Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

Навчально-науковий інститут комп'ютерних наук
та інформаційних технологій
(повне найменування інституту, назва факультету (відділення))

Кафедра комп'ютерних наук
(повна назва кафедри (предметної, циклової комісії))

Магістерська кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему: "Автоматизація процесів розгортання програмних сервісів за допомогою системи керування життєвим циклом Keptn"

Виконав: студент 6 курсу групи КН-63м
спеціальності
122 "Комп'ютерні науки"
(шифр і назва напряму підготовки, спеціальності)

Левченко О. І.
(прізвище та ініціали)

Керівник Борецька І. Б.
(прізвище та ініціали)

Рецензент Лущин С. О.
(прізвище та ініціали)

Львів – 2024

Національний лісотехнічний університет України
(повне найменування вищого навчального закладу)

ННІ комп'ютерних наук та інформаційних технологій
Кафедра комп'ютерних наук
Рівень вищої освіти другий (магістерський)
Спеціальність 122 "Комп'ютерні науки"
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Борецька І.Б.
"05" січня 2024 року

З А В Д А Н Н Я
НА ДИПЛОМНУ РОБОТУ СТУДЕНТУ

Левченко Олег Іванович
(прізвище, ім'я, по батькові)

1. Тема роботи "Автоматизація процесів розгортання програмних сервісів за допомогою системи керування життєвим циклом Keptn"

керівник роботи *Борецька Ірина Богданівна, к.т.н., доцент*,
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом вищого навчального закладу від "13" лютого 2023 року № С-49

2. Термін подання студентом роботи "05" січня 2024 року
3. Вихідні дані до роботи документація Keptn, документація Kubernetes, матеріали мережі Інтернет, що стосуються теми роботи

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити):
Розділ 1. Стан проблемної області
Розділ 2. Інформаційне забезпечення
Розділ 3. Математичне забезпечення
Розділ 4. Програмне забезпечення
Розділ 5. Розроблення стартап-проєкту
Висновки
5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): діаграми фаз розгортання в різних оточеннях; схеми архітектури інструментів розгортання; схема розгортання із кроками контролю якості; алгоритм процедури самостійної стабілізації сервісу; знімки екранів із відображенням результатів роботи конвеєру.

6. Дата видачі завдання "15" лютого 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1	Отримання завдання	15.02.2023	виконано
2	Огляд процесів розгортання сервісів	30.04.2023	виконано
3	Огляд існуючих інструментів розгортання	31.07.2023	виконано
4	Підготовка тестової платформи	30.08.2023	виконано
5	Створення конвеєру розгортання у декількох оточеннях	29.09.2023	виконано
6	Тестування роботи конвеєру	31.10.2023	виконано
7	Розробка стартап-проєкту	30.11.2023	виконано
8	Підготовка пояснювальної записки	29.12.2023	виконано
9	Подання готової роботи	05.01.2024	виконано

Студент _____ Левченко О.І.
(підпис) (прізвище та ініціали)

Керівник роботи _____ Борецька І.Б.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота містить 76 сторінки пояснювальної записки, 22 рисунків, 16 таблиць, 1 додаток, 35 джерел.

В даній роботі були проведені огляд процесів розгортання програмних сервісів, підходів до розгортання, проблеми контролю якості розгорнутих сервісів, огляд існуючих рішень автоматичного розгортання, а також визначені алгоритми та інструменти для реалізації конвеєру автоматичного розгортання. Реалізовано конвеєр під оркестрацією системи керування життєвим циклом розгортання Keptn з використанням стадій контролю якості.

Ключові слова: конвеєр, розгортання сервісів, автоматизація операцій, ворота контролю якості, цілі рівня обслуговування, оркестрація, Kubernetes, Keptn, життєвий цикл, yaml, DevOps.

ABSTRACT

The thesis contains 76 pages of explanatory note, 22 figures, 16 tables, 1 appendix, 35 literary sources.

In this work were reviewed: service deployment processes, deployment approaches, problem of quality control of deployed services, existing solutions for automated service deployment; also there were determined algorithms and tools for creation of automated deployment pipeline. Implemented multistaged deployment pipeline with quality gates under management of deployment lifecycle orchestration system Keptn.

Keywords: pipeline, service deployment, operations automation, quality gates, Service-Level Objectives, orchestration, Kubernetes, Keptn, lifecycle, yaml, DevOps.

ТЕХНІЧНЕ ЗАВДАННЯ

Дослідити процеси розгортання програмних сервісів та створити автоматизований конвеєр на базі системи керування життєвим циклом Keptn:

- оглянути процеси розгортання сервісів
- оглянути інструментарій розгортання
- підготувати тестову платформу
- створити конвеєр автоматичного розгортання у декількох оточеннях
- створити послідовності для автоматичних операцій:
    - відновлення попередньої стабільної версії
    - відновлення рівня обслуговування
- виконати тестування роботи коду конвеєру

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 8

ВСТУП 9

Розділ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ 11

1.1. Процес розгортання програмних сервісів 11

1.2. Фазовий підхід до розгортання 15

1.3. Контроль якості розгортання 20

Висновки до розділу 22

Розділ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ 23

2.1. Огляд існуючих рішень для автоматизації розгортання 23

2.2. Інструменти для створення конвеєру розгортання 42

2.3. Платформа для розгортання 43

Висновки до розділу 43

Розділ 3. АЛГОРИТМІЧНЕ ЗАБЕЗПЕЧЕННЯ 45

3.1. Просування артефакту по конвеєру 45

3.2. Автоматизація задач пост-розгортання 46

Висновки до розділу 48

Розділ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ 49

4.1. Підготовка платформи 49

4.2. Створення конвеєру розгортання 52

4.3. Тестування роботи конвеєру 55

Висновки до розділу 60

Розділ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ 62

5.1. Опис ідеї проєкту 62

5.2. Розроблення ринкової стратегії 67

5.3. Розроблення маркетингової програми 69

Висновки до розділу 70

ВИСНОВКИ..... 72
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ 75
ДОДАТКИ..... 77

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

CI	Continuous Integration
CD	Continuous Deployment
CO	Continuous Operations
AO	Automated Operations
QG	Quality Gates
VCS	Version Control System
VM	Virtual Machine
ОС	Операційна система (наприклад Windows, Linux)
ЦП	Центральний процесор
SLO	Service-Level Objectives
SLI	Service-Level Indicators
CLI	Command Line Interface
API	Application Programming Interface
CRUD	Create, Read, Update, Delete

ВСТУП

Актуальність. З розвитком Інтернету та технологій підхід до розробки та надання послуг кардинально змінився. Розвиток Інтернету та технологій сприяв еволюції програмних сервісів та способів їх розміщення на серверах. Спочатку компанії використовували виділені фізичні сервери, що серед недоліків мало потребу в штаті кваліфікованих спеціалістів та неефективне використання ресурсів серверів. Сервери були або «недовантажені», тобто не використовувались на 100%, або ж були «перевантажені», що відбивалось на якості сервісів та створювало потребу у введенні нових серверів, які, в свою чергу, були «недовантажені». Пізніше, з появою віртуалізації ефективність використання фізичних серверів підвищилась завдяки можливості запускати паралельно кілька віртуальних машин на одному фізичному сервері. Такий підхід також має свої недоліки щодо ефективності у вигляді необхідності віртуалізації фізичних компонентів та ОС для кожної віртуальної машини. Поява контейнеризації підвищила ефективність використання фізичних серверів, дозволивши віртуалізувати лише «корисне навантаження», тобто сам програмний сервіс. З розповсюдженням концепції мікросервісної архітектури кількість контейнерів, які розгортаються компанією, може налічувати тисячі та збільшуватись кратно кількості оточень, що ставить питання автоматизації розгортання сервісів. Саме таку проблему дозволяє вирішувати система Keptn, що дозволяє оркеструвати процес розгортання багатьох сервісів з урахуванням оточень.

Об’єкт дослідження. Об’єктом дослідження є системи автоматичного розгортання програмних сервісів.

Предмет дослідження. Предметом дослідження є конфігурування системи автоматичного розгортання сервісів.

Мета. Метою роботи є дослідження сучасних підходів і методів для розгортання програмних сервісів, вибір необхідних технологій та інструментів, а також створення базового автоматизованого конвеєру розгортання програмного сервісу на базі системи керування життєвим циклом Keptn, що дозволить полегшити прийняття рішень про якість розгорнутих сервісів.

Завдання. Для досягнення мети роботи необхідно вирішити такі завдання:

- проаналізувати процес розгортання програмних сервісів,
- проаналізувати наявні інструменти,
- створити конвеєр автоматичного розгортання, який дозволить полегшити або автоматизує прийняття рішень про якість розгорнутих сервісів,
- в межах конвеєру створити послідовності задач для автоматичних операцій:
    - відновлення попередньої стабільної версії
    - відновлення рівня обслуговування

Наукова новизна. Наукова новизна роботи полягає в дослідженні інструментів та поєднання різних систем для автоматичного розгортання програмних сервісів, а також вирішенні виклику контролю якості розгорнутих сервісів.

Практична значимість. Отриманий конвеєр продемонструє можливості сучасних систем розгортання програмних сервісів, особливо в частині автоматичного прийняття рішень про якість розгорнутих сервісів та автоматизації операції пост-розгортання. Впровадження такого конвеєру забезпечує однорідність та консистентність процесів розгортання, оскільки усі етапи виконуватимуться згідно визначених процедур, знижує рівень ризику доставки неякісного коду та помилок конфігурації, виявляючи помилки та деградацію якості на ранніх етапах, мінімізує негативний вплив на кінцевого користувача.

Розділ 1. СТАН ПРОБЛЕМНОЇ ОБЛАСТІ

1.1. Процес розгортання програмних сервісів

Процес розгортання програмного сервісу (надалі — сервісу) — доволі складний і важкий процес, в якому є багато етапів, на яких можуть бути задіяні різні люди, різних спеціалізацій, якщо це не автоматизований процес і багато різноманітних допоміжних інструментів — якщо автоматизований.

Раніше сервіси були невеликих розмірів, що дозволяло пройти увесь процес одній людині і доволі швидко у ручному режимі. Починаючи від самої розробки і закінчуючи тим, як готовим сервісом міг користуватись кінцевий користувач. Але, з розвитком інформаційних технологій, пропускної здатності Інтернету, збільшенням кількості користувачів таких сервісів і глобалізацією, потреби до відповідних сервісів ставали все більш вимогливими, оскільки через повсюдне використання Інтернет з’являються все більше конкуруючих продуктів і для отримання конкурентних переваг вже не достатньо лише хороших цін.

Тепер став важливим візуальний вигляд продукту, кількість анімацій, відео та іншого різноманітного контенту, що повинен бути у сервісі для утримання кінцевого користувача. Також від сервісів вимагається швидкість роботи та завантаження контенту, а також безпека користувача, його персональної інформації та захист інформації самої компанії, що володіє цим сервісом.

Всі вище наведені фактори сильно ускладнюють сервіс, що спричиняє до потреби розширення команди, яка працює над ним, а також до стрімкого збільшення кількості функцій у цьому сервісі. Як наслідок — постають проблема тестування усього продукту під час кожного нового оновлення сервісу та проблема автоматизації.

Також останнім часом набуває популярності розділяти сервіс на багато мікросервісів, кожен з яких відповідає за свою власну функцію, що робить процес ручного розгортання сервісу практично неможливим, коли кількість

таких сервісів наближається до десятків або сотень, оскільки для кожного мікросервісу потрібно провести всі етапи розгортання, що раніше проводились лише один раз для одного сервісу. Процес розгортання сервісу включає в себе такі етапи як:

- збереження вихідного коду;
- модульне тестування;
- інтеграційне тестування;
- збірка;
- розгортання.

Збереження вихідного коду

Збереження вихідного коду — це перший етап у розгортанні сервісу. У минулому, коли сервіси були меншого розміру вихідний код програми міг зберігатись на локальному комп'ютері або в якомусь загальному сховищі. Та, коли обсяг вихідного коду збільшувався, а команда, що працювала над ним, розросталась до десятків або сотень людей — цей підхід перестав задовольняти потреби, оскільки зі збільшенням кількості розробників, що приймає участь у написанні коду, постає проблема зростання кількості конфліктів між ними під час збереження результатів роботи.

Для вирішення таких конфліктів існує VCS — система контролю версій, яка знаходиться на локальному сервері або у хмарі. Кожен розробник, закінчуючи написання своєї частину коду, зберігає результат у цю систему, попередньо провівши синхронізацію з актуальною версією в репозиторії. В подальшому ця система допомагає у локалізації коду, що спричиняє помилки, а також у створенні документації [10].

Збірка

Збірка сервісу — це перетворення вихідного коду проєкту у декілька файлів бібліотек, або файлів, що можна запускати.

Збірка зазвичай відбувається після того, як в певний момент розробник вирішує, що та версія вихідного коду, що є у системі контролю версій на той момент, має певну логічну завершеність та повинна бути розгорнута для тестування або ж використання. В цей момент розробник виконує збірку проєкту або на своєму локальному комп'ютері, або ж за допомогою CI системи на сервері. Результатом виконання цього етапу є файли, які в подальшому будуть розгорнуті як сервіс.

Сама ж збірка проєкту відбувається за допомогою компілятора тієї мови програмування, якою написано код сервісу. Тому, в разі автоматизації процесу збірки, такий компілятор має бути попередньо встановлений та налаштований на сервері, де відбувається збірка.

Модульне тестування

Модульне тестування — це тестування сервісу, що є фундаментальною практикою програмування, яка включає в себе тестування окремих, розділених підсистем сервісу ізольовано від інших підсистем, зазвичай закриваючи зв'язки між такими підсистемами штучними зв'язками. Хоча більшість нетривіальних частин важко перевірити ізольовано, прикладом може виступати запис до бази даних якоїсь інформації.

Важко уникнути написання тестових наборів, які є неповними та складними в обслуговуванні й інтерпретації. Використання Mock Objects (об'єкти, що допомагають ізолювати зв'язки підсистеми з іншими підсистемами) для модульного тестування покращує як вихідний код, так і тестові набори. Вони дозволяють писати модульні тести для всього, спрощують структуру тестів і уникають забруднення коду сервісу інфраструктурою тестування. Модульне тестування, зроблене правильно, відрізняє невдалий успішний проєкт від невдалого, підтримувану базу коду та базу коду, до якої ніхто не наважується навіть торкнутися[34].

Цей етап у розгортанні сервісу може бути автоматизований, або виконуватись вручну. Модульні тести зберігають зазвичай разом із вихідними

кодами сервісів, їх ручний запуск потребує лише відкрити проєкт та запустити ці тести у роботу. Оскільки це ізольовані тести підсистем і вони не потребують використання зовнішніх систем, таких як файлова система, бази даних та інші зовнішні ресурси, то такі тести виконуються доволі швидко.

Для автоматизації етапу модульного тестування існують наступні можливості. Перша із них — це автоматичний запуск тестів кожного разу після завантаження розробником свого коду у систему контролю версій, що дозволяє завжди знати, яка саме порція нового коду спричинила проблеми. Але такий варіант автоматизації має і свої недоліки — оскільки тести будуть запускатись доволі часто, це створюватиме додаткове навантаження на сервери.

Інша можливість автоматизації — виконання тестів перед розгортанням сервісу на сервер, де він має працювати. Таким чином ми зможемо виявити показати недоліки ще до того, як сервіс почне використовуватись.

Інтеграційне тестування

Інтеграційне тестування — це тестування всієї системи в цілому, на відмінну від модульних тестів, які тестують окремі підсистеми. На цьому етапі підсистеми усієї системи уже протестовані та в основному відбувається тестування саме зв’язків між цими підсистеми, які в модульних тестах закривались за допомогою допоміжних об’єктів.

Таке тестування також може відбуватись вручну або бути автоматизованим. Ручне тестування зазвичай представляє собою роботу тестувальників, які виконують ряд певних дій вже на працюючому сервісі, який був розгорнутий в певному тестовому середовищі, і перевіряють, що після послідовності операцій, у виконанні яких приймають участь різні підсистеми сервісу, все відбувається відповідно до очікувань.

Щодо автоматизації, то зазвичай автоматизують основні процеси. Як приклад, можна навести такі: реєстрація користувача в системі, вхід

користувача до системи, вихід та інші базові операції, які найчастіше виконують користувачі у системі. Такі тести зазвичай завантажують сервіс та автоматично ініціюють натискання тих чи інші кнопок для перевірки коректності їх роботи та отриманого результату. Та все ж є складніші ситуації, що важко піддаються автоматизації, тож їх змушені перевіряти вручну.

Розгортання

Розгортання сервісу — це використання раніше збудованих і протестованих файлів сервісу для його запуску на виділених серверах або в хмарі для подальшого використання кінцевим користувачем.

На цьому етапі файли сервісу завантажуються вручну або через спеціальні допоміжні програми на сервер або у хмарну систему, де вони запускаються з певними конфігураційними налаштуваннями. Цей етап також можна легко автоматизувати. Як результат — файли проєкту, що знаходяться на сервері, на якому відбувалась збірка сервісу, відправляються до цільового серверу або хмари та автоматично там запускаються. Також на цьому етапі автоматично запускаються різні додаткові служби, наприклад, журналювання роботи з базою даних, до якої немає доступу у звичайних розробників, інші допоміжні сервіси, необхідні основному сервісу [1].

1.2. Фазовий підхід до розгортання

У найпростішому випадку розгортання виконується на тому ж комп’ютері, на якому збірка й зберігається. При промисловій розробці використовується фазовий підхід та виділяються наступні види оточень [2]:

- локальне (Local environment);
- оточення розробки (Development environment);
- інтеграційне оточення (Integration environment);
- тестове оточення (Testing environment);
- дзеркало оточення кінцевого користувача (Staging environment);
- оточення кінцевого користувача (Production environment).

Локальне (Local environment): оточення в якому відбувається розробка, найчастіше це просто комп’ютер розробника.

У контексті управління версіями, особливо при участі великої кількості розробників, проводяться більші тонкі відмінності: розробник має робочу копію вихідного тексту на своїй машині та зміни вносяться до репозиторію, фіксуючись у певній гілці.

Оточення розробки (Development environment): оточення на окремому сервері, де відпрацьовуються та випробовуються зміни. Це оточення також називають «пісочницею (sandbox)», оскільки розробники можуть експериментувати з кодом без жодного впливу на інші середовища.

Інтеграційне оточення (Integration environment): це основа для побудування неперервної інтеграції. Безперервна інтеграція (CI) спрямована на автоматизовану перевірку інтеграції між змінами внесеними розробником та іншим кодом. У цей процес може входити статичний аналіз коду на уразливості та невідповідність усталеним практикам або правилам розробки, збірка програми й автоматизоване тестування з динамічною перевіркою на відомі вразливості.

Оточення тестування (Testing environment): Ціллю тестового оточення полягає в тому, щоб дозволити людям, які проводять тестування, пропускати новий і змінений код або через автоматизовані перевірки, або через ручні методи тестування. Після того, як розробник пропускає новий код и конфігурації через модульне тестування в середовищі розробки, код переноситься на одне або декілька тестових середовищ.

Якщо тестування пройде успішно, тестове середовище або фреймворк безперервної інтеграції, яке контролює тести, може автоматично перемістити код у наступне оточення розгортання.

Дзеркало оточення кінцевого користувача (Staging environment): це оточення для тестування, яке в точності схоже на оточення кінцевого користувача. Основним призначенням Staging-оточення полягає в тестуванні всіх сценаріїв установки, конфігурації, додавання скриптів і процедур та їх

налаштувань, до того, як вони будуть застосовані у Production-оточенні. Це гарантує, що всі оновлення, навіть мінімальні та незначні, Production-оточення будуть завершені якісно, без помилок і в мінімальний термін. Іншим важливим призначенням Staging-оточення є тестування продуктивності, тестування навантаження, оскільки такі процедури надзвичайно впливають на ресурси та є неприйнятними для оточення кінцевого користувача.

Оточення кінцевого користувача (Production environment) – це оточення з яким взаємодіють кінцеві користувачі. Розгортання в виробничому середовищі є найбільш чутливим кроком.

Фазовий підхід має багато варіацій, які відрізняються кількістю оточень, яка залежить від практик компанії та від масштабів проєкту. Та всі вони починаються з локального оточення і завершуються оточенням кінцевого користувача.

Поширеними є такі, що складаються із 3-х оточень: development, testing, production (DTP) [3]; та 4-х: development, testing, acceptance and production (DTAP) [4].

DTP архітектура.

