ПОЛТАВСЬКИЙ ДЕРЖАВНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЕКОНОМІКИ, УПРАВЛІННЯ, ПРАВА ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

Пояснювальна записка

до кваліфікаційної роботи на здобуття ступеня вищої освіти магістр

на тему: **«Дослідження методів та алгоритмів налаштування CI/CD для Gitlab-репозиторіїв»**

Виконав: здобувач вищої освіти
за освітньо-професійною програмою
Інформаційні управляючі системи та
технології спеціальності
126 Інформаційні системи та технології
ступеня вищої освіти магістр
групи 126ІСТ_мд_21
Говоров І.С.
Керівник: Калініченко А.В.
Рецензент: Біловод О.І.

Полтава – 2023 року

ВСТУП

У сучасному світі програмної інженерії, неперервна інтеграція (Continuous Integration, CI) та неперервна доставка (Continuous Delivery, CD) є вирішальними компонентами ефективного процесу розробки та розгортання програмного забезпечення. Ці практики сприяють збільшенню продуктивності розробників та забезпечують високу якість кінцевого продукту завдяки швидкому виявленню та виправленню помилок, автоматизації тестування та забезпеченню надійності розгортання.

GitLab, як інтегрована платформа для управління кодом, проєктами та DevOps-процесами, відіграє ключову роль у реалізації CI/CD. Вона надає широкий спектр інструментів для автоматизації різних етапів розробки, від написання коду до його розгортання.

У даній роботі розглядаються методи та алгоритми налаштування CI/CD для GitLab-репозиторіїв, виконується аналіз можливостей, які надає GitLab для автоматизації процесів неперервної інтеграції та доставки, а також розробка рекомендацій щодо ефективного налаштування та оптимізації CI/CD pipelines. Окрема увага приділяється аспектам безпеки, масштабування та інтеграції з іншими сервісами та інструментами.

Через швидкий розвиток технологій та зростаючу складність програмних продуктів, важливість ефективного налаштування CI/CD не може бути недооцінена. Вона відіграє критичну роль у забезпеченні високої швидкості розробки та надійності програмних продуктів.

Актуальність роботи обумовлена кількома важливими факторами, які визначають сучасні тенденції в області розробки програмного забезпечення:

- у контексті постійної еволюції технологій, здатність швидко адаптуватися до змін та впроваджувати нові рішення є критичною для успіху IT-проєктів. CI/CD є ключовими елементами у забезпеченні цієї гнучкості;

- неперервна інтеграція та доставка сприяють підвищенню якості продуктів, завдяки регулярному тестуванню та автоматизації, вони дозволяють виявляти та виправляти помилки на ранніх стадіях розробки;

- зростання складності програмних продуктів та потреба у їх швидкому оновленні вимагають високого рівня автоматизації. CI/CD відіграють ключову роль у цій автоматизації;

- у сучасних розробках часто використовуються різноманітні інструменти та платформи, інтеграція яких є складним завданням. Вивчення методів налаштування CI/CD у GitLab допоможе ефективно інтегрувати ці інструменти;

- у сучасних умовах важливо забезпечувати безперервність та надійність програмних рішень, CI/CD сприяють досягненню цієї мети.

Таким чином, вивчення та оптимізація процесів CI/CD в GitLab має значний практичний інтерес і є актуальним для розробників, менеджерів проєктів, а також для широкої аудиторії IT-фахівців.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконана у відповідності до науково-дослідної ініціативної теми «Організаційно-методологічні аспекти впровадження інформаційно-комунікаційних систем і технологій в управлінні діяльністю сучасних організацій та підприємств за умов переходу до цифрової економіки» ДРН 0117U003099.

Мета дослідження: вивчення та аналіз методів та алгоритмів налаштування CI/CD для GitLab-репозиторіїв.

Завдання роботи:

- теоретичний аналіз CI/CD процесів – вивчення основних концепцій, переваг та викликів, пов'язаних з неперервною інтеграцією та неперервною доставкою в контексті розробки програмного забезпечення;

- дослідження можливостей GitLab як платформи для CI/CD – аналіз функціональності GitLab у контексті CI/CD, включаючи GitLab CI/CD pipelines, runners, та інші важливі компоненти;

- вивчення методів налаштування pipelines – детальний огляд та налаштування .gitlab-ci.yml файлу, визначення оптимальних конфігурацій для різних етапів процесу розробки (build, test, deploy);

- оптимізація та масштабування CI/CD pipelines – вивчення стратегій та підходів до оптимізації та масштабування CI/CD процесів, особливо для великих та складних проєктів;

- практичне застосування – використання CI/CD в GitLab, включаючи приклади які ілюструють ефективність запропонованих методів та підходів

Об'єктом дослідження є процеси неперервної інтеграції (CI) та неперервної доставки (CD) у контексті використання системи контролю версій та спільної роботи GitLab.

Предметом дослідження є методи та алгоритми налаштування CI/CD для GitLab-репозиторіїв.

Методи дослідження:

- аналіз наукових та професійних публікацій, статей, прикладів, технічної документації, блогів та інших ресурсів, що стосуються CI/CD, GitLab та суміжних технологій;

- вивчення конкретних прикладів використання CI/CD в реальних проєктах;

- практична реалізація та тестування різних конфігурацій CI/CD у GitLab на демонстраційних проєктах;

- аналіз отриманих результатів;

- синтез та формулювання рекомендацій та висновків дослідження.

Інформаційна база дослідження: науково-технічна література, книги та посібники з налаштування процесів неперервної інтеграції (CI) та неперервного розгортання (CD) для Gitlab-репозиторіїв; статті з актуальних досліджень у галузі автоматизації розробки, доступні в наукових базах даних; міжнародні та національні стандарти у сфері розробки програмного забезпечення; офіційна документація та навчальні ресурси платформи Gitlab, рекомендації з налаштування CI/CD; матеріали конференцій та семінарів з автоматизації процесів розробки та впровадження інноваційних підходів в IT.

Елементи наукової новизни: аналітичний огляд існуючих підходів до налаштування CI/CD у Gitlab; адаптація методів налаштування процесів неперервної інтеграції (CI) та неперервного розгортання (CD) для Gitlab-репозиторіїв, які оптимізують робочі процеси та підвищують ефективність розробки в цьому середовищі.

Практична значущість: результати аналізу існуючих методів CI/CD в контексті Gitlab-репозиторіїв допомагає визначити найефективніші підходи та стратегії, які можуть бути використані в різних проєктах розробки програмного забезпечення; рекомендації щодо оптимізації процесів неперервної інтеграції та неперервного розгортання в середовищі Gitlab є корисними для розробників і IT-фахівців, зацікавлених у покращенні робочих процесів.

Апробація результатів дослідження. За результатами проведеного дослідження опубліковано тези: І. Говоров. Ретроспектива практик CI/CD у контексті розробки програмного забезпечення. *Управління ресурсним забезпеченням господарської діяльності підприємств реального сектору економіки*: збірник тез доповідей VIII Всеукраїнської науково-практичної інтернет-конференції, 23 листопада 2023 року, м. Полтава. Полтава: ПДАУ, 2023.

Структура та обсяг кваліфікаційної роботи. Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел та додатків. Основний текст роботи викладений на 65 сторінках, містить 21 таблицю, 8 рисунків. Список використаних джерел налічує 49 найменувань.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ CI/CD

1.1 Ретроспектива практик CI/CD

Неперервна інтеграція (CI) та неперервна доставка (CD) є фундаментальними поняттями в сучасній практиці розробки програмного забезпечення [1]. Вони забезпечують ефективність, швидкість та якість у процесах розробки, тестування та розгортання продуктів. Цей розділ присвячений описанню теоретичних аспектів CI/CD, що є важливим для успішного впровадження та оптимізації цих процесів у виробничому середовищі.

Необхідність впровадження CI [1-4] та CD [5-9] виникла з потреби у швидкій та ефективній розробці програмного забезпечення. Ідея CI з'явилась у 1990-х роках, коли в рамках екстремального програмування (XP) була висунута концепція частого злиття коду розробників [10]. Це було зроблено з метою мінімізації конфліктів при злитті та забезпечення більшої сумісності роботи команди. Згодом, у 2000-х роках, з появою і розвитком інструментів автоматизації, таких як Jenkins, Travis CI та інших, CI стала набувати більш конкретних форм [11]. Ці інструменти дозволили автоматизувати процеси збірки та тестування коду, що стало важливим кроком у розвитку практик неперервної інтеграції.

Паралельно з розвитком CI, почала складатися практика неперервної доставки (CD), яка зосереджувалася на автоматизації не тільки збірки та тестування, але й розгортання програмного забезпечення. CD забезпечує можливість швидко та ефективно доставляти оновлення та нові функції до кінцевих користувачів, що є особливо важливим у сучасних умовах швидкого розвитку технологій та зростаючих очікувань користувачів.

З появою хмарних технологій та мікросервісної архітектури практики CI/CD набули ще більшої актуальності та стали важливою складовою DevOps-культури. Хмарні платформи, такі як AWS, Azure, Google Cloud, запропонували розширені

можливості для автоматизації та масштабування, що дало новий поштовх розвитку CI/CD (рис. 1.1).

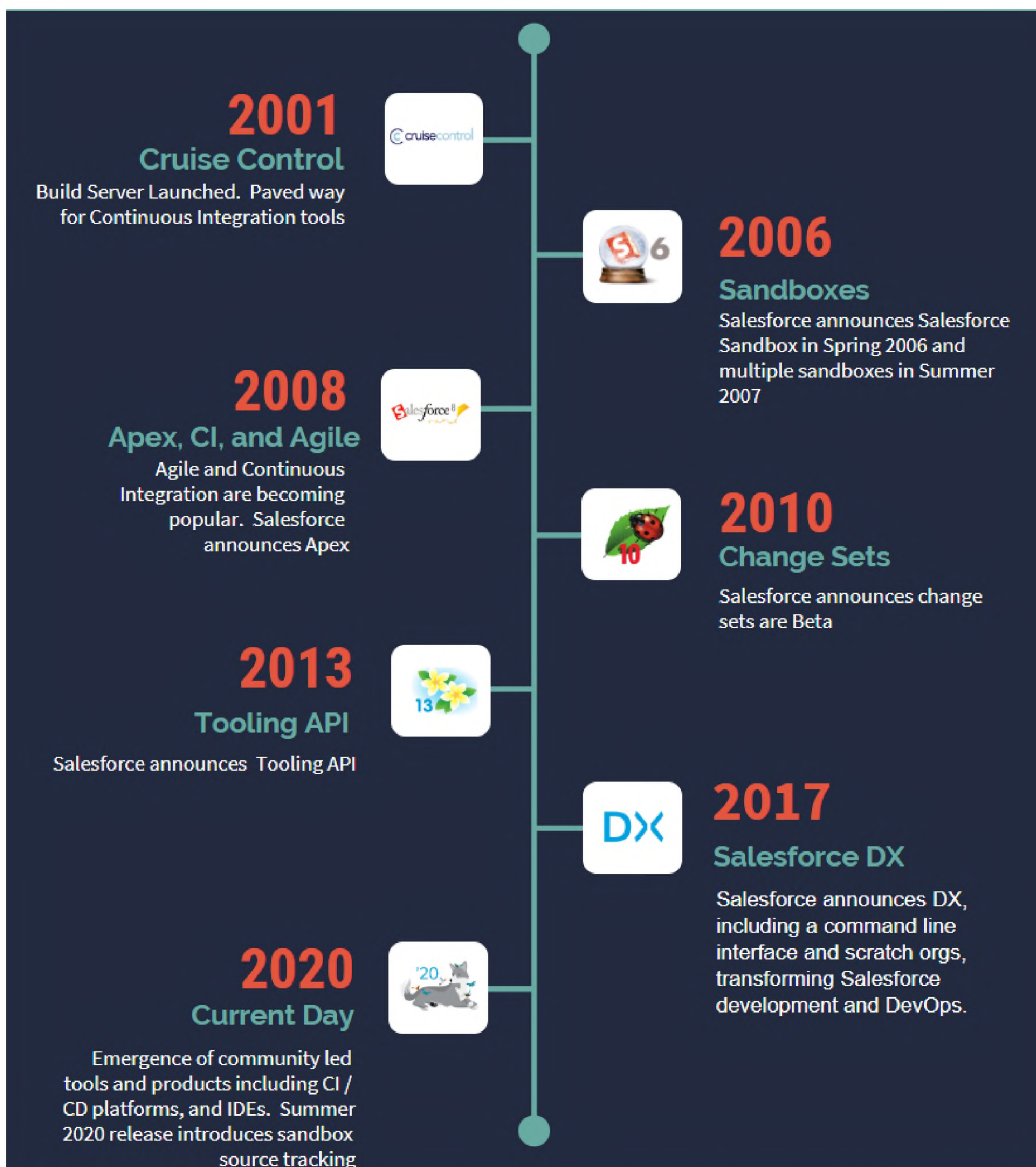

Рисунок 1.1 – Хронологія розвитку CI/CD [12]

Сьогодні CI/CD є не тільки технічною необхідністю, але й критичним елементом для успішної конкурентоспроможності на ринку програмного забезпечення.

У табл. 1.1 відображено поетапний розвиток CI/CD, від ранніх концепцій та інструментів до сучасного комплексного підходу, що включає хмарні технології,

мікросервіси та інтеграцію з різними платформами та інструментами, представлені основні етапи у розвитку методології та практики CI/CD.

Таблиця 1.1 – Основні етапи розвитку методології та практики CI/CD

Період	Основні етапи та зміни
1990-ті- початок 2000-х	Виникнення концепції CI в рамках екстремального програмування (XP). Ручні процеси інтеграції та мінімальна автоматизація.
2000-і роки	Поява інструментів автоматизації (Jenkins, CruiseControl). Інтеграція з системами контролю версій.
2010-ті роки	Розвиток CD та поширення DevOps культури. Інтеграція збірки, тестування та розгортання в єдиний процес.
2020-ті роки	Роль хмарних технологій та мікросервісів. Інтеграція CI/CD з різними інструментами для комплексного підходу до розробки.

Отже, історія розвитку CI/CD розпочинається з ранніх концепцій екстремального програмування з мінімальною автоматизацією в 1990-х, через появу інструментів автоматизації та розвиток DevOps у 2000-их і 2010-их, і продовжується до сучасного комплексного підходу, що охоплює хмарні технології та мікросервіси у 2020-их роках.

1.2 Основні концепції та термінологія CI/CD

Ключовими поняттями у сучасній практиці розробки програмного забезпечення, які відіграють важливу роль у забезпеченні ефективності, швидкості та якості процесу розробки та доставки продуктів є неперервна інтеграція, неперервна доставка та неперервне розгортання.

Неперервна інтеграція (Continuous Integration, CI) – практика в області розробки програмного забезпечення, яка полягає у частому та систематичному злитті змін коду до основної гілки репозиторію. Головна мета CI – швидке виявлення та вирішення конфліктів, помилок, забезпечення сумісності нових змін із існуючим кодом. CI включає автоматичну збірку та тестування коду після кожного коміту, що дозволяє забезпечити високу якість коду та ефективність розробки.

Неперервна доставка (Continuous Delivery, CD) – практика, яка включає автоматизацію всіх етапів доставки програмного продукту від розробки до розгортання. Основна мета CD – забезпечити готовність програмного продукту до розгортання у будь-який час. У CD, зміни, що пройшли всі етапи тестування та інтеграції, готові до випуску в продакшн, причому процес розгортання зазвичай вимагає ручного схвалення.

Неперервне розгортання (Continuous Deployment) – розширює поняття неперервної доставки. У неперервному розгортанні, зміни в коді, що успішно проходять усі етапи автоматизованого тестування, автоматично розгортаються у продакшн-середовище, що забезпечує ще більшу швидкість доставки змін до кінцевих користувачів і мінімізує втручання людини у процес доставки програмного продукту.

