

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ЕКОНОМІКИ І МЕНЕДЖМЕНТУ

Кафедра кібернетики та інформатики

КВАЛІФІКАЦІЙНА РОБОТА

за другим (магістерським) рівнем вищої освіти

на тему: Дослідження методів та інструментів організації управління
прикладним програмним інтерфейсом мікросервісів

Виконав:

Іван ЛАПЧЕНКО

Група:

ІСТ 2401м

Науковий керівник:

Юлія РУДЕНКО

Рецензент:

Неля ДЕГТЯРЬОВА

Суми – 2025

СУМСЬКИЙ НАЦІОНАЛЬНИЙ АГРАРНИЙ УНІВЕРСИТЕТ

Факультет Економіки і менеджменту
Кафедра Кібернетики та інформатики
Освітній ступінь «Магістр»
Спеціальність 126 «Інформаційні системи та технології»
 ОП «Інформаційні системи та технології»

ЗАТВЕРДЖУЮ:
 Завідувач кафедри
 В.о. Ігор ШЕЛЕХОВ
 « ____ » _____ 2025 р.

підпис

ЗАВДАННЯ

на кваліфікаційну роботу
Івана ЛАПЧЕНКА

- Тема роботи:** Дослідження методів та інструментів організації управління прикладним програмним інтерфейсом мікросервісів
- Керівник кваліфікаційної роботи:** *Юлія РУДЕНКО_д.п.н., доцент*
- Строк подання здобувачем проєкту (роботи)** «3» грудня 2025 року
- Вихідні дані до кваліфікаційної роботи (роботи):** Результати аналітичних досліджень зарубіжних та вітчизняних вчених, нормативні документи та чинне законодавство, Інтернет-джерела.
- Зміст розрахунково-пояснювальної записки:**
 - виконати постановку задачі;
 - вибрати середовище для реалізації програмного додатку;
 - провести аналіз наявних рішень для вирішення проблеми;
 - розробити програмний продукт;
 - розрахувати економічну ефективність розробки програмного продукту.
- Перелік графічного матеріалу:** 15 таблиць, 14 рисунків

Керівник роботи

підпис

Ім'я ПРІЗВИЩЕ

Завдання прийняв до
виконання

підпис

Ім'я ПРІЗВИЩЕ

Дата отримання
завдання

« ____ » _____ 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту(роботи)	Строк виконання етапів проекту	Примітка
1	Визначення об'єкту, предмету дослідження, формулювання мети та задач кваліфікаційної роботи, складання плану	<i>січень 2025 р.</i>	виконано
2	Підбір та вивчення літературних джерел, технічної документації	<i>січень 2025 р.</i>	виконано
3	Розробка технічного завдання	<i>січень 2025 р.</i>	виконано
4	Розробка методів та інструментів організації управління прикладним програмним інтерфейсом мікросервісів	<i>січень - березень 2025 р.</i>	виконано
5	Створення концептуальної архітектури інструменту керування прикладним програмним інтерфейсом	<i>серпень-вересень 2025 р.</i>	виконано
6	Оформлення пояснювальної записки	<i>вересень-листопад 2025 р.</i>	виконано
7	Оформлення презентації	<i>листопад 2025 р.</i>	виконано
8	Опрацювання зауважень керівника, перевірка на автентичність	<i>до 3 грудня 2025 р.</i>	виконано
9	Підготовка доповіді та її попередній захист	<i>5 грудня 2025 р.</i>	виконано
10	Перевірка з академічної доброчесності кваліфікаційної роботи	<i>3 грудня 2025 р.</i>	виконано
11	Представлення кваліфікаційної роботи на кафедрі, деканат	<i>11 грудня 2025 р.</i>	виконано
12	Захист кваліфікаційної роботи	<i>до 12 грудня 2025 р.</i>	виконано

Керівник роботи

підпис

Ім'я ПРІЗВИЩЕ

Здобувач

підпис

Ім'я ПРІЗВИЩЕ

АНОТАЦІЯ

Лапченко І. В. Дослідження методів та інструментів організації управління прикладним програмним інтерфейсом мікросервісів.

Кваліфікаційна робота зі спеціальності 126 «Інформаційні системи та технології» ОП Інформаційні системи та технології Сумського національного аграрного університету. Суми, 2025 р. – Рукопис.

Кваліфікаційна робота присвячена дослідженню методів та інструментів організації управління прикладним програмним інтерфейсом (API) у середовищі мікросервісної архітектури та розробці програмного рішення для автоматизації цього процесу. Представлено модель інструменту управління API, який забезпечує автоматичне формування конфігурації для API-шлюзу на основі задекларованого REST-інтерфейсу мікросервісів. Розроблений інструмент дозволяє динамічно змінювати маршрутизацію, підвищує надійність системи та прискорює процес розгортання хмарних застосунків.

Об'єктом дослідження є комунікація між мікросервісами у хмарному середовищі, предметом – методи та інструменти організації керування API.

Метою роботи є розробка програмна реалізація інструменту управління API, здатного автоматизувати конфігурацію API-шлюзу в мікросервісній архітектурі.

Для підтвердження працездатності інструменту проведено апробацію в тестовому хмарному середовищі з аналізом продуктивності та надійності. Результати показують, що розроблене рішення забезпечує стабільність, правильність маршрутизації та зниження витрат на підтримку API.

Результатом роботи є програмний продукт, що дозволяє оптимізувати та автоматизувати управління API у хмарних застосунках, забезпечуючи гнучкість, масштабованість та підвищений рівень керованості мікросервісних взаємодій.

Ключові слова: мікросервіси, API-gateway, хмарні обчислення, управління API, Kubernetes, REST, Envoy Proxy, PaaS.

SUMMARY

Lapchenko I. V. Research on methods and tools for organizing the management of the application program interface of microservices.

Qualification work in the specialty 126 "Information Systems and Technologies" of the Information Systems and Technologies Department of the Sumy National Agrarian University. Sumy, 2025 - Manuscript. Lapchenko I. V. Information subsystem for accounting for environmental protection measures.

The qualification work is devoted to the study of methods and tools for organizing the management of the application program interface (API) in the microservice architecture environment and the development of a software solution for automating this process. A model of the API management tool is presented, which provides automatic configuration formation for the API gateway based on the declared REST interface of microservices. The developed tool allows you to dynamically change routing, increases system reliability and speeds up the process of deploying cloud applications.

The object of the study is communication between microservices in the cloud environment, and the subject is methods and tools for organizing API management.

The purpose of the work is to develop a software implementation of an API management tool capable of automating the configuration of an API gateway in a microservice architecture.

To confirm the operability of the tool, testing was carried out in a test cloud environment with an analysis of performance and reliability. The results show that the developed solution provides stability, correct routing and reduced costs for API support.

The result of the work is a software product that allows you to optimize and automate API management in cloud applications, providing flexibility, scalability and an increased level of manageability of microservice interactions.

Keywords: microservices, API gateway, cloud computing, API management, Kubernetes, REST, Envoy Proxy, PaaS.

ЗМІСТ

ВСТУП.....	10
РОЗДІЛ 1 АНАЛІЗ МЕТОДІВ РОЗРОБКИ МІКРОСЕРВІСІВ.....	12
1.1 Сутність та переваги віртуалізації.....	12
1.2.Моделі обслуговування хмарних обчислень PaaS рішень	14
1.3 Мікросервіси. Мікросервісна архітектура	18
1.4 Аналіз підходів до управління прикладним програмним інтерфейсом мікросервісів.....	20
РОЗДІЛ 2 МЕТОДИ І ТЕХНОЛОГІЇ РОЗРОБКИ ТА РОЗГОРТАННЯ ІНСТРУМЕНТІВ КЕРУВАННЯ API МІКРОСЕРВІСІВ.....	26
2.1 Методи керування прикладним програмним інтерфейсом	26
2.2 Технології розробки мікросервісів.....	31
2.3 Методи розгортання мікросервісів у хмарі	33
РОЗДІЛ 3. МОДЕЛЬ ІНСТРУМЕНТУ УПРАВЛІННЯ API НА ОСНОВІ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ	38
3.1.Концептуальна архітектура інструменту керування API	38
3.2 Розробка моделі інструменту управління API.....	40
РОЗДІЛ 4 АПРОБАЦІЯ МОДЕЛІ ІНСТРУМЕНТУ КЕРУВАННЯ ПРИКЛАДНИМ ПРОГРАМНИМ ІНТЕРФЕЙСОМ.....	50
4.1 Процес впровадження інструменту керування прикладним програмним інтерфейсом	50
4.2 Процес інтеграції інструменту прикладного програмного інтерфейсу з готовим проектним рішенням.....	56
4.3 Розрахунок надійності розробленого інструменту.....	57
ВИСНОВОК	60
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ.....	61
ДОДАТКИ.....	64
Додаток А.....	65
Додаток Б	78
Додаток В.....	82
Додаток Г	86

ВСТУП

Проблема управління API мікросервісів у хмарних середовищах, можливість його значного спрощення при правильно спроектованому архітектурному підході має значення в умовах цифрової трансформації суспільства. Вона піднімається з погляду як розробника ПЗ, а й людини, управляючого проектом і відповідального за його архітектуру.

Управління API є частиною мікросервісної архітектури, і є передовою технологією розробки корпоративних додатків. При проектуванні програмного продукту з використанням мікросервісної архітектури гостро постає питання публікації інтерфейсу, з метою приховати кінцеву реалізацію та надати можливість заміни реалізації та гнучкість у розподілі ролей та функцій мікросервісів. Цю проблему вирішують спеціальні інструменти керування API. На додаток до цього ці інструменти можуть брати на себе зобов'язання щодо безпеки комунікації, надавати можливість відстежувати та збирати статистику використання API. Перед розробниками та архітекторами стоять проблеми, вирішення яких ще тільки потрібно знайти, це зумовлює актуальність теми дослідження.

Об'єктом дослідження є комунікація між мікросервісами у хмарному середовищі.

Предметом дослідження методи та інструменти організації керування API.

Метою дослідження є програмна реалізація інструменту управління API, здатного автоматизувати конфігурацію API-шлюзу в мікросервісній архітектурі.

Для досягнення поставленої мети потрібно вирішити такі завдання:

- проаналізувати підходи до організації управління API та мікросервісної комунікації;
- проаналізувати способи реалізації інструментів організації управління API та мікросервісної комунікації;

- розробити модель інструменту організації управління API;
- підтвердити ефективність запропонованої моделі практично.

Наукова новизна дослідження полягає в розробці моделі інструменту, що надає можливості простого управління API мікросервісів і автоматичне налаштування мікросервісної взаємодії.

Практична значущість роботи полягає у розробці інструменту управління API у хмарних середовищах, який знизить вартість та час розробки хмарних додатків.

Методи дослідження: методи та моделі розгортання компонентів інструменту управління API, об'єктно-орієнтований підхід до аналізу та проектування інструменту.

Апробація результатів дослідження. Наукові здобутки науково-дослідної роботи були представлені на науково-практичних конференціях: IX Міжнародній студентській науковій конференції, м.Рівне, 28листопада, ГО «Молодіжна наукова ліга»; II Міжнародній науково-практичній студентській конференції (15 жовтня 2025 року, м. Харків, ХНУ імені В. Н. Каразіна.

1. Лапченко І. В. Використання контейнеризації у розробці та розгортанні мікросервісів Актуальні питання та перспективи проведення наукових досліджень: матеріали IX Міжнародної студентської наукової конференції, м.Рівне, 28листопада, 2025рік / ГО«Молодіжна наукова ліга».— Вінниця: ТОВ «УКРЛОГОСГруп», 2025. —С. 80.

2. Лапченко І. В. Високодоступна архітектура мікросервісів ІТ-простір сьогодення: тенденції, інновації та перспективи розвитку: збірник тез доповідей II Міжнародної науково-практичної студентської конференції (15 жовтня 2025 року, м. Харків, Україна) [Електронний ресурс]. – Харків : ХНУ імені В. Н. Каразіна, 2025. С.395

Структура і обсяг магістерської роботи. Робота складається зі вступу, 4 розділів, висновків, а також списку використаних джерел, що нараховує 34 найменування. Загальний обсяг роботи становить 82 сторінки. Робота містить 18 таблиць, 22 рисунки.

РОЗДІЛ 1

АНАЛІЗ МЕТОДІВ РОЗРОБКИ МІКРОСЕРВІСІВ

1.1 Сутність та переваги віртуалізації

Хмарні обчислення ґрунтуються на технологіях віртуалізації, завдяки яким забезпечуються гнучкість, масштабованість, доступність, ізолюваність і раціональне використання обчислювальних ресурсів. Віртуалізація як технологічний підхід бере свій початок у 1960-х роках, коли компанія IBM розробила перші віртуальні машини для забезпечення одночасної роботи багатьох користувачів на великих мейнфреймах. На початку XXI століття, зі зростанням продуктивності та здешевленням комп'ютерної техніки, інтерес до віртуалізації значно посилювався, що сприяло її широкому впровадженню в IT-інфраструктуру різних масштабів.

Віртуалізація визначається як процес створення віртуальних ресурсів, ізолюваних від фізичного апаратного забезпечення за допомогою спеціального програмного шару — гіпервізора. До віртуальних ресурсів можуть належати сервери, робочі станції, операційні системи, мережеве обладнання, файлові сховища та програмні середовища. Основною метою віртуалізації є ефективний розподіл обчислювальних потужностей між багатьма користувачами або сервісами шляхом абстрагування фізичного обладнання та трансформації його ресурсів у віртуальні [12].

Найпоширенішим результатом застосування технологій віртуалізації є створення віртуальних машин (VM). Віртуальна машина функціонує як ізольоване програмне середовище з власною гостьовою операційною системою та набором віртуальних апаратних компонентів [6].

Переваги використання віртуальних машин

- можливість запуску застарілого програмного забезпечення та гостьових ОС, несумісних із сучасним апаратним забезпеченням;
- забезпечення безпечного користувацького середовища для

роботи у мережі: атаки або збої впливають лише на ВМ, а не на фізичну систему;

- ізоляція декількох віртуальних машин на одному фізичному сервері, що унеможливорює вплив збоїв однієї ВМ на інші;
- мобільність і переносність: віртуальні машини можна копіювати, переносити та відновлювати як звичайні файли;
- незалежність від конфігурації хост-комп'ютера: параметри ВМ можуть відрізнятися від реальних апаратних характеристик;
- підтримка емуляції апаратних компонентів, що дозволяє відтворювати пристрої, яких фізично немає;
- можливість створення «знімків» (snapshot), резервних копій та швидкого відкату системи до попереднього стану;
- можливість одночасного розгортання багатьох гостьових ОС, що взаємодіють у єдиному віртуальному середовищі.

Недоліки віртуальних машин

- потреба у значних апаратних ресурсах хост-машини, особливо при одночасному запуску багатьох ВМ;
- потенційне зниження продуктивності гостьової ОС порівняно з роботою безпосередньо на фізичному обладнанні;
- обмеження, пов'язані з частковою або неповною підтримкою деяких апаратних інтерфейсів у різних типах гіпервізорів.

У сучасних ІТ-системах застосовуються різні типи віртуалізації:

1. Віртуалізація операційної системи Передбачає одночасний запуск однієї або кількох гостьових операційних систем на одній фізичній машині. Найпоширеніший тип віртуалізації, що широко використовується у хмарних сервісах.

2. Віртуалізація програмного забезпечення Забезпечує запуск окремих програм у відокремленому середовищі без необхідності створення повноцінної віртуальної машини.

3. Віртуалізація мережі Об'єднує апаратні та програмні компоненти у

єдине логічне середовище [11].

Розрізняють зовнішню віртуалізацію — об'єднання декількох фізичних мереж у віртуальну і внутрішню — створення віртуальних мереж між ВМ або контейнерами в межах однієї хост-системи.

4. Віртуалізація апаратного забезпечення

Передбачає сегментацію або повну емуляцію фізичного обладнання, доступного гостьовим ОС.

5. Віртуалізація систем зберігання Вона набуває двох форм:

- файлова віртуалізація — відображення розташування файлів у розподілених системах зберігання;
- блочна віртуалізація — використання технологій RAID, iSCSI та інших для створення єдиного віртуального середовища зберігання.

Методи віртуалізації поділяють на програмну, апаратну та контейнеризацію. Як зазначає Д. Бернс, контейнеризація стала найбільш ефективним сучасним підходом завдяки мінімальному споживанню ресурсів та відсутності потреби в емуляції апаратних компонентів [9]. Проте, як наголошує А. Кислюк, її обмеженням є залежність контейнера від ядра хост-операційної системи [10].

Отже, віртуалізація як процес створення віртуальних ресурсів, ізольованих від фізичного апаратного забезпечення має свої переваги, недоліки види та методи.

1.2. Моделі обслуговування хмарних обчислень PaaS рішень

Хмарні обчислення розглядаються як сучасна модель надання обчислювальних ресурсів за запитом користувача без його безпосередньої участі в адмініструванні інфраструктури. На думку М. Армстронга, хмарні сервіси стали ключовою технологією цифрової трансформації завдяки доступності, гнучкості та ефективному використанню ресурсів [1]. Такі ресурси охоплюють сховища даних, обчислювальні потужності та мережеві

сервіси. У роботах Т. Ерла зазначається, що хмарні обчислення формують архітектуру глобальних дата-центрів, орієнтованих на багато користувачів [2].



