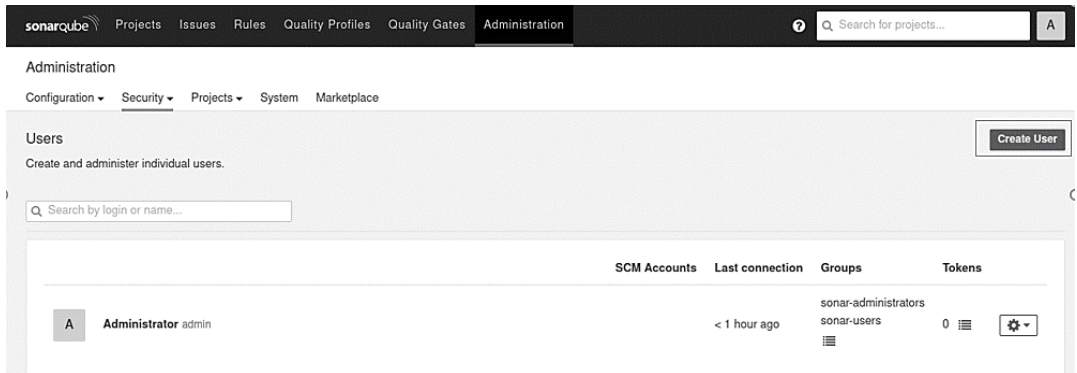


Рис. 12.14. Вибір пункту для додавання користувача

3. Підключення PostgreSQL до SonarQube

За замовчуванням SonarQube використовує вбудовану базу даних для опрацювання усіх значень. В якості базу даних для зберігання результатів аналізу коду та метрик якості зручно використовувати реляційну базу даних від PostgreSQL.

Одним з головних факторів, які зумовлюють використання Postgresql в якості бази даних для SonarQube є те, що вона дозволяє зберігати великі обсяги даних. Це корисно для збереження результатів аналізу коду та метрик якості для великих проектів. Також Postgresql має вбудовані можливості для оптимізації запитів, що дозволяє швидко виконувати запити до бази даних та отримувати швидкі результати.

Перед встановленням СКБД PostgreSQL потрібно встановити плагін:

```
sudo apt install postgresql postgresql-contrib
```

Після цього потрібно створити користувача для SonarQube в базі даних:

```
sudo -u postgres psql -c "CREATE USER sonar WITH ENCRYPTED  
PASSWORD 'password';"
```

Наступним кроком є створення нової бази даних в PostgreSQL:

```
sudo -u postgres createdb -O sonar sonar
```

Потрібно налаштувати відповідні параметри підключення до бази даних у файлі `/opt/sonarqube-9.1.0.47736/conf/sonar.properties`. Для цього потрібно знайти у файлі блок PostgreSQL, розкоментувати стрічки вказані нижче та додати ті, що відсутні:

```
sonar.jdbc.username=sonar  
sonar.jdbc.password=password  
sonar.jdbc.url=jdbc:postgresql://localhost/sonar
```

В даному випадку розглядається ситуація, за якої SonarQube встановлений на туж віртуальну машину, що й PostgreSQL. Якщо СКБД встановлюється на окремий хост, замість «localhost» має бути вказана адреса цього хосту.

Після зміни налаштувань потрібно запустити SonarQube:

```
sudo /opt/sonarqube-9.1.0.47736/bin/linux-x86-64/sonar.sh start
```

або перезапустити, у випадку, якщо він уже був запущений, для того, щоб застосувати нові конфігурації.

Якщо SonarQube не запускається, потрібно запустити його параметром *console*, що дає змогу переглянути процес запуску і зрозуміти на якому саме етапі він зупиняється. Якщо біля помилки фігурує `vm.max_map_count`, це свідчить про те, що для віртуальних Java середовищ не вистачає виділених комірок оперативної пам'яті і це число потрібно збільшити. Це можна виконати наступною командою:

```
sudo sysctl -w vm.max_map_count=262144
```

SonarQube буде доступний для використання з базою даних PostgreSQL на порту 9000. Варто зауважити, що при використанні СКБД усі дані наявні в SonarQube будуть оновлені. З огляду на це облікові дані будуть такими ж, як і при першому встановленні SonarQube.

4. Налаштування *Quality Gate*

SonarQube перевіряє якість коду на основі правил (Rules), що охоплюють покриття тестами, безпеку, дублювання коду, технічний борг і Code Smells (незручності у коді, які можуть спричинити майбутні помилки). Кожне правило має встановлене порогове значення, яке показує, коли правило не виконується. Всі встановлені правила можна переглянути на вкладці *Quality Gates* (рис. 12.15).

Quality Gates можна налаштовувати для конкретного проекту або групи проектів. Якщо якість коду не відповідає встановленим вимогам, SonarQube може автоматично відхилити зміни коду або відправити повідомлення про проблеми розробнику для вирішення проблеми.

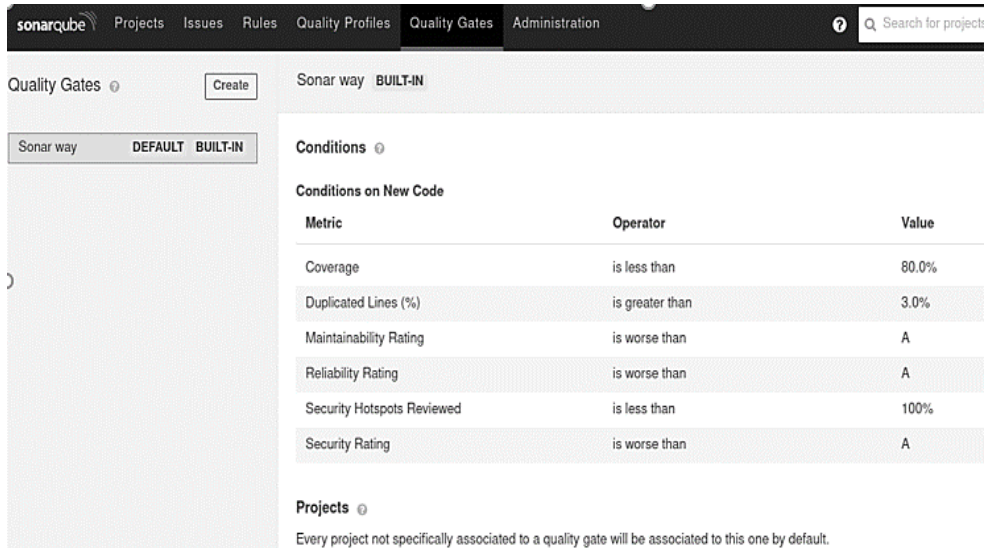


Рис. 12.15. Основні метрики та їх значення

Для створення правил аналізу коду потрібно вибрати кнопку «Create» на вкладці Quality Gates у SonarQube WebUI. Створеному правилу слід дати назву та додати декілька умов, що мають мати логічне пояснення. Наприклад, проект повинен мати не більше 6 критичних помилок, оскільки він має 6 000 рядків, і 6 критичних помилок відповідає 0,1% для такого проекту (рис.12.16).

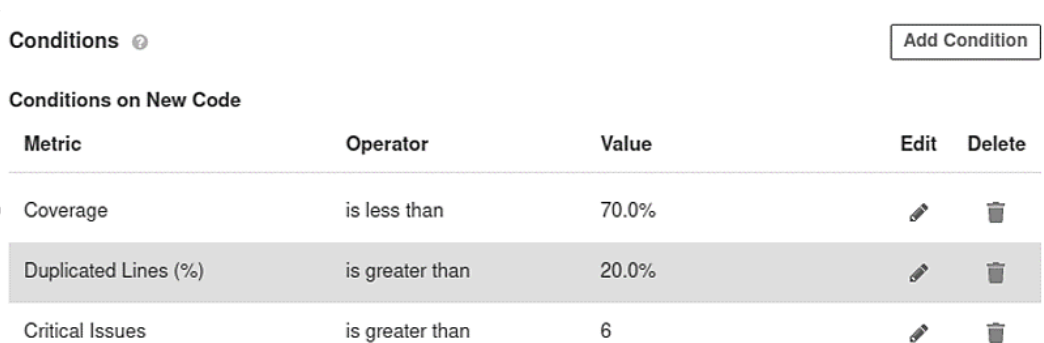


Рис. 12.16. Правила для створеного Quality Gates.

Також, наведені правила говорять, що кількість дубльованих рядків не можете бути більшим за 20%, а покриття проекту тестами має бути не менше, як 70%, щоб зменшити можливу кількість помилок та забезпечити відповідну якість.

5. Встановлення та налаштування Sonar-Scanner для збору та аналізу даних коду

Перед проведенням аналізу коду потрібно зклонувати git-репозиторій обраного для роботи тестового проекту, код якого і буде проаналізований:

git clone <url репозиторію>

Після цього потрібно зайти у SonarQube WebUI та створити новий проект на вкладці “Projects”, обравши при створенні опцію «Manually».

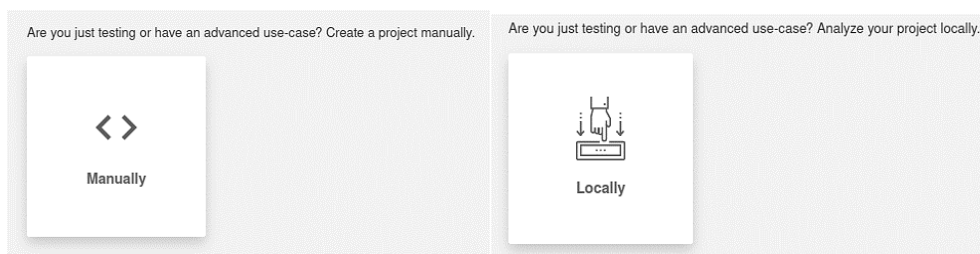


Рис. 12.17. Вибір типу проекту у SonarQube WebUI.

Назву проекту потрібно обрати на свій розсуд (вона має бути інформативною). В якості типу проекту потрібно обрати “Locally” та згенерувати токен у вікні, що відкриється (згенерований токен потрібно зберегти). Далі слід, дотримуючись вказівок, обрати з запропонованих всі деталі, які найбільше підходять до обраного тестового проекту (рис. 12.18).

1 Provide a token

vue-paper-dashboard:4f910bd977e973484a239134066d1b8b7d63db41

2 Run analysis on your project

What option best describes your build?

Maven

Gradle

.NET

Other (for JS, TS, Go, Python, PHP, ...)

What is your OS?

Linux

Windows

macOS

Download and unzip the Scanner for Linux

Visit the [official documentation of the Scanner](#) to download the latest version, and add the `bin` directory to the `PATH` environment variable

Execute the Scanner

Running a SonarQube analysis is straightforward. You just need to execute the following commands in your project's folder.

```
sonar-scanner \
-Dsonar.projectKey=vue-paper-dashboard \
-Dsonar.sources=. \
-Dsonar.host.url=http://127.0.0.1:9000 \
-Dsonar.login=4f910bd977e973484a239134066d1b8b7d63db41
```

Copy

Please visit the [official documentation of the Scanner](#) for more details.

Is my analysis done? If your analysis is successful, this page will automatically refresh in a few moments.

You can set up Pull Request Decoration under the project settings. To set up analysis with your favorite CI tool, see the [tutorials](#).

Check these useful links while you wait: [Branch Analysis](#), [Pull Request Analysis](#).

Рис. 12.18. Вибір налаштувань відповідно до тестового проєкту та ОС

Наприклад, для того щоб завантажити Sonar-Scanner на Ubuntu, потрібно перейти по відповідному посиланню та скопіювати посилання на “Linux 64-bit” і виконати в терміналі команду `wget <посилання>` та після її виконання розархівувати завантажений архів `unzip sonar-scanner-XX.zip`.

Так, наприклад, для обраного в якості прикладу проєкту «Vue Paper Dashboard» та вказаної вище ОС команди можуть мати вигляд:

```
sudo wget https://binaries.sonarsource.com/Distribution/sonar-scanner-
cli/sonar-scanner-cli-4.8.0.2856-linux.zip
sudo unzip sonar-scanner-cli-4.8.0.2856-linux.zip
```

```
~$ sudo unzip sonar-scanner-cli-4.8.0.2856-linux.zip
Archive: sonar-scanner-cli-4.8.0.2856-linux.zip
creating: sonar-scanner-4.8.0.2856-linux/
creating: sonar-scanner-4.8.0.2856-linux/jre/
creating: sonar-scanner-4.8.0.2856-linux/jre/conf/
creating: sonar-scanner-4.8.0.2856-linux/jre/conf/management/
```

Для того щоб запустити завантажений сканер, потрібно запустити виконуваний файл, який лежить в розархівованій директорії `sonar-scanner` в папці `bin`.

