НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

«На правах рукопису»
УДК 004.273

«До захисту допущено»
В.о. завідувача кафедри
_____ Едуард ЖАРІКОВ
«___»_____2021 р.

Магістерська дисертація

на здобуття ступеня магістра

за освітньо-професійною програмою «Інженерія програмного забезпечення комп'ютерних систем»

зі спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Архітектура багатокластерної системи на базі мікросервісу синхронізації»

Виконав:
студент ІІ курсу, групи ІТ-03мп
Малярчук Роман Васильович _____

Керівник:
доцент кафедри ІПІ, к.т.н., доц.,
Муха Ірина Павлівна _____

Рецензент:
доцент кафедри ІСТ, к.т.н., доц.,
Лісовиченко Олег Іванович _____

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2021 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії
Рівень вищої освіти – другий (магістерський)
Спеціальність – 121 «Інженерія програмного забезпечення»
Освітньо-професійна програма «Інженерія програмного забезпечення комп'ютерних систем»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Едуард ЖАРІКОВ
«___»_____2021р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Малярчуку Роману Васильовичу

1. Тема дисертації «Архітектура багатокластерної системи на базі мікросервісу синхронізації», науковий керівник дисертації Муха Ірина Павлівна, доцент кафедри ІПІ, к.т.н, затверджені наказом по університету від «27» жовтня 2021 р. № 3587-с

2. Термін подання студентом дисертації «6» грудня 2021 р.

3. Об'єкт дослідження – процес Continuous Delivery для багатокластерної системи

4. Предмет дослідження – архітектура багатокластерної системи на базі мікросервісу синхронізації

5. Перелік завдань, які потрібно розробити – дослідити наявні архітектурні рішення для підтримки процесу Continuous Delivery для багатокластерних систем; запропонувати архітектуру багатокластерної системи на базі мікросервісу синхронізації; реалізувати мікросервіс синхронізації; дослідити ефективність запропонованих рішень.

6. Орієнтовний перелік графічного (ілюстративного) матеріалу – 3 плакати

7. Орієнтовний перелік публікацій – одна публікація

8. Консультанти розділів дисертації

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

9.Дата видачі завдання «30» вересня 2020 р.

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Термін виконання	Примітка
1	Аналіз існуючих джерел інформації, збір необхідної інформації	08.09.2021	
2	Досліджено наявні архітектурні рішення для підтримки процесу Continuous Delivery для багатокластерних систем	15.09.2021	
3	Запропоновано архітектуру багатокластерної системи на базі мікросервісу синхронізації	22.09.2021	
4	Реалізація мікросервісу синхронізації	06.10.2021	
5	Тестування мікросервісу синхронізації	13.10.2021	
6	Виконання експериментальних досліджень	20.10.2021	
7	Оформлення пояснювальної записки	15.11.2021	
8	Подання дисертації на попередній захист	22.11.2021	
9	Подання дисертації на захист	6.12.2021	

Студент Роман МАЛЯРЧУК

Науковий керівник Ірина МУХА

РЕФЕРАТ

Актуальність дослідження. Відповідно методології CI/CD одним із важливих етапів життєвого циклу розробки програмного забезпечення є процес Continuous Delivery.

Наявні інструменти автоматизації CD для багатокластерних систем використовують архітектуру, яка вимагає адміністрування кожного з кластерів, що є достатньо складним і вартісним процесом. Окрім того, для керування workload-кластерами використовують management-кластер, який вимагає великого об’єму ресурсів та постійної підтримки у вигляді оновлення компонент management-кластеру.

Мета і завдання дослідження. Підвищення ефективності процесу автоматизації CD для багатокластерних систем за рахунок зменшення складності і вартості їх підтримки, підвищення рівня захисту облікових даних для доступу до кластерів, і як наслідок пришвидшення збірки програмного продукту.

Для досягнення зазначеної мети необхідно вирішити такі основні **завдання дослідження**:

– дослідити наявні архітектурні рішення для підтримки процесу Continuous Delivery для багатокластерних систем;

– запропонувати архітектуру багатокластерної системи на базі мікросервісу синхронізації;

– реалізувати мікросервіс синхронізації;

– дослідити ефективність запропонованих рішень.

Об’єкт дослідження - процес Continuous Delivery для багатокластерної системи.

Предмет дослідження - архітектура багатокластерної системи на базі мікросервісу синхронізації.

Наукова новизна одержаних результатів. Удосконалено архітектуру багатокластерної системи за рахунок використання мікросервісу синхронізації стану workload-кластерів з системою керування версіями, що зменшує складність

та вартість підтримки системи, підвищує захист облікових даних для доступу до кластерів, збільшує швидкість доставки програмного продукту.

Практичне значення одержаних результатів. Розроблений мікросервіс синхронізації стану workload-кластерів з системою керування версіями. Розроблена інструкція з розгортання мікросервісу в будь-якому середовищі.

КЛЮЧОВІ СЛОВА: Архітектура, кластерні системи, GitOps, IaS, система керування версіями, багатокластерні системи, мікросервіс.

SUMMARY

Actuality of theme. According to the CI / CD methodology, one of the important stages of the software development life cycle is the Continuous Delivery process.

Existing CD automation tools for multi-cluster systems use an architecture that requires the administration of each of the clusters, which is a rather complex and costly process. In addition, a management cluster is used to manage workload clusters, which requires a large amount of resources and constant support in the form of updating the components of the management cluster.

The aim of the research. Improving the efficiency of the CD automation process for multi-cluster systems by reducing the complexity and cost of their maintenance, increasing the level of protection of credentials for access to clusters, and as a consequence of accelerating the assembly of the software product.

To achieve this goal, the **following tasks** were formed:

– investigate existing architectural solutions to support the Continuous Delivery process for multicluster systems;

– to offer the architecture of a multicluster system based on a synchronization microservice;

– implement a synchronization microservice;

– investigate the effectiveness of the proposed solutions.

The object - Continuous Delivery process for a multicluster system.

The subject - architecture of a multicluster system based on a synchronization microservice.

Scientific Novelty. Improved multi-cluster system architecture through the use of microservice to synchronize the workload-clusters with the version control system, which reduces the complexity and cost of system support, increases the protection of credentials for access to clusters, increases the speed of software delivery.

The practical value. A microservice for synchronizing the status of workload clusters with a version control system has been developed. Developed instructions for deploying a microservice in any environment.

KEYWORDS: Architecture, cluster systems, GitOps, IaS, version control system, multicluster systems, microservice.

ЗМІСТ

ЗМІСТ 8

УМОВНІ СКОРОЧЕННЯ 11

ВСТУП 12

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ 14

1.1 Підходи до розробки програмного забезпечення на основі хмарних технологій 14

1.2 Безперервна доставка та розгортання 17

1.3 Kubernetes кластер 20

1.3.1 Контейнеризація 20

1.3.2 Концепція Kubernetes 21

1.3.3 Концепція Pod 23

1.3.4 Контролер реплікації і набір реплік 24

1.3.5 Концепція Service 24

1.3.6 Простір імен 25

1.3.7 Kubernetes API 25

1.3.8 Конфігурація Kubernetes-об’єктів 25

1.3.9 Багатокластерність 26

1.4 Система керування версіями 27

1.4.1 Локальні системи керування версіями 27

1.4.2 Централізована система керування версіями 28

1.4.3 Розподілена система керування версіями 29

1.5 GitOps 31

1.5.1 GitOps в кластерних системах 31

1.5.2 GitOps для багатокластерної архітектури 33

1.6 Типові архітектури програмного забезпечення 36

1.7 Постановка задач 38

1.8 Висновок 39

2 АРХІТЕКТУРА БАГАТОКЛАСТЕРНОЇ СИСТЕМИ НА БАЗІ МІКРОСЕРВІСУ СИНХРОНІЗАЦІЇ 41

2.1 Архітектура на основі GitOps агенту 41

2.2 Архітектура багатокластерної системи з безперервним розгортанням на основі мікросервісу синхронізації 42

2.2.1 Мікросервіс синхронізації 43

2.2.2 Мікросервіс роботи з Kubernetes 44

2.3 Висновок 45

3 РОЗРОБКА МІКРОСЕРВІСУ СИНХРОНІЗАЦІЇ 46

3.1 Огляд середовища для розробки мікросервісів 46

3.3 Розробка вимог 48

3.3.1 Вимоги в цілому 48

3.3.2 Вимоги до мікросервісу синхронізації 49

3.3.3 Вимоги до мікросервісу роботи з Kubernetes 49

3.4 Компоненти розробленого програмного забезпечення 50

3.4.1 Сторонні компоненти 51

3.4.2 Мікросервіс синхронізації 51

3.4.3 Мікросервіс роботи з Kubernetes 54

3.4.4 Модулі libmq, libgit 57

3.5 Висновок 58

4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ 60

4.1 Налаштування середовища 60

4.2 Тестування типової архітектури 62

3.3 Тестування архітектури на базі мікросервісу синхронізації 64

4.4 Порівняння отриманих результатів 65

4.5 Висновок 66
5 РОЗРОБКА СТАРТАП ПРОЕКТУ 67
5.1 Опис ідеї проекту 67
5.2 Технологічний аудит ідеї проекту 70
5.3 Аналіз ринкових можливостей запуску стартап-проекту 73
5.4 Розроблення ринкової стратегії проекту 83
5.5 Розроблення маркетингової програми стартап-проекту 87
5.6 Висновок 92
ВИСНОВКИ 93
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ 96
ДОДАТОК А. ЛІСТИНГ ПРОГРАМИ 99
ДОДАТОК Б. РЕЗУЛЬТАТ ПЕРЕВІРКИ НА СПІВПАДІННЯ 107

УМОВНІ СКОРОЧЕННЯ

ОС – операційна система

БД – База даних

ПЗ – програмне забезпечення

CD – Continuous Delivery (безперервна доставка та розгортання)

API – Application Program Interface (прикладний програмний інтерфейс)

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

ВСТУП

Життєвий цикл розробки програмного забезпечення (ПЗ) є застосуванням інженерних практик, спрямованих на підтримку працездатності, якості та надійності ПЗ [1].

Вагоме місце серед них займає DevOps (Development & Operations) – сукупність практик, що поєднують процес розробки ПЗ (Development) та його експлуатації (Operation), а також відповідні інструменти автоматизації для підвищення ефективності цих процесів за рахунок безперервної інтеграції і розгортання (Continuous integration & Continuous delivery, CI/CD) та активної взаємодії профільних фахівців [2].

Однією із таких практик є GitOps, яка здійснює підтримку CD [3]. GitOps дозволяє використовувати найкращі практики в галузі розробки ПЗ, такі як контроль версій, спільна робота, відповідність вимогам, CI/CD, для вирішення задач автоматизації управління конфігураціями інфраструктури та додатків.

Важливою відмінністю GitOps-підходу є автоматизація процесу управління інфраструктурою на основі CI/CD. Для підтримки процесу управління інфраструктурою зазвичай запускають спеціальну програму-агент (або gitops-агент), яка працює прямо в оточенні системи та змінює її зсередини. Агент є відповідальним за те, щоб здійснювати оновлення даних з системи контролю версій та при наявних змінах оновлювати стан Kubernetes- кластеру. На сьогодні існують такі реалізації gitops-агентів, як FluxCD, ArgoCD, PipeCD, Faros і т. д. Їхньою спільною властивістю є те, що вони вимагають встановлення додаткового ПЗ на кластер. При великій кількості кластерів в системі, необхідна кількість ресурсів для її роботи зростатиме пропорційно. Це призводить до збільшення витрат на обслуговування системи, зменшення її швидкодії, що може бути критичним для компаній будь-якого розміру. Більше того, необхідно підтримувати встановлені gitops-агенти в кластерах, що вимагає додаткових затрат на людський ресурс. Також є проблема з безпекою доступу до кластерів,

оскільки в разі, якщо зловмисник отримає доступ до основного кластеру, він легко зможе отримати доступ до інших кластерів.

Отже, основною метою створення архітектури є зменшення витрат на обслуговування системи, підвищення швидкодії безперервної доставки та розгортання, підвищення безпеки доступу до облікових даних. Об’єктом дослідження є процес Continuous Delivery для багатокластерної системи, а предметом дослідження - архітектура багатокластерної системи на базі мікросервісу синхронізації.

Для досягнення зазначеної мети необхідно вирішити такі основні завдання дослідження:

– дослідити наявні архітектурні рішення для підтримки процесу Continuous Delivery для багатокластерних систем;

– запропонувати архітектуру багатокластерної системи на базі мікросервісу синхронізації;

– реалізувати мікросервіс синхронізації;

– дослідити ефективність запропонованих рішень.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Підходи до розробки програмного забезпечення на основі хмарних технологій

Розробка сучасного програмного забезпечення (ПЗ) – складний і багатофакторний процес, який передбачає реалізацію ряду етапів, які проходить програмний продукт від появи ідеї до реалізації цієї ідеї в коді, імплементації в бізнес та подальшої підтримки [1].

Життєвий цикл розробки ПЗ є застосуванням інженерних практик, спрямованих на підтримку працездатності, якості та надійності ПЗ [4].

Серед них вагоме місце займає DevOps (Development & Operations) – сукупність практик, що поєднують розробку програмного забезпечення (Development) та його експлуатацію (Operation), а також відповідні інструменти автоматизації для підвищення ефективності цих процесів за рахунок безперервної інтеграції та активної взаємодії профільних фахівців [2].

З точки зору управління, DevOps позиціонується як agile-підхід для усунення організаційних та тимчасових бар'єрів між командами розробників та інших учасників життєвого циклу ПЗ, щоб учасники могли швидше і надійніше збирати, тестувати і випускати нові релізи програмних продуктів [5].

Основними перевагами DevOps над іншими методологіями є повна автоматизація, вищий рівень якості ПЗ, менша кількість часу на повний цикл розробки ПЗ [6]. Якщо механізм доставки змін даватиме регулярні збої, то час необхідний для всього циклу розробки збільшиться.

Також важливо розуміти, що під поняттям DevOps криється практика взаємодії двох команд розробників та адміністраторів.

DevOps підхід реалізується в декілька етапів, які зображені на рисунку 1.1, за допомогою певних інструментів [7].

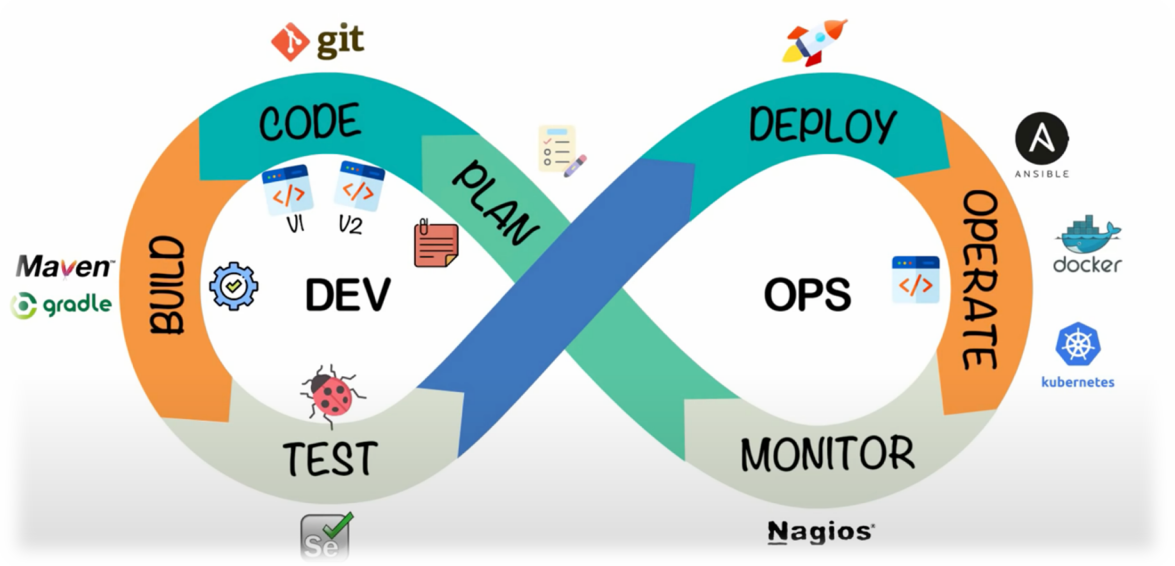


Рисунок 1.1 – Етапи DevOps підходу [7]

Розглянемо кожен з цих етапів.

1) Етап планування. На цьому етапі здійснюється планування, визначення того, що повинно бути реалізовано для клієнта.

2) Етап розробки. Здійснюється розробка ПЗ. На цьому етапі можлива робота з декількома версіями ПЗ. Версії зберігаються в системі керування версіями, наприклад Git, яка дозволяє записувати зміни в файлі або в наборі файлів (директорії) та в будь-який момент часу переключатися між різними записаними версіями.

3) Етап збірки. Код пакується у виконувані файли для того, щоб він був готовий для розгортання та виконання.

4) Етап тестування. На цьому етапі здійснюються різні типи тестування, в тому числі end-to-end тестування, яке дозволяє протестувати систему в цілому. Найбільш популярним інструментом тут є Selenium, який дозволяє автоматизувати процес тестування. Також запускаються unit-тести, які дозволяють виявити типові помилки на ранньому етапі.

5) Етап інтеграції. Ядро життєвого циклу DevOps. На цьому етапі відбувається заливка виконуваних файлів на віддалені сервери. При цьому виконувані файли позначаються певними мітками, відповідно до результатів тестування. Основним відкритим інструментом для цього етапу є Jenkins.

6) Етап доставки та розгортання. Як тільки ПЗ протестовано, воно може бути доставлено на продуктивне середовище та розгорнуто на ньому. Команда адміністраторів здійснює розгортання ПЗ, використовуючи такі системи автоматизації, як Docker (система керування контейнерами), Kubernetes.

7) Етап моніторингу. Після розгортання ПЗ його стан постійно моніториться з метою виявлявлення аномалій в роботі. На цьому етапі використовуються такі інструменти автоматизації моніторингу як Prometheus, Grafana.

Існує п'ять підходів до реалізації DevOps-практик. Розглянемо детально кожен із них.

1) Перший підхід передбачає, що команда DevOps-інженерів розглядається як основна при прийнятті рішень стосовно формування вимог до ПЗ. Такий підхід задовольняє умову того, що повинен забезпечуватися високий рівень якості моніторингу. Адміністрування має набір вимог, які стосуються логування та моніторингу, наприклад логування інформації з сервісів повинне бути корисним, легким для розуміння DevOps-інженером. Заохочування DevOps-інженерів до розробки вимог дасть гарантію того, що ці вимоги будуть дотримані.

2) Другий підхід передбачає, що розробники повинні нести відповідальність за збої в роботі ПЗ, які відбуваються в продуктивному середовищі. Така практика необхідна для того, щоб зменшити час між першим виявленням збою та відновленням коректності роботи розгорнутого ПЗ. Як правило, компанії, які практикують такий підхід, мають період часу, коли розробник є відповідальним за розгорнуте ПЗ. Після цього періоду відповідальність бере на себе адміністратор служби підтримки.

3) Третій підхід передбачає спільну роботу розробників та DevOps-інженерів в процесі розгортання ПЗ. Ця практика призначена для підвищення якісті процесу розгортань. Це дозволяє уникнути помилок, спричинених випадковими розгортаннями та, як наслідок, неправильними налаштуваннями. Це також стосується часу, необхідного для діагностики та усунення помилок. Зазвичай процес розгортання повинен дозволяти легко простежити історію

певного артефакту розгортання та знати компоненти, які були включені в цей артефакт.

4) Наступний підхід – це використання безперервного розгортання (CD). Підтримка процесу безперервної доставки та розгортання допомагає зменшити час між етапами внесення змін у вихідний код та його заливку в продуктивне середовище. Також безперервна доставка та розгортання стимулює створювати потужний арсенал тестів для автоматичного тестування, щоб підвищити якість коду перед заливкою на продуктивне середовище.

5) Останій підхід - це розробка інфраструктурного коду, який є програмним описом мережевих ресурсів. Даний програмний код реалізується у формі скриптів розгортання, до яких повинні застосовуватися такі ж вимоги, як і до коду ПЗ. Такий підхід дозволить мати високу якість розгорнутого ПЗ і майбутні розгортання відбуватимуться планово. Помилкові сценарії розгортання, наприклад неправильна конфігурація, можуть викликати помилки в роботі застосунку, в процесі розгортання, а також недопустимі зміни стану середовища. Застосування методів контролю якості, які використовуються при розробці ПЗ та сценаріїв розгортання, допоможе контролювати якість інфраструктурного коду.

Серед розглянутих підходів використання безперервного розгортання має ряд суттєвих переваг, зокрема можливість розгортання в будь-який момент часу без попереднього планування та підготовки [8]. Це дозволяє уникати технічного боргу, здійснювати дрібні релізи, які з меншою ймовірністю призведуть до збоїв в роботі ПЗ, яке розробляється. Якщо ж говорити про задачу мінімізації часу, то варіант з безперервною доставкою також є найоптимальнішим, оскільки дозволяє зменшити час даного етапу за рахунок того, що він відбувається в автоматичному режимі без необхідності залучення DevOps-інженерів.

1.2 Безперервна доставка та розгортання

Однією із головних проблем з якою стикаються при розробці ПЗ є питання швидкої доставки змін, внесених в програмний код, до кінцевого користувача [9].

Розглянемо декілька варіантів реалізації даного процесу.

1) Компанії здійснюють доставку та розгортання ПЗ вручну. Кожний крок розгортання здійснюється певним членом команди. Такі кроки є атомарними та ізольованими між собою. Різний порядок виконання цих кроків, людський фактор, різні часові проміжки можуть давати різний результат, що часто приводить до некоректної роботи ПЗ.

Основними недоліками, які притаманні цьому варіанту, є:

– ручне тестування перевірки працездатності ПЗ в штатному режимі;

– значний час розгортання ПЗ, оскільки всі команди розгортання виконуються вручну системним адміністратором;

– можливість отримання неочікуваного результату, що вимагає скасування поточної доставки та розгортання і повернення до попередньої версії ПЗ;

– велика база документації з детальними інструкціями щодо реалізації процесу розгортання, яку часто забувають оновлювати, що веде до збоїв у роботі ПЗ та неможливості здійснити його розгортання.