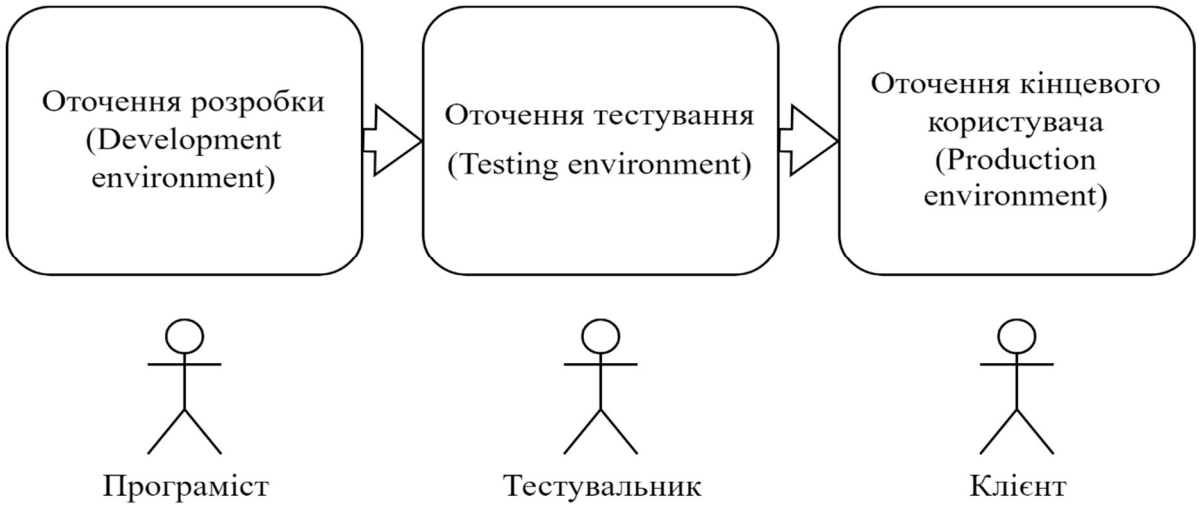

Рисунок 1.1 — DTP архітектура.

Ця архітектура необхідна для забезпечення стримувань і противаг, необхідних для ефективної роботи виробничого серверного середовища з

високою доступністю. Основна концепція оточень DTP проста. Розробники тестують свій код на оточені, яке призначене для розробки, щоб побачити, як працюватиме тестований код з іншим кодом.

Як тільки програміст переконається, що його частина коду готова до роботи, вона переміщується на проміжне оточення (оточення тестування). Проміжне оточення є дзеркальною копією оточення кінцевого користувача; його основною метою — тестування завершеного нового функціоналу (коду) на дзеркальній копії кінцевого середовища, щоб упевнитися, що він не порушує роботу існуючих частин проєкту у кінцевому оточенні.

Ніяка фактична розробка коду не повинна відбуватися на проміжному оточенні — тільки незначне налаштування параметрів ОС або налаштування параметрів та програм.

Проміжне оточення — це останній крок перед тим, як проєкт буде готовий до розгортання на Production-оточенні. У багатьох компаніях після проміжного оточення відбувається процес затвердження з метою отримання загальної згоди всіх учасників до запуску проєкту в оточенні кінцевого користувача. Коли проєкт отримає схвалення, він буде переміщений до Production-оточення. Якщо переміщення пройшло успішно, сервіс стає частиною оточення кінцевого користувача.

Оточення тестування у даній архітектури виступає у ролі пісочниці та у ролі оточення для тестування, що не є дуже добрим, тому що на нього постійно доставляються певні частини невідтестованого коду від розробників, що призводить до нестабільної роботи оточення. А у випадку нестабільності оточення неможливо провести якісне тестування нового функціоналу. А отже — через недостатню якість тестування виникає ризик перенесення помилок до оточення кінцевого користувача.

DTAP архітектура.

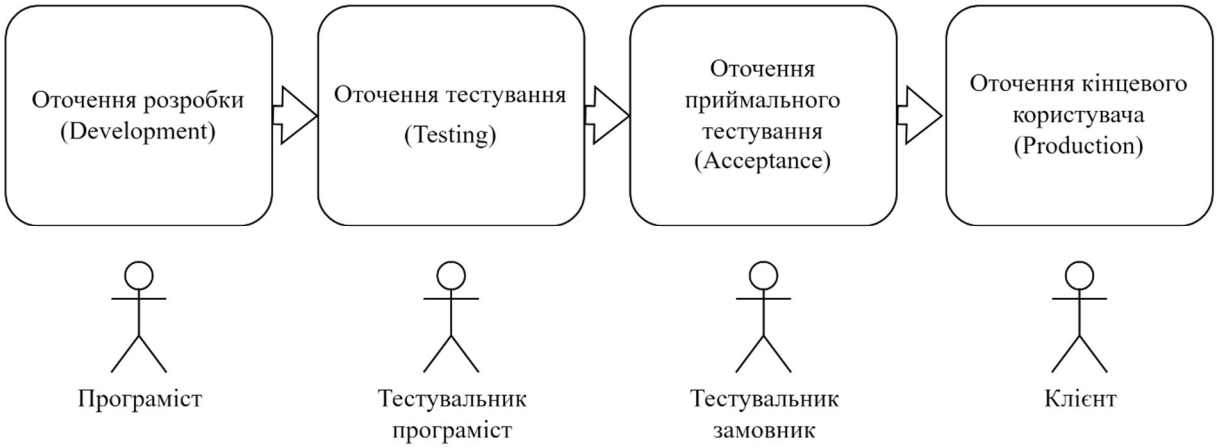

Рисунок 1.2 — DTAP архітектура.

DTAP архітектура має 4 оточення розгортання програмних сервісів, кожне з яких має своє призначення: development, testing, acceptance, production.

Development — оточення розробки. У більшості сучасних систем розробка відбувається на персональному комп'ютері окремого розробника. Це нормально за умови, що проєкт забезпечений належною системою контролю версій. Це середовище може не мати можливостей для тестування.

Testing — оточення тестування. Як тільки розробник упевнився, що його частина коду готова, вона копіюється у середовище тестування з метою перевірки, чи все працює так як очікується. Тут відбувається модульне тестування, потім відбувається інтеграційне та регресійне тестування, щоб переконатися, що всі частини працюють та ніякий функціонал, вже присутній раніше не був зламаний.

Acceptance — оточення приймального тестування. Це оточення повинно за налаштуваннями повністю дублювати оточення Production, тому його часто називають Staging. Деякі команди повторно запускають комплекс інтеграційного та регресійного тестування на цьому етапі у якості останньої перевірки, перш ніж клієнт побачить код. Приймальний тест для сервісів має бути там, де клієнт дійсно бачить код може протестувати його, використовуючи свій власний план та засоби тестування.

Production — оточення кінцевого користувача. Це останній етап розробки програмного сервісу. Якщо замовник продукту приймає його, він розгортається у оточенні кінцевого користувача та стає доступним усім споживачам.

Також існує проблема порушення термінів випуску нових версій. Оточення Staging використовується у 2 напрямках: тестування та стабілізації. Команда тестувальників приступає до роботи тільки на оточенні Staging, тому що тестувати на оточенні розробки немає сенсу, через те, що воно нестабільне, так як розробники постійно оновлюють його. Отже, коли тестувальник знаходить помилку, то розробник її виправляє, але швидко доставити виправлений код до оточення Staging не видається можливим, оскільки код має пройти усі попередні оточення та певні процедури затвердження. Це, разом із недостанім рівнем автоматизації, і призводить до затримок випуску нових версій зокрема та й усього процесу розробки загалом.

1.3. Контроль якості розгортання

Згідно з визначенням Інституту Управління Проєктами (*Project Management Institute, PMI*) проєкт це тимчасова діяльність, спрямована на створення унікального продукту, послуги або результату [5], яка вимагає активного керування ресурсами, вартістю, якістю, інтеграцією, розкладом та комунікацією. З цього випливає концепція так званих «воріт якості» («quality gates»,QG), що дозволяє просувати тільки ті артефакти, які відповідають наперед заданим критеріям, базованим на певних властивостях артефактів та проєкту загалом і можуть бути оцінені. Реалізація «воріт якості» може бути як ручною так і повністю автоматизованою, від найпростішої — контрольного списку або відомості результатів перевірки до складної автоматизованої системи тестування коду [6].

«Ворота якості» — вагома частина DevOps — це саме те, що контролює завершеність, узгодженість та цілісність. Кожна компанія має створити їх

таким чином, щоб вони максимально відповідали пріоритетам та цілям проєкту.

Останнім часом все частіше до процесів CI/CD додається ще процес Continuous Security («безперервна безпека»), що намагається підвищити пріоритет питання безпеки та позиціонувати її як ключовий аспект під час розробки, упродовж життєвого циклу розгортання та навіть після розгортання у оточенні кінцевого користувача. Як нефункціональній вимозі, безпеці часто надають низький пріоритет, навіть ненавмисно. Тому Continuous Security намагається впровадити легкий та розумний підхід до виявлення вразливостей [35].

Так, застосовуючи «ворота якості», компанія може бути впевнена, що код, створений розробниками, ніколи не буде використовуватись, якщо він не відповідає базовим стандартам (до прикладу, OWASP Top Ten [7], або PCI DSS [8]). «Ворота якості» зменшують шанси релізу низькоякісного або вразливого коду без зупинки циклу розробки.

Також запровадження «воріт якості» покращує процес комунікації в ході розробки проєкту, скорочує цикл розробки, дозволяє досягти більшого рівня успішності, фокусуючись тільки на якісних релізах [9].

Часто рекомендують впроваджувати «ворота якості» між усіма фазами розгортання сервісів. Так, архітектура DTAP буде мати вигляд, як на рис. 1.3.

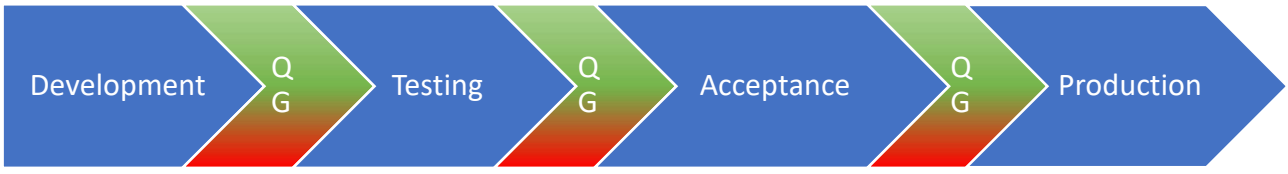

Рисунок 1.3 — Схема DTAP архітектури з "воротами якості"

Але тут постає проблема того, що кожна компанія має свій власний набір звичок, поведінки, поводження, цінностей, пріоритетів та усталених практик менеджменту, тому деяким компаніям може бути надзвичайно складно адаптувати нові політики та інструменти [33]. Отже, до процесу

впровадження «воріт якості» слід підходити виважено. Необхідно визначити, чи кожен екземпляр «воріт» може бути повністю автоматизований, чи необхідна оцінка людиною? На яких етапах розробки або між якими фазами розгортання їх включати? Надмірне використання «воріт якості» може загальмувати процес розгортання, особливо коли рішення про їх успішне або неуспішне проходження приймається людиною. В такому разі слід визначити критичні фази, що беззаперечно потребують оцінки якості розгортання.

Висновки до розділу

В цьому розділі були розглянуті етапи розгортання програмних сервісів, фазовий підхід до розгортання та контроль якості розгортання. Визначено, що основними є DTP та DTAP архітектури, що використовують кілька незалежних оточень, переміщуючи артефакти між ними. Такий підхід дозволяє паралельно розробляти, тестувати, налагоджувати та експлуатувати різні версії програмних сервісів ізольовано одну від іншої. Також визначено, що для випуску якісних програмних сервісів існує задача просувати між фазами тільки ті артефакти, які відповідають наперед заданим критеріям якості. Вирішувати таку задачу покликані «ворота якості», що зменшують шанси релізу низькоякісного або вразливого коду, покращують процес комунікації в ході розробки проєкту, скорочують цикл розробки, дозволяють досягти вищого рівня успішності, фокусуючись тільки на якісних релізах. Інструменти, що будуть розглянуті далі, дозволять нам в реалізувати ізольовані оточення та автоматизувати процес розгортання, включаючи автоматизацію просування артефактів між оточеннями, реалізацію автоматичних «воріт якості» та інші задачі.

Розділ 2. ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

В даному розділі розглянемо існуючі технології та рішення, що можуть бути використані для реалізації завдання. Основну увагу приділимо платформі та інструментам розгортання.

2.1. Огляд існуючих рішень для автоматизації розгортання

Docker

Docker — це технологія віртуалізації лише корисного навантаження — самих програмних сервісів. Docker дозволяє запакувати та запускати застосунки у ізольованому оточенні — контейнері, а також зберігати копії цих контейнерів та їх образи для подальшого поширення, співпраці з колегами. Docker-контейнери можна копіювати та розгортати будь де, від комп'ютера розробника для аналізу їх роботи на предмет пошуку помилок, зміни функціоналу та ін., до хмари — для використання кінцевими користувачами.

Є кілька проблем, які Docker вирішує. Перша із них — те, що VM є досить громіздким обчислювальним ресурсом. Загалом віртуальна машина — це екземпляр операційної системи, який працює під керівництвом та поверх гіпервізора — програмного забезпечення, що дозволяє паралельно запускати декілька операційних систем та керувати ними. В свою чергу гіпервізор запускається під керуванням іншої операційної системи (або сам є такою ОС) та працює на фізичному обладнання. Саме ж корисне навантаження, програмний сервіс, працює всередині VM. Це представляє певні виклики щодо швидкодії та продуктивності як під час роботи сервісу, так і під час запуску та зупинки, оскільки необхідно надавати певні обчислювальні ресурси для операцій однієї та більше ОС.

Отже, віртуалізація має ряд недоліків, що мають бути вирішені: проблема створення більш легкого та гнучкого обчислювального ресурсу. З Docker ви можете мати лише єдиний гіпервізор, що встановлений поверх ОС без потреби інсталювати та налаштовувати необхідну кількість віртуальних машин. Контейнери Docker запускаються за секунди, на фізичній або

віртуальній машині їх можна мати велику кількість і тим самим отримати досить велику масштабованість.

Docker значною мірою покладається на дві частини технології ядра Linux. Перший називається просторами імен. Якщо ви запускаєте новий процес на ядрі Linux, то ви здійснюєте системний виклик до просторів імен: «Я хочу створити новий процес». Якщо ви хочете створити новий мережевий інтерфейс, ви звертаєтесь до простору імен мережі. Ядро призначає вам простір імен, в якому існуватиме процес і будь-які інші потрібні ресурси. Наприклад, він може мати певний доступ до мережі, доступ тільки до деяких частин файлової системи, доступ лише до частини оперативної пам'яті або певної частини процесорного часу ЦП.

Для створення контейнеру Docker загалом виконує ряд звернень до ядра Linux, щоб отримати дозвіл на дії, що потрібні для роботи контейнеру, такі як доступ до файлової системи, мережевих ресурсів та інших подібних потреб. Контейнер повинен мати доступ до цієї конкретної файлової системи, доступ до центрального процесора та пам'яті, а також доступ до мережі, і він повинен знаходитися всередині власного простору імен. Всередині цього виділеного простору імен процес не може побачити інших просторів імен і процесів у них.

Другий компонент технології, який Docker використовує, називається контрольними групами, або групами. Вони призначені для управління ресурсами, доступними для контейнера. Це дозволяє робити такі речі, як обмеження конкретних контейнерів певними значеннями ресурсів, що доступні йому для роботи (прикладом можна навести отримання контейнером лише 128 МБ оперативної пам'яті при доступних гігабайтах на фізичному комп’ютері), те саме стосується і мережевих ресурсів, ресурсів жорсткого диску, процесорного часу (можливо навіть виділяти по половині ядра процесору на ряд контейнерів). Ви можете додавати та вилучати ресурси за потреби, і це робить його досить потужним, щоб мати можливість детально

керувати контейнером приблизно так само, як і з інтерфейсом віртуальної машини.

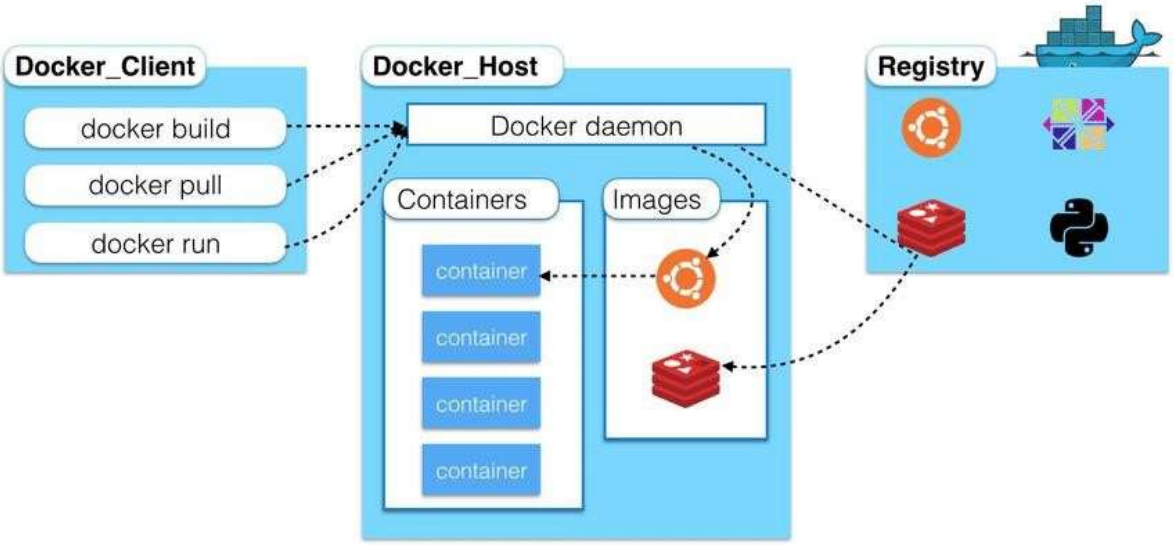

Рисунок 2.1 — Архітектура Docker.

Kubernetes

Kubernetes — це платформа з відкритим вихідним кодом для управління контейнеризованими робочими навантаженнями та супутніми службами. Її основні характеристики — кросплатформенність, розширюваність, успішне використання декларативної конфігурації та автоматизації [11]. Kubernetes використовується для організації груп контейнерів, що представляють екземпляри програм, які відокремлені від машин, на яких вони працюють. Оскільки кількість контейнерів у кластері збільшується до сотень або тисяч екземплярів, а окремі компоненти програм розгортаються як окремі контейнери, Kubernetes приходить на допомогу, забезпечуючи платформу для розгортання, управління, автоматичного масштабування, високої доступності та відповідних завдань.

Контейнери — це хороший спосіб об’єднати та запустити ваші сервіси. У виробничому середовищі вам потрібно керувати контейнерами, в яких запущені програми, і переконатися, що немає простоїв. Наприклад, якщо контейнер зупиняється, може бути необхідність у запуску іншого екземпляру контейнера. Ось так на допомогу приходить Kubernetes. Kubernetes надає

фреймворк для стійкого запуску розподілених систем. Він дбає про масштабування та відновлення після відмови для ваших сервісів, надає схеми розгортання тощо. Наприклад, Kubernetes може легко керувати розгортанням сервісів для вашої системи, або якщо був втрачений зв’язок із одним із контейнерів, або навіть вузлом, то Kubernetes знає, які контейнери та з якими параметрами були запущені на цьому хості та за лічені секунди запустить всі втрачені контейнери на інших вузлах. Хоча дані із оперативної пам’яті тих контейнерів будуть втрачені, та кількість працюючих сервісів не буде зменшена. Користувачу потрібно лише буде повторити свій запит до сервісу задля отримання відповіді. Весь цей процес можна автоматизувати та втручатись в нього лише для збільшення кількості контейнерів або для аналізу результатів роботи системи за певний час з метою прийняття відповідних рішень.

Kubernetes дозволяє вирішувати різні завдання, пов'язані з оркестрацією контейнерів, такі як: горизонтальне та вертикальне масштабування контейнерів, розподіл робочого навантаження між кількома контейнерами одного сервісу, розміщеними на різних вузлах.

Kubernetes дотримується традиційного типу архітектури клієнт-сервер, де головний вузол (control plane) відповідає за прийняття рішень. Користувачі можуть взаємодіяти з вузлом control plane через REST API, вебінтерфейс та інтерфейс командного рядка (CLI). Головний вузол взаємодіє з робочими вузлами (worker nodes), на яких фактично розміщуються контейнери (рис. 2.2) [11].