Якщо після проходження сканування буде повідомлення «EXECUTION SUCCESS» – перевірка пройшла успішно і можна ознайомитись з результатами у WebUI

Для ознайомлення з результатами та кращого розуміння функціоналу інструменту SonarQube потрібно переглянути та проаналізувати всі вкладки з результатами:

- «Overview» (загальний огляд стану проекту, включаючи основні метрики, статус аналізу коду та загальні показники якості),
- «Issues» (список знайдених проблем у коді, таких як баги, вразливості та інші порушення стандартів якості),
- «Security Hotspots» (місця в коді, які можуть містити потенційні загрози безпеці та потребують додаткової перевірки),
- «Measures» (різні показники якості коду, такі як покриття тестами, кількість рядків коду, кількість помилок та інші метрики),
- «Code» (огляд коду проекту з можливістю перегляду файлів, аналізу їхнього вмісту та виявлення проблем),
- «Activity» (історія змін у проекті, включаючи виконані аналізи, виправлення помилок і внесені зміни).

Після аналізу результатів можна знову запустити сканування знову, щоб перевірити, чи обраний проект пройшов Quality Gate в Overall Code вкладці (рис. 12.19).

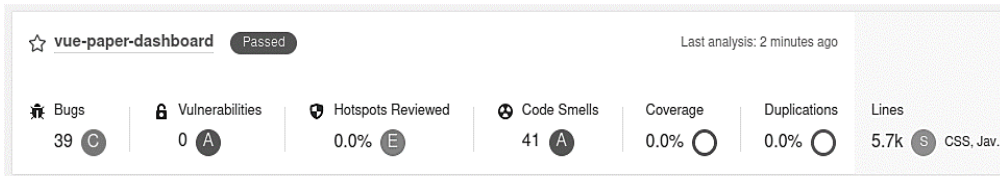


Рис. 12.19. Результат перевірки Quality Gate в Overall Code вкладці.

Розглянуті процеси ознайомлюють з особливостями встановлення, та практичного застосування інструменту SonarQube. Таким чином, можна зробити висновок, що для проведення аналізу коду проекту засобами SonarQube потрібно виконати наступні кроки:

- налаштувати проект в SonarQube: створити проект та налаштувати параметри, такі як мова програмування, система збірки та інші;
- встановити та налаштувати Sonar-Scanner для збору та аналізу даних коду: завантажити та встановити sonar-scanner та налаштувати файл конфігурації sonar-project.properties для вказання параметрів аналізу;
- запустити sonar-scanner для аналізу коду проекту та збору даних;

- отримати звіти та результати аналізу в SonarQube (кількість помилок, вразливостей та дублювання коду);
- налаштувати Quality Gates для забезпечення якості коду.

Після встановлення Quality Gates, SonarQube буде автоматично перевіряти код на відповідність цим стандартам та повідомляти про будь-які помилки або вразливості.

12.7. Робота з агентом у Jenkins

Агенти Jenkins – це процеси, які служать для виконання робочих завдань на віддаленому вузлі та забезпечують масштабованість та розподілення завдань між різними вузлами. Загалом, використання агентів в Jenkins дозволяє забезпечити розподілення завдань між різними вузлами, зменшити час виконання та підвищити ефективність процесу збірки та тестування.

Агенти можуть бути запущені як на фізичних машинах, так і на віртуальних машинах, контейнерах; на різних операційних системах та платформах, що дає можливість запускати тестування та збірку проекту в різних середовищах.

Використання агентів в Jenkins дозволяє забезпечити розподілення завдань між різними вузлами, зменшити час виконання та підвищити ефективність процесу збірки та тестування.

Розглянемо процес створення мультивузлової архітектури Jenkins з використанням агентів приєднаних по SSH. Наведені нижче кроки були відтворені на Oracle VirtualBox, Ubuntu 20.04, 2GB RAM, 10GB HDD з заздалегідь встановленим Jenkins та новоствореною віртуальною машиною від AWS з відповідними параметрами.

1. *Перевірка плагінів та налаштувань*

Перед початком роботи потрібно перейти на веб інтерфейс інструменту Jenkins за посиланням «http://<ваша_ip_адреса>:8080» та переконатись, що плагін для приєднання SSH агентів встановлений.

Це можна перевірити наступним чином: в меню “Manage Jenkins” слід обрати пункт “Plugin Manager” та підпункт “Installed Plugins” і в пошуковому рядку набрати в якості ключа для пошуку «SSH». Якщо в списку що з’явиться буде елемент «SSH Build Agents plugin», то необхідний для роботи плагін вже встановлений та включений (рис. 12.20).

Plugins

Name ↓	Enabled
Mina SSHD API :: Common 2.9.2-62.v199162f0a_2f8 Provides the Common module of Apache Mina SSHD to plugins.	☒
Mina SSHD API :: Core 2.9.2-62.v199162f0a_2f8 Provides the Core module of Apache Mina SSHD to plugins.	☒
SSH Build Agents plugin 2.877.v365f5eb_a_b_eec Allows to launch agents over SSH, using a Java implementation of the SSH protocol. Report an issue with this plugin	☒
SSH Credentials Plugin 305.v8f4381501156 Allows storage of SSH credentials in Jenkins Report an issue with this plugin	☒
SSH server 3.275.v9e17c10f2571 Adds SSH server functionality to Jenkins, exposing CLI commands through it. Report an issue with this plugin	☒

Рис. 12.20. Перевірка наявності плагіна приєднання SSH агентів.

Якщо відповідний плагін відсутній, його потрібно встановити за наступним алгоритмом: в меню “Manage Jenkins” обрати пункт “Plugin Manager” та перейти до підпункту “Available Plugins”, знайти вказаний плагін, відзначити його, та обрати кнопку “Install without restart”.

Наступним кроком є перевірка, чи налаштований зв’язок по SSH між машиною де встановлений Jenkins (master) та іншою машиною яка буде агентом (slave). Для цього можна вузла master зайти по SSH на термінал вузла slave використовуючи команду `ssh username@ip-address`.

В якості вузла slave можна використати локальну віртуальну машину або скористатись для створення машини відповідним сервісом хмарних платформ, таких як AWS, GSP чи Azur.

Розглянемо для прикладу використання віртуальної машини AWS. Для створення машини потрібно авторизуватись на AWS та перейти у вкладку EC2 створити машину та налаштувати її. При створенні нового інстансу рекомендується обрати тип t2.micro, який доступний у рамках безкоштовного рівня AWS Free Tier. Цей тип інстансу має 1 vCPU та 1 ГБ оперативної пам’яті, що є достатнім для базового тестування та навчальних цілей.

Після вибору типу інстансу потрібно налаштувати параметри безпеки, створити або обрати існуючу пару ключів для SSH-доступу при створенні машини (налаштувати inbound rules, щоб вони працювали з SSH з автоматичним визначенням IP головної машини) та запустити машину (рис. 12.21). При налаштуванні групи безпеки важливо відкрити порти, необхідні для комунікації з master-вузлом.

Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public IPv4
-	i-0c25237075d7e1a3a	Running	t2.micro	2/2 checks passed	No alarms	eu-west-2b	ec2-13-42-54-7.eu-wes...	13.42.54.7

Рис. 12.21. Запущений інстанс у AWS.

Після запуску створеної машини потрібно перевірити зв'язок зі створеною slave-машиною – зайти в термінал master-машини (на якій встановлено Jenkins) та виконати команду наступного формату:

```
ssh -i <path-to-private-key> <user>@<public-ipv4-address-or-public-dns-name>
```

Перед виконанням потрібно замінити <path-to-private-key> на шлях файлу до закритого ключа, який було вказано під час запуску, та замінити <user> на ім'я користувача для екземпляра (зазвичай «ec2-user» для Amazon Linux або «ubuntu» для Ubuntu). Також потрібно замінити <public-ipv4-address-or-public-dns-name> загальнодоступною адресою IPv4 або загальнодоступним DNS-ім'ям екземпляра.

```
~$ ssh -i /home/slave-machine.pem ec2-user@3.9.29.22
#_
Amazon Linux 2023
https://aws.amazon.com/linux/amazon-linux-2023
[ec2-user@ip-172-31-32-135 ~]$
```

2. Приєднання агента

Для того щоб Jenkins міг приєднатись до агента за допомогою SSH, потрібно вказати облікові дані цього агента (ім'я користувача та пароль). Для того, щоб налаштувати таку опцію, потрібно у вебінтерфейсі перейти у меню “Manage Jenkins”, обрати пункт “Manage Credentials” та перейти до пункту “System” і у підпункті “Global credentials” обрати “Add credentials” та внести відповідні налаштування. Слід обрати тип “SSH Username with private key”, заповнити відповідні поля та зберегти запис (рис.12.22).

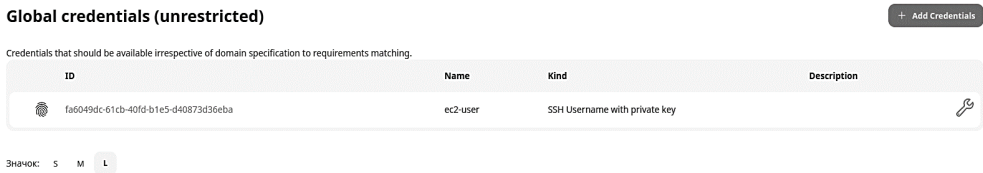


Рис. 12.22. Налаштовані Global credentials у Jenkins.

Для безпосереднього приєднання агента потрібно перейти в меню до пункту “Manage Nodes and Clouds” та обрати опцію “New node”. Далі слід обрати ім’я для агента, позначити властивість “Permanent agent” та натиснути кнопку “Create”.

Після цього потрібно:

- додати довільний опис, в полі Remote root directory, вказати директорію на віддаленій машині, яка буде використовуватись для операцій Jenkins. Наприклад це може бути домашня директорія (/home/<username>).
- в полі “Launch method” обрати варіант “Launch agents via SSH”. Вказати адресу slave сервера, та обрати обліковий запис який було створено вище;
- в полі “Host Key Verification Strategy” обрати варіант “Non verifying Verification Strategy”. Це поле визначає політику щодо так званого відбитку пальця SSH Unix систем. Якщо залишити перевірку, то доведеться додавати цей відбиток у /var/lib/Jenkins/.ssh/known_hosts, якщо ж використовувати “Non verifying Verification Strategy” то цього робити не потрібно (рис.12.23).
- зберегти налаштування.

Launch method ?

Launch agents via SSH

Host ?

18.130.67.179

Credentials ?

ec2-user

Add

Host Key Verification Strategy ?

Non verifying Verification Strategy

Додаткові

Рис. 12.23. Заповнені поля для приєднання агента у Jenkins

Після успішного виконання вказаних вище кроків потрібно перейти на вкладку “Nodes” у верхньому меню та впевнитися, що агент приєднано (рис. 12.24). Варто відмітити, що на slave-машині (в розглядуваному випадку – на віддаленому інстансі AWS) має бути встановлена Java/

Manage nodes and clouds + New Node

S	Id	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	Built-In Node	Linux (amd64)	In sync	7,91 GB	0 B	7,91 GB	0ms
	SSH-slave	Linux (amd64)	In sync	6,17 GB	0 B	476,49 MB	994ms
Data obtained		15 sec	15 sec	14 sec	14 sec	14 sec	14 sec

Рис. 12.24. Відображення приєднаного агента на вкладці “Nodes”

3. Використання агента

Спочатку потрібно перейти на вебінтерфейс інструменту Jenkins за посиланням «http://<ваша_ip_адреса>:8080», створити нову роботу, обравши опцію “New Item” та обрати для неї тип “Pipeline” (рис. 12.25).

Enter an item name

VirtualAgent

» Required field

Freestyle project
Це - основна відмінність Jenkins. Jenkins побудує ваш проект, комбінуючи будь-які системи управління кодом з будь-якими систем відмінних від збірки програмного забезпечення.

Pipeline
Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) as a style job type.

Рис. 12.25. Вибір типу для роботи у Jenkins

Після цього потрібно описати Pipeline, який використовуватиме команди локально та на налаштованому віддаленому агенті.

Оскільки в прикладі використовуватиметься команда “sudo” користувачу, під яким доєднаний агент, потрібно надати право використовувати “sudo” без паролю. Для цього на віртуальній машині агента у файлі /etc/sudoers потрібно додати рядок:

<ваш username> ALL=(ALL) NOPASSWD:ALL

```
GNU nano 5.8
ec2-user ALL=(ALL) NOPASSWD:ALL
## Sudoers allows particular users to run various commands as
## the root user, without needing the root password.
##
```

Надання прав `sudo` без пароля є потенційним ризиком безпеки. У продакшн-середовищі рекомендується налаштовувати окремі обмежені права через `sudoers` або використовувати безпечні методи передачі команд, такі як Ansible.