2) Розгортання ПЗ, розробка якого здійснюється на середовищі, подібному до продуктивного. Це відбувається у випадках, коли у команд, які здійснюють розробку ПЗ та/або розробку сценаріїв розгортання, немає прямого доступу до продуктивного середовища.

Недоліки такого варіанту:

– системні адміністратори, які здійснюють розгортання ПЗ, яке було відтестовано на середовищі, подібному до продуктивного, вперше стикаються з новими інструкціями розгортання та кодом ПЗ, що може призвести до помилок при розгортанні ПЗ за рахунок людського фактору;

– реалізація середовища, подібного до продуктивного, на якому відбувається розробка та тестування ПЗ, вимагає постійної підтримки та захисту від людських помилок;

– майже відсутня комунікація між командою розробників та командою системних адміністраторів, які займаються доставкою та розгортанням розробленого ПЗ;

– команда розробників повинна готувати конфігураційні файли для продуктивного середовища без тестування цих конфігурацій на ньому, адже розробка ПЗ здійснюється на іншому середовищі, подібному до продуктивного.

3) Безперервна доставка і розгортання (CD). Головними вимогами в цьому варіанті є те, щоб розгортання було автоматизованим та відбувалося часто. Ідея CD полягає в тому, що за відсутності автоматизації таких процесів, як збірка, тестування, розгортання, кожного разу під час розгортання ПЗ отримується різний результат, оскільки відбуваються зміни у вихідному коді та налаштуваннях продуктивного середовища. За відсутності автоматизації всі кроки розгортання ПЗ відбуваються в ручному режимі, що часто призводить збоїв у процесі розгортання ПЗ. При цьому системні адміністратори не мають надійних інструментів для того, щоб відновити попередню версію ПЗ, застосованих налаштувань в процесі розгортання ПЗ. Висока частота розгортання дозволяє зменшити різницю між версією, яка розгортається, та поточною версією ПЗ. Це, в свою чергу, зменшує ризик неуспішного розгортання та дає можливість легко відновити попередню версію ПЗ, оскільки різниця між ними не є великою.

Недоліками цього підходу є те, що іноді від замовників поступає вимога щодо меншої кількості розгортань ПЗ.

Отже, в контексті задачі мінімізації часу, необхідного на етап доставки та розгортання, варіант з безперервною доставкою має більше переваг, ніж інші. Серед них: висока частота розгортання, що сприяє зменшенню різниці між поточною версією та версією яка розгортається; менший технічний борг; спрощення процесу розробки ПЗ; спрощення процесу відслідковування помилок на продуктивному середовищі, оскільки зміни доставляються невеликими порціями.

1.3 Kubernetes кластер

Для того, щоб якісно реалізувати автоматизацію безперервної доставки та розгортання згідно до DevOps-підходу, необхідно мати надійну та зручну систему оркестрації. Оркестрація — це автоматизована конфігурація, керування та координація комп'ютерних систем, додатків і служб [10]. Основною задачею та функцією системи оркестрації є керування контейнерами, сутність яких детально розглянута в наступному підпункті. В якості такої системи оркестрації найчастіше використовується технологія Kubernetes, запропонована компанією Google в середині 2014 року.

Для того, щоб перейти до поняття Kubernetes-кластеру, розглянемо спершу таке поняття, як контейнеризація, на основі якого базується робота Kubernetes.

1.3.1 Контейнеризація

Контейнеризація – це підхід, в якому програмний застосунок, необхідні бібліотеки та інструменти для його роботи, ОС та її налаштування упаковуються в образ (image), який пізніше можна запустити (розгорнути) та отримати запущений контейнер (container) [11].

Контейнер — це стандартна одиниця ПЗ, яка упаковує вихідний код застосунку і всі його залежності, щоб програма швидко й надійно працювала від одного обчислювального середовища до іншого.

Контейнеризація дозволяє досягнути високого рівня ізоляції контейнерів, запущених на спільній ОС. Такий підхід стандартизує розгортання застосунків, дозволяючи використовувати для контейнера необхідну ОС (Windows, Linux) запускаючи їх на будь-якій з ОС (Windows, Linux)[11].

Перевагою використання контейнеризації при запуску програмних застосунків є можливість швидко запустити застосунок у точно визначеному середовищі, що значно спрощує запуск застосунків на різних ОС.

Варто зауважити, що контейнеризація надає не найвищий рівень ізоляції, на відміну від віртуалізації, яка використовує віртуальні машини для запуску

застосунків. Недоліком віртуалізації є те, що образи є значно більшими та вимагають більше ресурсів ОС для запуску.

На рисунку 1.2 наведено схематичне зображення різниці між контейнером та віртуальною машиною.

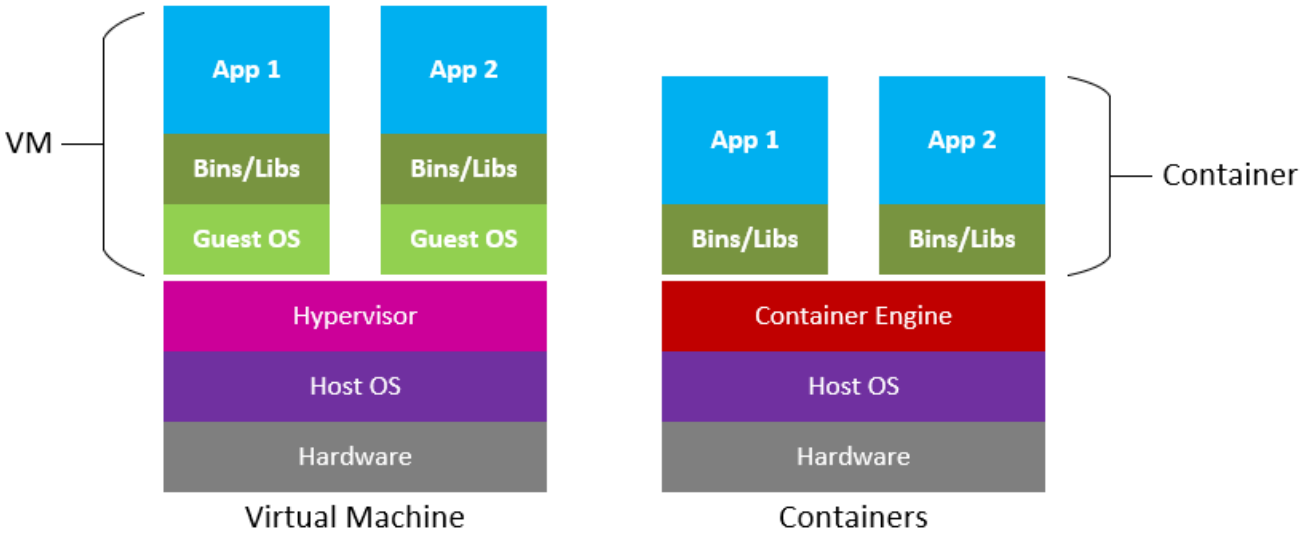


Рисунок 1.2 – Схематичне зображення різниці між контейнером та віртуальною машиною [12]

1.3.2 Концепція Kubernetes

Kubernetes – це система оркестрації для автоматизованого розгортання, управління, масштабування контейнеризованими застосунками [13]. Kubernetes-кластер містить набір хостів, необхідних для того, щоб запускати контейнери з ПЗ. Такі хости називаються нодами (nodes) Kubernetes-кластер. Нода може бути, як фічичною, так і віртуальною. Кожна Kubernetes-нода запускає декілька Kubernetes-компонент, таких як kubelet, kube-proxy. Кожний Kubernetes-кластер повинен мати як мінімум одну master-ноду. Master-нода є відповідальною за те, щоб підтримувати кластер в потрібному стані.

Master-нода Kubernetes-кластеру містить такі компоненти:

– etcd – розподілене сховище, що служить для зберігання даних про стан кластеру, метаданих про всі об’єкти, які були створені в Kubernetes-кластері;

– API server, що перевіряє та налаштовує дані для об’єктів API, які включають модулі, служби, контролери реплікації та інші. API server обслуговує операції REST (REpresentational State Transfer) і надає інтерфейс до спільного стану кластера, за допомогою якого взаємодіють усі інші компоненти;

– kube-scheduler, що є виконуваним файлом, який присвоює Pod`ам, що представляють собою мінімальні виконуванані одиниця в Kubernetes, відповідну ноду, на якій Pod`и буду запускатися. Kube-scheduler визначає набір нод, що є валідними для кожної з Pod, які очікують в черзі на розподілення, в залежності від наявних в кластері обмежень та ресурсів;

– kube-controller-manager, що є виконуваним файлом, в якому запускається головний нескінченний цикл регулювання стану кластеру. Kube-controller-manager моніторить стан кластеру за допомогою API-серверу і здійснює необхідні зміни, намагаючись наблизити поточний стан Kubernetes-кластеру якомога ближче до бажаного стану.

Мінімально рекомендованою кількістю master-нод є три ноди. Більше того, в кожен момент часу тільки одна master-нода є активною і вважається лідером. Інші ж master-ноди працюють в пасивному режимі, очікуючи, доки лідер дасть збій, для того, щоб переобрати нового лідера, серед доступних master-нод. Лідер обирається за допомогою алгоритму оптимістичного блокування, який передбачає, що кожна з master-нод намагається першою захопити ресурс і кому це вдається – автоматично стає лідером. Після цього, лідеру, кожні 2 секунди необхідно підтверджувати своє лідерство, оновлюючи метадані в захопленому ресурсі.

Кожний Kubernetes-кластер може мати worker-ноди, які є хостами що служать для запуску контейнеризованих застосунків. Для того, щоб запускати, моніторити та підтримувати всі необхідні сервіси, на worker-ноді необхідні наступні компоненти:

– docker, containter.io або інше середовище виконання контейнерів, які, власне, відповідають за запуск контейнерів;

– kubelet, що взаємодіє з API-сервером і керує контейнерами на ноді, на якій він запущений;

– kubernetes Service Proxy (kube-proxy), який розподіляє навантаження між різними програмними застосунками.

Кількість worker-нод є необмеженою, оскільки видаляти або додавати їх у Kubernetes-кластер можна в будь-який час.

На рисунку 1.3 схематично зображено компоненти Kubernetes-кластеру.

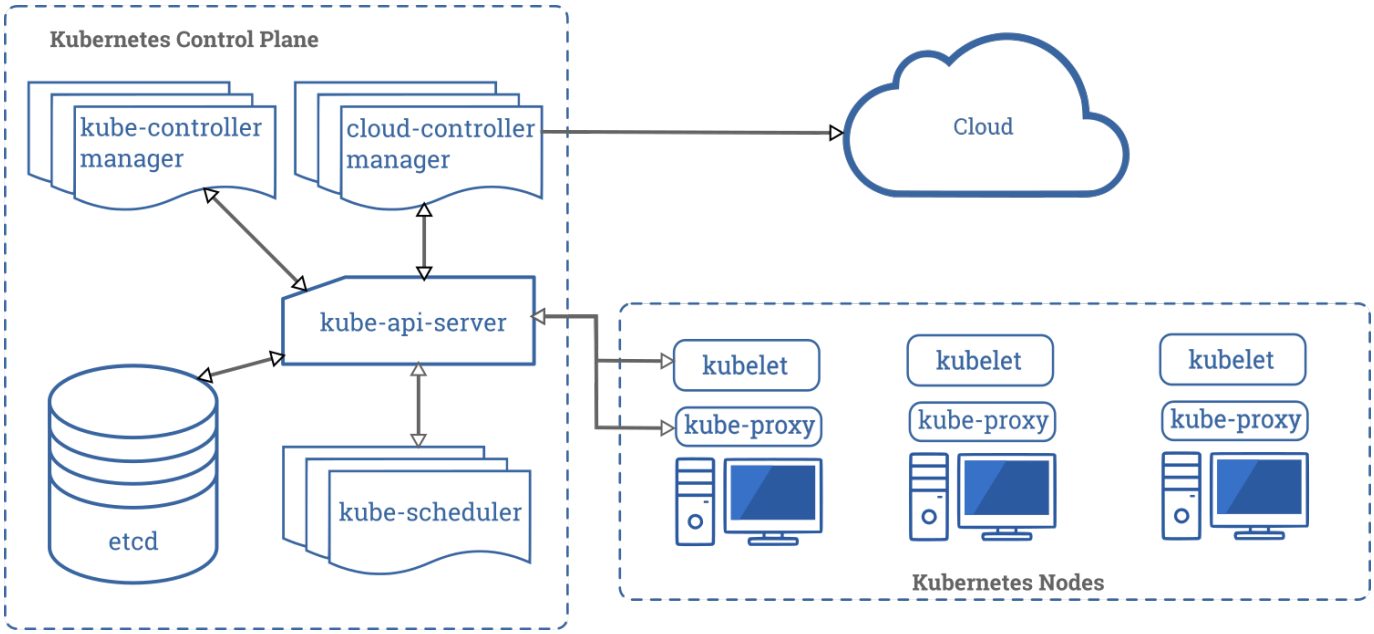


Рисунок 1.3 – Схема компонент Kubernetes-кластеру [14]

1.3.3 Концепція Pod

Як уже зазначалося, Pod (Под) є найменшою одиницею виконання в Kubernetes. Pod складається з, як мінімум, одного контейнера [13]. Pod завжди розміщується повністю на одній worker-ноді. Кожному Pod`у виділяється окремий IP-адрес, тому Pod`и можуть комунікувати між собою. При перезапуску Pod`а не зберігає свого стану. Також до Pod`у можна приєднати Volume (віртуальний жорсткий диск), який дозволяє зберігати дані і стан Pod`и, якщо це є необхідним. Кожна Pod`а має свій власний Universally Unique Identifier (UUID) для того, щоб Pod`и можна було відрізнити одна від одної.

1.3.4 Контролер реплікації і набір реплік

Контролер реплікації і набір реплік призначені для того, щоб керувати групою Pod`ів і завжди бути впевненим в тому, що бажана кількість Pod`ів є запущеною. Група Pod`ів, якою необхідно керувати, визначається за допомогою label selector, який являє собою пару ключ-значення. Kubernetes гарантує, що необхідна кількість Pod`ів завжди буде підтримуватися в статусі Running. Як тільки Pod перейде в інший статус, Kubernetes запустить новий Pod [13].

Контролеру реплікації і набір реплік є основою для реалізації таких сутностей, як Deployment, Job, DaemonSet. Deployment – це Kubernetes-об’єкт, що інкапсулює в собі інші Kubernetes-об’єкти, які є необхідними для розгортання ПЗ.

1.3.5 Концепція Service

Pod`и, хоча й мають власні IP-адреси, проте не є доступними для користувачів, які не знаходяться в середині Kubernetes-кластеру. Саме для цього і існують сервіси. Сервіси в Kubernetes використовуються для того, щоб надати доступ до Pod`ів для користувачів ззовні Kubernetes-кластеру [13]. Вони також використовують фільтр label selector для того, щоб визначити, до якої групи Pod`ів надавати доступ.

Є різні типи сервісів:

– ClusterIP – тип за замовчуванням, надає сервісу IP, який є валідним тільки в середині Kubernetes-кластеру;

– NodePort – надає доступ до сервісу, відкриваючи на кожній ноді вказаний у конфігурації порт;

– LoadBalancer – надає доступ до сервісу завдяки тому, що створює балансувальник навантаження на стороні хмарного провайдеру та надає сервісу статичну зовнішню IP-адресу;

– ExternalName – надає доступ до сервісу завдяки тому, що використовується externalName.

1.3.6 Простір імен

Простір імен (Namespace) в Kubernetes-кластері є сутністю, яка визначає віртуальний кластер всередині фізичного кластеру [13]. Namespace`и є ізольовані один від одного та їхня кількість є необмеженою. Оскільки Namespace`и ізольовані, то вся комунікація між об'єктами, які знаходяться в різних namespace`ах відбувається через публічні інтерфейси, наприклад service`и. Варто зауважити, що kube-scheduler може розміщувати об'єкти з різних namespace`ів на одній і тій ж ноді.

1.3.7 Kubernetes API

Вся робота з станом Kubernetes-кластеру відбувається через Kubernetes API [13]. Є декілька варіантів для того, щоб його викликати, серед яких, kubectl cli, Kubernetes SDK (для викликів API з власних програмних застосунків), використовуючи виклики на 7-ому рівні мережевої моделі OSI.

Загальний вигляд URL для API:

/api/v{version}/{objectName}/,

де version – версія, objectName – назва Kubernetes об'єкту, наприклад, Pods, Services, Deployments.

Для того, щоб мати доступ до Kubernetes API, необхідно мати ключі доступу. Їх можна скопіювати з файлу /var/run/secrets/kubernetes.io/serviceaccount. Перед цим з під root-користувача (користувач з необмеженими правами) попередньо необхідно створити serviceaccount з необхідними правами.

1.3.8 Конфігурація Kubernetes-об'єктів

Більшості ПЗ для своєї роботи потрібно багато Kubernetes-об'єктів, наприклад, deployment, service, configmap. Керування цими об'єктами може бути спрощене завдяки їхньому групуванню по yaml-файлах, які є декларативним

описом Kubernetes-об’єктів. Більше того, також можна розміщувати всі об'єкти в одному yaml-файлі. Для цього необхідно після кожного Kubernetes-об’єкту ставити рядок-розділювач «---».

Після цього простою командою

kubectl apply -f file

можемо створити всі об’єкти, які описані в файлі *file*. Kubernetes-об’єкти будуть створюватися в тому ж порядку, в якому вони розміщуються в yaml-файлі.

Для того, щоб видалити необхідні Kubernetes-об’єкти, достатньо запустити команду

kubectl delete -f file

Kubernetes на основі файлу *file*, визначить всі Kubernetes-об’єкти, які необхідно видалити.

Для того, щоб відредагувати потрібні Kubernetes-об’єкти, достатньо запустити команду

kubectl edit {objectType} {objectName},

де *objectType* – тип Kubernetes-об’єкту, *objectName* – назва Kubernetes-об’єкту.

Наприклад, *kubectl edit pods applicationPod*

1.3.9 Багатокластерність

Багатокластерність в Kubernetes - це середовище з багатьма Kubernetes-кластерами [15].

Велика кількість компаній використовують багатокластерність в Kubernetes, оскільки за такого підходу досягається високий рівень ізоляції кожного з кластерів. Також, як правило, кодова база та інфраструктура кожного з кластерів мають спільні основні налаштування та компоненти, відрізняючись тільки певними налаштуваннями, специфічними для кожного з кластерів. Для того, щоб створити кластери необхідні певні ресурси. Для цього можна використати ресурси, які надає хмарний провайдер. Як правило, хмарний провайдер надає список з регіонів, в яких розміщуються його фізичні дата-центри, де можна розміщувати кластери.

Налаштувати таке середовище можна декількома шляхами:

– один кластер на одну зону доступності, в одному регіоні;

– один кластер на регіон, але використовуючи один хмарний провайдер;

– багато кластерів в багатьох провайдерів.

1.4 Система керування версіями

Система керування версіями (version control system, VCS) дозволяє записувати зміни в файлі або в наборі файлів (директорії) та в будь-який момент часу переключатися між різними записаними версіями. Сучасні методології розробки ПЗ, зокрема DevOps-практики, часто потребують системи керування версіями. Розглянемо, які існують системи керування версіями.

1.4.1 Локальні системи керування версіями

Локальна система керування версіями вирішує базову проблему користувачів, а саме базове версіонування файлів. Локальна VCS дозволяє зберігати зміни файлу в локальній БД. Відповідно, користувач має змогу бачити ці версії та відновлювати необхідну йому версію. Найвідомішим прикладом такого типу систем є RCS. Великим недоліком таких систем є те, що при пошкодженні жорсткого диску, втрачається як і сам файл, так і його версії, адже БД зберігається локально.

На рисунку 1.4 наведено приклад роботи локальної системи керування версіями

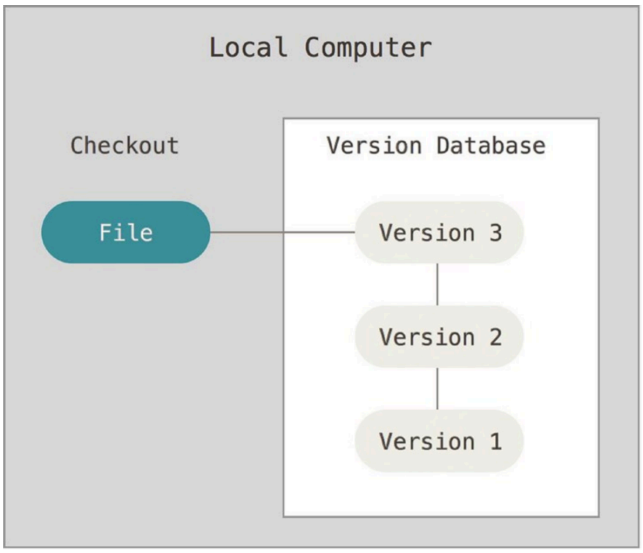


Рисунок 1.4 - Приклад роботи локальної VCS [16]

1.4.2 Централізована система керування версіями

Централізована VCS дозволяє різним користувачам бачити зміни, які вони зробили в файлі або в наборі файлів [16]. Це досягається за рахунок того, що є центральний сервер, який зберігає всі версії файлів в центральній БД. У користувача такої централізованої VCS локально доступною є тільки та версія, з якою він працює. Після внесення змін, користувач може надіслати оновлену версію файлу або набору файлів на збереження в центральну БД. Недоліком такої централізованої VCS є використання лише одного серверу, у разі відмови якого користувачі втратять доступ до всіх версій, окрім тієї, яка збереглася на їхньому локальному ПК. Основними прикладами централізованої VCS є CVS, Subversion, Perforce.