Деякі із загальноприйнятих термінів, що використовуються в екосистемі Kubernetes:

- контейнери — це одиниці упаковки, що використовуються для об’єднання бінарних файлів програм разом із їх залежностями, конфігурацією, фреймворком та бібліотеками;

- поди (podes) — це блоки розгортання в екосистемі Kubernetes, які містять один або кілька контейнерів разом на одному вузлі. Група контейнерів може працювати разом і спільно використовувати визначені ресурс;
- вузли (nodes) — це представлення однієї машини в кластері, на якому запущені програми Kubernetes. Вузлом може бути фізична або віртуальна машина;
- кластери (clusters) — це кілька вузлів, з'єднаних між собою, щоб утворити кластер для об'єднання ресурсів, які спільно використовуються подами, розгорнутими на кластері;
- постійне сховище (persistent volume) — оскільки контейнери можуть динамічно приєднуватися і залишати обчислювальне середовище, локально збереженні дані може бути втрачені. Постійне сховище допомагає зберігати дані контейнера на постійній основі.

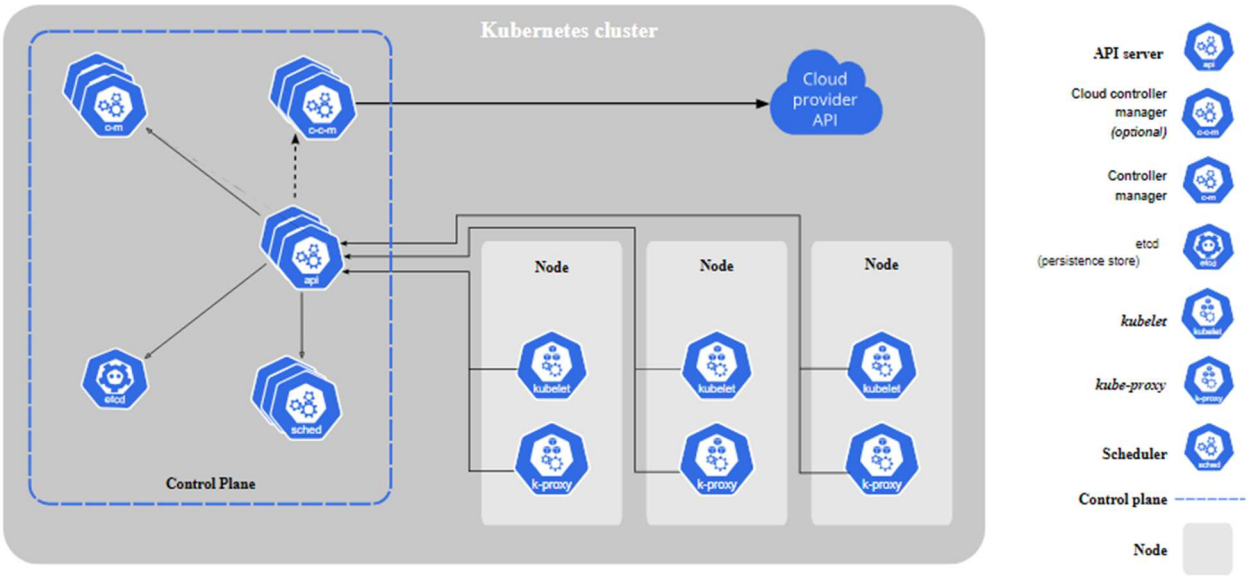

Рисунок 2.2 — Архітектура Kubernetes.

Docker Swarm/Docker compose

Docker Swarm — це альтернативний механізм оркестрації контейнерів від Docker, який координує розміщення та управління контейнерами між декількома хостами Docker Engine. Docker Swarm дозволяє вам оперувати безпосередньо множиною контейнерів, замість того, щоб оперувати кожним

контейнером окремо. Архітектура Docker Swarm складається з двох типів вузлів — вузли менеджера та робочі вузли.

- Вузол — машина, яка запускає екземпляр Docker Engine.
- Swarm — скупчення екземплярів Docker Engine.
- Вузол менеджера — розподіляє та планує завдання на вузли worker та підтримує роботу кластер. Вузли менеджера також можуть додатково запускати служби для вузлів worker.
- Робочі вузли — це екземпляри Docker Engine, відповідальні за запуск програм у контейнерах.
- Сервіс — це образ мікросервісу, наприклад веб-сервер або сервер баз даних.
- Завдання — Служба, запланована для запуску на вузлі worker.

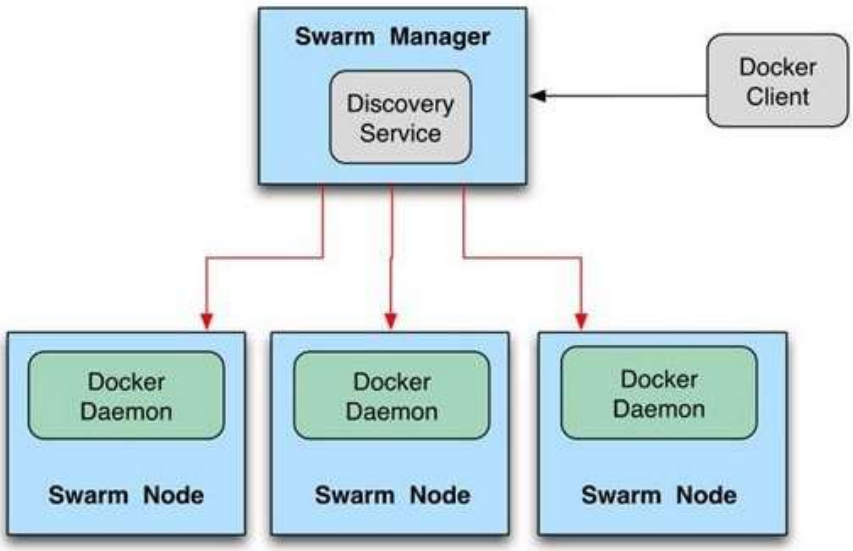

Рисунок 2.3 — Архітектура Docker Swarm [12]

Образ Docker створюється за допомогою Docker-файлу. Docker-файл — текстовий файл, що містить усі команди для створення Docker-образу та запуску коду.

Docker-образ дозволяє широкій групі людей створювати однакове середовище. Є можливість поділитися Docker-образом, що можна легко

досягти за допомогою Docker Hub. Можна завантажити образ до Docker Hub і надати доступ до нього через публічний репозиторій або ж приватний, створивши персональний корпоративний обліковий запис.

З Docker Hub користувачі можуть завантажувати образи та запускати контейнери, які відповідають оригінальному контейнеру Docker, створеному попередньо, тим самим отримуючи середовище відповідне оригіналу.

Часто є потреба у використанні двох або більше контейнерів в рамках одного сервісу, типовим прикладом є зв'язка веб-сервера та бази даних. Створення, запуск та підключення контейнерів з окремих файлів Docker є складним і займає багато часу, і тут на допомогу приходить Docker Compose, що полегшує роботу з кількома контейнерами одночасно.

Docker Compose — інструмент для опису та запуску Docker-застосунків, що складаються із кількох контейнерів. Для опису конфігурацій програмних сервісів використовується мова YAML. Потім, усього лиш однією командою ви створюєте та запускаєте всі сервіси, описані у вашій конфігурації. Docker Compose працює у всіх оточеннях (розробки, тестування, кінцевого користувача та ін.), а також у CI процесах. Він має команди для керування вашими сервісами: запускати, зупиняти та перестворювати сервіси, отримувати статуси запущених сервісів та виводити журнали їх роботи.

Ключові особливості Docker Compose [13]:

- дозволяє отримати кілька ізольованих оточень на одній машині;
- зберігає дані контейнерів
- перестворює тільки ті контейнери, конфігурація яких змінилась
- підтримує змінні та переміщення «композицій» між оточеннями.

Apache Mesos

Apache Mesos, загальнокластерний менеджер ресурсів, широко застосовується в кількох хмарах та центрах обробки даних. Mesos прагне забезпечити високе використання кластерів за допомогою деталізованого спільного планування ресурсів та справедливості ресурсів серед кількох

користувачів завдяки розподілу на основі домінуючої справедливості ресурсів (Dominant Resource Fairness)[14].

DRF враховує різні типи ресурсів (ЦП, пам’ять, дисковий ввід / вивід), що вимагаються кожною програмою, та визначає частку кожного ресурсу кластера, який може бути призначений розгорнутим сервісам. Mesos прийняв дворівневу політику планування: DRF для розподілу ресурсів на конкуруючі рамки та планування рівня завдань кожною структурою для ресурсів, виділених на попередньому кроці. Mesos об'єднує всі ресурси в кластері та дозволяє чітко розподілити ресурси, дозволяючи та застосовуючи декілька застосунків (так званих Mesos framework) для спільного планування своїх завдань на віртуальних машинах/вузлах. Mesos використовує DRF для розподілу ресурсів у фреймворки, а потім фреймворки використовують алгоритми планування для планування завдань у межах виділених ресурсів.

Фреймворки, які широко використовуються для спільної роботи з Apache Mesos, — це Apache Aurora для довготривалих служб, Mesosphere Marathon для контейнерної оркестрації та Chronos для cron-завдань.

Apache Mesos складається з трьох основних компонентів Mesos Master, Mesos Agent та Mesos Framework. Mesos Master керує ресурсами під час узгодження ресурсів між агентами Mesos. Агенти Mesos відповідають за виконання запитуваних завдань із наявними ресурсами. Усі доступні вільні ресурси від одного агента Mesos включені в пропозицію ресурсів. Фреймворки Mesos самостійно приймають рішення щодо планування, щоб зіставити одне або декілька завдань із пропонованими їм ресурсами (рис. 2.4) [15].

На кроці 1 агенти Mesos періодично анонсують Mesos Master щодо доступних вільних ресурсів (наприклад, процесора, оперативної пам'яті, диска), які можна використовувати для запуску завдань. На кроці 2 модуль розподілу Mesos Master починає пропонувати доступні ресурси для активних фреймворків.

Протягом кожного циклу розподілу Mesos Master сортує фреймворки на основі алгоритму DRF, і фреймворк з найменшою домінуючою часткою

першим отримує пропозицію. Після розподілу ресурсів індивідуальна система може вирішити, приймати чи відхиляти пропозицію. Відхилена пропозиція повертається до пулу ресурсів і буде запропонована іншим структурам у наступному циклі розподілу.

На кроці 3, якщо пропоновані ресурси задовольняють запити ресурсу фреймворку, фреймворк створює список завдань щодо пропозицій. Виходячи з політики індивідуальної структури, одне або декілька завдань можуть поєднуватися з однією пропозицією ресурсів. На кроці 4 Mesos Master запускає їх на вибраних агентах, і якщо необхідні ресурси перевищують доступні ресурси в пропозиції, Mesos Master відповідає помилкою. В іншому випадку він надсилає набір завдань окремим агентам, що відповідають пропозиції.

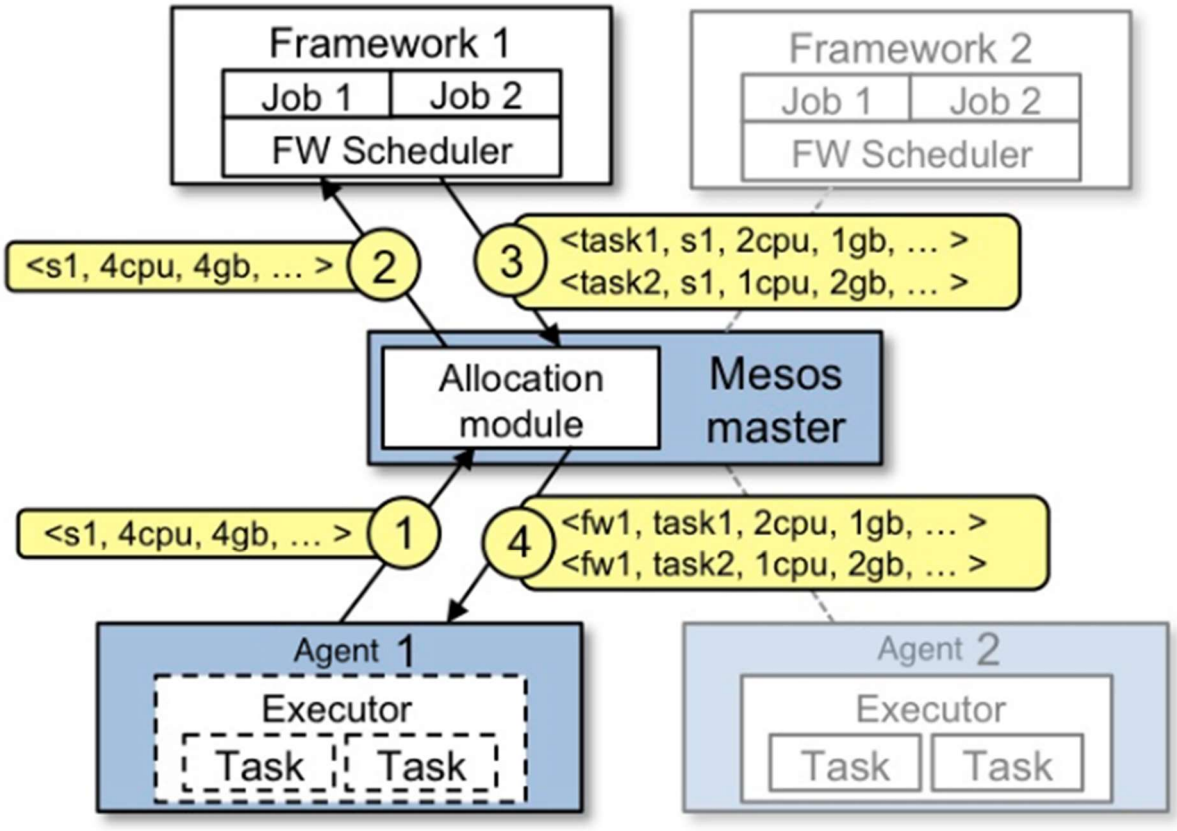

Рисунок 2.4 — Схема узгодження пропозиції ресурсів в Mesos

Rancher

Rancher — платформа для управління контейнерами, створена для компаній, які активно використовують контейнери для доставки своїх сервісів кінцевим споживачам. Rancher спрощує роботу з кластерами Kubernetes, дозоволяє відповідати усім IT-вимогам та підсилює можливості команд DevOps [16].

Rancher забезпечує інтуїтивно зрозумілий користувацький інтерфейс для інженерів DevOps щодо управління сервісами. Користувачеві не потрібно мати глибоких знань про концепції Kubernetes, щоб почати використовувати Rancher. Каталог Rancher містить набір корисних інструментів DevOps. Rancher сертифікований широким набором продуктів хмарних екосистем, включаючи засоби безпеки, системи моніторингу, реєстри контейнерів, драйвери для зберігання та мережі.

Рис. 2.5 ілюструє роль, яку Rancher відіграє в IT та DevOps організаціях. Кожна команда розміщує свої сервіси у загальнодоступних або приватних хмарах, які вони обрали. IT-адміністратори отримують видимість та застосовують політику для всіх користувачів, кластерів та хмар.

На рис. 2.5 зображена інсталяція сервера Rancher Server, яка управляє двома кластерами Kubernetes: один створений RKE (Rancher Kubernets Engine), інший — Amazon EKS (Elastic Kubernetes Service).

Rancher вже мав підтримку AD (Active Directory), LDAP та SAML (Security Assurance Markup Language, мова розмітки забезпечення безпеки), але Rancher 2.0 пішов далі та тепер включає такі функції, як розширена підтримка Helm, краща інтеграція з інструментами CI/CD, засоби інформування, попередження та централізований журнал. Завдяки застосуванню Helm Rancher розширив свої можливості автоматизації.

Rancher 2.0 також розширив свої агностичні можливості стосовно хмар. Спочатку він був розроблений щоби бути незалежним від хмар на рівні оркестрації, намагаючись об’єднати користувачів Kubernetes, Docker Swarm та власного Cattle. Тепер, коли Kubernetes зарекомендував себе як галузевий

стандарт для оркестрації контейнерів, Rancher перейшов на використання Kubernetes як єдиної платформи.

Rancher 2.0 дозволяє керувати кластерами Kubernetes на платформах усіх основних хмарних постачальників, а саме: GKE від Google, AKS від MS Azure та EKS від Amazon, а також кластерами Kubernetes інших вендорів, таких як Canonical. Rancher робить це шляхом абстрагування автентифікації від специфіки провайдерів, лишаючи кластери Kubernets працювати на рушіях та платформах різних провайдерів, та надаючи користувачам єдину консоль для виконання усіх операцій зі всіма Kubernetes кластерами [18].

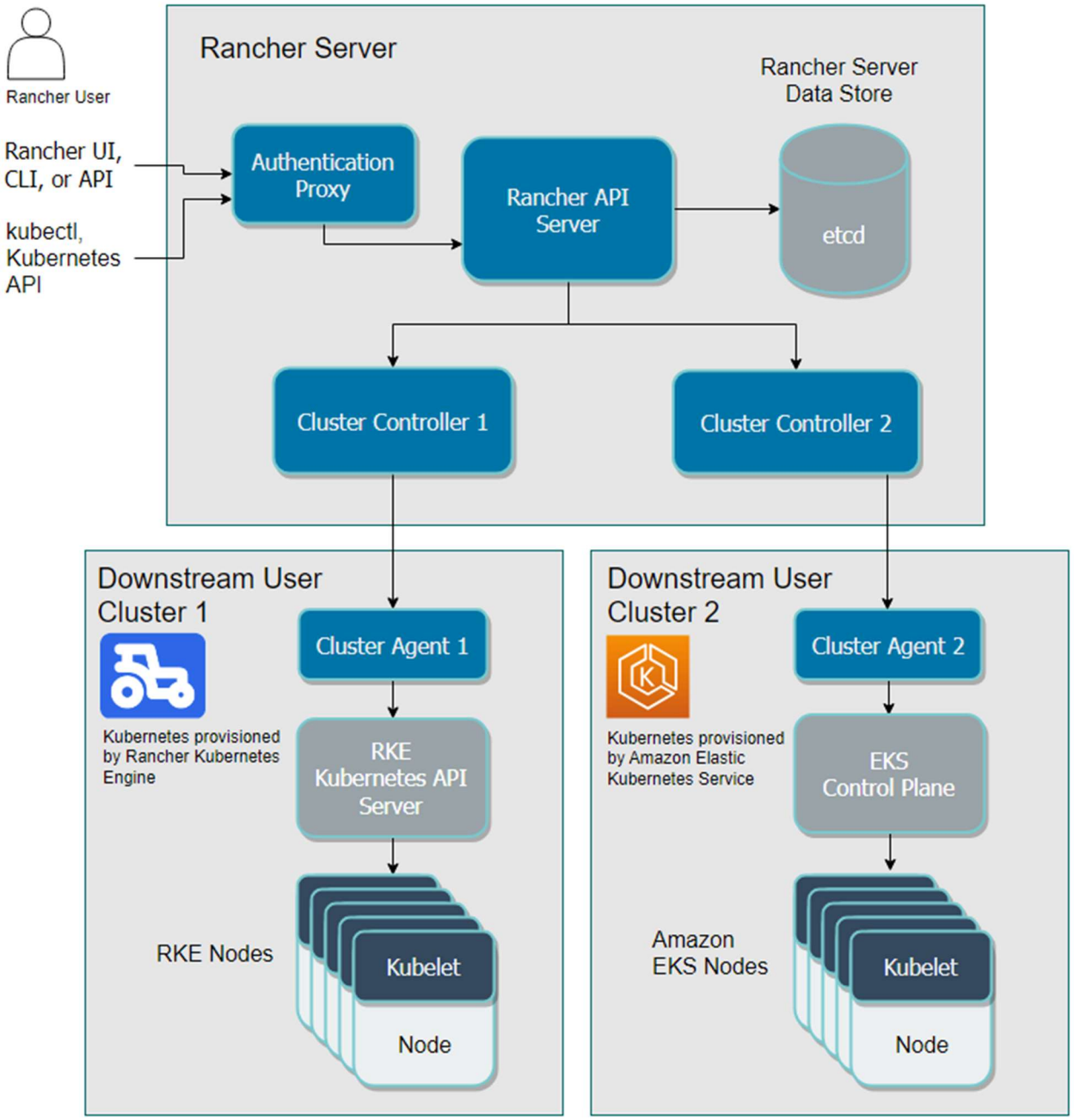

Рисунок 2.5 — Архітектура Rancher [17].

Для найкращої продуктивності та безпеки рекомендується використання виділеного кластеру Kubernetes для сервера управління Rancher. При цьому запускати робочі навантаження користувачів у цьому кластері не рекомендується. Після розгортання Rancher можна створювати або імпортувати вже розгорнуті кластери, на яких буде проводитись запуск робочих навантажень.