Нижче наведено можливий вміст Pipeline для використання створеного агента:

```
pipeline {
  agent any

  stages {
    stage('Testing local connection') {
      steps {
        sh "echo Hello from \$(hostname -I)!"
      }
    }
    stage('Testing remote connection') {
      steps {
        node("Slave"){
          sh "echo Hello from \$(hostname -I)!"
        }
      }
    }
    stage('Getting project') {
      steps {
        node("Slave"){
          git '<адреса репозиторію з проектом «Vue Paper Dashboard»>'
        }
      }
    }

    stage('Install dependencies') {
      steps {
        node("Slave"){
          sh 'sudo npm install'
        }
      }
    }

    stage('Compile') {
      steps {

```

```

node("Slave"){
  sh 'sudo npm run build'
}
}
}
}
}
}
}

```

Розглянемо детально всі зазначені етапи:

- *Testing local connection*: перевіряє з'єднання з локальним агентом Jenkins, виводячи його IP-адресу;
- *Testing remote connection*: перевіряє з'єднання з віддаленим агентом (Slave), виводячи його IP-адресу. Використовує директиву `node("Slave")` для виконання команд на конкретному агенті;
- *Getting project*: клонує вказаний Git-репозиторій на віддалений агент. Важливо переконатись, що на агенті встановлений Git;
- *Install dependencies*: встановлює залежності Node.js проєкту. В даному випадку передбачається, що на агенті вже встановлений Node.js та npm;
- *Compile*: виконує збірку проєкту за допомогою `npm script 'build'`.

В розглянутому прикладі агенти виглядають наступним чином

Manage nodes and clouds

+ New Node

S	Ім'я ↓	Architecture	Clock Difference	Free Disk Space	Free Swap Space	Free Temp Space	Response Time
	Built-in Node	Linux (amd64)	In sync	9,77 GB	8,00 GB	9,77 GB	0ms
	Slave	Linux (amd64)	In sync	3,23 GB	1,93 GB	2,72 GB	837ms
Data obtained		12 min	12 min	12 min	12 min	12 min	12 min

Для перевірки коректності роботи побудованої мультивузлової архітектури Jenkins з використанням агенту приєднаного по SSH слід переглянути інформацію, виведену в “Console output” пункту меню “Build”

```

DONE Build complete. The dist directory is ready to be deployed.
INFO Check out deployment instructions at https://cli.vuejs.org/guide/deployment.html

```

```

[Pipeline] }
[Pipeline] // node
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS

```

Таким чином, запропоновані завдання охоплюють створення та налаштування мультивузлової архітектури Jenkins з використанням SSH-агентів. Використання агентів в Jenkins є ефективним рішенням для організації розподіленої системи безперервної інтеграції та розгортання (CI/CD), яке дозволяє оптимізувати ресурси та підвищити продуктивність процесів розробки програмного забезпечення.

Важливо відзначити, що при налаштуванні подібної архітектури особливу увагу слід приділяти питанням безпеки, зокрема правильному налаштуванню прав доступу та політик безпеки SSH-з'єднання.

12.8. Використання інтеграцій SonarQube у Jenkins

Jenkins та SonarQube є популярними інструментами для розробки програмного забезпечення, які можна використовувати разом для автоматичної перевірки якості коду. Jenkins буде відповідати за автоматизацію процесу компіляції, тестування та розгортання програмного забезпечення, а SonarQube – за аналіз якості коду та знаходження потенційних проблем. Така інтеграція дозволить розробникам зменшити кількість помилок, вдосконалити процес розробки, зменшити витрати часу та зусиль на ручну перевірку коду та збільшити швидкість розгортання програмного забезпечення відповідно.

Наприклад, потрібно оптимізувати зазначені процеси під час розробки вебдодатку. Для цього можна встановити SonarQube на свій сервер та встановити відповідно SonarQube Scanner плагін на свій Jenkins-сервер. Після цього потрібно створити новий Jenkins Job для вебдодатку та налаштувати його для запуску SonarQube Scanner, що Jenkins автоматично запускати SonarQube Scanner при кожному запуску додатку та надавати результати аналізу якості коду в SonarQube.

Запропоновані нижче практичні завдання мають на меті ознайомлення з використанням спільної роботи інструментів GitHub, Jenkins, SonarQube та PostgreSQL, вироблення навичок прописувати інтеграцію SonarQube в Jenkins, GitHub в pipeline job та аналізувати результат роботи налаштованої системи.

Бажаний CI процес має наступні етапи (рис. 12.26):

- Внесення зміни в репозиторій проекту на GitHub;
- Jenkins шляхом опитування виявляє зміни, та автоматично тригерить конфігурацію;
- вивантаження проекту зі змінами на Jenkins сервер;
- Jenkins проводить сканування проекту та відсилає його на SonarQube;
- SonarQube дає відповідь, чи було успішно збережено дані в PostgreSQL та результат проходження Quality Gate;

- Jenkins проводить процес побудови незалежно від результату Quality Gate.

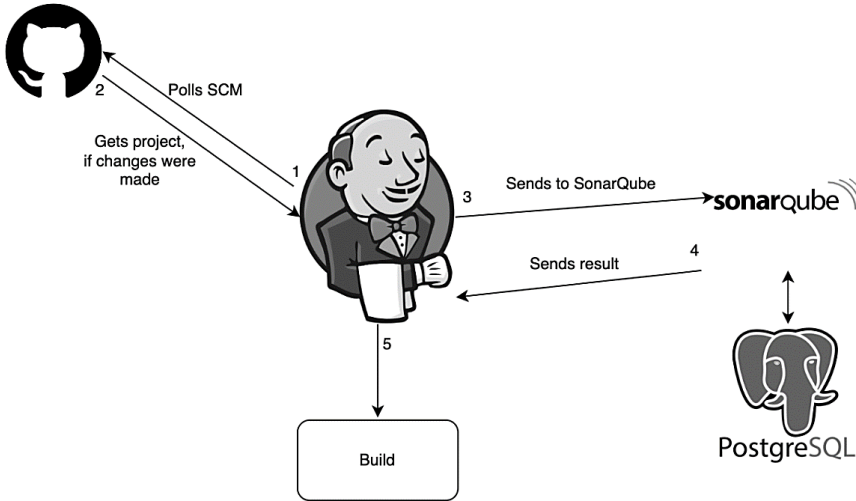


Рис. 12.26. Вибір типу для роботи у Jenkins

Описані нижче завдання орієнтовані на машину з ОС Ubuntu 20.04 із заздалегідь встановленим Jenkins та машиною з такими ж характеристиками із заздалегідь встановленими SonarQube та PostgreSQL.

1. **Налаштування GitHub trigger**

Для налаштування потрібно спочатку перейти на вебінтерфейс інструменту Jenkins за посиланням «http://<ваша_ip_адреса>:8080», створити окремий Pipeline та скопіювати в нього код Pipeline з попереднього розділу 12.7.

В блоці “General” потрібно поставити галочку біля опції “GitHub project” та вписати адресу Github-репозиторію з обраним для роботи проектом у відповідне текстове поле. Далі слід перейти до блоку “Build Triggers” та обрати тригер – в даному випадку використовується “Poll SCM”, проте можна обрати й інші за бажанням.

Для варіанту “Poll SCM” потрібно прописати періодичність, з якою Jenkins буде перевіряти репозиторій на предмет змін. Періодичність записується у форматі Cron (див. розділ 11) – наприклад, раз в 20 хвилин (рис. 12.27).

☒ Poll SCM ?

Schedule ?

H/20 ****

Would last have run at понеділок, 1 травня 2023 р. о 10:06:14 за східноєвропейським літнім часом.
(show details)

☐ Ignore post-commit hooks ?

☐ Період тишини ?

Рис. 12.27. Налаштування перевірки змін у репозиторії кожні 20 хв.

2. Встановлення та налаштування SonarScanner

Для подальшої роботи потрібно перейти в меню “Manage Jenkins”, обрати підпункт “Plugin Manager” та підпункт “Available Plugins” відповідно і в пошуку ввести “sonar”. Вибрати плагін “SonarQube Scanner”, почати встановлення плагіна і під час процесу встановлення відзначити галочкою пункт “Restart Jenkins when installation is complete...” (рис. 12.28).

Download progress

Підготовка

- Checking internet connectivity
- Checking update center connectivity
- Success

SonarQube Scanner (●●●) Очікування

Sonargraph Integration (●●●) Очікування

Перезапуск Jenkins (●●●) Очікування

→ [Повернутись до початку сторінки](#)
(you can start using the installed plugins right away)

→ ☒ Restart Jenkins when installation is complete and no jobs are running

Рис. 12.28. Встановлення плагіну SonarQube Scanner

Після встановлення плагіну потрібно знову перейти в меню “Manage Jenkins” та в пункті “Global Tool Configuration” знайти блок “SonarQube Scanner”, обрати йому назву SonarQube Scanner та натиснути на кнопку “Add installer”. У вікні, що відкрилось потрібно обрати варіант “Extract *.zip/*.tar.gz” та заповнити наступні поля:

Download URL for binary archive:

<https://binaries.sonarsource.com/Distribution/sonar-scanner-cli/sonar-scanner-cli-4.8.0.2856-linux.zip>

Subdirectory of extracted archive:

[sonar-scanner-4.8.0.2856-linux](#)

Після цього потрібно натиснути на кнопку “Advanced”, в полі Label ввести “built-in” та зберегти конфігурацію.

3. Підготовка SonarQube

Для подальшої роботи потрібно перейти на веб інтерфейс інструмента SonarQube: *http://<адреса_вашого_серверу>:9000* та залогінитись під обліковим записом “admin”. Потім треба проробити наступні кроки:

- перейти на вкладку “Administration”, обрати пункт “Configuration” та обрати варіант “Webhooks”;
- створити новий вебхук, дати йому назву і в полі URL ввести (рис. 12.29):
http://<адреса_вашого_!Jenkins!>:8080/sonarqube-webhook/.
- перейти в SonarQube на свій профіль, натиснувши на іконку користувача, обрати пункт “My account” та підпункт “Security”;
- в блоці “Tokens” в полі “Enter Token Name” ввести довільне ім’я та натиснути на кнопку “Generate”;
- після генерування токена скопіювати його та зберегти для подальшої роботи.

Create Webhook

All fields marked with * are required

Name *

JenkinsHook

URL *

http://http://127.0.0.1:8888//sonarqube-webhook/

Secret

.....

Server endpoint that will receive the webhook payload, for example:
"http://my_server/foo". If HTTP Basic authentication is used, HTTPS is recommended to avoid man in the middle attacks. Example:
"https://myLogin:myPassword@my_server/foo"

If provided, secret will be used as the key to generate the HMAC hex (lowercase) digest value in the 'X-Sonar-Webhook-HMAC-SHA256' header

Create

Cancel

Рис. 12.29. Створення вебхуку у SonarQube

Після отримання відповідних даних слід знову перейти до веб інтерфейсу інструменту Jenkins за посиланням «http://<ваша_ip_адреса>:8080», перейти до пункту “Manage credentials” в меню “Manage Jenkins” і на вкладці “System” обрати “Global credentials”.

Далі потрібно обрати варіант “Add Credentials”; в якості типу облікових даних обрати “Secret text”, ввести згенерований у SonarQube токен в поле “Secret” і зберегти налаштування (рис. 12.30).

Global credentials (unrestricted)

+ Add Credentials

Credentials that should be available irrespective of domain specification to requirements matching.

ID	Name	Kind
 fdda8967-98a6-470b-9c63-f617ba059399	Secret text	Secret text 

Значок: S M L

Рис. 12.30. Додавання токена у Jenkins

Наступним кроком є додавання в меню “Manage Jenkins” / “Configure System”. SonarQube у блоці “SonarQube servers”. Назву можна обрати довільним чином а у полі Server URL слід ввести:

http://<адреса_вашого_!SonarQube!>:9000.

У полі для токену потрібно обрати облікові дані, створені на попередньому кроці та зберегти налаштування (рис. 12.31).

The screenshot shows a configuration form for SonarQube in Jenkins. The form is enclosed in a dashed border. It has the following fields and elements:

- Name:** A text input field containing "SonarServer".
- Server URL:** A text input field containing "http://127.0.0.1:9000". Above the field, it says "Default is http://localhost:9000".
- Server authentication token:** A text input field containing "Secret text". Below the field, it says "SonarQube authentication token. Mandatory when anonymous access is disabled."
- Add:** A button with a dropdown arrow.
- Додаткові:** A button with a dropdown arrow.
- Save:** A dark button at the bottom left.
- Apply:** A light button at the bottom right.