На рисунку 1.5 наведено приклад роботи централізованої VCS

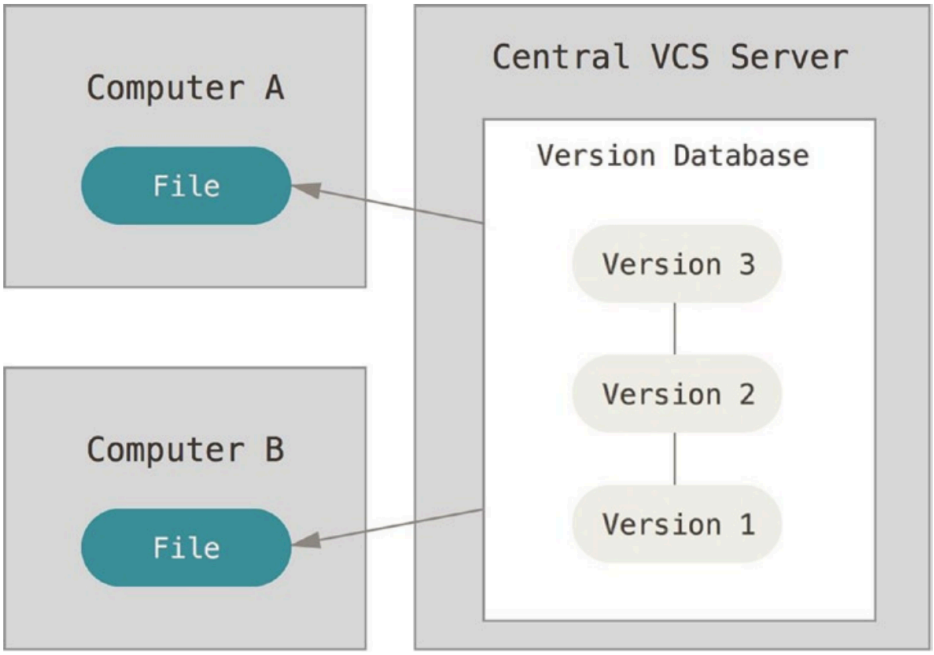


Рисунок 1.5 – Приклад роботи централізованої VCS [16]

1.4.3 Розподілена система керування версіями

Основною відмінністю розподіленої VCS від централізованої VCS є те, що користувач локально зберігає не одну версію файлу, з якою працює, а повну інформацію про всі версії всіх файлів. Це реалізовано завдяки тому, що локальна БД майже завжди є копією центральної БД. Основною перевагою систем цього типу є те, що при збої центральної БД завжди можна відновити репозиторій з локальної версії будь-якого користувача. Це значно підвищує надійність VCS. Основними прикладами систем такого типу є Git, Mercurial, Bazaar, Darcs.

На рисунку 1.6 наведено приклад роботи розподіленої VCS

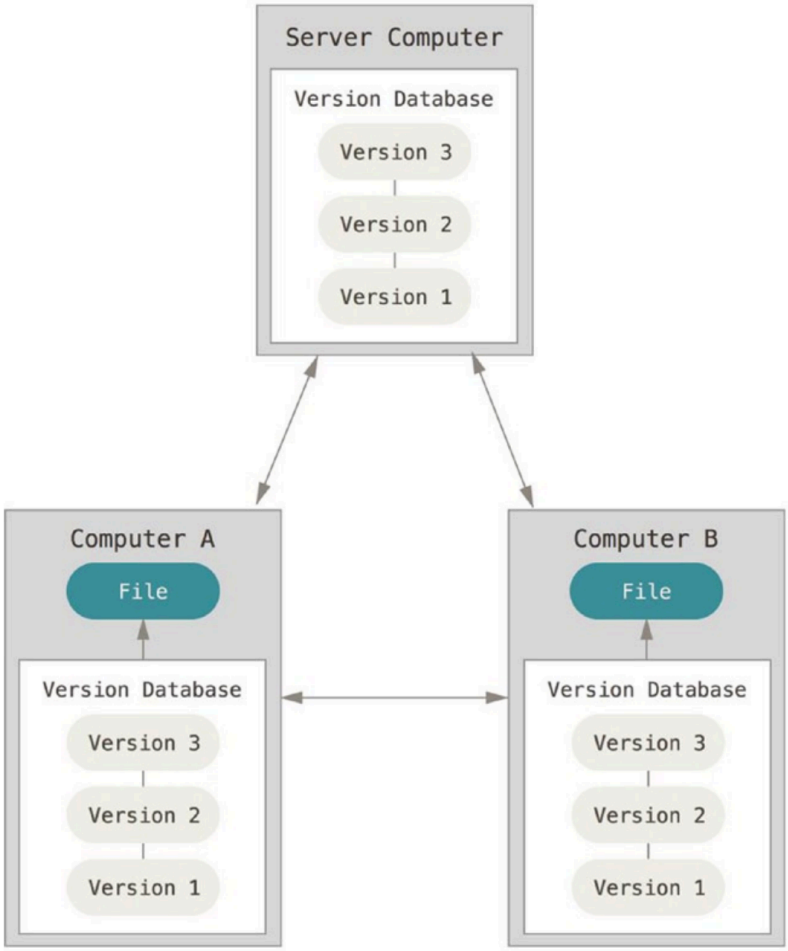


Рисунок 1.6 - Приклад роботи розподіленої VCS [16]

Найпопулярнішою з усіх розподілених систем керування версіями є Git.

В контексті задачі підвищення ефективності процесу автоматизації CD для багатокластерних систем найефективнішим рішенням є використання саме розподіленої VCS. Перевагами цього варіанту є розподіленість, що збільшує безпеку за рахунок багатьох копій репозиторія, велика кількість сучасних інструментів та сторонніх сервісів для роботи.

1.5 GitOps

1.5.1 GitOps в кластерних системах

GitOps є одним із варіантів реалізації DevOps, який здійснює підтримку безперервної доставки та розгортання [3]. Базовим елементом GitOps є IaS – модель відповідно якої опис і управління інфраструктурою проєкту здійснюється через конфігураційні файли, тобто програмно, а не через ручне редагування конфігурацій на серверах чи інтерактивну взаємодію. Опис інфраструктури здійснюється в декларативному стилі. Це дає змогу легко робити копії елементів інфраструктури або відновлювати її стан. В даному випадку поняття IaS слід розуміти в широкому сенсі: воно включає в себе також Network as Code, Policy as Code, Configuration as Code, Security as Code.

Популярними інструментами, які дозволяють реалізувати IaS є Terraform, CloudFormation, файли-маніфести Kubernetes. Отже, замість того, щоб в ручному режимі створювати сервери, мережі, конфігурації, політики в AWS та Kubernetes-кластер з потрібними елементами, можна описати всю нашу інфраструктуру за допомогою Terraform, CloudFormation, файлів-маніфесту Kubernetes. В такому випадку отримаємо файли декларативних форматів (наприклад, yaml), які повністю описують інфраструктуру, платформу та їхню конфігурацію.

GitOps передбачає версійність як програмного, так і конфігураційного коду шляхом розміщення коду інфраструктури у репозиторіях системи керування версіями, зазвичай Git, разом із ПЗ.

Git-репозиторій є єдиним джерелом істини в GitOps. Єдине джерело істини – це практика структурування вихідого коду і пов'язаних даних таким чином, що кожен елемент даних редагується лише в одному місці (git-репозиторії).

Всі зміни коду застосунку та інфраструктури здійснюються через Git та механізм Pull Request (PR), що дозволяє забезпечити взаємодію усіх розробників, які повинні схвалити запропоновані зміни, у процесі ревю (review) та тестування (testing) [18]. Це дозволить убезпечити систему від випадкових помилок, які можуть повністю її зламати.

Таким чином, Git і поділ доступу в Git стає єдиним місцем (єдиним джерелом істини) не тільки для коду та інфраструктури, але і для політики доступу до різних частин проєкту.

Ще однією відмінністю GitOps-підходу є автоматизація процесу управління інфраструктурою на основі CI/CD. Система сама визначає та усуває різницю між тим, що перебуває в Git та тим, що реально крутиться на серверах. Тобто відбувається процес синхронізації git-репозиторія із станом Kubernetes-кластеру, який представляє собою сукупність вузлів, які запускають контейнеризовані програми.

Для підтримки процесу управління інфраструктурою зазвичай запускають спеціальну програму-агент, яка працює прямо в оточенні системи та змінює її зсередини. Агент є відповідальним за те, щоб здійснювати оновлення даних з системи контролю версій та при наявних змінах оновлювати стан Kubernetes-кластеру. Найпопулярнішими реалізаціями агентів є Weave Flux, ArgoCD, Gitkube, Watchtower, keel.sh. На рисунку 1.2 наведемо діаграму процесу CD для Weave Flux.

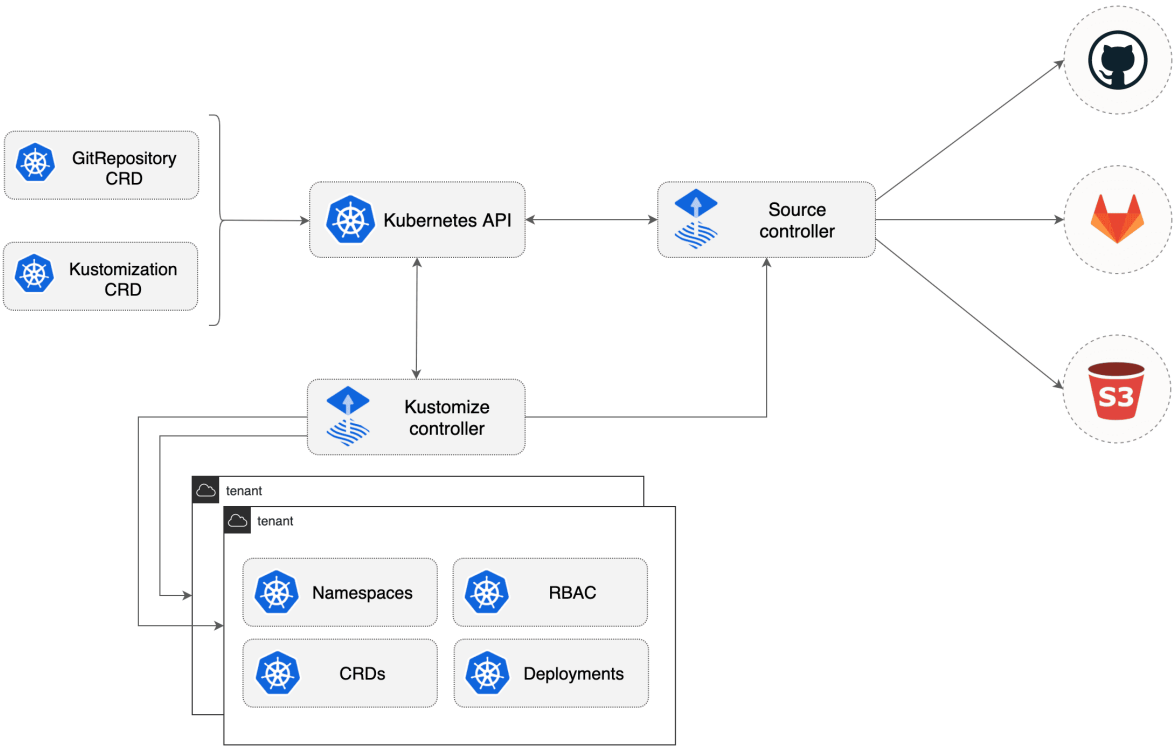


Рисунок 1.7 – Процес CD для Weave Flux [17]

Як можемо бачити з діаграми gitops-агент має два компоненти: source-контроллер та kustomize-контролер. Source-контролер є відповідальним за підтримку процесу авторизації в VCS, моніторингу змін вVCS, валідації цих змін у відповідності із синтаксисом мови розмітки (наприклад, yaml), створення CRD-об’єктів (Custom Resource Definition), що являють собою користувацькі Kubernetes-об’єкти на основі яких буде здійснюватися оновлення стану Kubernetes-кластеру.

Kustomize-контролер здійснює оновлення стану кластеру на основі CRD-об’єктів, створених source-контролером, видалення Kubernetes-об’єктів відповідно до змін в VCS.

Самі контролери знаходяться всередині Kubernetes та працюють як звичайні Kubernetes-оператори, які мають доступ до Kubernetes-API для того, щоб змінювати стан кластеру.

1.5.2 GitOps для багатокластерної архітектури

Насьогодні багато компаній потребують далеко не одного фізичного кластеру, відповідно виникає проблема в тому, як правильно побудувати архітектуру такої системи. Якщо розглядати цю проблему в контексті GitOps, то, як було сказано раніше, для роботи такого підходу необхідний агент, який буде керувати станом кластеру. Такий агент повинен бути встановлений для кожного кластеру. У разі виходу нової версії агенту необхідно здійснювати його оновлення на кожному кластері.

Отже, розглянемо детальніше цей підхід [18]. Архітектура багатокластерної системи зображена на рисунку 1.8

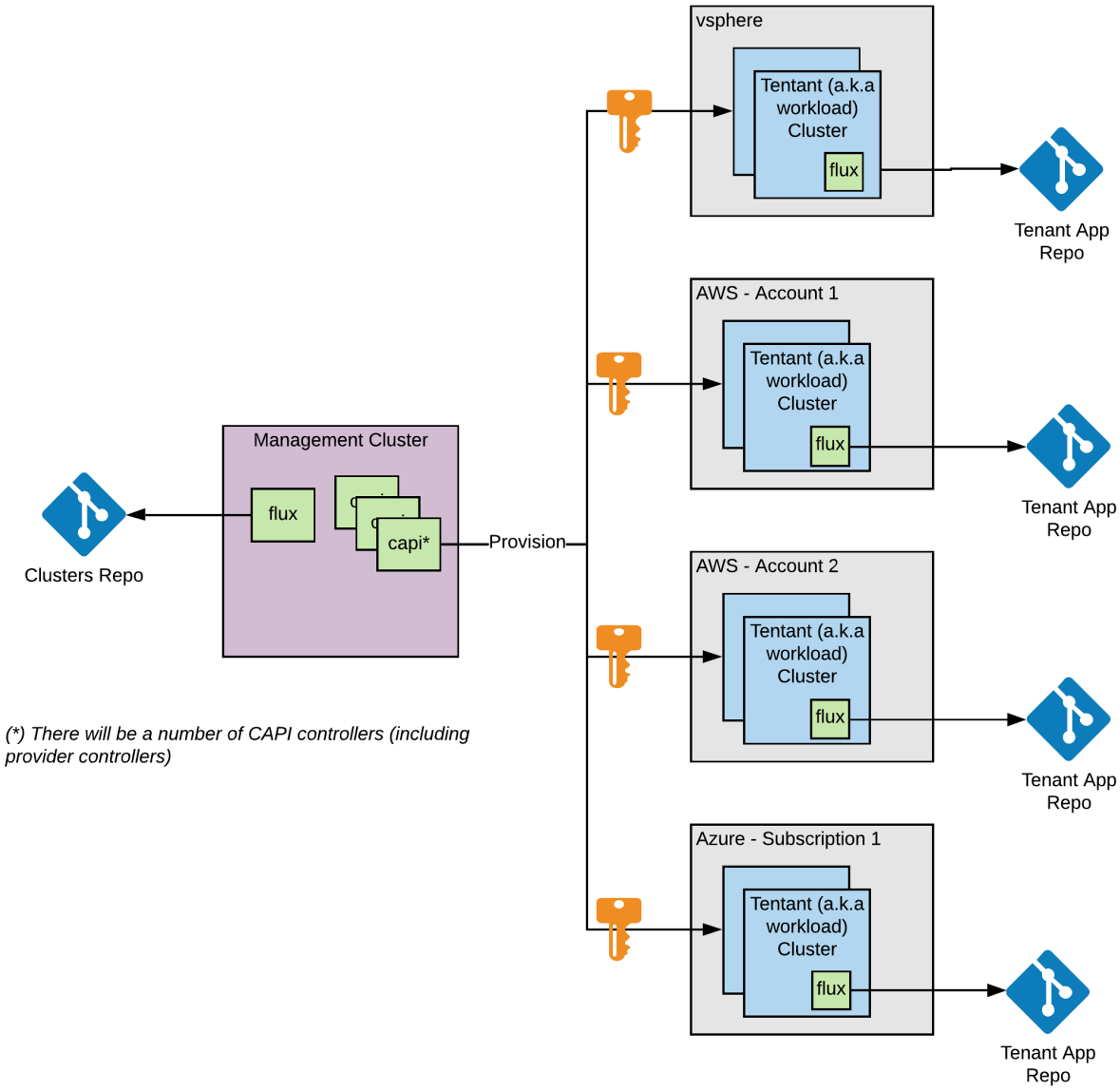


Рисунок 1.8 – Типова архітектура багатокластерної системи [19]

Компонентами даної архітектури є один management-кластер (кластер управління) та довільна кількість workload-кластерів (кластерів робочого навантаження).

Management-кластер служить для зберігання облікових даних для кожного із workload-кластерів та підтримка стабільного з'єднання з кожним із цих кластерів.

Основними задачами для workload-кластеру є:

- надання API для management-кластеру, щоб він міг здійснювати керування;

- підтримка gitops-агента, який буде здійснювати оновлення стану на основі інструкцій, отриманих від management-кластеру;
- підтримка власного репозиторія, в якому буде зберігатися поточна конфігурація.

Оскільки management-кластер зберігає облікові дані для кожного з workload-кластерів, він є вразливим місцем системи. Якщо зловмисник отримає доступ до management-кластеру, найбільш ймовірно, що він зможе отримати доступ і до відповідних workload- кластерів.

Також, певною вразливістю є те, що кожний із workload-кластерів керується одним і тим же management-кластером. У разі відмови management-кластеру відбудеться відмова усієї системи в цілому.

Окрім того, значним недоліком даної архітектури є необхідність встановлення та запуску на кожному з workload-кластерів gitops-агента, що призводить до збільшення витрат на обслуговування системи. Певних затрат вимагає і підтримка працездатності самого management-кластеру.

1.5.3 Синхронізація стану Kubernetes-кластеру з системою керування версіями

Основною задачею, яку покликаний вирішувати GitOps, є безперервна доставка та розгортання (CD) в Kubernetes-середовищі. Доставка (Continuous Delivery) в Kubernetes - це процес, який передбачає, що оператор, відповідальний за цей етап (системний адміністратор або спеціалізоване ПЗ) здійснює створення Kubernetes-об’єктів, які власне і будуть відповідати за роботу необхідного ПЗ. Розгортання в Kubernetes-кластері може відбуватися такими способами:

– розгортання здійснює Kubernetes-кластер. В цьому випадку використовуються нативні Kubernetes-об’єкти, які базовими;

– розгортання здійснюється завдяки спеціальному інструменту Kustomize. Це більш функціональний спосіб. У цьому випадку можна групувати Kubernetes-об’єкти. Kubernetes підтримує цей варіант без встановлення додаткових розширень та бібліотек;

– розгортання здійснюється завдяки спеціальному інструменту Helm. Це найбільш функціональний спосіб. Дозволяє групувати Kubernetes-об’єкти, застосовувати шаблонізацію, зберігати різні версії чартів - базових елементів Helm, що, як правило, складаються з опису метаданих чарту та файлів-шаблонів ПЗ, яке необхідно доставити та розгорнути. Для роботи Helm необхідно встановлення стороннього ПЗ, що являє собою виконуваний файл, який здійснює керування чартами (компіляцію, валідацію, завантаження з віддалених репозиторіїв).

Для вирішення задачі доставки та розгортання в Kubernetes-кластері найкраще використовувати комбінацію варіантів з Kustomize та нативними розгортаннями, які здійснює Kubernetes-кластер. Такий підхід дозволяє уникнути складнощів з встановленням та підтримкою роботи стороннього ПЗ. Також він є простим у розумінні та дозволяє вирішити будь-яку задачу з доставки та розгортання ПЗ в Kubernetes-кластері.

Отже, для задачі підвищення ефективності процесу автоматизації безперервної доставки та розгортання для багатокластерних систем найоптимальнішим є підхід GitOps. Серед основних переваг цього підходу, можна виділити наступні:

– зберігання кодової бази в одному місці;
– автоматизована доставка змін завдяки використанню gitops-агента;
– механізм PR, який дозволяє групувати зміни за задачами.

1.6 Типові архітектури програмного забезпечення

Для того, щоб реалізувати GitOps-підхід, вимагається наявність gitops-агенту, що являє собою спеціалізоване ПЗ, яке здійснює синхронізацію Git та стану Kubernetes-кластеру. Відповідно виникає проблема вибору архітектурного рішення для реалізації gitops-агенту.

Загалом існує декілька основних архітектурних рішень, які можна брати за основу при побудові програмного забезпечення, а саме: монолітна

(багатошарова), мікросервісна, event-driven, мікроядерна, space-based. На рисунку 1.9 зображена таблиця переваг та недоліків кожної з них.