Термін CI/CD об’єднує дві ключові практики у розробці програмного забезпечення: неперервну інтеграцію (CI) та неперервну доставку (CD) [13]. Розглянемо, що означає кожна з цих практик та їх взаємозв’язок.

На рис. 1.2 представлена діаграма, яка схематично зображено процеси CI/CD, які слідують один за одним у чітко визначені послідовності.

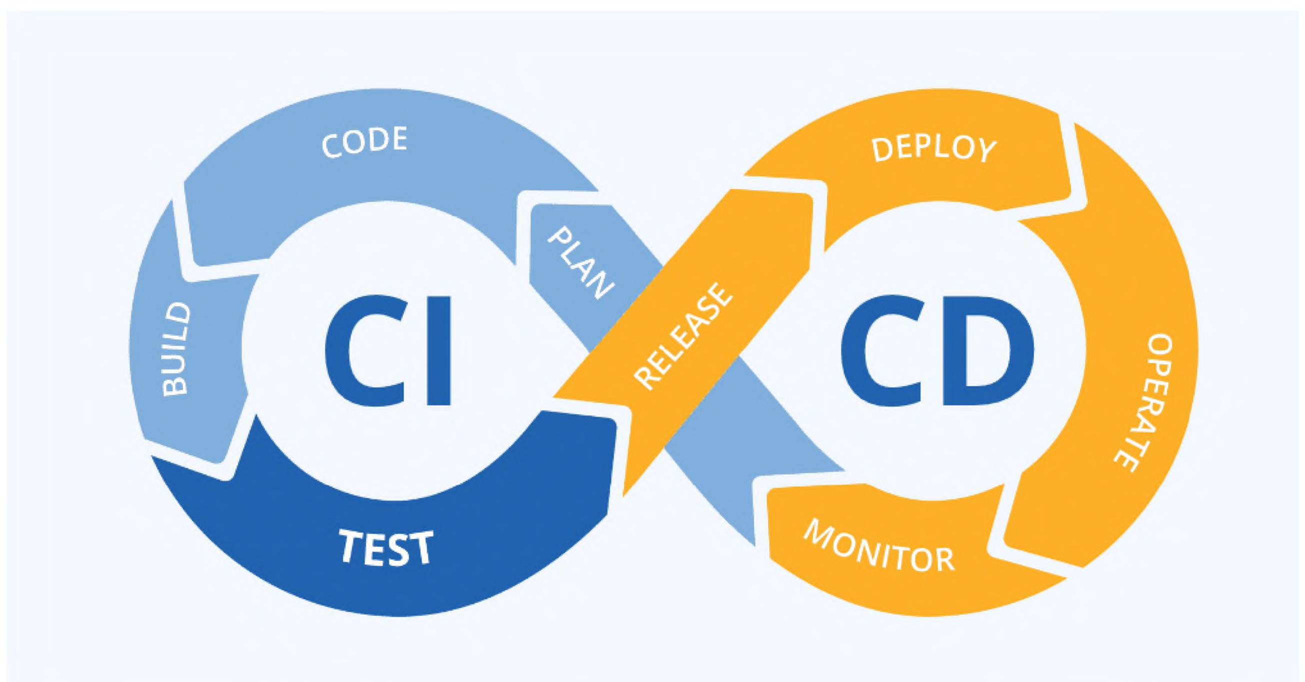

Рисунок 1.2 – Послідовність процесів CI/CD [14]

Неперервна інтеграція (CI) – це процес, у якому розробники регулярно (часто по кілька разів на день) інтегрують свої зміни коду у спільний репозиторій. Після кожної інтеграції здійснюється автоматична збірка та тестування коду, що дозволяє швидко виявляти та виправляти помилки, знижуючи ризики пов’язані зі злиттям коду.

Неперервна доставка (CD) – це практика, що є логічним продовженням CI. Вона передбачає автоматизацію процесу доставки зібраного та протестованого коду до середовища розгортання (наприклад, тестового або продакшн). Це забезпечує готовність програмного продукту до випуску в будь-який час.

Об’єднання процесів CI та CD утворює CI/CD-пайплайн [15-17], який забезпечує автоматизований та плавний процес від розробки коду до його доставки та, у випадку застосування неперервного розгортання, до його випуску. Це дозволяє розробникам зосередитися на розробці нових функцій, одночасно підтримуючи високу стабільність та якість продукту. Таким чином, завдяки CI/CD, організації можуть швидше реагувати на зміни ринку та потреби користувачів, забезпечуючи постійне вдосконалення своїх продуктів.

На рис. 1.3 представлена діаграма типового пайплайну (конвеєру) CI/CD.

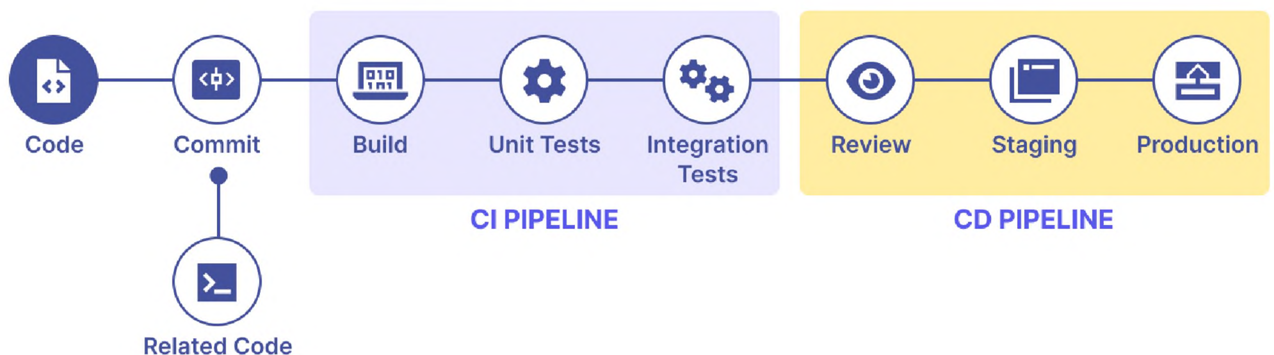

Рисунок 1.3 – Діаграма типового пайплайну (конвеєру) CI/CD [18]

CI/CD-пайплайн (pipeline, конвеєр) – це набір процедур та інструментів, які використовуються для автоматизації процесів розробки програмного забезпечення. Пайплайн об’єднує практики CI та CD для забезпечення

ефективності та швидкості розробки, тестування, доставки та, у деяких випадках, автоматичного розгортання продукту.

Основні компоненти пайплайну CI/CD включають такі елементи [19]:

- інтеграція коду – розробники регулярно зливають свій код у спільний репозиторій;

- автоматична збірка та тестування – після злиття коду автоматично запускається процес збірки та виконуються тести для виявлення помилок;

- розгортання – успішно протестований код переноситься до середовища розгортання;

- випуск продукту – в кінцевому варіанті, код може бути автоматично випущений в продакшн.

У табл. 1.2 представлений типовий пайплайн для проєкту з розробки вебдодатку.

Таблиця 1.2 – Типовий пайплайн розробки вебдодатку

Етап	Опис
Інтеграція коду	Розробники регулярно зливають зміни у головну гілку репозиторію на GitHub.
Автоматична збірка та тестування	Jenkins автоматично запускає процес збірки та виконує набір тестів (unit, integration).
Перевірка якості коду	Використання SonarQube для статичного аналізу коду та перевірки відповідності стандартам.
Розгортання в тестове середовище	Автоматичне розгортання коду в тестове середовище для подальшого ручного тестування.
Випуск у продакшн	Після успішного тестування, код розгортається в продакшн-середовище.

На практиці, такий пайплайн реалізується наступним чином:

1. Кожен розробник працює над певними завданнями і регулярно зливає свої зміни у головну гілку репозиторію на GitHub.

2. Після кожного коміту, Jenkins (або інший CI інструмент) автоматично запускає процес збірки та виконує набір тестів (unit tests, integration tests) для перевірки нових змін.

3. Інструменти атоматизованого аналізу коду (наприклад, SonarQube), перевіряють якість коду на відповідність стандартам.

4. Якщо всі автоматизовані тести пройдено успішно, код автоматично розгортається в тестове середовище для подальшого ручного тестування.

5. Після підтвердження, що код успішно пройшов усі тести та готовий до випуску, він розгортається в продакшн-середовище

В основу CI/CD покладено специфічні принципи та підходи, що створюють підґрунтя для ефективної, гнучкої та якісної розробки програмного забезпечення, а також сприяють створенню культури постійного поліпшення та інновацій у команді розробників [20]. До них належать:

- автоматизація процесів. Одним з ключових принципів CI/CD є повна автоматизація процесів збірки, тестування, доставки та розгортання програмного забезпечення. Це дозволяє мінімізувати ручну працю, знижувати ризики людських помилок та прискорювати цикл розробки;

- регулярна інтеграція та доставка. Неперервна інтеграція передбачає регулярне злиття робочих гілок розробників з основною гілкою, щоб уникнути конфліктів та помилок, які можуть виникати при довгостроковій розробці. Неперервна доставка гарантує, що кожна зміна, яка успішно проходить тести, готова до розгортання;

- швидке виявлення та виправлення помилок. CI/CD дозволяє оперативно виявляти та виправляти помилки, оскільки кожна зміна відразу проходить процедуру тестування. Це підвищує якість коду та зменшує час, необхідний на виявлення та вирішення проблєм;

- транспарентність та зворотний зв’язок. CI/CD сприяє підвищенню транспарентності процесу розробки. Команда розробників має можливість отримувати швидкий зворотний зв’язок про статус збірок та тестів, що підвищує ефективність співпраці та прийняття рішень;

- гнучкість та масштабованість. CI/CD підходи мають бути гнучкими, щоб адаптуватися до різних умов проєкту та потреб команди. Вони також мають забезпечувати масштабованість для підтримки зростання та розвитку проєкту;

- безперервне вдосконалення. CI/CD це не тільки автоматизація та ефективність, а також безперервне вдосконалення процесів та продукту, що передбачає постійний аналіз, оцінку та оптимізацію процесів розробки.

1.3 Переваги та проблєми впровадження CI/CD

CI/CD вносить значний вклад у ефективність розробки, забезпечує більшу швидкість, якість та автоматизацію процесів. Ці головні переваги CI/CD дозволяють командам розробників бути більш гнучкими та реактивними до змін, а також підтримувати високий рівень якості продукту.

Швидкість розробки. CI/CD значно прискорює процеси розробки. Через регулярну інтеграцію та автоматизоване тестування, команди можуть швидше виявляти та виправляти помилки, зменшуючи час, необхідний для ручного відстеження та виправлення помилок (багів). Автоматизація розгортання та доставки зменшує потребу в ручних операціях, що дозволяє командам швидше випускати оновлення продукту та нові функції.

Підвищення якості. Регулярне тестування, яке є частиною CI/CD-пайплайну, підвищує загальну якість продукту. Неперервне виявлення та виправлення помилок гарантує, що кінцевий продукт буде мінімально містити вади. Стабільність коду підвищується, оскільки всі зміни ретельно тестуються перед злиттям у основну гілку.

Автоматизація. Автоматизація є однією з основних переваг CI/CD. Вона не тільки зменшує ручну роботу, але й знижує ризик людських помилок. Автоматизовані процеси забезпечують консистентність та відтворюваність процесів розробки та розгортання, що є критично важливим для великих та складних проєктів.

Проілюструємо практику CI/CD на прикладі, коли команда розробників працює над великим вебдодатком, який вимагає регулярних оновлень, швидкої реакції на зміни та високої стабільності.

Швидкість розробки. Розробники регулярно інтегрують свої зміни в основний репозиторій. CI-система (наприклад, Jenkins) автоматично запускає збірку та тестування після кожного коміту. Це дозволяє виявляти помилки на ранніх стадіях, значно скорочуючи час на їх виправлення. Завдяки цьому, можливо швидше випускати нові функції та оновлення для додатку, підтримуючи високу конкурентоспроможність продукту на ринку.

Підвищення якості. Автоматизоване тестування в рамках CI гарантує, що будь-які нові зміни не порушують існуючу функціональність. Це веде до підвищення стабільності та надійності вебдодатку. CD дозволяє розробникам бути впевненими, що код, який проходить всі тести, готовий до розгортання в продакшн, забезпечуючи високу якість кінцевого продукту.

Автоматизація. Всі процеси збірки, тестування та розгортання вебдодатку повністю автоматизовані. Це знижує ризик людських помилок та звільняє час розробників для зосередження на розробці нових функцій та вдосконалення існуючого продукту. Автоматизоване розгортання у тестове середовище та продакшн-середовище забезпечує плавність та швидкість доставки оновлень кінцевим користувачам.

Результати. Команда розробників здатна швидко реагувати на зміни у вимогах та забезпечувати постійне вдосконалення додатку. Забезпечується висока стабільність та якість продукту, зменшуються простої та помилки в продакшні. Ефективність роботи команди зростає, оскільки значна частина рутинних процесів автоматизована.

Цей приклад демонструє, як CI/CD може позитивно вплинути на розробку вебдодатків, забезпечуючи швидкість, якість та ефективність в роботі команди розробників.

Можливі виклики та труднощі при впровадженні CI/CD включають наступне.

Спротив змінам всередині команди розробників. Одним з основних викликів є опір змінам з боку команди розробників, особливо якщо вони звикли

до традиційних підходів розробки. Перехід на CI/CD вимагає зміни менталітету та підходів до роботи.

Потреба у навчанні та розвитку навичок. Впровадження CI/CD вимагає від розробників володіння новими навичками, такими як робота з інструментами автоматизації, розуміння процесів тестування та деплойменту. Це може потребувати часу та ресурсів на навчання.

Вибір та налаштування інструментів. Вибір правильних інструментів для CI/CD є важливим і може бути складним. Слід враховувати сумісність з існуючим стеком технологій, масштабування, вартість та легкість використання [20].

У табл. 1.3 узагальнено переваги CI/CD, а також виклики та труднощі, які можуть виникати при впровадженні CI/CD.

Таблиця 1.3 – Переваги та проблєми впровадження CI/CD

Переваги CI/CD	Виклики та труднощі при впровадженні CI/CD
Швидкість розробки: регулярна інтеграція та швидке виявлення помилок.	Спротив змінам у команді: необхідність зміни менталітету.
Підвищення якості: стабільний код завдяки автоматизованому тестуванню.	Потреба у навчанні та розвитку навичок.
Автоматизація процесів: зниження ручної праці та людських помилок.	Вибір та налаштування інструментів: сумісність, масштабування, вартість.
	Інтеграція з існуючими процесами: складність координації.
	Підтримка та управління інфраструктурою: необхідність технічної підтримки.
	Забезпечення безпеки: захист інформації та дотримання стандартів.
	Вимірювання ефективності та ROI.

Інтеграція з існуючими процесами. Інтеграція CI/CD у вже існуючі робочі процеси може бути складною, особливо у великих організаціях зі складними системами. Це може вимагати значних зусиль для координації та планування.

Підтримка та управління інфраструктурою. CI/CD вимагає належної підтримки та управління інфраструктурою. Це включає налаштування серверів, моніторинг, забезпечення безперебійної роботи та вирішення проблєм.

Проблєма безпеки. Під час впровадження CI/CD важливо убезпечити процеси. Це означає захист інформації, регулярні перевірки на вразливості та дотримання стандартів безпеки.

Вимірювання ефективності та ROI. Визначення та вимірювання ефективності CI/CD є складним завданням. Кожній організації (команді) необхідно розробити власні метрики для оцінки повернення інвестицій (ROI) та загальної продуктивності.

Виклики та труднощі впровадження CI/CD, які відображені у табл. 1.3, вимагають ретельного планування, залучення всієї команди розробників, вибору правильних інструментів та неперервного вдосконалення процесів. Однак, не зважаючи на це, переваги впровадження CI/CD здатні значно поліпшити ефективність та якість розробки програмного забезпечення.