Рис. 12.31. Налаштування SonarQube у Jenkins

4. Підготовка локального середовища Jenkins

В завдання даного розділу в якості тестового проєкту обрано «Vue Paper Dashboard», тому в терміналі на сервері Jenkins потрібно встановити NodeJS командою:

sudo apt install nodejs

Для успішного її виконання користувачу, під яким Jenkins виконує дії у себе локально (jenkins), потрібно надати право використовувати “sudo” без паролю. Для цього у файлі /etc/sudoers потрібно додати в кінці рядок:

jenkins ALL=(ALL) NOPASSWD:ALL

5. Налаштування та запуску Pipeline Job

Оскільки в даному прикладі використовується локальний інстанс Jenkins а не агенти, кроки, такі як “Testing local connection” чи “Testing remote connection” як в попередньому пайплайні, є зайвими. Натомість, потрібно додати кроки для аналізу коду у SonarQube. Таким чином, пайплайн для цього завдання може мати наступний вигляд:

```

pipeline {
    Build and Analysis

    stages {
        stage('Checkout') {
            steps {
                git '<адреса репозиторію з проектом «Vue Paper Dashboard»>'
            }
        }

        stage('Install dependencies') {
            steps {
                sh 'sudo npm install'
            }
        }

        stage('Compile') {
            steps {
                sh 'sudo npm run build'
            }
        }

        stage("SonarQube analysis") {
            environment {
                SONAR_USER_HOME = "/jenkins-agent"
            }
            steps {
                script{
                    def scannerHome = tool 'SonarQube Scanner';
                    withSonarQubeEnv('SonarServer') {
                        sh "${scannerHome}/bin/sonar-scanner -Dsonar.login=sonar -
Dsonar.password=password"
                    }
                }
            }
        }
    }
}

```

Даний pipeline забезпечує автоматизований процес збірки та аналізу якості коду Vue.js проєкту з використанням Jenkins та SonarQube. Розглянемо детальніше вміст блок SonarQube analysis:

- налаштування середовища через блок environment;
- використовує script блоку для більш складної логіки;
- отримання шляху до SonarQube Scanner через функцію tool;
- використання withSonarQubeEnv для налаштування середовища аналізу;
- запуск сканера з параметрами автентифікації.

Після запуску пайплайну має бути отримано повідомлення про помилку на етапі аналізу, в якому буде фігурувати «Project root configuration file: NONE».

```
[ec2-user@ip-172-31-41-66 opt]$ sonar-scanner --version
INFO: Scanner configuration file: /opt/sonar-scanner/conf/sonar-scanner.properties
INFO: Project root configuration file: NONE
INFO: SonarScanner 4.8.0.2856
INFO: Java 11.0.17 Eclipse Adoptium (64-bit)
INFO: Linux 6.1.23-36.46.amzn2023.x86_64 amd64
```

Це пов'язано з тим, що всі необхідні дані авторизації уже збережені в налаштуваннях Jenkins, а для налаштувань проєкту потрібно перейти на GitHub репозиторій проєкту, зазначеного в Jenkins, та в корені створити файл з назвою *sonar-project.properties* з наступним вмістом (рис. 12.32):

```
sonar.host.url=http://127.0.0.1:9000
```

```
# required metadata
sonar.projectKey=lab11
sonar.projectName=lab11
sonar.projectVersion=1.0.0
```

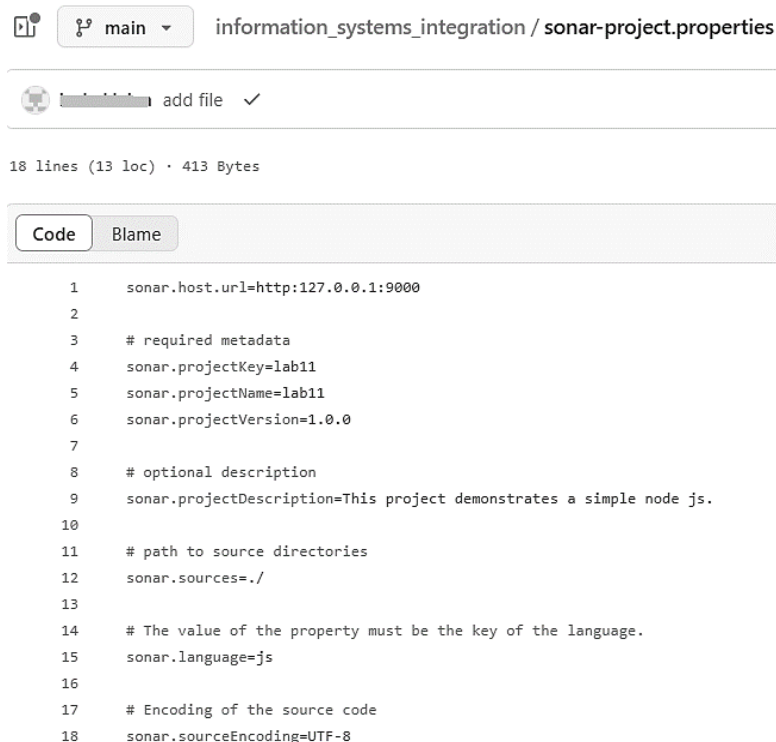
```
# optional description
sonar.projectDescription=This project demonstrates a simple node js.
```

```
# path to source directories
sonar.sources=.
```

```
# The value of the property must be the key of the language.
sonar.language=js
```

```
# Encoding of the source code
sonar.sourceEncoding=UTF-8
```

Протягом часу, зазначеного в налаштуваннях “Poll SCM”, побудова має запуснитись автоматично, оскільки було додано новий файл в корінь проекту. Варто зазначити, що для віртуальних машин, створених засобами VirtualBox деколи “Polling Log” не може знаходити змін. В такому випадку можна запуснути процес побудови вручну, використавши кнопку “Build Now”. Якщо використовуються машини, створені засобами Amazon EC2 AWS, відхилень спостерігатись не має.



```
1 sonar.host.url=http:127.0.0.1:9000
2
3 # required metadata
4 sonar.projectKey=lab11
5 sonar.projectName=lab11
6 sonar.projectVersion=1.0.0
7
8 # optional description
9 sonar.projectDescription=This project demonstrates a simple node js.
10
11 # path to source directories
12 sonar.sources=./
13
14 # The value of the property must be the key of the language.
15 sonar.language=js
16
17 # Encoding of the source code
18 sonar.sourceEncoding=UTF-8
```

Рис. 12.32. Вміст файлу *sonar-project.properties*

Після запуску пайплайна потрібно переконатися у коректній роботі побудованої системи (рис. 12.33). За бажанням можна переглянути та проаналізувати вміст “Console Output”.

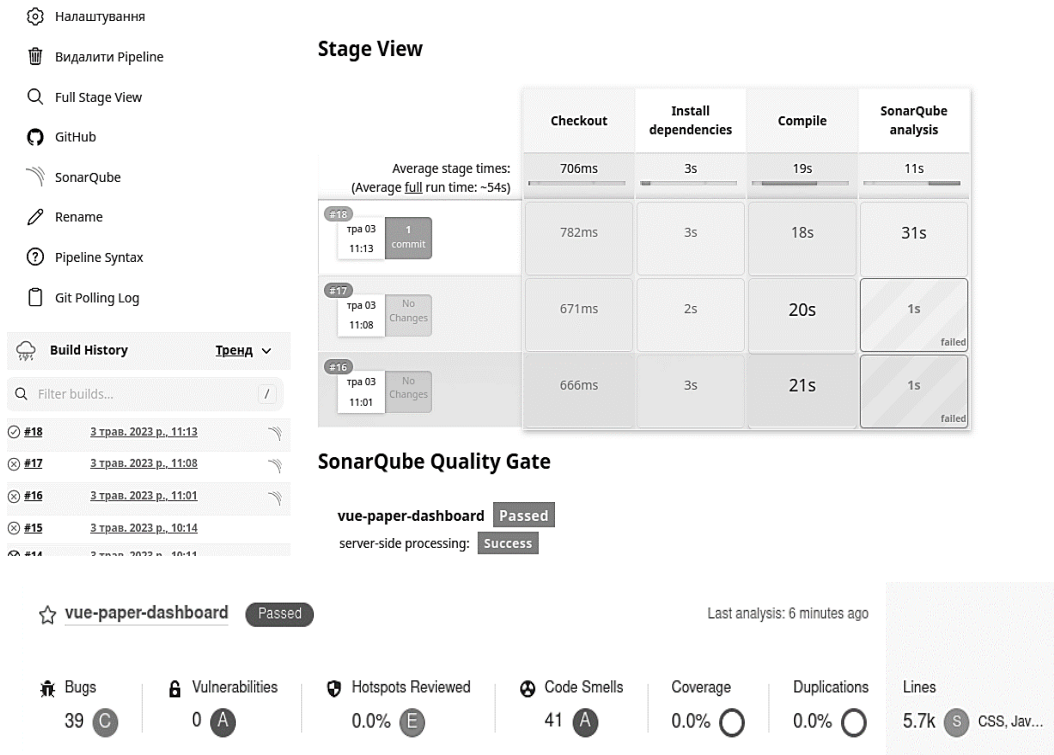


Рис. 12.33. Перегляд результату роботи побудованої системи.

Запропоновані практичні завдання дають змогу переконатись, що інтеграція Jenkins із SonarQube дозволяє створити потужну систему автоматизованого контролю якості коду. Під час роботи було налаштовано автоматичний запуск аналізу коду при кожному оновленні репозиторію на GitHub, що значно спрощує процес контролю якості розробки.

Важливим аспектом такої інтеграції є правильне налаштування всіх компонентів системи – від webhooks та токенів доступу до конфігураційних файлів проекту, що забезпечує безперебійну роботу всього процесу контролю якості. Така автоматизація дозволяє розробникам зосередитись на написанні коду, в той час як система автоматично відслідковує його якість та відповідність встановленим стандартам.

ВІДПОВІДІ НА КОНТРОЛЬНІ ПИТАННЯ

Розділ 1

1) DevOps – це підхід, який об'єднує розробку (Development) і операції (Operations) для забезпечення ефективної та безперервної доставки програмного забезпечення.

2) Основна мета DevOps – забезпечення безперервної інтеграції, тестування, розгортання та моніторингу програмного забезпечення з метою скорочення часу між розробкою та впровадженням.

3) DevOps об'єднує команди через спільні інструменти, автоматизацію процесів і регулярну взаємодію, спрямовану на досягнення спільної мети – стабільної та швидкої доставки продукту.

4) DevOps охоплює розробку коду, автоматизацію тестування, безперервну інтеграцію (CI), розгортання в продуктивне середовище (CD) та моніторинг роботи систем.

5) Безперервна інтеграція (CI) – це практика, яка передбачає автоматичне злиття коду, написаного різними розробниками, в загальний кодовий репозиторій кілька разів на день для запобігання конфліктів та забезпечення стабільності.

6) Безперервне розгортання (CD) – це процес, який автоматизує розгортання програмного забезпечення в продуктивному середовищі. Він дозволяє швидко впроваджувати нові версії продукту без простоїв і мінімальними ризиками.

7) Моніторинг дозволяє стежити за станом програмного забезпечення та інфраструктури, виявляти аномалії і проблеми в реальному часі, а також здійснювати своєчасні коригувальні дії для підтримки стабільної роботи продукту.

8) Автоматизація тестування в DevOps дозволяє швидко виявляти помилки на ранніх етапах розробки, знижує ризики людських помилок і прискорює цикл випуску нових версій програмного забезпечення.

9) DevOps вважається розширенням Agile, оскільки він додає до принципів гнучкої розробки акцент на автоматизацію процесів, тісну інтеграцію команд і забезпечення стабільної роботи продукту на всіх етапах його життєвого циклу.

10) Основна різниця між Agile і DevOps полягає в тому, що Agile фокусується на розробці програмного забезпечення через ітерації, наданні швидкого зворотного зв'язку і гнучкості в управлінні проектами, тоді як

DevOps додає до цього акцент на інтеграцію та автоматизацію процесів, включаючи тестування, розгортання та моніторинг, щоб забезпечити безперервну роботу продукту.

- 11) C
- 12) D
- 13) A
- 14) B
- 15) B
- 16) B
- 17) B
- 18) C
- 19) C
- 20) B
- 21) A-2, B-3, C-4, D-1
- 22) A-1, B-2, C-3, D-4
- 23) A-3, B-4, C-1, D-2
- 24) A-2, B-1, C-3, D-4
- 25) A-3, B-4, C-1, D-2

Розділ 2

1) Основна мета CI – швидке виявлення та виправлення помилок, що мінімізує ризики інтеграції та забезпечує стабільність системи.

2). До основних етапів CI можна віднести наступні:

- внесення змін у код та їх фіксація в репозиторії;
- автоматичне збирання проєкту;
- проведення автоматизованих тестів;
- статичний аналіз коду;
- перевірка відповідності стандартам якості.

3) CI сприяє зменшенню кількості помилок, прискорює розробку, забезпечує прозорість змін, мінімізує конфлікти коду і є одним з ключових факторів підтримки високої якості програмного забезпечення.

4) Під час CI можуть використовуватись наступні види автоматизованих тестів: юніт-тести (перевірка окремих модулів), інтеграційні тести (взаємодія компонентів), функціональні тести (відповідність функцій очікуванням), перформанс-тести (продуктивність та масштабованість).

5) До основних переваг інструменту Jenkins можна віднести його безкоштовність та надійність, велику кількість доступних плагінів, гнучкість

налаштування та масштабованість, можливість інтеграції з різними DevOps-інструментами.

6) Файл `.circleci/config.yml` використовується для налаштування всього процесу CI/CD у CircleCI та має на меті визначення які завдання виконуються, в якому порядку та у яких середовищах. Основні компоненти даного файлу: `version` (визначає версію формату файлу), `jobs` (описує відповідне завдання), `workflows` (визначає порядок і залежності виконання завдань), `commands` (задає повторно використовувані команди) та `executors` (налаштовує середовища виконання).