Сервер Rancher, незалежно від способу встановлення, завжди повинен працювати на вузлах, які відокремлені від підлеглих кластерів. Якщо Rancher встановлено на кластері Kubernetes з високою доступністю, він повинен працювати на кластері, відокремленому від кластера, яким він керує.

Отже, після того, як Kubernetes став найпопулярнішим інструментом для автоматичного розгортання сервісів, Rancher почав використовувати Kubernetes для своєї власної роботі, що, по суті, зробило Rancher інструментом розширення функціональність Kubernetes, допомагаючи зручніше та оперативніше керувати різною кількістю кластерів.

Octopus

Octopus Deploy — інструмент розгортання. Він бере пакети та артефакти, створені вашим сервером збірки та розподіляє їх до різних цілей, як то: сервери Windows, Linux, хмари Azure, AWS, GCP або кластери Kubernetes, у безпечному та послідовному процесі.

Незалежно від того, використовується власний сервер Octopus Server чи Octopus Cloud, веб-портал Octopus — це місце, де відбувається управління інфраструктурою, розгортаннями, випусками, операційними процесами, отримання доступу до вбудованого сховища, надання команді доступу до проєктів та створення автоматизованих процесів розгортання.

Інфраструктура Octopus представлена у вигляді середовищ, цілей розгортання та виконавців. Так, можливе створення середовищ Octopus у відповідності до архітектури розгортання компанії (наприклад, DTAP).

Octopus Deploy робить управління випусками простим, перевіреним та сумісним. Управління випусками спрощує процеси випуску завдяки послідовному просуванню коду у різних середовищах, затвердженню вручну та приміткам про випуск. Інформаційна панель надає команді загальний огляд того, що де розгорнуто.

Octopus є центральним місцем для команд задля управління, контролю, аудиту, планування та виконання запуску. Є можливість бачити, коли та який

runbook запускався останнім, будь які зміни в ньому, можна запускати один і той же runbook у різних середовищах. Члени команди можуть легко знайти журнал запуску, отримати результати останнього запуску та статус успішності.

Завдяки великій кількості вбудованих шаблонів кроків, Octopus спрощує найскладніші розгортання, незалежно від місць призначення розгортання програмних сервісів. Процес розгортання в Octopus використовує змінні, тобто один і той же процес розгортання може бути використаний у розробницькому, тестовому та середовищі кінцевого користувача. Octopus підтримує розширені стратегії розгортання, такі як “rolling”, “blue/green”, “canary” та “multitenant”, якщо є необхідність розгортання на кількох кінцевих клієнтах.

Автоматизація runbook надає можливість контролювати інфраструктуру та сервіси. Можлива автоматизація таких операційних задач, як планове технічне обслуговування та аварійне відновлення. Octopus runbooks мають усі необхідні безпекові налаштування та дозволи у інфраструктурі, на якій вони працюють, тому будь хто з команди може отримати авторизацію на виконання runbook, яка буде успішно виконана без додаткових запитів. В свою чергу, отримання авторизації та запуск runbook залишає повний слід у журналах для подальшого аудиту виконаних дій [19].

Рис. 2.6. ілюструє процес розгортання за допомогою Octopus, використовуючи сервер збірки TeamCity [20].

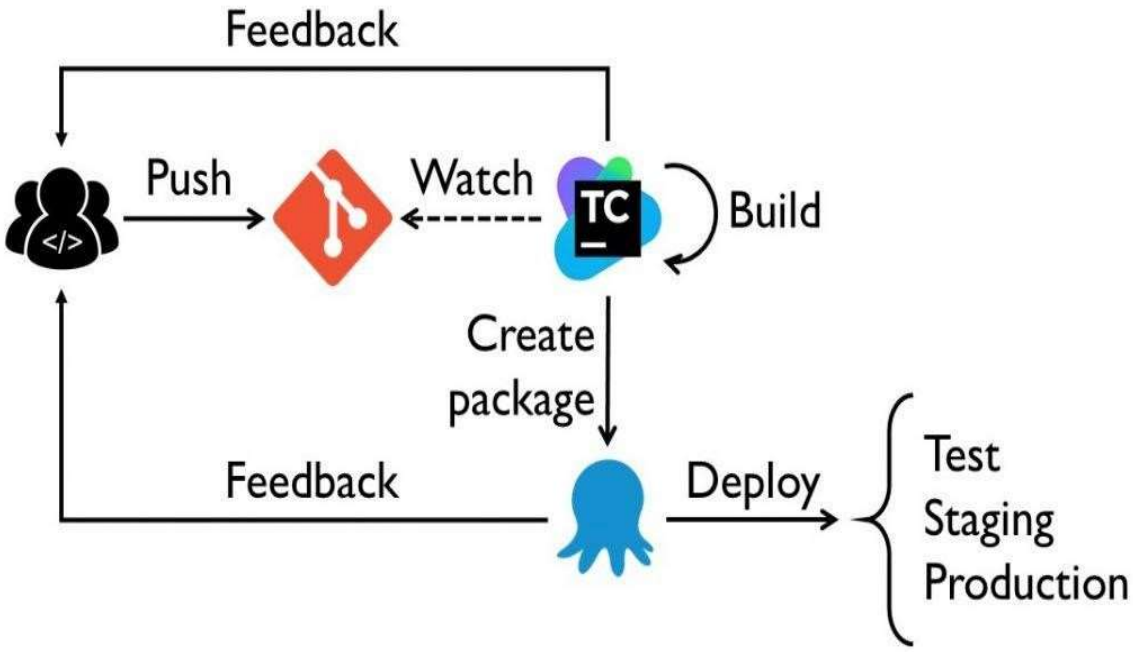

Рисунок 2.6 — Процес розгортання за допомогою Octopus.

Keptn

Keptn — продукт корпоративного рівня для централізованого управління процесами постійного розгортання (CD) та автоматизації операцій. Keptn створювався з метою допомогти компаніям використовувати усі переваги хмарних систем, надавши їм готову CD платформу, що включає автоматизоване тестування, контроль якості (QG) та автоматизацію операцій за допомогою координації інформації та інструментів корекції. Keptn допомагає створити на будь якому Kubernetes кластері систему управління життєвим циклом розгортання, що базується на автоматизованих QG та послідовностях операцій пост-розгортання (наприклад, самостабілізації та самовідновлення сервісів) [21].

Keptn вирішує такі проблеми:

- оркестрація та уніфікація CD процесу на рівні усієї компанії. Всередині компанії може існувати багато команд, які витрачають багато часу та ресурсів на створення конвеєрів, створення спеціальних інтеграцій між інструментами та керування власними

сховищами даних. У великих компаніях команди розкидані по різних підрозділах чи регіонах, вони створюють свої окремі CD конвеєри та витрачають значну кількість часу на підтримку та підживлення коду конвеєрів, який невдовзі ризикує стати «застаріло-монолітним». Keptn дозволяє оркеструвати всі CD та операційні інструменти без написання спеціального коду — усе відбувається відповідно до декларативних конфігурацій та GitOps підходу.

- оркестрація постійних (або автоматизованих) операцій (CO або AO) на додачу до постійного розгортання (CD). У CD процесі релізна команда відповідає за доставку артефакта до оточення кінцевого користувача. Під час проходження усіх фаз (оточень) команда має можливість зупинити процес розгортання за допомогою різних засобів, наприклад, налаштуванням QG або використанням стратегій blue/green чи canary, убезпечившись тим самим від наслідків релізу неякісного артефакту до кінцевого споживача. Тому Keptn підтримує цілісні конвеєри з багатьма оточеннями та забезпечує розгортання якісних релізів за допомогою QG. Щодо автоматизованих операцій — відповідальна команда має зробити все, щоб мінімізувати негативний вплив на кінцевих користувачів у разі виникнення проблем після розгортання сервісу. Це може бути масштабування в разі виникнення потреби в додаткових ресурсах, переспрямування трафіку, очистка журналів та ін. Щоб швидко зреагувати, необхідно автоматизувати якомога більше таких задач. Тому Keptn підтримує керовані подіями автоматизовані задачі задля усунення виявлених проблем в оточенні якомога швидше, з метою мінімізації та усунення негативного впливу на кінцевих користувачів.

Keptn побудований на керованій подіями (event-driven) архітектурі, тобто усі його підсистеми виконують операції базуючись на отриманих

повідомленнях. Повідомлення, які Keptn розуміє, відповідають специфікації CloudEvents v0.2 [22]. Приклади повідомлень:

sh.keptn.event.[task].[event status]	Про подію виконання певної задачі в межах певної послідовності: sh.keptn.event.deployment.triggered sh.keptn.event.deployment.started sh.keptn.event.deployment.status.changed sh.keptn.event.deployment.finished
sh.keptn.event.[stage].[task sequence].[event status]	Про подію щодо певної послідовності в певному оточенні: sh.keptn.event.production.delivery.triggered sh.keptn.event.production.delivery.finished

Event-driven підхід дозволяє доволі ефективно використовувати ресурси Kubernetes кластеру, на якому Keptn, власне, і працює — усі процеси запускаються та починають використовувати ресурси кластеру тільки тоді, коли з’явилось повідомлення про певну, релевантну щодо їх функціоналу, подію. По завершенню виконання процеси вивільняють використовувані ресурси.

Завдяки своїй архітектурі Keptn є нейтральним до вибору інструментів, які можуть бути використані у його конвеєрі. Компанії мають повну свободу у використання інструментів у відповідності до своїх політик, технологічного стеку та усталеної практики. Так, можливе використання наявного CI інструменту, який по завершенні роботи повідомить Keptn про готовий до розгортання артефакт. Також можливе використання наявного CD інструменту — Keptn повідомляє його про готовність артефакту та очікує передачі керування назад, продовжуючи оркестрацію автоматичних операцій. Також можливе використання третіх інструментів для усіх задач з конвеєру, делегувавши Keptn лише загальний контроль послідовності виконання конвеєру та проходження QG.

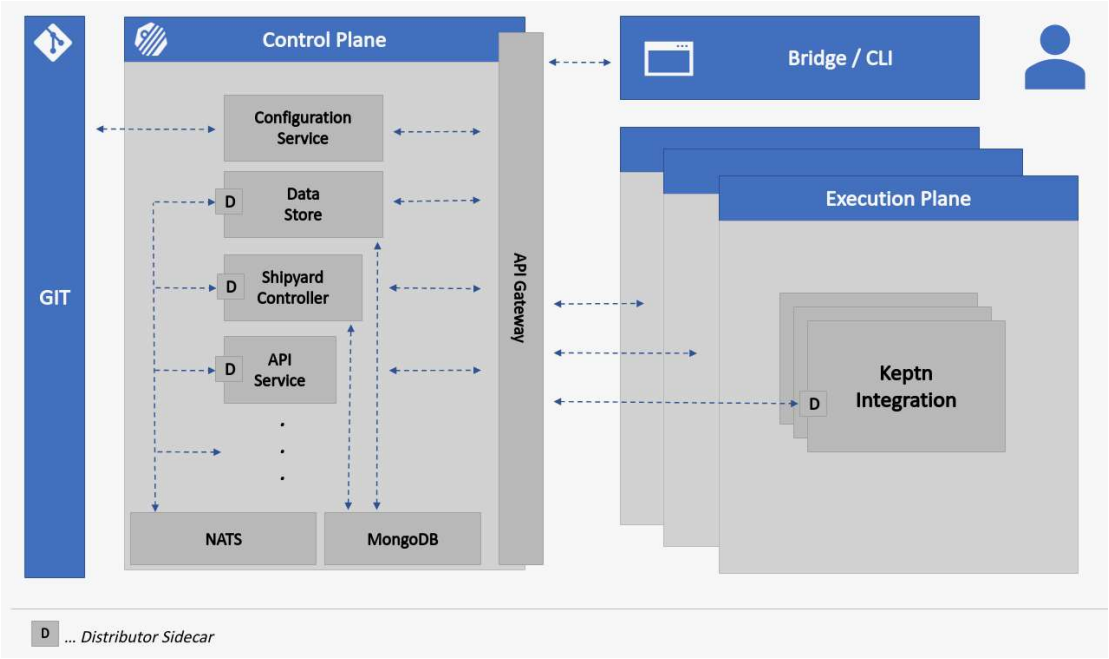

Рисунок 2.7 — Архітектура Keptn

Архітектура Keptn відображена на рис. 2.7 [23], складається з:

NATS: сервіс комунікації з виконавцем (Execution plane) [24].

Keptn CLI: інтерфейс для взаємодії з Keptn API, встановлюється на локальній машині для управління Keptn.

Keptn Bridge: веб-застосунок для перегляду та керування проєктами та сервісами.

Keptn Control Plane: контролер, включає необхідні компоненти для керування проєктами, оточеннями, сервісами. Включає обробку подій, оркеструє послідовності задач, але не виконує їх.

api-gateway-nginx: сервіс комунікації Keptn із зовнішнім світом, перенаправляє вхідні запити до відповідних внутрішніх кінцевих точок (API, Bridge, Resource-Service)

api-service: сервіс REST API для комунікації з Keptn

mongodb-datastore: сервіс для збереження подій

resource-service: основний компонент, що керує ресурсами проєктів, оточень, сервісів. Використовує репозиторій Git для збереження ресурсів з контролем версій

shipyard-controller: керує сутностями проєктів, оточень, сервісів, надає API для проведення CRUD операцій над ними; контролює виконання послідовностей задач, описаних проєктом, надсилаючи повідомлення на старт виконання задач (*.triggered*), а потім очікуючи відповідних повідомлень про те, що старт та завершення відбулись (*.started*, *.finished*)

Execution Plane Services: сервіси виконавця, які інтегрують інструменти для обробки задач. Кластер Keptn може мати єдиного виконавця, встановленого на тому ж Kubernetes кластері, що й контролер, але й можлива реалізація з кількома виконавцями на кількох Kubernetes кластерах.

Сервіси Keptn реагують на події *.triggered*, надіслані shipyard-controller'ом, та виконують задачі на кшталт розгортання або просування сервісів та автоматичних операцій. виконавець підписується на події, використовуючи один з наступних механізмів:

- **distributor sidecar** — розширення, яке перенаправляє вхідні *.triggered* події до сервісів виконавця. Ці ж розширення можуть також відсилати *.started* та *.finished* події назад до контролера;
- **cp-connector** (Control Plane Connector) — використовує код Go для обробки логіки інтеграції, з'єднуючись з контролером. Цей механізм з'явився у версії 0.15.x та використовується усіма ключовими сервісами Keptn. У цьому випадку distributor sidecar не потрібен, але вимагається більше кодування;
- **go-sdk** — обгортка до cp-connector'а, що додає додатковий функціонал. Майже всі внутрішні сервіси Keptn наразі використовують go-sdk.

Додатково до сервісів виконавця входять:

- lighthouse-service — виконує оцінку якості на базі сконфігурованих SLO/SLI;
- approval-service — імплементує автоматичний QG, якщо стратегія схвалень визначена як «автоматична», надсилаючи повідомлення про подію *approval.finished*, що містить інформацію про можливість подальшого продовження конвеєру. У випадку стратегії «ручного» схвалення сервіс не

надсилає повідомлень, а обов’язок погодити або відхилити продовження роботи покладається на користувача (який може використати Keptn Bridge або API);

- remediation-service — визначає дії, необхідні для процесів самовідновлення;
- mongodb-datastore — зберігає повідомлення.

2.2. Інструменти для створення конвеєру розгортання

Основним інструментом визначаємо Keptn — систему управління життєвим циклом розгортання. Основними перевагами від конкурентних рішень є ціна (Keptn повністю безкоштовний) та механізм QG одразу «з коробки». До прикладу, реалізація QG у Jenkins вимагає написання додаткового коду та виходить за межі концепції декларативного програмування [26].

Для написання декларативного коду в Keptn використовується YAML — зручний для читання людиною формат, концептуально близький до мов розмітки, але орієнтований на зручність введення-виведення типових структур даних багатьох мов програмування.

Назва YAML це рекурсивний акронім *YAML Ain't Markup Language* («YAML — не мова розмітки»). У назві відображена історія розвитку: на ранніх етапах мова називалася *Yet Another Markup Language* («Ще одна мова розмітки») і навіть розглядалася як конкурент XML, але пізніше була перейменована з метою акцентувати увагу на даних, а не на розмітці документів [27].

YAML в основному використовується як формат для файлів конфігурації, що ідеально підходить для написання декларативного коду конвеєру та створення конфігурацій SLI/SLO для QG.

В якості інструменту для написання коду використовуватимемо Visual Studio Code або, як його називають назагал, VS Code [28]. Це редактор коду,

створений Microsoft для ОС Windows, Linux та macOS та є найпопулярнішим інтегрованим середовищем розробки (Integrated Development Environment), за версією Stack Overflow [29].

2.3. Платформа для розгортання

В якості платформи для розгортання обираємо Kubernetes, що є лідером у області управління контейнерами та надає можливості високої доступності, масштабування та гнучкості. Також підтримка та реалізація Kubernetes присутня у всіх основних хмарних провайдерів (MS Azure, AWS, GCP). А тому наша реалізація отримує незалежність від постачальника та може бути вільно розгорнута як на локальному кластері компанії, так і у хмарного провайдера, або й навіть гібридно. Додатковим фактором вибору є те, що наш основний інструмент, Keptn — призначений до розгортання у Kubernetes кластері.

В якості платформи зі збереження коду будемо використовувати GitHub як один з найпопулярніших VCS сервісів.

Оскільки реалізація нашого завдання передбачає активне використання QG, насамперед автоматизованих, ми потребуємо платформу для моніторингу. В якості такої будемо скористаємось Dynatrace — платформою, що використовує штучний інтелект для моніторингу та надання пропозицій щодо оптимізації продуктивності програмних сервісів, застосування найкращих практик з безпеки, IT інфраструктури, а також покращення досвіду кінцевих користувачів[25].

Висновки до розділу

В даному розділі були проаналізовані існуючі засоби для автоматизації процесів розгортання програмних сервісів. Для подальшої реалізації нашого завдання зі створення конвеєру автоматичного розгортання ми визначили платформу та інструменти. Так, основним нашим інструментом визначено систему управління життєвим циклом розгортання Keptn, за допомогою якого

буде виконуватись конвеєр та задачі автоматизації. В якості платформи для роботи конвеєру визначені Kubernetes, Dynatrace та Github. Для написання коду конвеєру, конфігурації SLI/SLO буде використано мову YAML, в якості середовища розробки використовуватиметься VS Code.

Розділ 3. АЛГОРИТМІЧНЕ ЗАБЕЗПЕЧЕННЯ

В даному розділі визначимо алгоритми, що будуть використані для реалізації завдання у відповідності до поставленого завдання:

- алгоритм просування артефакту по конвеєру, в т.ч. проходження QG;
- автоматизація задач пост-розгортання:
  - відновлення попередньої стабільної версії;
  - відновлення рівня обслуговування.

3.1. Просування артефакту по конвеєру

Для реалізації завдання обираємо варіацію архітектури розгортання DTP з такими середовищами: розробки (Dev), дзеркало оточення кінцевого користувача (Stage), оточення кінцевого користувача (Prod). У кожному оточенні конвеєр послідовно буде виконувати задані послідовності задач розгортання. Завершення будь якої задачі або послідовності задач під час

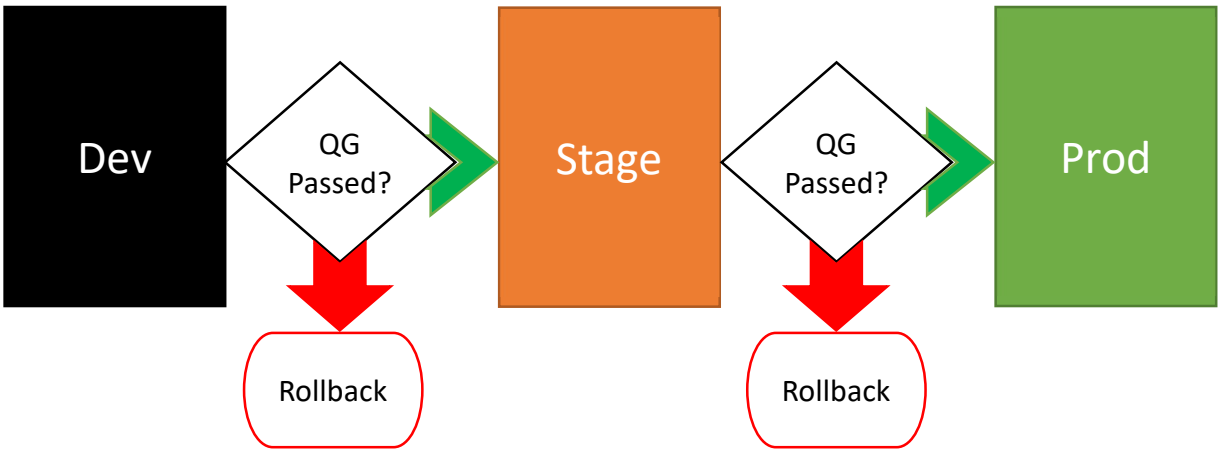

Рисунок 3.1 — Архітектура DSP з QG.