1.4 Ключові технічні компоненти CI/CD

До ключових компонентів, що технічно забезпечують функціонування CI/CD, належать: системи контролю версій, сервери CI, інструменти автоматизації процесів CI/CD, інструменти для тестування, системи моніторингу та логування, хмарні платформи CI/CD [21, 22].

Системи контролю версій, такі як Git, SVN, Mercurial, забезпечують зберігання, відстеження та управління змінами у коді. Вони дозволяють кільком розробникам одночасно працювати над проєктом, надаючи можливість злиття змін та вирішення конфліктів.

Сервери CI, такі як Jenkins, Travis CI, CircleCI, використовуються для автоматизації процесів збірки та тестування коду. Вони автоматично виконують збірку проєкту та запускають тести кожного разу, коли вносяться зміни в систему контролю версій.

Інструменти автоматизації, як Ansible, Puppet, Chef, використовуються для автоматизації розгортання та управління інфраструктурою. Вони дозволяють ефективно розповсюджувати та масштабувати застосунки в різних середовищах.

Інструменти для тестування, як Selenium, JUnit, PyTest, використовуються для автоматизації тестування. Вони дозволяють проводити різні види тестів (unit tests, integration tests, end-to-end tests) для перевірки функціональності та надійності коду.

Системи моніторингу та логування, такі як Prometheus, ELK Stack, Elasticsearch, Logstash, Kibana, допомагають відстежувати стан системи та додатків у реальному часі. Вони забезпечують збір, аналіз та візуалізацію логів, що дозволяє швидко реагувати на проблєми.

Хмарні платформи, як AWS, Azure, Google Cloud, надають інфраструктуру та сервіси для розгортання та масштабування додатків. Вони забезпечують гнучкість та масштабованість, що є важливим для CI/CD.

Ці компоненти разом створюють потужну екосистему для автоматизації процесів розробки, тестування, доставки та розгортання програмного забезпечення, сприяючи ефективності та якості продукту.

Загальні відомості про ключові технічні компоненти CI/CD представлені у табл. 1.4.

Таблиця 1.4 – Основні технічні компоненти CI/CD

Компонент	Інструмент	Опис
Системи контролю версій	Git, SVN, Mercurial	Зберігання, відстеження та управління змінами у коді.
Сервери CI	Jenkins, Travis CI, CircleCI	Автоматизація збірки та тестування коду.
Інструменти автоматизації	Ansible, Puppet, Chef	Автоматизація розгортання та управління інфраструктурою.
Інструменти для тестування	Selenium, JUnit, PyTest	Автоматизація тестування для перевірки функціональності та надійності коду.
Системи моніторингу та логування	Prometheus, ELK Stack	Збір, аналіз та візуалізація логів для моніторингу стану системи.
Хмарні платформи	AWS, Azure, Google Cloud	Надання інфраструктури та сервісів для розгортання та масштабування додатків.

Компоненти технічного забезпечення CI/CD мають різну роль та функціональність у процесі CI/CD.

Системи контролю версій забезпечують централізоване зберігання та управління кодом, дозволяючи командам розробників ефективно співпрацювати, а також ведення історії змін, злиття коду, вирішення конфліктів, збереження версій та відновлення попередніх версій коду.

Сервери CI автоматизують процеси збірки та тестування коду, гарантуючи його консистентність (узгодженість та цілісність, відсутність внутрішніх протиріч) [23] та якість, а також запуск процесів збірки та тестування при кожному коміті у репозиторій, забезпечення зворотного зв’язку для команди.

Інструменти автоматизації спрощують процеси розгортання та управління інфраструктурою, забезпечуючи їх автоматизацію та стандартизацію, автоматизують розгортання, конфігурацію середовищ, управління ресурсами та інфраструктурою.

Інструменти для тестування забезпечують автоматизацію тестування, що дозволяє швидко виявляти та виправляти помилки, автоматизують виконання різних видів тестів (unit tests, integration tests, end-to-end tests) для перевірки функціональності та надійності коду.

Системи моніторингу та логування надають інструменти для відстеження стану додатків та інфраструктури, виявлення та аналізу проблєм, автоматизують збір та аналіз логів, моніторинг продуктивності та доступності сервісів, оповіщення про помилки та проблєми.

Хмарні платформи надають гнучку та масштабовану інфраструктуру для розгортання та управління додатками, автоматизують розгортання сервісів та додатків у хмарному середовищі, забезпечення масштабування, високої доступності та безпеки.

У табл. 1.5 представлена роль та функціональність основних технічних компонентів CI/CD, таких я системи контролю версій, сервери CI, інструменти автоматизації, інструменти для тестування, системи моніторингу та логування, а також хмарні платформи.

Таблиця 1.5 – Роль та функціональність основних компонентів CI/CD

Компонент	Роль	Функціональність
Системи контролю версій	Забезпечення централізованого зберігання та управління кодом	Ведення історії змін, злиття коду, вирішення конфліктів
Сервери CI	Автоматизація процесів збірки та тестування коду	Запуск процесів збірки та тестів при змінах у репозиторії
Інструменти автоматизації	Спрощення процесів розгортання та управління інфраструктурою	Автоматичне розгортання, конфігурація середовищ
Інструменти для тестування	Забезпечення автоматизації тестування коду	Виконання різних видів тестів для перевірки коду
Системи моніторингу та логування	Відстеження стану додатків та інфраструктури	Збір та аналіз логів, моніторинг продуктивності
Хмарні платформи	Надання гнучкої та масштабованої інфраструктури	Розгортання сервісів у хмарному середовищі

Таким чином, дана таблиця показує, як кожен з технічних компонентів сприяє загальному процесу CI/CD, забезпечуючи ефективність та автоматизацію на різних етапах розробки програмного забезпечення.

1.5 Автоматизоване тестування у CI/CD

Автоматизоване тестування є ключовим елементом в процесах CI/CD, оскільки воно забезпечує ряд переваг. Основні з них розглянуті нижче [24].

Підвищення якості та надійності продукту. Автоматизоване тестування дозволяє ретельно перевіряти кожну зміну, що вноситься у код. Це зменшує ймовірність проникнення помилок та багів у фінальний продукт, підвищуючи його якість та надійність.

Економія часу та ресурсів. На відміну від ручного тестування, автоматизоване тестування виконується значно швидше, забезпечуючи постійний контроль якості без значних витрат часу та людських ресурсів.

Підвищення швидкості розробки. Автоматизоване тестування дозволяє швидше виявляти та усувати помилки, що сприяє більш ефективному процесу розробки та скорочує час випуску нових функцій та оновлень.

Раннє виявлення проблєм. Завдяки автоматизації тестів, можливо виявляти потенційні проблєми на ранніх етапах розробки, що знижує ризики та вартість їх виправлення у майбутньому.

Підтримка відтворюваності та консистентності. Автоматизоване тестування забезпечує відтворюваність результатів, що є важливим для забезпечення стабільності та консистентності функціональності продукту.

Можливість неперервного поліпшення. Наявність надійних автоматизованих тестів сприяє експериментам і вдосконаленням у коді, оскільки розробники можуть бути впевнені в тому, що будь-які негативні зміни будуть швидко виявлені.

До можливих проблєм та викликів автоматизованого тестування належать: складність підготовки тестів, високі початкові витрати, підтримка та оновлення тестів, потенційні помилки в тестах, обмеження автоматизованих тестів, залежність від інструментів та технологій.

Розробка автоматизованих тестів може бути складним завданням, особливо для комплексних систем та додатків, оскільки вона вимагає глибокого розуміння бізнес-логіки та взаємодій між компонентами. Впровадження автоматизованих тестів також вимагає значних витрат часу та ресурсів на початкових етапах, оскільки необхідно написати та налагодити велику кількість тестових сценаріїв. Автоматизовані тести потребують регулярної підтримки та оновлення для забезпечення їх актуальності та ефективності, особливо у відповідь на зміни в коді додатка. Тести можуть містити помилки або некоректні припущення, що може призвести до хибного відчуття безпеки або не виявлення реальних проблєм. Деякі аспекти програмного забезпечення, такі як користувацький інтерфейс або взаємодія з зовнішніми системами, можуть бути важко автоматизувати, вимагаючи ручного тестування. Вибір та використання інструментів для автоматизації тестування можуть вплинути на ефективність процесу. Неправильний вибір інструментів може призвести до додаткових труднощів [25].

Отже, автоматизоване тестування, хоча і забезпечує значні переваги для CI/CD процесу, також має низку потенційних проблєм та викликів. Вирішення

цих проблєм вимагає ретельного планування, постійної уваги до деталей тестових сценаріїв та вибору відповідних інструментів та підходів.

Табл. 1.6 узагальнює переваги та можливі проблєми та виклики автоматизованого тестування.

Таблиця 1.6 – Переваги та проблєми автоматизованого тестування.

Переваги	Проблєми та виклики
Підвищення якості та надійності продукту	Складність підготовки тестів
Економія часу та ресурсів	Високі початкові витрати
Підвищення швидкості розробки	Підтримка та оновлення тестів
Раннє виявлення проблєм	Потенційні помилки в тестах
Підтримка відтворюваності та консистентності	Обмеження автоматизованих тестів
Можливість неперервного поліпшення	Залежність від інструментів та технологій

Відрізняють різні види тестування залежно від їх призначення та інтеграції у CI/CD [26, 27].

Unit Testing (модульне тестування) – тестування окремих модулів або компонентів програми для перевірки їх функціональності ізольовано від решти системи; автоматично виконується при кожному коміті для перевірки нових змін у коді, забезпечуючи високу якість окремих компонентів.

Integration Testing (інтеграційне тестування) – тестування взаємодії між різними модулями чи компонентами системи для визначення коректності їх спільної роботи; часто автоматизоване після модульного тестування для перевірки сумісності компонентів.

System Testing (системне тестування) – повне тестування інтегрованої системи для виявлення будь-яких невідповідностей між вимогами та реальним функціонуванням; виконується на пізніших етапах, перед розгортанням, для забезпечення готовності продукту до випуску.

End-to-End Testing (E2E-тестування) – тестування повного потоку або процесу в додатку від початку до кінця, імітуючи поведінку реальних користувачів; здійснюється перед випуском продукту, забезпечуючи, що всі компоненти системи працюють коректно разом.

Performance Testing (тестування продуктивності) – перевірка швидкодії, стабільності та масштабованості системи під навантаженням; часто виконується на останніх етапах або у виробничому середовищі для оцінки продуктивності системи.

Табл. 1.7 узагальнює різновиди тестувань та їх місце у пайплайні CI/CD.

Таблиця 1.7 – Види тестувань залежно від їх інтеграції у процесі CI/CD

Вид тестування	Опис	Інтеграція у CI/CD
Модульне тестування (Unit Testing)	Тестування окремих модулів або компонентів програми для перевірки їх функціональності ізольовано від решти системи.	Автоматично виконується при кожному коміті для перевірки нових змін у коді.
Інтеграційне тестування (Integration Testing)	Тестування взаємодії між різними модулями чи компонентами системи для визначення коректності їх спільної роботи.	Часто автоматизоване після модульного тестування для перевірки сумісності компонентів.
Системне тестування (System Testing)	Повне тестування інтегрованої системи для виявлення будь-яких невідповідностей між вимогами та реальним функціонуванням.	Виконується на пізніших етапах, перед розгортанням, для забезпечення готовності продукту до випуску.
E2E-тестування (End-to-End Testing)	Тестування повного потоку або процесу в додатку від початку до кінця, імітуючи поведінку реальних користувачів.	Здійснюється перед випуском продукту, забезпечуючи, що всі компоненти системи працюють коректно разом.
Тестування продуктивності (Performance Testing)	Перевірка швидкодії, стабільності та масштабованості системи під навантаженням.	Часто виконується на останніх етапах або у виробничому середовищі для оцінки продуктивності системи.

Табл. 1.7 представляє різні види тестувань, інтегрованих у процесі CI/CD, починаючи з модульного тестування, яке перевіряє окремі компоненти програми і виконується автоматично при кожному коміті, до інтеграційного тестування, яке оцінює взаємодію між компонентами системи. Системне тестування забезпечує повну перевірку інтегрованої системи, в той час як E2E-тестування імітує поведінку реальних користувачів, обидва вони здійснюються перед випуском продукту. Тестування продуктивності оцінює швидкодію та стабільність системи, зазвичай виконується на останніх етапах розробки або у виробничому середовищі.

Крім розглянутих вище, існують також інші види тестувань (табл. 1.8), які так само можуть бути інтегровані в процеси CI/CD.

Таблиця 1.8 – Види тестувань залежно від їх цілі у CI/CD

Види тестувань	Опис	Ціль
Тестування безпеки (Security Testing)	Перевірка програмного забезпечення на вразливості, загрози та ризики, що можуть призвести до несанкціонованого доступу або втрати даних.	Забезпечення захисту програми від зовнішніх та внутрішніх загроз.
Тестування сумісності (Compatibility Testing)	Перевірка, як програма працює у різних середовищах: різні операційні системи, браузери, версії обладнання тощо.	Забезпечення коректної роботи програми в різних умовах та середовищах.
Тестування користувацького інтерфейсу (UI Testing)	Перевірка елементів користувацького інтерфейсу: лейаути, кнопки, меню та інші візуальні елементи.	Забезпечення зручності та інтуїтивності інтерфейсу для користувача.
Тестування доступності (Accessibility Testing)	Перевірка програми на доступність для користувачів з обмеженими можливостями, такими як слабозорі або незрячі.	Забезпечення того, що програма доступна та зручна для всіх категорій користувачів.
Тестування локалізації (Localization Testing)	Перевірка програми на коректність перекладу, адаптацію до місцевих культурних особливостей, форматів даних (дата, час, валюта).	Забезпечення адекватної локалізації програми для різних регіонів.
Тестування безпеки (Security Testing)	Перевірка програмного забезпечення на вразливості, загрози та ризики, що можуть призвести до несанкціонованого доступу або втрати даних	Забезпечення захисту програми від зовнішніх та внутрішніх загроз.
Тестування сумісності (Compatibility Testing)	Перевірка, як програма працює у різних середовищах: різні операційні системи, браузери, версії обладнання тощо.	Забезпечення коректної роботи програми в різних умовах та середовищах.
Тестування користувацького інтерфейсу (UI Testing)	Перевірка елементів користувацького інтерфейсу: лейаути, кнопки, меню та інші візуальні елементи.	Забезпечення зручності та інтуїтивності інтерфейсу для користувача.
Тестування доступності (Accessibility Testing)	Перевірка програми на доступність для користувачів з обмеженими можливостями, такими як слабозорі або незрячі.	Забезпечення того, що програма доступна та зручна для всіх категорій користувачів.
Тестування локалізації (Localization Testing)	Перевірка програми на коректність перекладу, адаптацію до місцевих культурних особливостей, форматів даних (дата, час, валюта).	Забезпечення адекватної локалізації програми для різних регіонів.
Тестування навантаження (Load Testing)	Оцінка здатності програми витримувати певний об’єм роботи, імітування одночасної роботи багатьох користувачів або запитів до системи.	Забезпечення стабільності та продуктивності програми під час високих навантажень.

Продовження таблиці 1.8

Тестування стресу (Stress Testing)	Визначення меж здатності системи шляхом введення екстремальних умов або наднавантажень.	Перевірка реакції системи на критичні умови та визначення її меж витривалості.
Тестування вразливостей (Vulnerability Testing)	Ідентифікація потенційних слабких місць у безпеці програмного забезпечення.	Захист програми від можливих кібератак та зловмисного використання.
Регресійне тестування (Regression Testing)	Перевірка, що нові зміни у коді не порушили існуючу функціональність та не внесли нових помилок.	Забезпечення консистентності та надійності програми після кожного оновлення або зміни.
Тестування юзабіліті (Usability Testing)	Оцінка зручності та інтуїтивності користувацького інтерфейсу, взаємодії з користувачем.	Забезпечення високої якості користувацького досвіду.
Тестування надійності (Reliability Testing)	Визначення ступеню надійності та стабільності програмного продукту в різних умовах.	Гарантія, що програма здатна надійно працювати протягом тривалого часу.