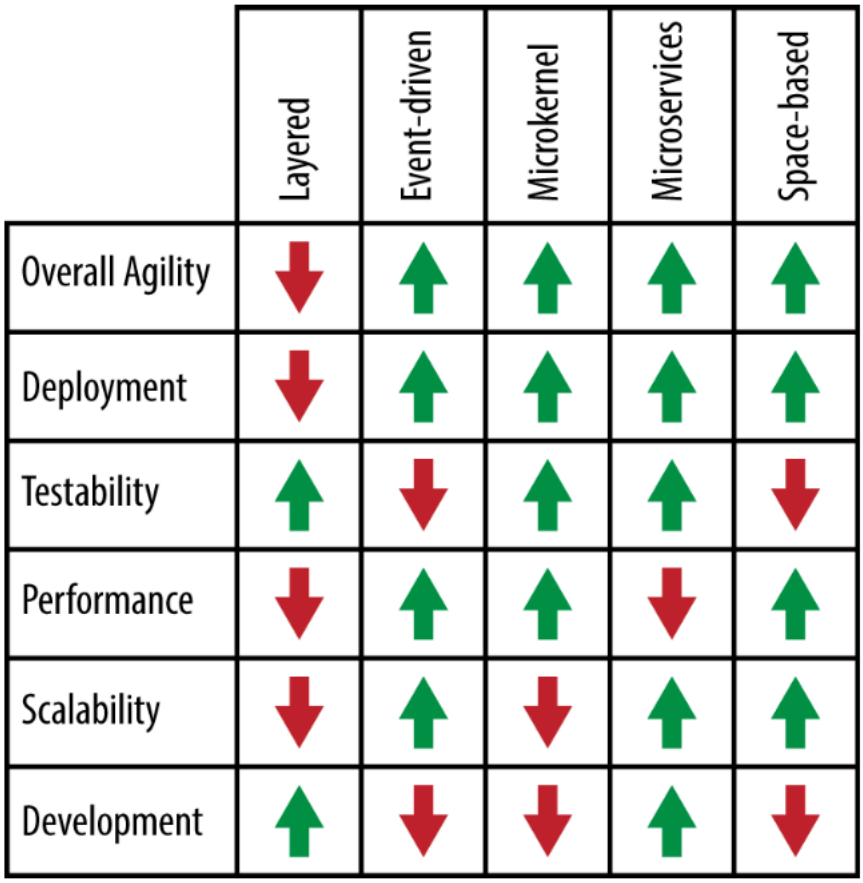


Рисунок 1.9 – Таблиця переваг та недоліків кожної з архітектур [20]

Монолітна архітектура до недавнього часу вважалася традиційним підходом. Основною її перевагою є те, що її розгортання відбувається доволі просто, оскільки весь вихідний код міститься в одному місці, написаний в одному стилі, використовуючи одні і ті ж технології. Проте, недоліками є те, що розробка такого ПЗ відбувається значно складніше, за рахунок того, що весь код знаходиться в одному місці, що може призводити до певного хаосу. Більше того, якщо над одним і тим ж монолітним застосунком працює декілька програмістів, то в них можуть виникати конфікти в версіях файлу, що в результаті призводить до сповільнення процесу розробки в цілому. Більше того, монолітний застосунок, за рахунок того, що є одним цілим несе в собі велику небезпеку: в разі відмови однієї частини застосунку ставиться під загрозу робота всього застосунку. Також для таких застосунків складно проводити поділ зон відповідальності [21].

Мікросервісна архітектура була вперше запропонована Мартіном Фовлером [22].

Основною її перевагою є те, що всі мікросервіси чітко відповідають за певний функціонал, тому відсутні перетини зон відповідальності. Також відсутня єдина кодова база, що дає змогу чітко розділити код по бізнес-процесах. Більше того, за рахунок того, що кожен мікросервіс є окремим, ізольованим, в контексті виконання, застосунком, то в нас відсутня проблема відмови всієї системи в разі відмови одного з мікросервісів.

Основними недоліками мікросервісної архітектури є те, що при великій кількості мікросервісів необхідно реалізовувати шини даних між ними, для їхнього спілкування, проте ця проблема легко вирішується за допомогою брокерів повідомлень.

В контексті задачі підвищення ефективності процесу автоматизації безперервної доставки та розгортання для багатокластерних систем оптимальнішою є мікросервісна архітектура. Вона дозволяє розбивати ПЗ на мікросервіси, які відповідають за певну частину функціональності. Це в свою чергу дозволяє ізолювати ці частини. Завдяки цьому ПЗ можна горизонтально масштабувати в залежності від потреб. Також проблема відсутня проблема відмови всієї системи в разі відмови одного з мікросервісів, оскільки вони ізольовані та слабкозв’язні.

1.7 Постановка задач

З метою підвищення ефективності процесу автоматизації CD для багатокластерних систем та для досягнення мети магістерської дисертації необхідно вирішити такі основні **завдання дослідження**:

– запропонувати архітектуру багатокластерної системи на базі мікросервісу синхронізації;

– реалізувати мікросервіс синхронізації;

– дослідити ефективність запропонованих рішень.

1.8 Висновок

Життєвий цикл розробки програмного забезпечення (ПЗ) є застосуванням інженерних практик, спрямованих на підтримку працездатності, якості та надійності ПЗ. Вагоме місце серед них займає DevOps. Існує п'ять підходів до реалізації DevOps-практик. В контексті задачі мінімізації часу необхідного на етап доставки та розгортання найкращим є підхід з безперервною доставкою та розгортанням (CD). Його переваги:

– висока частота розгортання, що дозволяє зменшити різницю між версією, яка розгортається та поточною;

– менший технічний борг, простіша розробка;

– простіше відслідковувати помилки на продуктивному середовищі, оскільки зміни доставляються невеликими порціями.

GitOps є одним із варіантів реалізації DevOps, який здійснює підтримку безперервної доставки та розгортання (CD). Основною задачею, яку покликаний вирішувати GitOps, є безперервна доставка та розгортання (CD) в Kubernetes-середовищі. Серед основних переваг цього підходу, можна виділити наступні:

– зберігання кодової бази в одному місці;

– автоматизована доставка змін завдяки використанню gitops-агента;

– механізм PR, який дозволяє групувати зміни за задачами.

Для реалізації GitOps-підходу необхідна VCS. В контексті задачі підвищення ефективності процесу автоматизації CD для багатокластерних систем найефективнішим рішенням є використання саме розподіленої VCS. Перевагами такого типу VCS є розподіленість, що збільшує безпеку за рахунок багатьох копій репозиторія, велика кількість сучасних інструментів та сторонніх сервісів для роботи.

Для того, щоб реалізувати GitOps-підхід, вимагається наявність gitops-агенту, що являє собою спеціалізоване ПЗ, яке здійснює синхронізацію Git та стану Kubernetes-кластеру. В контексті задачі розробки gitops-агенту найбільш оптимальною є мікросервісна архітектура. Вона дозволяє розбивати ПЗ на

мікросервіси, які відповідають за певну частину функціональності. Це в свою чергу дозволяє ізолювати ці частини. Завдяки цьому ПЗ можна горизонтально масштабувати в залежності від потреб. Також проблема відсутня проблема відмови всієї системи в разі відмови одного з мікросервісів, оскільки вони ізольовані та слабкозв’язні.

2 АРХІТЕКТУРА БАГАТОКЛАСТЕРНОЇ СИСТЕМИ НА БАЗІ МІКРОСЕРВІСУ СИНХРОНІЗАЦІЇ

2.1 Архітектура на основі GitOps агенту

Для кластерних систем безперервне розгортання (CD) відбувається з використанням GitOps-підходу, який грунтується на трьох базових елементах, а саме принципу IaS, єдиному джерелу істини у вигляді системи керування версіями Git та GitOps-агенті, який здійснює доставку внесенних змін в код на продуктивне середовище.

В багатокластерних системах також існує підхід безперервного розгортання в контексті GitOps-процесу. Реалізується він завдяки management-кластеру та workload-кластерам. Management-кластер зберігає облікові дані для доступу до всіх workload-кластерів. Також він має доступ до Git, тому в нього є змога відслідковувати будь-які зміни, внесені в репозиторій. Маючи облікові дані для доступу до workload-кластеру та зміни, внесені в VCS, management-кластер здійснює керування станом workload-кластерів. Відбувається це завдяки викликам до Kubernetes-API серверу, який є єдиною точкою доступу до Etcd.

Основні недоліки цього підходу:

– на кожному workload-кластері потребується встановлення GitOps-агенту, що вимагає його постійної підтримки та оновлення зі сторони команди DevOps-інженерів. Також наявні додаткові витрати на ресурси, необхідні для роботи GitOps-агенту;

– для роботи вимагається наявність management-кластеру, що є також великозатратним ресурсом, який також вимагає підтримки зі сторони команди системних адміністраторів та виділення фізичних потужностей для кластеру;

– облікові дані зберігаються на management-кластері. У разі, якщо зловмисник отримає доступ до management-кластеру, то всі workload-кластери, які керувалися ним, будуть скомпроментовані, що ставить під загрозу всю систему в цілому.

Враховуючи всі зазначені недоліки, було запропоновано власну архітектуру багатокластерної системи з безперервним розгортанням на основі мікросервісу синхронізації.

2.2 Архітектура багатокластерної системи з безперервним розгортанням на основі мікросервісу синхронізації

Архітектури багатокластерної системи, пропонується реалізувати архітектуру даної системи на основі мікросервісу синхронізації, що не вимагає жодної модифікації самих кластерів.

Запропонована архітектура багатокластерної системи з безперервним розгортанням на основі мікросервісу синхронізації зображена на рисунку 2.1.

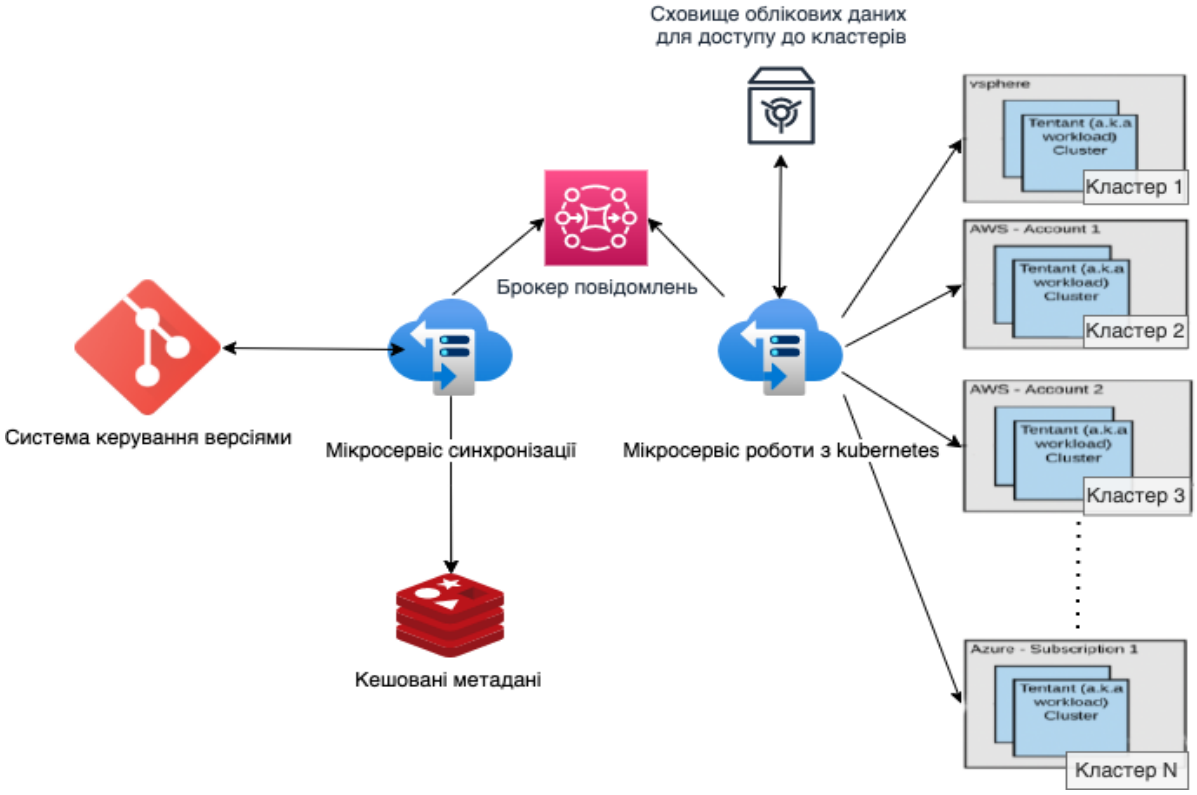


Рисунок 2.1 - Архітектура багатокластерної системи з безперервним розгортанням на основі мікросервісу синхронізації

Основними компонентами запропонованої архітектури, які відповідають за бізнес-логіку, є два мікросервіси: мікросервіс синхронізації та мікросервіс роботи з Kubernetes.

Основна задача мікросервісу синхронізації – отримувати зміни вихідного коду з системи керуваннями версіями (Git) та у разі їх наявності сформувати повідомлення на шину даних для того, щоб передати відповідне повідомлення в мікросервіс роботи з Kubernetes.

Мікросервіс роботи з Kubernetes служить для оновлення стану кожного з Kubernetes-кластерів. Зв’язок мікросервісу синхронізації та мікросервісу роботи з Kubernetes відбувається через брокер повідомлень.

Основною перевагою запропонованого архітектурного рішення є відсутність management-кластеру для керування workload-кластерами, що зменшує витрати на обслуговування системи. Окрім того, забезпечується вищий рівень безпеки, оскільки облікові дані, необхідні для авторизації до workload-кластерів, зберігаються на спеціалізованому сервері - так званому сховищі облікових даних.

Також важливою перевагою даного рішення є те, що на workload-кластерах не потрібно встановлювати додаткове ПЗ, на відміну від існуючого підходу, який вимагає встановлення gitops-агенту.

Розроблена мікросервісна архітектура є асинхронною, що дозволить уникнути блокування на мікросервісах.

2.2.1 Мікросервіс синхронізації

Основна задача мікросервісу синхронізації – отримувати зміни вихідного коду з системи керуваннями версій, визначати різницю змін та в разі, якщо присутні зміни, сформувати повідомлення на шину даних для того, щоб передати його в мікросервіс роботи з Kubernetes.

Для того, щоб працювати з віддаленим VCS-сервером, буде використовуватися GitHub API server, який є безкоштовним. За допомогою REST-підходу буде можливість отримувати всю необхідну інформацію з репозиторія. Для того, щоб мікросервіс синхронізації розумів, коли йому необхідно перевіряти зміни можна застосувати два підходи.

Перший підхід полягає в тому, щоб використовувати явно заданий період оновлень даних про репозиторій, наприклад, 30 секунд. Перевагами такого підходу є простота в реалізації. Проте мінусом є те, що буде затримка оновлень даних, що сповільнить процес безперервного розгортання.

Другий підхід полягає в тому, щоб використовувати сервіс Github Hooks. Цей механізм передбачає, що Github конфігурується так, щоб в разі оновлень репозиторія він міг надіслати запит на попередньо задану адресу мікросервісу синхронізації. Після того, як мікросервіс синхронізації отримає інформацію про те, що репозитарій був оновлений, він зможе опрацювати ці оновлення для подальшої передачі на шину даних.

Також мікросервіс синхронізації матиме кеш, в якому будуть зберігатися актуальні стани кожного з Kubernetes-кластерів, що дозволить швидко отримувати ці дані без прямого зв’язку з Kubernetes-API сервером.

Перевагами такого підходу є те, що буде відсутня затримка оновлень даних, оскільки відсутній період очікування. Проте мінусом є те, що для роботи цього підходу необхідно здійснювати додаткову конфігурацію Github та реалізовувати додатковий функціонал в мікросервісі синхронізації.

Діаграму цього процесу наведено на рисунку 2.2

Рисунок 2.2 – Використання сервісу Github hooks

2.2.2 Мікросервіс роботи з Kubernetes

Для того, щоб здійснювати оновлення стану workload-кластерів, необхідно мати інструмент для цього. Запропоноване рішення на основі мікросервісу роботи з Kubernetes є надійним та легким у використанні. Для того, щоб виконувати запити до Kubernetes-кластеру мікросервіс буде використовувати Kubernetes-API сервер. Зв’язок мікросервісу роботи з Git та мікросервісу роботи з Kubernetes

відбуватиметься через брокера повідомлень. Такий підхід дозволить уникнути блокувань роботи мікросервісів та зробити їхню взаємодію асинхронною. Більше того, такий підхід дозволить реалізувати функціонал повторної спроби відправки запиту на мікросервіс роботи з Kubernetes у разі неправильної попередньої обробки цього ж повідомлення.

Облікові дані, необхідні для авторизації до workload-кластерів будуть зберігатися на спеціалізованому сервері, так званому сховищі облікових даних. Це дозволить підвищити надійність системи за рахунок того, що облікові дані зберігаються не на мікросервісі, а в надійно захищеному місці. В разі, якщо зловмисник отримає доступ до мікросервісу роботи з Kubernetes, він не зможе отримати доступ до облікових даних, оскільки вони зберігаються в іншому, надійно захищеному місці.

2.3 Висновок

Проведено аналіз типової архітектури багатокластерної системи, який дозволив виявити такі її недоліки, як недостатній рівень безпеки облікових даних для доступу до кластерів, низька відмовостійкість, необхідність встановлення та запуску на кожному workload-кластері gitops-агента, що збільшує витрати на обслуговування системи.

Запропоновано удосконалену архітектуру багатокластерної системи на основі використання мікросервісу синхронізації стану workload-кластерів з системою керування версіями Git, що зменшує складність та вартість підтримки системи за рахунок відсутності необхідності адміністрування кожного з кластерів окремо, підвищує захист облікових даних для доступу до кластерів за рахунок використання сховища облікових даних, пришвидшує доставку програмного продукту за рахунок простоти процесу доставки змін та вбудованого кешування.

3 РОЗРОБКА МІКРОСЕРВІСУ СИНХРОНІЗАЦІЇ

3.1 Огляд середовища для розробки мікросервісів

ПЗ, яке створюється, повинне бути розроблене з перспективою на його подальшу підтримку. Відповідно, для розробки повинна використовуватися високорівнева мова програмування. Це значно спростить розробку системи, її підтримку, зменшить час та вартість розробки.

Найпопулярнішими високорівневими мови програмування є такі:

– Java. Основна задача, яка вирішується завдяки Java є розробка серверів, які здійснюють обробку клієнтських запитів. Великою перевагою цієї мови є її кроссплатформенність, що дозволяє реалізувати підхід WORA (Write once, run anywhere) [23]. На сьогодні мова потужно розвивається, активно випускаються нові версії;

– C#. Розроблялася під впливом вже існуючих на той час мов C++, Java, Object Pascal. Наразі мова вважається такою, що має низький поріг входження. Використовується для розробки десктопних застосунків та веб-застосунків [24];

– Python. Мова є інтерпретованою використовується для аналізу даних, написання веб-застосунків. Дозволяє писати зручні скрипти, завдяки тому, що проектувалася для максимальної зручності в розробці та читабельності програмного коду [25];

– JavaScript. JavaScript є кроссплатформною скриптовою мовою програмування. Найчастіше JavaScript використовується для створення web-сторінок (написання їх сценаріїв), створення односторінкових web-додатків, програмного доступу до об'єктів додатків тощо. [26].

Серед розглянутих мов програмування для написання ПЗ вирішено використати мову Java, як одну із найбільш популярних. Основними її перевагами є кроссплатформенність, строга типізація, велика кількість сторонніх бібліотек завдяки потужній спільноті розробників, зручні IDE для написання коду, що пришвидшують розробку ПЗ.

В контексті вирішення задачі проектування архітектури багатокластерної системи з безперервним розгортанням на основі мікросервісу синхронізації необхідно визначити в якому середовищі та яким чином будуть розгорнуті кластери.

Для імплементації кластеру використовується Kubernetes. Як було зазначено в першому розділі, для його роботи необхідні хости (сервери), на яких буде запущене ПЗ Kubernetes та його модулі. В середовищі для розробки є сенс не витрачати багато ресурсів, тому master-ноду та worker-ноду доцільно об’єднати в одну. Класична реалізація Kubernetes не дозволяє цього зробити, проте є такі інструменти, як Minikube та Kind, які надають реалізацію кластеру, який можна використати на етапі розробки ПЗ. Завдяки зменшенню розміру виконуваного файлу та адаптації Minikube- та Kind-реалізацій під виконання master-ноди та worker-ноди на одному хості, вдається досягти необхідної зручності в розробці, використовуючи максимально-мінімальну кількість ресурсів. Реалізації Kubernetes-кластеру у вигляді Minikube та Kind мають практично повну підтримку функціональності Kubernetes-кластеру. Оскільки для розробки ПЗ необхідна багатокластерна система, то в якості середовища для розробки використаємо обидва інструменти. Відповідно отримаємо один Minikube-кластер та один Kind-кластер.

Оскільки створене ПЗ буде кроссплатформенним, то для розробки можна використати будь-яку ОС:

– Linux (Ubuntu);
– Windows (Windows 10/11);
– MacOS (BigSure/Monterey).

В якості основної ОС для розробки використаємо MacOS BigSure, оскільки вона є сучасною unix-подібною ОС. В ній присутня підтримка всіх необхідних бібліотек та інструментів для розробки.

Для того, щоб ПЗ могло взаємодіяти з Kubernetes-кластерами, необхідне встановлення стороннього ПЗ, а саме Kubectl – консольного клієнта для командної взаємодії з Kubernetes-кластером. Детальний опис роботи Kubectl був

наведений в першому розділі даної магістерської роботи. Існує імплементація Kubectl для MacOS BigSure, яку можна завантажити з ресурсу GitHub в розділі репозиторію розробників.

Також необхідне встановлення середовища виконання контейнерів. Одним із найпопулярніших із таких середовищ є Docker. Він швидкий, зручний в використанні, має велику спільноту та використовується в продуктивних системах. Також є десктопний застосунок Docker Desktop, який дозволяє керувати контейнерами.

3.3 Розробка вимог

Вимоги до мікросервісів, що розробляються, можна поділити на функціональні та нефункціональні.

3.3.1 Вимоги в цілому

Базові функціональні вимоги до ПЗ:

– розроблене ПЗ повинно здійснювати синхронізацію змін, внесених у Git, з станом Kubernetes-кластеру;

– розроблене ПЗ повинне бути поділене на мікросервіси, які взаємодіють між собою через брокер повідомлень;

– розроблене ПЗ повинне вміти взаємодіяти з Kubernetes-кластером (тут розуміється здатність ПЗ створювати, редагувати, видаляти Kubernetes-об’єкти);

– всі облікові дані повинні зберігатися в надійному сховищі даних.

Базові нефункціональні вимоги до ПЗ:

– розроблене ПЗ повинне бути кроссплатформенним;

– розроблене ПЗ повинне підтримувати горизонтальне масштабування;

– один цикл роботи ПЗ не повинен бути довшим ніж 30 секунд, при використанні такої конфігурації: процесор 2,6 GHz 6-ядерний Intel Core i7, RAM 6 ГБ 2667 MHz DDR4, жорсткий диск Macintosh HD.

3.3.2 Вимоги до мікросервісу синхронізації

Git-репозиторій – містить папку .git, в якій зберігається вся необхідна інформація для Git.

Мікросервіс синхронізації повинен:

– працювати з Git-репозиторієм, як локальним так і віддаленим. Вміти виконати Git-команди pull, push, fetch для віддаленого репозиторію;

– при отриманні змін з віддаленого Git-репозиторію вміти визначити тип зміни та повний шлях до файлу, який було змінено;

– отримані зміни з віддаленого Git-репозиторію повинні бути закешовані в оперативній пам’яті;

– формувати повідомлення про наявність змін для мікросервісу роботи з Kubernetes в якому повинен бути вказаний шлях до зміненого файлу, вміст файлу та тип зміни;

– повідомлення про наявність змін для мікросервісу роботи з Kubernetes повинне відправлятися через брокер повідомлень;

– серед типів змін в файлі повинні бути такі: створення, редагування та видалення;

– в разі, якщо типом зміни є видалення, то вміст файлу є необов’язковим;

– мати можливість роботи за розкладом, наприклад, за cron-виразами.