7) Послідовність етапів у процесі роботи Travis CI зазвичай є наступною:

- внесення змін у репозиторій (наприклад, GitHub);
- читання конфігурації з `.travis.yml` для налаштування середовища та тестів;
- налаштування ізольованого середовища для виконання завдань;
- виконання збірки, встановлення залежностей, запуску тестів;
- автоматичне розгортання змін або зупинка процесу та сповіщення розробників за умови наявності помилок.

8) Інтеграція CI/CD з системами контролю версій забезпечує автоматичний запуск процесів CI/CD після кожного коміту, зберігає історію змін і сприяє прозорості роботи команди.

9) Головною особливістю GitHub Actions можна вважати його тісну інтеграцію з сервісом GitHub, що дозволяє автоматизувати процеси CI/CD безпосередньо у репозиторіях, використовуючи простий синтаксис YAML для налаштувань.

10) Регулярна інтеграція і тестування запобігають накопиченню технічного боргу, оскільки дають змогу вчасно виявляти й виправляти помилки.

11) D

12) B

13) D

14) B

15) B

16) E

17) C

18) B

19) C

20) B

21) 1-B, 2-A, 3- D, 4- C.

22) 1- A, 2- B, 3- D, 4- C.

23) 1-B, 2-A, 3-C, 4-D.

24) 1- C, 2- A, 3- B, 4-D

25) 1- D, 2- A, 3- B, 4- C

Розділ 3

1) Завдяки централізованому підходу створюється єдиний доступний для всіх учасників репозиторій з актуальною версією коду, що запобігає дублюванню та втраті даних.

2) Головним недоліком централізованих систем контролю версій є цілкова залежність від центрального сервера, оскільки за умови його недоступності вся команда втрачає доступ до репозиторію.

3) При роботі з розподіленою системою контролю версій кожен розробник має повну копію репозиторію на своєму пристрої, що дає більше гнучкості та можливостей для автономної роботи.

4) Централізовані системи контролю версій можна використовувати у проєктах із невеликими командами, чіткою ієрархією доступу та стабільною інфраструктурою.

5) Термін «commit» у Git призначений для опису змін у локальному репозиторії та допомагає відстежувати історію проєкту й повертатися до попередніх версій.

6) Зазвичай при роботі з великими проєктами більшість операцій виконується локально, що знижує затримки, пов'язані з швидкістю передачі даних мережею.

7) У процесі неперервної інтеграції використання системи контролю версій забезпечує ефективне управління змінами, сприяє автоматизації та підтримує колективну роботу розробників.

8) Команда `git status` призначена для відображення поточного стану робочого каталогу і області індексації (staging area). Вона показує які файли змінені, але ще не додані до індексації, які файли готові до коміту, які файли не відстежуються Git тощо.

9) Git Flow – це модель роботи з гілками при використанні системи Git. Основні типи гілок: `master` (для роботи зі стабільним кодом), `develop` (використовується для основної розробки проєктів), `feature` (використовується для роботи з новими функціями).

10) Використання системи контролю версій дає можливість за будь-яких змін в коді запускати процеси автоматизованого тестування, збірки та перевірки якості коду.

- 11) В
- 12) D
- 13) В
- 14) С
- 15) С
- 16) С
- 17) Е
- 18) D
- 19) В
- 20) D
- 21) 1- В, 2- С, 3- А, 4-D.
- 22) 1- D, 2- С, 3- А, 4- В.
- 23) 1-A, 2-D, 3 -C, 4-B.
- 24) 1-C, 2-D, 3-A, 4-B.
- 25) 1- С, 2-A, 3- В, 4-D.

Розділ 4

1) GitLab є платформою для управління життєвим циклом розробки програмного забезпечення, що забезпечує централізоване середовище для співпраці всіх учасників процесу розробки та об'єднує всі аспекти DevOps у єдиному середовищі, спрощуючи співпрацю та підвищуючи ефективність роботи команд.

2) До основних функцій GitLab CI/CD можна віднести такі, як автоматизація тестування, збірки, розгортання та перевірки змін у коді.

3) У GitLab можна виділити наступні ролі користувачів: Owner, Maintainer, Developer, Reporter, Guest.

4) GitHub і GitLab забезпечують керування доступом через ролі користувачів (Owner, Maintainer, Developer тощо), захищені гілки та налаштування дозволів.

5) GitLab Runners – це компоненти системи, які запускають визначені в пайплайнах процеси та відповідають за виконання завдань безперервної інтеграції та доставки.

6) Використання механізму захищених гілок (Protected Branches) дозволяє обмежити доступ до важливих гілок, дозволяючи вносити зміни тільки через merge requests та обмежуючи коло осіб, які можуть взаємодіяти з цими гілками.

7) Теги в GitLab являють собою мітки, що можуть створюватися як вручну, так і автоматично через CI/CD пайплайни та використовуються для ідентифікації та управління версіями в процесі розробки.

8) DevSecOps у контексті GitLab є підходом до роботи з даними, що включає інструменти безпеки на всіх етапах DevOps-процесу, включаючи SAST (статичний аналіз коду), DAST (динамічне тестування), Dependency Scanning (аналіз залежностей) та Container Scanning (перевірка контейнерів) для забезпечення безпеки розробки.

9) GitHub є платформою для управління репозиторіями Git, що працює за моделлю SaaS та забезпечує спільну роботу над кодом, обговорення змін, CI/CD і автоматизацію за допомогою GitHub Actions.

10) Bitbucket орієнтований на інтеграцію з Atlassian-сервісами (Jira, Trello), підтримує як Git, так і Mercurial, і надає приватні репозиторії безкоштовно для малих команд.

11) C

12) B

13) B

14) E

15) C

16) D

17) E

18) D

19) B

20) D

21) 1- D, 2- A, 3- C, 4 -B.

22) 1- A, 2- B, 3- C, 4- D.

23) 1- C, 2- A, 3 -B, 4- D.

24) 1- A, 2- B, 3- D, 4- C.

25) 1- A, 2- B, 3- C, 4-D.

Розділ 5

1) Оркестрація неперервної інтеграції - це процес управління і координації автоматизованих етапів розробки, тестування та збірки програмного забезпечення, що має на меті забезпечення швидкого виявлення помилок, підвищення якості продукту та оптимізація процесів розробки.

2) Оркестрація CI зазвичай включає створення pipeline, інтеграцію з репозиторієм, автоматизацію тестів, збірку коду, розгортання і моніторинг результатів.

3) Pipeline у контексті безперервної інтеграції є сукупністю автоматизованих етапів або процесів, які визначають послідовність дій для збірки, інтеграції, тестування та доставки програмного забезпечення.

4) Для збільшення швидкості виконання пайплайнів може бути використано кешування, паралельне виконання завдань, оптимізація тестів і вибір високопродуктивного обладнання.

5) Основними компонентами пайплайна можуть бути завдання для автоматизації процесів тригери (події для запуску), середовища виконання завдань (наприклад, сервери або контейнери) та артефакти (результати завдань).

6) Інтеграція з репозиторіями дозволяє автоматично запускати процеси після змін у коді, забезпечуючи швидку перевірку та виявлення помилок на ранніх етапах розробки.

7) Серед найбільш популярних та ефективних інструментів для оркестрації неперервної інтеграції можна виділити такі, як Jenkins, GitLab CI/CD, CircleCI, Travis CI, Azure DevOps, AWS CodePipeline, Google Cloud Build, GitHub Actions, TeamCity, Bamboo тощо.

8) Контейнери забезпечують ізольоване середовище для виконання тестів і збірок, що гарантує стабільність результатів незалежно від середовища виконання.

9) DevOps інженери налаштовують пайплайни, автоматизують процеси інтеграції та тестування та відповідають за забезпечення моніторингу і масштабованості систем.

10) Кешування в контексті CI є збереженням результатів попередніх інтеграцій, що має на меті пришвидшення повторних збірок.

11) А

12) D

13) B

14) C

15) D

16) B

17) B

18) C

19) E

20) B

21) 1- B, 2- C, 3- A, 4 - D.

22) 1- C, 2- D, 3- B, 4 - A.

23) 1- A, 2- C, 3- D, 4 - B.

24) 1- A, 2- C, 3- D, 4- B.

25) 1- B, 2- C, 3- A, 4-D.

Розділ 6

1) Jenkins зазвичай використовується для автоматизації процесів розробки програмного забезпечення (побудова, тестування, інтеграція та розгортання) та дозволяє ефективно керувати життєвим циклом додатків від розробки до впровадження.

2) Pipeline – це спосіб автоматизації всього процесу розробки програмного забезпечення, що використовується для створення складних багатоступеневих CI/CD процесів із використанням сценаріїв на мові Groovy.

3) Jenkins автоматично збирає звіти про тести, збірку та успішність виконання, генерує детальний звіт та може представити результати графічно.

4) Master-Slave забезпечує високу масштабованість, розподіл навантаження між агентами, підтримку різних середовищ та ізолюваність виконання завдань.

5) Файл з назвою Jenkinsfile зберігається разом з кодом у репозиторі і містить опис процесів CI/CD.

6) Multibranch Pipeline дає змогу автоматизувати процеси створення та управління пайплайнами для кожної гілки репозиторію.

7) Для налаштування середовища Java-проєкту в Jenkins можна використати Git для інтеграції з репозиторієм, Maven Integration для виконання команд Maven, JUnit для збору й аналізу результатів тестування.

8) Щоб підключити Jenkins до GitHub потрібно встановити плагін Git, додати URL-адресу репозиторію в налаштуваннях Jenkins та налаштувати вебхуки (webhook) в GitHub для автоматичного запуску робіт.

9) Slave (Agent) в архітектурі Jenkins – це вузли, що виконують завдання, передані Master, забезпечують виконання збірок, тестів, аналізу коду та інших операцій.

10) Плагіни розширюють функціональність Jenkins, дозволяючи інтегрувати його з іншими інструментами та платформами.

11) С

12) В

13) В

14) Е

15) С

16) D

17) Е

18) D

19) В

20) D

21) 1- C, 2- D, 3- A, 4 -B.

22) 1- B, 2- A, 3- D, 4- C.

23) 1- B, 2- C, 3- D, 4 - A.

24) 1- B, 2- A, 3- C, 4- D.

25) 1- C, 2- B, 3- A, 4-D.

Розділ 7

1) GitHub Actions має на меті спрощення автоматизації робочих процесів, пов'язаних із розробкою програмного забезпечення, об'єднання функцій CI/CD а також виконання завдань, пов'язаних з тестуванням, розгортанням додатків, управлінням релізами, моніторингом та інтеграцією зі сторонніми сервісами.

2) Файли Workflow являють собою YAML-файли, які описують робочі процеси GitHub Actions (кожен файл відповідає за певний робочий процес автоматизації) та зберігаються у каталозі `.github/workflows` репозиторія проєкту.

3) Декларативний підхід у GitHub Actions полягає в тому, що опис завдань зосереджений на результатах, а не процедурах. Для опису робочих процесів використовується YAML формат.

4) GitHub Actions дає змогу користувачам автоматизувати тестування коду, розгортання додатків, управління релізами, моніторинг робочих процесів та інтеграцію зі сторонніми сервісами.

5) GitHub Marketplace є платформою, що містить готові рішення (actions) для поширених завдань, таких як тестування, деплой або інтеграція з сервісами, які можна швидко інтегрувати у свої Workflow.

6) Параметр *needs* у налаштуванні завдань (Jobs) у GitHub Actions визначає залежності між завданнями в Workflow, що допомагає організувати послідовність виконання завдань у складних робочих процесах.

7) GitHub надає для виконання робочих процесів віртуальні машини, які підтримують Linux, Windows і macOS (GitHub-hosted Runners) та власні середовища, що налаштовуються і управляються користувачем на власних серверах або локальних машинах (Self-hosted Runners).

8) Кастомні дії створені розробниками для виконання унікальних завдань. Вони бувають скриптові (написані на мовах програмування, наприклад, JavaScript) та контейнерні (запаковані у Docker-контейнери).

9) GitHub зберігає конфіденційні дані у вигляді секретів (Secrets), які зашифровані та доступні лише під час виконання Workflow, що дає змогу захистити API-ключі, токени доступу та інші важливі дані.

10) GitHub Actions пропонує готові рішення для інтеграції з популярними хмарними платформами, що дозволяє автоматично розгортати додатки, налаштовувати сервіси та оновлювати відповідні ресурси.

11) C

12) B

13) C

14) D

15) C

16) A

17) D

18) D

19) A

20) E

21) 1- C, 2- A, 3- D, 4 -B.

22) 1- A, 2- B, 3- D, 4- C.

23) 1- B, 2- A, 3- C, 4 - D.

24) 1- B, 2- D, 3- C, 4- A.

25) 1- D, 2- B, 3- A, 4- C.