Наступна табл. 1.9 надає загальний огляд різних типів тестувань, кожен з яких відіграє важливу роль у забезпеченні різних аспектів якості та функціональності програмного забезпечення.

Таблиця 1.9 – Типи тестувань

Тип тестування	Опис
Модульне тестування	Тестування окремих модулів або компонентів програми.
Інтеграційне тестування	Тестування взаємодії між різними компонентами.
Системне тестування	Повне тестування інтегрованої системи.
E2E-тестування	Тестування повного потоку або процесу в додатку.
Тестування продуктивності	Перевірка швидкодії та стабільності системи.
Тестування безпеки	Перевірка програми на вразливості та ризики.
Тестування сумісності	Перевірка працездатності у різних середовищах.
Тестування користувацького інтерфейсу	Перевірка елементів користувацького інтерфейсу.
Тестування доступності	Перевірка доступності для користувачів з обмеженнями.
Тестування локалізації	Перевірка коректності перекладу та адаптації.
Тестування навантаження	Оцінка здатності витримувати високе навантаження.
Тестування стресу	Тестування реакції на екстремальні умови.
Тестування вразливостей	Ідентифікація слабких місць у безпеці.
Регресійне тестування	Перевірка стабільності після змін у коді.
Тестування юзабіліті	Оцінка зручності користувацького інтерфейсу.
Тестування надійності	Визначення ступеню надійності програми.

1.6 Інструменти та платформи для реалізації CI/CD

На сьогодні, найбільшою популярністю у розробників користується низка онлайн інструментів та платформ для реалізації CI/CD, які використовуються на різних етапах процесу (рис. 1.4) [28, 29].

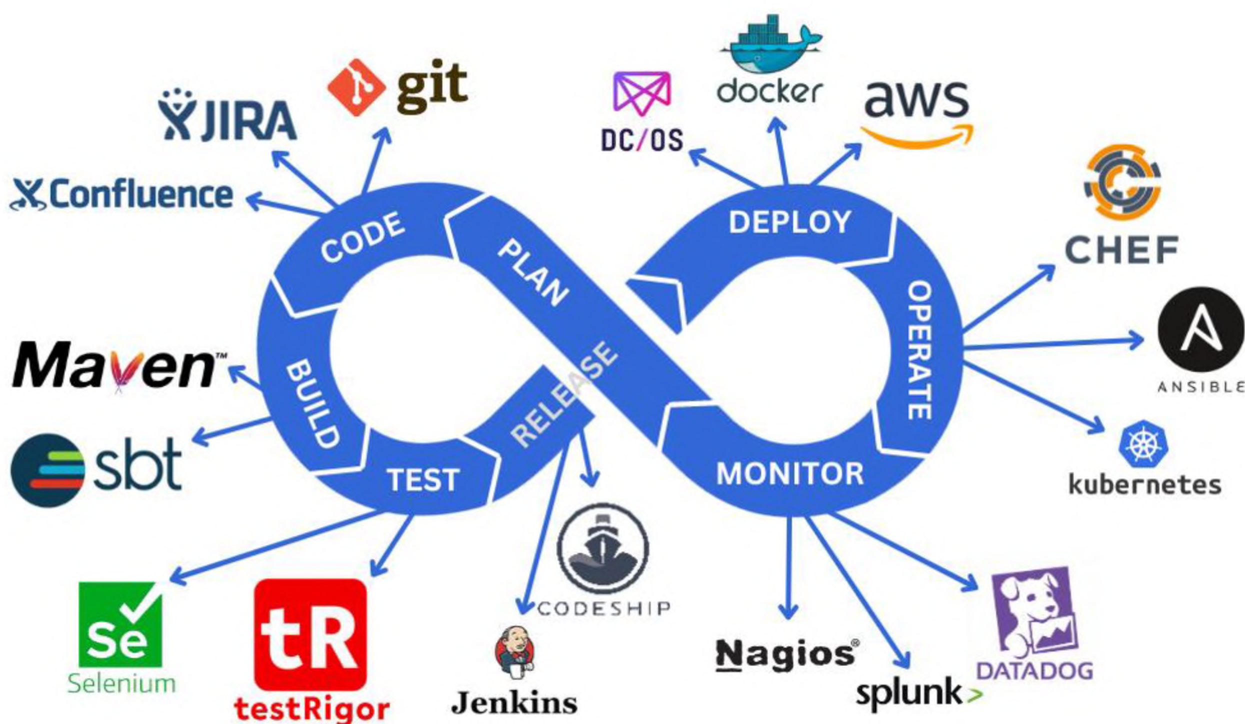

Рисунок 1.4 – Популярні інструменти CI/CD [30]

GitLab CI/CD – інтегрована система CI/CD, яка є частиною сервісу GitLab. Вона дозволяє автоматизувати етапи збірки, тестування та розгортання програмного забезпечення. Надає можливість визначення пайплайнів у файлах .gitlab-ci.yml, інтеграцію з GitLab репозиторіями, велику кількість вбудованих функцій та інструментів.

Jenkins – найбільш популярний відкритий інструмент для автоматизації різних завдань розробки, включаючи збірку, тестування та розгортання. Має велику спільноту, надає можливість розширення функціональності за допомогою плагінів, підтримує різні мови програмування та інструменти.

Travis CI – сервіс автоматизованого тестування та розгортання, який тісно інтегрований з GitHub. Легкий у налаштуванні, підтримує численні мови програмування, можливість легкої інтеграції з іншими сервісами.

CircleCI – платформа CI/CD, яка підтримує автоматизацію тестування та розгортання для проєктів на GitHub та Bitbucket. Надає можливість паралельного виконання завдань, легкість у налаштуванні, інтеграцію з багатьма популярними інструментами та сервісами [31].

Bamboo – інструмент CI/CD від Atlassian, який інтегрується з іншими продуктами цієї компанії, такими як Jira та Bitbucket. Підтримує різні мови програмування, надає можливість створення багатоступеневих пайплайнів, має глибоку інтеграцію з іншими інструментами Atlassian.

Табл. 1.10 у стислому вигляді представляє узагальнену інформацію про найбільш популярні інструменти та платформи CI/CD.

Таблиця 1.10 – Популярні інструменти та платформи CI/CD

Інструмент / платформа	Опис	Особливості
GitLab CI/CD	Інтегрована система CI/CD, яка є частиною сервісу GitLab.	Надає можливість визначення пайплайнів у файлах .gitlab-ci.yml, інтеграція з GitLab репозиторіями.
Jenkins	Найпопулярніший відкритий інструмент для автоматизації різних завдань розробки.	Має велику спільноту, надає можливість розширення функціональності за допомогою плагінів.
Travis CI	Сервіс автоматизованого тестування та розгортання, інтегрований з GitHub.	Легкий у налаштуванні, підтримує численні мови програмування.
CircleCI	Платформа CI/CD для проєктів на GitHub та Bitbucket.	Надає можливість паралельного виконання завдань, легкість у налаштуванні.
Bamboo	Інструмент CI/CD від Atlassian, інтегрований з Jira та Bitbucket.	Підтримка різних мов програмування, створення багатоступеневих пайплайнів.

Вибір інструменту або платформи CI/CD залежить від специфіки проєкту, потреб команди та бізнес-вимог. Для порівняння інструментів та платформ CI/CD можна використовувати різноманітні критерії, головними з яких є:

- сумісність з іншими інструментами та сервісами – наскільки легко інструмент або платформа інтегрується з іншими інструментами та сервісами, які вже використовуються (наприклад, системи контролю версій, хмарні платформи, інструменти моніторингу та логування тощо);

- гнучкість та масштабованість – важливо вибрати інструмент, який може адаптуватися до змін у проєкті та зростати разом з ним;

- спільнота та підтримка – активна спільнота та наявність підтримки можуть бути важливими факторами, особливо для вирішення складних проблєм та отримання нових оновлень;

- легкість використання та налаштування – інтуїтивно зрозумілий інтерфейс та простота налаштування значно спрощують роботу з інструментом, особливо для нових користувачів;

- функціональність та можливості – потрібно оцінити набір функцій, які пропонує інструмент, включаючи автоматизацію збірки, тестування, розгортання, підтримку різних мов програмування, інтеграцію з контейнерами тощо;

- вартість – бюджет є важливим, особливо для малих команд або стартапів. Деякі інструменти пропонують безкоштовні плани для невеликих проєктів або відкритого програмного забезпечення;

- безпека – важливо враховувати, як інструмент або платформа захищає ваш код та додатки, особливо при роботі з конфіденційними даними.

Додатково до вказаних критеріїв важливо звернути увагу на такі аспекти:

- швидкість виконання пайплайнів, Важливо оцінити, наскільки швидко інструмент або платформа може виконувати пайплайни CI/CD, особливо в умовах високого навантаження та для великих проєктів;

- інтегровані засоби тестування та аналізу коду, Деякі платформи CI/CD включають інтегровані інструменти для автоматичного тестування та аналізу якості коду, що може бути значною перевагою;

- гнучкість у налаштуванні процесів CI/CD – можливість налаштувати пайплайни відповідно до специфічних вимог проєкту та робочих процесів команди;

- вбудована підтримка контейнерізації та оркестрації. Для проєктів, що активно використовують контейнери (наприклад, Docker), важливо, щоб платформа CI/CD підтримувала контейнеризацію та мала можливості для оркестрації контейнерів;

- підтримка розгортання в різних середовищах. Здатність платформи CI/CD підтримувати розгортання в різноманітних середовищах, таких як різні хмарні провайдери, локальні сервери, і т.д.;

- сумісність з інструментами версіонування коду, Якщо проєкт використовує певні системи контролю версій, важливо, щоб обрана платформа CI/CD була сумісна з ними;

- можливості звітності та моніторингу. Інструменти, які надають детальні звіти про процеси збірки, тестування та розгортання, а також засоби для моніторингу статусу пайплайнів, можуть значно спростити управління процесами розробки;

- масштабування та обмеження ресурсів – можливість налаштувати використання ресурсів та масштабування пайплайнів залежно від потреб проєкту.

Табл. 1.11 та табл. 1.12 представляють порівняння популярних інструментів та платформ CI/CD за деякими основними критеріями [32].

Таблиця 1.11 – Порівняння інструментів та платформ CI/CD (частина 1)

Інструмент / платформа	Сумісність з іншими інструментами	Гнучкість та масштабованість	Спільнота та підтримка	Легкість використання
GitLab CI/CD	Висока (особливо з GitLab)	Висока	Висока	Середня
Jenkins	Висока (завдяки плагінам)	Висока	Дуже висока	Складна
Travis CI	Середня (оптимізована для GitHub)	Середня	Висока	Висока
CircleCI	Висока (оптимізована для GitHub та Bitbucket)	Висока	Висока	Висока
Bamboo	Висока (інтеграція з Jira та Bitbucket)	Середня	Середня	Середня

Таблиця 1.12 – Порівняння інструментів та платформ CI/CD (частина 2)

Інструмент / платформа	Функціональність та можливості	Вартість	Безпека
GitLab CI/CD	Висока	Безкоштовний план та комерційні плани	Висока
Jenkins	Висока	Безкоштовний (з відкритим кодом)	Середня (залежить від плагінів)
Travis CI	Середня	Безкоштовний план та комерційні плани	Висока
CircleCI	Висока	Безкоштовний план та комерційні плани	Висока
Bamboo	Висока	Комерційний	Висока

Ці таблиці допомагають зрозуміти ключові переваги та недоліки кожної платформи та інструменту, що спрощує вибір найбільш підходящого рішення для конкретних потреб та вимог проєкту.

Висновки до розділу 1

У даному розділі розглянуто ключові аспекти та еволюцію практик CI/CD, від історичного розвитку до сучасних тенденцій. Визначено основні концепції та термінологію, що лежать в основі CI/CD, включаючи неперервну інтеграцію, неперервну доставку та неперервне розгортання. Розглянуто переваги та виклики, пов’язані з впровадженням CI/CD, як ці практики сприяють ефективності розробки, якості продукту та швидкості доставки, а також які труднощі можуть виникати. Приділена увага ролі автоматизованого тестування у CI/CD, що є невід’ємною частиною для забезпечення високої якості та надійності програмного забезпечення. Детально розглянуто інструменти та платформи, які використовуються для реалізації CI/CD, порівнявши їх основні характеристики та можливості. Показано, що CI/CD є важливою складовою сучасного процесу розробки програмного забезпечення, яка дозволяє командам ефективно реагувати на зміни вимог, забезпечувати постійну якість та швидко впроваджувати нові функції в продукт. Опанування практик CI/CD та вибір підходящих інструментів є

ключовими для успішної розробки та підтримки програмних продуктів у динамічному світі сучасних технологій.

Підсумок основних теоретичних аспектів CI/CD, розглянутих у розділі.

Історія та розвиток CI/CD. Було висвітлено еволюцію практик CI/CD від ранніх етапів до сучасних методів. Це дозволило зрозуміти, як змінилися підходи до розробки та доставки програмного забезпечення відповідно до викликів часу та технологічного прогресу.

Основні концепції та термінологія. Були роз’яснені ключові терміни та концепції, такі як неперервна інтеграція, неперервна доставка та неперервне розгортання, які є фундаментом CI/CD.

Переваги та виклики. Обговорено переваги, такі як поліпшення якості продукту, зменшення часу на розробку та доставку, а також виклики, включаючи технічні та організаційні аспекти, що виникають під час впровадження CI/CD.

Ключові компоненти CI/CD. Визначено та описано ключові компоненти CI/CD, включаючи системи контролю версій, сервери CI, інструменти автоматизації та інші інструменти, що сприяють автоматизації процесу розробки.

Роль автоматизованого тестування у CI/CD. Підкреслено значення автоматизованого тестування для забезпечення надійності та якості програмного продукту у процесах CI/CD.

Інструменти та платформи для CI/CD. Оглянуто різноманітні інструменти та платформи, що використовуються для реалізації CI/CD, їх порівняння та вибір відповідно до потреб проєкту.

Сучасний ринок програмного забезпечення швидко розвивається, тому важливість ефективних методологій розробки є ключовою. Наступний розділ присвячений детальному вивченню CI/CD у контексті розподіленої розробки програмного забезпечення. Коли команда розробників працює віддалено, потреба у злагодженому, автоматизованому та ефективному процесі розробки стає ще більш актуальною. CI/CD об’єднує розробників, процеси та технології, забезпечуючи неперервність процесів інтеграції, тестування, доставки та розгортання продукту.

РОЗДІЛ 2
МЕТОДОЛОГІЯ CI/CD У РОЗПОДІЛЕНІЙ РОЗРОБЦІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Актуальні тенденції у CI/CD та їх вплив на розробку

Актуальні тенденції у галузі CI/CD суттєво впливають на розробку програмного забезпечення [33].

Інтеграція CI/CD з хмарними технологіями. Сучасні хмарні сервіси, як-от AWS, Azure, та Google Cloud, активно інтегруються з CI/CD. Це сприяє гнучкості, масштабуванню та оптимізації процесів розгортання. Використання хмарних платформ дозволяє розробникам швидше та ефективніше розгортати додатки, а також забезпечує більшу стабільність та доступність продукту.

Контейнеризація та оркестрація. Популярність контейнерів, зокрема Docker, та оркестраційних систем, як-от Kubernetes, росте. Ці технології покращують консистентність середовища та автоматизацію процесів. Контейнеризація спрощує управління залежностями та розгортання, а також сприяє DevOps-підходам у розробці.

Мікросервісна архітектура. Розробка на основі мікросервісів набирає популярності, що вимагає адаптації CI/CD процесів під численні незалежні компоненти. Мікросервіси забезпечують більшу гнучкість та незалежність команд, але водночас потребують складнішого управління збіркою та розгортанням.

