

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «магістр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)
на тему:

**«Використання архітектури мікросервісів та методів
Continuous Integration / Continuous Delivery за
допомогою хмарних технологій»**

Виконав студент групи ПЗм-22-2
МАЦАПУРА Владислав Олександрович

Керівник роботи:
ВЛАСЕНКО Олег Васильович

Рецензент:
КОНЦИДАЙЛО Андрій Михайлович

Житомир – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

зав. кафедри інженерії
програмного забезпечення
Тетяна ВАКАЛЮК
«18» вересня 2023 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: МАЦАПУРА Владислав Олександрович

Керівник роботи: ВЛАСЕНКО Олег Васильович

Тема роботи: «Використання архітектури мікросервісів та методів Continuous Integration / Continuous Delivery за допомогою хмарних технологій»,

затверджена Наказом закладу вищої освіти від **«18» вересня 2023 р. № 479/с**

Вихідні дані для роботи: **предметом дослідження є засоби розробки для CI/CD додатків, що використовують мікросервісну архітектуру. Об'єктом дослідження є застосування методів CI/CD для високонавантажених систем.**

Консультанти з магістерської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Власенко О.В.	22.09.2023р.	22.09.2023р.
2	Власенко О.В.	22.09.2023р.	22.09.2023р.
3	Власенко О.В.	22.09.2023р.	22.09.2023р.

РЕФЕРАТ

Індивідуальне завдання з кваліфікаційної магістерської роботи полягає в дослідженні та розробки системи для Continuous Integration/Continuous Deployment мікросервісних додатків в кластері, що працює в хмарному середовищі. Методи включають вибір хмарного провайдера, розробку та реалізацію системи, розгортання інфраструктури, розгортання мікросервісів, тестування CI/CD системи.

Кваліфікаційна магістерська робота на тему «Використання архітектури мікросервісів та методів Continuous Integration / Continuous Delivery за допомогою хмарних технологій» складається з теоретичного та програмного комплексу системи та пояснювальної записки. Пояснювальна записка до кваліфікаційної магістерської роботи складається зі вступу, трьох розділів, списку літератури з 26 найменувань, 27 рисунків, 6 таблиці, 2 додатків. Повний обсяг роботи становить 84 сторінки.

Ключові слова: МІКРОСЕРВІСНА АРХІТЕКТУРА, CI/CD, ХМАРНЕ СЕРЕДОВИЩЕ, AWS, KUBERNETES, DOCKER, GITOPS.

ABSTRACT

The individual task for the qualifying master's thesis is to research and develop a system for Continuous Integration/Continuous Deployment of microservice applications in a cluster operating in a cloud environment. Methods include selecting a cloud provider, developing and implementing a system, deploying infrastructure, deploying microservices, testing the CI/CD system.

The qualifying master's thesis on the topic "Using microservices architecture and Continuous Integration/Continuous Delivery methods using cloud technologies" consists of a theoretical and software system complex and an explanatory note. The explanatory note to the qualification master's thesis consists of an introduction, three chapters, a list of references of 26 titles, 27 figures, 6 tables, 2 appendices. The total volume of the work is 84 pages.

KEYWORDS: MICROSERVICE ARCHITECTURE, CI/CD, CLOUD ENVIRONMENT, AWS, KUBERNETES, DOCKER, GITOPS.

ЗМІСТ

РЕФЕРАТ.....	3
ABSTRACT.....	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	7
ВСТУП.....	8
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ	11
1.1 Постановка завдання	11
1.2 Мікросервісна архітектура.....	13
1.3 Аналіз методик розробки програмного забезпечення	17
1.4 Continuous Integration/Continuous Deployment	21
1.5 Аналіз хмарних технологій.....	22
1.6 Kubernetes кластер	25
1.7 Об'єкти Kubernetes	30
1.8 GitOps.....	35
1.9 Висновок до розділу 1	39
РОЗДІЛ 2 РОЗРОБКА АРХІТЕКТУРИ СИСТЕМИ ТА РІШЕННЯ ДЛЯ CI/CD В ХМАРНЕ ОТОЧЕННЯ	41
2.1 Розробка архітектури системи	41
2.2 Середовище розробки	45
2.3 Зовнішні компоненти розробки.....	47
2.4 Основні задачі системи	51
2.5 Висновок до розділу 2	57
РОЗДІЛ 3 ДОСЛІДЖЕННЯ СИСТЕМИ ТА ЕФЕКТИВНОСТІ РІШЕННЯ.....	59
3.1 Створення середовища.....	59
3.2 Використання конкурентних рішень.....	60
3.3 Аналіз отриманих результатів	66
3.4 Порівняння функціоналу систем	70
3.5 Висновок до розділу 3	71
ВИСНОВКИ	73

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....	75
ДОДАТКИ	78

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ОС – операційна система

БД – база даних

ПЗ – програмне забезпечення

CI – Continuous Integration (безперервна інтеграція)

CD – Continuous Delivery (безперервна доставка та розгортання)

API – Application Program Interface (прикладний програмний інтерфейс)

REST – (Representational State Transfer) – підхід до побудови веб-сервісів

VCS – система контролю версій.

ВСТУП

Актуальність теми. Інформаційні технології стали повсякденним явищем в компаніях і неминучим ключовим фактором успіху в бізнесі.

Сьогодні багато людей переглядають свою пошту онлайн через поштові клієнти, редагують спільні документи за допомогою веб-браузерів, створюють віртуальні альбоми для завантаження своїх фотографій з відпустки. Вони запускають програми та зберігають дані на серверах розташованих в Інтернеті, а не на власних комп'ютерах. Все що залишається робити користувачеві це прості введення в пошуковій системі — це єдине, що робить користувач, щоб почати користуватися послугами, які знаходяться на віддалених серверах і дозволяє йому обмінюватися конфіденційною інформацією, або використовувати обчислювальні потужності купи серверів, які він ніколи не побачить на власні очі. І з кожним днем все більше використовуються саме такі сервіси, які називаються хмарними комп'ютерними сервісами. Така назва дана через метафору про Інтернет, як про щось, що користувач бачить як хмару і не може доторкнутись, та побачити, що знаходиться всередині.

В наш час, компанії можуть створювати, спілкуватися і співпрацювати швидше, ефективніше і надійніше, ніж будь-коли раніше завдяки потужностям серверів. На сьогоднішній день багатьом підприємствам або користувачам необхідно обробляти великі обсяги інформації даних або потрібно виконати складні операції, наприклад, математичні обчислення, і їм потрібна велика кількість обчислювальних потужностей для їх вирішення. Можливо, потужність персонального комп'ютера може бути недостатньо, щоб виконати за певний час одне завдання, а може це завдання вимагає більших потужностей ніж має компанія, і для компанії немає сенсу купувати апаратні засоби, необхідні для його виконання, адже існують хмарні рішення, які можна отримати влюбий час, в зручному для компанії регіоні та не турбуватись, що дані можуть бути втрачені внаслідок будь-яких явищ.

Сам термін архітектури мікросервісів ще почав зароджуватись в 2010 році, цей метод описував інший підхід до побудови архітектури при розробці програмних додатків. Також цей метод став дуже популярним з розвитком практик поєднань гнучкої та безперервної розробки, безперервної інтеграції та доставки додатків, всі ці ідеї поєднуються в одній філософії — DevOps. Мета архітектури мікросервісів полягає в ідеї, що додаток повинен будуватись як сукупність невеликих сервісів, що спілкуються між собою за допомогою стандартних механізмів як HTTP, AMQP. Самі сервіси побудовані на основі бізнес потреб компанії та розгортаються в автоматизованому середовищі, а також сервіси взагалі можуть бути написані на різних мовах та використовувати різні сховища даних.

З розвитком хмарних технологій на ринку з'явилося чимало хмарних рішень від різних провайдерів, тому постає актуальність проблеми в детальному аналізі хмарних рішень від різних провайдерів, та якими способами компанії можуть розробляти та швидко доставляти до користувачів свої програмні додатки за допомогою зовнішніх систем.

Мета і завдання роботи. Дослідити та розробити систему для застосування використання методів CI/CD та розгортання додатку в робочому середовищі.

Досягнення поставленої мети передбачає розв'язання таких **завдань**:

- дослідити наявні рішення, які виконують процеси CD/CD в робочі середовища;
- дослідити, які існують робочі середовища для високонавантажених систем;
- розробити архітектуру для системи;
- розробити систему для безперервної інтеграції та доставки додатку в робоче середовище;

Об'єктом дослідження є застосування методів CI/CD для високонавантажених систем.

Предметом дослідження є засоби розробки для CI/CD додатків, що використовують мікросервісну архітектуру.

Наукова новизна отриманих результатів полягає в такому:

- розроблено систему для розгортання додатку з мікросервісною архітектурою, яка пришвидшує розгортання додатку в робочі середовища;
- проведено аналіз хмарних технологій у контексті застосування архітектури мікросервісів;
- досліджено процес розгортання додатків з використанням хмарних технологій та мікросервісної архітектури;
- проаналізовано та розроблено методику розробки додатків із застосуванням методів CI/CD для розробленої системи;

Апробація результатів роботи. Матеріали кваліфікаційної роботи доповідалися на V Всеукраїнської науково-технічної конференції «Комп’ютерні технології: інновації, проблеми, рішення» 01-02 грудня 2022 року [17].

РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка завдання

Обрана тема є актуальною в сучасній індустрії розробки програмного забезпечення з-за кількох ключових факторів.

Спочатку, стрімке зростання технологічних вимог та потреб користувачів призводить до необхідності швидкої та надійної поставки програмних продуктів. CI/CD надає засоби для автоматизації та прискорення цього процесу, що стає ключовим для успішної роботи на ринку.

Далі, динаміка сучасного ринку вимагає частіших оновлень та швидкого впровадження нового функціоналу. Система CI/CD дозволяє розробникам та командам швидко та безперервно інтегрувати зміни, зменшуючи час між внесенням змін і їх виконанням.

Крім того, зменшення людського втручання у процесі розгортання програмних оновлень за допомогою автоматизованих систем CI/CD сприяє підвищенню якості та стабільності програм, оскільки це допомагає уникнути людських помилок та забезпечує однакову послідовність процесів.

Робота над системою для Continuous Integration / Continuous Delivery (CI/CD) представляє собою важливий крок у сучасній сфері розробки програмного забезпечення. Розробка власного інструментарію для автоматизації процесів тестування, розгортання та постачання програм відображає ключові вимоги, що ставляться перед сучасними програмними системами.

Актуальність цієї роботи базується на необхідності швидкого, надійного та безперервного впровадження змін у програмних продуктах. Власна система, розроблена з урахуванням потреб конкретного проекту чи команди розробників, має велике значення для забезпечення ефективності, стабільності та безпеки у процесах розгортання програм.

Розробка власного інструментарію для CI/CD дає можливість розгорнути, тестувати та оновлювати програми автоматично, що є важливим для швидкого реагування на зміни та вимоги ринку програмного забезпечення.

Розробка системи, спрямована на оптимізацію процесів розгортання та постачання програм, яка може стати ефективним інструментом для покращення робочого процесу. Це також відкриває можливості для дослідження нових методик, що сприяють підвищенню ефективності розробки програм та забезпечують адаптацію інструментів під конкретні проекти.

Згідно зі звітом "State of DevOps Report 2021" [26] від компанії Google, організації, які використовують практики CI/CD, мають вищу продуктивність та швидкість реагування на зміни. Цей звіт також показує, що підприємства, які успішно впроваджують CI/CD, досягають зниження частоти виправлень помилок у програмному забезпеченні.

Крім того, конкурентне середовище та вимоги користувачів до швидкості впровадження нових можливостей спонукають компанії до використання CI/CD для швидкого та безперервного впровадження змін у програмах. Наприклад, відомі компанії, такі як Netflix, Google та Amazon, активно використовують підходи CI/CD для того, щоб реагувати на потреби своїх користувачів швидко та ефективно [24].

GitOps — це підхід до розгортання програмного забезпечення, який базується на використанні систем контролю версій(VCS), зокрема Git, для автоматизації процесів розгортання та керування конфігурацією. Цей підхід пропонує використовувати Git як основний інструмент для зберігання конфігураційних файлів і коду програм, що дозволяє здійснювати автоматизоване розгортання та керування [20].

Можна також обґрунтувати тему наступним чином:

– декларативний підхід до розгортання: GitOps сприяє використанню декларативного опису потрібного стану системи. Інфраструктура та процеси

ci/cd визначаються за допомогою конфігураційних файлів, що зберігаються в git. Це забезпечує чіткість, консистентність та відстежуваність змін.

- автоматизований процес розгортання: GitOps сприяє автоматизації процесів розгортання програм і інфраструктури. Зміни в коді або конфігурації, збережені в git, можуть спричинити автоматичне оновлення інфраструктури або програм, що спрощує та прискорює впровадження змін;
- аудит та контроль версій: використання Git для зберігання конфігурацій дозволяє вести точний аудит змін та версіонування інфраструктури та програм. Це дозволяє здійснювати відкат до попередніх версій у разі потреби;
- керування змінами та колаборація: GitOps сприяє ефективному керуванню змінами через pull-request'и, перегляд коду та контроль доступу. Це сприяє командній роботі та контролю змін;

1.2 Мікросервісна архітектура

Мікросервісна архітектура представляє собою методологію розробки програмного забезпечення, де система будується як набір дрібних, незалежних компонентів, які називають мікросервісами. Кожен з цих мікросервісів фокусується на виконанні конкретної функції або послуги, має свій власний контекст в базі даних та працює як самостійний модуль.

Основна ідея полягає в тому, щоб розбити великий та складний додаток на менші, більш керовані та легко керовані частини, що спрощує розробку, тестування та розгортання системи в цілому.

Кожен мікросервіс може бути реалізований окремо, використовуючи свої технології та мови програмування. Це означає, що різні команди можуть працювати над окремими сервісами, використовуючи технології, які найбільше підходять для конкретної функції або завдання.

Кожен мікросервіс має власний інтерфейс, який дозволяє іншим сервісам спілкуватися з ним. Ці інтерфейси зазвичай є стандартизованими і можуть використовувати HTTP, REST або інші протоколи. Потреба в порівнянні

мікросервісної та монолітної архітектур виникає з різниці у їхньому підході до структуризації програмного забезпечення та ефективності в різних сценаріях розробки [18].

Зважаючи на обидва підходи, варто відмітити їх унікальні переваги та недоліки. Порівняння дозволяє розкрити, який підхід краще підходить для конкретних потреб, проектів або випадків використання.

У монолітних додатках весь функціонал та логіка зосереджені в одному блоку, що спрощує розробку та тестування. Проте, при зростанні складності системи моноліт може стати важким для розуміння та підтримки. Більший моноліт може також ускладнити співпрацю різних команд, оскільки розвиток, тестування та впровадження нових функцій відбуваються в одному кодовому блоку.

Натомість, мікросервісна архітектура розбиває програмне забезпечення на невеликі, автономні та незалежні компоненти — мікросервіси. Це спрощує підтримку, розширення та модернізацію системи, оскільки кожен сервіс може бути розгорнутий, масштабований, тестований та оновлюваний окремо.

Порівняння цих підходів допомагає визначити, яка модель краще відповідає певним потребам проекту чи бізнесу. Наприклад, для початкового швидкого старту може підійти моноліт, проте з ростом проекту чи бізнесу перехід до мікросервісів може забезпечити більшу гнучкість та швидкість реагування на зміни. Такий аналіз допомагає вибрати належний підхід для конкретної ситуації.

Таблиця 1.1

Порівняння мікросервісної та монолітної архітектури

Особливості	Мікросервіси	Монолітний підхід
Архітектура	Складається з невеликих, незалежних компонентів, що реалізують окремі функції та взаємодіють через мережу	Один великий, монолітний додаток, де всі частини взаємодіють безпосередньо
Розгортання	Кожен мікросервіс може бути розгорнутий окремо, використовуючи власні контейнери (наприклад, Docker) та інфраструктуру	Весь додаток розгортається як єдиний модуль на одній апаратній платформі

Продовження таблиці 1.1

Масштабованість	Дозволяє масштабувати окремі компоненти відповідно до навантаження	Вимагає масштабування всього додатку навіть у випадках, коли не всі його частини потребують масштабування
Управління	Вимагає додаткових інструментів для моніторингу, керування та координації різних мікросервісів	Може використовувати централізовану систему управління, але вона може стати вузьким місцем
Залежність компонентів	Менша залежність між компонентами, помилки в одному сервісі не впливають на інші	Сильна залежність між компонентами, помилка в одній частині може вплинути на всю систему
Розвиток та тестування	Легший розвиток та тестування через ізоляцію та незалежність компонентів	Ускладнений розвиток через великий обсяг коду та потенційну взаємозалежність
Надійність	Зниження ризику системних збоїв через ізоляцію та розподіленість	Ризик системних збоїв вищий через спільність інфраструктури

Розробка мікросервісів вимагає використання інструментів, які полегшують розгортання, керування та моніторинг цих ізольованих компонентів системи. Одним з ключових елементів є контейнеризація. Вона використовує технології, такі як Docker, для упакування мікросервісів у контейнери, що спрощує їхнє розгортання та управління. Крім того, оркестрація контейнерів, наприклад, інструменти типу Kubernetes, автоматизує управління та масштабуванням контейнерами, що містять мікросервіси [18].

Велика перевага мікросервісної архітектури полягає у здатності швидко масштабувати, оновлювати та вдосконалювати окремі сервіси, що дозволяє більш гнучко реагувати на зміни вимог, спрощує роботу з додатком та підвищує його надійність. Мікросервісна архітектура має ряд переваг та недоліків.

Переваги мікросервісів:

- гнучкість та швидкість розробки: кожен мікросервіс може бути розроблений, тестований та вдосконалюваний окремо, що дозволяє розробникам швидко впроваджувати зміни та нові функції;
- скорочення часу відновлення та масштабованість: у випадку збоїв або проблем в одному мікросервісі, це не вплине на роботу інших частин системи.