Розділ 8

1) Автоматичне тестування полягає у автоматизації виконання тестів за допомогою інструментів для перевірки різних аспектів програмного забезпечення, що сприяє підвищенню ефективності та якості. Автоматизація швидша, точніша, повторювана, дозволяє тестувати великі набори даних і знижує ризик людських помилок.

2) Тестові скрипти – це автоматизовані сценарії для перевірки функціональності, продуктивності, безпеки тощо.

3) Основна мета функціональних тестів – перевірка функціональності системи з точки зору користувача, оцінка правильності роботи окремих функцій і сценаріїв відповідно до заданих вимог та специфікацій.

4) Автоматизація тестування інтегрується в процеси CI/CD для забезпечення швидкої та якісної перевірки програмного забезпечення на кожному етапі розробки, дозволяє швидко запускати тести після кожного коміту, виявляючи помилки на ранніх етапах розробки. Тести виконуються у пайплайнах, і лише після їх успішного проходження код просувається до наступного етапу.

5) Selenium – це відкрита платформа для автоматизації тестування вебдодатків, яка дозволяє проводити тестування вебдодатки на різних браузерах.

6) Можна виділити наступні типи автоматичних тестів: Unit-тести, інтеграційні тести, регресійні тести, функціональні тести, тести продуктивності системи та тести безпеки.

7) UI тести мають на меті перевірку зовнішнього вигляду, взаємодії та функціональності елементів користувацького інтерфейсу застосунку.

8) Apache JMeter є одним з найбільш відомих універсальних інструментів тестування продуктивності (Performance Testing), що дає змогу тестувати продуктивність та навантаження веб-додатків, баз даних і серверів.

9) Інтеграційні тести мають на меті перевірку взаємодії між окремими модулями або компонентами системи.

10) Завдяки регресійним тестам відбувається перевірка, чи не вплинули нові зміни негативно на існуючий функціонал системи.

11) C

12) B

13) C

14) D

15) C

16) B

17) E

18) B

19) B

20) A

21) 1 - B, 2 - C, 3 - D, 4 - A.

22) 1 - D, 2 - A, 3 - C, 4 - B.

23) 1 - D, 2 - B, 3 - C, 4 - A.

24) 1 - C, 2 - B, 3 - A, 4 - D.

25) 1 - B, 2 - A, 3 - D, 4 - C.

Розділ 9

1) Код-аналіз – це процес автоматизованої перевірки вихідного коду на відповідність заданим стандартам, правилам та можливим дефектам.

2) Можна виділити два основних типи код-аналізу: статичний аналіз (проводиться без запуску програми) та динамічний аналіз (під час виконання програми).

3) Основний принцип «вартості виправлення помилок» полягає у тому, що чим раніше проблему виявлено, тим дешевше її усунути.

4) Завданнями статичного код-аналізу є: перевірка стилю коду, виявлення «мертвого» коду (ідентифікація фрагментів, які більше не використовуються), синтаксичних помилок, дублювань, попередження потенційних помилок та перевірка на відповідність безпековим стандартам для виявлення потенційних вразливостей тощо.

5) До найбільш поширених вразливостей безпеки можна віднести: SQL-ін'єкції, міжсайтовий скриптинг (XSS), проблеми в автентифікації, помилки у доступі до пам'яті (Buffer Overflow).

6) До основних функцій аналізаторів коду можна віднести перевірку синтаксису та стилю, виявлення помилок і дефектів, аналіз продуктивності, пошук вразливостей безпеки та контроль технічного боргу.

7) SQL-ін'єкція – це вразливість, що дозволяє маніпулювати SQL-запитами до бази даних через незахищені входи та зазвичай має на меті отримання несанкціонованого доступу до даних. Код-аналіз допомагає виявити SQL-ін'єкції шляхом автоматичної перевірки вихідного коду на правильну обробку користувацьких даних, використання параметризованих запитів та інших практик безпеки.

8) Автоматизація перевірок коду дає змогу отримати зменшення людського фактору, швидке виявлення помилок, прискорення розробки проєктів та покращення масштабованості для великих проєктів.

9) «Технічний борг» – це накопичення неякісного або застарілого коду, який знижує ефективність роботи системи, ускладнює її підтримку та розвиток (причиною можуть бути компроміси в коді, погане дотримання стандартів або недостатнє тестування).

10) Інструменти код-аналізу автоматично запускаються після внесення змін в репозиторій та перевіряють код на помилки, відповідність стандартам стилю, безпеку та інші аспекти. У випадку критичних помилок процес CI/CD може бути зупинено до їх усунення.

11) В

12) В

13) А

14) D

15) E

16) В

17) E

- 18) В
- 19) В
- 20) Е
- 21) 1 - С, 2 - А, 3 - В, 4 - D.
- 22) 1 - D, 2 - В, 3 - С, 4 - А.
- 23) 1 - D, 2 - С, 3 - В, 4 - А.
- 24) 1 - А, 2 - В, 3 - С, 4 - D.
- 25) 1- В, 2 - А, 3 - D, 4 - С.

Розділ 10

1) SonarQube має на меті автоматичний аналіз якості коду, забезпечення відповідності стандартам та мінімізації технічного боргу та сприяє підвищенню безпеки, підтримуваності й читабельності коду.

2) Архітектура SonarQube включає Database Layer (рівень бази даних), Server Layer (відповідає за обробку даних та інтеграцію з іншими інструментами), SonarScanner (виконує статичний аналіз коду та передає результати на сервер) та Web Interface (відображає результати аналізу, дозволяє переглядати метрики та керувати проектами).

3) SonarQube присвоює проектам, що аналізуються, оцінки за категоріями Reliability (Надійність), Security (Безпека), Security Review (Огляд безпеки) та Maintainability (Підтримуваність).

4) Quality Gates – це набір критеріїв для ключових метрик якості коду (покриття тестами, кількість багів, технічний борг), на основі яких команди розробників приймають рішення про відповідність проекту стандартам якості.

5) Основні метрики, що використовуються в Quality Gates, включають в себе покриття тестами, кількість багів, вразливості, технічний борг, дублікацію коду та складність коду.

6) SonarScanner відповідає за збір, аналіз та передачу даних з вихідного коду до сервера SonarQube, виконуючи статичний аналіз проектів. Він збирає вихідний код, виконує статичний аналіз згідно з налаштованими профілями якості, генерує звіт і передає дані на сервер SonarQube.

7) Code Smells – це слабкі місця або потенційні проблеми в коді, які можуть вказувати на необхідність рефакторингу.

8) SonarQube може бути інтегрований у пайплайни CI/CD, виконуючи автоматичний аналіз під час кожного коміту чи пул-реквесту, надаючи зворотний зв'язок у режимі реального часу.

9) Jenkins може автоматично запускати SonarScanner для аналізу коду після кожного коміту, а Quality Gates перевіряють результати та можуть

блокувати збірку при невідповідності критеріям якості, що забезпечує раннє виявлення критичних помилок.

10) В DevSecOps SonarQube використовується для виявлення та усунення вразливостей у коді, таких, наприклад, як SQL-ін'єкції чи міжсайтовий скриптинг (XSS), класифікуючи їх за рівнем критичності.

11) D

12) D

13) B

14) C

15) D

16) B

17) A

18) B

19) B

20) C

21) 1 - D, 2 - C, 3 - A, 4 - B.

22) 1 - A, 2 - D, 3 - C, 4 - B.

23) 1 - B, 2 - A, 3 - D, 4 - C.

24) 1 - B, 2 - A, 3 - C, 4 - D.

25) 1 - B, 2 - D, 3 - A, 4 - C.

Розділ 11

1) Мінімальна CI-система має включати в себе систему контролю версій (наприклад, Git), інструмент для автоматизації процесів збірки, тестування та інтеграції (наприклад, Jenkins, GitLab CI/CD чи GitHub Action) та інструменти аналізу якості коду (SonarQube).

2) Webhooks є механізмом зворотного виклику, який дозволяє автоматично на основі HTTP-запитів передавати інформацію між сервісами у відповідь на певну подію, що дає змогу мінімізувати затримки та виключити необхідність періодичних перевірок.

3) Webhooks можуть обробляти широкий спектр подій (push, merge, створення задач тощо), забезпечують швидкий та автоматизований обмін даними між сервісами, дозволяючи автоматично запускати процеси відразу після виникнення подій, сприяють зменшенню навантаження на сервер.

4) Jenkins може інтегруватися з GitHub через Git Plugin для підключення репозиторію, Webhooks для миттєвого запуску завдань, а також Polling SCM для періодичної перевірки змін у репозиторії.

5) Артефакт в контексті CI є результатом успішної збірки проєкту (наприклад, виконуваний файл, Docker-образ, архів із залежностями або статичний вебсайт), який використовується для подальших етапів CI/CD, таких як тестування, розгортання або збереження в репозиторії для майбутнього використання.

6) Коректно побудована CI-система дає змогу автоматизувати процеси тестування, збірки, розгортання та перевірки коду, що дозволяє швидко виявляти та виправляти помилки на всіх етапах розробки.

7) Основні етапи CI-процесу можуть включати в себе push змін у репозиторій, запуск Webhook для виклику інструменту CI (наприклад, Jenkins), збірку проєкту, автоматичне тестування, аналіз якості коду та створення артефактів для подальшого розгортання.

8) Безпека під час інтеграції з зовнішніми сервісами може бути забезпечена через використання HTTPS, токенів доступу та SSH-ключів, а також обмеження доступу за IP-адресами та налаштування фаєрволів тощо.

9) Jenkins підтримує нотифікації через Email, Slack, Telegram, Discord і GitHub, що дозволяє оперативно інформувати команду про статус завдань, успіх або помилки збірки.

10) Cron-стиль дає змогу налаштувати точний розклад виконання завдань та використовується у Jenkins для планування регулярних завдань, таких як запуск тестів або перевірок коду, коли Webhooks недоступні.

11) D

12) A

13) B

14) C

15) B

16) B

17) E

18) C

19) A

20) C

21) 1 - D, 2 - B, 3 - A, 4 - C.

22) 1 - A, 2 - D, 3 - C, 4 - B.

23) 1 - B, 2 - C, 3 - D, 4 - A.

24) 1 - B, 2 - C, 3 - A, 4 - D.

25) 1 - B, 2 - C, 3 - A, 4 - D.

ГЛОСАРІЙ ТЕРМІНІВ

Action – виконуваний модуль, який можна використовувати для автоматизації певних задач

Actionable Insights – кроки або рекомендації, які можуть бути виконані для покращення якості коду

Agent – агент: вузол, на якому виконуються завдання Jenkins

Agile – гнучка методологія управління проектами, що зосереджується на коротких ітераціях і постійній адаптації до змін

API – інтерфейс програмування застосунків, що дозволяє взаємодіяти з іншими системами чи сервісами

Artifact – файли, створені під час виконання workflow, які можуть бути завантажені або використані пізніше

Automated Testing – процес використання спеціалізованих інструментів для виконання тестів без втручання людини

Automation – використання інструментів для автоматизації процесів, таких як тестування, інтеграція, розгортання

Azure DevOps – платформа від Microsoft для CI/CD і управління DevOps

Bamboo – CI/CD-інструмент від Atlassian для комплексного управління процесами

Blue-Green Deployment – синьо-зелене розгортання: стратегія оновлення, яка зменшує ризики простою

Branch – відгалуження основного репозиторію, яке використовується для роботи над окремими функціями чи виправленнями

Bug – проблема в програмі, що викликає її неправильну роботу

Build – процес компіляції вихідного коду в готову для запуску програму

Build Artifact – артефакт збірки: результат процесу збірки, наприклад виконуваний файл або архів

Build Automation – автоматизація процесу компіляції коду та створення артефактів

Build Breaker – компонент, який зупиняє збірку проекту, якщо якість коду не відповідає встановленим вимогам, наприклад, виявлення критичних помилок

Build Pipeline послідовність етапів, через які проходить код для його побудови, тестування та розгортання

Build Quality – показник якості збірки програмного продукту, який оцінюється через кількість помилок і проблем у коді на кожному етапі його побудови

Build Server – сервер збірки: сервер, який автоматично запускає процеси CI

Build Trigger – Тригер збірки: подія, яка запускає виконання пайплайну, наприклад зміна в репозиторії

Centralized Version Control System (CVCS) – центральна система контролю версій, яка використовує один сервер для зберігання історії змін, наприклад, Subversion (SVN)

CI Server – система, яка автоматизує процеси зборки та тестування програмного забезпечення в рамках практики неперервної інтеграції

CI/CD – безперервна інтеграція та доставка – підхід до автоматизації тестування та розгортання коду

CircleCI – хмарний сервіс CI/CD із широкими можливостями інтеграції

Clone – операція створення локальної копії віддаленого репозиторію

Code Coverage – відсоток коду, який проходить через тести, що дає уявлення про повноту тестування

Code Coverage – Покриття коду: метрика, яка показує, який відсоток коду перевірено тестами

Code Metrics – показники, що оцінюють різні аспекти коду, такі як складність, обсяг, або рівень покриття тестами