Автоматизація та інтелектуальне тестування. Розвиток інструментів автоматичного тестування, включаючи використання штучного інтелекту для оптимізації процесів тестування. Автоматизація тестування підвищує якість та надійність продуктів, знижуючи ризик помилок та скорочуючи час розробки.

Безпека в CI/CD (DevSecOps). Інтеграція практик безпеки безпосередньо в процеси CI/CD з метою забезпечення безпеки на всіх етапах розробки.

Зосередження на безпеці з самого початку розробки зменшує ризики та витрати, пов'язані з виправленням вразливостей на пізніших етапах.

Ці тенденції показують, що галузь CI/CD постійно розвивається, адаптуючись до нових викликів та можливостей. Вони значно впливають на підходи та методології в розробці програмного забезпечення, вимагаючи від розробників бути гнучкими та відкритими до нововведень.

2.2 Gitlab як інструмент для автоматизації процесу розробки

GitLab – це вебплатформа для управління життєвим циклом програмного забезпечення, яка об'єднує в собі інструменти для співпраці у розробці, систему контролю версій, управління завданнями та помилками, а також автоматизацію процесів CI/CD. GitLab допомагає розробникам працювати над спільними проєктами, від початкової стадії до кінцевого розгортання продукту [34].

Основним компонентом GitLab є система контролю версій на базі Git, що дозволяє користувачам створювати репозиторії, відстежувати зміни в коді, об'єднувати ветки та співпрацювати над спільними проєктами. Крім того, GitLab пропонує функції трекінгу завдань, код-рев'ю, управління випусками та моніторингу. GitLab є популярним серед команд розробників завдяки його гнучкості, масштабованості та інтегрованому підходу до управління проєктами та автоматизації CI/CD [35].

У GitLab використовуються декілька ключових понять, які важливі у налаштуванні та управлінні CI/CD процесами в середовищі GitLab [36-37].

Пайплайн (pipeline) – сукупність процесів та автоматизованих завдань, що виконуються при кожному коміті або злитті в репозиторій. Включає етапи збірки, тестування, розгортання та інші.

Етап (stage) – частина пайплайну, яка визначає групу завдань, що виконуються послідовно. Наприклад, етапи build, test, deploy.

Завдання (job) – окремі дії або скрипти, які виконуються на певному етапі пайплайну. Кожне завдання визначається в .gitlab-ci.yml.

.gitlab-ci.yml – конфігураційний файл, що використовується для визначення пайплайнів CI/CD у GitLab. Він містить інструкції щодо етапів, завдань та їх виконання.

Артефакти (artifacts) – файли або директорії, які генеруються та зберігаються після виконання завдань. Використовуються для передачі даних між етапами пайплайну.

Кеш (cache) – механізм для зберігання та використання залежностей та інших файлів між виконаннями завдань чи пайплайнів, що сприяє прискоренню процесу збірки.

Змінні середовища (environment variables) – змінні, які використовуються для зберігання специфічної інформації, такої як токени доступу, налаштування конфігурації та інше.

Середовища (environments) – використовуються для управління процесами розгортання в різних умовах, наприклад, в тестових, стейджингових або продакшн середовищах.

Вебхуки (webhooks) – сповіщення, які надсилаються на зовнішні сервіси або інструменти при певних подіях в репозиторії або пайплайні.

Розглянемо основні можливості та функціонал Gitlab. Система контролю версій. GitLab забезпечує повноцінну підтримку системи контролю версій Git, дозволяючи керувати кодом, ветками, злиттями та запитами на злиття (merge requests) [38-43].

Управління проєктами та задачами. Платформа Gitlab пропонує інструменти для управління проєктами, включаючи трекери завдань, діаграми Ганта, дорожні карти та систему відстеження помилок.

CI/CD пайплайни. GitLab CI/CD надає можливості для автоматизації процесів збірки, тестування та розгортання програмного забезпечення, забезпечуючи ефективність та швидкість розробки.

Контейнеризація та DevOps. Інтеграція Gitlab з Docker та Kubernetes сприяє використанню контейнерів і оркестрації контейнерів, підвищуючи гнучкість та масштабованість.

Безпека та якість коду. Вбудовані інструменти Gitlab для аналізу коду, ідентифікації вразливостей та автоматичного сканування залежностей допомагають підвищити безпеку та якість продукту.

Співпраця та рев'ю коду. Функції обговорення коду, спільного рев'ю та управління доступом сприяють зручній та ефективній командній роботі.

Інтеграція зі сторонніми сервісами. GitLab підтримує інтеграцію з багатьма популярними сервісами та інструментами, що розширює його функціональність та адаптованість.

Відкритий код та кастомізація. Будучи продуктом з відкритим кодом, GitLab дозволяє користувачам налаштовувати та розширювати його функціональність відповідно до специфічних потреб.

Табл. 2.1 представляє огляд можливостей та основного функціоналу GitLab.

Таблиця 2.1 – Основний функціонал GitLab

Функціонал GitLab	Опис
Система контролю версій	Підтримка системи контролю версій Git для керування кодом, ветками та злиттями.
Управління проєктами та задачами	Інструменти для управління проєктами, включаючи трекери завдань, діаграми Ганта, дорожні карти.
CI/CD пайплайни	Автоматизація процесів збірки, тестування та розгортання за допомогою вбудованих пайплайнів.
Контейнеризація та DevOps	Інтеграція з Docker та Kubernetes для використання контейнерів і оркестрації.
Безпека та якість коду	Вбудовані інструменти для аналізу коду та ідентифікації вразливостей.
Співпраця та рев'ю коду	Функції для обговорення коду, спільного рев'ю та управління доступом.
Інтеграція зі сторонніми сервісами	Підтримка інтеграції з численними зовнішніми сервісами та інструментами.
Відкритий код та кастомізація	Можливість налаштування та розширення функціональності, відкритий код.

На рис. 2.1 представлено робочий процес, який показує основні етапи процесу CI/CD у GitLab. Причому, при використанні GitLab не потрібні зовнішні

інструменти для доставки програмного забезпечення, і всі кроки робочого процесу можна візуалізувати в інтерфейсі користувача GitLab.

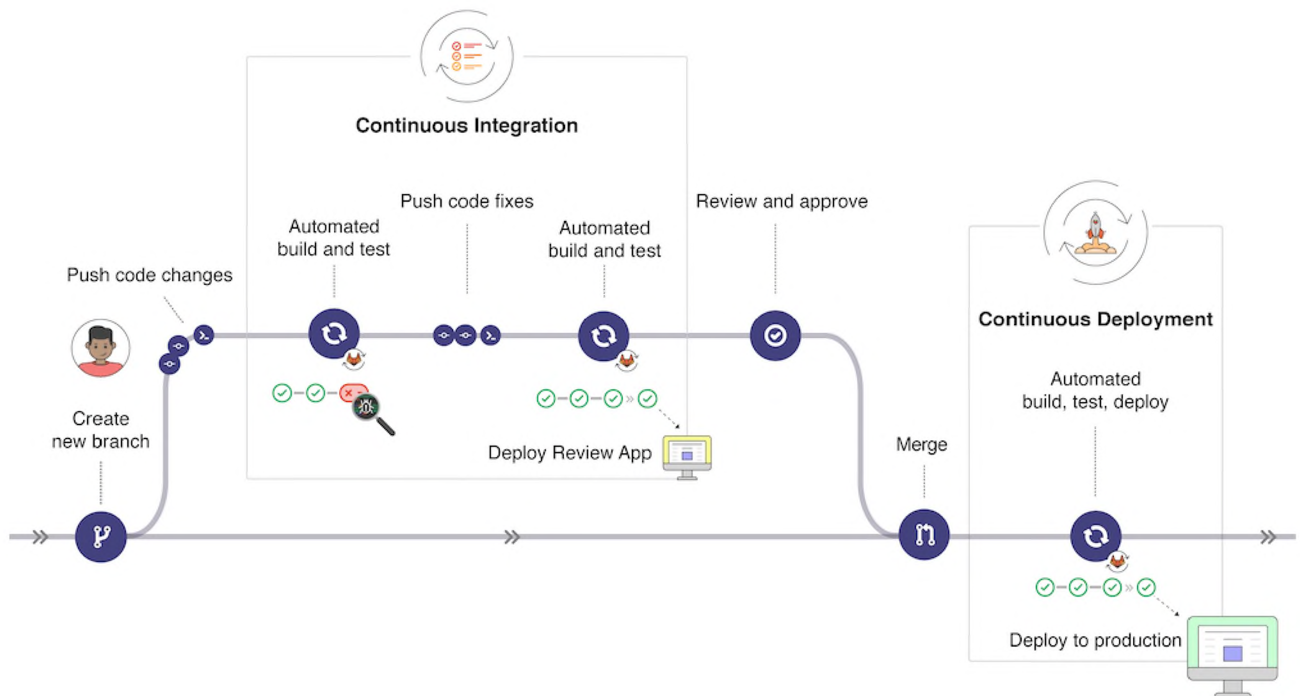

Рисунок 2.1 – Основні етапи процесу CI/CD у GitLab [1]

GitLab, як комплексна платформа DevOps [44], пропонує широкий спектр функцій, що охоплюють усі етапи життєвого циклу DevOps, зокрема це [45]:

- планування – дошки завдань (Issue Boards), візуалізація процесів роботи над проектом за допомогою гнучких дощок завдань; відстеження проблем (Issue Tracking), менеджмент завдань та проблем, їх класифікація та пріоритизація;

- створення – Git-репозиторій, зберігання коду, ведення версій та ревізій; merge requests (запити на злиття), інтеграція змін та співпраця над кодом;

- перевірка – CI, автоматизація збірки, тестування та інших перевірок коду; code review (перегляд коду), аналіз та обговорення змін у коді до їх інтеграції;

- розгортання – CD, автоматизація процесу розгортання коду на різних середовищах; environments and deployments, управління середовищами та відстеження розгортань;

- експлуатація – моніторинк, відстеження стану застосунків та інфраструктури; incident management, управління інцидентами та їх вирішення;

- моніторинг та аналітика – metrics and dashboards, візуалізація ключових показників ефективності за допомогою дашбордів; DevOps reports, звіти щодо ефективності DevOps-процесів;

- захист – security scanning, сканування коду на наявність вразливостей; container scanning, аналіз контейнерів на наявність вразливостей.

GitLab інтегрує ці інструменти та функції в єдиний потік роботи, що забезпечує плавність та ефективність DevOps-циклу, від ідеї до впровадження.

На діаграмі (рис. 1.5) представлені функції, доступні в GitLab на кожному етапі життєвого циклу DevOps.

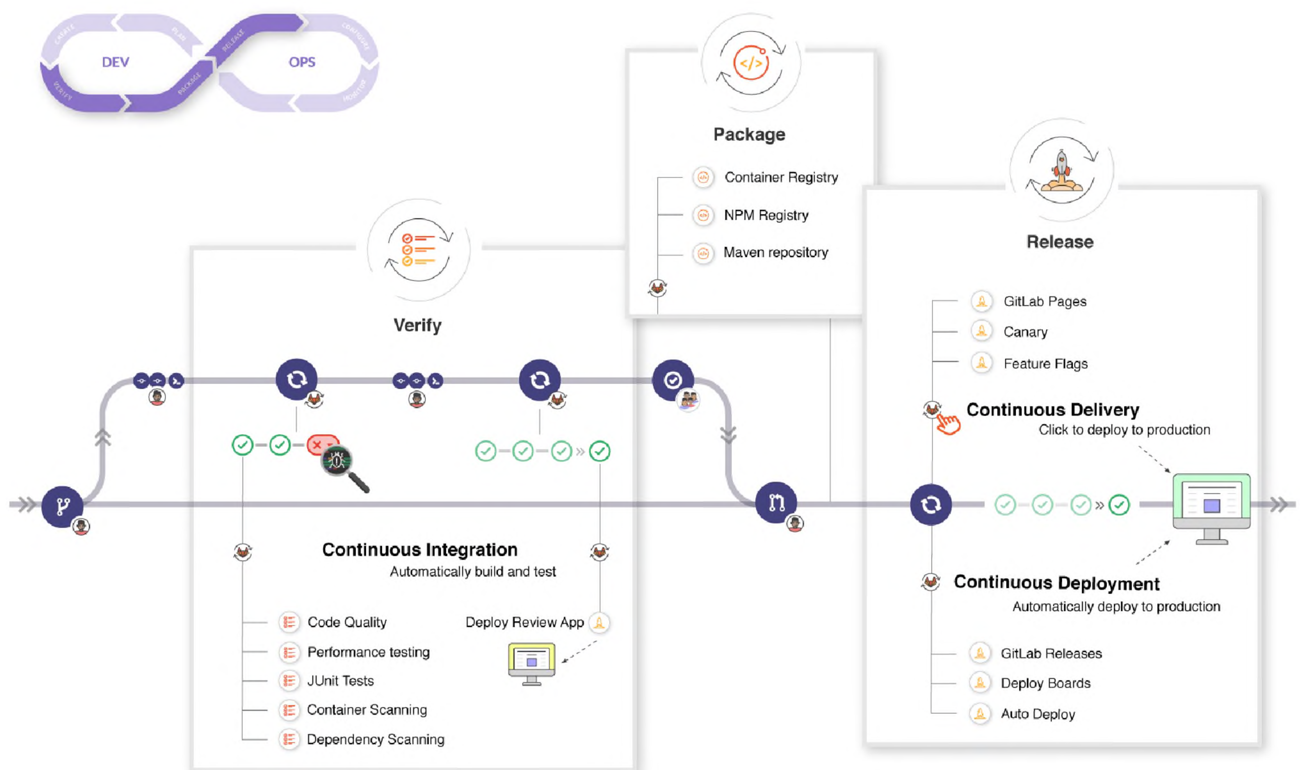

Рисунок 1.5 – Функції, доступні в GitLab на кожному етапі DevOps [1]

Таким чином, GitLab – це багатофункціональний інструмент, який об’єднує в собі різноманітні можливості для ефективної розробки, управління проєктами, автоматизації CI/CD та співпраці в команді.

2.3 Методи та алгоритми налаштування CI/CD для Gitlab

Ефективний та надійний пайплайн CI/CD у GitLab є ключовим для успішного управління проєктами розробки програмного забезпечення.

Налаштування пайплайну CI/CD у GitLab представляє собою доволі складний процес, оскільки воно вимагає розуміння різних аспектів автоматизації та інтеграції. Загальні рекомендації щодо організації процесу розробки CI/CD представлені на рис. 2.3.