3.3.3 Вимоги до мікросервісу роботи з Kubernetes

Мікросервіс роботи з Kubernetes повинен:

– отримувати повідомлення про зміни від мікросервісу синхронізації через брокер повідомлень та здійснити його парсинг, визначивши всю інформацію, яка міститься в повідомленні;

– отримувати облікові дані для доступу до кластерів з системи сховища даних;

– вміти надсилати команди декільком кластерам одночасно;

– вміти взаємодіяти з інструментом Kubectl для того, щоб здійснювати доставку змін в Kubernetes-кластер.

3.4 Компоненти розробленого програмного забезпечення

Архітектуру системи було наведено на рисунку 2.1. ПЗ, яке розробляється складається з двох мікросервісів: синхронізації (synchronization) та роботи з Kubernetes (kubernetes).

На рисунку 3.1 представлено діаграму послідовності для даного ПЗ.

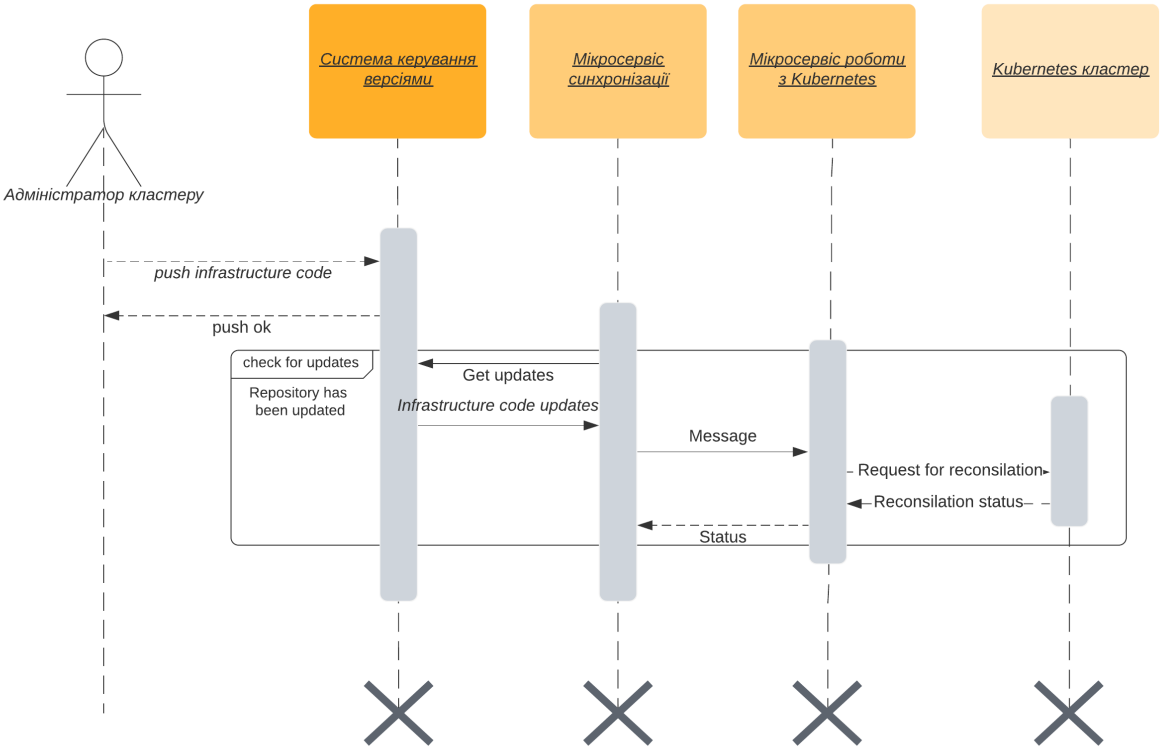


Рисунок 3.1 - Діаграма послідовності для мікросервісів

Також є допоміжні модулі libmq, libgit.

3.4.1 Сторонні компоненти

Відповідно із рисунком 2.1, архітектура мікросервісів передбачає використання таких компонент: VCS, кеш, брокер повідомлень, сховище облікових даних.

В якості VCS в підході GitOps використовується Git. Для розробки мікросервісів використовуватиметься версія Git 2.33.1. Для того, щоб розроблені мікросервіси могли взаємодіяти з Git, необхідна бібліотека для роботи з Git в Java. Найбільш зручною із таких бібліотек, яка вирішує дану задачу, є JGit. Вона підтримує всю функціональність нативного інструменту Git, дозволяє працювати з віддаленими репозиторіями, налаштовувати ручне ssh-з'єднання. Для розробки мікросервісів використовуватиметься версія JGit 5.13.0.

Кеш може використовуватися як вбудований в мікросервіси, так і сторонній. В якості вбудованого кешу можна використати одну із бібліотек фреймворку Spring, а саме Spring Cache. Це проста, проте ефективна реалізація In-memory кешу. Вона цілком підходить для того, щоб використовуватися в якості основного кешу.

Для того, щоб мікросервіси могли спілкуватися між собою, необхідний брокер повідомленнь. Одним із найпопулярніших брокерів повідомлень є RabbitMQ - надійний, швидкий, не ресурсозатратний брокер повідомлень, який підтримується Java фреймворком Spring.

Для того, щоб зберігати облікові дані доступу до кластерів необхідно мати надійне сховище облікових даних. Одним із найпопулярніших та найзручніших реалізацій такого сховища є Vault. Дане сховище підтримується фреймворком Spring та є сертифікованим інструментом, який може використовуватися в продуктивних системах.

3.4.2 Мікросервіс синхронізації

Основні задачі розробленого мікросервісу синхронізації:

– отримування змін вихідного коду з VCS;

– визначення типу змін;

– формування повідомлень зі змінами в VCS;

– відправка повідомлення зі змінами в VCS в брокер повідомлень для того, щоб мікросервіс роботи з Kubernetes міг його опрацювати.

Архітектура мікросервісу синхронізації базується на основі Java фреймворку Spring. Основна задача цього фреймфорку – це надання програмного каркасу з відкритим кодом та інверсії управління для платформи Java. Інверсія управління дозволяє зменшити залежність між класами.

Для того, щоб ініціалізувати проект, необхідно використати Spring Boot. Це спеціальна екосистема, яка дозволяє отримати всі необхідні залежності, версії яких не будуть конфліктувати між собою. Також вона надає значення певних змінних та об’єктів за замовчуванням. Це дозволяє писати код швидко, не поглиблюючись в деталі реалізації. Приклад ініціалізації Spring Boot проекту на рисунку 3.2

```
package org.romanchi.gitops.synchronization;

import org.eclipse.jgit.api.errors.GitAPIException;
import org.springframework.amqp.rabbit.annotation.EnableRabbit;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cache.annotation.EnableCaching;
import org.springframework.context.ApplicationContext;

import java.io.IOException;

@SpringBootApplication
@EnableCaching
@EnableRabbit
public class Synchronization {
    public static void main(String[] args) throws IOException, GitAPIException {
        ApplicationContext context = SpringApplication.run(Synchronization.class,
args);
    }
}
```

Рисунок 3.2 - Приклад ініціалізації Spring Boot проекту

Як можна бачити з рисунку, використовуються три Java-анотації:

– *SpringBootApplication*. Створює *ApplicationContext*, який відповідає за інверсію управління, по-суті, основа Spring Framework;

– *EnableCaching,* Включає механізм кешування, створюючи in-memory *CacheManager;*

– *EnableRabbit*. Включає механізм роботи з брокером повідомлень RabbitMQ.

Для роботи з Git використовується бібліотека JGit. На рисунку 3.3 наведено метод для отримання змін з віддаленого репозитарія, визначення типів змін та шляхів до файлів.

```
    public List<DiffEntry> updates() throws IOException, GitAPIException {
  Iterable<RevCommit> logs = git.log()
      .call();
  ObjectId localHead = logs.iterator().next().getId();

  git.fetch().setTransportConfigCallback(transportConfigCallback).call();
  logs = git.log()
      .add(repository.resolve("remotes/origin/main"))
      .call();
  ObjectId remoteHead = logs.iterator().next().getId();

  System.out.printf("Local head=[%s], remote head=[%s]\n", localHead,
remoteHead);
  git.pull().setTransportConfigCallback(transportConfigCallback).call();

  final List<DiffEntry> diffs = git.diff()
      .setOldTree(prepareTreeParser(repository, localHead.name()))
      .setNewTree(prepareTreeParser(repository, remoteHead.name()))
      .call();
  System.out.println("Found: " + diffs.size() + " differences");
  for (DiffEntry diff : diffs) {
    System.out.println(diff.getChangeType() + ": " +
        (diff.getOldPath().equals(diff.getNewPath()) ? diff.getNewPath() :
diff.getOldPath() + " -> " + diff.getNewPath()));
  }
  return diffs;
}
```

Рисунок 3.3 - Метод для отримання змін з віддаленого репозиторія, визначення типів змін та шляхів до файлів.

На рисунку 3.4 наведено UML-діаграму класів для мікросервісу синхронізації.

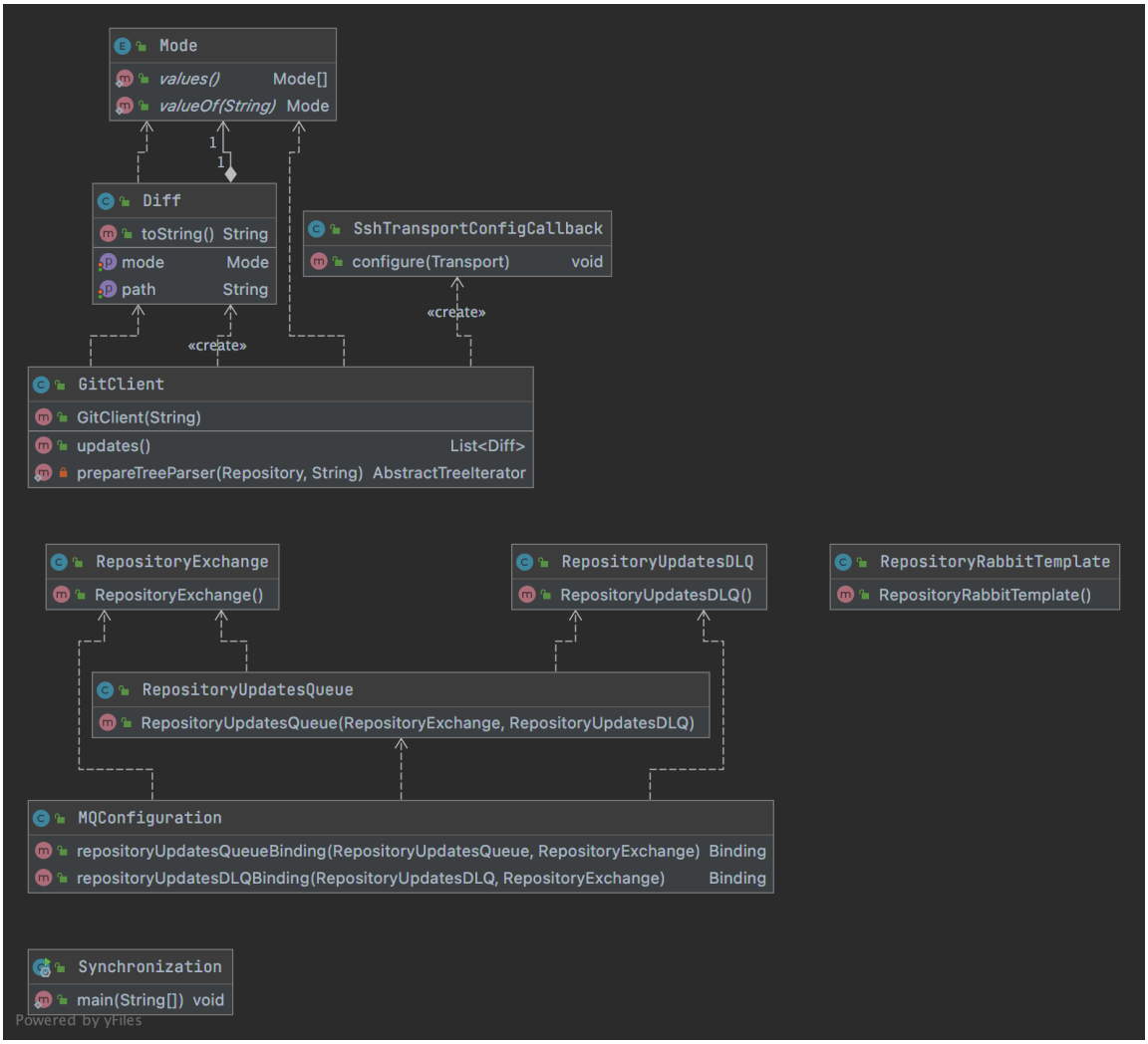


Рисунок 3.4 – UML-діаграма класів для мікросервісу синхронізації

Для того, щоб відправити повідомлення на брокер повідомлень RabbitMQ, використовується *RepositoryRabbitTemplate.*

3.4.3 Мікросервіс роботи з Kubernetes

Основна задача мікросервісу роботи з Kubernetes - отримувати повідомлення про зміни в Git-репозиторії. Ці повідомлення створює та відправляє через брокер повідомлень RabbitMQ мікросервіс синхронізації.

Архітектура мікросервісу роботи з Kubernetes, як і мікросервісу синхронізації, базується на основі фреймворку Spring. Для роботи з RabbitMQ використовується модуль RabbitMQ. Для того, щоб отримати повідомлення з брокера повідомлень, використовується Java-анотація

@RabbitListener(queues = “#{назва черги}”).

Вона дозволяє визначити Java-метод, який буде відповідальним за обробку повідомлень з конкретних черг, що задаються в параметрі анотації *queues*.

На рисунку 3.5 наведено обробник повідомлень, які надходять від мікросервісу синхронізації.

```
        @RabbitListener(queues = "#{repositoryUpdatesQueue}")
public void listener(Message message) throws Exception {
   List<Diff> diffs = mqJsonMapper.readValue(new String(message.getBody(),
StandardCharsets.UTF_8), new TypeReference<>() {});
   List<Diff> deleteDiffs = diffs.stream()
         .filter(it -> Mode.DELETE.equals(it.getChangeType()))
         .collect(Collectors.toList());
   List<Diff> nonDeleteDiffs = diffs.stream()
         .filter(it -> !Mode.DELETE.equals(it.getChangeType()))
         .collect(Collectors.toList());
   clusters.forEach(cluster -> {
      KubectlClient.getContextByName(cluster).ifPresent(KubectlClient::useContext);
      nonDeleteDiffs.forEach(diff -> {
         try{
            Arrays.stream(KubectlClient.apply(repoBaseFolder +
diff.getWorkPath())).forEachOrdered(logger::info);
         }catch (Exception ex){
            logger.info(ex.getMessage(), ex);
         }
      });
      deleteDiffs.forEach(diff -> {
         try{
            Arrays.stream(KubectlClient.delete(backupRepoBaseFolder + "/" +
diff.getWorkPath())).forEachOrdered(logger::info);
         }catch (Exception ex){
            logger.info(ex.getMessage(), ex);
         }
      });
   });

   backupRepoGitClient.pull();
}
```

Рисунок 3.5 – Обробник для повідомлень від мікросервісу синхронізації

Основна ідея методу оновлення стану Kubernetes-кластеру полягає в тому, що для зміни або створення нового Kubernetes-об'єкту використовується команда *kubectl apply*, а для видалення *kubectl delete*. Для того, щоб мікросервіс міг виконувати ці команди, необхідно здійснювати системні виклики з Java до виконуваного файлу *kubectl*. Для цього в Java слугує клас java.lang.Process. Об'єкти цього класу є дескрипторами системних процесів. Відповідно є три методи для того, щоб здійснювати I/O (Input/Output) операції з вхідними та вихідними потоками, а саме: getOutputStream(), getInputStream(), getErrorStream().

Для роботи з *kubectl* було створено спеціальний клас KubectlClient, який інкапсує всю роботу з ним *kubectl*. На рисунку 3.6 наведено метод-обгортку, який дозволяє виконати команди *kubectl* та повертає результат виконання.

```
        private static String [] execCommand(String[] commands) throws IOException
{
   Process process = Runtime.getRuntime().exec(commands);
   boolean error = process.getErrorStream().available() > 0;
   if(error){
      String command = String.join(" ", commands);
      throw new RuntimeException(String.format("Error during call [%s]",
command));
   }
   BufferedInputStream bufferedInputStream = new
BufferedInputStream(process.getInputStream());
   return new String(bufferedInputStream.readAllBytes(),
StandardCharsets.UTF_8).trim().split("\n");
}
```

Рисунок 3.6 - Метод-обгортка для виконань команди *kubectl*

На рисунку 3.7 наведено UML-діаграму класів для мікросервісу роботи з Kubernetes.

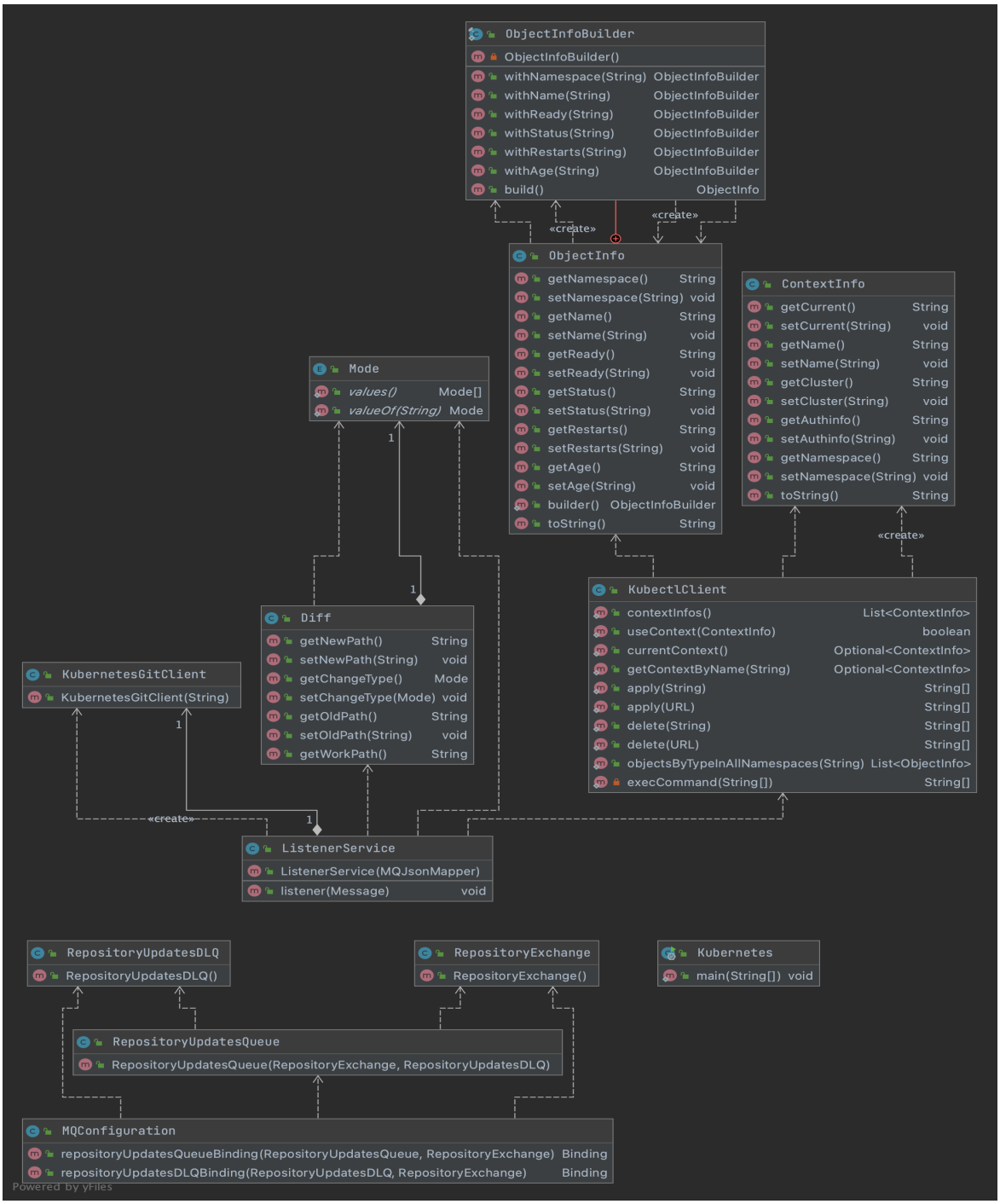


Рисунок 3.7 – UML-діаграму класів для мікросервісу роботи з Kubernetes

3.4.4 Модулі libmq, libgit

Для того, щоб максимально уникнути дублювання коду, спільний функціонал мікросервісів було винесено у два модулі: libmq, libgit.

Libmq містить класи, які необхідні для роботи з брокером повідомлень RabbitMQ, серед них:

– org.romanchi.gitops.libmq.BaseMQConnectionFactory – відповідає за з'єднання з кластером RabbitMQ, для його роботи необхідні 4 параметри: *mqHost(адреса серверу), mqPort(порт RabbitMQ кластеру), mqUser(ім'я користувача), mqPassword (пароль користувача);*

– org.romanchi.gitops.libmq.BaseDirectExchange – дозволяє успадкувати та створити *exchange* з назвою gitops.<exchange-name>.exchange;

– org.romanchi.gitops.libmq.BaseDeadLetterQueue – дозволяє успадкувати та створити *queue* (чергу) з назвою gitops.<exchange-name>.dlq, які в подальшому використовуватимуться для того, щоб зберігати повідомлення, при обробці яких, виникла помилка;

– org.romanchi.gitops.libmq.BaseQueue - дозволяє успадкувати та створити *queue* (чергу) з назвою gitops.<exchange-name>.queue, які в подальшому використовуватимуться для того, щоб зберігати повідомлення, які повинні обробитися.