Code Quality – характеристики коду, що визначають його ефективність, читабельність, підтримуваність та безпеку

- Code Quality** – якість коду: покращення шляхом автоматичного тестування й аналізу
- Code Refactoring** – процес покращення структури коду без зміни його зовнішнього поведінки, спрямований на підвищення його якості
- Code Repository** – репозиторій коду: місце зберігання вихідного коду (наприклад, GitHub, GitLab)
- Code Review** – перевірка коду для покращення якості перед об'єднанням у репозиторій
- Code Smells** – терміни, що описують потенційно погану практику програмування, яку слід виправити, щоб покращити якість коду
- Collaboration** – співпраця між різними командами для досягнення спільної мети
- Commit** – фіксація змін у локальному репозиторії з додаванням опису змін
- Configuration Management** – процес автоматизації та підтримки конфігурацій серверів та додатків
- Conflict** – Конфлікт: ситуація, коли зміни в різних гілках суперечать одна одній
- Container Registry** – сховище для збереження контейнерів Docker у GitLab
- Containerization** – технологія, яка ізолює додатки в контейнери для забезпечення їх портативності (наприклад, Docker)
- Continuous Delivery (CD)** – автоматизація процесу релізу, що дозволяє швидко випускати готовий до продакшн код
- Continuous Deployment** – безперервне розгортання: автоматичне розгортання змін у виробниче середовище
- Continuous Integration (CI)** – процес регулярного інтегрування змін до коду з автоматичним тестуванням для швидкого виявлення помилок
- Continuous Monitoring** – неперервне спостереження за системою для виявлення проблем продуктивності та безпеки

- Continuous Testing** – постійне тестування коду для забезпечення його функціональності та продуктивності
- Coverage** – покриття: показник, який демонструє, яка частина коду була перевірена тестами
- Cron** – спосіб опису запланованих завдань, що використовуються для періодичного запуску workflow
- Dependabot** – інструмент для автоматичного оновлення залежностей та виявлення вразливостей у них
- Dependency** – залежність – зовнішня бібліотека чи пакет, необхідний для роботи проєкту
- Dependency Management** – управління зовнішніми бібліотеками та компонентами, на які залежить програмне забезпечення
- Deployment** – розгортання додатка або сервісу в робочому середовищі, доступному для користувачів
- Deployment Automation** – автоматизація розгортання: інструменти й процеси для автоматичного оновлення програмного забезпечення
- Develop** – гілка для інтеграції змін та роботи над новими функціями
- DevOps** – підхід, який інтегрує розробку (Development) та операційну діяльність (Operations) для прискорення життєвого циклу розробки
- Diff** – порівняння двох версій коду для виявлення змін
- Distributed Version Control System (DVCS)** – розподілена система контролю версій, де кожен користувач має копію всієї історії змін (наприклад Git)
- Docker** – платформа для контейнеризації, яка дозволяє створювати ізольовані середовища для виконання програм
- Dynamic Analysis** – перевірка коду під час його виконання
- Dynamic Code Analysis** – аналіз коду під час його виконання для виявлення проблем, які проявляються лише в процесі роботи програми
- Environment** – оточення, в якому запускається програма (тестове, продуктивне тощо)

ESLint – інструмент для аналізу JavaScript-коду з метою виявлення потенційних помилок або невідповідності стилю

Fail Fast – принцип, згідно з яким тести зупиняються після першої виявленої помилки

Failure – невдалий результат тесту або етапу конвеєра

Faster Feedback – швидший зворотний зв'язок: швидке виявлення помилок

Feature Branch – гілка, створена для розробки окремої функції

Feedback Loop – цикл зворотного зв'язку між командами розробки та операцій для швидкого покращення

Firewall – мережевий екран, який контролює доступ між різними частинами мережі

Fork – створення копії репозиторію для незалежної роботи над проектом

Functional Test – перевірка відповідності функціональності програми вимогам

Git – розподілена система контролю версій, яка є основою для GitLab

Git Flow – методологія управління гілками в Git, що визначає правила для роботи над кодом

GitHub Actions – інтеграція CI/CD в GitHub

GitLab CI/CD – інструмент CI/CD, інтегрований у GitLab

Historical Trends – тренди і зміни якості коду, які можна відстежувати з часом за допомогою SonarQube, щоб побачити покращення або погіршення якості коду

Hotfix Branch – гілка для виправлення термінових помилок у продакшн-коді

Infrastructure as Code (IaC) – практика управління інфраструктурою за допомогою програмного коду замість ручної конфігурації

Integration Test – тест, що перевіряє взаємодію між окремими модулями програми

Issue – елемент для управління запитами, помилками чи функціями в проекті

Issue Tracker – система для відстежування і управління виявленими проблемами у коді, такі як баги або невідповідність стандартам

Jenkins – популярний сервер CI/CD із відкритим кодом

Job – одиниця роботи в CI/CD конвеєрі, наприклад, запуск тестів або збірка коду

Kubernetes – система оркестрації контейнерів для управління масштабуванням і розгортанням

Label – тег для категоризації та організації завдань або запитів

Lean – концепція, що спрямована на мінімізацію втрат у процесах розробки та доставки

Linting – автоматична перевірка коду на відповідність стилю та виявлення помилок

Local Repository – локальна копія репозиторію, що знаходиться на пристрої користувача

Log – журнал записів про виконання завдань пайплайну

Main (або Master) – основна гілка, яка зберігає стабільну версію коду

Marketplace – платформа GitHub для пошуку та використання готових дій (Actions)

Master/Main Branch – головна гілка репозиторію, що містить стабільну версію коду

Matrix Testing – тестування в різних конфігураціях середовищ, таких як різні операційні системи або версії бібліотек

Merge – процес об'єднання змін із різних гілок у спільний код

Merge Conflict – ситуація, коли зміни в різних гілках суперечать одна одній

Merge Request (MR) – запит на об'єднання змін між гілками

Metrics – кількісні показники, які використовуються для вимірювання якості коду, такі як кількість помилок, складність або покриття тестами

Microservices – архітектурний стиль, що розбиває додаток на невеликі незалежні сервіси

Milestone – ключові точки в проєкті для організації завдань та відстеження прогресу

Mocking – техніка імітації поведінки зовнішніх залежностей під час тестування

Orchestration – координація автоматизованих процесів у розгортанні (наприклад, Kubernetes)

Performance Test – оцінка швидкодії та стабільності програми при певних навантаженнях

Pipeline – послідовність етапів, які автоматично виконуються для перевірки, збірки, тестування та розгортання коду

Pipeline Job – окремий крок у процесі CI/CD

Prettier – інструмент для форматування коду відповідно до заданих правил

Production Environment – виробниче середовище: середовище, де працює реальний продукт

Protected Branch – гілка, що обмежує доступ до редагування для запобігання помилкам

Proxy – сервер, що діє як посередник між клієнтом і цільовим сервером для забезпечення безпеки або доступу до мережі

Pull – завантаження змін з віддаленого репозиторію до локального

Pull Request – запит на внесення змін до основної гілки репозиторію після перевірки та тестування

Pull Request Analysis – аналіз коду, що додається до основної гілки через pull request, що дозволяє виявити проблеми ще до злиття коду

Push – відправлення змін з локального репозиторію до віддаленого

Quality Gate – визначені пороги якості коду, які повинні бути досягнуті, щоб код був дозволений до злиття або випуску

Quality Profile – набір правил для аналізу коду, які визначають, які типи проблем будуть виявлені під час перевірки

Reduced Risks – зменшення ризиків: виявлення конфліктів коду на ранніх етапах

Refactoring – процес переписування частин коду для покращення його структури, збереження функціональності, без додавання нових фіч

Regression Test – перевірка, чи нові зміни не вплинули на вже працюючий функціонал

Release – створення версії продукту з певним набором функцій та виправленням помилок

Release Branch – гілка, що використовується для підготовки до релізу нової версії

Release Management – управління плануванням, розробкою, тестуванням і випуском програмного забезпечення

Release Pipeline – пайплайн випуску: послідовність кроків для підготовки програмного забезпечення до випуску

Remote Repository – віддалений репозиторій, що знаходиться на сервері та використовується для співпраці

Repository – сховище, де зберігається вихідний код проєкту разом із його історією змін

Rollback – повернення системи до попереднього стабільного стану через виявлення помилок

Runners – середовище (хост), у якому виконуються завдання, наприклад, GitHub-hosted або self-hosted сервери

Scalability – масштабованість: здатність системи ефективно працювати зі зростанням обсягу даних або користувачів

Secrets – зашифровані дані (наприклад, токени або ключі API), які використовуються в робочих процесах

Security Hotspot – частина коду, яка потенційно є вразливою і потребує додаткової перевірки з точки зору безпеки

Security Test – перевірка захищеності системи від потенційних загроз

Slack Notifications – інтеграція з платформою Slack для надсилання повідомлень про стан workflow

Smoke Test – базова перевірка працездатності системи перед більш детальним тестуванням

Snippet – невеликий шматок коду, яким можна поділитися

Sonar Scanner – сканер Sonar: інструмент для запуску аналізу коду у SonarQube

SonarLint – розширення для IDE, яке дає змогу виконувати локальний аналіз коду і попереджає про проблеми безпосередньо під час написання

SonarQube – інструмент для аналізу та покращення якості коду, що підтримує різні мови програмування і надає візуалізацію результатів, таких як виявлення помилок, вразливостей і проблем з кодом

SonarQube Dashboard – панель управління в SonarQube, на якій відображаються основні метрики якості коду, звіти та інша важлива інформація

SonarQube Instance – окремий екземпляр SonarQube, який може бути налаштований для конкретної команди або організації

SonarQube Plugins – розширення SonarQube, які додають підтримку нових мов програмування, аналітичних можливостей або інтеграцій з іншими інструментами

SonarQube Report – звіт, що генерується SonarQube після аналізу коду, який містить детальну інформацію про знайдені проблеми і їхнє вирішення

SonarQube Server – сервер, на якому розміщується інстанція SonarQube і відбувається обробка результатів аналізу коду

SonarScanner – клієнт SonarQube, який сканує проект, відправляє дані на сервер SonarQube і отримує звіти про якість коду

SSH Key – метод автентифікації для безпечного підключення до віддаленого репозиторію

SSL/TLS – протокол шифрування: технологія для захищеного з'єднання через Інтернет

Staging Environment – проміжне середовище: середовище для тестування, яке імітує виробниче

Static Code Analysis – метод перевірки коду без його виконання, з метою виявлення помилок, потенційних уразливостей та інших проблем у програмі

Step – одиничний крок у завданні, наприклад, виконання скрипта або запуск команди

Tag – маркер для позначення конкретних комітів, зазвичай для версій релізів

Team Collaboration – співпраця в команді: полегшення інтеграції змін різними розробниками

TeamCity – інструмент CI/CD із розширеною підтримкою JetBrains

Technical Debt – витрати на виправлення проблем у коді, які виникли через вибір швидших, але менш оптимальних рішень

Test Automation – автоматизація тестування: використання інструментів для автоматичного виконання тестів

Test Coverage – метрика, що показує, яку частину коду перевіряють тести

Test Suite – група тестів, об'єднана для спільного виконання

Testability – легкість, з якою код можна протестувати за допомогою автоматизованих тестів

Testing – процес перевірки коду на коректність виконання заданих функцій

Thresholds – порогові значення, які визначають межі допустимих значень для метрик якості коду, наприклад, максимальний рівень покриття тестами

Token – цифровий ключ, що використовується для автентифікації або авторизації

Travis CI – сервіс CI для відкритих і приватних проєктів

Trigger – подія, що запускає виконання робочого процесу (наприклад, push або pull request)

Unit Test – тест, який перевіряє окремий модуль або функцію програми

Variables – налаштовувані значення, що використовуються для конфігурації pipeline

Version Control System (VCS) – система контролю версій: програмне забезпечення для управління змінами в коді, наприклад Git

Virtual Machine (VM) – віртуальна машина: програмна імітація апаратної системи для виконання додатків

Vulnerability – слабе місце в програмі або системі, яке може бути використано для атаки

Vulnerability Detection – процес виявлення потенційних вразливостей у коді, які можуть бути використані зловмисниками для атак

Web IDE – вбудоване середовище розробки коду безпосередньо у GitLab

Webhook – механізм, який ініціює автоматичні дії (наприклад, запуск CI) у відповідь на події в репозиторії

Wiki – вбудована документація проекту у вигляді сторінок

Workflow – набір завдань, які виконуються автоматично для досягнення певної мети в рамках автоматизації