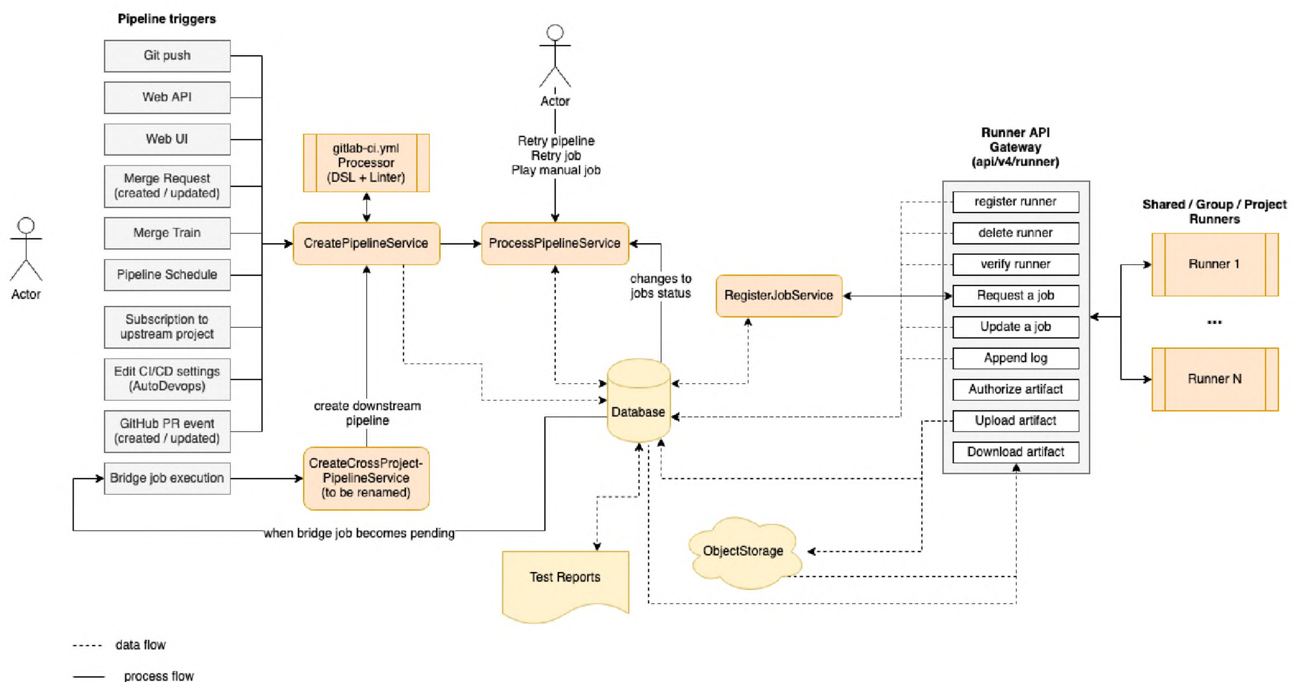

Рисунок 2.3 – Рекомендації GitLab щодо розробки CI/CD [39]

Діаграма на рис. 2.3, яка ілюструє використання різних методів та алгоритмів GitLab CI/CD, включає в себе кілька ключових компонентів та етапів, що забезпечують ефективне виконання процесу неперервної інтеграції та розгортання. Ця діаграма також включає взаємозв'язки та взаємодію між різними компонентами та етапами CI/CD, показуючи, як вони спільно сприяють плавному та ефективному процесу розробки та розгортання додатків у GitLab.

Основні алгоритми GitLab CI/CD представлені у табл. 2.2.

Таблиця 2.2 – Основні алгоритми GitLab CI/CD.

Алгоритм	Опис	Методи
Створення та налаштування .gitlab-ci.yml	Ключовий елемент налаштування CI/CD в GitLab, що містить визначення пайплайнів та завдань.	Визначення етапів роботи, налаштування скриптів, використання змінних.
Автоматизація тестування	Важлива частина CI/CD для перевірки якості коду.	Налаштування скриптів для тестів, інтеграція з інструментами тестування.
Конфігурація розгортання	Автоматичне розміщення продукту у виробниче або інші середовища.	Використання скриптів розгортання, налаштування роботи з середовищами.
Інтеграція з зовнішніми сервісами	Зв'язок GitLab CI/CD з іншими інструментами та сервісами.	Використання Webhooks та API, налаштування сповіщень.
Оптимізація та масштабування	Підтримка ефективності та масштабування пайплайнів.	Кешування, паралелізація завдань, налаштування правил запуску.

Наступна табл. 2.3 містить опис різних підходів до налаштування CI/CD у середовищі Gitlab.

Таблиця 2.3 – Основні підходи до налаштувань CI/CD в середовищі Gitlab

Налаштування	Опис	Підходи
Створення та налаштування .gitlab-ci.yml	Ключовий файл для визначення пайплайнів та завдань CI/CD.	Визначення етапів роботи, скриптів, змінних та умов запуску.
Автоматизація тестування	Автоматичне виконання тестів для забезпечення якості коду.	Налаштування скриптів для автоматичного тестування, інтеграція з інструментами тестування.
Конфігурація розгортання	Налаштування процесів для розгортання програмного забезпечення.	Використання скриптів для розгортання, налаштування середовищ розгортання.
Інтеграція з зовнішніми сервісами	Зв'язок GitLab CI/CD з хмарними сервісами, базами даних тощо.	Використання Webhooks, API для інтеграції, налаштування сповіщень.
Оптимізація та масштабування	Підвищення продуктивності та ефективності пайплайнів.	Кешування, паралелізація завдань, налаштування правил запуску.

Табл. 2.3 надає огляд основних підходів до налаштувань CI/CD в середовищі GitLab, які включають створення та налаштування файлу .gitlab-ci.yml для визначення етапів пайплайнів, автоматизацію тестування з метою забезпечення якості коду, конфігурацію процесів розгортання програмного забезпечення, інтеграцію з зовнішніми сервісами, такими як хмарні платформи та бази даних, а також оптимізацію та масштабування пайплайнів через кешування,

паралелізацію завдань та налаштування правил запуску. Ці підходи спрямовані на підвищення продуктивності та ефективності процесів CI/CD.

2.4 Конфігурація CI/CD у Gitlab

Загальний алгоритм використання GitLab CI/CD з погляду користувача системи включає кілька кроків, які відображає діаграма, що представлена на рис. 2.4.

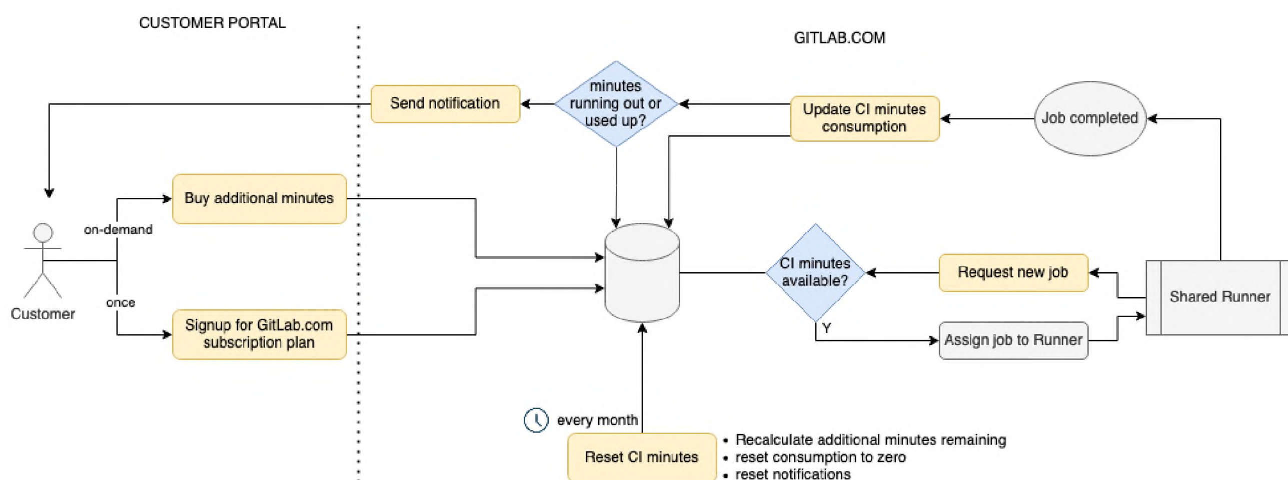

Рисунок 2.4 – Загальний алгоритм використання GitLab CI/CD [39]

Для визначення та управління процесами CI та CD у GitLab, для кожного проєкту створюється свій конфігураційний файл .gitlab-ci.yml, що знаходиться в кореневій директорії проєкту, і містить інструкції для GitLab CI/CD про те, як збирати, тестувати та розгортати кодс[46, 47].

Файл .gitlab-ci.yml повинен описує такі головні елементи (блоки):

1. Етапи пайплайну – тут визначаються різні етапи (stages) у пайплайні, такі як збірка, тестування та розгортання;

2. Завдання (jobs) – тут описуються завдання кожного етапу. Кожен етап може містити одне або декілька завдань, які виконують певні дії, наприклад, запуск тестів або збірка проєкту;

3. Скрипти – тут для кожного завдання можна вказати скрипти, які автоматично виконують необхідні команди;

4. Залежності та артефакти – тут описуються залежності між завданнями та визначаються артефакти, які потрібно зберегти після виконання кожного завдання;

5. Змінні та середовища – файл .gitlab-ci.yml дозволяє визначати змінні, які можуть бути використані у скриптах, та налаштувати середовища для різних стадій розгортання.

Отже, загальна структура файлу .gitlab-ci.yml для GitLab CI/CD включає наступні розділи, що мають певне призначення [48]:

stages – тут визначається послідовність етапів (stages) в пайплайні, таких як build, test, deploy;

jobs – кожне завдання визначається під окремим ім’ям та може включати скрипти, які виконуються на певному етапі;

script – команди, які виконуються в рамках завдання;

artifacts – визначає файли чи директорії, які потрібно зберегти після виконання завдання;

only/except – описуються умови, які визначають, коли завдання має виконуватися;

variables – змінні середовища, які можна використовувати в скриптах.

Приклад коду файлу .gitlab-ci.yml:

```
stages:
  - build
  - test
  - deploy
build_job:
  stage: build
  script:
    - echo "Збірка проєкту"
    - build_script
```

```
test_job:
  stage: test
  script:
    - echo "Виконання тестів"
    - test_script
  artifacts:
    paths:
      - test_reports/
deploy_job:
  stage: deploy
  script:
    - echo "Розгортання проєкту"
    - deploy_script
  only:
    - master
```

У цьому прикладі: визначено три етапи: build, test, deploy; для кожного етапу створено по одному завданню (build_job, test_job, deploy_job); у кожному завданні вказані команди, які виконуються, зокрема, у завданні test_job використовуються артефакти для збереження результатів тестування. Завдання deploy_job виконується тільки для гілки master.

Висновки до розділу 2

Розглянуто сучасні тенденції в галузі CI/CD, такі як інтеграція з хмарними сервісами, контейнеризація, мікросервісна архітектура, автоматизація тестування та DevSecOps. Встановлено, що ці тенденції значно впливають на методи та підходи в розробці, забезпечуючи більшу гнучкість, надійність та безпеку продукту.

Показано, що GitLab виступає як потужний інструмент для автоматизації процесів розробки, забезпечуючи інтеграцію з системою контролю версій, управлінням проєктами та CI/CD пайплайнами. Його функціональність і гнучкість роблять GitLab ключовим інструментом у сучасних процесах розробки.

Описано різні методи та алгоритми для оптимізації та налаштування CI/CD процесів у GitLab, включаючи оптимізацію та масштабування пайплайнів, кешування, паралелізацію завдань. Ці методи сприяють ефективності розробки, зменшуючи час збірки та розгортання.

Детально описано процес конфігурації CI/CD в GitLab, включаючи створення та налаштування .gitlab-ci.yml, автоматизацію тестування, конфігурацію розгортання та інтеграцію з зовнішніми сервісами. Це дозволяє підвищити ефективність та автоматизацію розробки.

Загалом, охоплено широкий спектр аспектів CI/CD у контексті використання GitLab, продемонстровано їх значущість та вплив на сучасні методики розробки програмного забезпечення. Зроблено висновок про те, що забезпечення автоматизації, гнучкості та безпеки в процесах CI/CD є важливим для досягнення високої продуктивності та якості у розробці програмного забезпечення.

РОЗДІЛ 3
ПРАКТИЧНІ АСПЕКТИ НАЛАШТУВАННЯ АЛГОРИТМІВ CI/CD ДЛЯ GITLAB-РЕПОЗИТОРІЇВ

3.1 Створення базового пайплайну CI/CD у Gitlab

Для створення базового пайплайну CI/CD в GitLab необхідно описати його основні етапи: збірку, тестування та розгортання. Це робиться за допомогою конфігураційного файлу .gitlab-ci.yml, який розміщується в кореневій директорії репозиторію. Далі розглянемо основні етапи налаштування [49].

Етап збірки (build). На цьому етапі відбувається компіляція коду та підготовка виконуваних файлів або артефактів. Це основа для подальшого тестування та розгортання.

Етап тестування (test). На цьому етапі проводиться автоматичне тестування коду. Це можуть бути юніт-тести, інтеграційні тести або інші види автоматизованих тестів.

Етап розгортання (deploy). Після успішного тестування коду відбувається його розгортання. Це може бути розгортання на тестовому сервері, стейджингу або у продакшн.

Приклад коду файлу .gitlab-ci.yml.

```
stages:
  - build
  - test
  - deploy
build_job:
  stage: build
  script:
    - echo "Виконання збірки проєкту"
    - build_command
test_job:
```

```
  stage: test
  script:
    - echo "Виконання тестів"
    - test_command
deploy_job:
  stage: deploy
  script:
    - echo "Розгортання проєкту"
    - deploy_command
  only:
    - master
```

У цьому прикладі: у файлі .gitlab-ci.yml визначено три основних етапи пайплайну CI/CD: build, test та deploy; build_job, test_job та deploy_job – це завдання, які виконуються на відповідних етапах; кожне завдання містить набір команд (script), які виконуються на цьому етапі. Наприклад, build_command, test_command та deploy_command – це плейсхолдери для реальних команд збірки, тестування та розгортання; у завданні deploy_job використовується директива only, щоб розгортання відбувалося лише при комітах у гілку master.

3.2 Інтеграція тестування та аналізу коду

Автоматизоване тестування – ключова частина процесу CI/CD, яка дозволяє забезпечити якість коду та запобігти регресії функціоналу. У GitLab CI/CD воно виконується через налаштування відповідних завдань у файлі .gitlab-ci.yml у такий спосіб.

Налаштування етапу тестування. Перш за все, потрібно визначити етап тестування у загальному пайплайні CI/CD:

```
stages:
  - build
```

```
  - test
  - deploy
```

Тут test – це етап, на якому будуть виконуватися тести.

Конфігурація завдань тестування. Далі створюємо завдання для тестування, яке буде виконуватися на етапі test:

```
test_job:
  stage: test
  script:
    - echo "Запуск тестів"
    - run_tests_command
```

У цьому коді run_tests_command – це команда або скрипт, який запускає тести, яка може відрізнятися в залежності від мови програмування та фреймворку.

Проілюструємо прикладами конкретних команд тестування.

1. Для Node.js/JavaScript проєктів із використанням фреймворку Jest:

```
test_job:
  stage: test
  script:
    - echo "Запуск тестів Jest"
    - npm install
    - npm test
```

2. Для Python проєктів із використанням pytest:

```
test_job:
  stage: test
  script:
    - echo "Запуск тестів pytest"
    - pip install -r requirements.txt
    - pytest
```

Налаштування збереження артефактів та звітів тестування. GitLab CI/CD дозволяє зберігати артефакти, наприклад, звіти тестування:

```
test_job:
```

```
  stage: test
  script:
    - run_tests_command
  artifacts:
    paths:
      - test_reports/
```

У цьому коді результати тестування будуть зберігатися в директорії test_reports/.

Ці налаштування дозволяють автоматизувати процес тестування в GitLab CI/CD, що є важливим для підтримки високої якості коду та ефективної розробки.

Інтеграція інструментів статичного аналізу коду в GitLab CI/CD забезпечує автоматичне виявлення потенційних проблєм на ранніх етапах розробки, що підвищує якість коду та сприяє ефективності процесів розробки.

Загальні засади інтеграції. Інструменти статичного аналізу коду використовуються для виявлення потенційних помилок, вразливостей та неефективних практик написання коду без його виконання. Їх інтеграція з CI/CD дозволяє автоматично виконувати перевірку коду на кожному етапі розробки.

Налаштування завдання для статичного аналізу коду. Статичний аналіз коду зазвичай включається як частина етапу тестування в пайплайні CI/CD:

```
stages:
  - build
  - test
  - deploy
static_analysis:
  stage: test
  script:
    - echo "Запуск статичного аналізу коду"
    - static_analysis_command
```

У цьому коді static_analysis_command – це команда або скрипт, який запускає статичний аналіз.

Приклади інтеграції конкретних інструментів для статичного аналізу коду.