Рисунок 1.1 – Моделі комп'ютерних обчислень у вигляді шарів

Джерело: [розроблено автором]

Поява моделі «pay-as-you-go» стала можливою завдяки здешевленню вартості одиниці обчислювальних ресурсів, що, за Д. Лінксом, сприяло переходу ІТ-індустрії до нових економічних моделей споживання [3].

Національний інститут стандартів і технологій США (NIST) визначає п'ять базових характеристик хмарних обчислень [4]:

1. Самообслуговування на вимогу. Користувач може самостійно ініціювати виділення ресурсів без взаємодії з адміністратором провайдера.
2. Широкий мережевий доступ. Послуги доступні з будь-яких клієнтських платформ, що забезпечує мобільність і універсальність [2].
3. Об'єднання ресурсів. Як зазначає Р. Коутс, пул спільних ресурсів забезпечує динамічний їх розподіл між користувачами залежно від навантаження [5].
4. Еластичність. Швидке масштабування ресурсів в автоматичному режимі.

5. Облік споживання. Точний облік використаних ресурсів, що дає змогу реалізувати гнучку модель ціноутворення [6].

Хмарні обчислення реалізуються через три моделі сервісів: SaaS, PaaS, IaaS, упорядковані у вигляді ієрархії над фізичною інфраструктурою [2].

Software-as-a-Service (SaaS). За твердженням Дж. Сміта, ця модель передбачає доступ користувача до прикладного програмного забезпечення провайдера без контролю над інфраструктурою [7]. Управління серверами, мережевими компонентами й ОС здійснюється повністю провайдером.

Platform-as-a-Service (PaaS). На думку Д. Бернса, PaaS є ключовою моделлю для розробників програмного забезпечення, оскільки забезпечує повний набір інструментів для створення, тестування та розгортання програм [8]. Платформа включає СУБД, middleware, середовища виконання мов програмування. Керування інфраструктурою здійснює провайдер, тоді як користувач керує застосунками та частково — конфігурацією платформи.

Infrastructure-as-a-Service (IaaS). Ця модель, за визначенням Голдберга, дозволяє користувачу контролювати операційні системи, сховища, мережеві налаштування та інсталивати будь-яке програмне забезпечення [9].

Провідні хмарні сервіси (AWS, Azure, Google Cloud, Alibaba Cloud) підтримують роботу з контейнерними платформами, такими як Docker, Kubernetes та OpenShift [8].

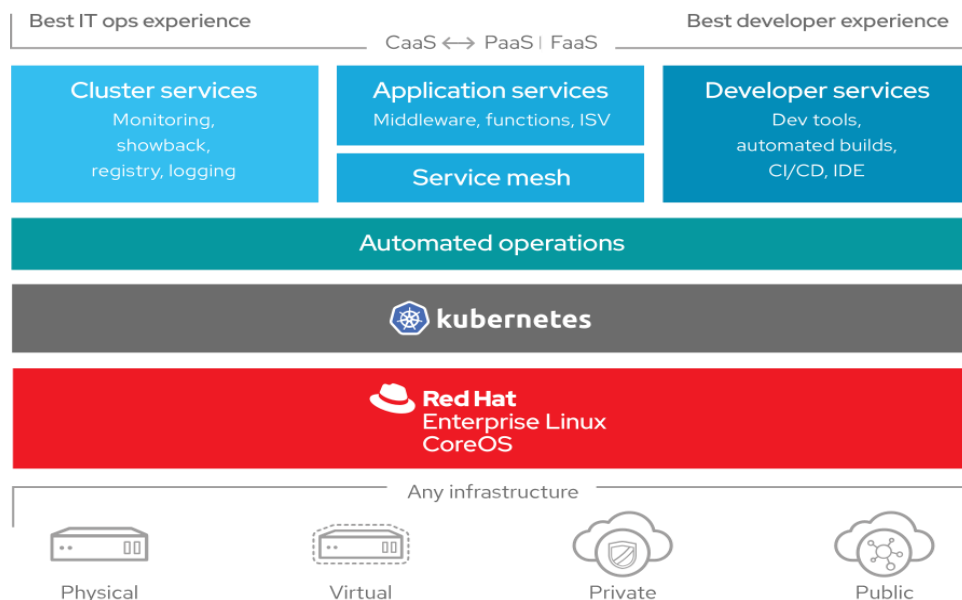


Рисунок 1.2 – Структура OpenShift

Джерело: [5]

Docker забезпечує контейнеризацію застосунків, що, за А. Кислюком, спрощує стандартизоване розгортання програмних рішень [10].

Kubernetes, як зазначає Бернс (2022), є найбільш поширеним оркестратором контейнерів, який забезпечує автоматизацію масштабування, балансування навантаження, відновлення та керування конфігураціями [8].

OpenShift реалізує повноцінну модель PaaS, поєднуючи Kubernetes, Docker та ОС Red Hat Enterprise Linux у єдине середовище [11].

У розділі було проаналізовано сутність хмарних обчислень, їх ключові характеристики та сучасні моделі надання хмарних сервісів, що визначені міжнародними стандартами. Встановлено, що хмарні технології формують інноваційну інфраструктуру цифрової економіки, забезпечуючи гнучке, масштабоване та економічно ефективне використання обчислювальних ресурсів. Особлива увага приділена моделям SaaS, PaaS та IaaS, у межах яких відбувається надання програмних, платформних та інфраструктурних сервісів. Показано, що платформа як послуга (PaaS) є ключовим середовищем для створення мікросервісів і розроблення бізнес-логіки, а такі технології, як Docker, Kubernetes та OpenShift, становлять основу сучасних хмарних рішень та забезпечують високу сумісність, автоматизацію й

ефективність розгортання програмних систем.

1.3 Мікросервіси. Мікросервісна архітектура

Оскільки метою науково-дослідної роботи є теоретичне обґрунтування, і практична реалізація моделі автоматизованого управління прикладним програмним інтерфейсом та конфігурації мережі мікросервісів варто зупинитися на тому, що таке мікросервіс, в чому полягає мікросервісна архітектура додатків.

Щоб краще зрозуміти, чим є мікросервіси, варто розглянути архітектурні стилі, що використовуються до появи мікросервісів і те, як архітектура корпоративних додатків еволюціонувала в мікросервісну. Найкращий спосіб зрозуміти ключові моменти та характеристики мікросервісів – це детально розглянути еволюцію архітектур корпоративних систем з монолітних додатків до мікросервісів.

Програмні програми рівня підприємства проектуються щоб полегшити реалізацію численних бізнес-вимог.

В архітектурі моноліту всі бізнес-функціональності зібрані всередині одного монолітного додатка і упаковані в один єдиний модуль. Щоб покращити розуміння наведемо реальний приклад монолітної програми, виберемо в цій якості інтернет-магазин (див. рисунок 1.3)

В цілому інтернет-магазин є набором декількох компонентів, таких як управління замовленнями, платежі, управління продуктом і так далі.

Кожен із цих компонентів пропонує широкий набір бізнес-функцій.

Додавання або модифікація функціональних можливостей компонент дуже дорого через його монолітної природи.

Також, щоб покрити всі бізнес вимоги, ці компоненти повинні обмінюватися між собою даними.

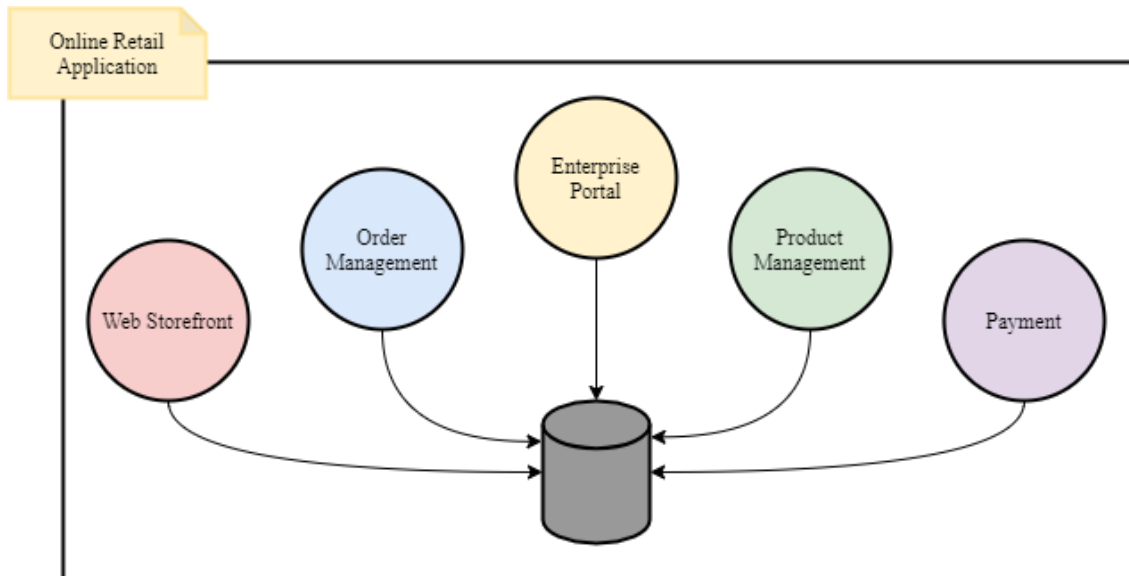


Рисунок 1.3. – Додаток для інтернет магазину, розроблений за допомогою мікросервісів [6]

Комунікація між компонентами часто будується на пропрієтарних протоколах та стандартах і засновані на схемі взаємодії крапка-точка.

Тому зміна або заміна цього компонента також пов'язані з великими труднощами.

Наприклад, якщо магазин хотів би використовувати іншу систему управління замовленнями, то здійснення такої вимоги зазнало величезної кількості змін у кількох існуючих компонентах. Підсумуємо основні характеристики монолітних додатків:

- проектуються, реалізуються та розгортаються як єдиний модуль/пакет;
- переважна складність реалізацій більшості реальних бізнес сценаріїв, що перетворює обслуговування, оновлення та додавання нових функцій у нічний жах;
- важко підходить для практики методологій гнучкої розробки та доставки додатків замовнику. Так як додаток збирається в один великий модуль, більшість бізнес-функцій не має свого виділеного життєвого циклу;

- зі зростанням монолітного додатку зростає час його запуску та вартість такого запуску;
- один нестабільний сервіс може вивести з ладу всю програму;
- у разі потреби може бути масштабований тільки як повноцінний додаток (немає можливості зробити це частинами), що зробити складно у випадках неоднорідної вимоги компонентів до ресурсів. Наприклад, одна бізнес-функціональність вимагає більше часу CPU, в той час як інший необхідно більше пам'яті. У разі дуже складно задовольнити індивідуальні потреби кожного компонента;
- міграція на нові технології та бібліотеки скрутна, оскільки монолітна структура передбачає використання єдиної технології.

1.4 Аналіз підходів до управління прикладним програмним інтерфейсом мікросервісів

Будь-які мікросервісні програми, що розробляються, повинні представити бізнес-можливості для споживачів таким чином, щоб ці можливості можна було легко створювати, керувати ними, захищати, аналізувати та масштабувати. Такі можливості надаються споживачам як API, які регулюються процесом широко відомий як управління API (API management). Аналогічно, мікросервіси можуть використовувати зовнішні API у процесі своєї роботи. У більшості випадків API є синхронним обміном повідомленнями типу запит-відповідь. Будь-які бізнес-можливості, які надаються споживачеві (внутрішні чи зовнішні) можна вважати API. Розкривати такі бізнес-можливостей варто таким чином, щоб була можливість створювати, керувати, захищати, аналізувати та масштабувати це API. Ця можливість називається керуванням API. Використовуючи рішення з управління API, є можливість включити перевірку політик та ключів, управління версіями сервісів, управління квотами, примітивні перетворення, авторизацію та контроль доступу,

можливість бути спостереженим(observability), можливості самообслуговування, рейтинги тощо для кожного мікросервісу.

Коли справа доходить до управління API, є кілька основних компонентів, які повинні бути частиною будь-якого рішення:

- API Publisher;
- API Developer Portal / Store;
- API Gateway;
- API Analytics / Observability;
- API Monetization;
- API QoS (якість обслуговування, такі як безпека, регулювання, кешування тощо).

Зупинимось докладніше опис сервісу публікації API (API Publisher) і так званому шлюзі API (API Gateway). Сервіс публікації може мати більшу зону відповідальності, ніж просто розташування задекларованих API в шлюзі. Він може надавати всі необхідні можливості розробникам API для проектування, розробки, публікації, версіонування, управління, моніторингу доступності та вимірювання продуктивності.

API-шлюз є єдиною точкою входження для клієнта, розташованою над сервіс-орієнтованим бекендом. При публікації API міститься в API-шлюзі. Шлюз API захищає, керує та відповідає за масштабування дзвінків API. Всі запити, що надсилаються користувачами, перехоплюються шлюзом або надсилаються безпосередньо до нього. Це дозволяє використовувати різні політики для цілої групи API, такі як безпека, фільтри, статистика використання API. Крім того, в деяких випадках є можливість об'єднати багато бізнес-функцій на рівні шлюзу API.

Описані вище принципи управління API відкривають дуже добре накладаються на методологію гнучкої розробки та дозволяють реалізовувати конкретний бізнес-функціонал окремо від інших. Єдина проблема, яка стоїть перед розробниками на мікросервісній архітектурі – це реалізації цих шаблонів. Наприклад, існує чудова платформа для

проектування API - Swagger, але вона добре розвинена саме для цього і не надає відкритої можливості або будь-якої інтеграції з реалізаціями на API Gateway. Такі інтеграції зараз розроблені тільки для хмарних рішень на Amazon. Сама ж платформа Swagger є комерційним продуктом, це порушує питання про реальну корисність і вигоду продукту для конкретних випадків вирішення бізнес проблем. У реаліях новизни технології готові рішення часто свіжі і не можуть точно відповідати тим вимогам, які стоять перед розробниками через це реалізацію сервісів управління API. Або ж такі реалізації існують, але мають свою специфіку і зовсім інший погляд розробників такого продукту на реалізацію таких шаблонів і вартість такої реалізації дуже дорого, як дорого і впровадження у свій продукт.

Розвиток технології віртуалізації, зокрема контейнеризації дало поштовх і мікросервісної архітектурі.

На даний момент завдяки популярності цього підходу до побудови архітектури корпоративних програм реалізовано безліч інструментів для управління API.

Незважаючи на це реалізація більша частина сформованих архітектурою сервісів залишається на плечах розробників конкретної програми або платформи для розробки рішень.

Так, наприклад, немає еталонної реалізації сервісу публікацій API у його явному вигляді й у вільному доступі, тоді як різних реалізацій API-шлюзів велика різноманітність.

Від компанії до компанії, що займається проектуванням та реалізацією різноманітних програмних продуктів для бізнесу, здійснюються різні підходи до розробки. У цій роботі ми розглядатимемо модель, коли загальний набір завдань, які потрібно вирішити діляться на загальні та завдання бізнесу. Реалізація спільних завдань, до яких належать архітектурні послуги, та додатки обслуговування називається платформою. На реалізацію платформи зосереджено сили кількох команд

розробки. Платформа повинна бути однаково застосовна для різних бізнес-завдань і різних проектів, які реалізують ці бізнес-завдання. Розглянуті вище архітектурні шаблони проектуються і реалізуються командами, відповідальними за платформу. У переважній більшості саме проекти, що реалізують безпосередньо бізнес завдання, формують API та реалізують його у своїх мікросервісах, тоді як завдання з управління цього API кладеться на плечі команд платформи. Це кардинально відрізняється від процесів проектування, розробки та публікації API, які використовують у невеликих компаніях, у яких одна велика команда працює над цілісним проектом та використовує сукупність готових рішень як платформу. Таким невеликим проектам справді достатньо тих реалізацій, які може запропонувати IT співтовариство.

Розглянемо найпопулярніші реалізації API-шлюзу та реалізації управління API. До списку таких реалізацій входить:

- Spring Cloud Gateway,
- Kong Gateway,
- Swagger,
- NGINX,
- Envoy Proxy.

Розглядати ці реалізації варто з боку процесу управління API, яке вони пропонують, якщо це реалізація шаблону API Publisher, а також з точки зору процесу доставки конфігурації API API-шлюз.

Spring Cloud Gateway проект є частиною проекту Spring Cloud, метою якого є надати розробникам хмарних рішень максимальну інструментальну підтримку та позбавити їх необхідності реалізовувати те саме. Сам проект Spring Cloud Gateway надає бібліотеку для побудови шаблону API Gateway. Мета проекту – надати простий та ефективний спосіб маршрутизації запитів до API. Функціонал цього рішення з коробки пропонує статичний спосіб програмної декларації маршрутів, що робить його складним використовуючи як загальне і настроюване

рішення для різних проектів і додатків. Із очевидних плюсів – відкритий вихідний код.