YAML – формат файлу для опису конфігурації робочих процесів у GitHub Actions

СПИСОК ВИКОРИСТАНОЇ ТА РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

1. Agarwal Gaurav. Modern DevOps Practices: Implement and secure DevOps in the public cloud with cutting-edge tools, tips, tricks, and techniques / G. Agarwal. – Packt Publishing Ltd, 2021. – 432 p.
2. Al-Shakargi Seif. Investigating Continuous Delivery as a Self-Service / S. Al-Shakargi. – LU-CS-EX, 2019. – 98 p.
3. Appel Frank. Testing with JUnit / F. Appel. – Packt Publishing Ltd, 2015. – 284 p.
4. Armstrong Steven. DevOps for Networking / S. Armstrong. – Packt Publishing Ltd, 2016. – 350 p.
5. Bass Len, Ingo Weber, Liming Zhu. DevOps: A software architect's perspective / L. Bass, I. Weber, L. Zhu. – Addison-Wesley Professional, 2015. – 288 p.
6. Belmont Jean-Marcel. Hands-On Continuous Integration and Delivery: Build and release quality software at scale with Jenkins, Travis CI, and CircleCI / J.-M. Belmont. – Packt Publishing Ltd, 2018. – 384 p.
7. Brikman Yevgeniy. Terraform: Up and Running / Y. Brikman. – O'Reilly Media, Inc., 2022. – 368 p.
8. Chapman Rob, and Peter Holmes. Observability with Grafana: Monitor, control, and visualize your Kubernetes and cloud platforms using the LGTM stack / R. Chapman, P. Holmes. – Packt Publishing Ltd, 2024. – 452 p.
9. Chhajed Saurabh. Learning ELK stack / S. Chhajed. – Packt Publishing Ltd, 2015. – 312 p.
10. Cockburn Alistair. Agile software development: the cooperative game / A. Cockburn. – Pearson Education, 2006. – 504 p.
11. Cowell Christopher, Nicholas Lotz, Chris Timberlake. Automating DevOps with GitLab CI/CD Pipelines: Build efficient CI/CD pipelines to verify, secure, and deploy your code using real-life examples / C. Cowell, N. Lotz, C. Timberlake. – Packt Publishing Ltd, 2023. – 382 p.
12. Craig, Rick David, and Stefan P. Jaskiel. Systematic software testing. Artech House, 2002.
13. Davis Jennifer, Ryn Daniels. Effective DevOps: Building a culture of collaboration, affinity, and tooling at scale / J. Davis, R. Daniels. – O'Reilly Media, Inc., 2016. – 410 p.
14. Dougherty Carol. Prometheus / C. Dougherty. – Routledge, 2006. – 245 p.

15. Dustin, Elfriede, Jeff Rashka, and John Paul. Automated software testing: introduction, management, and performance. Addison-Wesley Professional, 1999.
16. Duvall Paul M., Steve Matyas, Andrew Glover. Continuous Integration: Improving software quality and reducing risk / P. M. Duvall, S. Matyas, A. Glover. – Pearson Education, 2007. – 336 p.
17. Evertse, Joost. Mastering GitLab 12: Implement DevOps culture and repository management solutions / J. Evertse. – Packt Publishing Ltd, 2019. – 424 p.
18. Farcic Viktor. The DevOps 2.0 toolkit: Automating the continuous deployment pipeline with containerized microservices / V. Farcic. – CreateSpace Independent Publishing Platform, 2016. – 440 p.
19. Farcic Viktor. The DevOps 2.1 Toolkit: Docker Swarm / V. Farcic. – Packt Publishing Ltd, 2017. – 350 p.
20. Freeman Emily. DevOps for Dummies / E. Freeman. – John Wiley & Sons, 2019. – 368 p.
21. Gibb Donald, Sabaratnam Arulkumaran, eds. Fetal Monitoring in Practice-E-Book / D. Gibb, S. Arulkumaran. – Elsevier Health Sciences, 2023. – 270 p.
22. Graham, Dorothy, and Mark Fewster. Experiences of test automation: case studies of software test automation. Addison-Wesley Professional, 2012.
23. Gross Kenneth. Puppet: An essay on uncanny life / K. Gross. – University of Chicago Press, 2011. – 224 p.
24. Gundecha Unmesh. Selenium Testing Tools Cookbook / U. Gundecha. – Packt Publishing Ltd, 2015. – 432 p.
25. Hall Daniel. Ansible Configuration Management / D. Hall. – Packt Publishing, 2015. – 250 p.
26. Hethey Jonathan M. GitLab repository management / J. M. Hethey. – Packt Publishing, 2013. – 120 p.
27. Hochstein Lorin, Rene Moser. Ansible: Up and Running: Automating configuration management and deployment the easy way / L. Hochstein, R. Moser. – O'Reilly Media, Inc., 2017. – 380 p.
28. Httermann Michael. DevOps for developers / M. Httermann. – Apress, 2012. – 196 p.
29. Humble Jez, David Farley. Continuous delivery: reliable software releases through build, test, and deployment automation / J. Humble, D. Farley. – Pearson Education, 2010. – 512 p.

30. Husain Aatif M., Saurabh R. Sinha. Continuous EEG monitoring: Principles and practice / A. M. Husain, S. R. Sinha. – Springer, 2017. – 346 p.
31. Jahnke James A. Continuous emission monitoring / J. A. Jahnke. – John Wiley & Sons, 2022. – 336 p.
32. Kaczanowski Tomek. Practical Unit Testing with TestNG and Mockito / T. Kaczanowski. – Tomasz Kaczanowski, 2012. – 320 p.
33. Keating Jesse. Mastering Ansible / J. Keating. – Packt Publishing Ltd, 2017. – 270 p.
34. Kim Gene. The DevOps handbook: How to create world-class agility, reliability, & security in technology organizations / G. Kim. – It Revolution, 2021. – 480 p.
35. Krief Mikael. Learning DevOps: A comprehensive guide to accelerating DevOps culture adoption with Terraform, Azure DevOps, Kubernetes, and Jenkins / M. Krief. – Packt Publishing Ltd, 2022. – 560 p.
36. Kuhn Daniel. Distributed Version Control Systems / D. Kuhn. – 2010. – 75 p.
37. Laster Brent. Professional Git / B. Laster. – John Wiley & Sons, 2016. – 344p.
38. Latshaw George. The complete book of puppetry / G. Latshaw. – Courier Corporation, 2000. – 240 p.
39. Leszko Rafal. Continuous Delivery with Docker and Jenkins / R. Leszko. – Packt Publishing Ltd, 2017. – 340 p.
40. Loeliger Jon, Matthew McCullough. Version Control with Git: Powerful tools and techniques for collaborative software development / J. Loeliger, M. McCullough. – O'Reilly Media, Inc., 2012. – 456 p.
41. Loukides Mike. What is DevOps? / M. Loukides. – O'Reilly Media, Inc., 2012. – 120 p.
42. Luksa Marko. Kubernetes in action / M. Luksa. – Simon and Schuster, 2017. – 608 p.
43. Mahalingam Manoj. Learning Continuous Integration with TeamCity / M. Mahalingam. – Packt Publishing Ltd, 2014. – 164 p.
44. Martin Robert Cecil. Agile software development: principles, patterns, and practices / R. Cecil Martin. – Prentice Hall PTR, 2003. – 552 p.
45. Mason Helen. Chef / H. Mason. – Gareth Stevens Publishing LLLP, 2014. – 32 p.
46. McQuaid Mike. Git in practice / M. McQuaid. – Simon and Schuster, 2014. – 272 p.

47. Miell Ian, Aidan Sayers. Docker in practice / I. Miell, A. Sayers. – Simon and Schuster, 2019. – 424 p.
48. O'grady Adam. GitLab Quick Start Guide: Migrate to GitLab for all your repository management solutions / A. O'grady. – Packt Publishing Ltd, 2018. – 218 p.
49. Pathania Nikhil. Learning Continuous Integration with Jenkins / N. Pathania. – Packt Publishing Ltd, 2016. – 240 p.
50. Pipinellis Achilleas. GitHub essentials. Vol. 2 / A. Pipinellis. – Packt Publishing, 2015. – 172 p.
51. Poulton Nigel. The kubernetes book / N. Poulton. – NIGEL POULTON LTD, 2023. – 350 p.
52. Rady Ben, Rod Coffin. Continuous Testing: with Ruby, Rails, and JavaScript / B. Rady, R. Coffin. – Pragmatic Bookshelf, 2011. – 168 p.
53. Rajput Dinesh. Hands-On Microservices–Monitoring and Testing: A performance engineer's guide to the continuous testing and monitoring of microservices / D. Rajput. – Packt Publishing Ltd, 2018. – 358 p.
54. Rooney Garrett, Daniel Berlin. Practical Subversion / G. Rooney, D. Berlin. – Apress, 2007. – 296 p.
55. Rossel Sander. Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment / S. Rossel. – Packt Publishing Ltd, 2017. – 362 p.
56. Salituro Eric. Learn Grafana 7.0: A beginner's guide to getting well versed in analytics, interactive dashboards, and monitoring / E. Salituro. – Packt Publishing Ltd, 2020. – 280 p.
57. Sayfan Gigi. Mastering Kubernetes: Master the art of container management by using the power of Kubernetes / G. Sayfan. – Packt Publishing Ltd, 2018. – 652 p.
58. Shah Gourav. Ansible Playbook Essentials / G. Shah. – Packt Publishing Ltd, 2015. – 202 p.
59. Sharma Sanjeev. The DevOps adoption playbook: a guide to adopting DevOps in a multi-speed IT enterprise / S. Sharma. – John Wiley & Sons, 2017. – 384 p.
60. Soni Mitesh. DevOps for Web Development / M. Soni. – Packt Publishing Ltd, 2016. – 298 p.
61. Soni Mitesh. Implementing DevOps with Microsoft Azure / M. Soni. – Packt Publishing Ltd, 2017. – 332 p.

62. Tovmasyan Karen. Mastering AWS CloudFormation: Plan, develop, and deploy your cloud infrastructure effectively using AWS CloudFormation / K. Tovmasyan. – Packt Publishing Ltd, 2020. – 350 p.
63. Tsitoara Mariot. Beginning Git and GitHub / M. Tsitoara. – Springer, New York, 2020. – 348 p.
64. Turnbull James. Monitoring with Prometheus / J. Turnbull. – Turnbull Press, 2018. – 298 p.
65. Turnbull James. The Docker Book. Containerization is the new virtualization / J. Turnbull. – Turnbull Press, 2019. – 320 p.
66. Turnbull James. The Terraform Book / J. Turnbull. – James Turnbull, 2016. – 250 p.
67. Utting, Mark, and Bruno Legeard. Practical model-based testing: a tools approach. Elsevier, 2010.
68. Vadapalli Sricharan. DevOps: continuous delivery, integration, and deployment with DevOps: dive into the core DevOps strategies / S. Vadapalli. – Packt Publishing Ltd, 2018. – 300 p.
69. Van Baarsen Jeroen. GitLab Cookbook / J. Van Baarsen. – Packt Publishing Ltd, 2014. – 240 p.
70. van Merode Henry. Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines / H. van Merode. – Apress, 2023. – 180 p.
71. Vehent Julien. Securing DevOps: security in the cloud / J. Vehent. – Simon and Schuster, 2018. – 352 p.
72. Verona Joakim, Michael Duffy, Paul Swartout. Learning DevOps: Continuously Deliver Better Software / J. Verona, M. Duffy, P. Swartout. – Packt Publishing Ltd, 2016. – 360 p.
73. Verona Joakim. Practical DevOps / J. Verona. – Packt Publishing Ltd, 2016. – 240 p.
74. Verona Joakim. Practical DevOps: Implement DevOps in your organization by effectively building, deploying, testing, and monitoring code / J. Verona. – Packt Publishing Ltd, 2018. – 280 p.
75. Wilson Christie. Grokking Continuous Delivery / C. Wilson. – Simon and Schuster, 2022. – 350 p.
76. Wingerd Laura. Practical Perforce / L. Wingerd. – O'Reilly Media, Inc., 2005. – 320 p.
77. Wolff Eberhard. A Practical Guide to Continuous Delivery / E. Wolff. – Addison-Wesley Professional, 2017. – 396 p.

Навчальне видання

Михайло ЛУЧКЕВИЧ, Ірина ШАКЛЕІНА, Олекса ГУДЬ

ІНТЕГРАЦІЯ ІНФОРМАЦІЙНИХ СИСТЕМ

НАВЧАЛЬНИЙ ПОСІБНИК

Дизайн обкладинки

Ірина Шаклеїна

Комп'ютерне оформлення

Михайло Лучкевич

Ірина Шаклеїна

Комп'ютерна верстка

Наталія Дум'як

Підписано до друку 05.08.2025 р. Формат 70х100/16.

Папір офсетний. Гарнітура Minion Pro. Друк офсетний. Наклад 300 прим.

Умов. друк. арк. 32,4

Видавець і виготовлювач – «Коло»

вул. Олега Дяківа, 8, м. Дрогобич, Україна, 82100

Свідоцтво суб'єкта видавничої справи

ДК № 498 від 20.06.2001

kolodruk@gmail.com

Друк ПП «Коло»

вул. Олега Дяківа, 8, м. Дрогобич, Україна, 82100

Свідоцтво суб'єкта видавничої справи

ДК № 498 від 20.06.2001

kolodruk@gmail.com