Це дозволяє швидко виправляти та відновлювати роботу сервісів без впливу на загальну систему;

- технологічна різноманітність: різні мікросервіси можуть використовувати різні технології, мови програмування та інструменти в залежності від їх функціональних потреб;
- швидкість масштабування: кожен мікросервіс може бути масштабований окремо від інших частин системи, що дозволяє ефективне використання ресурсів і масштабування лише тих сервісів, які це потребують;

Недоліки мікросервісів:

- складність управління: зі збільшенням кількості мікросервісів складність управління системою зростає (рис.1.1), оскільки потрібно відслідковувати та керувати кожним окремим сервісом;
- складність тестування та відлагодження: важко забезпечити цілісність та комплексність тестування при взаємодії багатьох різних сервісів;
- збільшення обсягу мережевого трафіку: багато взаємодій між мікросервісами можуть призвести до збільшення обсягу мережевого трафіку;
- складність конфігурації та розгортання: розгортання та конфігурування багатьох окремих сервісів може бути складним завданням, особливо при великій кількості сервісів;

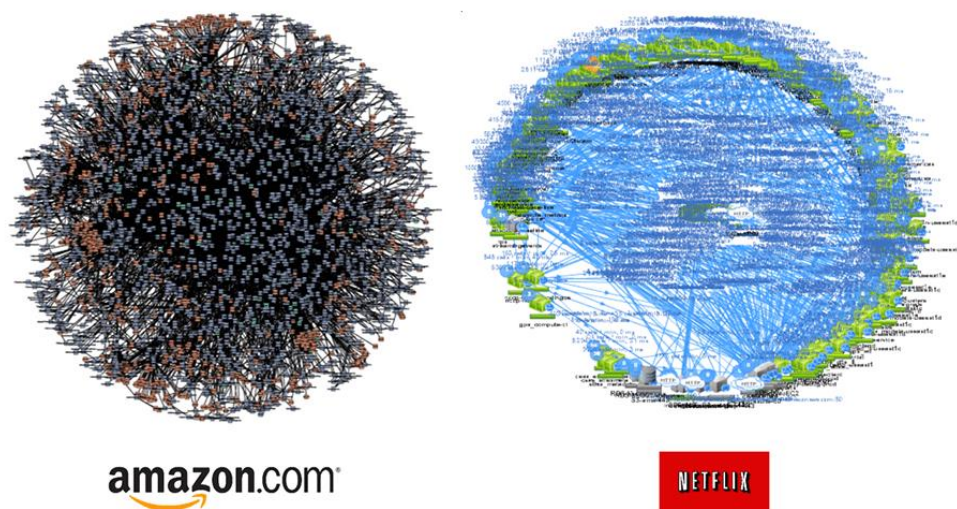


Рис. 1.1. Приблизне графічне зображення мікросервісної архітектури у великих компаніях

1.3 Аналіз методик розробки програмного забезпечення

Розробка програмного забезпечення (ПЗ) з використанням DevOps практик є сьогодні необхідністю для підтримки швидкості, гнучкості та якості в процесі розробки. Цей підхід дозволяє створювати та впроваджувати програми швидше, забезпечуючи високу якість та реагуючи на зміни вимог користувачів та ринкові потреби [1].

DevOps — це методологічний та культурний підхід у сфері розробки програмного забезпечення, що спрямований на покращення співпраці між розробниками програмного забезпечення (Dev — Development) та операторами (Ops — Operations) з метою автоматизації процесів розробки, тестування, поставки та моніторингу програм.

Основна мета DevOps — збільшення швидкості розробки та впровадження програм, підвищення їх якості та надійності, а також зменшення часу реакції на зміни вимог користувачів та ринкові потреби. Це досягається за допомогою автоматизації процесів та покращення співпраці між різними командами,

включаючи розробників програмного забезпечення, тестувальників, системних адміністраторів та інших учасників розробки та експлуатації програм[3].

DevOps орієнтується на різні принципи, такі як автоматизація, контейнеризація, інфраструктура як код, Continuous Integration (CI), Continuous Deployment (CD), моніторинг та зворотний зв'язок. Ці принципи спрямовані на створення надійного та ефективного процесу розробки програмного забезпечення, який дозволяє швидше реагувати на зміни та забезпечує стабільність та якість роботи програм.

DevOps впроваджує культурні зміни в організації, спрямовані на злагоджену роботу команди розробників, тестувальників та операторів, щоб автоматизувати та стандартизувати процеси розробки та експлуатації програм. Це сприяє зниженню часу на впровадження нового функціоналу та виправлення помилок, підвищує швидкість реакції на зміни та забезпечує стабільність роботи програм.

Інструменти DevOps дозволяють автоматизувати процеси збірки, тестування, розгортання та моніторингу програм, що робить розробку більш ефективною та прогнозованою. Це також сприяє реалізації концепцій Continuous Integration та Continuous Deployment, дозволяючи швидко виводити новий функціонал на реальне використання користувачами.

Мета DevOps полягає в досягненні кількох стратегічних цілей, які охоплюють усі етапи процесу постачання програмного забезпечення.

Основні цілі DevOps включають:

- швидкість виходу на ринок: DevOps спрямований на прискорення часу від ідеї до запуску продукту на ринок, забезпечуючи швидкий цикл розробки та впровадження;
- зменшення частоти відмов нових релізів: ця практика спрямована на зниження кількості помилок у випущених версіях програмного забезпечення;
- швидкість виправлення помилок: DevOps прагне зменшити час виявлення та виправлення помилок, щоб продукт був більш надійним та стабільним;

– мінімізація часу відновлення: при збоях чи відключенні системи, devops спрямований на забезпечення швидкого відновлення операційної працездатності;

Методологія DevOps робить процеси розробки та впровадження програмного забезпечення більш програмованими та динамічними. Вона дозволяє підвищити передбачуваність, ефективність, безпеку та можливість ремонту операційних процесів [13].

Застосування DevOps підходу відображає сучасні вимоги ринку до швидкості розробки та надійності програмного забезпечення. Це дозволяє компаніям забезпечити конкурентні переваги через швидше реагування на зміни та більш оперативне впровадження нововведень.

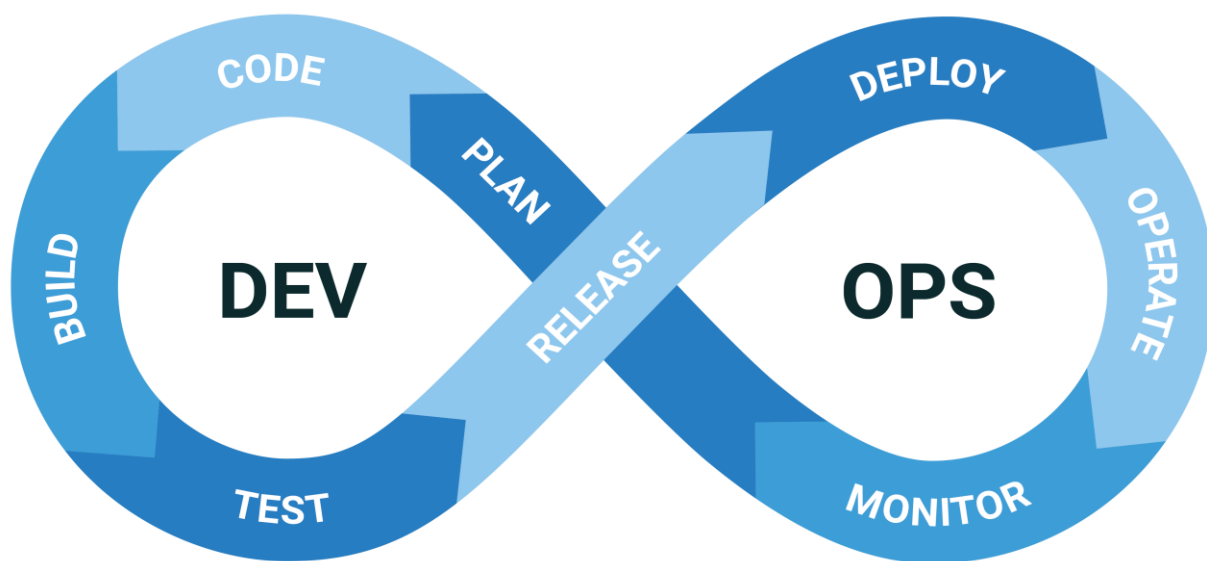


Рис. 1.2. Графічне зображення етапів DevOps

Процес DevOps включає в себе кілька етапів, що охоплюють усі аспекти життєвого циклу розробки програмного забезпечення:

- планування — це початковий етап, де визначаються цілі проекту, обсяг робіт, ресурси та терміни виконання;
- розробка полягає у створенні коду, тестуванні та вдосконаленні програм на основі вимог та специфікацій;

- побудова застосунку, цей етап включає в себе інструменти для безперервної інтеграції;
- тестування включає проведення тестів для перевірки працездатності, відповідності вимогам та виявлення помилок;
- інтеграція означає об'єднання компонентів програми в єдину систему;
- впровадження — це момент введення програми в дію в продуктивному середовищі. DevOps спрямований на автоматизацію цього процесу;
- моніторинг та оптимізація включають постійне спостереження за роботою програми, аналіз її продуктивності та вдосконалення на основі отриманих даних;

Кожен етап є важливим і доповнює інші, утворюючи цикл змін та покращень для досягнення високої якості та ефективності програмного забезпечення.

Не використання практик DevOps може призвести до кількох недоліків і проблем у розробці та впровадженні програмного забезпечення:

- повільна поставка продукту: відсутність автоматизації та оптимізованих процесів може призвести до затримок у поставці нових функцій або виправлень;
- низька якість програмного забезпечення: відсутність системи автоматизованого тестування та контролю якості може призвести до появи помилок та недоліків у випущеному продукті;
- нестабільність та ризик втрати даних: брак автоматизованих процесів резервного копіювання та відновлення може стати причиною втрати даних у випадку аварії системи або несправності;
- низька ефективність та надійність: відсутність автоматизованої моніторингу та управління може призвести до неефективного використання ресурсів та незабезпечення надійності роботи системи;
- складнощі у співпраці та комунікації: відсутність взаємодії між розробкою та оперативною групою може призвести до конфліктів, затримок у вирішенні проблем та неефективної комунікації;

– втрата конкурентоспроможності: у сучасному швидкому світі відсутність швидкого реагування та вдосконалення може призвести до втрати конкурентоспроможності компанії на ринку;

Загалом, відсутність впровадження DevOps може призвести до багатьох проблем, пов'язаних із продуктивністю, якістю та конкурентоспроможністю продукту та компанії в цілому [8].

1.4 Continuous Integration/Continuous Deployment

Continuous Integration (CI) — це практика в розробці програмного забезпечення, яка полягає в постійному об'єднанні коду та його автоматичній перевірці. Основна мета CI — забезпечення інтеграції змін у код з репозиторієм частіше та безперервно для підтримки якості та зниження ризику виникнення конфліктів між різними частинами програми [4].

Головний принцип CI — це регулярне злиття коду розробників в спільний репозиторій, за яким настає автоматичне виконання низки тестів. Це може включати модульні тести, інтеграційні тести, а також перевірки якості коду. Кожне злиття змін в коді призводить до автоматичного проведення цих тестів. Якщо тестування пройшло успішно, код інтегрується в основний репозиторій, що сприяє швидшому виявленню можливих помилок або конфліктів між кодом, які можуть виникнути через зміни.

Переваги CI включають зменшення часу, який витрачається на виявлення та виправлення помилок у коді, оскільки вони виявляються на ранніх стадіях розробки. Це також допомагає забезпечити стабільність кодової бази та знижує ризик внесення непередбачених змін в програму. Крім того, CI сприяє полегшенню командної роботи, оскільки вона змушує розробників регулярно об'єднувати свою роботу.

Враховуючи швидкі зміни та вимоги ринку, CI є важливою практикою, оскільки вона дозволяє розробникам зберігати високу якість коду та забезпечує швидкість реагування на зміни вимог користувачів.

Continuous Deployment/Delivery (CD) — це практика в розробці програмного забезпечення, спрямована на автоматизацію та спрощення процесу впровадження змін у програмне забезпечення в продуктивне середовище.

Continuous Deployment означає автоматичне розгортання кожної успішної зміни, яка пройшла всі необхідні тести і перевірки, безпосередньо на робочий сервер або середовище. Це означає, що кожна зміна, яка пройшла перевірку в рамках CI, автоматично впроваджується в реальне середовище без додаткового людського втручання. Цей підхід сприяє швидкому та безпечному впровадженню нового функціоналу або змін у програмі.

Continuous Delivery, у свою чергу, подібний до Continuous Deployment, проте рішення про розгортання вирішується людиною. Зміни готуються для автоматичного розгортання, проте факт їх впровадження в реальне середовище робиться за участю людини.

Головна мета CD — це забезпечення та автоматизація процесу впровадження нових функцій чи змін в програмі з мінімальними ризиками та максимальною швидкістю. Вона допомагає зберігати стабільність та доступність програми під час внесення змін, зменшує час від концепції до впровадження нових функцій, та полегшує процес управління версіями програмного забезпечення [4].

1.5 Аналіз хмарних технологій

Хмарні послуги від провайдерів, таких як Amazon Web Services, Microsoft Azure, Google Cloud, надають безліч можливостей для користувачів. Ці провайдери пропонують різні сервіси, включаючи хмарне сховище, обчислювальні ресурси, машинне навчання та інші. Amazon Web Services (AWS) надає широкий спектр послуг, включаючи Amazon S3 для сховища, EC2 для обчислювання та AWS Lambda для безкомп'ютерних(Serverless) обчислень. Однак, кожен провайдер має свої унікальні переваги та обмеження, які можуть бути вирішальними для різних видів додатків та бізнес-потреб. Наприклад, Azure

від Microsoft спеціалізується на інтеграції з усіма іншими продуктами Microsoft, тоді як Google Cloud від Google має потужність у сфері штучного інтелекту та аналізу даних та інтеграції з внутрішніми сервісами Google.

Хмарні технології значно спрощують розробку та розгортання додатків. Вони дозволяють розробникам мати доступ до різних сервісів через мережу, прискорюючи процес розробки та знижуючи витрати на обладнання. Це також дозволяє масштабувати додатки за потреби, забезпечуючи доступність та відмовостійкість. Хмарні технології впливають на архітектуру додатків, дозволяючи створювати більш гнучкі та масштабовані системи, що можуть забезпечувати швидку реакцію на зміни бізнес-вимог.

Amazon Web Services є одним з найбільших та найпопулярніших хмарних провайдерів. Цей сервіс пропонує широкий спектр послуг, таких як хмарне сховище (Amazon S3), обчислювальні ресурси (EC2), аналіз даних (Amazon Redshift), машинне навчання (Amazon SageMaker) та інші. AWS відомий своєю широкою географічною доступністю центрів обробки даних, роблячи їх привабливим вибором для глобальних бізнесів. Однак, слабкі сторони включають складність ціноутворення та можливість створення складних архітектур, які можуть вимагати додаткової конфігурації та управління [24].

Microsoft Azure пропонує послуги, орієнтовані на інтеграцію з іншими продуктами Microsoft та широкий спектр інструментів для розробки, тестування та розгортання додатків. Azure пропонує хмарне сховище (Azure Blob Storage), обчислювальні ресурси (Virtual Machines), інструменти для машинного навчання (Azure Machine Learning) та інші. Вони часто підкреслюють свою сумісність з існуючими рішеннями у сфері корпоративних технологій, проте можуть мати обмежені можливості в порівнянні з AWS або Google Cloud у певних областях, таких як масштабування та широкий спектр сервісів.

Google Cloud Platform пропонує послуги, які акцентуються на штучному інтелекті, аналізі даних та інструментах для розробки. Платформа має хмарне сховище (Google Cloud Storage), обчислювальні ресурси (Compute Engine),

інструменти машинного навчання (Google AI Platform) та інші. Google Cloud відомий своєю високою швидкістю та ефективністю в роботі з великими обсягами даних, проте може бути менш диверсифікованим порівняно з AWS чи Microsoft Azure в інших сегментах послуг.

Кожен провайдер має свої унікальні переваги та слабкі сторони, і вибір хмарного провайдера може залежати від конкретних потреб, обсягу роботи, специфіки бізнесу та бюджетних обмежень компанії.

Таблиця 1.2

Порівняння послуг хмарних провайдерів

Критерії	Amazon Web Services (AWS)	Microsoft Azure	Google Cloud Platform (GCP)
Різноманітність сервісів	Широкий спектр послуг і продуктів, включаючи обчислення, зберігання, бази даних, штучний інтелект, інтернет речей тощо.	Послуги інфраструктури, штучного інтелекту, аналізу даних, розробки, безпеки тощо.	Великий спектр послуг, включаючи обчислення, зберігання, машинне навчання, аналіз даних, інтернет речей та інше.
Глобальна присутність	Наявність регіонів та зон для обслуговування в більшій кількості країн, що дозволяє глобально розгортати рішення.	Широка географічна присутність, проте деякі регіони можуть бути менш розвинуті.	Значна кількість регіонів, що дозволяє розгортати додатки в різних точках світу.
Ціна	Конкурентоспроможні ціни, але в деяких випадках можуть бути вищими в порівнянні з іншими хмарними провайдерами.	Різноманітність моделей ціноутворення, зі значною кількістю безкоштовних послуг та знижок для нових користувачів.	Часто пропонує конкурентоспроможні ціни, але може бути менш конкурентоспроможним для деяких послуг порівняно з іншими провайдерами.
Аналітика та інтелектуальний аналіз	Послуги для обробки великих обсягів даних, машинного навчання та інші аналітичні інструменти.	Продукти та інструменти для аналітики даних, штучного інтелекту, машинного навчання.	Машинне навчання, інструменти для аналізу даних та штучний інтелект.
Управління сервісами	AWS Management Console - веб-інтерфейс для управління всіма послугами AWS.	Azure Portal - центральний інтерфейс управління послугами Microsoft Azure.	Google Cloud Console - інтерфейс для управління послугами та ресурсами Google Cloud.

1.6 Kubernetes кластер

Kubernetes є відкритою платформою для оркестрації контейнеризованих додатків, яка дозволяє ефективно керувати та управляти цими додатками у великому масштабі. Це програмне забезпечення дає можливість розгортання, масштабування і автоматизацію управління контейнерами. Kubernetes використовується для керування архітектурою мікросервісів, де складні додатки розбиваються на невеликі компоненти, які запуснені у віртуальних середовищах [21].

Ця система дозволяє створювати та керувати подами (pods) — групами контейнерів, які спільно працюють, а також визначає мережеві політики, ресурси, моніторинг та інші параметри для кожного пода. Kubernetes забезпечує рівновагу навантаження, автоматичну масштабованість, самовідновлення в разі збоїв та гнучкість в управлінні ресурсами.

Технічно, Kubernetes має архітектуру майстра-слуги (master-slave), де майстер-компонент (Master Node) керує кластером і контролює додавання нових контейнерів та їх розміщення на вузлах (Node). Вузли виконують обчислювальні завдання, приймаючи команди від майстра та запускаючи контейнери відповідно до специфікацій [21].

Основними перевагами Kubernetes є спрощене управління контейнеризованими додатками, забезпечення високої доступності, автоматичне масштабування, безпека та універсальність у роботі з різними типами інфраструктури. Тим самим, використання Kubernetes дозволяє підвищити ефективність та надійність роботи інфраструктури та додатків компанії.