Kong Gateway також є проектом з відкритим вихідним кодом та написаний мовою Lua. Надає користувачам повноцінну реалізацію шаблону «API Gateway» з функціями маршрутизації трафіку, аутентифікації, моніторингу, протоколювання, кешування, контролю трафіку. Має різні формати налаштувань, як статичних через конфігураційний файл, так і динамічну за допомогою запитів HTTP. Є хорошим з погляду продуктивності рішенням, оскільки заснований на "nginx". Але так як повністю складається з набору скриптів на Lua, то загальна продуктивність і явні недоліки з розміром пам'яті, що споживається, залишаються під питанням. До того ж налаштування за допомогою HTTP має свої недоліки, наприклад, стежити за актуальністю маршрутів в API-шлюзі без міцного зворотного зв'язку стає надзвичайно складно.

NGINX - являє собою реалізацію проксі-сервісу з багатою конфігурацією, на його основі побудований проект Kong, але також розробник NGINX пропонує з коробки безліч можливостей: балансування, кешування, протоколювання і т. д.

Swagger – це проект із відкритим вихідним кодом, метою якого є надати інструменти та платформу для проектування, тестування та документації програмним інтерфейсом додатків. Фокус уваги цього проекту зосереджений на веб-сервісах, побудованих на концепціях REST. Swagger є частковою реалізацією шаблону API Publisher. Swagger диктує свій процес розробки API, який починається з дизайну, а закінчується автоматичною генерацією коду на основі існуючої специфікації.

Envoy Proxy. Назва проекту говорить сама за себе, це реалізація проксі сервера мовою C. Цей проект пропонує великий список можливостей, але більш головне – широкі можливості щодо його налаштування. Цей проект часто використовується в особистих розробках

різних компаній як реалізацію шаблону API Gateway.

Перевагою проекту є його динамічна настройка і спеціальна бібліотека, що надається, для управління конфігурацією проксі-сервера з поза. Такий шар у термінології Envoy proxy є "Control plane". Протокол управління конфігурацією будується на gRPC. Це спрощує реалізацію моделі видавця-передплатника у конфігуруванні envoy proxy

Таким чином, аналіз показує суттєву роль платформного підходу, коли архітектурні сервіси реалізує окрема платформа, тоді як бізнес-логіка втілюється у мікросервісах, що формують API. Це підкреслює важливість вибору відповідних інструментів управління API та їх інтеграції в корпоративну інфраструктуру для забезпечення надійності, масштабованості та ефективності сучасних програмних систем.

РОЗДІЛ 2

МЕТОДИ І ТЕХНОЛОГІЇ РОЗРОБКИ ТА РОЗГОРТАННЯ ІНСТРУМЕНТІВ КЕРУВАННЯ API МІКРОСЕРВІСІВ

2.1 Методи управління API

Управління прикладним програмним інтерфейсом – це підхід до організації мікросервісної архітектури, який забезпечує формування чіткого контракту для кожного сервісу та приховує від користувача внутрішню реалізацію конкретного мікросервісу. Крім того, управління API виступає точкою розширення для реалізації важливих функцій усередині мікросервісного застосунку, таких як інтроспекція авторизаційних токенів, обмеження трафіку, балансування навантаження та інші механізми забезпечення безпеки й надійності.

Так як підхід до управління API і сама технологія хмарних середовищ і мікросервісна архітектура досить «молоді» напрямки в розробці прикладного програмного забезпечення, то й методів такого підходу існує не так багато. Однак є два методи, які досить прості та мають широке застосування у різних програмних рішеннях. Перший і найпростіший метод полягає в ручному налаштуванні API-шлюзу (див. рис. 2.1).

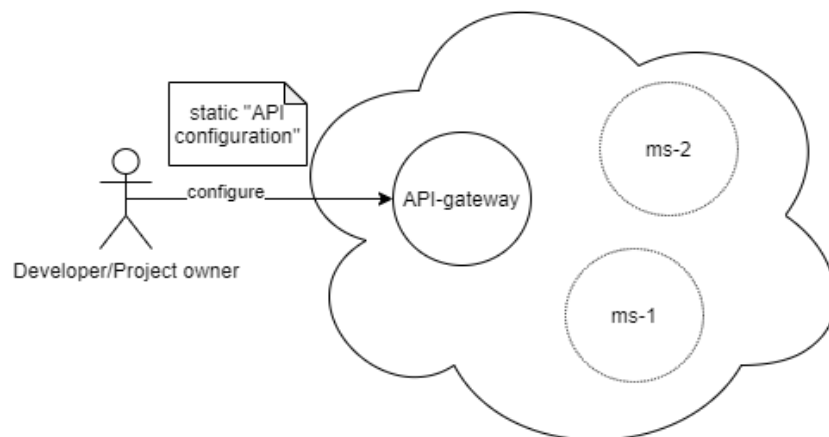


Рисунок 2.1 - Метод ручного управління API [6]

Як API-шлюз вибирається будь-який програмний продукт з можливостями маршрутизації мережевого трафіку, а також можливістю гнучкого налаштування цих маршрутів. Як API-шлюз може виступати навіть простий проксі-сервер NGINX, налаштування якого здійснюється за допомогою спеціального конфігураційного файлу. Однак для здійснення такого методу управління API необхідні глибокі знання в налаштуванні API-шлюзу. Конкретний підхід у налаштуванні API шлюзу, звичайно, залежить від вибраного програмного продукту, що реалізує функції API-шлюзу, але досить часто такий підхід виявляється у конфігураційному файлі з конкретним контрактом. Контракт налаштування може абсолютно відрізнятися від одного програмного продукту до іншого і здебільшого складний в освоєнні. Тому очевидним недоліком такого підходу є, по-перше, складність у налаштуванні API-шлюзу, по-друге, консервативність конфігурації до динамічних змін маршрутів, що виключає можливість динамічного перенаправлення мережевого трафіку без переведення всього хмарного рішення в режим обслуговування. Також такий метод застосовується у відносно невеликих за функціональними можливостями програмних рішеннях у хмарі, оскільки зі зростанням можливостей зростає і кількість мікросервісів і активно розширюється API. Вручну керувати та стежити за актуальністю API, а також вручну змінювати конфігурацію API-шлюзу стає трудомістким завданням та неефективним витрачанням людських ресурсів. Плюсом такого підходу є гнучке налаштування API-шлюзу, обмежене можливостями програмного продукту, що його реалізує.

Розглянемо наступний метод управління API, який широко застосовується в компанії NetCracker. Цей метод зображено на рис. 2.2. Суть методу у наявності двох додаткових складових в архітектурі управління API, а саме сервісу публікації API та спеціальної бібліотеки. Бібліотека використовується як залежність у мікросервісах і при старті

мікросервісу збирає всі метадані з вихідного коду самого мікросервісу (наприклад анотації Java), формуючи далі запит зміну конфігурації в API-шлюзі. Цей запит відправляється в API-публікатор, де готується, трансформується і відправляється безпосередньо в сам API-шлюз. У застосуванні такого методу є переваги:

- API-шлюз налаштовується в автоматичному режимі, що унеможливорює прості та дурні помилки;
- не потрібно стежити за актуальністю конфігурації API в API-шлюзі, так як при кожному старті програми в хмарі актуальна конфігурація відразу буде відправлена в сервіс публікації API і сформована актуальна конфігурація;
- щоб уникнути використання бібліотеки, опис API мікросервісу можна відправити в публікатор вручну.

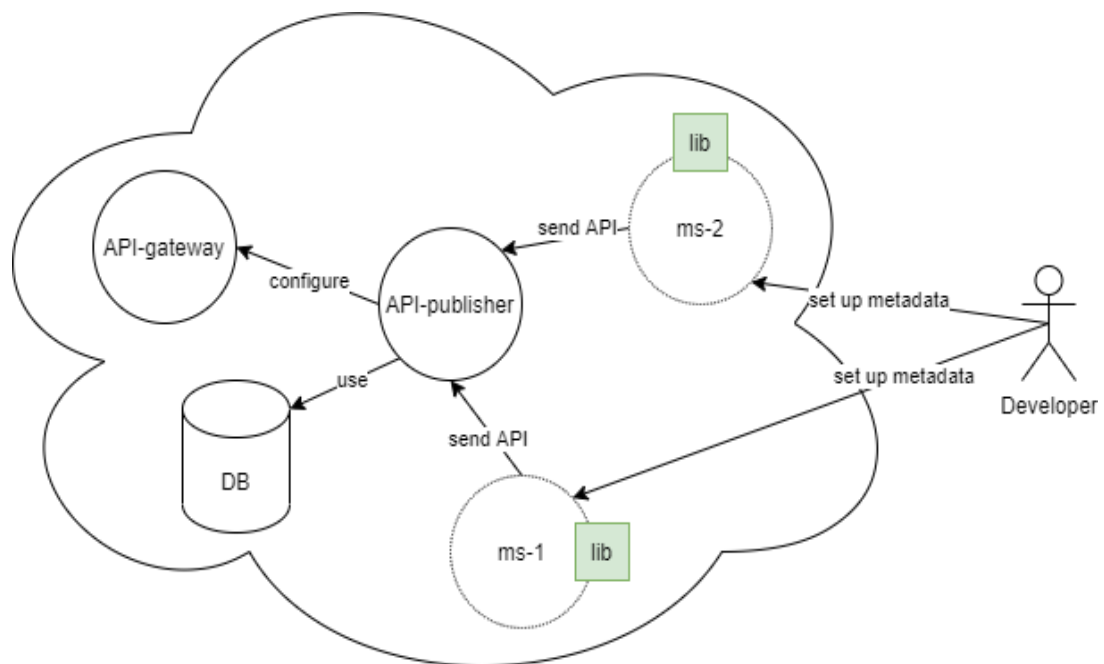


Рисунок 2.2 – Метод авто-конфігурації з метаданими і сховищем [6]

Такий метод не позбавлений недоліків. Бібліотеку для мікросервісів має бути реалізована багатьма мовами, оскільки мікросервісна архітектура не накладає обмежень на мову, розробники можуть

використовувати будь-яку, і щоб скористатися перевагами цього методу, бібліотека повинна бути реалізована на всіх. Також, у цьому недоліку можна знайти ще й технічні обмеження. Не всі мови програмування мають гарні засоби рефлексії коду. І часто використання рефлексії погіршує продуктивність. Наступним істотним недоліком є те, що конфігурація, яка формується та зберігається в публікаторі API (для подальшого поширення в API-шлюз) має життєвий цикл лише на додавання та оновлення списку API. Інакше кажучи, з погляду публікатора неможливо оцінити, які API втратили актуальність і мають бути видалені з конфігурації. Цей недолік накладає обов'язок на власника продукту стежити за актуальністю конфігурації та видаляти із публікатора API, які вже не використовуються. Цей метод, при додаванні публікатора, створює залежність від бази даних або зовнішнього сховища, через що при їхньому збої весь ланцюг управління API стає недієздатним, втрачаючи можливість конфігурації, а при перезапуску API-шлюзу — також і маршрутизацію трафіку.

Аналіз розглянутих методів та оцінка їхніх переваг і недоліків свідчать, що ці підходи не забезпечують поставленої мети зручності використання реалізацій, розширюваності та доступності. Зокрема, перший метод передбачає ручне налаштування API-шлюзу, яке вимагає спеціалізованих знань щодо конфігурації програмного забезпечення з функціями проксіювання запитів.

До того ж, ручний метод унеможливорює динамічну зміну стану API-шлюзу, що для продуктового рішення стає неприйнятним. Другий метод покриває недоліки першого, але приносить нові. Очевидно, що додаткова ланка в архітектурі у вигляді сховища даних зобов'язує користувача цього методу додатково стежити за надійністю сховища даних, а також має проблеми в самій реалізації, оскільки виникає проблема конфігурації. Проблема змінюваності конфігурації полягає в тому, що користувачу не очевидно в якому, на даний момент, стані

знаходиться конфігурація, якщо дозволено динамічно змінювати різні її частини різним користувачам. Ця проблема вирішується наявністю статичної, наочної конфігурації. Щоб уникнути недоліків розглянутих методів, передовсім необхідно сформулювати спрощений контракт конфігурації по відношенню до конфігурації API-шлюзу, який у вигляді конфігураційного файлу буде зберігатися в сховищі даних, за доступність якого відповідає шар PaaS.

Такий метод розроблено в рамках цієї науково-дослідної роботи на основі знань про інфраструктуру PaaS шарів та їх технологічних можливостей та зображено схематично на рисунку 2.3.

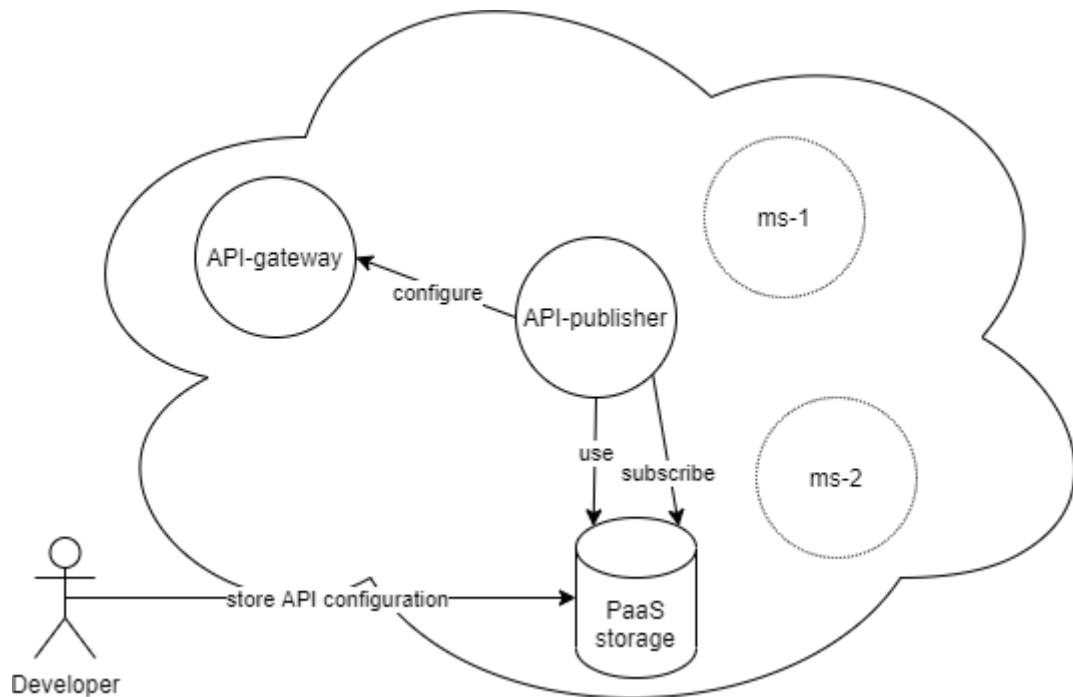


Рисунок 2.3 – Метод управління API з перевагами використання PaaS [6]

Метод розширює і доповнює можливості методу вище, розробленого в компанії NetCracker, проте позбавляється очевидних недоліків і додає нові точки розширення.

Суть цього методу в тому, щоб як сховище конфігурації використовувати вбудований PaaS шар механізм. Він є надійним,

оскільки сам PaaS шар на ньому і побудований. Як реалізацію цього сховища використовується etcd. Це сховище організовує дані за принципом ключ-значення та надає цілісність даних та їх доступність на вимогу. Також PaaS шар надає можливість відстежувати зміни даних конкретного типу. У цьому методі публікатор підписується на відстеження конкретних типів даних, які називаються конфігурацією API, в PaaS шарі і при кожній зміні здійснює повне перескладання всієї конфігурації API-шлюзу.

Цей метод усуває недоліки у вигляді залежності від зовнішнього сховища даних, оскільки вже залежить безпосередньо від сховища, вбудованого в PaaS шар. Однак сховище PaaS шару невіддільне від нього і поставляється відразу разом з ним, що означає, що скрізь, де є цей PaaS шар - значить є і це сховище. Загалом, складність публікатора в такому методі значно спрощується, він стає простим.

«Конвертер» простого формату конфігурації API-шлюзу в більш складний. Необхідність у бібліотеці відпадає, але конфігурацію необхідно створювати вручну та зберегти її в PaaS шарі.

2.2 Технології розробки мікросервісів

Як і під конкретну платформу, архітектуру процесора, операційну систему є своя специфіка розробки, так і для хмарного середовища є своя. Незважаючи на те, що контейнеризація дає широкий вибір мов програмування для реалізації мікросервісів, зупинимось на мовах, адаптованих для роботи в хмарному середовищі - Java та Golang.

Java на даний момент одна з найпопулярніших мов для розробки корпоративних додатків та й загалом у галузі прикладної розробки має лідируючі позиції за рахунок великого співтовариства, багатого вибору бібліотек та готових рішень. У сукупності рахунок швидкості розробки і досить хороші показники продуктивності ця мова зараз і обирається як

основний інструмент реалізації програмних продуктів для бізнесу. Для швидкого старту в хмарі мікросервісу, написаного на java, існує бібліотека Spring Boot, вона дозволяє з мінімальною кількістю коду (1 клас) запустити повноцінний HTTP сервіс, що обробляє запити і це все запаковано в один великий пакет з байт-кодом і готовим набором бібліотек. Все, що потрібно, щоб запустити цю програму – Java машина, встановлена в середовищі запуску.