Такий підхід дозволяє легко створювати черги з підтримкою dead letter queue (черга мертвих повідомлень або DLQ). DLQ містить всі повідомлення з відповідної звичайної черги при обробці яких виникла помилка. Це в свою чергу, підвищує конзистентність даних в системі, оскільки жодне повідомлення, яке потрапило в брокер повідомлень, не буде втраченим.

Libgit містить класи, необхідні для роботи з віддаленими та локальними репозитаріями Git. Основим класом, яких необхідний для роботи з Git-репозиторіями, є BaseGitClient. Він дозволяє успадкувати та розширити свій базовий функціонал, яки передбачає виконання команд *git pull* та *git fetch.*

3.5 Висновок

Для вирішення задачі написання мікросервісу синхронізації, який виконуватиме роль gitops-агенту було вирішено використовувати мову програмування Java. Основними її перевагами є кроссплатформенність, строга типізація, велика кількість сторонніх бібліотек завдяки потужній спільноті

розробників, зручні IDE для написання коду, що пришвидшують розробку ПЗ. Також при розробці використано такі технології та бібліотеки: RabbitMQ, Spring, Vault, kubectl, JGit. В результаті було отримано бібліотеку для роботи з git libmq, для роботи з RabbitMQ libmq та два мікросервіси: синхронізації та роботи з Kubernetes. Написаний код відповідає всім сучасним стандартам розробки.

4 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ

Для того, щоб перевірити ефективність спроектованої архітектури багатокластерної системи на базі мікросервісу синхронізації, необхідно її порівняти з уже існуючими типовими архітектурними рішеннями. Оскільки, однією із найпопулярніших реалізацій GitOps-агентів є Flux, то використовуватимемо його для порівняння саме його.

Оскільки, як було зазначено у третьому розділі, для роботи багатокластерної системи доцільно використати minikube та kind, розгорнемо для тестування саме ці два кластери.

4.1 Налаштування середовища

До того, як перейти до встановлення необхідних інструментів та підняття кластерів, необхідно впевнитися у тому, що на робочому комп'ютері встановлене та коректно працює середовище для виконання контейнерів. Можуть використовуватися такі середовища, як Docker, containerd, CRI-O. Для тестування використовувався Docker версії 20.10.5, build 55c4c88.

Для того, щоб підняти кластер minikube необхідно встановити утиліту minikube. Зробити це можна послідовністю команд *curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-amd64&& chmod +x minikube*; *sudo mv minikube /usr/local/bin.*

Перевірити версію встановленого minikube можна командою

minikube version.

Для тестування використовувалася версія minikube version: v1.23.2 (commit: 0a0ad764652082477c00d51d2475284b5d39ceed). Для того, щоб підняти кластер minikube необхідно виконати команду

minikube start.

В результаті повинне початися завантаження docker-образу для найновішої версії Kubernetes та його запуск. Для minikube-кластеру було використано версію 1.22.2. На рисунку 4.1 зображено результат виконання команди *minikube start.*

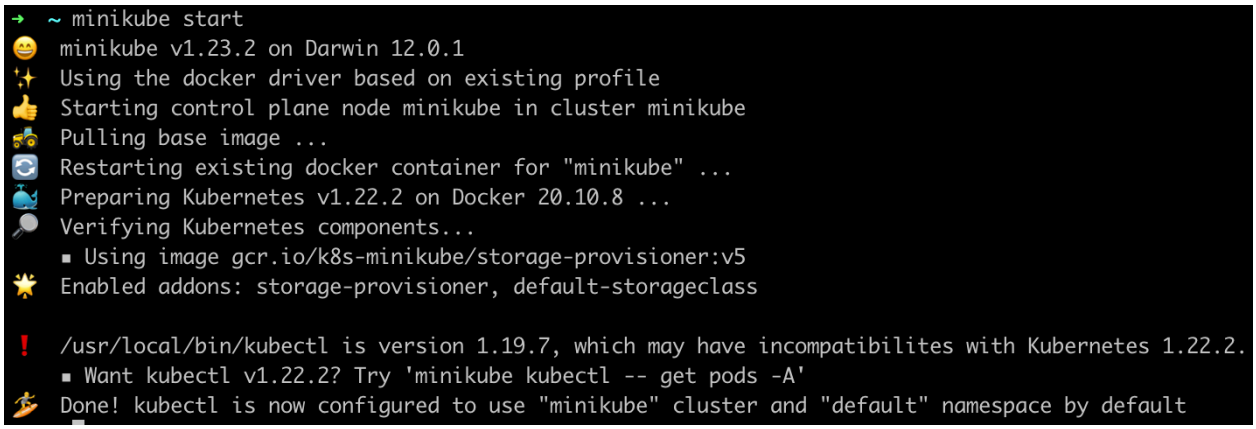


Рисунок 4.1 - Результат виконання команди minikube start

Для того, щоб підняти кластер kind необхідно встановити утиліту kind. Зробити це можна послідовністю команд: *curl -Lo ./kind https://kind.sigs.k8s.io/dl/v0.11.1/kind-darwin-amd64*; *chmod +x ./kind*; *mv ./kind /some-dir-in-your-PATH/kind.*

Перевірити версію встановленого kind можна командою *kind version.* Для тестування використовувалася версія kind version: v0.10.0 go1.15.7 darwin/amd64. Для того, щоб підняти кластер kind необхідно виконати команду

kind create cluster.

Для kind-кластеру було використано версію 1.20.2. На рисунку 4.2 зображено результат виконання команди kind create cluster.

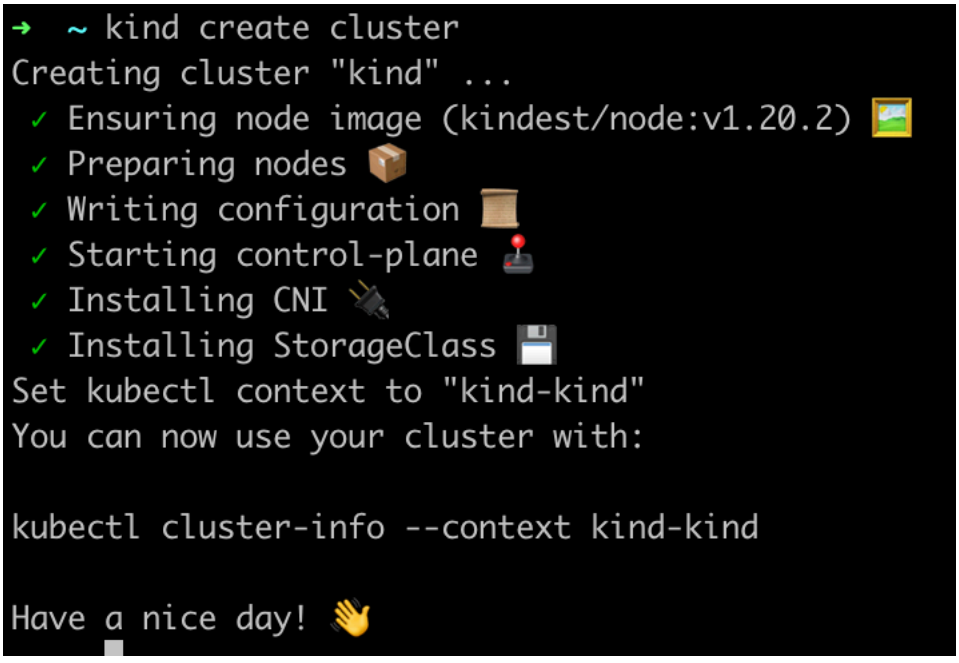


Рисунок 4.2 - Результат виконання команди kind create cluster

4.2 Тестування типової архітектури

Для того, щоб встановити flux, необхідно встановити утиліту flux. Для цього необхідно виконати команду

curl -s https://fluxcd.io/install.sh | sudo bash.

Щоб встановити зв'язок між віддаленим репозитарієм та станом Kubernetes-кластеру, використовуючи flux, необхідно виконати команду

flux bootstrap git --url=https://github.com/<repository> --username=<uname> --password=<password> --token-auth=true --path=clusters/my-cluster,

де *repository* – назва репозиторію, *uname, password* – ім'я та пароль, користувача, який є власником GitHub-профілю. Змінна *path* містить в собі шлях в репозиторії, який буде моніторитися flux-застосунком. Це дозволяє зберігати в одному репозитарії конфігурації для різних кластерів. Перед виконанням цієї команди необхідно створити приватний/публічний GitHub-репозиторій.

Також необхідно визначити змінну оточення GITHUB_TOKEN, перед цим згенерувавши токен доступу до GitHub API з правами на редагування, створення та видалення репозиторів. На рисунку 4.3 зображено результат виконання команди *flux bootstrap*

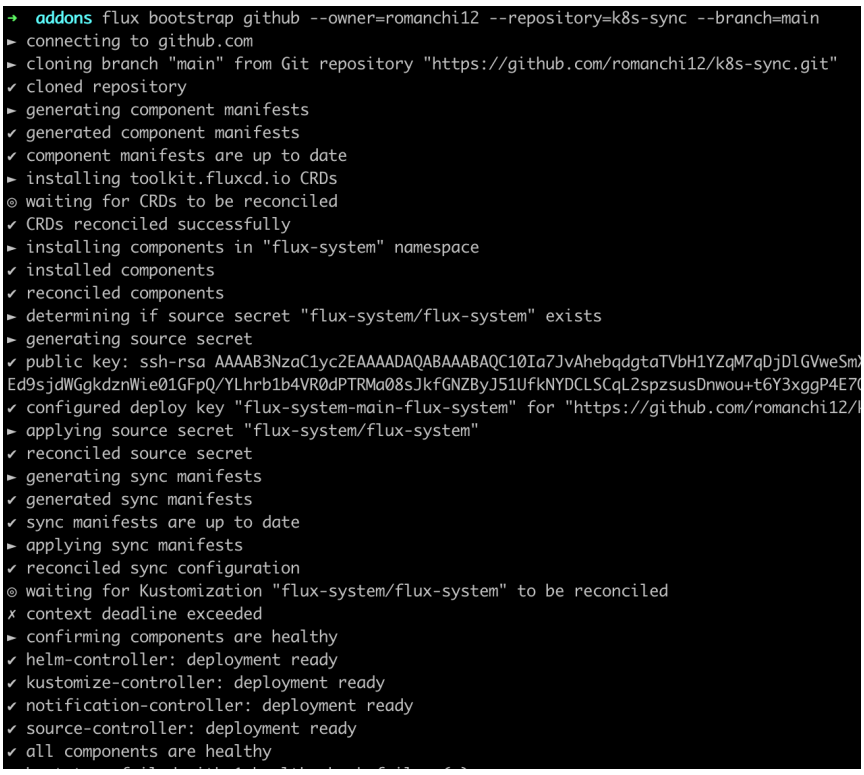


Рисунок 4.3 - Результат виконання команди *flux bootstrap*

Після визначення токена можна протестувати процес безперервної доставки та розгортання (CD). Для цього необхідно закомітити файл *deployment.yaml* з тестовим Kubernetes-об'єктом у віддалений репозиторій Git. Суть файлу *deployment.yaml* в тому, що він дозволяє розгорнути тестовий *nginx deployment* та зібрати необхідні метрики. Вміст файлу наведено на рисунку 4.4.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-minikube-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

Рисунок 4.4 - Вміст файлу deployment.yaml

Після цього в логах kustomize-controller можемо спостерігати повідомлення про успішне завершення *reconciliation* (застосування змін):

{"level":"info","ts":"2021-11-23T22:56:54.715Z","logger":"controller.kustomization","msg":"Reconciliation finished in 4.374977014s, next run in 1m0s","reconciler group":"kustomize.toolkit.fluxcd.io","reconciler kind":"Kustomization","name":"app-minikube-deployment","namespace":"default","revision":"master/0247403d6e81d18e6ab615113335e4cd89c70c80 "}

3.3 Тестування архітектури на базі мікросервісу синхронізації

Для того, щоб протестувати архітектурне рішення на базі мікросервісу синхронізації потрібне середовище для виконання контейнерів. Для тестування було використано Docker версії 20.10.5, build 55c4c88. Для того, щоб запустити необхідні мікросервіси та сторонні сервіси необхідно використати команду

docker-compose up.

На вхід даної команди подається файл *infrastructure.yaml*, в якому описуються всі контейнери, мережі, віртуальні диски, які необхідно створити в Docker. На рисунку 4.5 показано вміст файлу *infrastructure.yaml*

```
        version: "3.9"
services:
  rabbitmq:
    image: rabbitmq:management
    networks:
      - rabbit
    ports:
      - "5672:5672"
      - "15672:15672"
  syncronization:
    image: romanchi-syncronization:1.0
    networks:
      - rabbit
  kubernetes:
    image: romanchi-kubernetes:1.0
    networks:
      - rabbit
networks:
  rabbit:
```

Рисунок 4.5 – Вміст файлу infrastructure.yaml

Після виконання команди *docker-compose up* запустяться необхідні контейнери.

Для того, щоб протестувати процес безперервної доставки та розгортання необхідно додати файл *deployment.yaml* з тестовим Kubernetes-об’єктом в віддалений Git-репозиторій. Вміст файлу аналогічний тому, що використовувався

для типової архітектури. Після цього можемо бачити в логах контейнерів повідомлення

deployment.apps/nginx-deployment created. Time elapsed: 1945ms

4.4 Порівняння отриманих результатів

Було проведено ряд тестових випробувань та зібрано велику кількість метрик завдяки інструментам Prometheus, Grafana. Серед основних важливих метрик були такі, як середня завантаженість центрального процесора (CPU) по кожній кластер-ноді, середня завантаженість оперативної пам'яті (RAM) по кожній ноді, час необхідний для застосування змін.

Отримані результати тестування для середньої завантаженості центрального процесора наведено в таблиці 4.1.

Таблиця 4.1. Середня завантаженість CPU

Конфігурація	Flux	Рішення на базі мікросервісу синхронізації
Minikube	15.739008606311083	9.340182800720376
Minikube + Kind	17.872439789178397	8.038692461641368

Для вимірювання використовувалася метрика, результатом якої є завантаженість CPU у відсотках в середньому за хвилину. (формула 4.1)

$$sum\ (rate\ (container\_cpu\_usage\_seconds\_total\{id="/"\}[1m])) \ /\ sum\ (machine\_cpu\_cores) * 100, \quad (4.1)$$

Як можемо бачити, запропоноване архітектурне рішення на основі мікросервісу синхронізації показало кращі результати.

Результати тестування для середньої завантаженості RAM наведено в таблиці 4.2.

Таблиця 4.2 Середня завантаженість RAM (У відсотках)

Конфігурація	Flux	Рішення на базі мікросервісу синхронізації
Minikube	46.47708034461492	41.26005978747474
Minikube + Kind	49.48306246229581	41.894027206558874

Для вимірювання використовувалася метрика, результатом якої є завантаженість RAM у відсотках в середньому за хвилину. (формула 4.2)

$$100 * (1 - ((avg\_over\_time(node\_memory\_MemFree\_bytes[1m]) + avg\_over\_time(node\_memory\_Cached\_bytes[1m]) + avg\_over\_time(node\_memory\_Buffers\_bytes[1m])) / avg\_over\_time(node\_memory\_MemTotal\_bytes[1m]))), \quad (4.2)$$

Отримані результати тестування для середньої тривалості *reconciliation* (застосування змін) наведено в таблиці 4.3.

Таблиця 4.3 Середня тривалість reconciliation (застосування змін)

Конфігурація	Flux	Рішення на базі мікросервісу синхронізації
Minikube	*4374ms*	*1945ms*
Minikube + Kind	*7367ms*	3824*m*s

Для вимірювань у flux-тестах використовувалися дані логування kustomize-controller. Для вимірювань у запропонованому архітектурному рішенні використовувалася різниця двох викликів Java-методу *System.nanoTime()*.

4.5 Висновок

Результати тестування показали, що розроблене архітектурне рішення є швидшим за рахунок того, що воно є асинхронним, використовує значно менше RAM та CPU. Також воно є повністю незалежним від Kubernetes-кластерів, що дозволяє не використовувати їхні ресурси. У випадку великої кількості кластерів буде спостерігатися значний виграш у ресурсах. Це відбуватиметься за рахунок того, що кількість мікросервісів не залежить від кількості кластерів, на відміну він варіанту з типовою архітектурою де для кожного кластеру необхідне встановлення компонентів, які є досить вимогливими до ресурсів.

5 РОЗРОБКА СТАРТАП ПРОЕКТУ

5.1 Опис ідеї проекту

Проаналізуємо зміст ідеї, її можливі напрямки застосування, чим запропонована ідея відрізняється від існуючих аналогів, а також основні вигоди, які може отримати користувач продукту. Результати аналізу представлені у таблиці 5.1.

Таблиця 5.1 - Опис ідеї стартап-проекту

Зміст ідеї	Напрямки застосування	Вигода для користувачів
Розробити ПЗ для автоматизації CD процесу для багатокластерних систем	Застосування змін, які є простими yaml файлами	Можливість застосування змін, які є простими yaml файлами
	Застосування змін, які є kustomize файлами	Можливість застосування змін, які є kustomize файлами
	Видалення Kubernetes-об’єктів	Можливість видалити Kubernetes-об’єкти
	Редагування Kubernetes-об’єктів	Можливість редагувати Kubernetes-об’єкти
	Створення Kubernetes-об’єктів	Можливість створити Kubernetes-об’єкти

Таблиця 5.2 – Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№	Техніко-економічні характеристики ідеї	Можливі конкуренти			W(слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Запропонована архітектура	Flux CD	Argo CD			
1	Робота з простими yaml файлами	+	-	-			+
2	Робота з kustomize	+	-	-			+
3	Робота з helm release	-	+	+	+		
4	Необхідність встановлення в Kubernetes-кластер	+	-	-			+

Продовження таблиці 5.2

5	Техніко-економічні характеристики ідеї	Можливі конкуренти			W(слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Запропонована архітектура	Flux CD	Argo CD			
6	Клієнт для терміналу	-	+	+	+		
7	Використання кешування	+	-	+		+	

Основними відмінностями є робота з kustomize та необхідність встановлення в Kubernetes-кластер, а також робота з простими yaml файлами.

5.2 Технологічний аудит ідеї проекту

В даному підрозділі наведені технології, за допомогою яких можна реалізувати проект. Результат наведено у таблиці 5.3.

Таблиця 5.3 – Технології для реалізації ідеї проекту

№	Ідея проекту	Технології реалізації	Наявність технологій	Доступність технологій
1	Робота з простими yaml файлами Робота з	Компонент для взаємодії з Kubernetes	Необхідні: Kubectl	Доступні, Безкоштовні
2	Робота з kustomize			
3	Використання кешування	Компонент для зберігання закешованих даних	Необхідні: spring in-memory cache	Доступні, безкоштовні

Продовження таблиці 5.3

4	Видалення Kubernetes-об’єктів	Клієнт для командного рядка. На вході приймає ідентифікатор набору даних чи частини набору даних, на виході змінюється стан Kubernetes-кластеру	Необхідні: Kube-fabric-8	Доступні, Безкоштовні
5	Редагування Kubernetes-об’єктів			
6	Створення Kubernetes-об’єктів			

Продовження таблиці 5.3

№	Ідея проекту	Технології реалізації	Наявність технологій	Доступність технологій
7	Зберігання облікових даних в сховищі даних	Компонент архітектури. Зберігає облікові дані для доступу в кластери	Необхідні: vault	Доступні, безкоштовні
8	Використання мікросервісної архітектури	Серверний компонент, необхідні для розділення бізнес-логіки застосунку	Необхідні: Spring Framework	Доступні, безкоштовні
9	Використання брокера повідомлень	Серверний компонент. Запис файлів даних до MQ, видалення/додаваннянаборів даних та їх частин	Необхідні: RabbitMQ	Доступні, безкоштовні
Обрані технології реалізації ідеї проекту: RabbitMQ, Sping Framework, Kubectl, vault, spring in-memory cache. Обрані технології є фактично набором інструментів та фреймворків дляреалізації мети додатку, не потребують доробки.				

5.3 Аналіз ринкових можливостей запуску стартап-проекту

Таблиця 5.4 – Характеристика потенційних клієнтів стартап-проекту

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачів до товару
Автоматизація CD процесу для однокластерної системи	Невеликі та середні компанії та приватні підприємці, які мають необхідність в автоматизація CD процес, але не мають значних фінансів для здійснення цих операцій	– Наявність фінансового забезпечення – Різна кількість кластерів	– Стабільність – Постійна підтримка – Впровадження оновлень
Автоматизація CD процесу для багатокластерної системи	Великі та середні компанії та приватні підприємці, які мають необхідність в автоматизація CD процесі, володіють декількома кластерами		

Продовження таблиці 5.4

Потреба, що формує ринок	Цільова аудиторія (цільові сегменти ринку)	Відмінності у поведінці різних потенційних цільових груп клієнтів	Вимоги споживачівдо товару
Зручне керування конфігураціями	Середні за розміром та невеликі компанії, а також фізичні особи - підприємці	- Наявність фінансового забезпечення Різний об’єм конфігурацій	- Стабільність - Постійна підтримка Впровадження оновлень

Також необхідно оцінити фактори, які можуть завадити реалізації проекту. Оцінку наведено у таблиці 5.5.