роботи конвеєра із помилкою розцінюється як неуспішне розгортання та має запустити послідовність задач відновлення до попередньої стабільної версії (rollback). Переміщення артефакту у наступне оточення буде відбуватись тільки після успішного проходження автоматизованих QG. Схематично це виглядатиме, як на рис. 3.1.

Автоматизовані QG будуть реалізовані за допомогою збору метрик (SLI) системою моніторингу та оцінки їх на відповідність визначеному рівню обслуговування (SLO).

Оцінювати якість розгорнутого сервісу будемо аналізуючи так звані «золоті сигнали» [32], які увійдуть до нашого SLI:

- відсоток помилок (error_rate);
- кількість оброблених запитів (throughput);
- час на відповідь (response time) з розподілом за перцентилями P90, P95.

Для визначення SLO опишемо правила оцінки якості роботи нашого сервісу:

- відсоток помилок має бути ≤ 1%; значення у діапазоні (1;2] — попередження;
- кількість оброблених запитів має бути > 40;
- час на відповідь за P95 має бути <500 мс одночасно з приростом відносно попередньої оцінки ≤50%; значення у діапазоні [500мс;1000мс] — попередження;
- приріст часу на відповідь за P90 відносно попередньої оцінки має бути ≤50%; приріст у діапазоні (50%;100%] означатиме попередження.

Оскільки рівень помилок у роботі програмного сервісу є важливішим за решту індикаторів, підвищимо його пріоритетність при оцінці якості роботи сервісу призначивши йому коефіцієнт ваги 2. За замовчуванням, усі правила мають базовий коефіцієнт 1.

3.2. Автоматизація задач пост-розгортання

Після розгортання у оточення кінцевого користувача нам необхідно впевнитись у якості роботи сервісу так якомога оперативніше відреагувати на можливі проблеми. Для цього ми скористаємось функціоналом

автоматизованих операцій та створимо послідовність дій для самостійної стабілізації сервісу за допомогою масштабування (рис. 3.2):

1) оцінка якості на відповідність визначеному SLO;

2) у випадку невідповідності проводиться додавання ще одного поду із сервісом з метою підвищення продуктивності;

3) кроки 1) та 2) повторюються до стабілізації (підвищення продуктивності до рівня, визначеного SLO) сервісу, але не більше визначеної кількості ітерацій, при досягненні якої викликається процедура відновлення попередньої, стабільної версії сервісу.

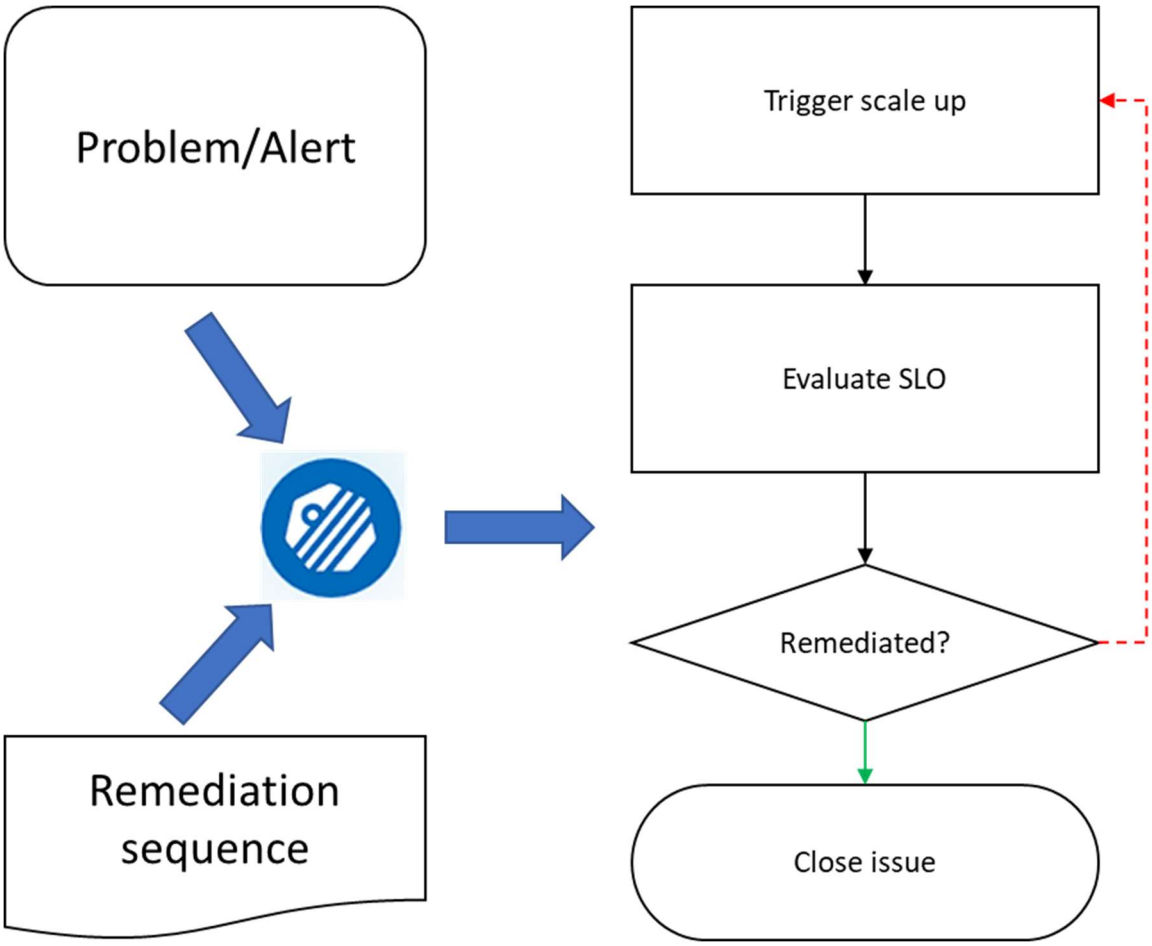

Рисунок 3.2 — Алгоритм процедури самостійної стабілізації сервісу за допомогою масштабування.

Висновки до розділу

Для подальшої реалізації конвеєру в даному розділі в якості архітектури розгортання визначено варіацію з трьома фазами DTP, визначені індикатори оцінки якості SLI, описані правила оцінки якості SLO, а також описані алгоритми переміщення артефактів по конвеєру, зокрема перехід між фазами за допомогою «воріт якості», відновлення до попередньої стабільної версії (rollback) та самостійної стабілізації сервісу за допомогою масштабування.

Розділ 4. ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

4.1. Підготовка платформи

Keptn працює поверх будь якого Kubernetes кластеру, тому можливе використання як вже наявного, так і розгортання виділеного виключно для потреб Keptn. Процедура встановлення Kubernetes описана в документації Keptn [30], там же присутня детальна інструкція по встановленню Keptn [31], тому зупинятись на ній не будемо.

Для подальшої роботи нам знадобиться токен для доступу до Dynatrace. Для генерації можна скористатись веб-інтерфейсом Dynatrace. Для цього необхідно в інтерфейсі Dynatrace зайти в “Settings”→”Access tokens” та натиснути “Create token”. Необхідно дати токену назву та вибрати такі зони дії токену (scopes):

```
Read entities                Capture request data
Read logs                    Access problem and event feed,
Ingest metrics               metrics, and topology
Read metrics                 Create and read synthetic monitors,
Read problems                locations, and nodes
Write problems               Read configuration
Read settings                Read synthetic monitors, locations,
Read SLO                     and nodes
Read synthetic locations
```

Оскільки зон дії велика кількість, для пошуку потрібних варто скористатись рядком пошуку «Select scopes», як видно на рис. 4.1. Отриманий токен необхідно зберегти, так як пізніше в інтерфейсі Dynatrace ми його не зможемо побачити.

Альтернативний спосіб отримання токену — API запит, для виконання якого необхідно в тому ж Dynatrace отримати персональний токен користувача, під яким здійснюються операції. Команда для генерації токена:

```
curl -X POST "https://<TENANT-ID>.live.dynatrace.com/api/v2/apiTokens" -H
"accept: application/json; charset=utf-8" -H "Content-Type:
application/json; charset=utf-8" -d
"{\"name\":\"Keptn\",\"scopes\":[\"entities.read\",\"logs.read\",\"metric
s.ingest\",\"metrics.read\",\"problems.read\",\"problems.write\",\"settin
gs.read\",\"slo.read\",\"syntheticLocations.read\",\"CaptureRequestData\"
,\"DataExport\",\"ExternalSyntheticIntegration\",\"ReadConfig\",\"ReadSyn
theticData\"]}" -H "Authorization: Api-Token <PERSONAL-TOKEN>"
```

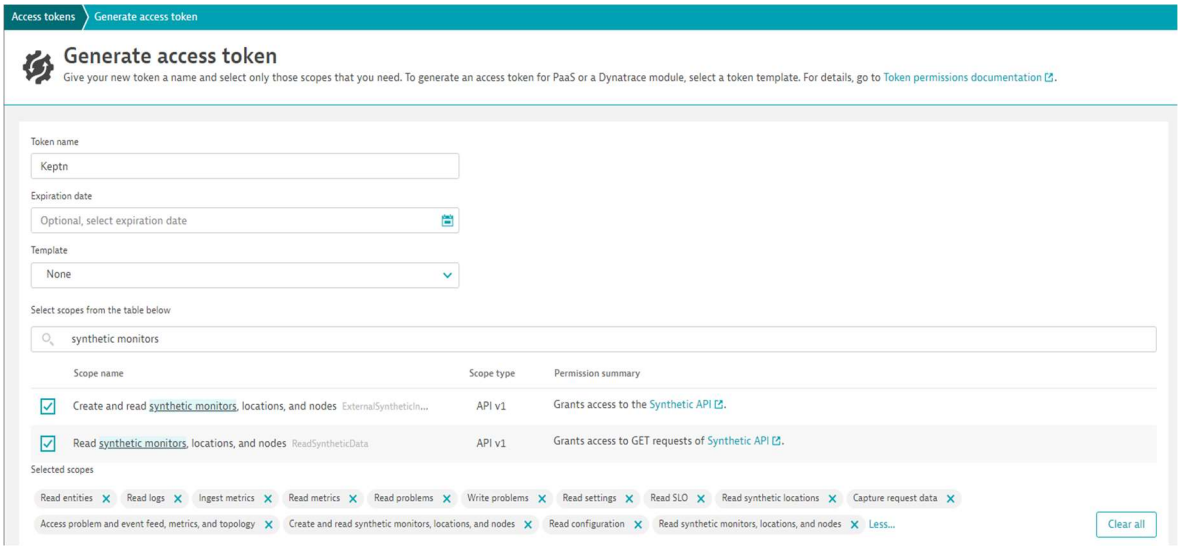

Рисунок 4.1 — Створення токену доступу до Dynatrace

На GitHub створюємо новий репозиторій (рис. 4.2).

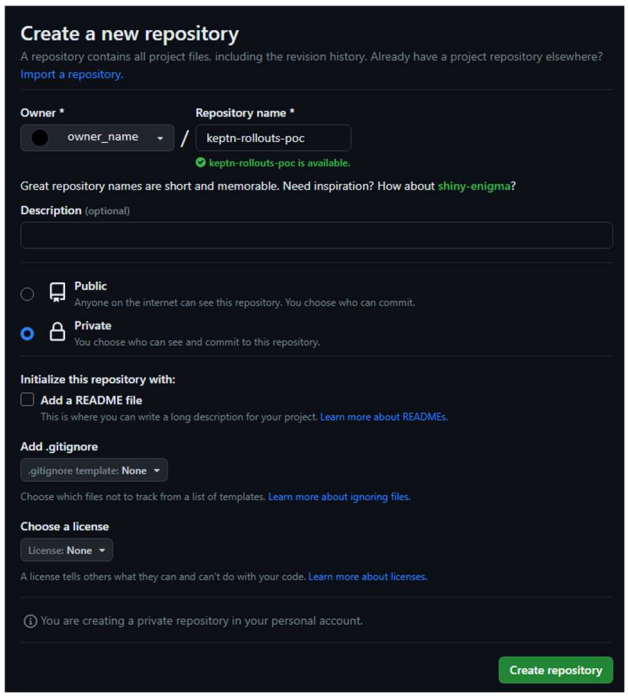

Рисунок 4.2 — Створення нового репозиторію GitHub

Keptn також потребує токен для доступу до GitHub, тому, в налаштуваннях облікового запису, в розділі “Developer settings” в пункті “Tokens (classic)” тиснемо “Generate new token” та обираємо “Generate new token (classic)”. Вводимо текст нотатки для ідентифікації токена, вказуємо термін його життя та відмічаємо “repo” в переліку зон дії (рис. 4.3). Завершуємо генерацію токена та зберігаємо його для подальшого використання.

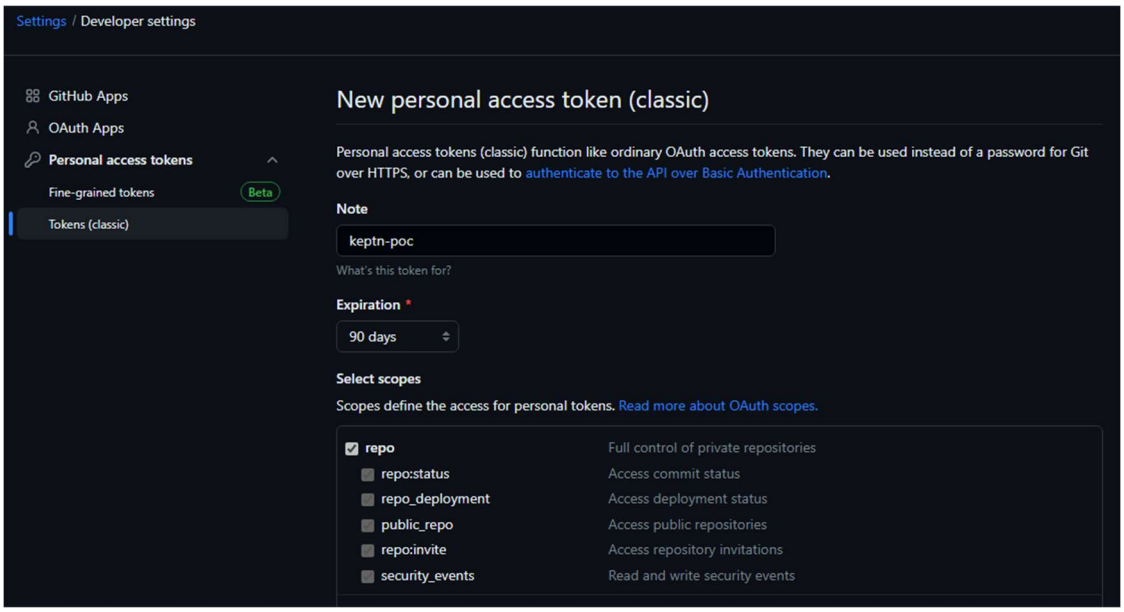

Рисунок 4.3 — Генерація токена GitHub

На локальному комп'ютері встановлюємо VS Code, клієнт Git та Keptn CLI.

4.2. Створення конвеєру розгортання

Для створення проєкту в Keptn необхідно описати конфігурацію конвеєру у shipyard.yaml файлі. Мінімально конфігурація потребує перелічення усіх запланованих оточень (секція stages), кожне з яких має включати щонайменше одну послідовність задач (секція sequences), що, в свою чергу, має містити мінімум одну задачу (секція tasks).

Лістинг коду shipyard.yaml подано у додатках.

Створимо сутності Keptn, а саме проєкт та сервіс. Проєкт назвемо "keptn-poc", сервіс — "node1".

CLI команда для створення проєкту "keptn-poc":

```
keptn.exe create project keptn-poc --git-user=<GH-USER> --
git-token=<GH_TOKEN> --git-remote-url=https://github.com/<GH-
USER>/keptn-rollouts-poc --shipyard="shipyard.yaml"
```

CLI команда для створення сервісу "node1" в межах проєкту "keptn-poc":

```
keptn.exe create service node1 --project=keptn-poc
```

Також необхідно додати токен Dynatrace до налаштування проєкту. Досягти цього можна через графічний інтерфейс Keptn (рис. 4.4) або CLI командою:

```
keptn create secret dynatrace --from-
literal="DT_TENANT=<TENANT-URL>" --from-
literal="DT_API_TOKEN=<DYNATRACE-TOKEN>" --scope="dynatrace-
service"
```

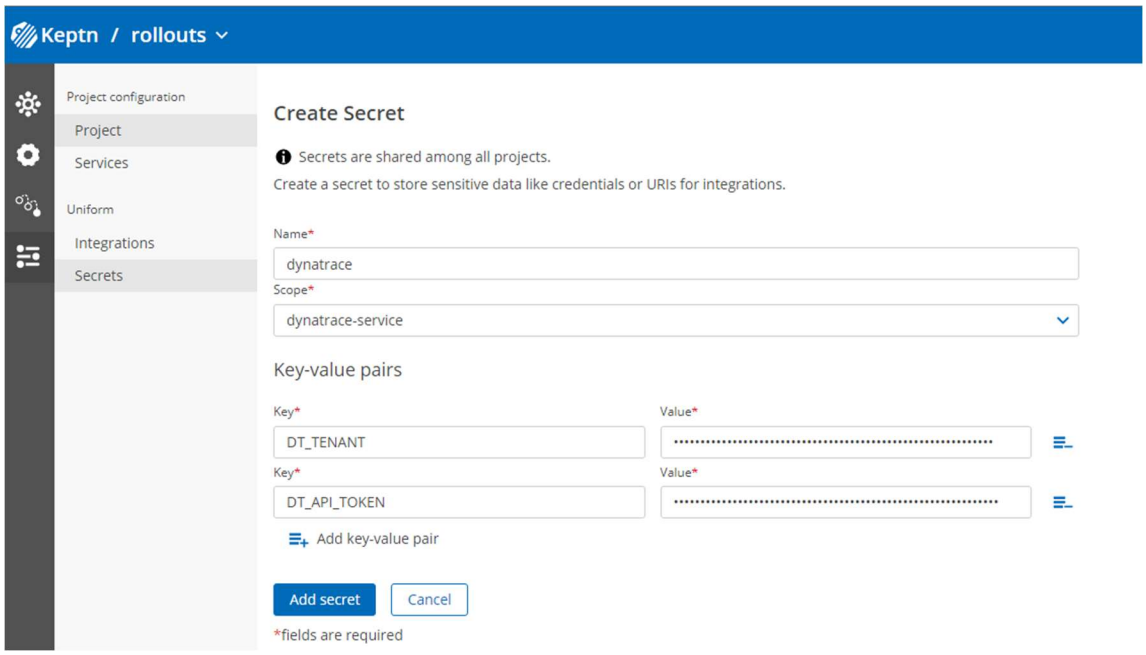

Рисунок 4.4 — Додавання секретних ключів в Keptn

Оскільки операції в різних оточеннях можуть відбуватись паралельно та відрізнятись, Keptn для збереження конфігурацій оточень використовує гілки репозиторію. Основна гілка master (або main) служить для збереження глобальної конфігурації проєкту, тобто конвеєру та налаштувань Dynatrace, включно з SLI (лістинг sli.yaml подано у додатках). Файли конфігурацій усіх оточень зберігаються окремо у гілках “dev”, “stage” та “prod” відповідно.

Базова структура файлів для оточення виглядає, як на рис. 4.5. Директорія “helm” використовується для збереження конфігураційних файлів розгортання сервісу за допомогою helm. Директорія “job” призначена для зберігання конфігурації задач, які мають виконуватись в межах

послідовностей задач, а також автоматизованих задач пост-розгортання, наприклад, відправка повідомлень про результат розгортання, відновлення до попередньої версії в разі помилок або низької якості релізу. Додатково можуть бути присутні інші директорії та файли, якщо вони необхідні для виконання задач конвеєру.

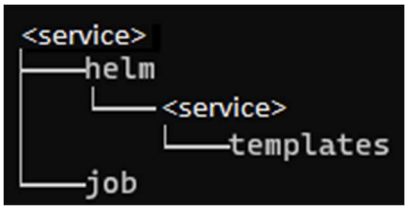

Рисунок 4.5 — Базова структура директорій оточення.