Водночас Java та, зокрема, бібліотека Spring Boot, не позбавлені недоліків. Використання цієї комбінації потребує великого часу старту та великого обсягу пам'яті. Хоча і на середньостатистичному комп'ютері час старту Spring Boot програми займає 1 секунду - це все рівно досить багато для програми, яка нічого не робить. З додаванням так званих бінів (Bean) додаток збільшує час. Водночас, споживання пам'яті у додатку Spring Boot починається від 64 Мб і зростає зі зростанням функціоналу мікросервісу та може досягати 1Gb і навіть більше.

Golang себе зарекомендував як мова програмування, орієнтована на створення високопродуктивних додатків. Він поєднує в собі простоту мови та надає дуже прості, але ефективні інструменти для реалізації багатопотокових додатків. Слід зауважити, багато частин PaaS шару Kubernetes та засоби контейнерної віртуалізації Docker були написані цією мовою. Програми мовою golang компілюються в один файл, натомість, для кожної платформи є свій компілятор. Час початку простого мікросервісу з подібним функціоналом з Spring Boot додатком на Java обчислюється мілісекундами (5-10 ms), а обсяг пам'яті, що використовується - 2-5 Мб. Водночас, golang не має такої кількості якісних та перевірених часом бібліотек, як Java, тому часто виникає потреба реалізовувати необхідний шаблонний функціонал самотійно. Розглянуті аспекти збільшують час розробки цією мовою, тому «великі» мікросервіси з різноманітною бізнес-логікою та функціоналом писати такою мовою недоцільно через значні витрати на розробку і на супровід

такого мікросервісу. Більш детальне порівняння характеристик технологій розробки наведено в таблиці 2.1.

Таблиця 2.1 – Порівняння характеристик технологій розробки для хмари

Характеристика	Java + SpringBoot	Golang + mux
Час старту	1 секунда	5-10 мілісекунд
Використання пам'яті	від 64 Мб	від 2-5 Мб
Порог входження	середній	низький
Підтримка gRPC	є	є
Бібліотека роботи з Envoyпроху	є	є
Управління пам'яттю	збірка сміття	збірка сміття
Багатопоточна модель	важка (потоки, блокування)	полегшена (goroutine, канали, блокування)
Швидкість розробки	висока, за рахунок великої кількості готових бібліотек	умовно середня (з ростом додатку - зростає), дуже мало готових рішень

Обидві ці мови надають можливості створення HTTP-сервера без особливих труднощів. У go lang це надається стандартною бібліотекою мови, в java більша частина реалізації покладена на сторонні бібліотеки. Обидві мови підтримують роботу з протоколом gRPC, а також чудово підтримують роботу і навіть складання програми в контейнері.

2.3 Методи розгортання мікросервісів у хмарі

З розробкою програмного продукту необхідно розробити і схему процесу його автоматизованого розгортання у хмарі, оскільки конфігурація розгортання має бути опублікована в PaaS, щоб почався процес життєвого циклу програми.

Розгортання мікросервісів у хмарі та методи того, як це зробити заслуговують на окрему книгу, проте в рамках науково-дослідної роботи буде

проведено короткий огляд.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Рисунок 2.4 – Приклад Deployment конфігурації для Kubernetes

Однією з особливостей хмарних оточень є конфігураційний файл розгортання, який безпосередньо публікується в середовищі PaaS, що власне так і називається - Deployment. Наприклад візьмемо PaaS середу Kubernetes і розглянемо приклад конфігурації.

Дуже важливим пунктом конфігурації є властивість "image", в ньому вказано ім'я образу контейнера, який братиметься з налаштованого на хост репозиторію машини docker-образів. За промовчанням це Docker Hub.

Для більшої наочності зобразимо весь шлях контейнера з моменту його створення (складання) до моменту потрапляння на хост машини з Kubernetes.

Свій життєвий шлях докер образ бере з вихідного коду, а якщо бути точніше з Dockerfile, де описано з чого створюється докер образ і що в ньому має бути. Саме в цьому файлі описується те, що образ міститиме в собі, які файли, які інструменти та програми.

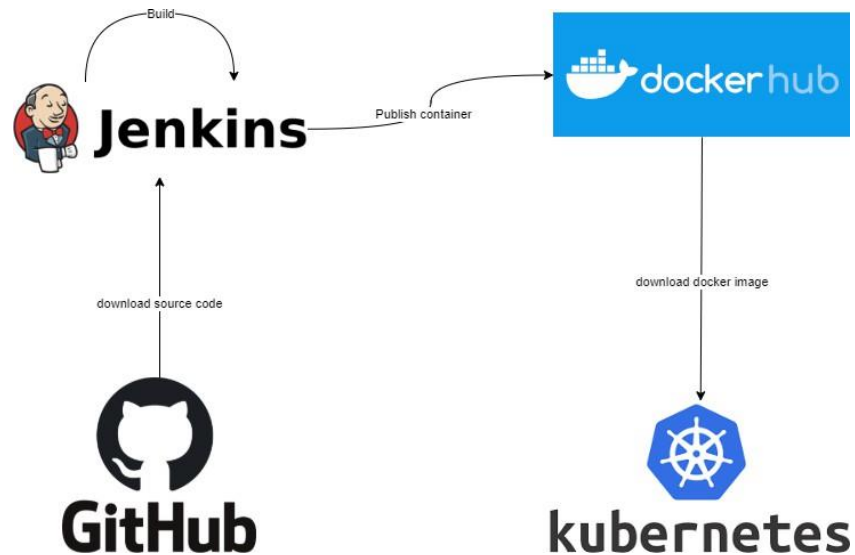


Рисунок 2.5 – Процес збірки і публікації Docker контейнера [30]

Після того, як докер образ збирається разом із скомпільованим мікросервісом, він публікується в репозиторії докер образів, після чого йому дається ім'я і проставляється тег, що формує унікальне ім'я образу. Після публікації цей образ може бути завантажений та запущений у Kubernetes. Алгоритм складання зображений на рис. 2.6.

Однак сам мікросервіс у хмарі з'явитися не може без конфігурації, а значить, вона повинна бути кимось опублікована в PaaS.

Є два способи, як це зробити автоматично:

1. Розгортання можна легко автоматизувати за допомогою звичайних скриптових програм. Так працює більшість деплоєрів, побудованих на основі Jenkins;
2. Написати програму-оператор, яка на основі одного

конфігураційного файлу розгорне кілька мікросервісів як модуль.

Незважаючи на наявність різних способів автоматизування розгортання логіка залишається однаковою і включає кілька простих кроків. Алгоритм логіки розгортання зображено на рис. 2.6.

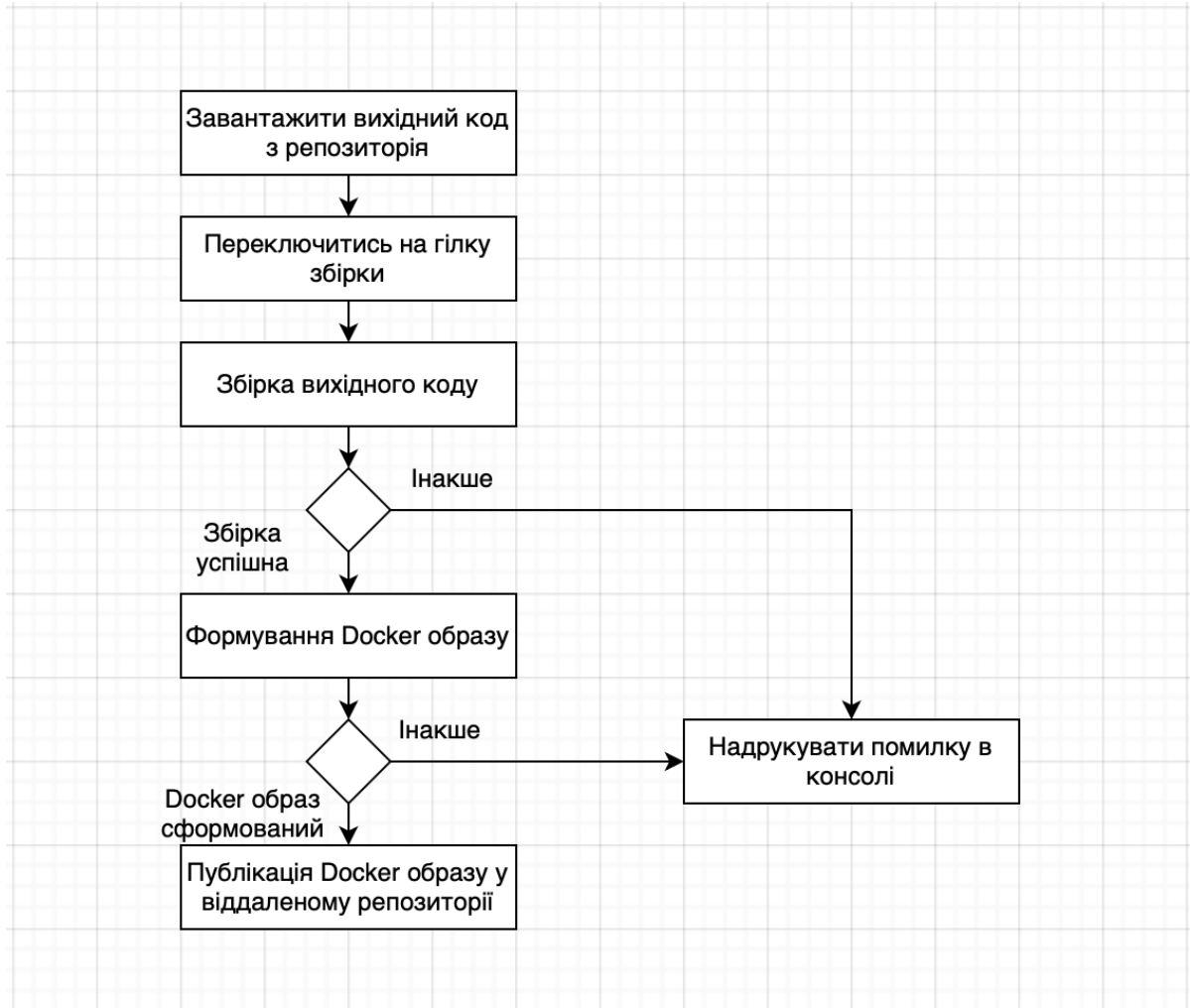


Рисунок 2.6 – Алгоритм збірки вихідного коду мікросервісу в docker-образ [31]

Схема розгортання з допомогою деплоєра (скрипта) зручна, коли деплоєр розробляється під розгортання конкретного набору мікросервісів і під конкретний проект.

Це пов'язано з тим, що набір мікросервісів визначається заздалегідь і якщо ми хочемо надати наш програмний продукт як набір мікросервісів, ми повинні і надати деплоєр користувачеві, щоб він міг ним

скористатися.

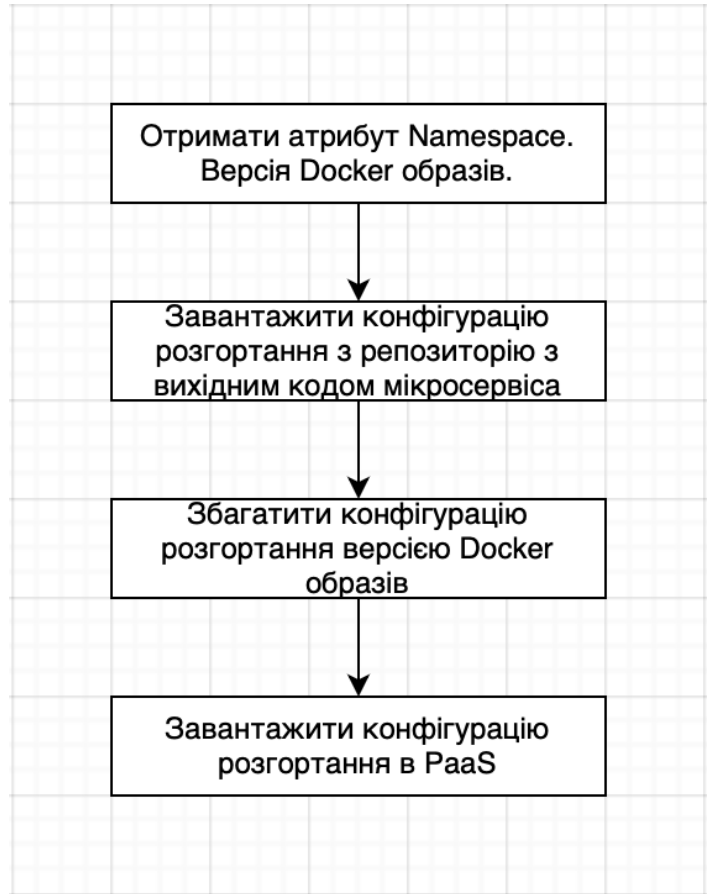


Рисунок 2.7 – Алгоритм розгортання мікросервіса [31]

Для розгортання програмних продуктів на основі архітектури мікросервісів рекомендується використання спеціалізованої програми-оператора, яка встановлюється вручну один раз і забезпечує подальше розгортання рішення. Такий підхід дозволяє реалізовувати гнучкі конфігурації розгортання з можливістю їхнього налаштування користувачем, що становить одну з стратегій надання програмного забезпечення. У спрощених сценаріях ця стратегія реалізується через надання користувачеві скрипта для автоматизованої інсталяції та конфігурації продукту.

РОЗДІЛ 3. МОДЕЛЬ ІНСТРУМЕНТУ УПРАВЛІННЯ API НА ОСНОВІ МІКРОСЕВІСНОЇ АРХІТЕКТУРИ

3.1. Концептуальна архітектура інструменту керування API

У якості API-шлюзу оберемо програмне забезпечення Envoy Proxy, яке поширюється з відкритим кодом і характеризується високою гнучкістю конфігурації. До його переваг можна віднести широке застосування у великих компаніях-розробниках програмного забезпечення, таких як IBM і Google. До того ж, Envoy Proxy реалізований мовою програмування C++, що забезпечує його високу продуктивність та мінімальне споживання пам'яті. Натомість цей підхід вимагає розробки окремого сервісу API Publisher для надання динамічної конфігурації маршрутизації через протокол gRPC, оскільки Envoy Proxy для динамічного керування потребує зовнішнього джерела конфігурації.

Розглянемо концептуальну схему роботи нашого програмного продукту. На рисунку 3.1 зображено хмарне середовище під керуванням PaaS Kubernetes.

Фігури, окрім акторів, позначають ресурси хмари. Круглим позначені сервіси. Сервіси Kubernetes призначені для приховування набору Pod під одним DNS ім'ям.

Наприклад, виходячи зі схеми, у нас може бути кілька екземплярів API-gateway запущених у хмарі, проте звернення до них можливе через DNS ім'я api-gateway.

Завдяки service запису, опублікованому в хмарі kubernetes зрозуміє, що користувачеві, який звернувся у внутрішній мережі по доменному імені, потрібен будь-який із запущених екземплярів api-gateway.

Для більш простого розуміння можна сказати, що service це DNS запис, що приховує кілька IP-адрес запущених екземплярів програми.

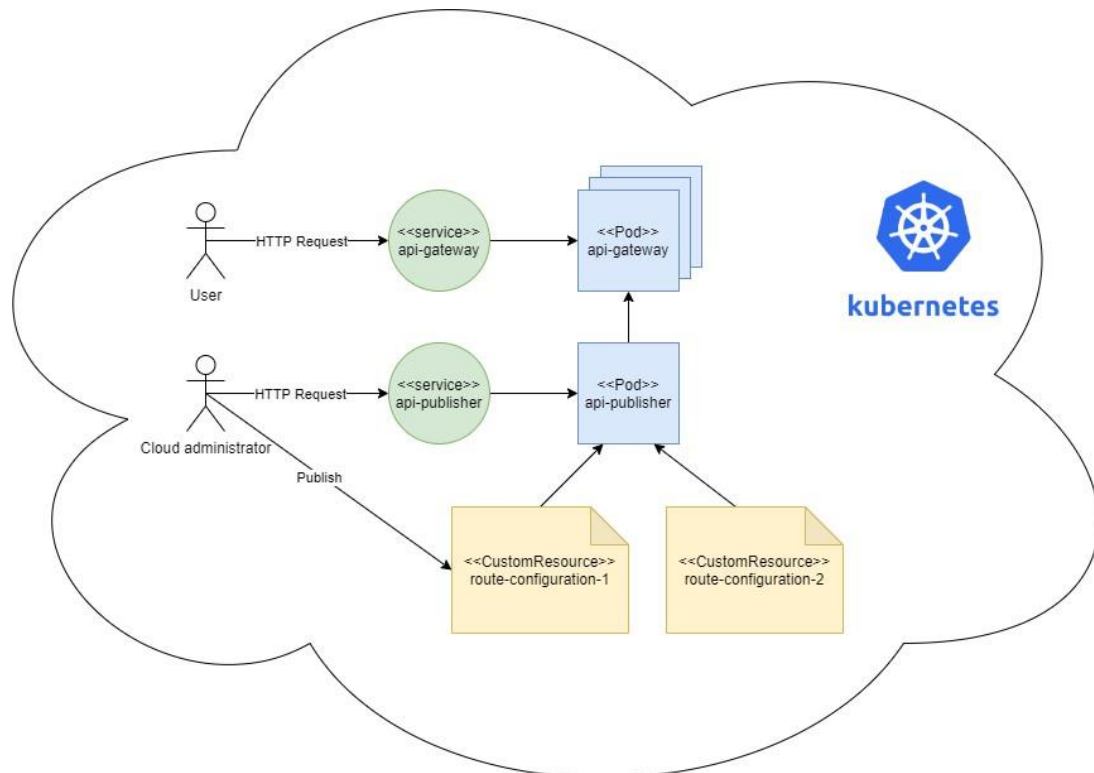


Рисунок 3.1 - Концептуальна архітектура [3]

Pod'и на рисунку уособлюють запущені Docker-контейнери. Життєвий цикл Pod визначається ресурсом Deployment, яке задля спрощення на схемі не відображено.