Зважаючи на різні варіанти використання kubernetes, розглянемо їх детальніше:

- Minikube: це інструмент, який дозволяє створювати одноузловий кластер kubernetes на локальному комп'ютері. Minikube корисний для розробників, оскільки він дозволяє створити ізольоване середовище для розгортання, тестування та експериментів з додатками. Користувач може локально перевірити

роботу свого додатку в середовищі, подібному до реального хмарного кластера Kubernetes.

- Elastic Kubernetes Service (EKS): це керований сервіс Kubernetes, який пропонується amazon web services (AWS). Eks дозволяє швидко створювати та керувати кластерами Kubernetes на інфраструктурі AWS. Це рішення забезпечує безпеку, масштабованість та високу доступність кластерів у хмарному середовищі AWS;
- Google Kubernetes Engine (GKE): це керований сервіс Kubernetes від Google Cloud Platform (GCP). GKE дозволяє розгортати, керувати та масштабувати кластери Kubernetes на інфраструктурі Google Cloud. Він забезпечує автоматичне масштабування ресурсів, моніторинг, безпеку та інші функції для ефективного управління кластерами
- Azure Kubernetes Service (AKS): це аналогічний сервіс до EKS і GKE від Microsoft Azure. AKS дозволяє створювати та керувати кластерами Kubernetes на інфраструктурі Azure, забезпечуючи можливість розгортання та управління контейнеризованими додатками в хмарному середовищі Azure;

Ці різні варіанти використання Kubernetes надають можливість адміністраторам кластеру чи розробникам обирати найбільш відповідний інструмент для своїх конкретних потреб — від локальної розробки до масштабування систем в хмарному середовищі.

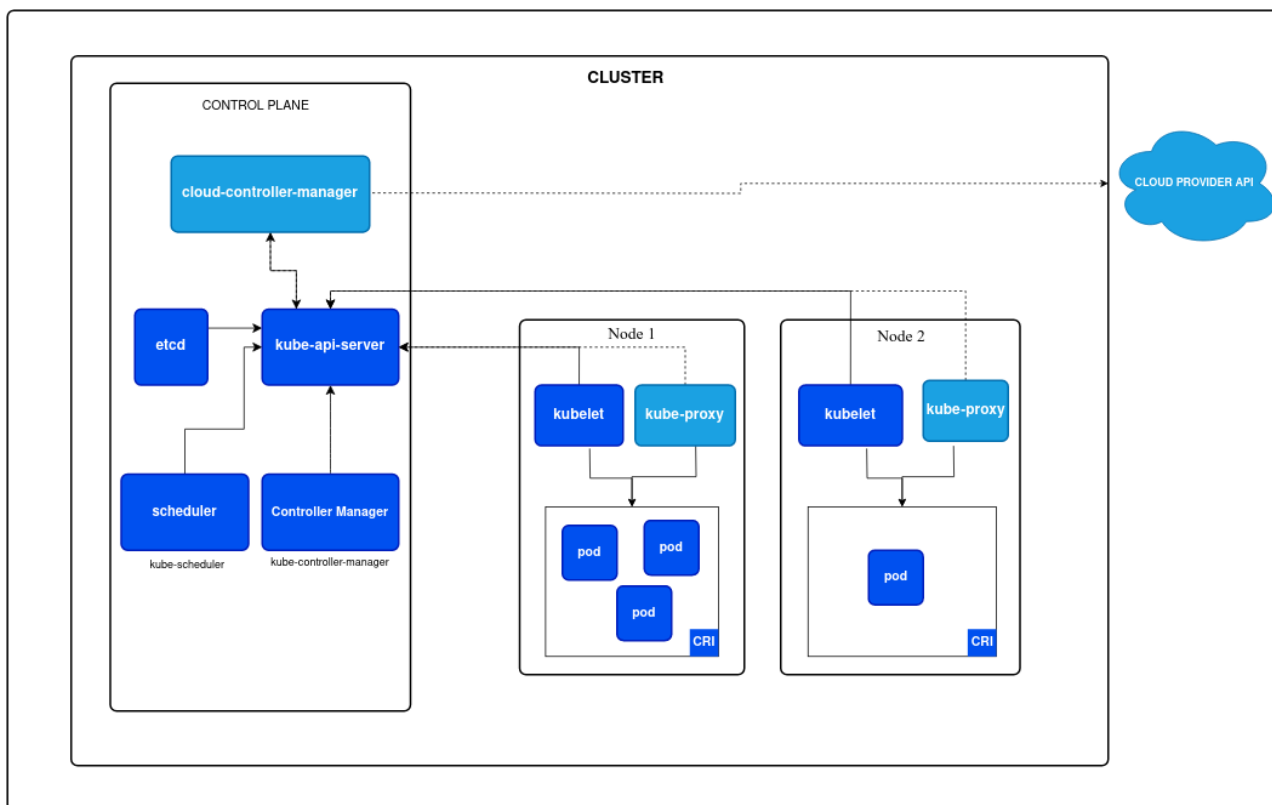


Рис. 1.3. Архітектура кластеру Kubernetes

Архітектура Kubernetes складається з кількох ключових компонентів, які співпрацюють для керування та управління контейнеризованими додатками.

Основні компоненти архітектури Kubernetes включають:

– майстер-вузол (control plane) — центральний елемент керування. Включає декілька компонентів:

- Kube-apiserver: сервер API є компонентом площини керування Kubernetes, який надає доступ до API Kubernetes. Сервер API є передньою частиною для площини керування Kubernetes. Основною реалізацією сервера Kubernetes API є kube-apiserver. Kube-apiserver розроблено для горизонтального масштабування, тобто він масштабується шляхом розгортання більшої кількості екземплярів. Можливо запустити кілька модулів kube-apiserver і збалансувати трафік між цими модулями [21];

- Etcd: послідовне та високодоступне сховище ключових значень, яке використовується як резервне сховище Kubernetes для всіх даних кластера [21];
 - Kube-scheduler: компонент, який спостерігає за новоствореними контейнерами без призначеного вузла та вибирає вузол для їх запуску. Фактори, які враховуються для прийняття рішень щодо планування, включають: індивідуальні та колективні вимоги до ресурсів, обмеження апаратного/програмного забезпечення/політики, специфікації спорідненості та антиспорідненості, локальність даних, перешкоди між робочими навантаженнями та кінцеві терміни [21];
 - Kube-controller-manager: це процес, який вбудовує основні цикли керування, які постачаються з Kubernetes. У додатках робототехніки та автоматизації контур керування — це непереривний контур, який регулює стан системи. У Kubernetes контролер — це цикл керування, який спостерігає за спільним станом кластера через apiserver і вносить зміни, намагаючись перемістити поточний стан до бажаного. Прикладами контролерів, які сьогодні постачаються з Kubernetes, є контролер реплікації, контролер кінцевих точок, контролер простору імен і контролер сервісних облікових записів [21];
- вузли (node): фізичні або віртуальні машини, на яких розгортаються контейнери. Включають декілька компонентів:
- Kubelet: агент, який співпрацює з майстер-вузлом та забезпечує керування контейнерами на вузлі [21];
 - Kube-proxy: проксі мережі Kubernetes, який працює на кожному вузлі. Це відображає служби, визначені в API Kubernetes на кожному вузлі, і може виконувати просте пересилання потоку TCP, UDP і SCTP або циклічне пересилання TCP, UDP і SCTP через набір серверних модулів. IP-адреси та порти кластерів служб можна знайти за допомогою змінних середовища, сумісних із посиланнями Docker, які визначають порти,

відкриті проксі-сервером служби. Існує додатковий аддон, який надає DNS кластера для цих IP-адрес кластера. Користувач повинен створити службу з API apiserver, щоб налаштувати проксі [21];

- Container runtime: основний компонент, який дозволяє Kubernetes ефективно запускати контейнери. Він відповідає за керування виконанням і життєвим циклом контейнерів у середовищі Kubernetes [21];

Ці компоненти працюють разом для керування та управління контейнеризованими додатками у кластері Kubernetes, забезпечуючи масштабованість, надійність та управління ресурсами.

1.7 Об'єкти Kubernetes

Об'єкти Kubernetes — це абстракції, які використовуються для опису стану системи та ресурсів, які потрібно керувати. Кожен об'єкт має свою унікальну функцію та використовується для конфігурації, розгортання, моніторингу, керування ресурсами та взаємодії між компонентами кластера. Кожен об'єкт має свою унікальну роль і призначення в екосистемі Kubernetes. Вони дозволяють користувачам описувати, розгортати, моніторити та управляти різними компонентами додатків та ресурсами кластера. Об'єкти дозволяють користувачам описати стан, конфігурацію та потреби додатків та сервісів, які розгортаються в Kubernetes кластері. Вони дозволяють описувати потреби, структуру, конфігурацію та залежності між частинами додатку або сервісу, що дозволяє ефективно управляти ними в різних умовах та сценаріях роботи. Об'єкти Kubernetes дозволяють створювати та підтримувати складні і розширювані додатки в інфраструктурі контейнерів [6].

Под (pod) в Kubernetes — це базова одиниця роботи, яка об'єднує один або декілька контейнерів, які спільно взаємодіють та мають спільний контекст для мережі та простору імен(namespace). Детальні аспекти, які характеризують pod:

- контейнери в поді: под може містити один або кілька контейнерів, які спільно працюють та мають спільні ресурси, такі як ір-адреси та простори імен;
- спільна мережа: контейнери в поді мають спільний мережевий стек. Вони можуть зв'язуватися один з одним через localhost і використовувати внутрішню мережу для спілкування;
- спільний простір імен процесів: контейнери в поді бачать один одного через спільний простір імен процесів, що дозволяє їм спільно виконувати задачі та взаємодіяти;
- рівень ізоляції: под надає відносну ізоляцію для свого вмісту, проте контейнери всередині пода ділять загальний ресурс;

- життєвий цикл: под є тимчасовою або постійною оболонкою, яка виконується на вузлі Kubernetes. Він може бути створений, змінений або видалений у відповідь на потреби додатку або відтворення;
- спільний додатковий контекст: под може мати додаткові об'єкти, такі як томи для зберігання даних або конфігмапи для налаштування контейнерів;
- специфікація под: у специфікації пода описується конфігурація, така як образ контейнера, порти, об'єми, інструкції запуску та інше;
- моніторинг та управління: Kubernetes надає можливість моніторити та управляти подами для забезпечення їх стану, розгортання та надійності;

Под — це базова абстрація у Kubernetes, що надає ізольоване середовище для виконання контейнерів, забезпечуючи їх взаємодію та спільність. Спільний контекст Pod — це набір просторів імен Linux, контрольних груп і потенційно інших аспектів ізоляції — тих самих речей, які ізолюють контейнер. У контексті модуля окремі програми можуть застосовувати додаткові субізоляції. Pod схожий на набір контейнерів зі спільними просторами імен і спільними томами файлової системи [6].

Деплоймент в контексті Kubernetes описується як об'єкт, що визначає, як контейнеризовані додатки будуть розгортатися та оновлюватися в кластері Kubernetes. Це ключовий об'єкт конфігурації, який дозволяє автоматизувати і контролювати розгортання програмних застосунків, надаючи декларативний спосіб описувати бажаний стан системи.

Деплойменти в Kubernetes використовуються для опису параметрів розгортання, таких як кількість реплік (екземплярів) додатків, контейнери, що використовуються для створення цих екземплярів, образи контейнерів, політику оновлення та управління версіями. Вони дозволяють забезпечити стабільність, масштабованість та безперервність роботи додатків під час їхнього розгортання, оновлення та експлуатації в середовищі Kubernetes.

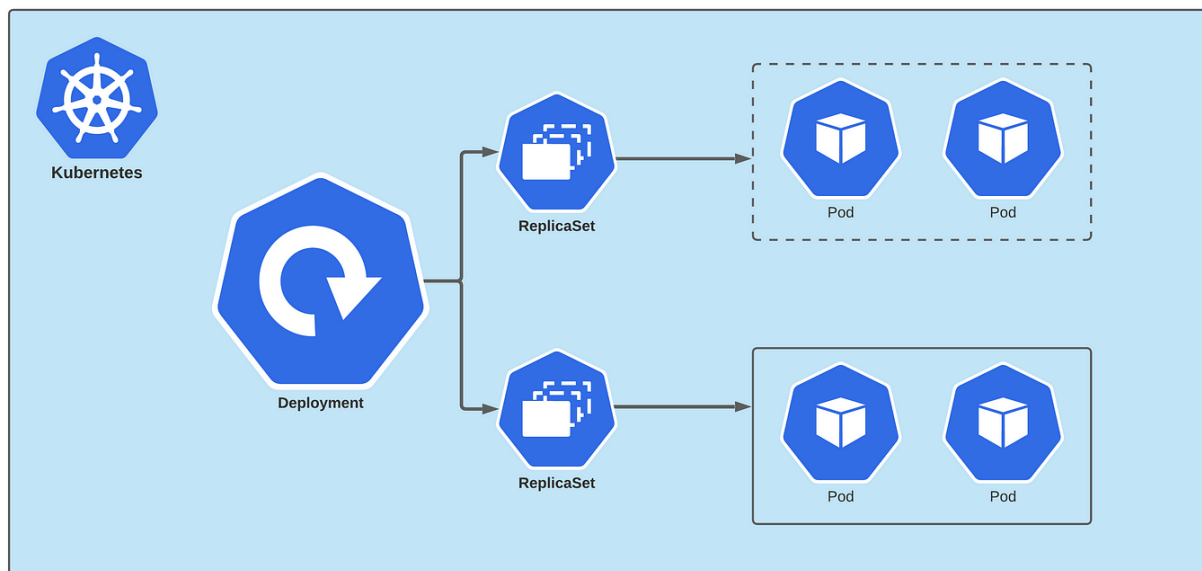


Рис. 1.4. Процес розгортання додатку за допомогою абстрації Kubernetes деплоймент

Деплойменти автоматично контролюють стан додатків у Kubernetes-кластері, забезпечуючи автоматичне створення, оновлення або видалення екземплярів додатків, які описані у конфігураційному файлі деплойменту. Вони відповідають за реалізацію і контроль життєвого циклу додатків, управляючи їхнім розгортанням, масштабуванням та оновленнями [6].

Особливістю деплойментів у Kubernetes є їхня здатність забезпечити неперервність і безперервність процесу оновлення програмних застосунків. Вони дозволяють організувати рівномірне оновлення додатків, контролювати процеси розгортання та відслідковувати стан кожного компонента системи, забезпечуючи безперебійну роботу навіть під час впровадження оновлень.

Деплойменти в Kubernetes є важливим інструментом управління і роботи з контейнеризованими додатками. Важливі аспекти, які можна включити у розгляд про деплоймент:

- автоматичне оновлення: деплойменти дозволяють автоматизувати процеси оновлення програмних застосунків. Вони визначають опис стану системи, і Kubernetes вживає необхідних заходів для забезпечення відповідності цьому стану;

- стратегії оновлення: Kubernetes надає кілька стратегій оновлення, таких як rolling update (плавне оновлення), recreate (повне перезавантаження), blue/green deployment (паралельне розгортання), і canary deployment (поступове розгортання), які дозволяють здійснювати оновлення з мінімальним впливом на роботу додатків;
- моніторинг і логування: завдяки можливостям Kubernetes, можна налаштовувати моніторинг і логування деплойментів, щоб відстежувати їхню продуктивність та стан;
- шаблонізація: використання шаблонів конфігурації дозволяє зручно налаштовувати та повторно використовувати конфігураційні файли деплойменту для різних аспектів роботи додатків;
- імутабельність: підходи, які використовують деплойменти, сприяють імутабельності, що означає, що стан додатків не змінюється в процесі роботи, а зміни робляться шляхом створення нових екземплярів;
- управління ресурсами: Kubernetes дозволяє управляти ресурсами, надаючи можливість налаштування рівня обчислювальних та мережевих ресурсів, які виділені для кожного пода;

Ці аспекти дозволяють краще розуміти можливості деплойментів у Kubernetes та їх вплив на ефективність, безпеку та стабільність роботи контейнеризованих додатків.

Об'єкт сервіс (Service) в Kubernetes — це абстракція, яка дозволяє забезпечити постійний доступ до групи робочих контейнерів (Pods) за допомогою стабільної точки доступу.