Таблиця 5.5 – Фактори загроз

Фактор	Зміст загрози	Можлива реакція компанії
Поява конкурентів	Можлива поява конкурентів, які спроможуться створити більш якісний продукт або розширити вже популярний та існуючий сервіс. Можлива поява більш продуктивних продуктів	Удосконалення існуючого рішення новими технологіями, пришвидшення швидкодії за рахунок написанння низькорівневих бібліотек

Продовження таблиці 5.5

Фактор	Зміст загрози	Можлива реакція компанії
Значне зменшення популярності технології Kubernetes	Можливе зменшення популярності технології Kubernetes, що призведе до того, що розроблене ПЗ не буде потрібним	Адаптація програмного забезпечення до нової технології оркестрації, розроблення спеціальних адаптерів та інструкцій з міграції для користувачів
Відсутня підтримка деяких з бібліотек	Може статися так, що певні бібліотеки, від яких розроблене ПЗ залежить можуть перестати підтримувати нові версії Kubernetes, що призведе до неможливості підтримки цих версій.	Використовувати надійні та популярні бібліотеки, які здійснюють часте оновлення функціоналу та підтримують останні версії Kubernetes, як тільки вони з’явилися
Небезпека відмови сторонніх сервісів	Існує загроза того, що сторонні сервіси, а саме віддалені сервери, які містять репозиторії, можуть стати недоступними. Це призведе до неможливості роботи ПЗ та оновлення стану Kubernetes-кластеру	Розробити функціонал, який дозволяє здійснювати оновлення стану Kubernetes-кластеру з віддаленого репозиторію

Продовження таблиці 5.5

Фактор	Зміст загрози	Можлива реакція компанії
Відсутність репутації сервісу	Середні та малі за обсягом компанії можуть боятися використовувати невідомийсервіс,так як бояться, що цей сервіс зможе призвести до збоїв	Розробка функціоналу шифрування даних та отримання сертифікату безпеки від стороннього та відомого аудитора

У таблиці 5.6 представлено фактори можливості системи.

Таблиця 5.6 – Фактори можливостей

Фактор	Зміст можливості	Можлива реакція компанії
Невелика кількість конкурентів	На сьогоднішній день існує не так багато імплементацій GitOps підходу, які можна використати в продуктивних системах. Це дозволяє зайняти свою нішу в цьому напрямку.	Розвиток продукту, додаючи нові функції, які дозволять покращити якість роботи ПЗ та підвищити зручність використовування

Продовження таблиці 5.6

Фактор	Зміст можливості	Можлива реакція компанії
Можливість побудови власної репутації	Новий сервіс на ринку автоматизації процесів CD має всі можливості побудови власної репутації без попередньої історії	Пошук замовників, розповсюдження інформації про створену платформу, мінімальна націнка на вартості обчислень
Демонстраці явигоди використання розробленого додатка	Демонстрація потужності та економічності розробленого сервісу порівнянні з конкурентами	Створення веб сторінки додатку з інформацією про алгоритми роботита ефективність обраних методів, а також візуалізація графіків, порівняння швидкодії до кластерів у розробленому середовищі та середовищі конкурентів
Демонстраці япростоти користування розробленим додатком	Швидке занурення до інтерфейсу аналізу наборів даних та налаштування ПЗ	Розробка безкоштовних тимчасових підписок для звертання уваги потенційних користувачів на можливість використання створеної системи
Зростання попиту на системи автоматизації CD	Збільшення клієнтської бази	Привертання уваги потенційних користувачів та реклама

Основними загрозами у майбутньому є зменшення популярності технології Kubernetes, що можуть негативно вплинути на кількість потенційних клієнтів. Для запобігання даної загрози необхідно адаптаптувати ПЗ до нової технології оркестрації, розроблення спеціальних адаптерів та інструкцій з міграції для користувачів. Важливим фактором виходу на ринок є аналіз ринку пропозиції.

Для цього потрібно визначити загальні риси конкуренції на ринку. У таблиці 5.7 наведено ступеневий аналіз конкуренції на ринку.

Таблиця 5.7 – Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
1. Вказати тип конкуренції - чиста	Можливість вільно конкурувати на ринку	Можливість чесної конкуренції
2. За рівнем конкурентної боротьби – міжнародний	На ринку присутні іноземні фірми-конкуренти	Розробити додаток та інструкції іноземними мовами для збільшення потенційних користувачів на міжнародному ринку
3. За галузевою ознакою – міжгалузева	Розроблена архітектура може бути використана в різних галузях	Підтримка різних доменних областей, кластерів різних реалізацій.
4. Конкуренція за видами товарів – товарно-видова	Види товарів схожі за функціональністю	Враховувати недоліки інструментів автоматизації CD
6. За характером конкурентних переваг – цінова	Ціна - дуже важливий фактор при виборі продукту	Приділяти більше уваги вартості розробки та впровадження
7. За інтенсивністю – немарочна	Бренд не впливає на впізнаваність продукції	

Також, для кращого розуміння ринку, необхідно провести детальний аналіз умов конкуренції в галузі. Результати проведеного аналізу наведені у таблиці 5.8.

Таблиця 5.8 – Аналіз конкуренції в галузі за М.Портером

Складові аналізу	Прямі конкуренти в галузі	Потенційні конкуренти	Постачальники	Клієнти	Товари-замінники
	FluxCD	Jenkins	Значна кількість постачальників	Активно диктують умови	Збільшення кількості аналогічних товарів
Висновки	Доволі відома компанія, тому конкуренція інтенсивна	Компанія має потенціал запуску технологій автоматизації CD для кластерних систем	Постачальник не впливає на ринок	Мають основний вплив	Велика кількість товарів замінників

За результатами аналізу конкурентів зроблено висновок про можливість роботи на ринку, навіть при наявній конкуренції. Для подальшого розвитку проекту необхідно покращити швидкодію всього циклу безперервної доставки змін. Дані результати були враховані при порівняльному аналізі факторів конкурентоспроможності в таблиці 5.9.

Таблиця 5.9 – Обґрунтування факторів конкурентоспроможності

№ п/п	Фактор конкурентоспроможності	Обґрунтування (наведення чинників, щороблять фактор для порівняння конкурентних проектів значущим)
1	Ціна	Розроблене архітектурне рішення спрямоване на зниження ціни автоматизації CD процесу для багатокластерних систем, завдяки тому, що зменшується кількість необхідних ресурсів.
2	Швидкість оновлення стану Kubernetes-кластеру	Використання docker-контейнерів для того, щоб робота з Kubernetes-кластерами відбувалася паралельно. Це дозволить оновлювати стани Kubernetes-кластерів паралельно, що підвищить швидкість
3	Ізольованість даних	Використання docker-контейнерів для того, щоб ізолювати роботу з Kubernetes-кластерами. Це дозволить оновлювати стани Kubernetes-кластерів ізольовано, що підвищить конзистентність даних.
4	Безпека облікових даних доступу до Kubernetes-кластерів	Використання сховища облікових даних, яке пройшло всі необхідні сертифікації з безпеки даних

Використовуючи дані конкурентоспроможності проекту, необхідно проаналізувати сильні і слабкі сторони проекту (таблиця 5.10) та навести SWOT-аналіз

стартап-проекту (таблиця 5.11).

Таблиця 5.10 – Порівняльний аналіз сильних та слабких сторін архітектурного рішення для аналізу великого обсягу статистичних даних на пристроях з низькими технічними можливостями

№ п/п	Фактор конкурентоспроможності	Бали 1-20	Рейтинг товарів-конкурентів у порівнянні з хмарними технологіями для даних						
			-3	-2	-1	0	+1	+2	+3
1.	Ціна	20		+					
2.	Швидкість оновлення стану Kubernetes-кластеру	14					+		
3.	Ізольованість даних	16					+		
4.	Безпека облікових даних доступу до Kubernetes-кластерів	12			+				

Таблиця 5.11 – SWOT-аналіз стартап-проекту

Сильні сторони: Комбінація рішень для швидкого розгортання ПЗ для автоматизації CD на будь-який сервер, невелике використання дорогих компонентів серверу, що призводить до дешевизни продукту	Слабкі сторони: Вихід на ринок невідомою компанією,що потребує потужної рекламної компанії
Можливості: Новий сервіс на ринку автоматизації процесів CD має всі можливості побудови власної репутації без попередньої історії	Загрози: Можливе зменшення популярності технології Kubernetes, що призведе до того, що розроблене ПЗ не буде потрібним

Сильними сторонами розробленого проекту є низька вартість підтримки, висока швидкість виконання операцій та гнучкість роботи з Kubernetes-кластерами. Слабкими сторонами є недовіра до подібних проектів на перших етапах та наявність більш відомих конкурентів.

На основі SWOT-аналізу було розроблено альтернативи ринкової поведінки для виведення стартап-проекту на ринок. Далі необхідно проаналізувати строки реалізації та ймовірність отримання ресурсів. Результатианалізу наведені у таблиці 5.12.

Таблиця 5.12 – Альтернативи ринкового впровадження стартап-проекту.

№ п/п	Альтернатива (орієнтований комплекс заходів) ринкової поведінки	Ймовірність отримання результатів	Строки реалізації
1	Створення мінімального продукту з набором найголовніших функцій та швидкий вихід на ринок	Висока	Від 2 до 5 місяців
2	Створення потужного сервісу з великою кількістю різноманітних операцій з даними	Середня	Від 8 до 12 місяців

Було розглянуто дві основні стратегії – створення мінімального продукту для виходу на ринок для оцінки відгуків від користувачів, та створення продукту, що буде містити весь запланований функціонал. Обраною альтернативою єперша, тому що дуже важливим фактором вірного напряму розвитку проекту є відгуки та побажання користувачів. Реалізувавши основний функціонал системиу першу чергу, ми зможемо зрозуміти яких додаткових функцій не вистачає майбутнім користувачам та зможемо задовільнити їх потреби, не витрачаючи час на реалізацію методів, якими вони не будуть користуватися.

5.4 Розроблення ринкової стратегії проекту

Для створення ринкової стратегії проаналізуємо цільові групи потенційних користувачів. Результати аналізу наведені у таблиці 5.13.

Таблиця 5.13 – Вибір цільових груп потенційних споживачів

№ п/п	Опис профілю цільової групи потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи (сегменту)	Інтенсивність конкуренції в сегменті	Простота входу у сегмент
1	Фізичні особи підприємці	Середня	Середній	Низька	Низька
2	Компанії малого бізнесу	Висока	Високий	Низька	Низька
3	Компанії середнього бізнесу	Середня	Низький	Висока	Середня
Які цільові групи обрано: в першу чергу необхідно звернути увагу на компанії малого бізнесу, оскільки вони не мають можливості витрачати великі суми коштів на автоматизацію CD, але відчувають гостру необхідність в цьому.					

На основі отриманої інформації розробимо базову стратегію розвитку. Результати аналізу наведено у таблиці 5.14.

Таблиця 5.14 – Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Вихід на профільний ринок з сервісом автоматизації CD	Масовий маркетинг серед компаній малого бізнесу	Швидкість оновлення стану Kubernetes-кластеру.	Стратегія лідерства по витратах

На наступному етапі необхідно обрати стратегії конкурентної поведінки. На основі базової стратегії розвитку розробимо стратегію конкурентної поведінки (таблиця 5.15).

Таблиця 5.15 – Визначення базової стратегії конкурентної перевірки

Чи є проект «першопрохідцем» на ринку?	Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів?	Чи буде компанія копіювати основні характеристики товару конкурента, і які?	Стратегія конкурентної поведінки*
Ні, проект не є першопрохідцем на ринку, але володіє унікальною архітектурою в автоматизації CD	Компанія буде пропонувати свій продукт, як новим споживачам у даній сфері, так і тим, хто вже користується сервісами конкурентами	Так, оновлення стану Kubernetes-кластерів використовуючи GitOps-підхід	Стратегія наслідуваннялідеру

За стратегію конкурентної поведінки було обрано стратегію наслідування лідеру, за допомогою якої вдасться уникнути боротьби з лідерами ринку.

За допомогою проведених досліджень визначена стратегія позиціонування з урахуванням основних вимог до проекту та стратегій його розвитку, які наведені у таблиці 5.16.

Таблиця 5.16 – Визначення стратегій позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкуренто-спроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
Низька ціна, швидкість, функціональні можливості	Стратегія лідерства по витратах	Низька вартість підтримки автоматизації CD, висока швидкість, зручний інтерфейс роботи з ПЗ	Економічність, простота, швидкість,зручність

Отже, основою ринкової стратегії буде залучення уваги малих бізнес компаній, що відчувають необхідність у автоматизації CD для подальшого розвиткуна ринку, але не мають достатнього фінансового забезпечення для його проведення.

5.5 Розроблення маркетингової програми стартап-проекту

Першим кроком є формування маркетингової концепції товару, який отримає споживач. Для цього у таблиці 5.17 підсумовуються результати попереднього аналізу конкурентоспроможності проекту.

Таблиця 5.17 – Визначення ключових переваг концепції потенційноготовару

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити)
1	Швидкість CD	Швидкість оновлення стану Kubernetes-кластеру	Використання docker-контейнерів для того, щоб робота з Kubernetes-кластерами відбувалася паралельно. Це дозволить оновлювати стани Kubernetes-кластерів паралельно, що підвищить швидкість
2	Ізольованість даних	Ізолювати роботу з Kubernetes-кластерами завдяки використанню docker-контейнерів	Використання docker-контейнерів для того, щоб ізолювати роботу з Kubernetes-кластерами. Це дозволить оновлювати стани Kubernetes-кластерів ізольовано, що підвищить конзистентність даних
3	Безпека облікових даних	Безпека облікових даних доступу до Kubernetes-кластерів	Використання сховища облікових даних, яке пройшло всі необхідні сертифікації з безпеки даних

Надалі розробляється тривіальна маркетингова модель товару: уточняється ідея продукту та/або послуги, його фізичні складові, особливості процесу його надання (таблиця 5.18).

Таблиця 5.18 – Опис трьох рівнів моделі товару

<table>
<tr><td>Рівні товару</td><td>Сутність та складові</td></tr>
<tr><td>I. Товар за задумом</td><td>ПЗ для автоматизації CD для багатокластерних систем</td></tr>
<tr><td rowspan="3">II. Товар у реальному виконанні</td><td>Властивості/характеристики</td></tr>
<tr><td>1. ПЗ складається з декількох мікросервісів, які піднімаються в будь-якому середовищі.
2. Консоль для доступу до клієнту.</td></tr>
<tr><td>Марка: назва організації-розробника + назва товару</td></tr>
<tr><td rowspan="2">III. Товар із підкріпленням</td><td>До продажу: базова версія</td></tr>
<tr><td>Після продажу: постійна модернізація</td></tr>
<tr><td colspan="2">За рахунок чого потенційний товар буде захищено від копіювання: архітектура автоматизації CD для багатокластерних систем. Тобто захист ідеї товару буде відбуватися шляхом захисту інтелектуальної власності
на комплексне поєднання властивостей і характеристик.</td></tr>
</table>

Наступним кроком є визначення цінових меж, якими необхідно керуватисьпри встановлені ціни на потенційний товар (остаточне визначення ціни відбувається під час фінансово-економічного аналізу проекту), яке передбачає аналіз ціни на товари-аналоги або товари замінники, а також аналіз рівня доходівцільової групи споживачів (таблиця 5.19). Аналіз проводиться експертним методом.

Таблиця 5.19 – Визначення меж встановлення ціни

Рівень цін на товари-замінники	Рівень цін на товари-аналоги	Рівень доходів цільової групи споживачів	Верхня та нижня межі встановлення ціни на товар/послугу
Ціна починається від 0,4$ - 0.55$ за годину використання та залежить від технологій, які входятьу конкретний пакет. При оформленні пакету на рік, можливі знижкиу розмірі від 6% до 33%. Мінімальний тарифний план на рік дорівнює 25 тис. дол., а максимальний – 2 млн. дол. та залежить від кількості кластерів та розміру репозиторія.	Хмарні платформи в залежності від кількості кластерів, розміру репозиторіїв та кількості операцій починаються від 504 дол. на місяць і можуть досягати 14.5 тис. дол. намісяць.	1 тис. дол –7 тис. дол	200 дол. – 700 дол.

Наступним кроком є визначення оптимальної системи збуту (таблиця 5.20).

Таблиця 5.20 – Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Клієнти прагнуть отримувати товар за меншою ціною за рахунок конкуренції	Постійний доступ до сервісу	Однорівневий чи нульовий канал збуту	Власноруч тачерез посередників

Останньою складовою маркетингової програми є розроблення концепції маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування та визначену специфіку поведінки клієнтів (таблиця 5.21).

Таблиця 5.21 – Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Ручний або не повністю автоматизований CD	Соціальні мережі, dou, habrahabr, stackoverflow,	Моніторинг та прогнозування	Звернути увагу на ціну, зручність та швидкість порівняно з конкурентами	Дешевий та якісний автоматизований CD для багатокластерних систем

5.6 Висновок

Основною ідеєю проекту є створення ПЗ для автоматизації CD із налаштуванням та розгортанням та економічним швидким циклом CD у розробленому середовищі.

Основні технології, що будуть використані для реалізації проекту є технології та бібліотеки: Java, Spring, Kubectl, Vault, RabbitMQ. Обрані технології є фактично набором інструментів та фреймворків для реалізації мети додатку, не потребують доробки.

Аналіз майбутнього ринку збуту показав, що на сьогодні попит на автоматизацію CD зростає з неймовірною швидкістю. Розуміння своїх клієнтів та їх потреб єважливим фактором для будь якої компанії, тому ця галузь користується попитом майже у всіх сферах бізнесу.

Головним фактором конкурентоспроможності є невисока ціна за використання сервісу та надання прийнятної швидкості циклу CD для задоволення потреб потенційних користувачів.

Потенційною цільовою аудиторією користувачів розробленого проекту ємалий бізнес, який не спроможний сплачувати за потужні сервіси хмарних обчислень та має необхідність у автоматизації CD для подальшого розвитку на ринку. Основною стратегією є стратегія лідерства за ціновим сегментом на ринку, тобто надання потенційним користувачам якісних послуг за невеликою вартістю. Монетизація проекту буде відбуватися шляхом оформлення підписок, різниця між якими буде лише у обмеженні кількості кластерів та розмірів репозиторіїв.

У результаті аналізу, можна зробити висновок, що даний проект є досить перспективним, так як сфера автоматизації CD зростає з кожним роком, а вартість таких систем лише збільшується. Існуючі альтернативні рішення є значно дорожчими у використанні і тому стають не досить привабливими для ведення бізнесу. Завдяки здешевленню автоматизації CD можна залучити нові компанії, які раніше не мали можливості користування даною сферою.

ВИСНОВКИ

Життєвий цикл розробки програмного забезпечення (ПЗ) є застосуванням інженерних практик, спрямованих на підтримку працездатності, якості та надійності ПЗ. Вагоме місце серед них займає DevOps. Існує п'ять підходів до реалізації DevOps-практик. В контексті задачі мінімізації часу необхідного на етап доставки та розгортання найкращим є підхід з безперервною доставкою та розгортанням (CD). Його переваги:

– висока частота розгортання, що дозволяє зменшити різницю між версією, яка розгортається та поточною;

– менший технічний борг, простіша розробка;

– простіше відслідковувати помилки на продуктивному середовищі, оскільки зміни доставляються невеликими порціями.

GitOps є одним із варіантів реалізації DevOps, який здійснює підтримку безперервної доставки та розгортання (CD). Основною задачею, яку покликаний вирішувати GitOps, є безперервна доставка та розгортання (CD) в Kubernetes-середовищі. Серед основних переваг цього підходу, можна виділити наступні:

– зберігання кодової бази в одному місці;

– автоматизована доставка змін завдяки використанню gitops-агента;

– механізм PR, який дозволяє групувати зміни за задачами.

Для реалізації GitOps-підходу необхідна VCS. В контексті задачі підвищення ефективності процесу автоматизації CD для багатокластерних систем найефективнішим рішенням є використання саме розподіленої VCS. Перевагами такого типу VCS є розподіленість, що збільшує безпеку за рахунок багатьох копій репозиторія, велика кількість сучасних інструментів та сторонніх сервісів для роботи.

Для того, щоб реалізувати GitOps-підхід, вимагається наявність gitops-агенту, що являє собою спеціалізоване ПЗ, яке здійснює синхронізацію Git та стану Kubernetes-кластеру. В контексті задачі розробки gitops-агенту найбільш

оптимальною є мікросервісна архітектура. Вона дозволяє розбивати ПЗ на мікросервіси, які відповідають за певну частину функціональності. Це в свою чергу дозволяє ізолювати ці частини. Завдяки цьому ПЗ можна горизонтально масштабувати в залежності від потреб. Також проблема відсутня проблема відмови всієї системи в разі відмови одного з мікросервісів, оскільки вони ізольовані та слабкозв'язні.

Запропоновано архітектуру багатокластерної системи з безперервним розгортанням на основі мікросервісу синхронізації, що дозволяє зменшити кількість необхідних ресурсів для автоматизації CD.

Запропонований gitops-агент для синхронізації складається з таких компонент як мікросервіс роботи з git та мікросервіс роботи з Kubernetes.

Основними перевагами розробленого рішення є відсутність management-кластеру, для керування workload-кластерами, що зменшує до нуля витрати на його підтримку. Також розроблене рішення підвищує безпеку, оскільки зберігає облікові дані, необхідні для авторизації до workload-кластерів будуть зберігатися на спеціалізованому сервері, так званому сховищі облікових даних. Також важливою перевагою є те, що з самих workload-кластери не вимагається встановлення жодного додаткового ПЗ, на відміну від існуючого підходу, в якому вимагається встановлення gitops-агенту. Розроблена мікросервісна архітектура є асинхронною, що дозволить уникнути блокування на мікросервісах.

У ході тестування було показано, що власне архітектурне рішення є швидшим за рахунок того, що воно є асинхронним, використовує значно менше пам'яті та процесорного часу. Також воно є повністю незалежним від Kubernetes-кластерів, що дозволяє не використовувати їхні ресурси. У випадку, якщо кластерів є велика кількість, то буде спостерігатися значний виграш у ресурсах для рішення на базі мікросервісу синхронізації. Це відбуватиметься за рахунок того, що кількість мікросервісів не залежить від кількості кластерів. На відміну він варіанту з flux або

інших інструментів, де для кожного кластеру необхідне встановлення компонентів, які є досить вимогливими до ресурсів.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) M. Syahmi, D. Mejia Comparative study on software development methodologies. Database Systems Journal vol. V, no. 3/2014. 56 c.

2) Loukides, M. What is DevOps? O'Reilly Media, Inc., 2012.

3) Cook R. GitOps, an Elegant Tool for Hybrid Cloud Kubernetes. O'Reilly Media, Inc., 2019.

4) Jevtic G. What is SDLC? Phases of Software Development, Models, & Best Practices. URL: https://phoenixnap.com/blog/software-development-life-cycle (дата звернення: 20.11.2021).