Фігурою документа (прямокутник з загнутим кутом) зображені самописні ресурси, опубліковані в хмарі, які створюються і описуються розробником. Для того, щоб такий самописний ресурс опублікувати, необхідно створити опис самописного ресурсу (Custom Resource Definition), який не був вказаний на схемі для простоти та наочності.

Під користувачем (User) мається на увазі або людина, або інший мікросервіс, який використовує опубліковане API. Це використання виражено у відправленні HTTP-запитів у gateway, який надалі, за наявності відповідного правила маршрутизації, перенаправить запит у потрібний мікросервіс.

Під адміністратором хмари розуміється користувач, що здійснює

безпосередньо публікацію API.

3.2 Розробка моделі інструменту управління API

Архітектура програмного продукту, що розробляється, складатиметься з трьох модулів: API-шлюз, API-публікатор, модуль розгортання. API-шлюз оброблятиме запити та перенаправлятиме їх відповідно до конфігурації, яку отримає від API-публікатора. А модуль розгортання забезпечить легке встановлення всього програмного продукту в хмару.

На початкових етапах проектування програми було створено діаграму варіантів використання. Діаграма варіантів використання визначає функціональне призначення моделі, що моделюється. Кожен варіант використання послідовності дій, які повинні бути виконані системою.

Варіанти використання є овалом з текстом сценарію. Участь актора у системі моделюється лінією між актором та сценарієм використання. Для зображення межі системи використовують рамку навколо варіанта використання.

Діаграми варіантів використання ідеально підходять для:

- зображення цілей взаємодії системи з користувачем;
- визначення та організація функціональних вимог до системи;
- вказівку контексту та вимог системи;
- моделювання основного потоку подій у сценарії використання.

Діаграма варіантів використання продукту управління API представлена на рисунку 3.2.

Система містить 4 актори. В якості користувача може виступати, як зовнішній користувач хмарного програмного продукту, так і внутрішній мікросервіс. Використання це полягає у виклику конкретного API, який налаштований в API шлюзі. Завдання адміністратора хмари опублікувати

конфігурацію для API-шлюзу у хмарі. Формат конфігурації буде описано далі. Далі архітектор хмарного програмного рішення може написати файл конфігурації API вручну і надати його адміністратору хмари на публікацію. Розробнику надається бібліотека, використовуючи яку він може анотувати свій вихідний код метаданими, якими можна буде згенерувати конфігурацію API, готову для публікації в хмарі.

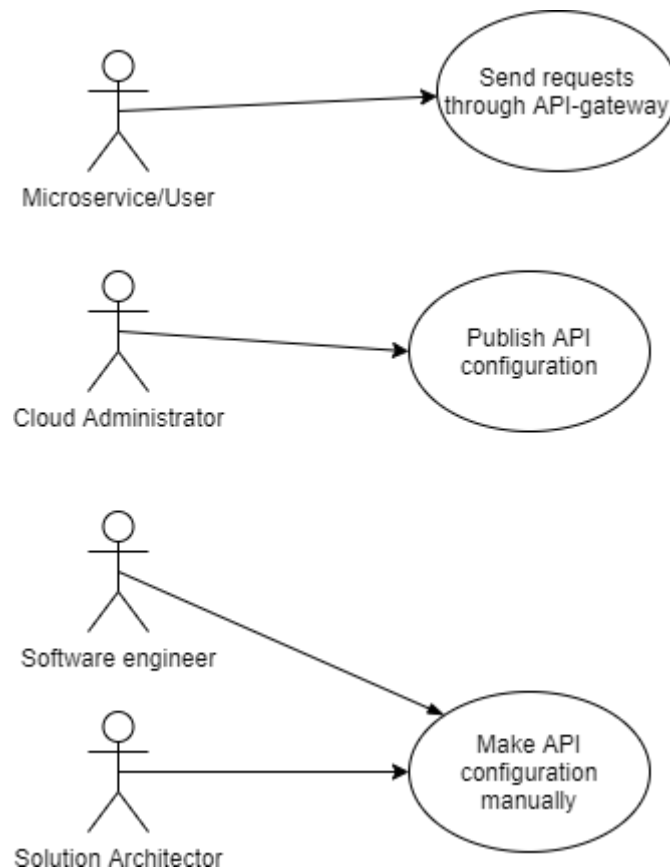


Рисунок 3.2 – Діаграма варіантів використання

Джерело: [Розроблено автором]

Далі було спроектовано діаграму сутнісних класів у тому, щоб виділити основні сутності до роботи системи.

«Клас-сутність» - об'єкти сутнісних класів є блоки тривало збереженої інформації, що використовуються для організації баз даних та знань, файлових систем зберігання, даних різної логічної структури в основному в цих класах розвинений атрибутний розділ, проте є невелика кількість

операцій контролю обмежень цілісності як стандартних, так і специфічних для даної предметної області.

Діаграми класів пропонують низку переваг, що дозволяє використовувати для наступних завдань:

- зображення моделі даних для інформаційних систем, незалежно від того, наскільки вони прості чи складні;
- кращого розуміння програмної архітектури програми;
- візуально вирази будь-яких специфічних потреб системи;
- створення докладних діаграм, для виділення класів, які необхідно реалізувати в описаній структурі;
- надати незалежний від реалізації опис типів, що використовуються в системі, які згодом передаються між її компонентами.

Діаграма класів в першу чергу призначена для розробників, щоб забезпечити концептуальну модель та архітектуру системи, що розробляється. Як правило, діаграма класів складається з більш як одного класу або всіх створених класів для системи.

Для конфігурації API у програмному продукті виділено такі сутності: «APIConfig», "APIConfigSpec", "Route", "Destination", "Rule", "Match", "MatchHeader".

Наявність на діаграмі сутностей (див.рис.3.3) «APIConfig» та «APIConfigSpec» на перший погляд є надмірним, проте специфікація платформи Kubernetes з розширення зобов'язує нас дотримуватися саме такого способу поділу. Сутність "APIConfig" в даному випадку уособлює сутність з набором метаданих конфігурації, включаючи ім'я, тип, а також наявність ключа група + версія конфігурації для здійснення вбудованих можливостей щодо валідації конфігурації.

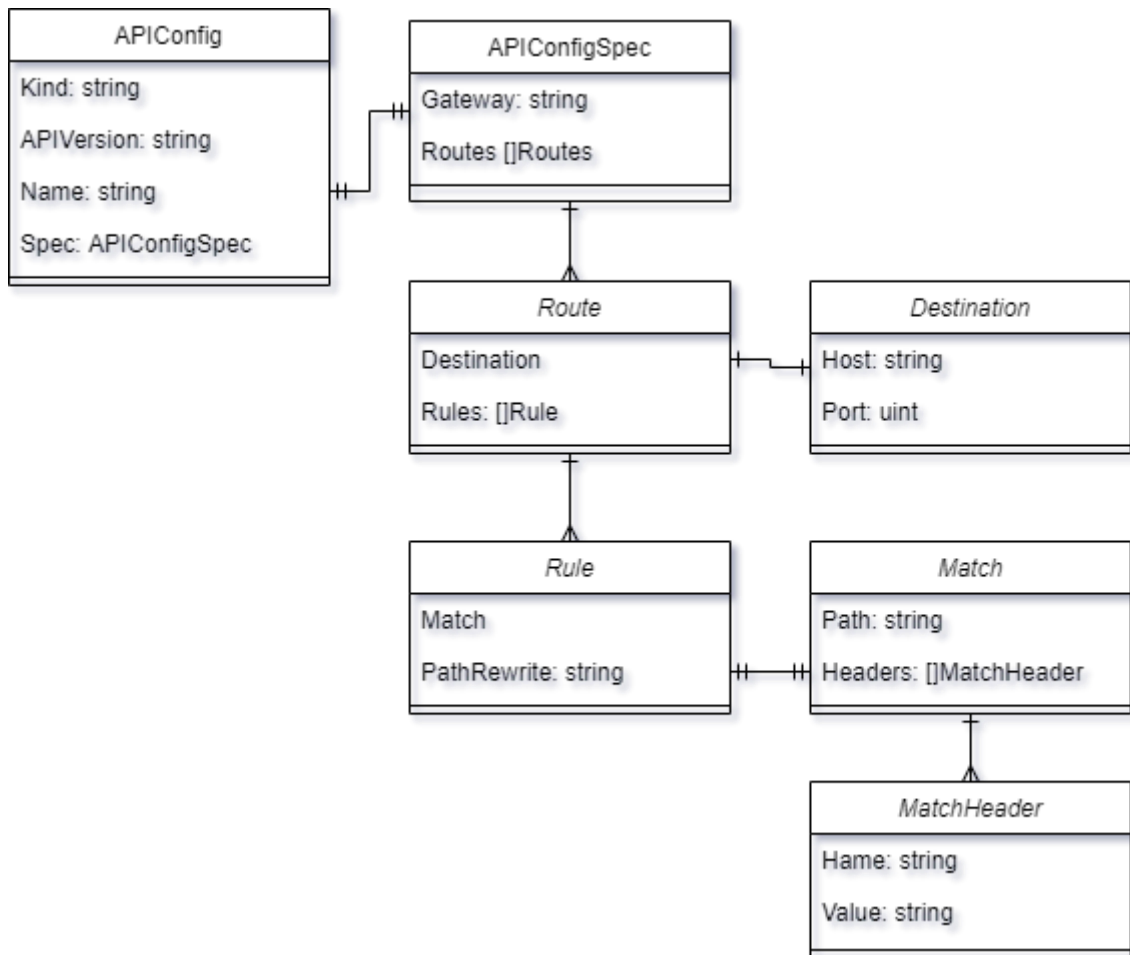


Рисунок 3.3 - Діаграма сутностей API конфігурації

Джерело: [Розроблено автором]

Сутність «ApiConfigSpec» є кореневою сутністю, в якій міститься список правил маршрутизації, а також вказано ім'я шлюзу, до якого буде застосована ця конфігурація. На основі конфігурації API опублікованої у хмарі у форматі YAML буде сформовано сутність "ApiConfigSpec" включаючи всі інші сутності, розташовані за ієрархією нижче.

Сутність Route агрегує набір правил визначення HTTP запитів, пов'язуючи їх з конкретною точкою призначення. Іншими словами, визначають маршрут для запитів HTTP.

Сутність Rule описує правило, за яким будуть фільтруватися зовнішні запити для подальшої маршрутизації. Сутність «Match» у разі визначає критерій, яким вибиратиметься HTTP запит, а поле "PathRewrite" визначає

модифікацію шляху HTTP запиту. На кінцеву точку Destination запит прийде вже з іншим Path якщо поле "PathRewrite" має конкретне значення.

Сутність MatchHeader містить опис заголовка в HTTP запиту, на який повинна проводитися подальша маршрутизація мережного трафіку. Сутність включає ім'я заголовка та його значення. На рисунках 3.4 та 3.5 зображені спрощені діаграми, що описують внутрішню структуру конфігурації EnvoyProxy.

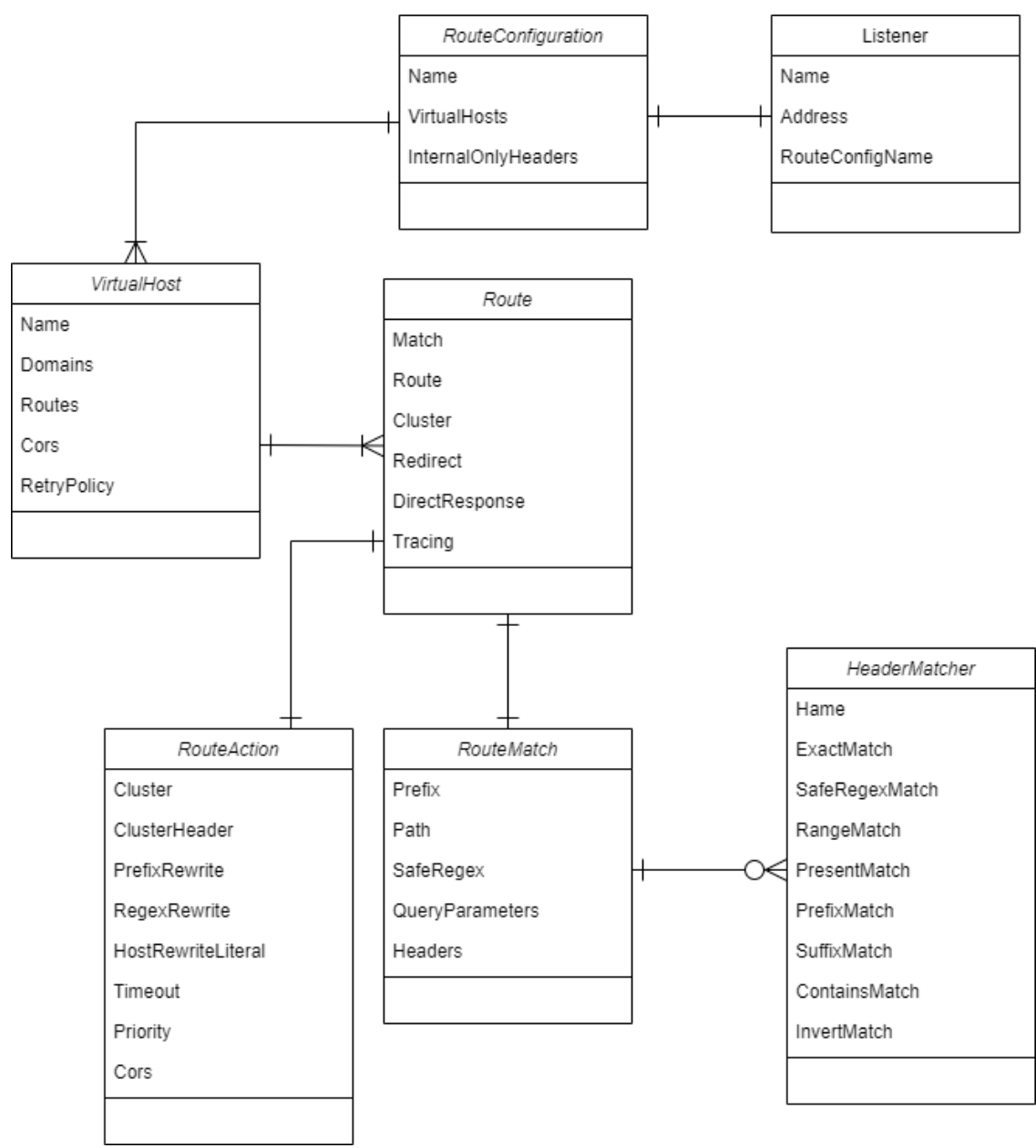


Рисунок 3.4 –Діаграма сутностей EnvoyProху – конфігурація маршрутів

Джерело: [Розроблено автором]

Ця структура складна і широка, але в нашому випадку не потрібно використовувати всі великі можливості, які поставляє компонент. Відсутня частина значенні, які неможливо буде дістати зі структури на рис. 16 буде заповнюватися підібраними значеннями за замовчуванням.