1. Для JavaScript проєктів з ESLint:

```
static_analysis:
  stage: test
  script:
    - echo "Запуск ESLint"
    - npm install
    - ./node_modules/.bin/eslint .
```

2. Для Python проєктів з Pylint:

```
static_analysis:
  stage: test
  script:
    - echo "Запуск Pylint"
    - pip install pylint
    - pylint my_module.py
```

Налаштування збереження результатів аналізу. Результати статичного аналізу можуть бути збережені як артефакти для подальшого використання:

```
static_analysis:
  stage: test
  script:
    - static_analysis_command
  artifacts:
    paths:
      - static_analysis_reports/
```

3.3 Автоматизація розгортання та управління середовищами

В GitLab CI/CD використовується поняття стратегій розгортання (deploy strategies) – це підходи до випуску та оновлення програмного забезпечення у

виробничому середовищі. Вони визначають, як і коли оновлення будуть застосовані, щоб мінімізувати перерви у роботі та забезпечити безпеку. Стратегії розгортання дозволяють контролювати процес оновлення програмного забезпечення, зменшуючи ризики та забезпечуючи стабільність виробничого середовища. Вибір конкретної стратегії залежить від вимог проєкту, доступності ресурсів та прийняття рішень щодо рівня ризику. Розглянемо основні стратегії розгортання.

Стратегія «Rolling Update» – це плавне оновлення коду, при якому нова версія програми поступово замінює стару без перерв у роботі. Приклад:

```
deploy_job:
  stage: deploy
  script:
    - echo "Запуск Rolling Update"
    - rolling_update_command
  only:
    - master
```

У цьому прикладі rolling_update_command – це команда, яка виконує плавне оновлення.

Стратегія «Blue-Green Deployment» передбачає наявність двох ідентичних середовищ: «Blue» (старе) та «Green» (нове). Оновлення спочатку відбувається в «Green», після чого трафік перемикається з «Blue» на «Green». Приклад:

```
blue_green_deploy:
  stage: deploy
  script:
    - echo "Запуск Blue-Green Deployment"
    - deploy_to_green
    - switch_traffic_to_green
  only:
    - master
```

Стратегія «Canary Releases» передбачає випуск оновлення для невеликої частини користувачів спочатку, щоб перевірити стабільність та прийнятність оновлення перед його повним випуском. Приклад:

```
canary_release:
  stage: deploy
  script:
    - echo "Запуск Canary Release"
    - deploy_canary_version
    - monitor_and_expand
  only:
    - master
```

Налаштування різних середовищ (development, staging, production). У розробці програмного забезпечення часто використовуються різні середовища, такі як розробницьке (development), тестувальне (staging) та продуктивне (production). Кожне середовище має свої особливості та призначення. Налаштування середовищ дозволяє забезпечити адекватний рівень тестування та перевірки перед фінальним розгортанням програмного забезпечення, сприяє зниженню ризиків та підвищенню якості кінцевого продукту.

Середовище розробки (Development) – це первинне середовище, де розробники працюють над новими функціями та виправленнями. Приклад налаштування:

```
development_deploy:
  stage: deploy
  script:
    - echo "Розгортання у розробницьке середовище"
    - deploy_to_dev
  only:
    - branches
```

Тут deploy_to_dev – команда для розгортання у розробницьке середовище. Зазвичай використовується для гілок, крім master чи staging.

Тестувальне середовище (Staging) – використовується для передпродуктивного тестування та перевірки функціоналу. Приклад налаштування:

```
staging_deploy:
  stage: deploy
  script:
    - echo "Розгортання у тестувальне середовище"
    - deploy_to_staging
  only:
    - staging
```

Тут deploy_to_staging – команда для розгортання у тестувальне (staging) середовище, яка виконується, наприклад, тільки для гілки staging.

Продуктивне середовище (Production) – це фінальне середовище, де програмне забезпечення доступне кінцевим користувачам. Приклад налаштування:

```
production_deploy:
  stage: deploy
  script:
    - echo "Розгортання у продуктивне середовище"
    - deploy_to_production
  only:
    - master
```

Тут deploy_to_production – команда для розгортання у продуктивне середовище, яка виконується для гілки master.

3.4 Налаштування кешування та оптимізація пайплайнів

Налаштування кешування у GitLab CI/CD. Кешування в GitLab CI/CD використовується для тимчасового зберігання залежностей та інших важливих

файлів між виконаннями пайплайнів. Це дозволяє уникнути повторного завантаження або збірки тих самих залежностей, що зменшує час збірки проєкту. Використання кешування в GitLab CI/CD зменшує час збірки проєкту, зменшуючи час завантаження залежностей та компіляції, підвищує загальну продуктивність процесу CI/CD.

Налаштування кешу в .gitlab-ci.yml. Для налаштування кешування потрібно додати секцію cache у файл .gitlab-ci.yml. Далі розглянемо практичне налаштування кешу для Node.js проєкту та Python проєкту.

1. Приклад налаштування кешування для Node.js проєкту:

```
cache:
  paths:
    - node_modules/
build_job:
  stage: build
  script:
    - npm install
    - npm run build
```

У цьому прикладі каталог node_modules, де зберігаються залежності Node.js, кешується між виконаннями пайплайнів.

2. Приклад налаштування кешуванні для Python проєкту:

```
cache:
  paths:
    - .venv/
build_job:
  stage: build
  script:
    - pip install -r requirements.txt
    - python setup.py build
```

Тут кешується віртуальне середовище Python .venv, в якому знаходяться залежності.

Специфікація ключа кешу. Для забезпечення актуальності кешу можна використовувати ключі. Це дозволяє змінювати кеш при змінах у файлах проєкту.

```
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
    - node_modules/
```

У цьому коді ключ кешу змінюється відповідно до гілки коміту, що дозволяє використовувати окремий кеш для кожної гілки.

Табл. 3.1 узагальнює головні аспекти налаштування кешування для оптимізації пайплайнів у GitLab CI/CD, враховуючи розглянуті вище конкретні приклади налаштування для різних типів проєктів.

Таблиця 3.1 – Налаштування кешування для оптимізації пайплайнів у GitLab CI/CD

Розділ	Опис
Налаштування кешування у GitLab CI/CD	Використання кешування для тимчасового зберігання залежностей та файлів між виконаннями пайплайнів для уникнення повторного завантаження або збірки.
Налаштування кешу в .gitlab-ci.yml	Додавання секції cache у файл .gitlab-ci.yml для налаштування кешування, з прикладами для Node.js (node_modules/) та Python (.venv/) проєктів.
Специфікація ключа кешу	Використання ключів кешу, наприклад ${CI_COMMIT_REF_SLUG}, для забезпечення актуальності кешу, що змінюється відповідно до гілки коміту.

Підходи до оптимізації пайплайнів у GitLab CI/CD включають також низку методів, які дозволяють підвищити ефективність та продуктивність пайплайнів у GitLab, за рахунок швидшого виконання завдань та оптимізації ресурсів, зокрема, це: паралелізація завдань, використання кешування та артефактів, використання специфікацій only/except для контролю запуску завдань, використання правил (rules) для більш гнучкого контролю запуску завдань.

Паралелізація завдань дозволяє виконувати декілька завдань одночасно, зменшуючи загальний час виконання пайплайну. Приклад налаштування паралелізації завдань:

```
test:
  stage: test
  script: run_tests.sh
  parallel: 5
```

У цьому прикладі завдання test виконується в п’яти паралельних потоках, що зменшує загальний час тестування.

Ефективне використання кешування та артефактів зменшує необхідність повторного виконання деяких кроків пайплайну. Приклад кешування:

```
build:
  stage: build
  script: build_script.sh
  cache:
    paths:
      - build_output/
```

У цьому випадку вивід збірки зберігається в кеші, що дозволяє уникнути повторної збірки при наступних запусках.

Специфікації only та except дозволяють контролювати, при яких умовах завдання має виконуватися. Приклад налаштування:

```
deploy_to_production:
  stage: deploy
  script: deploy_script.sh
  only:
    - master
```

Тут завдання deploy_to_production виконується тільки для гілки master.

Правила (rules) дозволяють визначати складніші умови для виконання завдань. Приклад налаштування правил:

```
deploy:
  stage: deploy
  script: deploy_script.sh
  rules:
```

- if: ‘$CI_COMMIT_BRANCH = = "master"‘

- if: ‘$CI_PIPELINE_SOURCE = = "schedule"‘

Тут завдання deploy виконується, якщо воно знаходиться на гілці master або запущене за розкладом.

Табл. 3.2 узагальнює відомості про методи та стратегії оптимізації пайплайнів у GitLab CI/CD.

Таблиця 3.2 – Налаштування кешування та оптимізація пайплайнів у GitLab CI/CD

Розділ	Опис
Паралелізація завдань	Виконання декількох завдань одночасно, наприклад, налаштування test завдання, що виконується паралельно в 5 потоках.
Ефективне використання кешування та артефактів	Зменшення необхідності повторного виконання деяких кроків пайплайну.
Специфікації only та except	Контроль виконання завдань в залежності від умов, наприклад, налаштування deploy_to_production завдання, яке виконується лише для гілки master.
Правила (rules)	Визначення складніших умов для виконання завдань, наприклад, налаштування deploy завдання, яке виконується для гілки master або за розкладом.

Ця таблиця узагальнює головні аспекти оптимізації пайплайнів у GitLab CI/CD, включаючи конкретні приклади налаштування для різних типів проєктів та різних стратегій оптимізації.

Для дослідження ефективності методів та алгоритмів налаштування CI/CD для GitLab-репозиторіїв використовуються такі основні метрики [50].

Час виконання CI/CD процесу – час, який потрібен для виконання різних CI/CD процесів (наприклад, збірки, тестування, розгортання) для різних налаштувань. Наприклад:

- налаштування A: 15 хвилин;

- налаштування B: 20 хвилин;

- налаштування C: 12 хвилин.

Кількість помилок або невдач під час CI/CD процесу для різних налаштувань. Наприклад:

- налаштування A: 3 помилки;

- налаштування B: 1 помилка;

- налаштування C: 5 помилок.

Результати тестів, такі як кількість успішних і неуспішних тестів, покриття коду [27], час виконання тощо. Наприклад:

- налаштування A: 95% покриття коду, 10 неуспішних тестів;

- Налаштування B: 98% покриття коду, 2 неуспішних тестів;

- Налаштування C: 92% покриття коду, 15 неуспішних тестів.

Швидкість розгортання – час, який потрібен для розгортання додатку на виробничому сервері для різних налаштувань. Приклад:

- налаштування A: 30 хвилин;

- налаштування B: 25 хвилин;

- налаштування C: 40 хвилин.

Додаткові приклади метрик та даних, які можуть бути використані у контексті дослідження методів та алгоритмів налаштування CI/CD для GitLab-репозиторіїв.

Використання ресурсів (пам’ять, процесор). Можна вимірювати, скільки ресурсів (пам’ять, процесорний час) використовує кожне налаштування CI/CD під час виконання процесів. Це важливо для ефективності та оптимізації ресурсів. Наприклад:

- налаштування A: використання 4 ГБ пам’яті, 20% процесорного часу;

- налаштування B: використання 6 ГБ пам’яті, 25% процесорного часу;

- Налаштування C: використання 3 ГБ пам’яті, 15% процесорного часу.

Кількість релізів – скільки релізів (версій вашого додатку) створено для різних налаштувань CI/CD. Показує інтенсивність розгортання та випуску нових функцій. Приклад:

- налаштування A: 5 релізів;

- налаштування B: 7 релізів;

- налаштування C: 3 релізи.

Витрати на хмарові сервіси: важливо для оцінки витрат та планування бюджету. Приклад:

- налаштування A: $200 на місяць;
- налаштування B: $250 на місяць;
- налаштування C: $150 на місяць.

Кількість виконаних задач (jobs), які виконуються в рамках CI/CD пайплайну для різних налаштувань. Показує складність та обсяг пайплайну. Приклад:

- налаштування A: 10 виконаних задач;
- налаштування B: 15 виконаних задач;
- налаштування C: 8 виконаних задач.

Використання метрик допомагає оцінити ефективність різних налаштувань CI/CD та їх вплив на процес розробки. Вони використовуються для аналізу та порівняння. Наведені міркування узагальнимо у табл. 3.3.

Таблиця 3.3 – Основні метрики ефективності методів та алгоритмів налаштування CI/CD для GitLab-репозиторію.

Метрика	Налаштування A	Налаштування B	Налаштування C
Час виконання CI/CD процесу	15 хвилин	20 хвилин	12 хвилин
Кількість помилок	3 помилки	1 помилка	5 помилок
Результати тестів	95% покриття, 10 помилок	98% покриття, 2 помилки	92% покриття, 15 помилок
Швидкість розгортання	30 хвилин	25 хвилин	40 хвилин
Використання ресурсів	4 ГБ, 20% CPU	6 ГБ, 25% CPU	3 ГБ, 15% CPU
Кількість релізів	5 релізів	7 релізів	3 релізи
Витрати на хмарові сервіси	$200/місяць	$250/місяць	$150/місяць
Кількість виконаних задач	10 задач	15 задач	8 задач

Таблиця відображає метрики, які використовуються при аналізі CI/CD пайплайнів. Ці метрики дозволяють оцінити ефективність різних налаштувань CI/CD та їх вплив на загальний процес розробки.

3.5 Інтеграція з зовнішніми сервісами та інструментами

Інтеграції з базами даних, хмарними сервісами, системами моніторингу у контексті GitLab CI/CD дозволяють автоматизувати різні аспекти розробки та експлуатації програмного забезпечення, від тестування та розгортання до моніторингу стану додатку та інфраструктури. Вони сприяють підвищенню ефективності та надійності процесів CI/CD.

Інтеграція з базами даних часто використовується для автоматизації процесів тестування та розгортання, особливо при роботі з даними. Приклад:

```
test:
  stage: test
  script:
    - setup_database
    - run_tests
```

У цьому прикладі перед виконанням тестів запускається скрипт setup_database, який готує базу даних для тестування.

Інтеграція з хмарними сервісами. Хмарні сервіси, такі як AWS, Azure або Google Cloud, часто інтегруються у GitLab CI/CD для зберігання артефактів, розгортання додатків та інших операцій. Приклад:

```
deploy:
  stage: deploy
  script:
    - deploy_to_cloud_service
  environment:
    name: production
    url: http://production.example.com
```

Тут використовується команда deploy_to_cloud_service для розгортання додатку на хмарний сервіс.

Інтеграція з системами моніторингу. Системи моніторингу, такі як Prometheus, Grafana чи New Relic, можуть використовуватися для спостереження за станом додатку та інфраструктури. Приклад інтеграції:

```
monitoring:
  stage: deploy
  script:
    - setup_monitoring
  only:
    - master
```

У цьому коді setup_monitoring встановлює та налаштовує моніторинг для додатку після його розгортання.

Використання Webhooks та API у GitLab CI/CD дозволяє розширити можливості автоматизації, забезпечуючи високу гнучкість та інтеграцію з різноманітними зовнішніми сервісами та інструментами. Це підвищує ефективність процесів розробки та дозволяє автоматизувати багато рутинних завдань.

Webhooks в GitLab CI/CD використовуються для сповіщення зовнішніх сервісів про певні події в репозиторії, такі як push, merge request або виконання пайплайну. Приклад використання Webhook:

```
notify_service:
  stage: deploy
  script:
    - curl -X POST -H "Content-Type: application/json" -d ‘{"text":"Розгортання завершено"}’ http://example.com/webhook
  only:
    - master
```

У цьому прикладі використовується скрипт curl для відправки POST-запиту на зовнішній вебхук після розгортання. Так можна повідомляти, наприклад, службу моніторингу або командний чат про завершення розгортання.