Основні аспекти сервісу в Kubernetes:

- балансування навантаження: сервіс автоматично розподіляє вхідний трафік між контейнерами, які входять до групи. Це дозволяє вирівнювати навантаження і забезпечувати більшу доступність додатку;

- стійкість до змін в конфігурації: якщо конфігурація контейнерів (таких як IP-адреси чи порти) змінюється, Сервіс автоматично оновлює свою конфігурацію, щоб врахувати ці зміни;
- типи сервісів: Kubernetes підтримує кілька типів сервісів, таких як ClusterIP, NodePort, LoadBalancer, та ExternalName. Кожен з цих типів має свої особливості і застосування в залежності від потреб;
- доступ до сервісів всередині кластеру: сервіси дозволяють іншим ресурсам Kubernetes взаємодіяти з групою контейнерів, що є частиною Сервісу, без необхідності знати їхні індивідуальні IP-адреси чи деталі конфігурації;
- стабільний DNS-запис: кожен сервіс має власне ім'я DNS, що дозволяє іншим додаткам звертатися до нього за цим ім'ям, не зважаючи на зміни в конфігурації контейнерів;
- можливість зовнішнього доступу: деякі типи сервісів, такі як LoadBalancer, дозволяють надавати доступ до групи контейнерів із зовнішньої мережі;

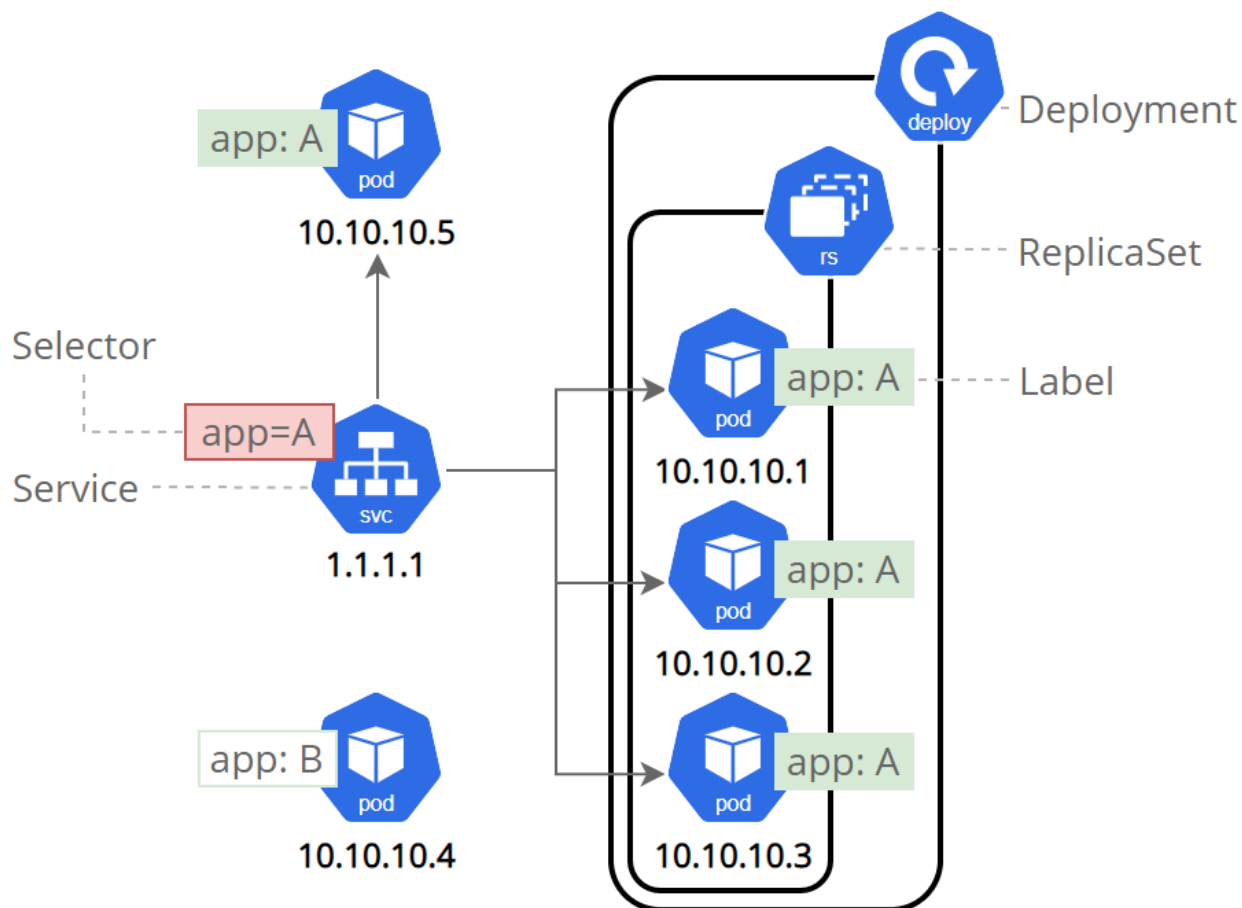


Рис. 1.5. Використання сервісів Kubernetes

Сервіси в Kubernetes є важливим елементом для забезпечення стабільності та доступності додатків, які працюють у середовищі контейнерів. Вони дозволяють ізолювати та управляти доступом до різних частин додатку, що працюють в Kubernetes-кластері [6].

1.8 GitOps

Більша частина життєвого циклу розробки програмного забезпечення була автоматизована, інфраструктура залишалася здебільшого ручним процесом, який потребує спеціалізованих команд. З огляду на вимоги до сучасної інфраструктури, автоматизація інфраструктури стає все більш важливою. Сучасна інфраструктура має бути еластичною, щоб вона могла ефективно керувати хмарними ресурсами, необхідними для безперервного розгортання [20].

Сучасні хмарні додатки розроблено з урахуванням швидкості та масштабу. Організації зі зрілою культурою DevOps можуть розгорнути код у реальних

середовищах сотні разів на день. Команди DevOps можуть досягти цього за допомогою найкращих практик розробки, таких як контроль версій, перегляд коду та методів CI/CD, які автоматизують тестування та розгортання.

GitOps використовується для автоматизації процесу надання інфраструктури, особливо сучасної хмарної інфраструктури. Подібно до того, як команди використовують вихідний код програми, операційні групи, які використовують GitOps, використовують файли конфігурації, що зберігаються як код (інфраструктура як код). Конфігураційні файли GitOps генерують те саме середовище інфраструктури кожного разу, коли його розгортають, так само, як вихідний код програми генерує ті самі двійкові файли програми кожного разу, коли його збирають [20].

GitOps — це підхід до управління інфраструктурою та процесом розгортання програмного забезпечення, який базується на використанні системи контролю версій Git. Цей підхід дозволяє автоматизувати процеси розгортання, оновлення та керування конфігурацією програмних систем і інфраструктури.

У GitOps, конфігурація системи, інфраструктура та різні параметри описуються у вигляді коду (часто у вигляді YAML-файлів) та зберігаються в Git-репозиторії. Цей підхід робить інфраструктуру та конфігурацію програмного забезпечення перевіряємими, версіонованими та відновлюваними. Кожен аспект системи може бути описаний у вигляді коду і зберігатися в репозиторії, що дозволяє відслідковувати зміни, вносити виправлення та впроваджувати нові функції шляхом простого застосування змін у Git [9].

GitOps дозволяє створювати декларативні описи потрібного стану системи та автоматизувати процеси синхронізації стану інфраструктури з описом у Git. Це забезпечує однорідність стану інфраструктури, прозору історію змін, легкість відкату та відновлення системи до попередніх станів.

Однією з ключових переваг GitOps є забезпечення більшої безпеки та надійності, оскільки кожна зміна пройде через код-ревізію та процеси затвердження. Крім того, цей підхід спрощує роботу розробників та

адміністраторів, оскільки усі зміни у системі керуються через один інструмент Git.

GitOps має такі основні аспекти:

- декларативна конфігурація: GitOps використовує декларативний підхід до конфігурації системи, де стан системи описується у вигляді коду (наприклад, YAML-файли), які зберігаються в репозиторії Git. Ця концепція спрощує управління конфігурацією та забезпечує стійкість до помилок через використання версій та перевірку змін;
- операції через Git: Усі зміни в конфігурації та коді обробляються через Git. Це означає, що зміни в конфігурації системи, такі як зміни параметрів середовища, розгортання нових версій додатків тощо, відбуваються шляхом коміту змін у відповідний репозиторій;
- інтеграція з CI/CD [9]: GitOps використовує CI/CD практики для автоматизації процесів розгортання та оновлення системи. Після внесення змін до конфігураційних файлів у репозиторії Git, CI/CD процес автоматично спрацьовує, тестує, пакує та розгортає нові версії додатків або інфраструктури;
- синхронізація стану: система GitOps володіє можливістю автоматично синхронізувати стан інфраструктури та додатків з відповідним станом, описаним у репозиторії Git. Це забезпечує послідовність та точність стану системи у відповідності з вимогами;
- аудит та відстеження змін: завдяки використанню Git для управління конфігурацією, GitOps забезпечує можливість аудиту та відстеження всіх змін, включаючи автора змін, дату та час внесення змін, що є важливим для контролю і безпеки системи;

GitOps спрощує розгортання та керування інфраструктурою, забезпечуючи повторюваність, автоматизацію та стабільність у процесах розробки та розгортання програмного забезпечення.

Як і будь-який новий технологічний термін, GitOps не визначається суворо однаково всіма представниками галузі. Принципи GitOps можна застосовувати

до всіх типів автоматизації інфраструктури, включаючи віртуальні машини та контейнери, і можуть бути дуже ефективними для команд, які хочуть керувати інфраструктурою на основі Kubernetes. У той час як багато інструментів і методологій обіцяють швидше розгортання та безперебійне керування між кодом та інфраструктурою, GitOps відрізняється тим, що зосереджується на досвіді, орієнтованому на розробника. Управління інфраструктурою через GitOps відбувається в тій самій системі контролю версій, що й розробка додатків, що дає змогу командам ширше співпрацювати в централізованому місці, користуючись перевагами вбудованих функцій Git [20].

Як відбувається синхронізація стану між системою для контролю версій та робочим середовищем? Коли зміни в цьому репозиторії відбуваються, інструменти GitOps автоматично виявляють ці зміни та розгортають нову версію програмного забезпечення або змінюють стан інфраструктури у відповідності з визначеними у репозиторії налаштуваннями.

Такий підхід дозволяє здійснювати розгортання коду в кластер Kubernetes таким чином:

- опис конфігурації в Git-репозиторії: розробники визначають конфігураційні файли (наприклад, YAML-файли) для ресурсів Kubernetes (наприклад, Pod, Deployment, Service тощо) і зберігають їх у репозиторії Git;
- автоматичне виявлення змін: система GitOps постійно слідкує за змінами у відповідному Git-репозиторії. Якщо відбуваються зміни, GitOps-інструменти реагують на них;
- синхронізація стану Kubernetes з Git-репозиторієм: при виявленні змін інструменти GitOps виконують процес синхронізації між конфігураційними файлами у Git та станом Kubernetes. Вони можуть ініціювати створення нових ресурсів, оновлення чи видалення існуючих, щоб забезпечити, що актуальний стан кластера відповідає опису у Git-репозиторії;

– автоматизація процесу деплою: цей процес відбувається автоматично без необхідності вручному втручанні, що дозволяє покращити швидкість та надійність розгортання, а також забезпечує консистентність стану системи;

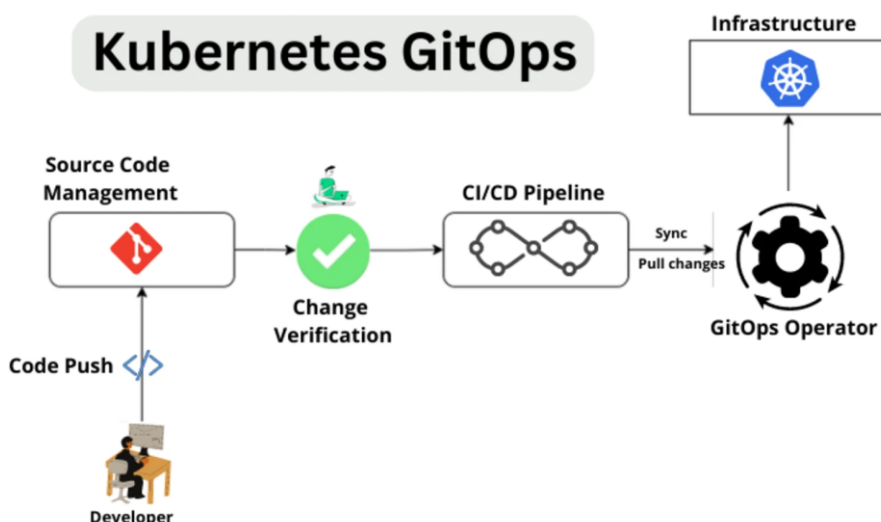


Рис. 1.6. Графічне зображення реалізації GitOps

Отже, GitOps дозволяє автоматизувати процеси розгортання, оновлення та управління станом Kubernetes-кластера за допомогою конфігураційних файлів, що зберігаються у Git-репозиторії.

1.9 Висновок до розділу 1

Отже, в цьому розділі було виконано аналіз ключових аспектів сучасної розробки програмного забезпечення, який виявив велику вагомість методів, які спрямовані на оптимізацію та підвищення ефективності у процесах створення, тестування та розгортання програм. Передові технології, такі як Continuous Integration/Continuous Deployment (CI/CD), мікросервісна архітектура, DevOps практики, хмарні рішення, Kubernetes та GitOps, не лише прискорюють ці процеси, але й забезпечують високу якість продукту та гнучкість у відповіді на змінність вимог користувачів та ринкових умов.

Впровадження CI/CD дозволяє регулярно та безперервно інтегрувати зміни в коді програм, забезпечуючи стабільність та зниження ризику виникнення конфліктів. Мікросервісна архітектура дозволяє розбити складні додатки на більш керовані та легше управляються компоненти, що полегшує розробку та

тестування. DevOps підхід дозволяє поєднувати у роботу розробників і операторів для автоматизації процесів, що веде до швидкого реагування на потреби ринку.

Хмарні технології відкривають широкий спектр можливостей, проте вибір провайдера впливає на підбір оптимального рішення для конкретного додатку чи бізнес-потреб. Kubernetes дозволяє ефективно керувати контейнеризованими додатками у великому масштабі, використовуючи об'єкти для оптимізації та керування ресурсами.

Завдяки підходу GitOps, автоматизація інфраструктури стає більш важливою, забезпечуючи еластичність та ефективне управління хмарними ресурсами для безперервного розгортання.

Узагальнюючи, ці технології і методики визначають нові стандарти у розробці програмного забезпечення, дозволяючи створювати, тестувати та розгортати програми з високою якістю, ефективністю та здатністю адаптуватися до змін, що відбуваються в індустрії та вимог користувачів.

РОЗДІЛ 2 РОЗРОБКА АРХІТЕКТУРИ СИСТЕМИ ТА РІШЕННЯ ДЛЯ CI/CD В ХМАРНЕ ОТОЧЕННЯ

2.1 Розробка архітектури системи

Розробка системи для деплою в Kubernetes кластер може включати кілька компонентів та процесів для ефективного управління розгортанням програмного забезпечення. Основні етапи розгортання включають побудову pipeline для автоматизації процесу, від збирання коду до створення Docker-образів і генерації маніфестів Kubernetes для автоматичного розгортання. Інтеграція з Kubernetes API дозволяє взаємодіяти з його ресурсами для управління та розгортання артефактів. Ось ключові вимоги до архітектури такої системи:

- інтеграція з Git-репозиторієм: система повинна вміти взаємодіяти з обраним Git-репозиторієм, виявляти зміни у конфігураційних файлах, які відповідають за деплоймент;
- автоматичне виявлення змін: наявність механізму, що постійно відстежує зміни в обраному репозиторії. Це може бути компонент, який періодично перевіряє Git на наявність нових комітів чи подій;
- pipeline для автоматизації процесу: створення pipeline, який охоплює весь процес від виявлення змін до розгортання в Kubernetes. Це включає етапи: збірка коду, тестування, створення Docker-образів, створення маніфестів Kubernetes і їх розгортання;
- інтеграція з Kubernetes API: система повинна вміти взаємодіяти з API Kubernetes для створення, оновлення, та видалення ресурсів, таких як Deployment, Service, ConfigMap тощо;
- моніторинг і логування: додавання компонентів для моніторингу та логування процесу розгортання в Kubernetes кластер. Це допоможе виявляти помилки, відстежувати статуси розгортання і забезпечити безперервність процесу;

- безпека: розробка механізмів безпеки для системи деплою в Kubernetes, що включає управління доступом, шифрування та інші аспекти безпеки інфраструктури.;
- система керування конфігурацією: можливість зберігати та керувати конфігураційними параметрами, які використовуються під час розгортання;
- резервне копіювання та відновлення: механізми для резервного копіювання конфігурацій, щоб уникнути втрати даних або конфігурацій у випадку непередбачуваних ситуацій;

Зазначені аспекти важливі для розробки системи, яка здійснюватиме деплой в Kubernetes кластер, і їх узгоджений функціонал може забезпечити ефективність, надійність та безпеку процесу успішного деплою [6].

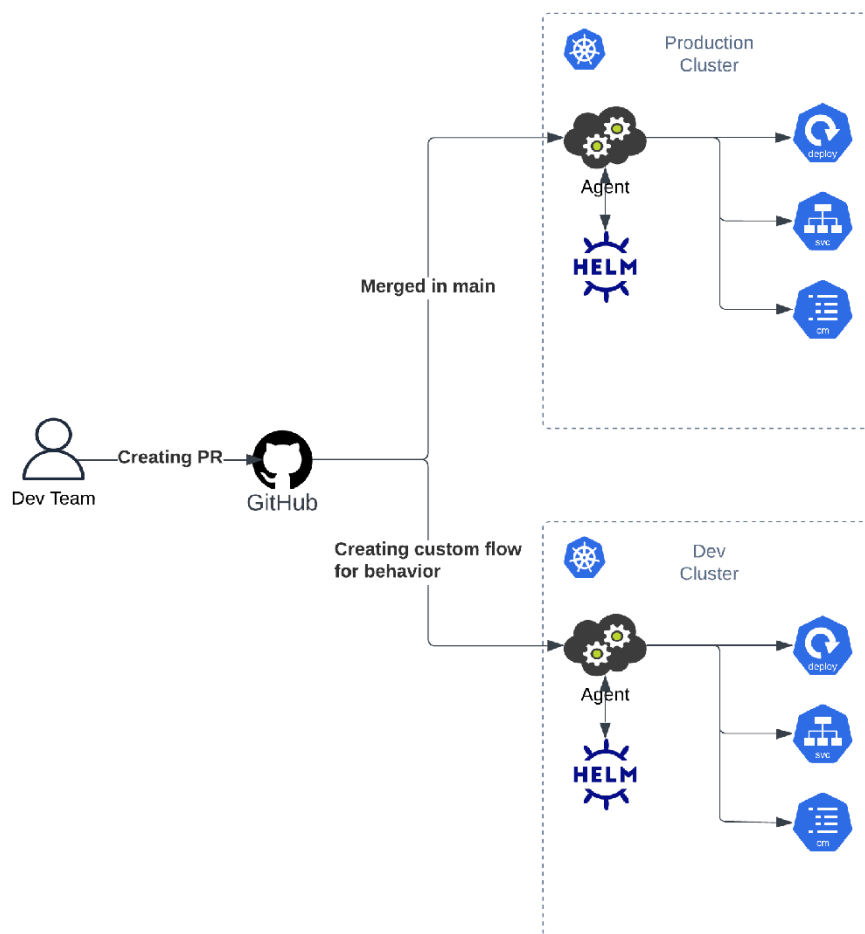


Рис. 2.1. Архітектура роботи системи

Такий підхід до автоматичного деплою в гілку є типовим для процесу CI/CD та GitOps. Коли відбувається зміна в гілці, агент чи CI/CD пайплайн виявляє зміни у відповідному Git-репозиторії. Після цього виконується автоматичний процес, що охоплює збірку, тестування, створення Docker-образу та автоматичне оновлення Kubernetes кластера з використанням нової версії додатку.

Така архітектура (рис.2.1.) є типовим використанням інструментів GitOps, такі як Flux або ArgoCD, які слідкують за Git-репозиторієм, аналізують зміни та автоматично оновлюють стан кластера відповідно до конфігураційних змін. Цей підхід дозволяє швидко та безпечно впроваджувати зміни в тестові середовища з використанням механізмів контролю версій і автоматизації.