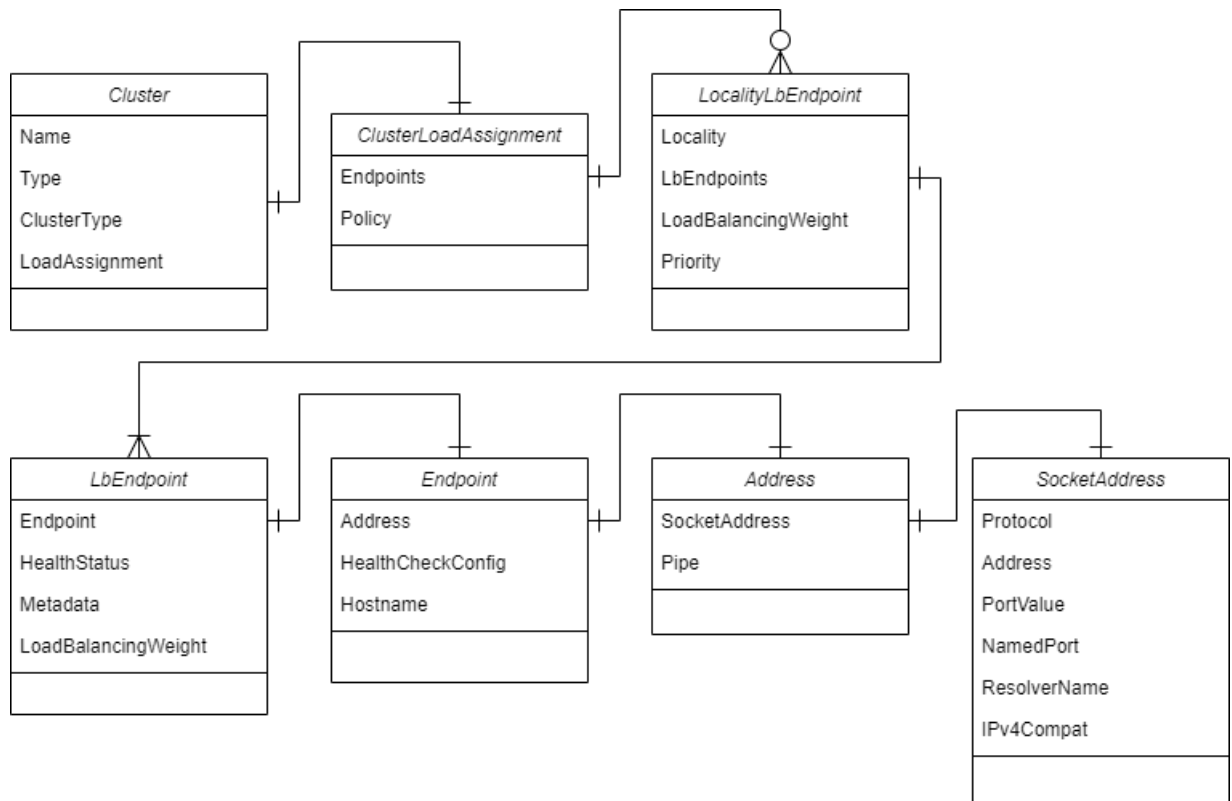


Рисунок 3.5 – Діаграма сутностей конфігурації EnvoyProху - конфігурація кластерів

Джерело: [Розроблено автором]

Структура інструмента буде розбита на компоненти із чітко розділеними обов'язками. На рисунку 3.6 зображена діаграма компонентів та їх способи взаємодії між собою.

Компонент «Operator» відповідає за розгортання API-publisher та API-gateway у хмарному середовищі. Якщо розглядати Operator більш детально, то це компонент, завдання якого спростити налаштування двох інших компонентів. інструмента будет разбита на компоненты с четко

разделенными обязанностями.

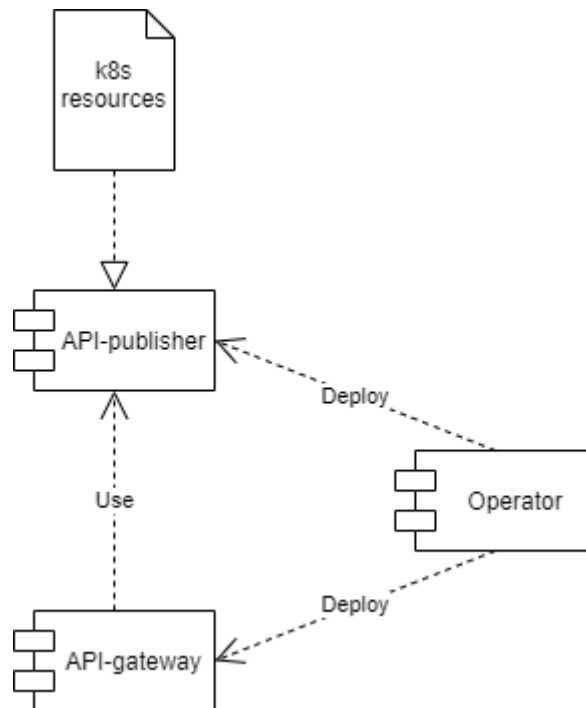


Рисунок 3.6 - Діаграма компонентів

Джерело: [Розроблено автором]

Можна було б сказати, що він керує життєвим циклом публікатора та шлюзу, проте це не зовсім правильно. Так, оператор здатний видаляти, оновлювати та створювати публікатори та шлюзи, проте це не життєвий цикл контейнера. Було б правильно сформулювати цю можливість як керування життєвим циклом конфігурації розгортання компонентів API-publisher та API-gateway.

Розглянемо детальніше алгоритм управління прикладним програмним інтерфейсом.

Його можна поділити на дві частини. В одній буде зображено формування конфігурації для API-gateway (див. рис. 3,7) в іншій, процес отримання конфігурації компонентом API-gateway з API-publisher (див. рис. 3,8).

Як можна бачити з рисунка 3.7, алгоритм ґрунтується на

поширеному шаблоні проектування Observer. Компонент API-publisher підключається до шару Kubernetes через Kubernetes API з метою відстеження подій, що відбуваються з ресурсом APIConfig.

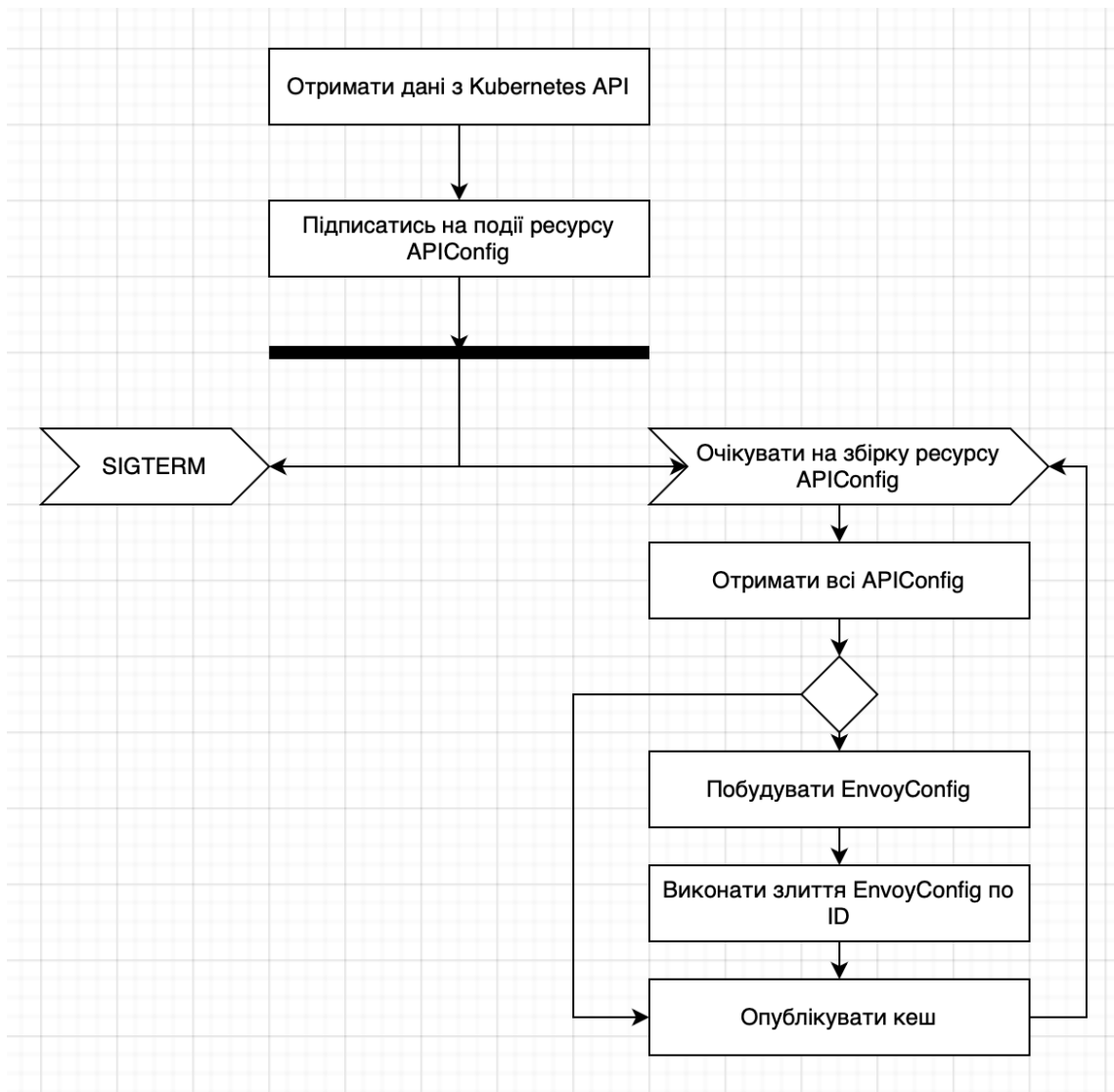


Рисунок 3.7 – Алгоритм формування API-gateway конфігурації

Джерело: [Розроблено автором]

Після успішної підписки потік виконання поділяється на дві частини, одна очікує завершення роботи програми, а інша очікує подій (Events) від Kubernetes API. Коли подія від Kubernetes API надходить в API-publisher відбувається отримання всіх ресурсів типу APIConfig у тому самому просторі імен (namespace) де розташований API-publisher. Далі, для кожного знайденого ресурсу формується повний набір

сутностей, необхідний роботи API-gateway. Далі сформовані конфігурації поєднуються в унікальний набір (сутності з одним ім'ям поєднуються), після чого опубліковуються в кеш пам'яті API-publisher. Варто зазначити, що кеш конфігурацій для API-gateway надається бібліотекою go-control-plane з коробки.

На рисунку 3.8 наведено алгоритм отримання конфігурації компонентом API-gateway з API-publisher.

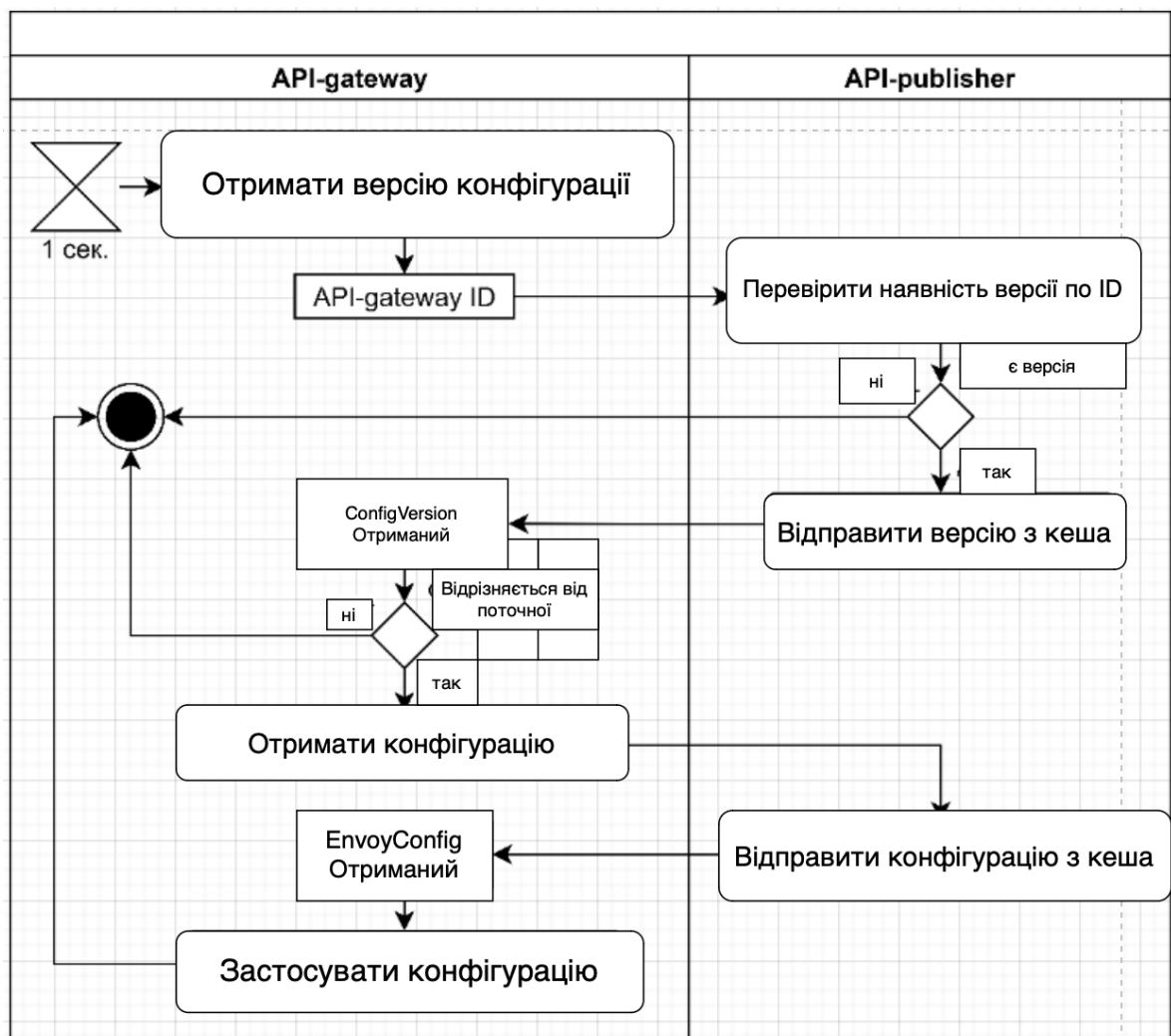


Рисунок 3.8 – Алгоритм отримання конфігурації з API-publisher

Джерело: [Розроблено автором]

У цьому зображенні алгоритму опущені деталі серіалізації/десеріалізації та особливості взаємодії компонентів між

собою за протоколом gRPC. API-gateway налаштований так, що кожен проміжок часу (1 сек.) він надсилає запит до API-publisher на перевірку версії конфігурації, опублікованій у ньому. У запиті надсилається ідентифікатор API-gateway. Після отримання версії, якщо така є, API-gateway порівнює її з відомою йому, і якщо версії різняться, починається процес отримання конфігурації.

```
FROM flant/shell-operator:v1.0.0-rc.1
ADD hooks /hooks
ADD deploy /deploy
RUN chmod -R 777 /hooks
RUN chmod -R 777 /var/run/

USER 10001
```

Рисунок 3.9 – Dockerfile компонента operator

Джерело: [Розроблено автором]

Як зазначалося вище, компонент заснований на shell-operator і за інструкцією з документації створені директорії /hooks і команда в рядку 2 копіює скрипти в цей каталог і далі лунають права, щоб були права запускати ці скрипти.

РОЗДІЛ 4

АПРОБАЦІЯ МОДЕЛІ ІНСТРУМЕНТУ КЕРУВАННЯ ПРИКЛАДНИМ ПРОГРАМНИМ ІНТЕРФЕЙСОМ

4.1 Процес впровадження інструменту керування прикладним програмним інтерфейсом

Процес впровадження інструменту управління прикладним програмним інтерфейсом є попереднім настроюванням оточення, для розгортання компонента «operator» і безпосередньо самого розгортання компонента «operator». Далі сам компонент «operator» управляє подальшим процесом розгортання інших складових частин інструменту управління прикладним програмним інтерфейсом, саме компонентів «api-gateway» і «api-publisher». Розгортання інструмента показано рисунку.