5) Mittal A. Is Devops still a good career choice in 2021? URL: https://devopscurry.com/why-you-should-look-for-a-career-in-devops-in-2021/(дата звернення: 20.11.2021).

6) Fowler M., Highsmith J. The Agile Manifesto. URL: http://users.jyu.fi/~mieijala/kandimateriaali/Agile-Manifesto.pdf (дата звернення: 20.11.2021).

7) L. Bass, I. Weber, L. Zhu, DevOps. A software architect`s perspective. Addison-Wesley. 608 c.

8) Smeds, J., Nybom, K., & Porres, I. DevOps: a definition and perceived adoption impediments. In International conference on agile software development. Springer, Cham, 2015. C. 166-177.

9) Fowler M. ContinuousDelivery. URL: https://martinfowler.com/bliki/ContinuousDelivery.html (дата звернення: 20.11.2021).

10) Misra J., Cook W. R. Computation orchestration. Software & Systems Modeling, 2007. C. 83-110.

11) Cesar de la Torre. Containerized Docker Application Lifecycle with Microsoft Platform and Tools. Microsoft Developer Division, .NET and Visual Studio product teams.

12) Haley J. Demystifying containers, Docker, and Kubernetes. URL: https://cloudblogs.microsoft.com/opensource/2019/07/15/how-to-get-started-containers-docker-kubernetes/ (дата звернення: 20.11.2021).

13) Sayfan, G.. Mastering kubernetes. Packt Publishing Ltd, 2017.

14) Компоненты Kubernetes. URL: https://kubernetes.io/ru/docs/concepts/overview/components/ (дата звернення: 20.11.2021).

15) Wickramasinghe Sh. Kubernetes Multi-Clusters: How & Why To Use Them. URL: https://blogs.bmc.com/kubernetes-multi-clusters/?print=pdf (дата звернення: 20.11.2021).

16) Chacon, S., & Straub, B. Pro git. Springer Nature, 2014. 456 с.

17) Kustomize Controller. URL: https://fluxcd.io/docs/components/kustomize/ (дата звернення: 20.11.2021).

18) Limoncelli, T. A. GitOps: A Path to More Self-service IT: IaC+ PR= GitOps. Queue, 16(3), 2018. С. 13-26.

19) Case R., GitOps and Cluster API: Multi-cluster Manager. URL: https://www.weave.works/blog/gitops-and-cluster-api-master-of-masters (дата звернення: 20.11.2021).

20) Gnatyk R., Microservices vs Monolith: which architecture is the best choice for your business? URL: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/ (дата звернення: 20.11.2021).

21) Fowler M. Don`t start monolith. URL: https://martinfowler.com/articles/dont-start-monolith.html (дата звернення: 20.11.2021).

22) Fowler M. Microservices URL: https://martinfowler.com/articles/microservices.html\ (дата звернення: 20.11.2021).

23) Arnold, K., Gosling, J., & Holmes, D. The Java programming language. Addison Wesley Professional, 2005.

24) Hejlsberg, A., Torgersen, M., Wiltamuth, S., & Golde, P. The C# programming language. Pearson Education, 2008.

25) Van Rossum, G., & Drake Jr, F. L. Python tutorial (Vol. 620). Amsterdam: Centrum voor Wiskunde en Informatica, 1995.

26) Crockford, D. JavaScript: The Good Parts: The Good Parts. O'Reilly Media, Inc., 2008.

ДОДАТОК А. ЛІСТИНГ ПРОГРАМИ

```
package org.romanchi.gitops.synchronization.git;

import org.eclipse.jgit.api.errors.GitAPIException;
import org.eclipse.jgit.diff.DiffEntry;
import org.eclipse.jgit.lib.ObjectId;
import org.eclipse.jgit.revwalk.RevCommit;
import org.romanchi.gitops.libgit.BaseGitClient;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.io.IOException;
import java.util.List;


public class SynchronizationGitClient extends BaseGitClient {

    private final Logger logger =
LoggerFactory.getLogger(SynchronizationGitClient.class.getName());

    public SynchronizationGitClient(String repoPath) throws IOException {
        super(repoPath);
    }

    public List<DiffEntry> updates() throws IOException, GitAPIException {
        Iterable<RevCommit> logs = git.log()
                .call();
        ObjectId localHead = logs.iterator().next().getId();

        fetch();

        logs = git.log()
                .add(repository.resolve("remotes/origin/main"))
                .call();
        ObjectId remoteHead = logs.iterator().next().getId();

        logger.info("Local head=[{}], remote head=[{}]", localHead, remoteHead);

        pull();

        final List<DiffEntry> diffs = git.diff()
                .setOldTree(prepareTreeParser(repository, localHead.name()))
                .setNewTree(prepareTreeParser(repository, remoteHead.name()))
                .call();

        if(!localHead.name().equals(remoteHead.name()) && diffs.isEmpty()){
            logger.info("Heads changed, but diffs lost");
        }else{
            logger.info("Found: {} differences", diffs.size());
        }

        for (DiffEntry diff : diffs) {
            logger.info(diff.getChangeType() + ": " +
                    (diff.getOldPath().equals(diff.getNewPath()) ? diff.getNewPath()
: diff.getOldPath() + " -> " + diff.getNewPath()));
        }
        return diffs;
    }
```

```
}


package org.romanchi.gitops.synchronization.services;

import org.eclipse.jgit.api.errors.GitAPIException;
import org.eclipse.jgit.diff.DiffEntry;
import org.romanchi.gitops.libmq.MQUtils;
import org.romanchi.gitops.libmq.SimpleRabbitTemplate;
import org.romanchi.gitops.synchronization.git.SynchronizationGitClient;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Service;

import java.io.IOException;
import java.util.List;

@Service
public class UpdatesService {

    private final Logger logger = LoggerFactory.getLogger(UpdatesService.class);
    private final SimpleRabbitTemplate simpleRabbitTemplate;

    public UpdatesService(SimpleRabbitTemplate simpleRabbitTemplate) {
        this.simpleRabbitTemplate = simpleRabbitTemplate;
    }

    @Scheduled(cron = "*/15 * * * * *")
    public void pullUpdatesFromOriginRepo(){
        try {
            SynchronizationGitClient synchronizationGitClient = new
SynchronizationGitClient("/Users/romanmaliarchuk/IdeaProjects/k8s-sync/.git");
            List<DiffEntry> diffList = synchronizationGitClient.updates();
            if(!diffList.isEmpty()){
                simpleRabbitTemplate.convertAndSend("gitops.repository.exchange",
                        MQUtils.getRoutingForQueue("repository.updates"), diffList);
            }
        } catch (GitAPIException | IOException e) {
            logger.info(e.getMessage(), e);
        }
    }

}


package org.romanchi.gitops.kubernetes.kubectl;


import org.apache.commons.lang3.StringUtils;
import org.romanchi.gitops.kubernetes.dto.ContextInfo;
import org.romanchi.gitops.kubernetes.dto.ObjectInfo;

import java.io.BufferedInputStream;
import java.io.IOException;
import java.net.URL;
import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.Collections;
```

```
import java.util.List;
import java.util.Optional;
import java.util.logging.Logger;
import java.util.stream.Collectors;

public class KubectlClient {
    private static final Logger logger =
Logger.getLogger(KubectlClient.class.getName());

    public static List<ContextInfo> contextInfos(){
        String [] command = new String[]{"kubectl", "config", "get-contexts"};

        try {
            String [] result = execCommand(command);
            return Arrays.stream(result).skip(1).map(it -> {
                String[] columns = it.split("\\s+");
                ContextInfo contextInfo = new ContextInfo();
                contextInfo.setCurrent(columns[0]);
                contextInfo.setName(columns[1]);
                contextInfo.setCluster(columns[2]);
                contextInfo.setAuthinfo(columns[3]);
                return contextInfo;
            }).collect(Collectors.toList());
        } catch (Exception e) {
            e.printStackTrace();
            return Collections.emptyList();
        }
    }

    public static boolean useContext(ContextInfo contextInfo){
        String name = contextInfo.getName();
        String[] command = new String[]{"kubectl", "config", "use-context", name};
        try {
            String[] result = execCommand(command);
            return result[0].contains("Switched");
        } catch (IOException e) {
            e.printStackTrace();
            return false;
        }
    }

    public static Optional<ContextInfo> currentContext(){
        List<ContextInfo> contextInfos = contextInfos();
        return contextInfos.stream()
                .filter(it -> StringUtils.isNotBlank(it.getCurrent()))
                .findFirst();
    }

    public static Optional<ContextInfo> getContextByName(String name){
        List<ContextInfo> contextInfos = contextInfos();
        return contextInfos.stream()
                .filter(it -> name.equals(it.getName()))
                .findFirst();
    }

    public static String[] apply(String path) throws IOException {
        String[] applyCommand = new String[]{"kubectl", "apply", "-f", path};
        String[] result = execCommand(applyCommand);
        return result;
```

```
    }

    public static String[] apply(URL url) throws IOException {
        return apply(url.getPath());
    }

    public static String[] delete(String path) throws IOException {
        String[] deleteCommand = new String[]{"kubectl", "delete", "-f", path};
        String[] result = execCommand(deleteCommand);
        return result;
    }

    public static String[] delete(URL url) throws IOException {
        return delete(url.getPath());
    }

    public static List<ObjectInfo> objectsByTypeInAllNamespaces(String type) {
        String[] command = new String[]{"kubectl", "get", type, "-A"};
        String[] result;
        try {
            result = execCommand(command);
            return Arrays.stream(result).skip(1).map(it -> {
                String[] columns = it.split("\\s+");
                return ObjectInfo.builder()
                        .withNamespace(columns[0])
                        .withName(columns[1])
                        .withReady(columns[2])
                        .withStatus(columns[3])
                        .withRestarts(columns[4])
                        .withAge(columns[5])
                        .build();
            }).collect(Collectors.toList());
        } catch (IOException e) {
            e.printStackTrace();
            return Collections.emptyList();
        }

    }

    private static String [] execCommand(String[] commands) throws IOException {
        Process process = Runtime.getRuntime().exec(commands);
        boolean error = process.getErrorStream().available() > 0;
        if(error){
            String command = String.join(" ", commands);
            throw new RuntimeException(String.format("Error during call [%s]",
command));
        }
        BufferedInputStream bufferedInputStream = new
BufferedInputStream(process.getInputStream());
        return new String(bufferedInputStream.readAllBytes(),
StandardCharsets.UTF_8).trim().split("\n");
    }
}


package org.romanchi.gitops.kubernetes.services;

import com.fasterxml.jackson.core.type.TypeReference;
import org.romanchi.gitops.kubernetes.dto.Diff;
```

```
import org.romanchi.gitops.kubernetes.dto.Mode;
import org.romanchi.gitops.kubernetes.kubectl.KubectlClient;
import org.romanchi.gitops.libmq.MQJsonMapper;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

@Component
public class ListenerService {

    private final Logger logger =
LoggerFactory.getLogger(ListenerService.class.getName());
    private final MQJsonMapper mqJsonMapper;
    private final String repoBaseFolder = "/Users/romanmaliarchuk/IdeaProjects/k8s-
sync/";
    private final String backupRepoBaseFolder = "/Users/romanmaliarchuk/k8s-sync";
    private final List<String> clusters = Arrays.asList("minikube"/*, "kind-kind"*/);
    private final KubernetesGitClient backupRepoGitClient;

    public ListenerService(MQJsonMapper mqJsonMapper) throws IOException {
        this.mqJsonMapper = mqJsonMapper;
        this.backupRepoGitClient = new KubernetesGitClient(backupRepoBaseFolder +
"/.git");
    }

    @RabbitListener(queues = "#{repositoryUpdatesQueue}")
    public void listener(Message message) throws Exception {
        List<Diff> diffs = mqJsonMapper.readValue(new String(message.getBody(),
StandardCharsets.UTF_8), new TypeReference<>() {});

        List<Diff> deleteDiffs = diffs.stream()
                .filter(it -> Mode.DELETE.equals(it.getChangeType()))
                .collect(Collectors.toList());

        List<Diff> nonDeleteDiffs = diffs.stream()
                .filter(it -> !Mode.DELETE.equals(it.getChangeType()))
                .collect(Collectors.toList());

        clusters.forEach(cluster -> {
            long start = System.nanoTime();
            logger.info();

KubectlClient.getContextByName(cluster).ifPresent(KubectlClient::useContext);
            nonDeleteDiffs.forEach(diff -> {
                try{
                    Arrays.stream(KubectlClient.apply(repoBaseFolder +
diff.getWorkPath())).forEachOrdered(logger::info);
                }catch (Exception ex){
                    logger.info(ex.getMessage(), ex);
                }
            });
```

```
            deleteDiffs.forEach(diff -> {
                try{
                    Arrays.stream(KubectlClient.delete(backupRepoBaseFolder + "/" +
diff.getWorkPath())).forEachOrdered(logger::info);
                }catch (Exception ex){
                    logger.info(ex.getMessage(), ex);
                }
            });
            long end = System.nanoTime();
            logger.info("Time elapsed: {}ms", (end - start)/1_000_000);
        });

        backupRepoGitClient.pull();
    }

}
```

```
package org.romanchi.gitops.libmq;

import org.springframework.amqp.core.Queue;

public abstract class BaseQueue extends Queue {
    private String baseQueueName;

    public BaseQueue(BaseDirectExchange exchange, BaseDeadLetterQueue
deadLetterQueue) {
        super("gitops." + deadLetterQueue.getBaseQueueName() + ".queue");
        this.baseQueueName = deadLetterQueue.getBaseQueueName();
        this.getArguments().put("x-dead-letter-exchange", exchange.getName());
        this.getArguments().put("x-dead-letter-routing-key",
MQUtils.getRoutingForDeadLetterQueue(deadLetterQueue));
    }

    public String getBaseQueueName() {
        return baseQueueName;
    }
}
```

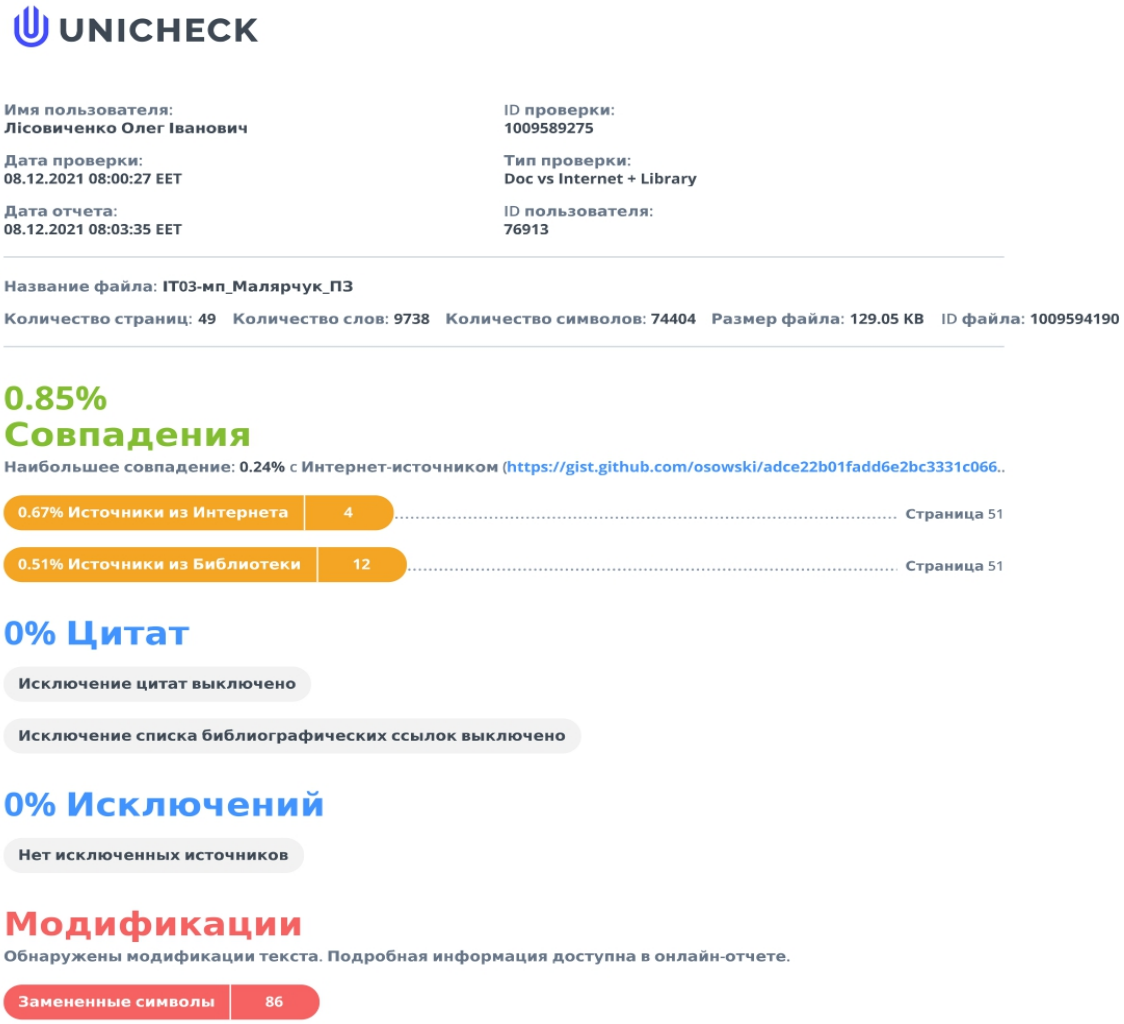
```
package org.romanchi.gitops.libmq;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.stereotype.Component;

@Component
public abstract class BaseJSONRabbitTemplate extends RabbitTemplate {
    public BaseJSONRabbitTemplate(BaseMQConnectionFactory connectionFactory){
        super(connectionFactory);
        Jackson2JsonMessageConverter messageConverter = new
Jackson2JsonMessageConverter();
        messageConverter.setCreateMessageIds(true);
        this.setMessageConverter(messageConverter);
        this.setChannelTransacted(true);
    }
}
```

```
package org.romanchi.gitops.libgit;

import org.eclipse.jgit.api.Git;
import org.eclipse.jgit.api.TransportConfigCallback;
import org.eclipse.jgit.api.errors.GitAPIException;
import org.eclipse.jgit.lib.ObjectReader;
import org.eclipse.jgit.lib.Repository;
import org.eclipse.jgit.revwalk.RevCommit;
import org.eclipse.jgit.revwalk.RevTree;
import org.eclipse.jgit.revwalk.RevWalk;
import org.eclipse.jgit.storage.file.FileRepositoryBuilder;
import org.eclipse.jgit.treewalk.AbstractTreeIterator;
import org.eclipse.jgit.treewalk.CanonicalTreeParser;

import java.io.File;
import java.io.IOException;


public class BaseGitClient {

    protected final Git git;
    protected final Repository repository;
    private final TransportConfigCallback transportConfigCallback;

    public BaseGitClient(String repoPath) throws IOException {
        this.repository = new FileRepositoryBuilder()
                .setGitDir(new File(repoPath))
                .build();
        this.git = new Git(repository);
        this.transportConfigCallback = new SshTransportConfigCallback();
    }

    public void pull() {
        try {
            git.pull().setTransportConfigCallback(transportConfigCallback).call();
        } catch (GitAPIException e) {
            e.printStackTrace();
        }
    }

    protected void fetch() {
        try {
            git.fetch().setTransportConfigCallback(transportConfigCallback).call();
        } catch (GitAPIException e) {
            e.printStackTrace();
        }
    }

    protected static AbstractTreeIterator prepareTreeParser(Repository repository,
String objectId) throws IOException {

        try (RevWalk walk = new RevWalk(repository)) {
            RevCommit commit = walk.parseCommit(repository.resolve(objectId));
            RevTree tree = walk.parseTree(commit.getTree().getId());

            CanonicalTreeParser treeParser = new CanonicalTreeParser();
            try (ObjectReader reader = repository.newObjectReader()) {
                treeParser.reset(reader, tree.getId());
            }
```

```
            walk.dispose();

            return treeParser;
        }
    }

}
```

```
package org.romanchi.gitops.libgit;

import com.jcraft.jsch.JSch;
import com.jcraft.jsch.JSchException;
import com.jcraft.jsch.Session;
import org.eclipse.jgit.api.TransportConfigCallback;
import org.eclipse.jgit.transport.*;
import org.eclipse.jgit.util.FS;

public class SshTransportConfigCallback implements TransportConfigCallback {

    private final SshSessionFactory sshSessionFactory = new
JschConfigSessionFactory() {
        @Override
        protected void configure(OpenSshConfig.Host hc, Session session) {
            session.setConfig("StrictHostKeyChecking", "no");
        }

        @Override
        protected JSch createDefaultJSch(FS fs) throws JSchException {
            JSch jSch = super.createDefaultJSch(fs);
            jSch.addIdentity("~/.ssh/new_rsa", "frdfhtkm12".getBytes());
            return jSch;
        }
    };

    @Override
    public void configure(Transport transport) {
        SshTransport sshTransport = (SshTransport) transport;
        sshTransport.setSshSessionFactory(sshSessionFactory);
    }

}
```

ДОДАТОК Б. РЕЗУЛЬТАТ ПЕРЕВІРКИ НА СПІВПАДІННЯ



UNICHECK

Имя пользователя:
Лісовиченко Олег Іванович

ID проверки:
1009589275

Дата проверки:
08.12.2021 08:00:27 EET

Тип проверки:
Doc vs Internet + Library

Дата отчета:
08.12.2021 08:03:35 EET

ID пользователя:
76913

Название файла: IT03-мп_Малярчук_ПЗ

Количество страниц: 49 Количество слов: 9738 Количество символов: 74404 Размер файла: 129.05 KB ID файла: 1009594190

0.85%
Совпадения

Наибольшее совпадение: 0.24% с Интернет-источником (https://gist.github.com/osowski/adce22b01fadd6e2bc3331c066..

0.67% Источники из Интернета 4 ..... Страница 51

0.51% Источники из Библиотеки 12 ..... Страница 51

0% Цитат

Исключение цитат выключено

Исключение списка библиографических ссылок выключено

0% Исключений

Нет исключенных источников

Модификации

Обнаружены модификации текста. Подробная информация доступна в онлайн-отчете.

Замененные символы 86