У випадку сервісів GitOps, таких як Flux та ArgoCD, графічні схеми можуть бути корисними для наочного представлення їх архітектури, зокрема:

- компоненти системи: вказання основних складових частин сервісу та їх залежностей;
- потоки даних або процесів: відображення процесів або потоків даних, що відбуваються всередині сервісу;
- взаємодія з іншими системами: показ взаємодії з іншими сервісами, зокрема з Git-репозиторієм, Kubernetes кластером та іншими елементами інфраструктури;

ArgoCD — це інструмент для Continuous Delivery, спеціально призначений для Kubernetes. Він дозволяє автоматизувати процеси розгортання, моніторингу та управління додатками в Kubernetes-кластері, відповідно до конфігураційних файлів, які зберігаються у Git-репозиторії. ArgoCD працює на основі декларативного підходу, спостерігаючи за змінами в Git-репозиторії та автоматично забезпечуючи, щоб стан додатків у кластері відповідав їх конфігурації у репозиторії. Він має веб-інтерфейс для спостереження, керування версіями та проведення операцій оновлення, а також підтримує інтеграцію з Helm для керування додатками за допомогою Helm-чартів [10].

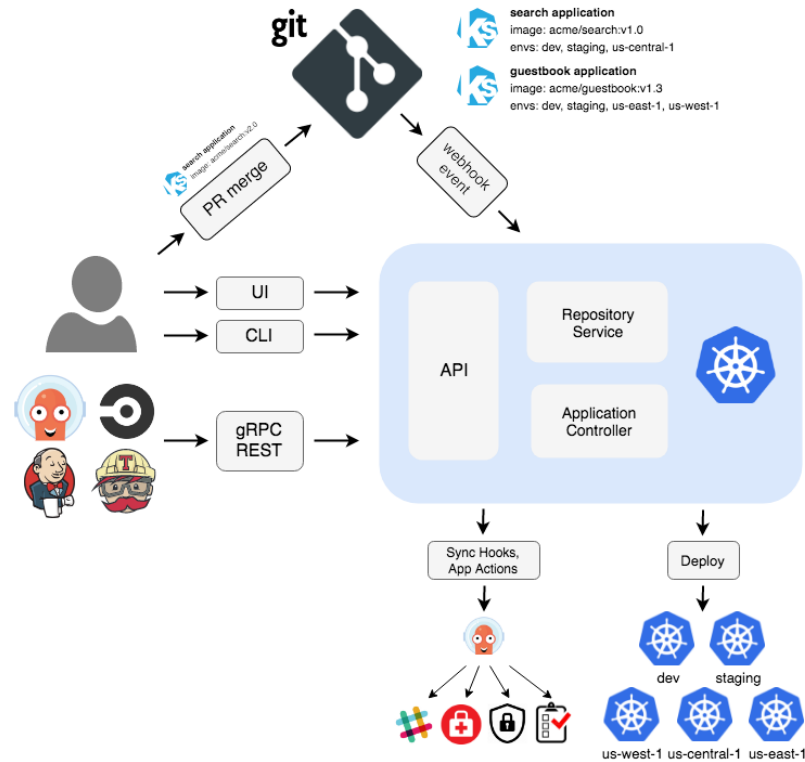


Рис. 2.2. Архітектура ArgoCD

Flux — це інструмент для Continuous Delivery та GitOps в середовищі Kubernetes. В основному, він використовується для автоматизації процесу оновлення та розгортання додатків в Kubernetes, використовуючи дані конфігурацій, які зберігаються у Git-репозиторії. Flux спостерігає за змінами в Git-репозиторії і автоматично синхронізує стан кластера Kubernetes з оновленнями, що знаходяться в репозиторії [10].

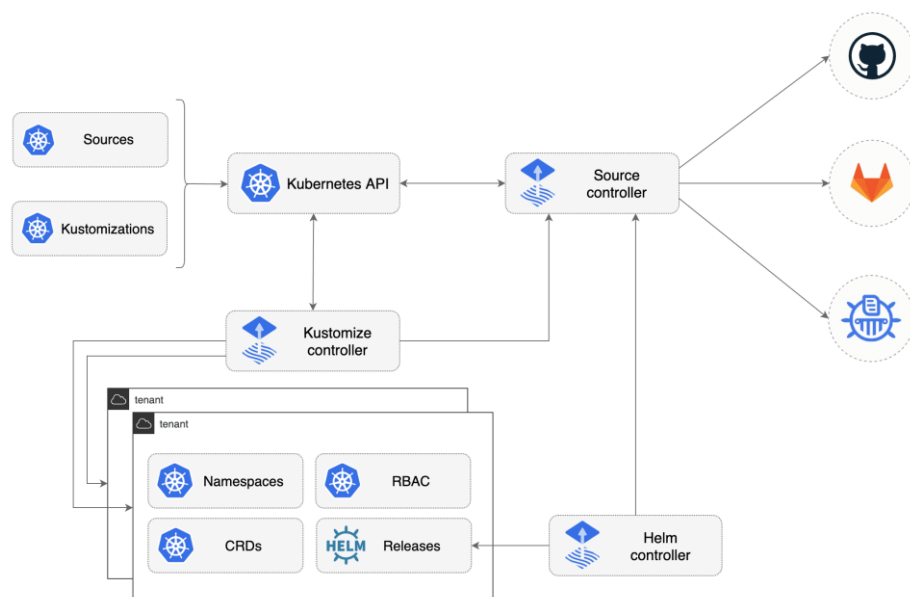


Рис. 2.3. Архітектура Flux

Flux і ArgoCD — це обидва інструменти для Continuous Delivery та GitOps в Kubernetes, але з деякими відмінностями у функціоналі та підході [10].

Однак, Flux і ArgoCD мають різні особливості. Flux більше спрямований на керування версіями, політикою оновлень та автоматичною синхронізацією. Він підтримує Helm-чарти та акцентується на простоті та гнучкості в роботі з конфігураціями через Git.

У той час як ArgoCD має розширений графічний інтерфейс та може надавати більше можливостей для спостереження за змінами та синхронізації стану додатків. ArgoCD також інтегрується з Helm та надає можливість керувати додатками через веб-інтерфейс.

Також важливим елементом розробки для системи, є порівняння цих рішень.

Таблиця 2.1

Порівняння функціоналу Flux та ArgoCD

Особливості / Функції	Flux	ArgoCD
Архітектура	Контролери Kubernetes та CRDs	Декларативний підхід, власний сервер
Функціонал	Керування версіями, політики оновлення, автосинхронізація, підтримка Helm	Графічний інтерфейс, підтримка Helm, спостереження та синхронізація стану додатків
Спосіб використання	Через CI/CD, спрощена робота з декларативною конфігурацією	Більш розвинена система спостереження та автоматичної синхронізації
Інтерфейс	Менше графічний, спрямований на роботу з конфігурацією через CRDs та Git	Розвинений графічний інтерфейс для керування додатками через веб

2.2 Середовище розробки

Коли порівнюємо рішення на різних мовах програмування з Python у випадку створення інструменту для розгортання в Kubernetes, варто враховувати деякі особливості:

– JavaScript/Node.js: JavaScript також може бути використаний для створення веб-серверів (за допомогою Node.js) та реалізації логіки додатків. Однак Python

частіше використовується у зв'язку зі своєю простотою, розширеною функціональністю та широкою підтримкою для різних бібліотек;

- Go: Go має перевагу в швидкодії та конкурентній обробці, але може бути складніше в освоєнні, особливо для розробників, які не мають досвіду в цій мові. Python, навпаки, зазвичай вважається більш легким для вивчення та розуміння;
- Ruby: Ruby також має свої фреймворки для веб-розробки, як Rails, які забезпечують продуктивність. Проте Python зазвичай має більшу кількість бібліотек для роботи з Kubernetes та використовується більш широко для DevOps-завдань;
- Java: Java має широкий спектр використання, але може вимагати більше коду для досягнення тих самих результатів, що й у Python. Python, навпаки, відомий своєю читабельністю та зручністю для розробки, що зазвичай приводить до більш швидкого розгортання рішення;

Отже, Python є більш привабливим у випадку створення інструментів для Kubernetes через свою простоту, зручність та широкий вибір бібліотек, спрямованих на роботу з цією технологією.

- простота використання: Python відомий своєю легкістю вивчення та читання, що робить його відмінним вибором для швидкої розробки прототипів та демонстраційних проектів [25];
- широкий спектр бібліотек: мова має багато готових бібліотек для взаємодії з API, роботи з Kubernetes (наприклад, `kubernetes-client`), роботи з HTTP-запитами, обробки JSON і інших завдань, пов'язаних з веб-розробкою;
- зручність для розгортання [25]: Python працює на багатьох платформах, його можна легко використовувати в конфігураціях контейнерів та у середовищі Kubernetes;
- екосистема і спільнота: мова має активну спільноту розробників, що призводить до швидкого виправлення помилок та підтримки актуальних бібліотек;

- можливості веб-розробки: Flask, який є мінімалістичним фреймворком для створення веб-додатків на Python, дозволяє легко реалізувати API та веб-сервери;
- легкість інтеграції: Python можна легко інтегрувати з різними інструментами та мовами програмування, що робить його універсальним вибором для проєктів з великою кількістю інтеграцій;

Узагальнюючи, Python є зручною мовою програмування для створення таких додатків завдяки своїй простоті, багатій екосистемі бібліотек та зручності в розгортанні [25].

2.3 Зовнішні компоненти розробки

Після аналізу розробленої архітектури (див.рис.2.1.) та архітектур наявних рішень, можливо перейти до огляду зовнішніх компонентів системи. Були використані такі зовнішні компоненти як GitHub, Kubernetes, Helm, Docker. Код містить наступні імпорти та використання бібліотек:

- git: ця бібліотека використовується для роботи з Git репозиторіями. Вона дозволяє клонувати репозиторії, здійснювати пул запити, визначати активну гілку та інші операції;
- os: ця бібліотека надає функціонал для взаємодії з операційною системою. У коді вона використовується для роботи з файловою системою;
- tempfile: вона використовується для створення тимчасових файлів та директорій. У коді використовується для створення тимчасової директорії;
- kubernetes: ця бібліотека використовується для взаємодії з Kubernetes. Конкретно, вона використовується для завантаження конфігурації Kubernetes з інтегрованого середовища та створення API-об'єкту для взаємодії з Kubernetes API;
- Flask: ця бібліотека використовується для створення web-застосунків з використанням Python. В коді вона створює об'єкт Flask для обробки HTTP запитів;

Вибір Flask для розробки застосунку може бути обґрунтований кількома причинами:

- простота використання: Flask — це легкий та простий у використанні веб-фреймворк для Python. Він має простий синтаксис, що дозволяє швидко створювати веб-додатки та API;
- гнучкість: Flask має модульну структуру, яка дозволяє розширювати його функціональність за допомогою додаткових бібліотек та розширень за потреби. Це дає можливість розробникам використовувати лише ті компоненти, які необхідні для їхнього проекту;
- документація та спільнота [25]: Flask має добре структуровану документацію та широку спільноту користувачів. Це дозволяє швидко знайти відповіді на питання чи рішення для проблем, що виникають під час розробки;
- модульність та розширюваність: Flask дозволяє використовувати тільки необхідні компоненти для конкретного проекту. Він не нав'язує жорстких правил стосовно структури проекту, що дозволяє розробникам використовувати свої підходи та архітектурні рішення;
- розробка API: Flask добре підходить для розробки API, оскільки дозволяє швидко створювати та підтримувати RESTful та інші типи веб-сервісів [5];

Отже, обрання Flask для розробки веб-додатку може бути зумовлене його легкістю використання, гнучкістю, наявністю добре структурованої документації та спільноти, а також можливістю модульності та розширюваності для відповіді на потреби в розробці.

Код сервера Flask створює маршрут /webhook, який очікує POST-запити. При отриманні POST-запиту від GitHub API з подією "push", він створює тимчасову директорію, отримує дані про репозиторій, виконує ряд дій із залежності від цих даних та повертає відповідь з результатом операцій. Для локального тестування використовувався Curl, утиліта за допомогою якої можливо надсилати запити на відповідні ендпоінти.

```
+ magister git:(main) x curl -X POST -H "X-GitHub-Event: push" http://192.168.49.2:32175/webhook
{"message":"Deployment triggered from push event"}
```

Рис. 2.4. Приклад запиту до менеджера

Для реалізації кластера використовується Kubernetes. Буде використане локальне середовище для тестування функціоналу ПЗ. У класичній реалізації Kubernetes це складно здійснити, але Minikube — інструмент, який надають можливість створення кластера, що можна використовувати на етапі розробки ПЗ. У традиційній реалізації Kubernetes, конфігурація кластера зазвичай включає окремо встановлену master-ноду та набір worker-нод. Master-нода відповідає за керування кластером, управління кластерними ресурсами та прийняття рішень про розподіл завдань між worker-нодами. В свою чергу, worker-ноди виконують навантаження, вони отримують і виконують завдання, які передаються з master-ноди.

Така архітектура забезпечує розділення обов'язків між вузлами та стабільність роботи кластера [6]. Однак для розробки на місці часто буває доцільно об'єднати master- та worker-ноди в одному середовищі для економії ресурсів та спрощення розробки.

Адаптуючи реалізації Minikube під виконання master-ноди та worker-ноди на одному хості, ми зменшуємо розмір виконуваного файлу та оптимізуємо використання ресурсів. Це дозволяє забезпечити зручність у розробці, використовуючи мінімальну кількість ресурсів. Реалізації Kubernetes-кластеру у вигляді Minikube має практично повну функціональність Kubernetes-кластера.

При роботі з Kubernetes важливо мати програмне забезпечення (ПЗ), яке забезпечує просте розгортання та управління кластером. Надійність системи та її стабільність є ключовими аспектами, які потрібно враховувати, включаючи можливість виявлення та виправлення помилок. Крім того, ефективне масштабування, точне керування ресурсами та можливість інтеграції з іншими інструментами розробки є не менш важливими аспектами для успішної роботи з Kubernetes. Якість документації та підтримка користувачів також відіграють важливу роль у використанні цієї технології.

Потрібно також визначити кілька вимог до програмного забезпечення (ПЗ), яке використовується для розробки та тестування на основі Kubernetes:

- легкість розгортання та використання: ПЗ повинне мати простий процес встановлення та запуску Minikube, а також легкий інтерфейс для управління та конфігурації Kubernetes-кластером на локальному середовищі;
- повна функціональність Kubernetes: ПЗ має забезпечувати практично повну функціональність стандартного Kubernetes-кластера, зокрема, здатність розгортати, керувати та масштабувати контейнеризовані додатки, а також взаємодіяти з їхніми сервісами;
- ефективне використання ресурсів: ПЗ має оптимізувати використання ресурсів комп'ютера шляхом об'єднання master- та worker-нод на одному хості, що дозволяє зменшити розмір виконуваного файлу та ефективніше використовувати ресурси;
- стабільність та надійність: ПЗ повинне гарантувати стабільну роботу Kubernetes-кластера, навіть при об'єднанні master та worker-нод на одному хості, та уникати можливих конфліктів управління та виконання завдань;
- сумісність та оновлення: важливо, щоб ПЗ було сумісним з останніми версіями Kubernetes і забезпечувало можливість оновлення для виправлення помилок та отримання нових функцій;
- підтримка та документація: Наявність докладної документації та підтримки з боку розробників є важливою для вирішення можливих проблем або невідомих питань під час використання ПЗ;

Ці вимоги допоможуть забезпечити ефективну розробку, тестування та управління Kubernetes-кластером в реальному хмарному середовищі та за допомогою Minikube у локальному середовищі для тестування роботи застосунку. Також можливо визначити вимоги для стратегічного бачення розробки цього ПЗ, а саме:

- інтеграція з іншими інструментами: якщо у буде потреба у взаємодії з іншими інструментами розробки, тестування чи систем управління версіями, потрібно зазначати вимоги щодо сумісності та можливостей інтеграції;
- автоматизація: вимоги до автоматизації процесів управління, розгортання та масштабування додатків у Kubernetes-кластері. Автоматичне виявлення та вирішення проблем також може бути корисним аспектом;
- масштабованість та продуктивність: потреба у забезпеченні ефективної роботи Kubernetes-кластера при збільшенні обсягу додатків чи завдань, а також можливості планування та управління ресурсами для оптимізації продуктивності;

Для побудови образу Docker для розробленої системи, використовувався сервіс GitHub Actions. Що в свою чергу надає багато переваг проти локальної збірки та пушу образу системи в реєстр образів [23].

2.4 Основні задачі системи

Визначення основних задач для програми є важливим, оскільки це надає чітку картину того, що потрібно досягти та які цілі передбачає розробка. Це спрощує управління проектом, допомагає команді розробників концентруватися на вирішенні основних проблем і завдань, а також уникнути відволікань на другорядні або непродуктивні аспекти. Визначення цих задач сприяє підвищенню продуктивності, зменшенню ризику помилок та забезпеченню ефективного розподілу завдань у команді для досягнення поставлених цілей розробки.

Основні задачі системи можна виділити наступним чином. Для обробки подій з репозиторія потрібен Webhook. Webhook у контексті GitHub — це механізм для надсилання сповіщень про події в реальному часі з GitHub до зовнішніх систем.

Ці сповіщення викликаються різними подіями, що відбуваються в репозиторії або організації, такими як виштовхування коду, створення проблем або pull-запитів, злиття гілок тощо. Веб-хуки дозволяють розробникам створювати інтеграції, які автоматично реагують на ці події.

```
@app.route("/webhook", methods=["POST"])
def handle_webhook():
    if "push" in request.headers.get("X-GitHub-Event", ""):
        try:
            repo_data = get_repo(temp_dir)

            if repo_data is not None:
                deploy = deploy_helm_chart(
                    chart_path=os.path.join(temp_dir, chart_path),
                    values_file=os.path.join(temp_dir, values_file),
                    release_name=release_name,
                )
                if deploy is not None:
                    create_configmap(repo_data)

            except Exception as e:
                logger.error(
                    f"Error while cloning the repository or deploying Helm chart: {e}"
                )

            return jsonify({"message": "Deployment triggered from push event"}), 200

        return jsonify({"message": "No action taken"}), 200
```

Рис. 2.5. Лістинг коду

Програма визначає маршрут /webhook(рис.2.5.), який очікує POST-запити від GitHub у відповідь на події репозиторію, зокрема події пушу в репозиторій.

Обробка події push в контексті GitHub є важливою, оскільки вона означає внесення змін до сховища. Коли відбувається подія push, це означає, що до сховища були перенесені нові коміти, які представляють зміни в коді. Для обробки події push до репозиторію система використовує таку логіку, що при отриманні POST-запиту на /webhook програма перевіряє(див.рис.2.5), чи подія в заголовку запиту відповідає події "push" в GitHub. Якщо так, вона продовжує обробку.