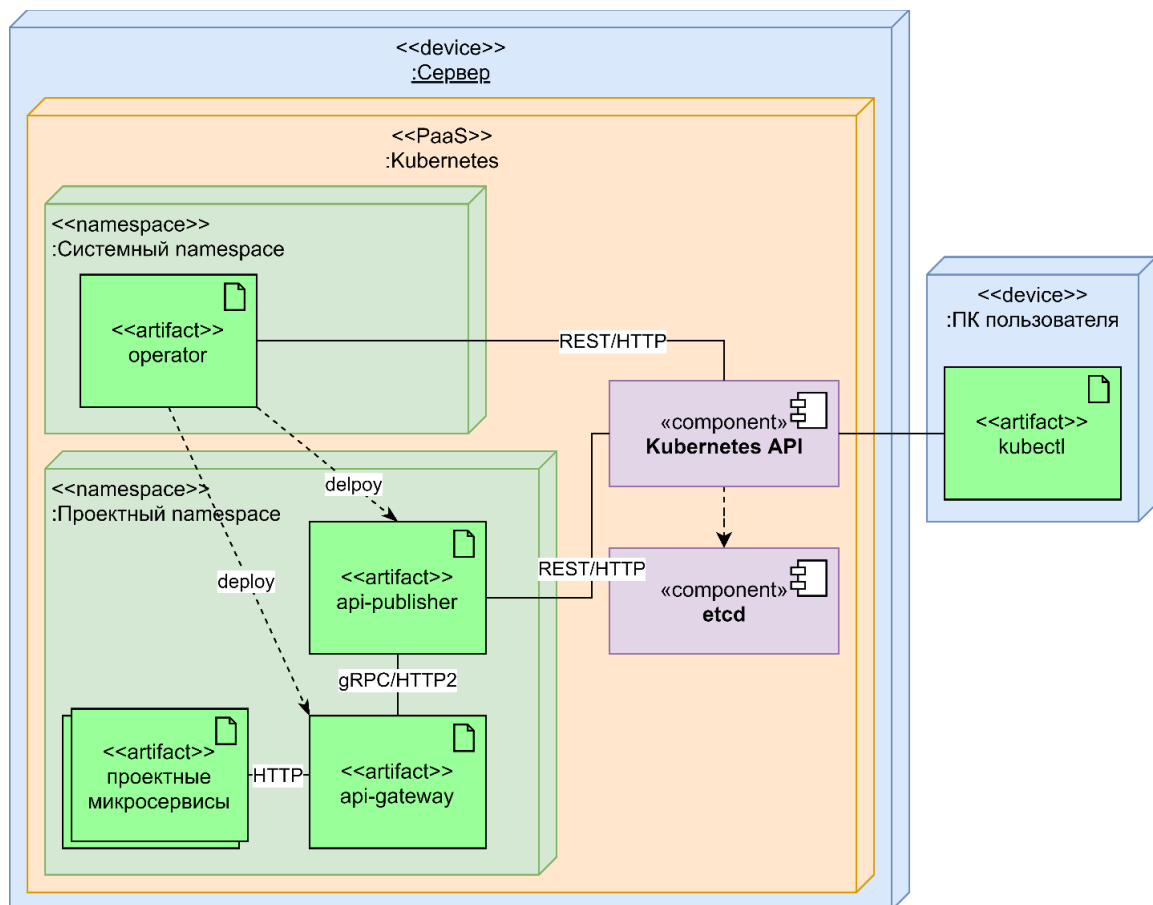


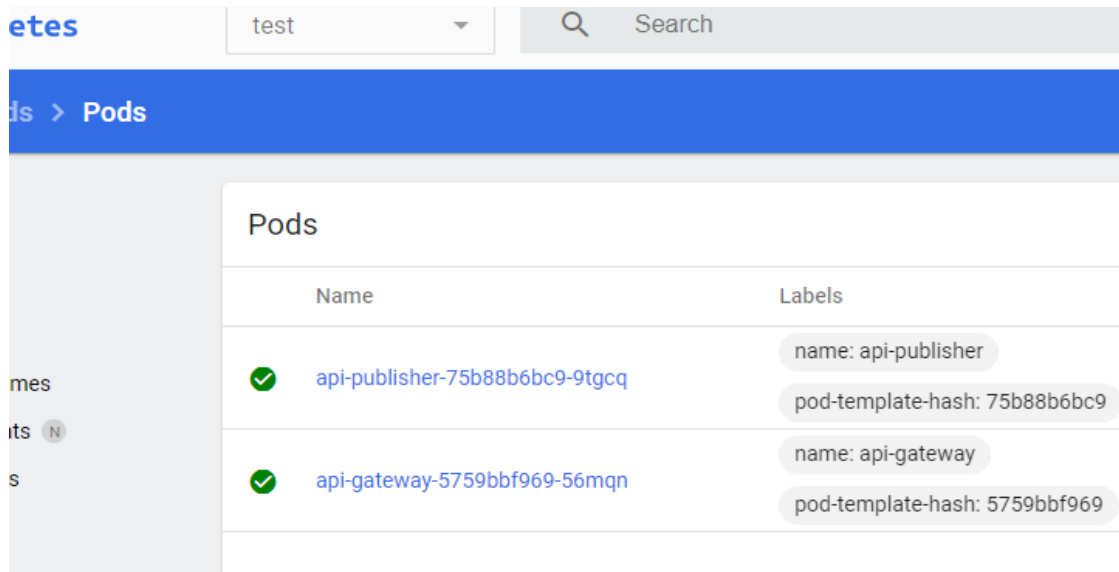
Рисунок 4.1 – Діаграма розгортання [12]

Перш ніж описувати діаграму розгортання сказати, що діаграма розгортання - це один з типів діаграм UML стандарту, створений з метою відобразити відношення між компонентами програмного продукту, що виконуються, і фізичної обчислювальної системи, відобразити відношення компонентів в оточенні. Діаграма розгортання показує архітектуру системи виконання, включаючи такі вузли як апаратні або програмні середовища виконання, а також проміжне програмне забезпечення як компонент, що їх з'єднує.

Діаграми розгортання використовуються для візуалізації програмного забезпечення та безпосередньо фізичного обладнання. Використовуючи цей тип діаграм стандарту UML, можна зрозуміти, як програмний продукт буде фізично розгорнутий на обладнанні. Також діаграми розгортання підходять для моделювання топології обладнання системи, якщо їх порівнювати з іншими типами UML діаграм, які описують в основному логічні компоненти системи.

Розроблений інструмент має вузли сервер та клієнт. Клієнт, це людина, яка використовує клієнтську утиліту `kubectl` здійснює публікацію ресурсів безпосередньо в `Kubernetes`. На сервері розгорнутий програмний продукт `Kubernetes` зі своїми компонентами `Kubernetes API` та `etcd`. Тут `etcd` це надійне сховище даних типу «ключ-значення», а `Kubernetes API` це прикладний програмний інтерфейс, заснований на протоколі `HTTP` і надає можливості розширення платформи самописними ресурсами та їх обробниками. Також усередині `Kubernetes` існує поділ на простори імен, що дозволяє організовувати розгортання різних проектів усередині одного `Kubernetes` та уникнути конфліктів імен. Артефакт `operator` розгорнуть в окремому `namespace`, щоб мати можливість розгортати решту компонентів у будь-якому іншому. За фактом кожен артефакт уособлює `Pod` – ресурс `Kubernetes`, що розширює докер-контейнери і є запущеним екземпляром програми.

Кожен запущений сервіс представлений на вкладці Pods. Скріншот вкладки представлений на рисунку 4.2.



Pods	
Name	Labels
✓ api-publisher-75b88b6bc9-9tgcq	name: api-publisher pod-template-hash: 75b88b6bc9
✓ api-gateway-5759bbf969-56mqn	name: api-gateway pod-template-hash: 5759bbf969

Рисунок 4.2. – Список сервісів системи

Джерело: [Розроблено автором]

Після стадії успішного розгортання програмного продукту необхідно провести повноцінне функціональне тестування, щоб у його працездатності. Функціональне тестування використовується для доказу того, що функції програмного забезпечення мають очікуваний результат, необхідний кінцевому користувачеві або бізнесу. Функціональне тестування, як правило, включає оцінку кожної функції та порівняння з вимогами бізнесу. ПЗ тестується шляхом надання йому деяких вхідних даних, щоб оцінити, як воно відповідає, пов'язане або змінюється порівняно з його базовими вимогами. Крім того, функціональне тестування допомагає розкрити проблеми використання програмного забезпечення та оцінити його зручність застосування. Методи функціонального тестування включають інтеграційне тестування, тестування білої скриньки, тестування чорної скриньки, модульне тестування (воно ж unit тестування), приймальне тестування тощо. Інструмент тестувався за допомогою модульного, інтеграційного та системного тестування. Послідовність тестування має такі етапи:

Системне – Ітераційне – Модульне

Модульне тестування – це рівень тестування програмного забезпечення, на якому проводиться тестування окремих модулів або компонентів програмного забезпечення. Основна мета полягає в тому, щоб перевірити, що кожна мінімальна одиниця програмного забезпечення працює так, як задумано. Під мінімальною одиницею ПО розуміється найменша частина будь-якого програмного забезпечення, що тестується. Оскільки ми використовуємо мову програмування `golang`, то нашому разі мінімальною частиною є функція. В рамках модульного тестування тестами було покрито більшість написаних функцій. Завдяки цим тестам ми зможемо виключити помилки компіляції, внесені після виправлення або розробки нового функціоналу.

На наступному етапі ми опрацювали інтеграційні тести. На цьому рівні вже тестуються взаємодія різних наборів компонентів програмного забезпечення між собою в обчислювальному оточенні, наближеному до реальних умов. Метою інтеграційного тестування є виявлення помилок у взаємодії набору компонентів між собою. У нашому випадку точкою входу для тестування була публікація ресурсів через `Kubernetes API`, а також перевірка правильності маршруту, шляхом відправки `HTTP` запиту через `api-gateway`. Цей етап тестування дозволив виявити помилки при формуванні маршрутів в `api-gateway` і виправити їх шляхом їх сортування за критерієм довжини. Найбільш "широкі" маршрути, наприклад `/"` (кореневий) розташовуються нижче, тоді як більш конкретні, наприклад, `"/api/v1/service/endpoint1/resourceid/operation"` розташовуються вище. Як основу інтеграційних тестів було обрано набір інструментів та бібліотек `Arquillian`.

`Arquillian` – це платформа для інтеграційного тестування. Головною її перевагою є легкість і швидкість написання тестів, як це були модульні тести.

Наступним етапом було проведено системне тестування, у процесі якого вже було перевірено міжсервісну взаємодію. Воно включало:

- тестування інструменту з розгорнутим рядом проектом, публікація ресурсів та конфігурацій, з метою перевірити як компоненти взаємодіють один з одним і з середовищем виконання;
- перевірка, що налаштований API-gateway пропускає мережевий трафік відповідно до очікуваного.

В результаті системного тестування виявлено помилки при обробці конфігурації. Наявність кількох змін у одному просторі імен виробляло до обробки лише однієї, тоді як очікувалося їх злиття. Також було враховано можливу наявність двох однакових конфігурацій, але з різними іменами, їх наявність тепер призводить до коректного злиття та усунення дублікатів.

Також було вирішено зробити тестування навантаження. Таке тестування перевіряє стабільність, масштабованість, швидкість та надійність програмного забезпечення. Навантажувальні тести можуть орієнтуватися різні показники. У разі показниками є час оновлення зміни від її публікації, до актуального стану компонента api-gateway, і навіть кількість конфігурацій, оброблюваних за прийнятний час. В результаті тестування було виявлено велике споживання пам'яті компонентом api-publisher, незважаючи на те, що кількість конфігурацій за тестовим сценарієм збільшувалася і зменшувалася, пам'ять у цьому компоненті тільки зростала.

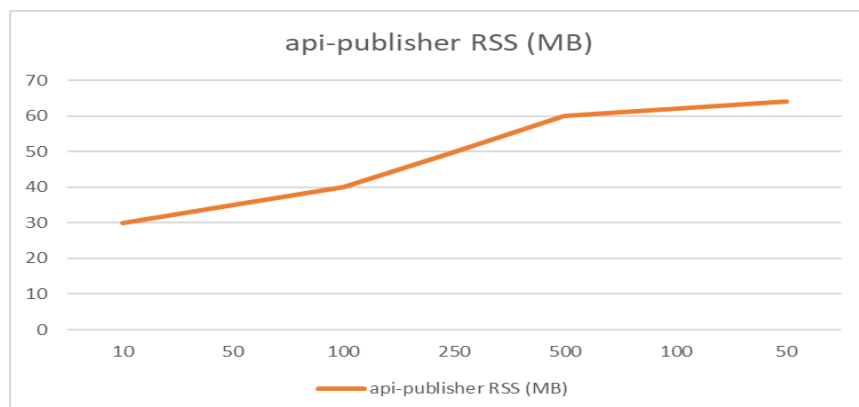


Рисунок 4.3 – Діаграма споживання пам'яті компонентом api-publisher на тестуванні навантаження

Джерело: [розроблено автором]

Зображено графік залежності споживання пам'яті компонентом api-publisher стосовно кількості конфігурацій фіксованого розміру ~10Kb. Причиною цього часто є витікання пам'яті.

Однак після ретельного аналізу та інструментів профілювання ми дійшли висновку, що проблема у специфіці роботи збирача сміття у додатках написаних на go lang. Вирішенням проблеми було виставлення змінної оточення GOGC для контейнера з сервісом api-publisher у більш низьке значення, що змушувало збирача сміття працювати більш агресивно.

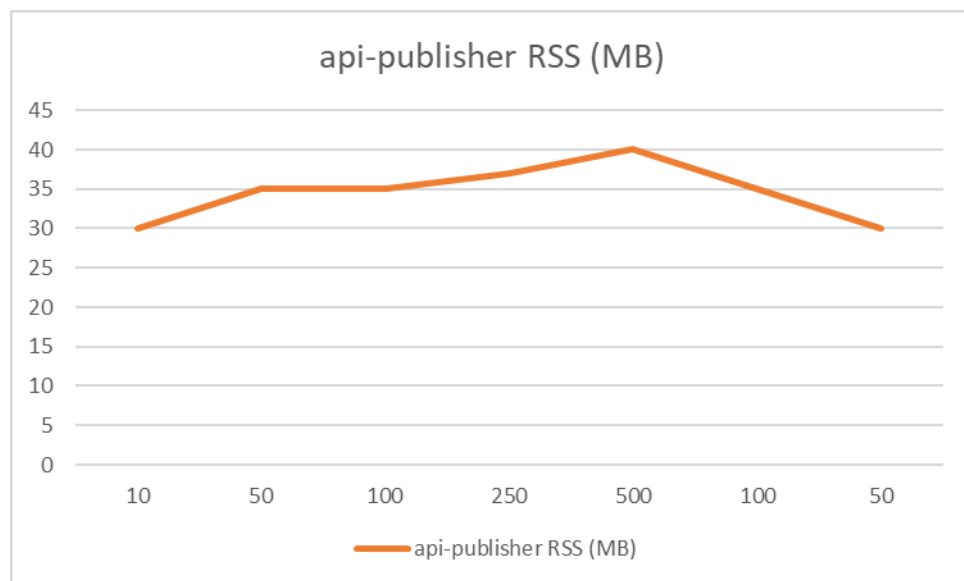


Рисунок 4.4 – Споживання пам'яті компонентом api-publisher після оптимізації

Джерело: [Розроблено автором]

Після тонкого налаштування збирача сміття у додатку api-publisher графік споживання пам'яті змінився і став виглядати більш об'єктивно та логічно.

4.2 Процес інтеграції інструменту прикладного програмного інтерфейсу з готовим проектним рішенням

У рамках процесу інтеграції передбачається, що інструмент управління прикладного програмного інтерфейсу вже розгорнутий у PaaS інфраструктурі поряд з інтегрованим додатком. Далі залежно від того, яке API у інтегрованих мікросервісів необхідно скласти конфігурацію.

Також, як частина інтеграції, доведеться в мікросервісах використовувати комунікацію до інших мікро-сервісів через шлюз, а це передбачає доопрацювання мікросервісів.

Однак якщо інструмент буде використовуватися для зовнішніх запитів, то максимум, що потрібно буде додати, так це Ingress ресурс, щоб «відкрити» шлюз для запитів поза хмарою.

```
apiVersion: apimanagement.cloud/v1alpha1
kind: APIConfig
metadata:
  name: example-api-config
spec:
  gateway: gateway
  routes:
  - destination:
      address:
        host: service
        port: 8080
    rules:
    - match:
        path: /api/v1/service
        headers:
        - name: X-Header
          value: x-value
      pathRewrite: /
```

Рисунок 4.5 – Приклад конфігурації API

Джерело: [Розроблено автором]

Підрахуємо потенційні трудовитрати на інтеграцію інструменту з існуючим проектним рішенням на мікросервісній архітектурі.

Таблиця 4.1 – Трудовитрати на інтеграцію

	Розробник	Старший-розробник
Розгортання	20 хвилин	10 хвилин
Написання конфігурації	~4 години	~2 години
Публікація конфігурації	1 хвилина	1 хвилина

Джерело: [Розроблено автором]

На основі даних таблиці 2 можна зробити висновок, що максимальний час, необхідний інтеграцію дорівнює 5 годин, що є дуже коротким періодом часу між рішенням інтегруватися і виходом у використання.

4.3 Розрахунок надійності розробленого інструменту

Критерієм якості прикладного програмного забезпечення, що розробляється, є ступінь її надійності.

Існує безліч прикладів, коли розроблена система падала, що призводило до величезних фінансових збитків компанії.

Щоб уникнути такого результату, необхідно оцінити надійність розробленої системи.

Існуючі моделі оцінки надійності програмного забезпечення об'єднані в три групи:

- статичні моделі (не враховують час появи збою);

- дискретні динамічні моделі (помилки усуваються);
- безперервні динамічні моделі (під час тестування помилки не виправляють).

Виберемо дискретні динамічні моделі, оскільки вони дозволяють у процесі тестування усунути помилки.

Як таку модель виберемо модель Шумана.

Вона будується на тому, що оцінка надійності програмного забезпечення проводиться на підставі проведеного тестування, що зручно, оскільки тестування ми вже провели.

Тестування проводиться поетапно і після завершення етапу всі виявлені проблеми виправляються і починається наступний етап тестування.

Було проведено 4 етапи тестування. Перший етап тривав 7 годин, потім другий – 5 годин, третій – 8 годин та останній – 3 години. На цих етапах виявлено 3, 2, 2, 0 помилок. Отримані результати наведено у таблиці 4.2.

Таблиця 4.2 – Результати тестування

Етап (k)	Тривалість (t), ч	Кількість збоїв (m)
1	7	3
2	5	2
3	8	2
4	3	0

Джерело: [Розроблено автором]

Загальний час тестування буде дорівнювати $T = 23$ години.

Загальна кількість виявлених та виправлених помилок $n = 7$.

Кількість помилок, виправлених на початок $(i + 1)$ етапу дорівнює $n_0 = 0; n_1 = 0; n_2 = 3; n_3 = 5; n_4 = 7$.