Файл визначення SLO має бути збережений у директорії “<service>”, лістинг подано у додатках.

Додавання файлів до проєкту загалом або конкретного сервісу може бути здійснена за допомогою CLI Keptn або ж за допомогою командної оболонки локального комп’ютера із використанням git. Другий варіант виглядає більш доцільним для задач розробки та тестування конвеєру, оскільки ми можемо редагувати файли та проводити маніпуляції з репозиторієм за допомогою зручних нам інструментів (як то: VSCode та git), на відміну від досить громіздких команд Keptn.

Отже, клонуємо репозиторій:

```
mkdir keptn
cd keptn
git@github.com:<GH-USER>/keptn-rollouts-poc.git
```

У гілці master в корені створюємо директорію “dynatrace” та додаємо до неї файли конфігурації Dynatrace (лістинг dynatrace.conf.yaml та sli.yaml наведено у додатках). Зберігаємо зміни та відправляємо їх до репозиторію:

```
git add .
git commit -m “Added Dynatrace configuration files”
git push
```

Для того, щоб включити моніторинг Dynatrace для нашого проєкту, необхідно виконати команду:

```
keptn configure monitoring dynatrace --project=keptn-poc
```

Переключаючись між гілками, створюємо директорії та додаємо файли конфігурацій до всіх оточень (лістинги файлів наведено у додатках). Приклад для оточення “dev”:

```
git -checkout dev
cd node1
mkdir helm
mkdir job
mkdir locust
cd helm
mkdir node1
cd node1
mkdir templates
```

По завершенню додавання усіх файлів до оточення “dev” зберігаємо зміни та відправляємо їх до репозиторію:

```
git add .
git commit -m “Added configuration for Dev environment”
git push
```

4.3. Тестування роботи конвеєру

Запуск конвеєру розгортання сервісу у оточення “dev” можна здійснити через GUI або за допомогою Keptn CLI:

```
keptn trigger delivery --project=keptn-poc --service=node1 --
stage=dev --
image=docker.io/grabnerandi/simplenodeservice:1.0.0 --
values="replicaCount=3"
```

Оскільки створений нами shipyard.yaml описує послідовності дій для усіх оточень та тригери виконання задач, нам не потрібно примусово запускати послідовності задач оточень “stage” та “prod”. Так, згідно з нашим алгоритмом, розгортання сервісу у «наступному» оточенні має відбуватись

тільки після завершення розгортання у «попередньому» оточенні та успішному проходженні контролю якості.

Спостереження за ходом роботи можливе у Keptn GUI, результат роботи конвеєра видно на рис. 4.6.

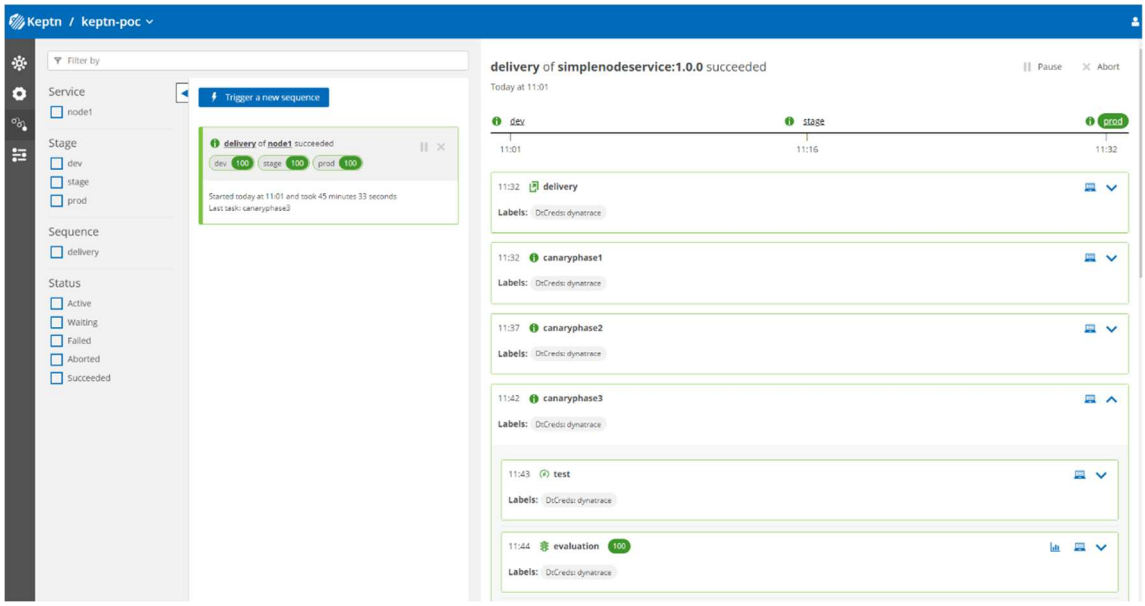

Рисунок 4.6 — GUI Keptn із ходом виконання конвеєру.

Отже, наш сервіс успішно пройшов автоматичне розгортання послідовно у всіх середовищах, без втручання людини. Переходи між середовищами відбулись автоматично завдяки успішним проходженням QG, що свідчить про відповідність якості розгорнутого сервісу нашим наперед встановленим SLO.

Але тестовий запуск виявив надмірність витрат часу, та, відповідно — обчислювальних ресурсів, оскільки конвеєр описує задачі запуску тестів, проведення оцінки та прийняття рішення про просування далі після запуску кожного пода. Така ретельність необхідна у Production оточенні та може бути усунута у оточеннях Dev та Stage.

Тому, перед наступними запусками нам необхідно відкоригувати код конвеєру, залишивши тестування та оцінку якості лише у Production оточенні та при переходах між оточеннями. Відповідно, у файлі shipyard.yaml ми коментуємо (або видаляємо) задачі (секція tasks) "test", "evaluation", "approval"

у послідовностях (секція sequences) "canaryphase1" та "canaryphase2" оточень (секція stages) "dev" та "stage".

Додатково використовуватимемо сценарій Powershell з метою генерації запитів до API нашого сервісу для отримання відповідей із випадковими затримками — таким чином ми зімітуємо «відвідуваність» та певну «активність» нашого сервісу. Лістинг файлу Imitate-Activity.ps1 подано у додатках.

Робимо декілька запусків конвеєру та спостерігаємо за ходом виконання (рис. 4.7). Завдяки зменшенню кількості тестів та оцінок час роботи конвеєру у оточеннях Dev та Stage зменшився з 16 хв. до 7-8 хв., тобто на 50-56%. Також видно вплив роботи сценарію імітації активності: випадкові затримки спричиняють збільшення середнього часу на відповідь та погіршення оцінки якості.

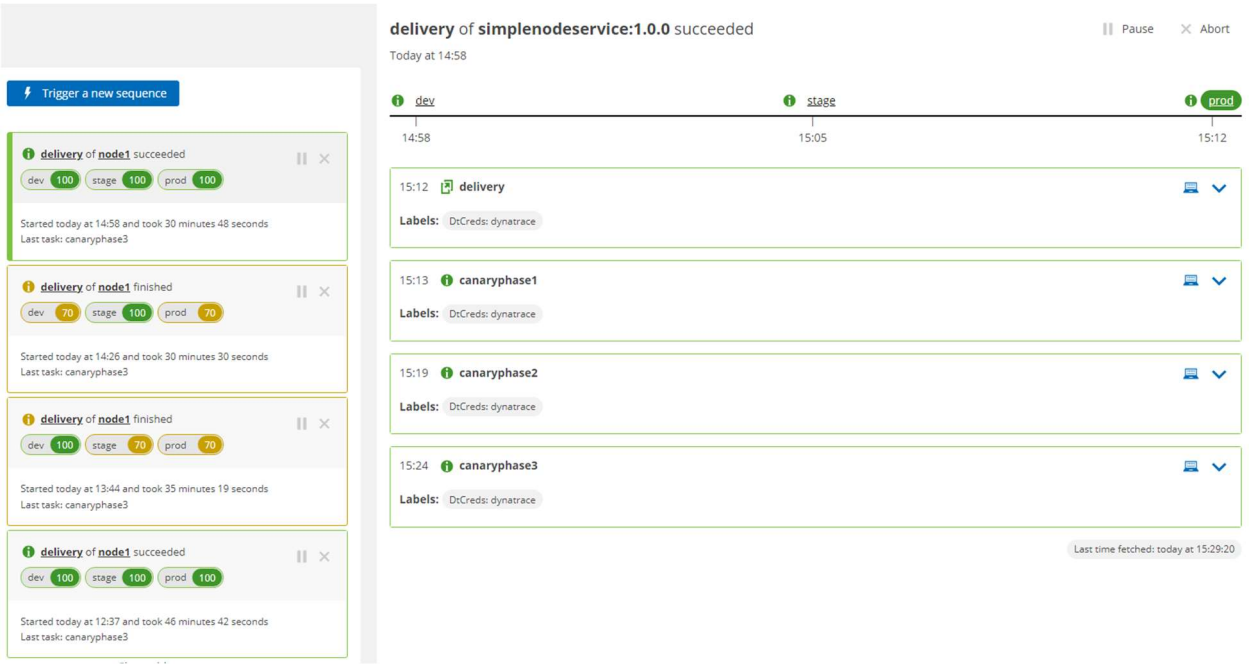

Рисунок 4.7 — Результати роботи конвеєру

Результати оцінювання якості можна побачити у графічному вигляді — як загальні, так і по кожному індикатору зокрема. Так, на рис. 4.8 представлені результати оцінювання Production оточення, в табличній частині відображаються детальні результати.

Рисунок 4.8 — Графічне представлення результатів оцінки якості

Протестуємо автоматичне відновлення попередньої версії в разі помилки. Запускаємо конвеєр розгортання версії 2.0.0 нашого сервісу:

```
keptn trigger delivery --project=keptn-poc --service=node1 --
stage=dev --
image=docker.io/grabnerandi/simplenodeservice:2.0.0 --
values="replicaCount=3"
```

Моніторимо процес розгортання та бачимо, що задача "test" завершилась з помилкою, та запустилась задача "rollback" (рис. 4.9).

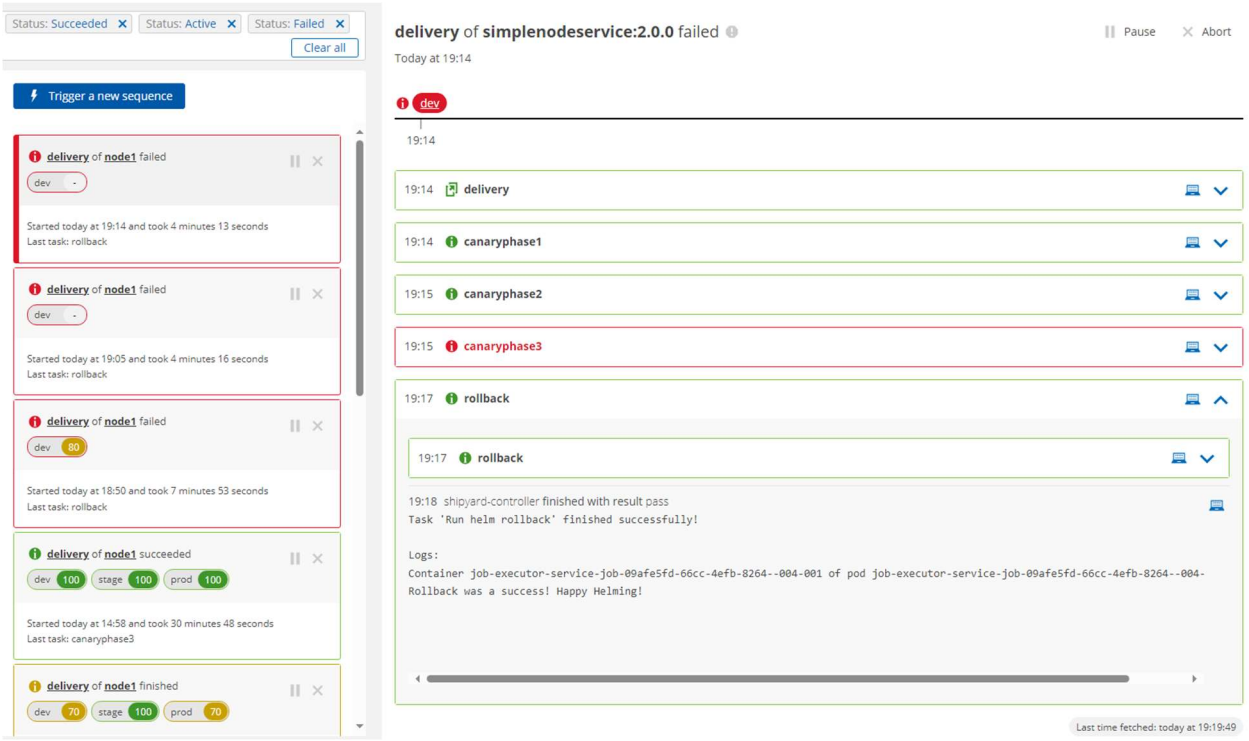

Рисунок 4.9 — Виконання задачі відновлення попередньої версії

Відновлення попередньої версії відбулось успішно, реліз неякісного програмного код не був допущений.

Протестуємо роботу послідовності самовідновлення рівня обслуговування сервісу. Для цього виконуємо команду:

```
keptn trigger sequence remediation1 --project=keptn-poc --service=node1 --stage=dev
```

Хід виконання та результат видно на рис. 4.10. Оскільки наші коригуючі дії полягають у горизонтальному масштабуванні сервісу (задача action), то нам необхідно встановити певну затримку на додавання нового поду в репліку та виходу сервісу на робочий режим. Час затримки може варіюватись в залежності від завантаженості кластера Kubernetes та особливостей самого сервісу. В нашому випадку старт повторної оцінки якості після виконання коригуючих дій відбувається із затримкою 3 хв. (`triggeredAfter: "3m"`). Оцінювання якості після процедури горизонтального масштабування сервісу показало успішне відновлення рівня обслуговування.

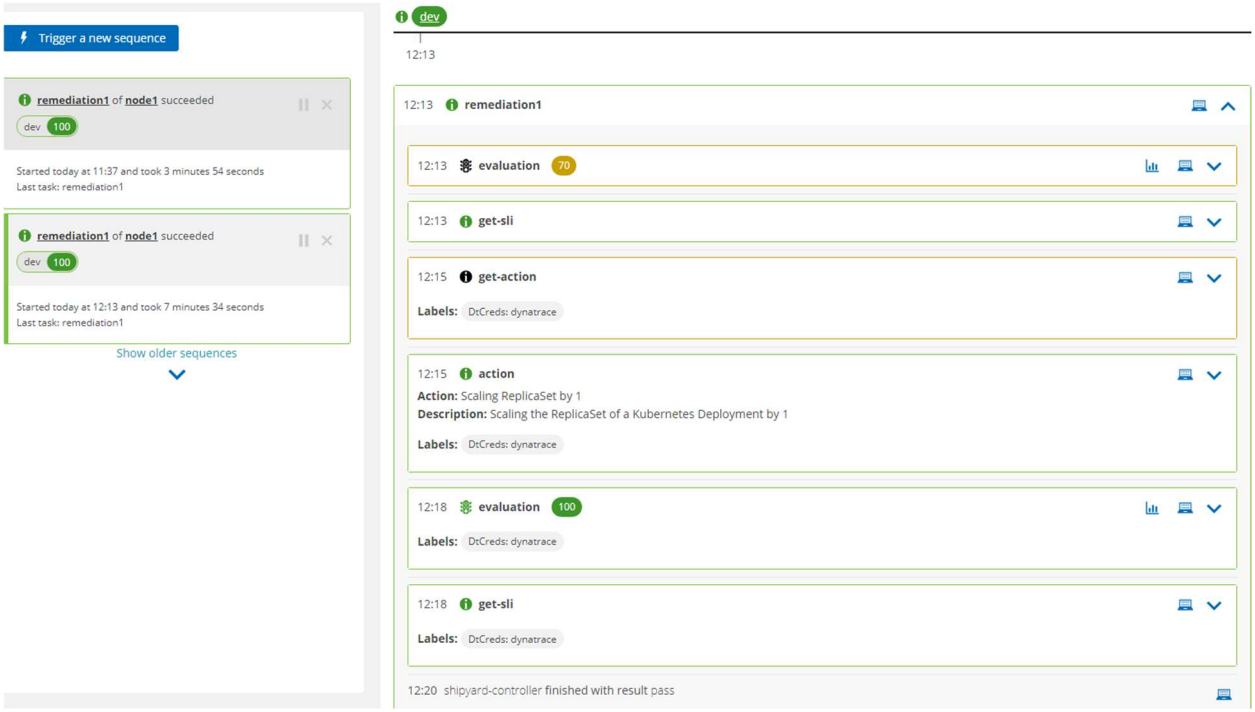

Рисунок 4.10 — Хід виконання послідовності самовідновлення

Висновки до розділу

В даному розділі викладена послідовність дій з налаштування платформи для роботи конвеєру та процес створення конвеєру. Проведено визначення індикаторів (SLI) та описані правила оцінки якості релізу (SLO). Програмний код конвеєру та допоміжних конфігурацій наведено у додатках.

Також проведені тестові запуски конвеєра та отримані результати його роботи. Під час тестування використовувався сценарій для імітації відвідуваності та активності розгорнутого сервісу. Отримані результати доводять здатність успішного функціонування конвеєру на базі системи керування життєвим циклом Keptn та використання його для автоматизації процесів розгортання програмних сервісів, задач контролю якості (QG), автоматизації задач пост-розгортання, зокрема послідовностей самовідновлення рівня обслуговування.

Також отримані результати підтвердили, що надмірне використання «воріт якості» спричиняють значні витрати ресурсів. Тому їх необхідно використовувати, спираючись на доцільність та пріоритети замовника. Так,

контроль якості розгортання слід використовувати при переходах між оточеннями та у оточенні кінцевого користувача, але обмежити використання такого контролю у dev, stage та інших «нижчих» оточеннях.

Розділ 5. РОЗРОБЛЕННЯ СТАРТАП-ПРОЄКТУ

5.1. Опис ідеї проєкту

Ідея проєкту описана в табл. 1.

Таблиця 1. Зміст ідеї, напрямки застосування та вигоди стартап-проекту

Зміст ідеї	Напрямки застосування	Вигоди для користувачів
Надання послуг з впровадження контролю якості розгортання сервісів та автоматизації операції пост-розгортання (quality gates and post-deployment operations)	Компанії, що використовують методологію DevOps, CI/CD процеси для розробки програмних сервісів та Kubernetes як платформу розгортання	Покращення якості випущених сервісів. Прискорення випусків нових версій. Запобігання випуску неякісного коду. Підвищення рівня видимості процесів Використання наявних, «знайомих» інструментів

Порівняльний аналіз показників ідеї наведено у таблиці 2.

Таблиця 2. Визначення сильних, слабких та нейтральних характеристик ідеї проекту

№ п/п	Техніко-економічні характеристики ідеї	Товари/концепції конкурентів (потенційні)		W (слабка сторона)	N (нейтральна сторона)	S (сильна сторона)
		Проєкт	Конкурент			
1	Фізичне розміщення	На площадці клієнта	На власній площадці		+	+
2	Інтеграція з різними інструментами та система моніторингу	Є	Є, тільки з власною системою моніторингу			+
3	Вартість імплементації	Висока	Низька	-		
4	Плата за використання	Відсутня	Присутня			+
5	Функціонал	Quality gates та автоматизація операцій	Тільки Quality Gates			+

В табл. 3 подано перелік технологій, що можуть бути використані для реалізації проєкту.

Таблиця 3. Технологічна здійсненність ідеї проекту

Ідея	Наявність технологій	Технології та реалізації	Доступність технологій
Реалізація воріт якості та автоматизованих операцій пост-розгортання	Наявні	Система управління життєвим циклом розгортання Keptn Контейнеризація та оркестрація контейнерів: Kubernetes Менеджер пакетів: Helm Система управління версіями: Git Мова написання конвеєру та конфігурацій: YAML Системи моніторингу: Dynatrace, Prometheus Інструменти CI/CD: Jenkins, ArgoCD, інші Інтеграція з інструментами CI/CD: Groovy Фреймворки тестування: Locust, Jmeter, OWASP ZAP, інші	Доступні

Отже, проєкт технічно можливо реалізувати, використовуючи наявні на ринку технології. Оскільки проєкт планується, як незалежний від конкретних інструментів, в кожній конкретній реалізації будуть використані ті, які вже наявні в замовників та їм знайомі. Так, можуть використовуватись різні системи управління версіями для збереження коду конвеєра та допоміжних конфігурацій, різні системи моніторингу та фреймворки тестування.