Для роботи з репозиторієм використовувалась така логіка, щоб запускати процес отримання даних з репозиторію, викликом функції get_repo(temp_dir), яка клонує репозиторій та повертає інформацію про нього. Ця функція отримує аргумент temp_dir, який представляє каталог, де буде зберігатися сховище. Вона

перевіряє, чи існує вказаний каталог (`temp_dir`). Якщо існує, вона намагається оновити сховище, витягуючи зміни з віддаленого джерела. Якщо каталог не існує, він клонує сховище за вказаною URL-адресою (`repo_url`) за допомогою модуля `git`. Якщо токен доступу доступний (`github_access_token`), він змінює URL-адресу сховища, щоб включити токен доступу для автентифікації. Після того, як сховище клоновано або оновлено, він отримує інформацію про сховище, таку як ім'я активної гілки (`branch`) та інші потенційні деталі сховища. Створюється словник, що містить інформацію про сховище, зокрема назву сховища, URL-адресу, активну гілку тощо. Якщо під час роботи виникають помилки (наприклад, при отриманні або оновленні сховища), програма реєструє помилки за допомогою наданого логера і повертає значення `None`.

Важливо звернути увагу, що код використовує бібліотеку `git Python` для взаємодії зі сховищем `Git`, виконуючи такі операції, як клонування або витягування змін на основі наданої директорії та URL-адреси сховища.

```

def deploy_helm_chart(chart_path, values_file, release_name):
    try:
        start_time = time.time() # Record start time
        helm_command = [
            "helm",
            "upgrade",
            "--install",
            release_name,
            chart_path,
            "-f",
            values_file,
            "--namespace",
            namespace,
        ]

        helm_result = subprocess.run(
            helm_command, check=True, capture_output=True, text=True
        )
        end_time = time.time() # Record end time
        deployment_time = end_time - start_time # Calculate deployment time
        logger.info(
            f"Helm chart deployed successfully in {deployment_time:.2f} seconds!"
        )
        logger.info(helm_result.stdout) # Log Helm command output
        return True

    except subprocess.CalledProcessError as e:
        logger.error(f"Error: Helm command failed with error: {e}")
        logger.error(e.output) # Log Helm command error output
        return None

```

Рис. 2.6. Лістинг коду

Якщо отримана інформація про репозиторій є валідною, програма спробує розгорнути Helm Chart з отриманих даних. Для цього викликається функція `deploy_helm_chart` (рис.2.6), яка отримує шляхи до файлів Helm Chart та файлу значень, необхідних для розгортання. Реалізація деплою додатку буде відбуватись за допомогою виконання команди оператора Helm, також такий підхід має переваги над іншими більш простими методами деплою, кожне оновлення або перша інсталяція додатку в кластер записується в ConfigMap самим оператором, що дає змогу в разі потреби зробити rollback на попередню версію додатку. Функція `deploy_helm_chart` починається із запису часу запуску для точного вимірювання тривалості розгортання. Потім вона створює команду Helm CLI, використовуючи вказані аргументи, такі як шлях до діаграми, файл значень, ім'я релізу та простір імен. Ця команда виконується як підпроцес за

допомогою `subprocess.run()`. У разі успішного виконання, вона записує час розгортання та результати виконання команди. У разі помилки, на яку вказує `subprocess.CalledProcessError`, вона записує деталі помилки і повертає `None`. У разі успішного розгортання повертає значення `True` як індикатор завершення. Загалом, ця функція обробляє розгортання додатку, відстежує час, записує відповідну інформацію та керує потенційними помилками під час процесу розгортання.

```
def setup_logging():
    logging.basicConfig(
        level=log_level,
        format="%(asctime)s - %(name)s - %(levelname)s - %(message)s",
    )

    return logging.getLogger(__name__)
```

Рис.2.7. Лістинг коду

Для логування результатів та можливих помилок системи було використано стандартну бібліотеку Python, а саме `logging`(рис.2.7). Цей модуль логування надає простий і зрозумілий API для базових потреб логування.

Процес налаштування простих конфігурацій журналювання простий, що робить його доступним для розробників. Оскільки це частина стандартної бібліотеки Python, не потрібно встановлювати сторонні бібліотеки для використання логування. Він легко доступний і може бути використаний у будь-якому додатку Python без додаткових залежностей. Функція `setup_logging` налаштовує систему логування. Вона встановлює рівень логування і визначає формат повідомлень журналу. Рівень логування визначається змінною `log_level` з іншого місця коду. Формат повідомлення журналу включає такі дані, як мітка часу, ім'я журналу, рівень журналу і власне повідомлення журналу. Нарешті, функція отримує об'єкт журналу, специфічний для поточного модуля, за допомогою функції `logging.getLogger(__name__)`, сконфігурований відповідно до налаштувань, визначених раніше, і вмикає ведення журналу у модулі на основі цієї конфігурації.

Якщо розгортання Helm Chart пройшло успішно, програма створює ConfigMap на основі отриманих даних з репозиторію за допомогою функції `create_configmap(repo_data)`, створення ConfigMap потрібно щоб зберігати в ньому дані про репозиторій, такий підхід дозволяє використовувати нативні ресурси Kubernetes без використання баз даних або брокерів повідомлень. Функція `create_configmap` керує ConfigMap Kubernetes за допомогою клієнта Kubernetes Python. Вона починається із завантаження конфігурації Kubernetes для операцій у кластері. Вона створює клієнт Kubernetes API для зв'язку з кластером Kubernetes. Ця функція визначає і налаштовує об'єкт ConfigMap на основі наданих даних. Потім вона перевіряє, чи існує об'єкт ConfigMap з вказаним ім'ям у заданому просторі імен. Якщо ConfigMap існує, вона намагається оновити його новими даними конфігурації. У разі успішного оновлення, програма записує повідомлення, що підтверджує оновлення. Якщо ConfigMap не існує, вона створює новий ConfigMap з наданими даними і записує повідомлення про успішне створення ConfigMap. Функція обробляє створення або оновлення ConfigMap в кластері Kubernetes і реєструє відповідні повідомлення на основі результатів цих операцій. Якщо під час процесу виникають будь-які винятки, вона реєструє відповідне повідомлення про помилку.

```
if __name__ == "__main__":
    app.run(debug=True)
```

Рис. 2.8. Лістинг коду

Основний код цих операцій знаходиться в методі `handle_webhook()`, який відповідає за обробку POST-запитів, отриманих від GitHub. При виклику програми (`__name__ == "__main__"`)(рис.2.8), вона запускає веб-сервер Flask для обробки цих запитів.

Отже, основні завдання програми включають обробку подій GitHub або будь якої системи контролю версій, отримання даних з репозиторію, розгортання

Helm Chart та створення ConfigMap на основі цих даних у відповідь на подію пушу до репозиторію.

Ось загальний функціонал системи:

- Git-операції:
 - включає в себе функціонал для взаємодії з Git-репозиторіями за допомогою бібліотеки git Python;
 - код включає функції для клонування сховищ, їх оновлення та отримання інформації про сховище, наприклад, активні гілки;
- Helm Chart Deployment:
 - функція, яка призначена для розгортання додатків за допомогою Helm на кластері Kubernetes за допомогою Helm CLI через модуль підпроцесів Python. Вона фіксує час розгортання, реєструє статус розгортання та обробляє потенційні помилки під час розгортання;
- Kubernetes Operations:
 - код взаємодіє з кластером Kubernetes через клієнт Kubernetes Python для виконання таких операцій, як створення або оновлення ConfigMaps. Він налаштовує клієнт Kubernetes і керує ConfigMaps, перевіряючи їх існування і створюючи нові або оновлюючи існуючі з новими конфігураціями;
- налаштування ведення журналу:
 - код ініціалізує конфігурації логуювання за допомогою модуля логуювання, щоб полегшити ведення логуювання у всій системі. Він встановлює рівні логів, форматує повідомлення логуювання і налаштовує логери для збору і запису інформації про різні операції;

2.5 Висновок до розділу 2

Можна зробити висновок, що розробка системи для деплою в Kubernetes кластер включає у себе кілька компонентів та процесів для ефективного управління розгортанням програмного забезпечення. Зокрема, будівництво

pipeline для автоматизації процесу — від збирання коду до створення Docker-образів і генерації маніфестів Kubernetes — стає важливим аспектом цієї архітектури. Інтеграція з Kubernetes API відкриває можливості для взаємодії з ресурсами для управління та розгортання артефактів.

Порівнюючи мови програмування у розробці інструментів для Kubernetes, Python виділяється своєю простотою, розширеною функціональністю та широкою підтримкою для різних бібліотек, спрямованих на роботу з цією технологією. Порівняно з іншими мовами, такими як JavaScript, Node.js, Go, Ruby чи Java, Python виявляється більш привабливим для розробки інструментів для Kubernetes.

Процес вибору мови програмування на основі їхніх переваг та особливостей є ключовим для забезпечення ефективного розгортання та розвитку системи, спрямованої на роботу з Kubernetes.

Далі було розглянуто зовнішні компоненти розробки, такі як GitHub, Kubernetes, Helm та Docker. Аналіз використання цих зовнішніх компонентів дозволяє краще розуміти структуру та інтеграцію системи у контексті роботи з Kubernetes.

Далі звертається увага на основні задачі системи, які визначають її функціональність та область застосування. Чітке визначення цих завдань сприяє ефективному управлінню проектом.

Загальний висновок полягає в тому, що аналіз архітектури, мов програмування, зовнішніх компонентів та задач системи в контексті їхнього впливу на розробку системи для Kubernetes допомагає зрозуміти ключові аспекти, які необхідно враховувати для ефективного створення та управління програмним забезпеченням у таких середовищах.

РОЗДІЛ 3 ДОСЛІДЖЕННЯ СИСТЕМИ ТА ЕФЕКТИВНОСТІ РІШЕННЯ

Цей розділ описує дослідження системи та ефективність прийнятих рішень.

В цьому розділі важливо провести детальний аналіз системи, а саме: вивчення структури, компонентів та взаємодії між ними, що допомагає краще зрозуміти, як працює система в цілому, та чому саме це рішення є кращим за наявних аналогів.

Не менш важливим елементом є оцінка ефективності цього рішення. Аналізуючи рішення, прийняті для створення такої системи та вдосконалення загальної архітектури таких додатків, можна визначити ефективність. Це оцінюється через різні метрики, які дозволяють визначити, наскільки успішно впроваджені рішення вирішують визначені проблеми та чи досягають поставлених цілей.

Порівняння системи на основі досліджень, що надає змогу запропоновувати рекомендації для поліпшення системи. Це може включати нові підходи, технології або стратегії, спрямовані на оптимізацію функціонування системи та підвищення її ефективності.

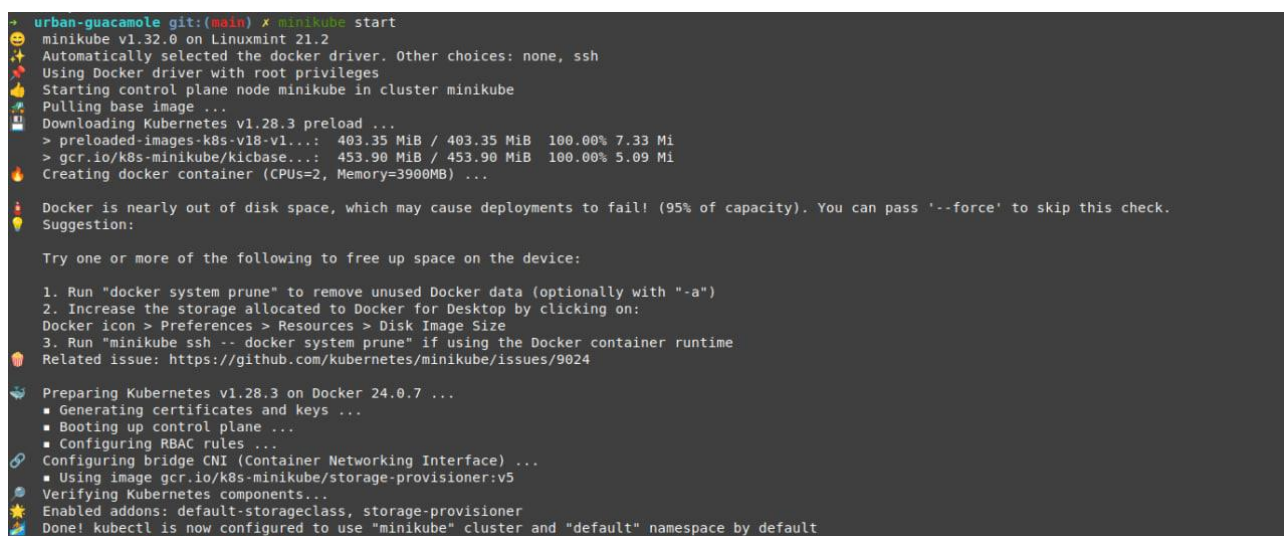
Важливим елементом розділу також є обґрунтування створеного рішення на основі проведеного дослідження. Це створює основу для переконливого пояснення науковості роботи.

3.1 Створення середовища

Для тестування створеної системи, було обрано локальне середовище Minikube, такий підхід має декілька переваг. Minikube надає можливість створення локального одноразового кластера Kubernetes, що дозволяє ізолювати тестове середовище від продуктивного середовища. Це забезпечує безпеку й уникнення впливу тестових експериментів на реальну систему. Це робить процес розробки більш ефективним, оскільки ви можете швидко перевіряти нові функції або зміни в коді без затрат на довгочасне розгортання в реальному хмарному

середовищі. Також таке середовище дозволяє легко вносити зміни у конфігурацію Kubernetes, тестувати різні параметри та параметризувати додатки. Це дає можливість експериментувати з різними налаштуваннями без впливу на продуктивну систему.

Для створення такого середовища було встановлено Minikube на персональний комп'ютер. За допомогою команди старту кластеру було запущено тестову систему(рис.3.1).



```

urban-guacamole git:(main) x minikube start
minikube v1.32.0 on Linuxmint 21.2
Automatically selected the docker driver. Other choices: none, ssh
Using Docker driver with root privileges
Starting control plane node minikube in cluster minikube
Pulling base image ...
Downloading Kubernetes v1.28.3 preload ...
> preloaded-images-k8s-v18-v1...: 403.35 MiB / 403.35 MiB 100.00% 7.33 Mi
> gcr.io/k8s-minikube/kicbase...: 453.90 MiB / 453.90 MiB 100.00% 5.09 Mi
Creating docker container (CPUs=2, Memory=3900MB) ...

Docker is nearly out of disk space, which may cause deployments to fail! (95% of capacity). You can pass '--force' to skip this check.
Suggestion:

Try one or more of the following to free up space on the device:

1. Run "docker system prune" to remove unused Docker data (optionally with "-a")
2. Increase the storage allocated to Docker for Desktop by clicking on:
   Docker icon > Preferences > Resources > Disk Image Size
3. Run "minikube ssh -- docker system prune" if using the Docker container runtime
Related issue: https://github.com/kubernetes/minikube/issues/9024

Preparing Kubernetes v1.28.3 on Docker 24.0.7 ...
  Generating certificates and keys ...
  Booting up control plane ...
  Configuring RBAC rules ...
Configuring bridge CNI (Container Networking Interface) ...
  Using image gcr.io/k8s-minikube/storage-provisioner:v5
Verifying Kubernetes components...
Enabled addons: default-storageclass, storage-provisioner
Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
  
```

Рис. 3.1. Створення тестового середовища Minikube

Також для полегшення фіксування тестів ефективності розробленого рішення було використано програму Lens. За допомогою Lens можливо відстежувати стан ресурсів та компонентів Kubernetes, а також шукати й виправляти можливі проблеми, що допомагає вдосконалювати ефективність розробленого рішення. Lens надає зручний інтерфейс для візуалізації та управління кластером Kubernetes. Можна швидко отримати зображення стану свого кластера, його компонентів та ресурсів, що дозволяє краще розуміти архітектуру та здійснювати відлагодження.

3.2 Використання конкурентних рішень

В першому розділі цієї роботи зазначались 2 наявних рішення такої системи, тому важливим аспектами цього параграфу є встановлення конкурентних рішень на тестове середовище, та проведення аналізу

ефективності кожного конкурентного рішення в контексті поставлених цілей чи вимог. Це може включати порівняння показників ефективності, таких як продуктивність, швидкість, масштабованість, чи інші метрики. Порівняння різних рішень щодо їхніх переваг та недоліків. Це дозволяє визначити сильні та слабкі сторони кожного підходу, архітектури, коду систем та їхній вплив на досягнення поставлених цілей.

Для тестування конкурентних рішень було розроблено простий Python додаток який використовує Flask для веб частини, за допомогою одного додатку можливо протестувати ефективність наявних рішень та виділити переваги та недоліки цих систем. Для створення однакових умов тестування було створено образ Docker веб додатку, і для тестування функцій синхронізації з репозиторієм було створено конфігураційні файли, а саме: `deployment.yaml`(рис.3.1), та `service.yaml`(рис.3.2). В цих файлах було визначено прості з точки зору архітектури додатку ресурси.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: web-app-flux
5    namespace: staging
6    labels:
7      "service": "web-app-flux"
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       app: "web-app-flux"
13   template:
14     metadata:
15       labels:
16         app: "web-app-flux"
17     spec:
18       containers:
19         - name: "web-app-flux"
20           image: "vmatsapura/web-app:latest"
21           ports:
22             - containerPort: 5000
```

Рис. 3.1. Конфігураційний файл `deployment.yaml`

```

1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: "web-app-flux"
5    namespace: "staging"
6  spec:
7    type: ClusterIP
8    ports:
9      - port: 8081
10      targetPort: 5000
11    selector:
12      app: "web-app-flux"

```

Рис. 3.2. Конфігураційний файл service.yaml

Система Flux [10] складається з інструменту командного рядка (Flux CLI) та серії контролерів Kubernetes.

Щоб встановити FluxCD, спочатку потрібно завантажити flux CLI. Потім, використовуючи CLI, можливо розгорнути контролери Flux на своїх кластерах і налаштувати свій перший конвеєр доставки GitOps.

Спочатку потрібно встановити FluxCD на персональний комп'ютер і далі для того щоб встановити FluxCD на тестове середовище було використано таку команду

flux install

Ця команда встановлює компоненти для стабільної роботи FluxCD(рис.3.3).

Pods										
6 items			Namespace: flux-system		Search Pods...					
☐ Name	Namespace	Contai...	CPU	Memory	Restarts	Controlled By	Node	QoS	Age	Status
☐ helm-controller-5f964c6579-snq56	flux-system	■	0.001	14.2MiB	0	ReplicaSet	minikube	Burstable	15m	Running
☐ image-automation-controller-b7c5b48f6-hmf7l	flux-system	■	0.001	11.1MiB	0	ReplicaSet	minikube	Burstable	18m	Running
☐ image-reflector-controller-6658b59bc4-gp5mn	flux-system	■	0.014	30.3MiB	0	ReplicaSet	minikube	Burstable	18m	Running
☐ kustomize-controller-9c588946c-ts5pb	flux-system	■	0.001	40.5MiB	0	ReplicaSet	minikube	Burstable	15m	Running
☐ notification-controller-76dc5d768-v7dgr	flux-system	■	0.001	12.0MiB	0	ReplicaSet	minikube	Burstable	15m	Running
☐ source-controller-6c49485888-k7hjp	flux-system	■	0.002	24.8MiB	0	ReplicaSet	minikube	Burstable	12m	Running

Рис. 3.3. Компоненти системи FluxCD

Далі потрібно синхронізувати репозиторій з компонентами системи FluxCD

flux bootstrap github --token-auth --owner=vladelleus --repository=urban-guacamole --branch=main --path=clusters/minikube-flux/ --personal

Як результат FluxCD синхронізується з персональним репозиторієм за допомогою GitHub Private Access Token, який також використовується в розробленій системі. Тобто підходи до надання доступу до репозиторію з кодом є однаковими.