Використання API. GitLab API дозволяє автоматизувати різні дії, такі як створення або оновлення merge requests, управління issues, отримання інформації про проєкти та багато іншого. Приклад використання API:

```
create_merge_request:
  stage: post-deploy
  script:
    - curl -X POST -H "Private-Token: $GITLAB_API_TOKEN" "https://gitlab.example.com/api/v4/projects/1/merge_requests" -d "source_branch=feature" -d "target_branch=master" -d "title=New Feature"
  only:
    - develop
```

У цьому коді за допомогою GitLab API створюється новий merge request з гілки feature до master після виконання певних дій у гілці develop.

3.6 Налаштування безпеки у CI/CD процесах

Існує низка конкретних методів та практик, які можна впровадити в GitLab CI/CD для підвищення безпеки. Одним з таких методів є застосування практик DevSecOps, що включає інтеграцію безпеки безпосередньо в процес розробки. Також, важливим аспектом безпеки є ефективний контроль доступу та управління секретами, які забезпечують захист конфіденційної інформації та даних.

Застосування практик DevSecOps в GitLab CI/CD. DevSecOps – це підхід, який інтегрує практики безпеки в процеси розробки (DevOps), спрямований на раннє виявлення та виправлення проблєм безпеки. Застосування практик DevSecOps у GitLab CI/CD дозволяє прискорити процеси розробки та розгортання, а також значно підвищити безпеку програмного продукту, вбудовуючи практики безпеки безпосередньо в цикл розробки.

Інтеграція інструментів безпеки. Одним з ключових елементів DevSecOps є автоматичний аналіз безпеки коду та інфраструктури. Приклад інтеграції сканера вразливостей:

```
security_scan:
  stage: test
  script:
    - echo "Запуск сканування вразливостей"
    - security_scan_tool
  only:
    - master
```

У цьому прикладі security_scan_tool може бути інструментом Snyk або SonarQube, який сканує код на вразливості.

Автоматизація тестування безпеки. Автоматичне тестування безпеки додатків та інфраструктури допомагає виявляти потенційні загрози на ранніх етапах розробки. Приклад налаштування тестів безпеки:

```
security_tests:
  stage: test
  script:
    - run_security_tests
  only:
    - develop
```

Тут run_security_tests – це набір автоматизованих тестів безпеки.

Ревізія коду та контроль доступу. Ретельна ревізія коду та обмежений доступ до чутливих частин системи є важливими аспектами DevSecOps. Приклад налаштування правил доступу:

```
review:
  stage: review
  script:
    - code_review_tool
  only:
```

```
  - merge_requests
```

У цьому прикладі code_review_tool може бути інструментом або процесом, що забезпечує ретельний огляд коду перед злиттям.

Контроль доступу та управління секретами є критично важливими для забезпечення безпеки CI/CD процесів у GitLab. Вони допомагають уникнути несанкціонованого доступу та витоку конфіденційної інформації, забезпечуючи безпечне та надійне середовище для автоматизованої розробки.

Контроль доступу в GitLab CI/CD важливий для забезпечення того, щоб лише уповноважені особи мали доступ до репозиторію, налаштувань CI/CD та інших критичних ресурсів. Приклад налаштування контролю доступу:

```
deploy_production:
  stage: deploy
  script:
    - deploy_script.sh
  only:
    - master
  when: manual
  allow_failure: false
```

У цьому прикладі розгортання в продуктивне середовище вимагає ручного схвалення (when: manual), і дозволено виконувати його тільки з гілки master.

Управління секретами у GitLab CI/CD – це процес забезпечення надійного зберігання, доступу та використання конфіденційної інформації у системах і програмах. Секрети можуть включати паролі, токени доступу, приватні ключі, сертифікати та інші види даних, які необхідно захищати від несанкціонованого доступу та витоку. У контексті розробки програмного забезпечення управління секретами забезпечує, що такі дані безпечно передаються та використовуються у різних частинах програми або процесу розробки. Наприклад, під час автоматизованого процесу збірки та розгортання, секрети можуть використовуватися для доступу до баз даних, API, хмарних сервісів тощо.

Ефективне управління секретами допомагає запобігти витоку важливої інформації та забезпечує, що секрети використовуються належним чином в рамках безпечного середовища. Воно включає:

- зберігання секретів у безпечному сховищі – використання спеціалізованих інструментів або сервісів для зберігання конфіденційної інформації;

- контрольований доступ – надання доступу до секретів лише уповноваженим особам або процесам;

- шифрування – захист секретів за допомогою шифрування, як під час їх зберігання, так і під час передачі;

- аудит та моніторинг – відстеження доступу до секретів та реєстрація дій, пов'язаних з ними.

Управління секретами в GitLab CI/CD здійснюється за допомогою змінних середовища, які зберігають конфіденційні дані, такі як паролі, токени доступу тощо. Приклад налаштування управління секретами:

```
deploy:
  stage: deploy
  script:
    - echo "Розгортання з використанням секрету"
    - deploy_script.sh
  environment:
    name: production
    url: https://production.example.com
  variables:
    SECRET_KEY: $PRODUCTION_SECRET_KEY
```

У цьому випадку SECRET_KEY є змінною середовища, яка містить секретний ключ. Цей ключ може бути встановлений у налаштуваннях GitLab CI/CD та не відображається у логах.

Безпечне зберігання секретів. Секрети не повинні зберігатися в коді або .gitlab-ci.yml. Вони повинні зберігатися в безпечних місцях, таких як змінні

середовища GitLab або сховища секретів. Приклад використання змінних середовища:

```
build:
  stage: build
  script:
    - echo "Збірка з використанням змінної середовища"
    - build_script.sh
  variables:
    API_KEY: $API_KEY
```

Тут API_KEY використовується для збірки, і ця змінна середовища встановлюється в налаштуваннях GitLab.

3.7 Економічна оцінка проєкту

Оцінка економічної ефективності впровадження CI/CD включає аналіз витрат, розрахунок економії та визначення показника ROI (Return on Investment).

Статті витрат:

- витрати на IT-інфраструктуру – придбання або оренда серверів, обладнання, а також витрати на хмарні сервіси (якщо CI/CD реалізовується в хмарі);
- ліцензійні витрати – витрати на придбання ліцензій інструментів CI/CD, як-от Jenkins, GitLab CI тощо;
- трудові ресурси – витрати на залучення та навчання персоналу, що займатиметься впровадженням та підтримкою CI/CD;
- оперативні витрати – витрати на підтримку, оновлення і моніторинг системи CI/CD.

Розрахунок витрат. Для кожної статті витрат робиться розрахунок, виходячи з потреб проєкту. Наприклад, витрати на IT-інфраструктуру можуть складати $X, ліцензійні витрати - $Y, трудові ресурси - $Z тощо.

Фактори економічної ефективності:

- зменшення часу на розгортання – CI/CD дозволяє автоматизувати багато процесів, скорочуючи час розгортання проєктів;

- покращення якості продукту – неперервна інтеграція та розгортання зменшують кількість помилок і збільшують якість продукту;

- зменшення витрат на усунення помилок – швидке виявлення та виправлення помилок знижує витрати на їх усунення в майбутньому.

ROI розраховується як відношення вигод (економія та додатковий дохід) до загальних витрат на впровадження CI/CD за формулою:

$$ROI = ((ЕК + ДД) – ЗВ) / ЗВ * 100\% \quad (3.1)$$

де: ЕК – економія; ДД – додатковий дохід; ЗВ – загальні витрати.

Для оцінки економічної ефективності впровадження CI/CD припустимо, що компанія планує впровадити CI/CD для своїх проєктів і для цього потрібні такі витрати:

- інфраструктура та IT-обладнання – $20000.
- ліцензії на інструменти CI/CD (Jenkins, GitLab CI тощо) – $5000.
- трудові витрати (налаштування, навчання) – $15000.
- оперативні витрати (у т.ч. підтримка) – $10000.

Загальні витрати складають:

$20000 + $5000 + $15000 + $10000 = $50000.

Табл. 3.4 показує розрахунки для визначення витрат для впровадження процесу CI/CD в IT- компанії.

Таблиця 3.4 – Розрахунок витрат для впровадження CI/CD

Стаття витрат	Сума, $
Інфраструктура та IT-обладнання	20000
Ліцензії на інструменти CI/CD	5000
Трудові витрати	15000
Оперативні витрати	10000
Загальні витрати	50000

Оцінка економічної ефективності. Припускаємо, що після впровадження CI/CD компанія очікує зниження часу на розгортання проєктів на 30%, що веде до збільшення продуктивності, а також зменшення кількості помилок на 25%, що відповідно знижує витрати на їх усунення. Так, збільшення продуктивності дозволяє компанії виконувати на 30% більше проєктів за рік, забезпечуючи додатковий дохід у $100000. Економія на усуненні помилок оцінюється у $20000 за рік. Таким чином, загальний додатковий дохід та економія становлять:

$100000 (дохід) + $20000 (економія) = $120000.

Табл. 3.5 містить розрахунок економічної вигоди.

Таблиця 3.5 – Розрахунок економічної вигоди від впровадження CI/CD

Стаття доходів та економії	Сума, $
Додатковий дохід від збільшення продуктивності	100000
Економія на усуненні помилок	20000
Загальна економія та доходи	120000

Розрахунок ROI виконаємо за формулою (3.1). Одержимо:

ROI = ($120000 – $20000) / $50000 * 100% = 140%.

З отриманого результату видно, що впровадження CI/CD принесло компанії значну економічну вигоду, оскільки ROI становить 140%, що свідчить про високу ефективність інвестицій у впровадження цієї системи.

Табл. 3.6 містить підсумок витрат та економічної вигоди та розрахунок ROI.

Таблиця 3.6 – Загальна таблиця доходів, витрат та розрахунок ROI

Категорія	Сума, $
Витрати	50000
Доходи та економія	120000
ROI, %	140%

Отриманий показник ROI свідчить про високу економічну ефективність інвестицій у впровадження системи CI/CD.

Висновки до розділу 3

Запропоновано базовий пайплайн CI/CD у GitLab, що поєднує етапи збірки, тестування та розгортання, забезпечуючи неперервний цикл розробки.

Розглянуто інтеграцію тестування та аналізу коду, що є ключовими завданнями для забезпечення високої якості програмного продукту. Зроблено висновок, що використання різних інструментів та методів CI/CD у GitLab дозволяє виявляти помилки та вразливості на ранніх етапах.

Встановлено, що автоматизація розгортання та управління різними середовищами (development, staging, production) підвищує ефективність процесів розгортання та спрощує керування версіями продукту.

Показано, що використання кешування та оптимізаційних технік, таких як паралелізація завдань, значно зменшує час виконання пайплайнів, оптимізуючи ресурси та час. Інтеграція з зовнішніми сервісами, такими як бази даних, хмарні платформи, системи моніторингу, забезпечує більш гнучку та функціональну інфраструктуру, сприяючи розширенню можливостей CI/CD. Впровадження практик DevSecOps та забезпечення безпеки на всіх етапах CI/CD є критично важливим для запобігання витокам даних, забезпечення конфіденційності та інтеграції інформації.

Розглянута економічна оцінка ефективності впровадження CI/CD. Отриманий показник ROI свідчить про високу економічну ефективність інвестицій у впровадження системи CI/CD.

ВИСНОВКИ

Робота була зосереджена на аналізі та оптимізації процесу налаштування CI/CD для GitLab та включає розгляд різних методів, алгоритмів та практик. Загальною метою було покращення ефективності та надійності процесу розробки та розгортання програмного забезпечення. За допомогою розглянутих змін та рекомендацій, дослідження дозволило досягнути цієї мети.

Головні результати роботи:

1. Досліджено існуючий процес налаштування CI/CD в GitLab та виявлено проблєми та виклики, такі як тривалий час виконання, часті помилки та втручання;

2. Розглянуто нові методи та практики налаштування CI/CD в GitLab, включаючи автоматизацію, використання інфраструктури як коду, контейнеризацію;

3. Проведено експерименти для порівняння різних методів налаштування CI/CD в GitLab. Запропонована оцінка покращень, отриманих завдяки новим методам, зокрема зменшення часу виконання, зменшення кількості помилок та підвищення якості коду;

4. Надано практичні рекомендації для розробників щодо вдосконалення процесу CI/CD в GitLab, включаючи використання автоматизації, контейнеризації та моніторингу.

Результати роботи відповідають поставленим завданням, а саме:

- проведено докладний аналіз існуючого стану процесу налаштування CI/CD в GitLab, що відповідає поставленому завданню;

- запропоновані нові методи та практики CI/CD, які були впроваджені в реальному середовищі розробки, відповідаючи завданню з оптимізації;

- проведено експериментальне порівняння ефективності різних методів, які були частиною завдання дослідження;

- зібрані та проаналізовані дані під час експериментів, що відповідає завданню з оцінки результатів;

- надано практичні рекомендації для покращення процесу CI/CD, що відповідає завданню надати практичні рекомендації.

Наукова новизна роботи включає:

- аналіз існуючих підходів, методів та алгоритмів налаштування CI/CD в GitLab, включаючи порівняльний аналіз з іншими платформами та інструментами;

- аналіз методів інтеграції практик безпеки та керування доступом у процеси CI/CD;

- запропонована власна методика налаштування CI/CD pipelines, яка враховує специфіку використання GitLab та сучасні вимоги до процесів розробки програмного забезпечення;

- аналіз та рекомендації щодо оптимізації та масштабування CI/CD pipelines для великих та складних проєктів, що є важливим для підприємств із великими IT-структурами.

Практичне значення роботи:

- надано конкретні рекомендації щодо налаштування та оптимізації CI/CD pipelines у GitLab, що сприятиме підвищенню ефективності розробки та доставки програмного забезпечення;

- рекомендації щодо оптимізації та масштабування CI/CD pipelines можуть бути корисними для великих та складних проєктів, а також для малих та середніх підприємств;

- запропоновано методики, що дозволяють зменшити кількість помилок у коді та забезпечити більш стабільне та безпечне розгортання програм, що позитивно впливає на якість кінцевих продуктів;

- результати роботи можуть бути корисними для студентів, викладачів та практиків у галузі програмної інженерії та DevOps.

Особистий внесок автора розкривається у наступних аспектах:

- аналіз наявної літератури, статей, технічної документації, що стосуються CI/CD процесів, з акцентом на особливості GitLab, та систематизував отримані знання;

- розробка методики налаштування CI/CD pipelines у GitLab, що включає конкретні рекомендації;

- практичне тестування розроблених методик, що дозволило оцінити їх ефективність;

- формулювання висновків та рекомендацій, які можуть бути корисними для розробників, менеджерів проєктів та інших фахівців у галузі IT.

Перспективи для подальших розвідок у галузі налаштування CI/CD для GitLab-репозиторіїв:

- розширення дослідження з інтеграцією інших інструментів – дослідження інтеграції GitLab CI/CD з іншими популярними інструментами та платформами, такими як Docker, Kubernetes, облачні сервіси тощо;

- автоматизація та оптимізація процесів тестування – дослідження ефективності різних стратегій автоматизованого тестування в контексті CI/CD, зокрема, розробка методик для оптимізації процесів тестування;

- аналіз безпеки та вразливостей у CI/CD pipelines – дослідження аспектів безпеки, включаючи аналіз потенційних вразливостей та розробка методів їх усунення або мінімізації;

- проведення експериментальних досліджень та аналіз існуючих практик CI/CD у реальних проєктах, щоб оцінити практичну ефективність розроблених методик;

- розробка та аналіз моделей CI/CD, які можуть бути адаптовані до різних типів проєктів – від стартапів до великих корпоративних систем;

- аналіз впливу нових технологій на CI/CD процеси – дослідження потенційного впливу новітніх технологій, таких як штучний інтелект та машинне навчання, на оптимізацію та автоматизацію процесів CI/CD.

Ці перспективи відображають потенціал для розвитку та поглиблення дослідження у цій сфері, відкриваючи нові можливості для інновацій та вдосконалення практик у галузі програмної інженерії та DevOps.