Для аналіз ринкових можливостей запуску стартап-проєкту, опишемо основну групу клієнтів (табл. 4), фактори загроз (табл. 5) та можливостей (табл. 6), проведемо ступеневий аналіз конкуренції на ринку (табл. 7), проведемо аналіз сильних та слабких сторін проєкту (SWOT-аналіз) (табл. 8).

Таблиця 4. Основні групи потенційних клієнтів та їх потреби

Група клієнтів	Потреби, що задовольняються за допомогою проєкту
Компанії, які працюють за DevOps методологією та використовують Kubernetes як платформу розгортання	Контроль за ходом розгортання та прозоре інформування щодо версій розгорнутих сервісів у всіх оточеннях. Звіти та аналітика щодо якості розгорнутих сервісів. Можливість визначення метрик якості (SLI) та рівнів якості обслуговування (SLO). Автоматизація прийняття рішень щодо відповідності рівню якості обслуговування. Автоматизація CI/CD процесів для швидкого та безпечного розгортання у різних оточеннях, забезпечення індемпотентності. Інтеграція з наявними інструментами моніторингу та аналізу результатів тестування. Виявлення та вирішення проблем розгортання в реальному часі.

Таблиця 5. Фактори загроз.

Фактор	Зміст	Можливі заходи
Зміни в технологічному стеку	Поява нових інструментів та технологій	Постійний моніторинг технологій, вивчення можливостей інтеграції
Конкуренція та стратегії конкурентів	Вихід на ринок іншої компаній з подібним сервісом або аналогічним продуктом	Вивчення конкурентної пропозиції, розширення функціоналу
Зниження попиту	Зміни в уподобаннях клієнтів або трендах ринку можуть вимагати змін у функціональності	Постійний моніторинг ринку, проведення кампаній з отримання зворотного зв’язку від клієнтів щодо функціональності; проведення кампанії з пропагування важливості аналізу якості розгортання
Фінансова нестабільність клієнтів	Фінансова нестабільність клієнтів може призвести до відкладення або скорочення проектів	Диверсифікація клієнтської бази

Таблиця 6. Фактори можливостей

Фактор	Зміст	Можливі заходи
Підвищення попиту / зростання ринку	Збільшення кількості клієнтів, зацікавлених в запровадженні контролю якості розгортання	Проведення рекламної кампанії для залучення нових клієнтів
Зниження конкуренції	Втрата ефективності конкурентом	Проведення рекламної кампанії для переконання потенційних клієнтів у високій ефективності наших послуг

Таблиця 7. Ступеневий аналіз конкуренції на ринку

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною)
Тип конкуренції: олігополія	На ринку невелика кількість гравців	Врахувати вартість послуг конкурентів при виході на ринок
За рівнем конкурентної боротьби: міжнародна	Усі гравці присутні на міжнародному рівні	Можливість вийти одразу на міжнародний ринок; Співпраця з вендорами технологій та інструментів
За галузевою ознакою: міжгалузева	IT технології використовуються у багатьох галузях	Можливе використання усюди, де використовується DevOps методологія, CI/CD процеси
Конкуренція за видами товарів: товарно-видова	Однаковий вид послуг	Вивчити досвід конкурентів, врахувати їх недоліки
За характером конкурентних переваг: нецінова	Різні способи надання послуг; клієнти готові платити більше за зручнішу реалізацію	Гнучкість та адаптація під потреби клієнтів
За інтенсивністю: не марочна	Різні способи задоволення потреб споживачів	Розвиток власного бренду

Таблиця 8. SWOT-аналіз проєкту

Сильні сторони: - звіти та аналітика щодо якості розгорнутих сервісів; - гнучкість індивідуальної реалізації; - ширший функціонал; - розміщення на площадці клієнта, надання йому повного контролю	Слабкі сторони: - висока вартість реалізації; - необхідність індивідуальної адаптації для кожного клієнта; - необхідність залучення висококооплачуваних спеціалістів
Можливості: - вихід на міжнародний ринок; - популяризація DevOps методології; - популяризація концепції Quality Gates; - створення нового продукту після накопичення досвіду	Загрози: - поява нових конкурентів; - поява нових продуктів у ніші; - зміна пріоритетів у клієнтів; - зміна тенденцій на ринку

На основі SWOT-аналізу розроблені альтернативи ринкової поведінки для виведення проєкту на ринок (табл. 9). З огляду на потенційні проєкти конкурентів. Серед визначених альтернатив найприйнятнішою є укладання угоди з ІТ компанією.

Таблиця 9. Альтернативи ринкового впровадження проєкту

№ п/п	Альтернатива (орієнтовний комплекс заходів) ринкової поведінки	Ймовірність отримання ресурсів	Строки реалізації
1	Співпраця із вендором основного інструменту	Середня	12 міс.
2	Укладання угоди з ІТ компанією	Висока	12 міс.
3	Рекламна кампанія	Низька	3 міс.

5.2. Розроблення ринкової стратегії

Першим кроком в розробленні ринкової стратегії опишемо цільову групу потенційних споживачів (табл. 10).

Таблиця 10. Вибір цільових груп потенційних споживачів

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
Компанії, які працюють за DevOps методологією та використовують Kubernetes в якості платформи розгортання	Існує зацікавленість в підвищенні видимості та інформативності процесів розгортання	Середній	Існує конкурент, що надає доступ до власної реалізацію на власних потужностях	Середня

Оскільки ми зосереджуємось на одному сегменті споживачів, тому обираємо стратегію концентрованого маркетингу.

Формуємо базову стратегію розвитку (табл. 11).

Таблиця 11. Визначення базової стратегії розвитку

Обрана альтернатива розвитку проекту	Стратегія охоплення ринку	Ключові конкурентоспроможні позиції відповідно до обраної альтернативи	Базова стратегія розвитку
Укладання угоди з IT компанією	Концентрацію на потребах цільового сегменту, задоволення потреб краще, ніж конкурент	Гнучкість реалізації	Стратегія спеціалізації

Наступним кроком обираємо стратегію конкурентної поведінки (табл. 12).

Таблиця 12. Визначення базової стратегії конкурентної поведінки

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	Шукати нових	Так, функціонал, що буде розширений	Стратегія зайняття конкурентної ніші

Так, обрана нами базова стратегія — зайняття конкурентної ніші, орієнтована на клієнтів, що вимагають гнучкого та індивідуального рішення.

Відповідно до основних потреб клієнтів (табл. 4), базової стратегії розвитку (табл. 11) та базової стратегії конкурентної поведінки (табл. 12), розробляємо стратегію позиціонування (табл. 13).

Таблиця 13. Визначення стратегії позиціонування

Вимоги до товару цільової аудиторії	Базова стратегія розвитку	Ключові конкурентоспроможні позиції власного стартап-проекту	Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових)
Контроль за ходом розгортання та прозоре інформування щодо версій розгорнутих сервісів у всіх оточеннях. Звіти та аналітика щодо якості розгорнутих сервісів. Автоматизоване прийняття рішень щодо результатів тестування. Інтеграція з іншими інструментами	Стратегія спеціалізації	Автоматизовані «ворота якості»; графічна візуалізація проходження «воріт якості»; автоматизацій операцій пост-розгортання; інтеграція з різними інструментами замовника	Контроль якості розгортання; автоматизація прийняття рішень; гнучка інтеграція з наявними інструментами

5.3. Розроблення маркетингової програми

Для формування маркетингової програми визначаємо ключові переваги нашого проєкту (табл. 14).

Таблиця 14. Визначення ключових переваг

№ п/п	Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами (існуючі або такі, що потрібно створити
1	Автоматизація прийняття рішень про якість розгортання	Автоматизовані «ворота якості»	Можливість автоматизації операцій пост-розгортання
2	Інтеграція з наявними інструментами	Гнучка інтеграція з різними інструментами	Гнучка, індивідуальна інтеграція з усіма необхідними інструментами

Визначаємо систему збуту (табл. 15).

Таблиця 15. Формування системи збуту

Специфіка закупівельної поведінки цільових клієнтів	Функції збуту, які має виконувати постачальник товару	Глибина каналу збуту	Оптимальна система збуту
Замовлення послуг розробки певного функціоналу	Інформаційне забезпечення	Нульова, прямий продаж послуги	Власна

Нами обрана власна система збуту, базована на прямих продажах, оскільки планується надання послуги у відповідності до індивідуальних потреб замовників та орієнтуючись на вирішення конкретних задач.

Наступним кроком розробимо концепцію маркетингових комунікацій, що спирається на попередньо обрану основу для позиціонування та визначену специфіку поведінки клієнтів (табл. 16).

Таблиця 16. Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Канали комунікацій, якими користуються цільові клієнти	Ключові позиції, обрані для позиціонування	Завдання рекламного повідомлення	Концепція рекламного звернення
Готовність використання інноваційних технологій для досягнення конкретного результату	Інтернет, участь в сторонніх заходах (он-лайн та офф-лайн)	Автоматизація прийняття рішень про якість розгортання, індивідуальна інтеграція з існуючими інструментами, підвищення видимості та якості процесів розгортання	Залучення клієнтів	Рекламне звернення з акцентом раціональних мотивах досягнення конкретного результату, полегшення виконання задач, отримання конкурентних переваг.

Отже, концепція наших маркетингових комунікацій полягатиме у використанні як цифрових, так і традиційних каналів комунікацій для розповсюдження рекламних повідомлень з метою залучення клієнтів. Концепція самого рекламного звернення акцентуватиметься на раціональній мотивації потенційних замовників досягти конкретного результату за допомогою наших послуг.

Висновки до розділу

В даному розділ проведено аналіз можливості запуску стартап проєкту, ідеєю якого є надання послуг з впровадження контролю якості розгортання сервісів та автоматизації операції пост-розгортання. Отримані результати показують, що така можливість існує, оскільки на ринку наявний запит на рішення з автоматизації контролю рівня якості розгорнутих сервісів та

підвищення видимості й інформативності процесів розгортання у різних оточеннях. З огляду на конкуренцію запуск буде ускладнений, оскільки є один гравець, що пропонує готове рішення за SaaS моделлю, тому обраною стратегією конкурентної поведінки є стратегія зайняття конкурентної ніші. Наш проєкт буде позиціонуватись як індивідуальне, гнучке, повністю пристосоване під потреби замовника та орієнтуватись на клієнтів, що потребують кращого задоволення своїх потреб.

ВИСНОВКИ

В результаті виконання роботи було розглянуто процес розгортання сервісів, розділено його на декілька етапів, проведено аналіз кожного із етапів та встановлено, що кожен із таких етапів може бути автоматизовано незалежно від інших, що дозволяє розподілити процес розгортання між кількома інструментами.

В ході роботи був досліджений фазовий підхід до розгортання сервісів, що полягає у використанні різних оточень розгортання при виконанні IT-проєктів. Дослідження показало, що кількість оточень може варіюватись від компанії до компанії, залежно від їх усталених практик, та всі оточення мають свої призначення.

Також було розглянуто вагому частину DevOps — контроль якості сервісів, що розгортаються, за допомогою quality gates. Так, впровадження QG покращує якість виконання проєкту, дозволяючи досягти вищого рівня успішності, фокусуючись на розгортанні тільки якісних релізів, та допомагаючи контролювати відповідність SLO, особливо у оточенні кінцевого користувача.

Було розглянуто рішення та інструменти для розгортання сервісів, та встановлено, що провідною платформою для розгортання є Kubernetes — де-факто лідер у області управління контейнерами. Підтримка та реалізація Kubernetes присутня у всіх основних хмарних провайдерів (MS Azure, AWS, GCP), що означає незалежність від постачальника при побудові рішення на базі Kubernetes. Щодо інструментів розгортання — встановлено, що багато з них також орієнтовані на використання Kubernetes.

Для реалізації завдання даної роботи був обраний Keptn — інструмент для централізованого управління життєвим циклом розгортання програмних сервісів, який, окрім класичних задач CD, вирішує задачі з оркестрації розгортань у багатьох оточеннях одночасно, а також задачі з автоматизації QG та автоматизації операцій CO (пост-розгортання).

Так, було створено конвеєр для автоматичного розгортання програмного сервісу у оточеннях згідно архітектури DTP. В рамках конвеєру під оркестрацією Keptn було реалізовано:

- використання різних інструментів для різних типів задач;
- автоматизовані QG на базі відповідності заданим SLO, з метою контролю якості розгортання та можливості просування до наступної фази;
- автоматизована послідовність дій з відновлення попередньої стабільної версії після помилки або непроходження QG;
- автоматизована послідовність дій для відновлення сервісом рівня обслуговування у відповідності до SLO.

Були проведені експериментальні запуски створеного конвеєру та отримані результати його роботи. Під час тестування використовувався сценарій для імітації відвідуваності та активності розгорнутого сервісу. Отримані результати доводять здатність успішного функціонування конвеєру на базі системи керування життєвим циклом Keptn та використання його для автоматизації процесів розгортання програмних сервісів, задач контролю якості (QG), автоматизації задач пост-розгортання, зокрема послідовностей самовідновлення рівня обслуговування.

Також отримані результати підтвердили, що надмірне використання «воріт якості» спричиняють значні витрати ресурсів. Тому не слід використовувати QG на кожному кроці задля переслідування цілі підвищення якості за будь яку ціну. Кроки з використанням QG слід впроваджувати, спираючись на їх доцільність та пріоритети замовника. Так, QG варто використовувати при переходах між оточеннями та у оточенні кінцевого користувача, але обмежити їх використання у dev, stage та інших «нижчих» оточеннях.

Аналіз ринкових можливостей та розробка стартап-проєкту показали, що проєкт з надання послуг із реалізація QG та автоматизації операцій пост-розгортання може бути реалізований, оскільки на ринку існує запит на рішення з автоматизації контролю рівня якості розгорнутих сервісів та

підвищення видимості й інформативності процесів розгортання у різних оточеннях.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Software deployment [Електронний ресурс]. https://en.wikipedia.org/wiki/Software_deployment
2. Deployment environment [Електронний ресурс]. https://en.wikipedia.org/wiki/Deployment_environment
3. A. Jones. The Development, Staging, and Production Model [Електронний ресурс]. https://www.itprotoday.com/devops-and-software-development/development-staging-and-production-model
4. Development, testing, acceptance and production [Електронний ресурс]. https://en.wikipedia.org/wiki/Development,_testing,_acceptance_and_production
5. Project Management Institute, Inc. (2021). Настанова до Зводу знань з управління проєктами (НастановаPMBOK), 7-е видання. с. 31
6. How DevOps Quality Gates Improve Deployments [Електронний ресурс]. https://www.copado.com/devops-hub/blog/how-devops-quality-gates-improve-deployments-cddd
7. OWASP Top Ten [Електронний ресурс]. https://owasp.org/www-project-top-ten/
8. PCI DSS [Електронний ресурс]. https://www.pcisecuritystandards.org/document_library/
9. C. Bernstein. Quality Gate [Електронний ресурс]. https://www.techtarget.com/searchsoftwarequality/definition/quality-gate
10. Version control [Електронний ресурс]. https://en.wikipedia.org/wiki/Version_control
11. Kubernetes Components [Електронний ресурс]. https://kubernetes.io/docs/concepts/overview/components/
12. N. Darshan Docker Swarm: A Complete Guide for Beginners [Електронний ресурс]. https://k21academy.com/docker-kubernetes/docker-swarm/
13. Docker Compose overview [Електронний ресурс]. https://docs.docker.com/compose/
14. Resource Allocation in Mesos: Dominant Resource Fairness [Електронний ресурс]. https://datastrophic.io/resource-allocation-in-mesos-dominant-resource-fairness-explained/
15. Mesos Architecture [Електронний ресурс]. https://mesos.apache.org/documentation/latest/architecture/
16. Rancher. Getting started. [Електронний ресурс]. https://ranchermanager.docs.rancher.com/getting-started/overview/
17. Rancher Server and Components [Електронний ресурс]. https://ranchermanager.docs.rancher.com/reference-guides/rancher-manager-architecture/rancher-server-and-components
18. Rancher 2.0: Exciting Features You Should Know About [Електронний ресурс]. https://www.cloudops.com/blog/rancher-2-0-exciting-features/
19. An overview of Octopus Deploy concepts [Електронний ресурс]. https://octopus.com/docs/getting-started
20. B. Mihaylov. Creating a deployment pipeline using TeamCity and Octopus Deploy [Електронний ресурс]. https://boyan.io/deployment-pipeline-using-teamcity-octopus/
21. Keptn Docs. [Електронний ресурс]. https://keptn.sh/
22. Cloud Events Documentation [Електронний ресурс]. https://github.com/cloudevents/spec/tree/v0.2
23. Keptn. Architecture [Електронний ресурс]. https://keptn.sh/docs/concepts/architecture/
24. NATS. [Електронний ресурс]. https://nats.io/about/

25. Dynatrace [Електронний ресурс]. https://en.wikipedia.org/wiki/Dynatrace
26. Declarative Programming [Електронний ресурс]. https://en.wikipedia.org/wiki/Declarative_programming
27. YAML [Електронний ресурс]. https://en.wikipedia.org/wiki/YAML
28. Visual Studio Code [Електронний ресурс]. https://en.wikipedia.org/wiki/Visual_Studio_Code
29. 2022 Developer Survey [Електронний ресурс]. https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment
30. Create or bring a Kubernetes cluster [Електронний ресурс]. https://keptn.sh/docs/1.0.x/install/k8s/
31. Install Keptn using the Helm chart [Електронний ресурс]. https://keptn.sh/docs/1.0.x/install/helm-install/
32. Site Reliability Engineering. [Електронний ресурс]. https://sre.google/sre-book/monitoring-distributed-systems/
33. Mojtaba S. Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices / Mojtaba Shahin, 2017. – с. 16.
34. Mackinnon T. Endo-Testing: Unit Testing with Mock Objects / Mackinnon Tim, 2016. – с. 1-2.
35. B. Fitzgerald and K.-J. Stol, ‘‘Continuous software engineering: A roadmap and agenda,’’ J. Syst. Softw., vol. 123, pp. 176–189, Jan. 2017.