```

➔ ~ flux bootstrap github \
  --token-auth \
  --owner=vladelleus \
  --repository=urban-guacamole \
  --branch=main \
  --path=clusters/minikube-flux \
  --personal
Please enter your GitHub personal access token (PAT):
✓ connecting to github.com
✓ cloning branch "main" from Git repository "https://github.com/vladelleus/urban-guacamole.git"
✓ cloned repository
✓ generating component manifests
✓ generated component manifests
✓ component manifests are up to date
✓ reconciled components
✓ determining if source secret "flux-system/flux-system" exists
✓ generating source secret
✓ applying source secret "flux-system/flux-system"
✓ reconciled source secret
✓ generating sync manifests
✓ generated sync manifests
✓ sync manifests are up to date
✓ applying sync manifests
✓ reconciled sync configuration
⊗ waiting for Kustomization "flux-system/flux-system" to be reconciled
✓ Kustomization reconciled successfully
✓ confirming components are healthy
✓ helm-controller: deployment ready
✓ kustomize-controller: deployment ready
✓ notification-controller: deployment ready
✓ source-controller: deployment ready
✓ all components are healthy

```

Рис. 3.4. Результат виконання команди

Як результат успішного виконання команди компонент системи Kustomize намагається синхронізувати наявні yaml файли в репозиторії з Kubernetes кластером.

Для перевірки ефективності часу синхронізації можна перевірити логи цього компоненту

```

{"level":"info","ts":"2023-12-10T17:51:38.657Z","msg":"Reconciliation
finished          in          627.806339ms,          next          run          in
10m0s","controller":"kustomization","controllerGroup":"kustomize.toolkit.fluxcd.io",
"controllerKind":"Kustomization","Kustomization":{"name":"flux-
system","namespace":"flux-system"},"namespace":"flux-system","name":"flux-
system","reconcileID":"83a85814-715e-4e4e-888c-

```

`c30f883d0a82", "revision": "main@sha1:bbba57600e4c5fd966ef659ca723ef9a29b4b9b3"} }`

З отриманих логів компоненту бачимо, що деплой додатку відбувся за 0,628 секунди.

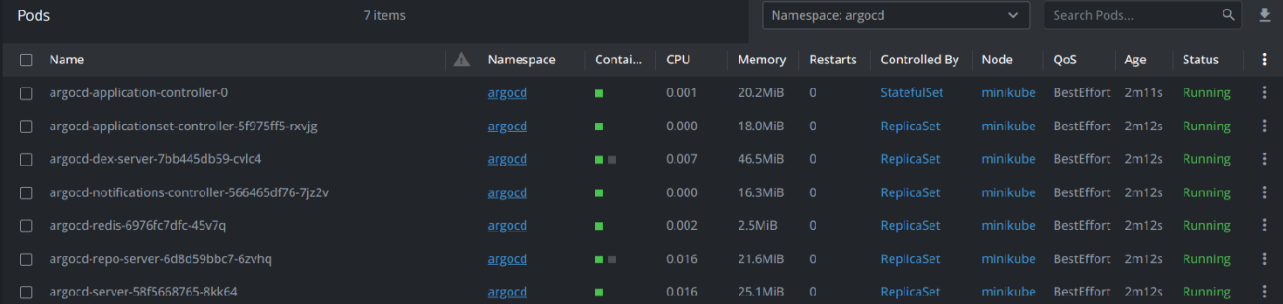
Для тестування ефективності системи ArgoCD потрібно також зробити певні команди для встановлення компонентів цієї системи в тестове середовище.

Спочатку потрібно створити namespace для цієї системи

```
kubectl create namespace argocd
```

Далі використовуємо конфігураційний файл для встановлення компонентів

```
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```



Name	Namespace	Contai...	CPU	Memory	Restarts	Controlled By	Node	QoS	Age	Status
argocd-application-controller-0	argocd	■	0.001	20.2MiB	0	StatefulSet	minikube	BestEffort	2m11s	Running
argocd-applicationset-controller-5f975ff5-xxvjg	argocd	■	0.000	18.0MiB	0	ReplicaSet	minikube	BestEffort	2m12s	Running
argocd-dex-server-7bb445db59-cvkc4	argocd	■ ■	0.007	46.5MiB	0	ReplicaSet	minikube	BestEffort	2m12s	Running
argocd-notifications-controller-566465df76-7jz2v	argocd	■	0.000	16.3MiB	0	ReplicaSet	minikube	BestEffort	2m12s	Running
argocd-redis-6976fc7dfc-45v7q	argocd	■	0.002	2.5MiB	0	ReplicaSet	minikube	BestEffort	2m12s	Running
argocd-repo-server-6d8d59bbc7-6zvnhq	argocd	■ ■	0.016	21.6MiB	0	ReplicaSet	minikube	BestEffort	2m12s	Running
argocd-server-58f5668765-8kk64	argocd	■	0.016	25.1MiB	0	ReplicaSet	minikube	BestEffort	2m12s	Running

Рис. 3.5. Встановлені компоненти системи ArgoCD

ArgoCD [10] має зручний інтерфейс та гарну веб частину системи(рис.3.6). Використання веб частини в GitOps застосунку немає серйозної вагомості, адже важливим аспектом таких систем є час за який відбувається деплой додатку.

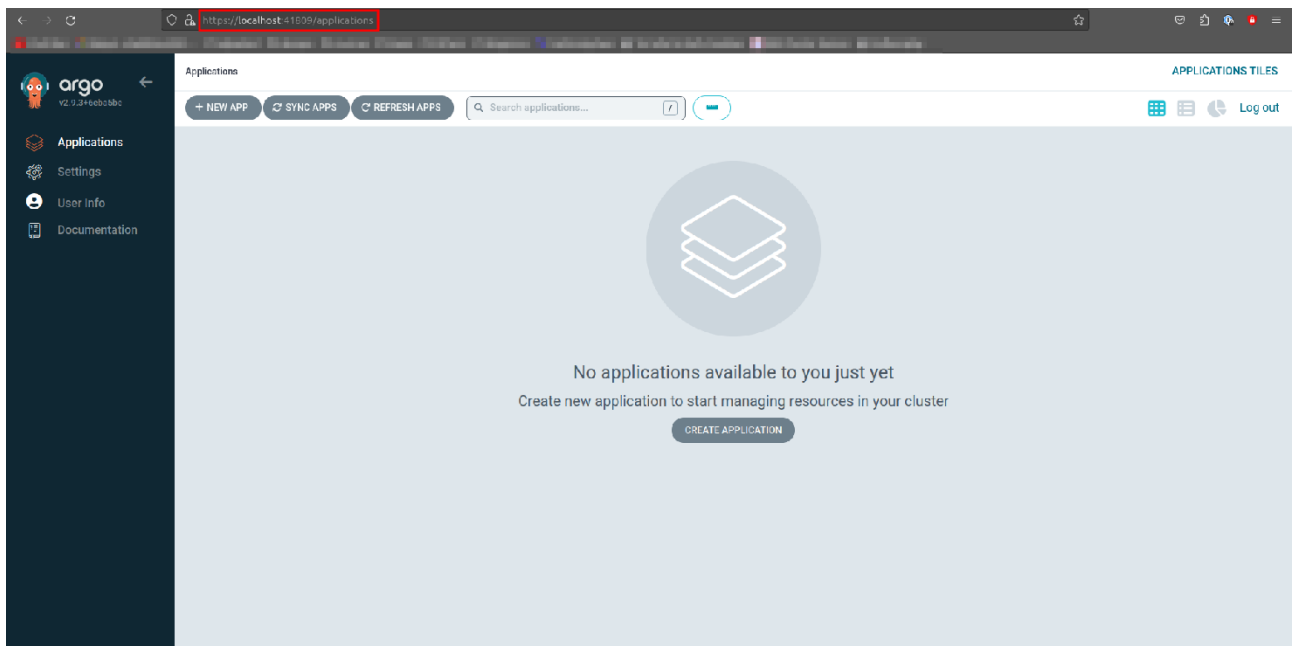


Рис. 3.6. Веб частина системи ArgoCD

Було додано репозиторій зі шляхом до попередньо створеної директорії де були конфігураційні файли уaml

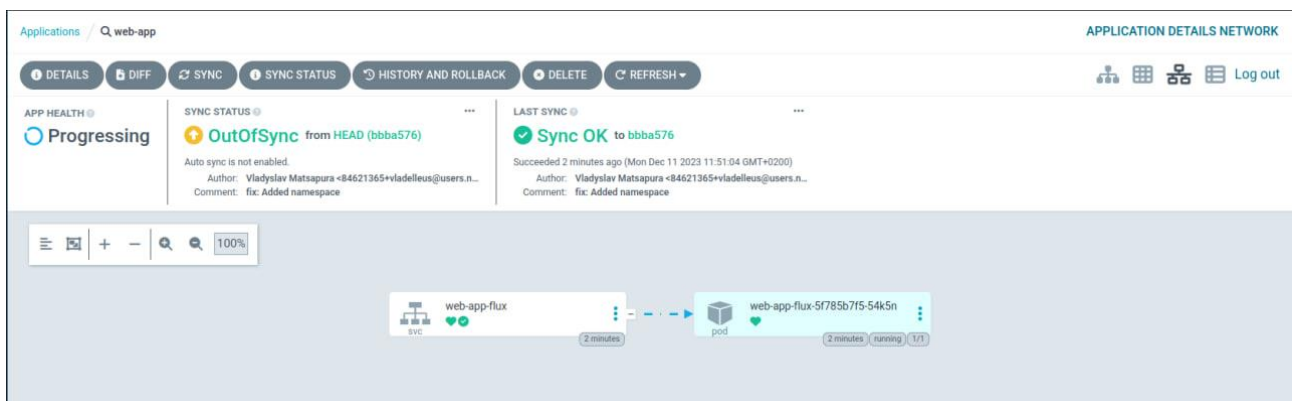


Рис. 3.7. Синхронізація застосунку в ArgoCD

З отриманих результатів синхронізації отримуємо такі логи в одному з компонентів системи:

```
time="2023-12-11T09:51:07Z" level=info msg="manifest cache hit:
&ApplicationSource{RepoURL:https://github.com/vladdelleus/urban-
guacamole,Path:clusters/minikube-
flux/resources,TargetRevision:HEAD,Helm:nil,Kustomize:nil,Directory:nil,Plugin:ni
l,Chart:,,Ref:,}/bbba57600e4c5fd966ef659ca723ef9a29b4b9b3"
```

```
time="2023-12-11T09:51:07Z" level=info msg="finished unary call with code
OK" grpc.code=OK grpc.method=GenerateManifest
```

grpc.service=repository.RepoServerService *grpc.start_time="2023-12-11T09:51:08Z"* *grpc.time_ms=0.738* *span.kind=server* *system=grpc*

Далі потрібно протестувати результати розробленої системи. Потрібно спочатку додати систему в тестове середовище.

Name	Namespace	Container	CPU	Memory	Restarts	Controlled By	Node	QoS	Age	Status
manager-app-75c4455c66-cgmzp	staging		0.000	75.1MiB	2	ReplicaSet	minikube	BestEffort	15h	Running
web-app-64678fd8bf-7ct69	staging		0.004	38.2MiB	2	ReplicaSet	minikube	BestEffort	16h	Running

Рис. 3.8. Створення розробленої системи в Kubernetes кластері

Після цього система за допомогою /webhook ендпоїнту отримує запити від GitHub, що в свою чергу запускає процес деплою(рис.3.9).

```
2023-12-10 16:09:15,355 - app.logging - INFO - Helm chart deployed successfully in 0.35 seconds!
2023-12-10 16:09:15,355 - app.logging - INFO - Release "web-app" has been upgraded. Happy Helming!
```

Рис. 3.9. Логи системи про успішний деплой додатку

3.3 Аналіз отриманих результатів

Для порівняння використання ресурсів було зібрано метрики на основі використаних ресурсів серверу де працювали тестові системи, для аналізу було обрано метрики завантаженості CPU, та Memory.

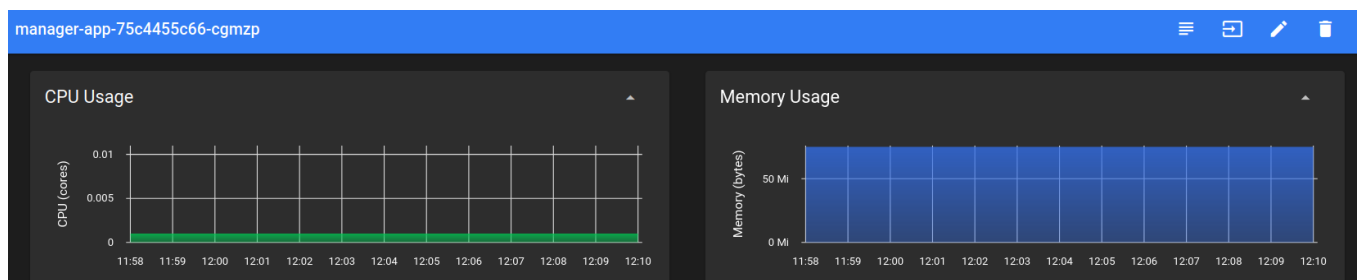


Рис. 3.10. Метрики розробленої системи

З графіків можна побачити, що система не використовувала більше ніж 0,0001 ядра центрального процесору, та не більше 75,1 байтів оперативної пам'яті(рис.3.10).

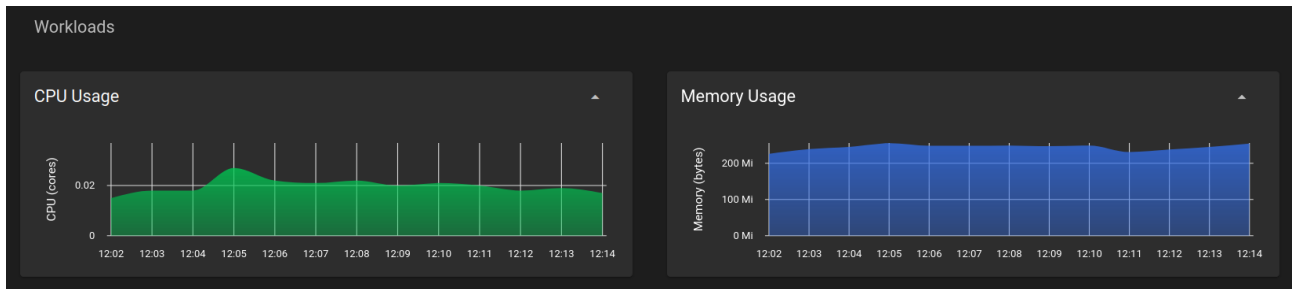


Рис. 3.11. Використання ресурсів системою ArgoCD

Система ArgoCD, яка включає в себе 6 компонентів, які в середньому використовували 0,019 ядра центрального процесору, та 250 байтів оперативної пам'яті(рис.3.11).

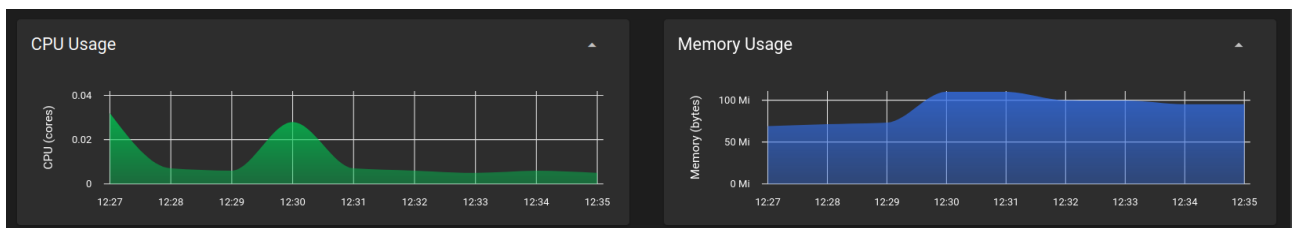


Рис. 3.12. Використання ресурсів системою FluxCD

Система FluxCD, яка включає в себе 4 основних компоненти, які в середньому використовували 0,0014 ядра центрального процесору, та майже 100 байтів оперативної пам'яті(рис.3.12).

Таблиця 3.1

Використання фізичних ресурсів

Система	Кількість компонентів в системі	Використання CPU (середнє, ядро)	Використання ОЗП (середнє, байт)
Розроблена система	1	0.0001	75.1
ArgoCD	6	0.019	250
FluxCD	4	0.001	100

За даними, представленими у таблиці, які були отримані після аналізу графіків. Можна зробити кілька висновків про розроблену систему порівняно з системами ArgoCD та FluxCD:

— кількість компонентів: розроблена система складається з одного компонента, що є значно меншим за кількістю компонентів у системах ArgoCD (6

компонентів) та FluxCD (4 компоненти). Це може вказувати на меншу складність або простоту розробленої системи порівняно з іншими;

- використання ресурсів: розроблена система має дуже низьке використання CPU (середньо 0.0001 ядра) і помірне використання оперативної пам'яті (середньо 75.1 байтів). У порівнянні з ArgoCD та FluxCD, розроблена система використовує менше ресурсів;
- ефективність використання ресурсів: враховуючи кількість компонентів, розроблена система демонструє ефективне використання ресурсів, оскільки здебільшого вона використовує незначні обсяги CPU та ОЗП для своєї роботи;
- порівняно з іншими системами: у порівнянні з ArgoCD, що складається також з 6 компонентів, розроблена система використовує значно менше ресурсів. Щодо FluxCD з 4 компонентами, розроблена система використовує менше ОЗП, але трохи більше CPU;

Отже, зазначені дані свідчать про те, що розроблена система має низьку складність, ефективно використовує ресурси, зокрема, має низьке споживання CPU, що може вказувати на її легкість та ефективність у використанні ресурсів сервера.