ДОДАТКИ

Лістинг файлу shipyard.yaml

```
apiVersion: spec.keptn.sh/0.2.4
kind: "Shipyard"
metadata:
  name: "keptn-poc"
spec:
  stages:
    - name: "dev"
      sequences:
      - name: "delivery"
        tasks:
        - name: "deployment"
          properties:
            deploymentstrategy: "user_managed"
            remediation_strategy: "automated"
      - name: "canaryphase1"
        triggeredOn:
          - event: dev.delivery.finished
            selector:
              match:
                result: "pass"
        tasks:
        - name: "test"
          triggeredAfter: "1m"
          properties:
              teststrategy: "real-user"
              canarywaitduration: "1m"
        - name: "evaluation"
          properties:
            timeframe: "1m"
        - name: "approval"
          properties:
            pass: "automatic"
            warning: "manual"
        - name: "release"
      - name: "canaryphase2"
        triggeredOn:
          - event: dev.canaryphase1.finished
            selector:
              match:
                result: "pass"
        tasks:
        - name: "test"
          triggeredAfter: "1m"
          properties:
              teststrategy: "real-user"
              canarywaitduration: "1m"
        - name: "evaluation"
          properties:
            timeframe: "1m"
        - name: "approval"
          properties:
            pass: "automatic"
            warning: "manual"
        - name: "release"
      - name: "canaryphase3"
        triggeredOn:
          - event: dev.canaryphase2.finished
            selector:
              match:
```

```
            result: "pass"
    tasks:
    - name: "test"
      triggeredAfter: "1m"
      properties:
          teststrategy: "real-user"
          canarywaitduration: "1m"
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "release"
  - name: "rollback"
    triggeredOn:
    - event: dev.delivery.finished
      selector:
        match:
          result: "fail"
    - event: dev.canaryphase1.finished
      selector:
        match:
          result: "fail"
    - event: dev.canaryphase2.finished
      selector:
        match:
          result: "fail"
    - event: dev.canaryphase3.finished
      selector:
        match:
          result: "fail"
    tasks:
    - name: "rollback"
  - name: "remediation1"
    triggeredOn:
      - event: dev.canaryphase3.finished
        selector:
          match:
            evaluation.result: "warning"
    tasks:
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "get-action"
    - name: "action"
    - name: "evaluation"
      triggeredAfter: "3m"
      properties:
        timeframe: "1m"

- name: "stage"
  sequences:
  - name: "delivery"
    triggeredOn:
      - event: dev.canaryphase3.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "deployment"
      properties:
```

```
      deploymentstrategy: "user_managed"
  - name: "canaryphase1"
    triggeredOn:
      - event: stage.delivery.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "test"
      triggeredAfter: "1m"
      properties:
          teststrategy: "real-user"
          canarywaitduration: "1m"
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "release"
  - name: "canaryphase2"
    triggeredOn:
      - event: stage.canaryphase1.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "test"
      triggeredAfter: "1m"
      properties:
          teststrategy: "real-user"
          canarywaitduration: "1m"
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "release"
  - name: "canaryphase3"
    triggeredOn:
      - event: stage.canaryphase2.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "test"
      triggeredAfter: "1m"
      properties:
          teststrategy: "real-user"
          canarywaitduration: "1m"
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "release"
  - name: "rollback"
    triggeredOn:
```

```
    - event: stage.delivery.finished
      selector:
        match:
          result: "fail"
    - event: stage.canaryphase1.finished
      selector:
        match:
          result: "fail"
    - event: stage.canaryphase2.finished
      selector:
        match:
          result: "fail"
    - event: stage.canaryphase3.finished
      selector:
        match:
          result: "fail"
    tasks:
    - name: "rollback"
  - name: "remediation1"
    triggeredOn:
      - event: stage.canaryphase3.finished
        selector:
          match:
            evaluation.result: "warning"
    tasks:
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "get-action"
    - name: "action"
    - name: "evaluation"
      triggeredAfter: "3m"
      properties:
        timeframe: "1m"


- name: "prod"
  sequences:
  - name: "delivery"
    triggeredOn:
      - event: stage.canaryphase3.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "deployment"
      properties:
        deploymentstrategy: "user_managed"
  - name: "canaryphase1"
    triggeredOn:
      - event: prod.delivery.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "test"
      triggeredAfter: "1m"
      properties:
          teststrategy: "real-user"
```

```
        canarywaitduration: "1m"
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "release"
  - name: "canaryphase2"
    triggeredOn:
      - event: prod.canaryphase1.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "test"
      triggeredAfter: "1m"
      properties:
          teststrategy: "real-user"
          canarywaitduration: "1m"
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "release"
  - name: "canaryphase3"
    triggeredOn:
      - event: prod.canaryphase2.finished
        selector:
          match:
            result: "pass"
    tasks:
    - name: "test"
      triggeredAfter: "1m"
      properties:
          teststrategy: "real-user"
          canarywaitduration: "1m"
    - name: "evaluation"
      properties:
        timeframe: "1m"
    - name: "approval"
      properties:
        pass: "automatic"
        warning: "manual"
    - name: "release"
  - name: "rollback"
    triggeredOn:
    - event: prod.delivery.finished
      selector:
        match:
          result: "fail"
    - event: prod.canaryphase1.finished
      selector:
        match:
          result: "fail"
    - event: prod.canaryphase2.finished
      selector:
        match:
          result: "fail"
```

```
        – event: prod.canaryphase3.finished
          selector:
            match:
              result: "fail"
          tasks:
          – name: "rollback"
        - name: "remediation1"
          triggeredOn:
            - event: prod.canaryphase3.finished
              selector:
                match:
                  evaluation.result: "warning"
          tasks:
          - name: "evaluation"
            properties:
              timeframe: "1m"
          - name: "get-action"
          - name: "action"
          - name: "evaluation"
            triggeredAfter: "3m"
            properties:
              timeframe: "1m"
```

Лістинг файлу dynatrace/dynatrace.conf.yaml

```
---
spec_version: '0.1.0'
dtCreds: dynatrace
attachRules:
  tagRule:
    – meTypes:
        – SERVICE
      tags:
        – context: CONTEXTLESS
          key: $SERVICE
        – context: CONTEXTLESS
          key: keptn_project
          value: $PROJECT
        – context: CONTEXTLESS
          key: keptn_service
          value: $SERVICE
        – context: CONTEXTLESS
          key: keptn_stage
          value: $STAGE
        – context: CONTEXTLESS
          key: keptn_managed
```

Лістинг файлу dynatrace/sli.yaml

```
---
spec_version: '1.0'
indicators:
  throughput:
"metricSelector=builtin:service.requestCount.total:splitBy():sum&entitySelect
or=tag(keptn_project:$PROJECT),tag(keptn_stage:$STAGE),tag(keptn_service:$SER
VICE),tag(keptn_deployment:$DEPLOYMENT),type(SERVICE)"
  error_rate:
"metricSelector=builtin:service.errors.total.rate:splitBy():avg&entitySelecto
r=tag(keptn_project:$PROJECT),tag(keptn_stage:$STAGE),tag(keptn_service:$SERV
ICE),tag(keptn_deployment:$DEPLOYMENT),type(SERVICE)"
```

```
  response_time_p50:
"metricSelector=builtin:service.response.time:splitBy():percentile(50):toUnit
(microSecond,milliSecond)&entitySelector=tag(keptn_project:$PROJECT),tag(kept
n_stage:$STAGE),tag(keptn_service:$SERVICE),tag(keptn_deployment:$DEPLOYMENT)
,type(SERVICE)"
  response_time_p90:
"metricSelector=builtin:service.response.time:splitBy():percentile(90):toUnit
(microSecond,milliSecond)&entitySelector=tag(keptn_project:$PROJECT),tag(kept
n_stage:$STAGE),tag(keptn_service:$SERVICE),tag(keptn_deployment:$DEPLOYMENT)
,type(SERVICE)"
  response_time_p95:
"metricSelector=builtin:service.response.time:splitBy():percentile(95):toUnit
(microSecond,milliSecond)&entitySelector=tag(keptn_project:$PROJECT),tag(kept
n_stage:$STAGE),tag(keptn_service:$SERVICE),tag(keptn_deployment:$DEPLOYMENT)
,type(SERVICE)"
```

Лістинг файлу node1/slo.yaml

```
---
spec_version: "0.1.0"
comparison:
  compare_with: "single_result"
  include_result_with_score: "pass"
  aggregate_function: avg
objectives:
  – sli: response_time_p95
    pass:
      – criteria:
          – "<=+50%"
          – "<500"
    warning:
      – criteria:
          – "<=1000"
  – sli: throughput
    pass:
      – criteria:
          – ">40"
  – sli: error_rate
    weight: 2
    pass:
      – criteria:
          – "<=1%"
    warning:
      – criteria:
          – "<=2%"
  – sli: response_time_p90
    pass:
      – criteria:
          – "<=+50%"
    warning:
      – criteria:
          – "<=+100%"
total_score:
  pass: "95%"
  warning: "70%"
```

Лістинг файлу node1/remediation.yaml

```
apiVersion: spec.keptn.sh/0.1.4
kind: Remediation
metadata:
  name: node1-remediation
```

```
spec:
  remediations:
  - problemType: Response time degradation
    actionsOnOpen:
      - action: scaling
        name: Scaling ReplicaSet by 1
        description: Scaling the ReplicaSet of a Kubernetes Deployment by 1
        value: "1"
  - problemType: default
    actionsOnOpen:
      - action: scaling
        name: Scaling ReplicaSet by 1
        description: Scaling the ReplicaSet of a Kubernetes Deployment by 1
        value: "1"
```

Лістинг файлу node1/helm/node1/Chart.yaml

```
apiVersion: v1
description: A Helm chart for Keptn
name: node1
type: application
version: 0.1.0
```

Лістинг файлу node1/helm/node1/values.yaml

```
image: docker.io/grabnerandi/simplenodeservice:2.0.0
replicaCount: 1
tag: v2.0.0

keptn:
  project: project
  service: service
  stage: stage
  strategy: user_managed

hostname: helloservice.test.cloud
```

Лістинг файлу node1/helm/node1/templates/_helpers.tpl

```
{{/* vim: set filetype=mustache: */}}
{{/*
Expand the name of the chart.
*/}}
{{- define "node1.name" -}}
{{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-" -}}
{{- end -}}

{{/*
Create a default fully qualified app name.
We truncate at 63 chars because some Kubernetes name fields are limited to
this (by the DNS naming spec).
If release name contains chart name it will be used as a full name.
*/}}
{{- define "node1.fullname" -}}
{{- if .Values.fullnameOverride -}}
{{- .Values.fullnameOverride | trunc 63 | trimSuffix "-" -}}
{{- else -}}
{{- $name := default .Chart.Name .Values.nameOverride -}}
{{- if contains $name .Release.Name -}}
{{- .Release.Name | trunc 63 | trimSuffix "-" -}}
{{- else -}}
```

```
{{- printf "%s-%s" .Release.Name $name | trunc 63 | trimSuffix "-" -}}
{{- end -}}
{{- end -}}
{{- end -}}

{{/*
Create chart name and version as used by the chart label.
*/}}
{{- define "node1.chart" -}}
{{- printf "%s-%s" .Chart.Name .Chart.Version | replace "+" "_" | trunc 63 |
trimSuffix "-" -}}
{{- end -}}
```

Лістинг файлу node1/helm/node1/templates/mappings.yaml

```
apiVersion: getambassador.io/v2
kind: Mapping
metadata:
  name: "{{ .Values.keptn.service }}-{{ .Values.keptn.stage}}"
  labels:
    app: {{ .Values.keptn.service }}
spec:
  host: {{ .Values.hostname }}
  prefix: "/{{ .Values.keptn.service }}-{{ .Values.keptn.stage}}/"
  service:  "{{ .Values.keptn.service }}-stable"
  timeout_ms: 30000

---
apiVersion: getambassador.io/v2
kind: Mapping
metadata:
  name: "{{ .Values.keptn.service }}-{{ .Values.keptn.stage}}-canary"
  labels:
    app: {{ .Values.keptn.service }}
spec:
  host: {{ .Values.hostname }}
  prefix: "/{{ .Values.keptn.service }}-{{ .Values.keptn.stage}}-canary/"
  service:  "{{ .Values.keptn.service }}-canary"
  timeout_ms: 30000
```

Лістинг файлу node1/helm/node1/templates/rollout.yaml

```
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: "{{ .Values.keptn.service }}"
  labels:
    app: {{ .Values.keptn.service }}
    chart: {{ template "node1.chart" . }}
    serviceGroup: test
    version: {{ (split ":" .Values.image)._1 | default "latest" }}
spec:
  replicas: {{ .Values.replicaCount }}
  revisionHistoryLimit: 3
  selector:
    matchLabels:
      app: {{ .Values.keptn.service }}
  strategy:
    canary:
      canaryService: "{{ .Values.keptn.service }}-canary"
      stableService: "{{ .Values.keptn.service }}-stable"
      trafficRouting:
```

```
        ambassador:
          mappings:
            - {{ .Values.keptn.service }}
      steps:
        - setWeight: 33
        - pause:
            {}
        - setWeight: 66
        - pause:
            {}
  template:
    metadata:
      labels:
        app: {{ .Values.keptn.service }}
        version: {{ (split ":" .Values.image)._1 | default "latest" }}
        app.kubernetes.io/managed-by: Keptn
        app.kubernetes.io/part-of: {{ .Values.keptn.project }}
        app.kubernetes.io/name: {{ .Values.keptn.service }}
        app.kubernetes.io/instance: "{{ .Values.keptn.service }}-{{
.Values.keptn.stage }}"
        app.kubernetes.io/component: api
        app.kubernetes.io/version: {{ (split ":" .Values.image)._1 | default
"latest" }}
    spec:
      serviceAccountName: {{ .Values.keptn.service }}
      containers:
        - name: {{ .Values.keptn.service }}
          image: "{{ .Values.image }}"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              protocol: TCP
              containerPort: 8080
          env:
            - name: DT_CUSTOM_PROP
              value: >-
                keptn_project={{ .Values.keptn.project }}
                keptn_service={{ .Values.keptn.service }}
                keptn_stage={{ .Values.keptn.stage}}
                keptn_deployment={{ .Values.keptn.strategy }}
                keptn_managed
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: "metadata.name"
            - name: DEPLOYMENT_NAME
              valueFrom:
                fieldRef:
                  fieldPath: "metadata.labels['deployment']"
            - name: CONTAINER_IMAGE
              value: "{{ .Values.image }}"
            - name: KEPTN_PROJECT
              value: {{ .Values.keptn.project }}
            - name: KEPTN_STAGE
              value: {{ .Values.keptn.stage }}
            - name: KEPTN_SERVICE
              value: {{ .Values.keptn.service }}
            - name: KEPTN_DEPLOYMENT
              value: {{ .Values.keptn.strategy }}
            - name: DT_RELEASE_VERSION
              valueFrom:
                fieldRef:
                  fieldPath: "metadata.labels['app.kubernetes.io/version']"
```

```
        - name: DT_RELEASE_STAGE
          value: {{ .Values.keptn.stage }}
        - name: DT_RELEASE_PRODUCT
          value: {{ .Values.keptn.service }}
        - name: DT_RELEASE_BUILD_VERSION
          valueFrom:
            fieldRef:
              fieldPath: "metadata.labels['app.kubernetes.io/version']"
      livenessProbe:
        httpGet:
          path: /
          port: 8080
        initialDelaySeconds: 60
        periodSeconds: 10
        timeoutSeconds: 15
      readinessProbe:
        httpGet:
          path: /
          port: 8080
        initialDelaySeconds: 60
        periodSeconds: 10
        timeoutSeconds: 15
      resources:
        limits:
          cpu: 200m
          memory: 500Mi
        requests:
          cpu: 200m
          memory: 500Mi
```

Лістинг файлу node1/helm/node1/templates/serviceaccount.yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ .Values.keptn.service }}
```

Лістинг файлу node1/helm/node1/templates/services.yaml

```
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: {{ .Values.keptn.service }}
  name: "{{ .Values.keptn.service }}-stable"
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: {{ .Values.keptn.service }}
---
apiVersion: v1
kind: Service
metadata:
  labels:
    app: {{ .Values.keptn.service }}
  name: "{{ .Values.keptn.service }}-canary"
```

```
spec:
  type: ClusterIP
  ports:
  - name: http
    port: 80
    protocol: TCP
    targetPort: 8080
  selector:
    app: {{ .Values.keptn.service }}
```

Лістинг файлу node1/job/config.yaml

```
apiVersion: v2
actions:
  - name: "Deploy using helm"
    events:
      - name: "sh.keptn.event.deployment.triggered"
    tasks:
      - name: "Run helm"
        files:
          - /helm
        env:
          - name: IMAGE
            value: "$.data.configurationChange.values.image"
            valueFrom: event
          - name: REPLICACOUNT
            value: "$.data.configurationChange.values.replicaCount"
            valueFrom: event
          - name: DEPLOYMENTSTRATEGY
            value: "$.data.deployment.deploymentstrategy"
            valueFrom: event
        image: "alpine/helm:3.7.2"
        serviceAccount: "helm-service"
        cmd: ["helm"]
        args:
          [
            "upgrade", "--create-namespace", "--install",
            "-n", "$(KEPTN_PROJECT)-$(KEPTN_STAGE)",
            "$(KEPTN_SERVICE)",
            "/keptn/helm/$(KEPTN_SERVICE)",
            "--set", "image=$(IMAGE)",
            "--set", "replicaCount=$(REPLICACOUNT)",
            "--set", "keptn.project=$(KEPTN_PROJECT)",
            "--set", "keptn.service=$(KEPTN_SERVICE)",
            "--set", "keptn.stage=$(KEPTN_STAGE)",
            "--set", "keptn.strategy=$(DEPLOYMENTSTRATEGY)",
            "--wait"
          ]
      - name: "Execute Argo get info"
        image: "quay.io/argoproj/kubectl-argo-rollouts"
        args: ["get", "rollout", "$(SERVICE)", "-n", "$(PROJECT)-$(STAGE)",
"--no-color"]
        serviceAccount: "keptn-argo-service"
        env:
          - name: SERVICE
            value: "$.data.service"
            valueFrom: event
          - name: STAGE
            value: "$.data.stage"
            valueFrom: event
          - name: PROJECT
            value: "$.data.project"
            valueFrom: event
```

```
  - name: "Scale using kubectl"
    events:
      - name: "sh.keptn.event.action.triggered"
        jsonpath:
          property: "$.data.action.action"
          match: "scaling"
    tasks:
      - name: "Run kubectl"
        serviceAccount: "helm-service"
        env:
          - name: SCALING
            value: $.data.action.value
            valueFrom: event
        image: "alpine/k8s:1.20.15"
        cmd: ["sh"]
        args:
          [
            "-c",
            "REPLICAS=$(kubectl -n ${PROJECT}-${STAGE} get rollout ${SERVICE}
-o go-template='{{.spec.replicas}}');DESIRED=$((${SCALING}+${REPLICAS}));echo
Scaling deployment to ${DESIRED} && kubectl -n ${PROJECT}-${STAGE} scale
rollout ${SERVICE} --replicas=${DESIRED}",
          ]

  - name: "Run tests using locust"
    events:
      - name: "sh.keptn.event.test.triggered"
    tasks:
      - name: "Run locust"
        files:
          - locust/basic.py
          - locust/locust.conf
        image: "locustio/locust:latest"
        cmd: ["locust"]
        args:
          [
            "--config", "/keptn/locust/locust.conf",
            "--locustfile", "/keptn/locust/basic.py",
            "--host", "http://$(KEPTN_SERVICE)-canary.$(KEPTN_PROJECT)-
$(KEPTN_STAGE)",
            "--only-summary",
            "--loglevel", "INFO",
            "--exit-code-on-error", "0"
          ]

  - name: "Run Argo promote"
    events:
      - name: "sh.keptn.event.release.triggered"
    tasks:
      - name: "Execute Argo promote"
        image: "quay.io/argoproj/kubectl-argo-rollouts"
        args: ["promote", "$(SERVICE)", "-n", "$(PROJECT)-$(STAGE)"]
        serviceAccount: "keptn-argo-service"
        env:
          - name: SERVICE
            value: "$.data.service"
            valueFrom: event
          - name: STAGE
            value: "$.data.stage"
            valueFrom: event
          - name: PROJECT
            value: "$.data.project"
```

```
            valueFrom: event
      - name: "Execute Argo get info"
        image: "quay.io/argoproj/kubectl-argo-rollouts"
        args: ["get", "rollout", "$(SERVICE)", "-n", "$(PROJECT)-$(STAGE)",
"--no-color"]
        serviceAccount: "keptn-argo-service"
        env:
          - name: SERVICE
            value: "$.data.service"
            valueFrom: event
          - name: STAGE
            value: "$.data.stage"
            valueFrom: event
          - name: PROJECT
            value: "$.data.project"
            valueFrom: event

  - name: "Run Rollout abort"
    events:
      - name: "sh.keptn.event.rollback.triggered"
    tasks:
      - name: "Execute argo rollouts abort"
        image: "quay.io/argoproj/kubectl-argo-rollouts"
        args: [ "abort", "$(SERVICE)", "-n", "$(PROJECT)-$(STAGE)" ]
        serviceAccount: "keptn-argo-service"
        env:
          - name: SERVICE
            value: "$.data.service"
            valueFrom: event
          - name: STAGE
            value: "$.data.stage"
            valueFrom: event
          - name: PROJECT
            value: "$.data.project"
            valueFrom: event
      - name: "Run helm rollback"
        serviceAccount: "helm-service"
        image: "alpine/helm:3.7.2"
        cmd: ["helm"]
        args:
          [
            "rollback",
            "-n", "$(KEPTN_PROJECT)-$(KEPTN_STAGE)",
            "$(KEPTN_SERVICE)",
            "--wait",
          ]

  - name: "Argo test canary"
    events:
      - name: "sh.keptn.event.test.triggered"
        jsonpath:
          property: "$.data.test.teststrategy"
          match: "real-user"
    tasks:
      - name: "Wait for canary wait duration"
        image: "alpine"
        cmd: [ "sleep" ]
        args: [ "60s" ]
```

Лістинг файлу node1/locust/basic.py

```
from locust import HttpUser, between, task
```

```
class WebsiteUser(HttpUser):
    wait_time = between(5, 15)

    @task
    def index(self):
        self.client.get("/")
```

Лістинг файлу node1/locust/locust.conf

```
locustfile = /locust/locust.py
headless = true
users = 10
run-time = 1m
```

Лістинг файлу Imitate-Activity.ps1

```
[array]$envs = "dev","stage","prod"
$base_uri = "https:// helloservice.test.cloud/node1-"
While ($true) {
  foreach ($env in $envs) {
    $req_uri = $base_uri + $env + "-canary/api/version?sleep=" + $(Get-Random
-Minimum 0 -Maximum 600)
    Write-Host $req_uri
    $response = Invoke-WebRequest -Uri $req_uri -UseBasicParsing
    Write-Host $response.Content
    $req_uri = $base_uri + $env + "/api/version?sleep=" + $(Get-Random -
Minimum 0 -Maximum 200)
    Write-Host $req_uri
    $response = Invoke-WebRequest -Uri $req_uri -UseBasicParsing
    Write-Host $response.Content
  }
}
```