Таблиця 3.2

Порівняння швидкодії систем

Система	Кількість компонентів	Час деплою (середній, сек)
Розроблена система	1	0.30-0.35
ArgoCD	6	0.738
FluxCD	4	0.628

Розроблена система, яка має лише один компонент, демонструє значно швидші показники деплою порівняно з ArgoCD та FluxCD. При цьому, навіть у порівнянні з системою FluxCD, яка має менше компонентів, розроблена система виявилася швидшою у процесі деплою. Таким чином, в контексті часу деплою, розроблена система виявляється більш ефективною та швидше виконує процес розгортання порівняно з ArgoCD та FluxCD.

Також важливо буде зазначити, чому розроблена система працює швидше за аналоги.

Розроблена система, що демонструє вищу швидкість деплою порівняно з аналогами ArgoCD та FluxCD, може бути більш ефективною з кількох причин:

- використання ConfigMap замість баз даних: в розробленій системі для зберігання конфігурацій та параметрів використовується механізм Kubernetes під назвою ConfigMap. У порівнянні з базами даних, які використовуються в ArgoCD та FluxCD, ConfigMap є більш легким і простим інструментом для зберігання конфігураційних даних. Це дозволяє системі швидше виконувати процес завантаження та оновлення конфігурацій;
- простіша архітектура: розроблена система має меншу кількість компонентів та менш складну архітектуру порівняно з ArgoCD та FluxCD, які мають більшу кількість компонентів. Наприклад, в ArgoCD велику кількість ресурсів системи забирає веб частина, яка для механіки GitOps, не має великого значення. Простіша архітектура може означати менше число процесів, які потрібно виконати під час деплою, та менше обчислювальних операцій для виконання завдання;

Процес деплою розробленої системи, з використанням ConfigMap та простішою архітектурою, може включати такі кроки:

- збір конфігураційних файлів та параметрів з ConfigMap;
- завантаження змінених конфігурацій на Kubernetes кластер;
- перезапуск компонентів системи для застосування оновлених конфігурацій;
- перевірка та валідація оновлених компонентів;

Ці кроки у розробленій системі можуть виконуватися швидше через меншу кількість операцій та обчислювальних завдань, а також через використання більш легковагомих інструментів для зберігання конфігурацій. Такий підхід дозволяє системі працювати ефективніше та швидше в рамках процесу деплою.

3.4 Порівняння функціоналу систем

Аналіз функціоналу допомагає виявити переваги та недоліки кожної системи. Це дозволяє уникнути можливих проблем у майбутньому, обираючи систему, яка найкраще відповідає конкретним вимогам та потребам проекту.

Таблиця 3.3

Порівняння функціоналу систем

Характеристика	Розроблена система	FluxCD	ArgoCD
Робота з простими YAML-файлами	+	+	+
Робота з Kustomize	+	+	+
Використання Helm Release	+	+	+
Необхідність встановлення в Kubernetes-кластер	+	+	+
Використання баз даних	-	+	+
Використання нативних Kubernetes ресурсів для збереження стану конфігурацій додатку	+	-	-
Робота з Git	+	+	+
Робота з багатьма провайдерами VCS систем	-	+	+
Зручний UI інтерфейс	-	-	+

На основі порівняльного аналізу характеристик(табл.3.3.) між розробленою системою та конкурентами FluxCD та ArgoCD, можна зробити такий загальний висновок щодо розробленої системи.

Розроблена система демонструє відмінності та спільні характеристики у порівнянні з конкурентами FluxCD та ArgoCD. Вона має значну кількість спільних функціональних можливостей з цими системами, таких як робота з YAML-файлами, Kustomize, Helm Release, необхідність встановлення в Kubernetes-кластер, що свідчить про високий рівень функціональності та спрямованість на роботу з контейнеризованими додатками.

Однак, вона має відмінності у деяких аспектах, таких як відсутність використання баз даних та використання нативних Kubernetes ресурсів для збереження стану конфігурацій додатків, що може відрізняти її від конкурентів.

3.5 Висновок до розділу 3

Отже, можна дійти таких висновків, що для тестування та аналізу системи використовувався Minikube, щоб забезпечити ефективність та безпеку тестових експериментів, не впливаючи на реальну систему. Використання Minikube дозволило швидко перевіряти нові функції без довгочасного розгортання в реальному хмарному середовищі. Таке середовище також дозволяє легко вносити зміни у конфігурації Kubernetes та параметризувати додатки.

Далі, було досліджено використання конкурентних рішень та проведення аналізу їхньої ефективності в контексті поставлених цілей. Для цього було розроблено Python-додаток, що використовує Flask для веб-частини, та створено образ Docker веб-додатку для тестування. Аналіз показав, що розроблена система використовує менше ресурсів CPU та ОЗП порівняно з ArgoCD та FluxCD. Крім того, розроблена система складається з одного компонента, що може вказувати на її меншу складність порівняно з іншими системами, що мають більшу кількість компонентів.

Аналіз отриманих результатів вказує на те, що розроблена система ефективно використовує ресурси, що є важливим фактором у контексті розгортання та управління програмним забезпеченням. Порівнюючи її з конкурентними рішеннями ArgoCD та FluxCD, вона демонструє менше

споживання ресурсів, в той час як досить ефективно виконує свої завдання. Це робить її привабливою альтернативою для розгортання у відповідних середовищах, де ефективне використання ресурсів має значення. Розроблена система має певні відмінності та спільні характеристики порівняно з конкурентами FluxCD та ArgoCD. Вона має велику кількість спільних функціональних можливостей з цими системами, таких як робота з YAML-файлами, Kustomize, Helm Release, та необхідність встановлення в Kubernetes-кластер, що свідчить про високий рівень функціональності та орієнтацію на роботу з контейнеризованими додатками. Проте вона відрізняється в деяких аспектах. Наприклад, розроблена система не використовує бази даних та не використовує нативні Kubernetes ресурси для збереження стану конфігурацій додатків, що може виділити її від конкурентів.

Отже, розроблена система демонструє конкурентоспроможність та широкі можливості управління контейнеризованими додатками. Її спільні характеристики з іншими системами у цьому розділі підкреслюють її ефективність та гнучкість в контексті розгортання та керування програмним забезпеченням. Однак, присутність певних відмінностей від конкурентів вказує на унікальність та можливість вибору даної системи залежно від конкретних потреб та обмежень проекту.

ВИСНОВКИ

У кваліфікаційній магістерській роботі було вирішене завдання по розробці системи для застосування методів CI/CD та розгортання додатку в робочому середовищі. Обґрунтування актуальності теми дало змогу оцінити цінність такої роботи та яка є її актуальність для розробників додатків. Також було розв'язано завдання які передбачались з досягненням поставленої мети, адже в ході роботи було виконанно поставлені завдання.

Було проведено дослідження наявних рішень, які виконують процеси CD/CD в робочі середовища.

Проводилось дослідження, які існують робочі середовища для високонавантажених систем.

Також було розроблено архітектуру для майбутньої системи, та зроблено аналіз архітектур наявних систем.

Також був проведений аналіз конкурентних систем, які вже виконують процеси CI/CD в Kubernetes кластер. Було проаналізовано ці системи, та доведено за допомогою метрик, чому саме розроблене рішення є кращим за наявні системи.

Наступним етапом було оцінювання якості системи за допомогою метрик: використання ресурсів процесору, використання ресурсів пам'яті, швидкодія. Аналіз даної розробки показав кращі результати, ніж наявні рішення.

І як результат було розроблено систему для безперервної інтеграції та доставки додатку в робоче середовище, а саме в Kubernetes cluster

Розроблена система має кращі метрики в швидкодії та споживанні ресурсів кластера, а також було використано іншу логіку для збереження конфігураційних даних, що сприяє швидшому виконанню операцій, використовує менше пам'яті та процесору, та не потребує встановлення додаткових компоненті на Kubernetes кластер.

Під час тестування виявлено, що запропонована архітектура є ефективнішою, завдяки меншому використанню ресурсів пам'яті та процесору.

У випадку багатьох кластерів ця система буде споживати менше ресурсів, тому для розробників та операторів кластерів це є великою перевагою. Це стає можливим завдяки тому, що система не залежить від кількості компонентів. У порівнянні з іншими інструментами, такими як Flux чи ArgoCD, для кожного кластеру потрібна встановлення окремих компонентів, які можуть бути досить вимогливими до ресурсів.

Отже, розроблена система має потенціал стати ефективним інструментом для оптимізації процесу автоматизації доставки та розгортання ПЗ в кластерних середовищах, забезпечуючи збалансованість ефективності, безпеки та надійності рішення.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Kim G. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations / G. Kim, W. John, A. John., 2016. – 480 с.
2. Sanjeev S. The DevOps Adoption Playbook: A Guide to Adopting DevOps in a Multi-Speed IT Enterprise / Sharma Sanjeev., 2017. – 416 с.
3. Joakim V. Practical DevOps: Harness the power of DevOps to boost your skill set and make your IT organization perform better / Verona Joakim., 2016. – 240 с.
4. Henry M. Continuous Integration (CI) and Continuous Delivery (CD): A Practical Guide to Designing and Developing Pipelines / Merode Henry., 2023. – 436 с.
5. Wes M. Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter / McKinney Wes., 2022. – 579 с.
6. Marko L. Kubernetes in Action / Luksa Marko., 2018. – 624 с.
7. Nigel P. The Kubernetes Book: 2023 Edition (Mastering Kubernetes Book 2) / Poulton Nigel., 2017. – 311 с.
8. Jennifer D. Effective DevOps: Building a Culture of Collaboration, Affinity, and Tooling at Scale / D. Jennifer, D. Ryb., 2016. – 408 с.
9. Vinto N. GitOps Cookbook: Kubernetes Automation in Practice / Natale Vinto., 2023. – 242 с.
10. GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X, and Flux / B.Yuen, A. Matyushentsev, J. Suen, T. Ekenstam., 2021. – 344 с.
11. James P. C. DevOps for the Modern Enterprise: Winning Practices to Transform Legacy IT Organizations / Patrick Cronin James., 2018.
12. Mills M. The Cloud Revolution: How the Convergence of New Technologies Will Unleash the Next Economic Boom and a Roaring 2020s / M. Mills, M. Steve., 2023.
13. Gene K. The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win / Kim Gene., 2013. – 345 с.

14. Gruver G. Leading the Transformation: Applying Agile and DevOps Principles at Scale / Gary Gruver., 2015. – 112 с.
15. Arundel J. Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud / J. Arundel, J. Domingus., 2019. – 344 с.
16. Farcic V. The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices / Viktor Farcic., 2016.
17. Мацапура В. О. Можливості використання сучасних хмарних технологій для мікро-сервісних додатків [Електронний ресурс] / В. О. Мацапура, Т. Вакалюк. – 2022. – Режим доступу до ресурсу:
<https://conf.ztu.edu.ua/kompyuterni-tekhnohiiyi-innovatsiyyi-problemy-rishennya-01-02-hrudnya-2022-r/>.
18. Newman S. Building Microservices: Designing Fine-Grained Systems / Sam Newman., 2015.
19. Cornelia D. Cloud Native Patterns: Designing change-tolerant software / Davis Cornelia., 2019. – 400 с.
20. What is GitOps [Електронний ресурс] – Режим доступу до ресурсу:
<https://about.gitlab.com/topics/gitops/>.
21. Kubernetes Overview [Електронний ресурс] – Режим доступу до ресурсу:
<https://kubernetes.io/docs/concepts/overview/>.
22. Smith G. Next Gen DevOps: Creating the DevOps Organisation / Grant Smith., 2014. – 188 с.
23. Laster B. Learning GitHub Actions: Automation and Integration of CI/CD with GitHub / Brent Laster., 2023. – 411 с.
24. What is cloud computing? [Електронний ресурс] – Режим доступу до ресурсу: <https://aws.amazon.com/what-is-cloud-computing/>.
25. Ramalho L. Fluent Python: Clear, Concise, and Effective Programming / Luciano Ramalho., 2022. – 1012 с.

26. State of DevOps 2021 [Электронный ресурс] // Google. – 2021. – Режим доступа до ресурсу: <https://services.google.com/fh/files/misc/state-of-devops-2021.pdf>.

ДОДАТКИ

Лістинг коду файлу main.py

```

import git
import os
import tempfile
from kubernetes import client, config
from flask import Flask, request, jsonify, render_template
from app.utils import generate_random_id, get_repo
from app.deploy import deploy_helm_chart
from app.config import chart_path, values_file, release_name, logger
from app.data import create_configmap

# Load Kubernetes configuration from in-cluster configuration
config.load_incluster_config()

# Create API object for interaction with Kubernetes
api_instance = client.ApiClient()

app = Flask(__name__)

temp_dir = tempfile.mkdtemp()

@app.route("/webhook", methods=["POST"])
def handle_webhook():
    if "push" in request.headers.get("X-GitHub-Event", ""):
        try:
            repo_data = get_repo(temp_dir)

            if repo_data is not None:
                deploy = deploy_helm_chart(
                    chart_path=os.path.join(temp_dir, chart_path),
                    values_file=os.path.join(temp_dir, values_file),
                    release_name=release_name,
                )
                if deploy is not None:
                    create_configmap(repo_data)

        except Exception as e:
            logger.error(
                f"Error while cloning the repository or deploying Helm chart: {e}"
            )

```

```
        return jsonify({"message": "Deployment triggered from push
event"}), 200
```

```
        return jsonify({"message": "No action taken"}), 200
```

```
if __name__ == "__main__":
    app.run(debug=True)
```

Лістинг коду файлу config.py

```
import os
from .logging import setup_logging

release_name = os.environ.get("RELEASE_NAME", "my-release")
namespace = os.environ.get("NAMESPACE", "default")
repo_url = os.environ["REPO_URL"]
github_access_token = os.environ["GITHUB_ACCESS_TOKEN"]
chart_path = os.environ.get("CHART_PATH", "/app/chart")
values_file = os.environ["VALUES_FILE"]

logger = setup_logging()
```

Лістинг коду файлу data.py

```
from kubernetes import client, config
from kubernetes.client.rest import ApiException
from .config import namespace, release_name, logger

def create_configmap(data: dict):
    # Load Kubernetes configuration from in-cluster configuration
    config.load_incluster_config()

    # Create Kubernetes API client
    v1 = client.CoreV1Api()

    configmap_name = f"{release_name}"

    # Define the ConfigMap data
    config_data = data

    # Create a ConfigMap object
    configmap = client.V1ConfigMap(
        api_version="v1",
        kind="ConfigMap",
        metadata=client.V1ObjectMeta(name=configmap_name),
        data=config_data,
    )
```

```

    # Check if the ConfigMap exists, and create or update accordingly
    try:
        try:
            existing_configmap = v1.read_namespaced_config_map(
                name=configmap_name, namespace=namespace
            )
            if existing_configmap:
                logger.info(
                    f"ConfigMap '{configmap_name}' already exists.
Attempting to update."
                )
                api_response = v1.replace_namespaced_config_map(
                    name=configmap_name, namespace=namespace,
body=configmap
                )
                logger.info(f"ConfigMap '{configmap_name}' updated
successfully.")
            except:
                if e.status == 404:
                    logger.info(
                        f"ConfigMap '{configmap_name}' does not exist.
Creating a new one."
                    )
                    api_response = v1.create_namespaced_config_map(
                        namespace=namespace, body=configmap
                    )
                    logger.info(f"ConfigMap '{configmap_name}' created
successfully.")
        except ApiException as e:
            logger.error(
                f"Exception when calling CoreV1Api to create or update
ConfigMap: {e}"
            )

```

Лістинг коду файлу deploy.py

```

import time
import subprocess
from .config import namespace, logger

def deploy_helm_chart(chart_path, values_file, release_name):
    try:
        start_time = time.time() # Record start time
        helm_command = [
            "helm",
            "upgrade",
            "--install",
            release_name,

```

```

        chart_path,
        "-f",
        values_file,
        "--namespace",
        namespace,
    ]

    helm_result = subprocess.run(
        helm_command, check=True, capture_output=True,
text=True
    )
    end_time = time.time() # Record end time
    deployment_time = end_time - start_time # Calculate de-
ployment time
    logger.info(
        f"Helm chart deployed successfully in {deploy-
ment_time:.2f} seconds!"
    )
    logger.info(helm_result.stdout) # Log Helm command output
    return True

except subprocess.CalledProcessError as e:
    logger.error(f"Error: Helm command failed with error:
{e}")
    logger.error(e.output) # Log Helm command error output
    return None

```

Лістинг коду файлу logging.py

```

import logging
import os

log_level = os.environ.get("LOG_LEVEL", "INFO")

def setup_logging():
    logging.basicConfig(
        level=log_level,
        format="%(asctime)s - %(name)s - %(levelname)s - %(mes-
sage)s",
    )

    return logging.getLogger(__name__)

```

Лістинг коду файлу utils.py

```

import random
import string
import git
import os

```

```

from .config import repo_url, github_access_token
from .config import logger

def generate_random_id(length=8):
    # Function to generate a random ID
    characters = string.ascii_lowercase + string.digits
    return "".join(random.choice(characters) for i in
range(length))

def get_repo(temp_dir: str):
    try:
        # Check if the directory already exists
        if os.path.exists(temp_dir):
            repo = git.Repo(temp_dir)
            origin = repo.remotes.origin
            origin.pull()
            logger.info("Repository updated successfully!")
        else:
            # If the directory doesn't exist, clone the repository
            if github_access_token:
                repository_path = repo_url.split("https://")[-1]
                modified_repo_url = (
                    f"https://oauth2:{github_access_token}@{repos-
itory_path}"
                )
            else:
                modified_repo_url = repo_url

            git.Repo.clone_from(modified_repo_url, temp_dir)
            repo = git.Repo(temp_dir)
            logger.info("Repository cloned successfully!")

            # Get information about the repository
            branch = repo.active_branch.name # Get the active branch
name
            # Other repository information can be fetched similarly

            # Return the repository information
            return {
                "repoName": repo.remotes.origin.url,
                "repoURL": repo_url,
                "branch": branch,
                # Include other repository information as needed
            }
    except Exception as e:
        logger.error(f"Error while fetching repository into: {e}")
        return None

```

Лістинг коду файлу Dockerfile

```
# Use the official Python image as the base image
FROM python:3.9-alpine

# Set environment variables
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
ENV FLASK_RUN_PORT=5000
ENV GIT_PYTHON_GIT_EXECUTABLE=/usr/bin/git

# Set the working directory inside the container
WORKDIR /app

# Copy the requirements.txt file to the container
COPY requirements.txt .

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt && \
    apk add git helm

# Copy the entire app directory to the container
COPY . .

# Expose the port the Flask app runs on
EXPOSE 5000

# Command to run the Flask application using Gunicorn
CMD ["gunicorn", "--bind", "0.0.0.0:5000", "main:app"]
```