Функція надійності за моделлю Шумана описується наступною формулою:

$$R_i(t) = e^{-\lambda_i t}, \quad (4.1)$$

де λ –інтенсивність появи помилок, яка обчислюється за формулою:

$$\lambda = (N - n) * C, \quad (4.2)$$

де N –первісна кількість помилок;

C – коефіцієнт пропорційності

Щоб знайти початкову кількість помилок N звернемося до наступної формули:

$$\sum_{i=1}^k m_i \frac{\sum_{i=1}^k t_i}{\sum_{i=1}^k \frac{m_i}{N - n_{i-1}}} = \sum_{i=1}^k (N - n_{i-1}) t_i \quad (4.3)$$

$$C = \frac{\sum_{i=1}^k \frac{m_i}{N - n_{i-1}}}{\sum_{i=1}^k t_i} \quad (4.4)$$

З цього рівняння шляхом підбору знайдемо $N = 8$.

Тепер знаючи значення N знайдемо коефіцієнт C за формулою 4.4, і він дорівнює 0,05.

Розрахувавши коефіцієнт C отримуємо, що інтенсивність появи помилок дорівнює:

$$\lambda = (8 - (3 + 2 + 2)) * 0.05 = 0.05$$

Виведемо функцію надійності

$$R(t) = e^{-0.05t}$$

За отриманими результатами ПЗ є надійним і може експлуатуватися.

Оскільки розроблений інструмент використовує платформу Kubernetes, яка за замовчуванням надає механізми скалірування, самовідновлення (автохілінгу) та інші функції, це сприяє підвищенню загальної ефективності та стабільності програмного продукту. Завдяки вбудованим можливостям Kubernetes можливе автоматичне масштабування ресурсів відповідно до навантаження, а також самостійне відновлення компонентів системи при виникненні збоїв, що значно покращує продуктивність і надійність системи.

ВИСНОВОК

У ході проведення дослідження та написання випускної кваліфікаційної роботи було проведено аналіз існуючих підходів до управління прикладним програмним інтерфейсом мікросервісів, а також інструментів, що дозволяють ці підходи зробити ефективними та простими.

Для досягнення результату було виконано поставлені завдання:

- проаналізовано підходи до управління прикладним програмним інтерфейсом у хмарному середовищі;
- проаналізовано існуючі інструменти керування прикладним програмним інтерфейсом мікросервісів;
- розроблено модель інструмента з ефективним керуванням прикладним програмним інтерфейсом;
- підтверджено ефективність запропонованої моделі практично.

Описаний у роботі інструмент керування прикладним програмним інтерфейсом мікросервісів дозволить за допомогою простих конфігурацій організувати публікацію прикладного програмного інтерфейсу, а також забезпечить надійну маршрутизацію трафіку всередині хмарного середовища.

Розроблений інструмент побудований з урахуванням впровадження в мікросервісну архітектуру з використанням сучасних платформ хмарних технологій, таких як Kubernetes, який містить зручні вбудовані інструменти балансування навантаження, масштабування, а також надійного сховища даних, що дозволяє досягти великого ступеня надійності інструменту в цілому.

Доробка інструменту силами проекту дозволить легко розширити можливості управління API, а також додати новий функціонал, такий як авторизація, консистентне балансування, захищені з'єднання TLS.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Денисенко І. С. Хмарні технології та віртуалізація в інформаційних системах підприємств. Київ: КНЕУ, 2019. 214 с.
2. Кислюк А. Контейнеризація та мікросервіси у сучасних інформаційних системах. Львів: Новий Світ, 2020. 198 с.
3. Лапченко І. В. Використання контейнеризації у розробці та розгортанні мікросервісів Актуальні питання та перспективи проведення наукових досліджень: матеріали IX Міжнародної студентської наукової конференції, м.Рівне, 28 листопада, 2025 рік / ГО «Молодіжна наукова ліга». — Вінниця: ТОВ «УКРЛОГОСГруп», 2025. — 998 с.
4. Лапченко І. В. Високодоступна архітектура мікросервісів ІТ-простір сьогодення: тенденції, інновації та перспективи розвитку: збірник тез доповідей II Міжнародної науково-практичної студентської конференції (15 жовтня 2025 року, м. Харків, Україна) [Електронний ресурс]. – Харків : ХНУ імені В. Н. Каразіна, 2025.
5. Матвеев В. Віртуалізація і хмарні сервіси: теорія та практика. Харків: Фоліо, 2021. 276 с.
6. Armstrong M. Virtualization Technologies in Cloud Computing. New York: Springer, 2020. 284 p.
7. Bucchiarone A. Microservices: Science and Engineering. Springer, 2020.
8. Burns D., Oppenheimer D. Kubernetes: Up and Running. 3rd ed. O'Reilly Media, 2022. 368 p.
9. Burns D., Oppenheimer D. Kubernetes: Up and Running. O'Reilly Media, 2022. 368 p.
10. Carnell J. Spring Microservices in Action. Manning Publications, 2017.
11. Coats R. Resource Pooling in Distributed Systems. Cambridge: MIT Press, 2017. 310 p.
12. Doerrfeld B., Krohn A., Sandoval K., Pedro B. The API Lifecycle: An Agile Process for Managing the Life of an API. Nordic APIs AB, 2024.

- 13.Erl T., Cope R. Cloud Computing Design Patterns. Prentice Hall, 2015. 832 p.
- 14.Fowler S. J. Production-Ready Microservices. O'Reilly Media, 2017.
- 15.Goldberg R. Architecture of Virtual Machines. IEEE Computer. 1974. Vol. 7(6). P. 34–45.
- 16.Huerga J. R., Kovalevych A., Buchanan R., Lee D., Mooy C., Bruhiere X. Kong: Becoming a King of API Gateways. Bleeding Edge Press, 2018.
- 17.Indrasiri K., Siriwardena P. Microservices for the Enterprise. Edited by Berendson L. Apress Media LLC, 2018.
- 18.Jin B., Sahni S., Shevat A. Designing Web APIs: Building APIs That Developers Love. O'Reilly Media, 2018.
- 19.Kavis M. J. Architecting the Cloud: Design Decisions for Cloud Computing Service Models (SaaS, PaaS, and IaaS). John Wiley & Sons, Inc., 2014.
- 20.Kleppmann M. Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems. O'Reilly UK Ltd, 2017.
- 21.Lynx T. Modern Hypervisor Architectures: Concepts and Evolution. London: Elsevier, 2018. 392 p.
- 22.Morgan B., Pereria P. A. Microservices in Action. Edited by Stephens M. Manning Publications Co., 2019.
- 23.Nadareishvili I., Mitra R., McLarty M., Amundsen M. Microservice Architecture: Aligning Principles, Practices, and Culture. O'Reilly, 2016.
- 24.Nayyar A. Handbook of Cloud Computing: Basic to Advance Research on the Concepts and Design of Cloud Computing. BPB Publications, 2019.
- 25.Newman S. Building Microservices: Designing Fine-Grained Systems. O'Reilly Media, 2015.
- 26.Pacheco V. F. Microservice Patterns and Best Practices: Explore Patterns like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices. Packt Publishing, 2018.
- 27.Red Hat OpenShift Documentation. Red Hat Inc., 2023.

28. Richardson C. *Microservices Patterns*. Edited by Michael M. Manning Publications Co., 2019.
29. Rosser D. Performance Issues in Virtualized Environments // *ACM Computing Surveys*. 2016. Vol. 48(2). P. 1–25.
30. Sayfan G. *Hands-On Microservices with Kubernetes: Build, Deploy, and Manage Scalable Microservices on Kubernetes*. Packt Publishing, 2019.
31. Sharma R., Mathur A. *Traefik API Gateway for Microservices: With Java and Python Microservices Deployed in Kubernetes*. Springer Nature Customer Service Center LLC, 2020.
32. Smith J., Nair R. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, 2005. 648 p.
33. Subramanian H. *Hands-On RESTful API Design Patterns and Best Practices*. Packt Publishing, 2019.
34. Vernon V. *Implementing Domain-Driven Design*. Addison Wesley, 2013.

ДОДАТКИ

Додаток А

ТЕХНІЧНЕ ЗАВДАННЯ НА РОЗРОБКУ ІНСТРУМЕНТУ КЕРУВАННЯМ ПРИКЛАДНИМ ПРОГРАМНИМ ІНТЕРФЕЙСОМ МІКРОСЕРВІСІВ

Погоджено:

Д.п.н, доцент кафедри кібернетики та
інформатики

Юлія РУДЕНКО

Суми 2025

1.Розробка компонента API-gateway

Компонент API-gateway виконує функції маршрутизації HTTP трафіку. Як основу цього компонента було обрано програмний продукт «Envoyproxy». Розробник «Envoyproxy» опубліковує програмний продукт у різних формах, включаючи готові до запуску бінарні файли та формат докер-контейнера. У реалізації компонента API-gateway ми візьмемо за основу envoyproxy/envoy-alpine:v1.17.0. На даний момент версія 1.17.0 є останньою версією Envoyproxy. Приписка alpine вказує на те, що цей образ заснований на мінімалістично можливої версії ОС Linux - Alpine.

Докер-образ «Envoyproxy» надається без конфігурації і має на увазі, що користувач її напише і помістить всередину цього образу. Наше завдання вказати «Envoyproxy», звідки забирати конфігурацію після старту.

Структура конфігурації Envoyproxy має свою специфіку. Для початку настройте сторінку адміністрування, щоб було просто переглянути поточний стан програми в контейнері, а також побачити повний зліпок поточної конфігурації. На рис. 22 зображено частину конфігурації, яка відповідає за налаштування сторінки адміністрування. З такою конфігурацією Envoyproxy прослуховуватиме на локальному хосту порт 9903 в режимі сумісності зі стандартом IPv4.

Для того, щоб «Envoyproxy» завантажував конфігурацію з компонента API-publisher за протоколом gRPC потрібно скористатися секцією "dynamic_resources".

```

admin:
  access_log_path: /dev/null
  address:
    socket_address: { address: 0.0.0.0, port_value: 9903, ipv4_compat: true

```

Рисунок 1. – Конфігурація сторінки адміністрування «Envoyproxy»

На рис. 1. зображено конфігурацію, в якій описано, що брати сутності Cluster і Listener потрібно з кластера з ім'ям `xds_cluster`. Також зазначено, що джерело конфігурації `"xds_cluster"` працює з ресурсами версії `"V3"`, використовує протокол `"GRPC"`.

```
dynamic_resources:
  cds_config:
    resource_api_version: V3
    api_config_source:
      api_type: GRPC
      transport_api_version: V3
      grpc_services:
        - envoy_grpc:
            cluster_name: xds_cluster
  lds_config:
    resource_api_version: V3
    api_config_source:
      api_type: GRPC
      transport_api_version: V3
      grpc_services:
        - envoy_grpc:
            cluster_name: xds_cluster
```

Рисунок 2. - Конфігурація динамічних ресурсів «Envoyпроху»

Наступним кроком оголосимо конфігурації кластер, який ми згадали в секції динамічних ресурсів вище. Ця настройка оголошується в статичній секції конфігурації, як наперед відома і незмінна. На рис. 24 зображено повну конфігурацію кластера `«xds_cluster»`.

```

static_resources:
  clusters:
  - name: xds_cluster
    connect_timeout: 1s
    type: STRICT_DNS
    lb_policy: ROUND_ROBIN
    typed_extension_protocol_options:
      envoy.extensions.upstreams.http.v3.HttpProtocolOptions:
        "@type": type.googleapis.com/envoy.extensions.upstreams.http.v3.HttpProto
colOptions
        explicit_http_config:
          http2_protocol_options:
            connection_keepalive:
              interval: 30s
              timeout: 5s
    load_assignment:
      cluster_name: xds_cluster
      endpoints:
      - lb_endpoints:
        - endpoint:
            address:
              socket_address:
                address: api-publisher
                port_value: 15010

```

Рисунок 3 - Конфігурація джерела конфігурації «Envoyproxу»

Вона включає безліч налаштувань рівня протоколу транспорту, проте для нас важливо, щоб як тип кластера був вказаний «STRICT_DNS», що дозволяє вказувати як адресу кінцевої точки не IP адресу, а DNS ім'я, що дуже важливо, враховуючи, що в хмарі у сервісу IP адреса може змінюватися від перезапуску і призначається динамічно. Як адресу вкажемо ім'я сервісу api-publisher, а прив'язку конкретного запущеного контейнера прив'яжемо до цього імені відповідною конфігурацією вже рівня PaaS. Також далі як номер порту вкажемо «15010» і зафіксуємо це значення, щоб потім під час реалізації компонента «api-publisher» прослуховувати саме цей порт.

Далі нам необхідно створити свій докер-образ, в якому ми скопіюємо конфігурацію та вкажемо яким шляхом в контейнері вона розташована, а також прив'яжемо запущений екземпляр «Envoyproxу» до кластера.

Прив'язка до кластера потрібна для того, щоб зі збільшенням кількості запускених екземплярів «Envoyproxу», кожен екземпляр забирав з «api-publisher» одну й ту саму конфігурацію.

Для запуску Envoyproxу напишемо простий shell-скрипт, який додає кілька параметрів для позначки з яким ім'ям кластера (- service-cluster) запустити екземпляр, з яким ім'ям вузла (-service-node) і в якому каталозі розташований статичний файл конфігурації, написаний нами раніше. Envoyproxу буде передавати в запиті зазначені параметри вузла і кластера при зверненні до джерела конфігурації (api-publisher).

```
#!/bin/sh  
  
envoy --service-cluster "${POD_HOSTNAME}" --service-node "${POD_HOSTNAME}" -  
с '/envoy/envoy.yaml'
```

Рисунок 4– Shell скрипт запуску «Envoyproxу»

Файл конфігурації та скрипт запуску – це все, що необхідно, щоб створити свій докер-образ.

На рис. 26 зображено готову конфігурацію для запуску сервісу «api-gatewa» у хмарному середовищі, прокоментуємо її. На першому рядку вказано з якого докер-образу зібрати поточний. Далі ми всередині образу встановлюємо утиліти gettext і coreutils для зручності використання контейнера.

Наступна інструкція додає всередину образу змінну оточення ENVOY_UID зі значенням нуль, що запущений Envoyproxу розпізнає як прапор для роботи від імені кореневого користувача.

```
FROM envoyproxy/envoy-alpine:v1.17.0

RUN apk --no-cache add gettext coreutils

ENV ENVOY_UID=0
USER 10001

COPY config.yaml /envoy/envoy.yaml
COPY run-gateway.sh /envoy/run-gateway.sh

EXPOSE 8080 9903

USER root
RUN chmod -R 777 /envoy
USER 10001

CMD ["/envoy/run-gateway.sh"]
```

Рисунок 5 – Dockerfile для створення образу «api-gateway»

Далі йде помітка, що скрипт складання докер-образу працює від імені користувача «1001» це необхідно для того, щоб контейнер з цього докер-образу міг працювати в середовищі «Kubernetes», де політикам безпеки заборонено запускати докер-контейнери від імені root користувача. Наступні команди копіюють файл конфігурації та скрипт до каталогу /envoy в образі. Далі робиться помітка, що контейнер з відкритими портами 8080 і 9903. Ця команда в середовищі «Kubernetess» не має значення, оскільки такі налаштування задаються на рівні конфігурації розгортання, однак поза середовищем PaaS без цих налаштувань docker-container не буде працювати так, як очікується. Наступною парою здійснюється видача прав на каталог /envoy у вигляді «повний доступ всім». На останньому рядку ми вказуємо, яким чином контейнер буде запускатися, якою командою. І як цю команду вказуємо написаний нами скрипт.

2. Розробка компонента API-publisher

Завдання компонента API-publisher слідкувати за тим, чи була опублікована в Kubernetes конфігурація, що описує маршрути мережі і якщо така є опублікувати її в компоненті API-gateway.

Для успішної реалізації цього компонента необхідно вирішити три проблеми:

- описати схему конфігурації використовуючи за основу діаграму сутностей із розділу 0;
- реалізувати функцію відстеження зміни конфігурації на хмарі;
- реалізувати реакцію на подію зміни конфігурації на хмарі з подальшою її конвертацією та публікацією у компоненті apigateway.

Щоб Kubernetes міг зберігати та обробляти конфігурації певного нами формату, нам необхідно сформувати опис нашого формату.

Kubernetes для самописних ресурсів вимагає сформувати ресурс типу "CustomResourceDefinition". Перед тим, як його створювати, варто сформувати приблизний формат ресурсу, який ми будемо використовувати для опису нашої конфігурації, беручи за основу модель, зображену на малюнку 16.

На рис. 27 зображено повноцінний приклад такої конфігурації. Він повністю відповідає описаній нами раніше моделі, включаючи такі поля як apiVersion, kind, metadata, spec, які додані виключно через специфіку Kubernetes, який вимагає наявності цих полів. На основі конфігурації вище складемо його опис. На рис. 28 зображено опис ресурсу конфігурації API, а сама схема через громіздкості винесена в додаток А.

Kubernetes вимагає від нас в описі ресурсу вказувати групу (group), тип ресурсу (kind), а також список версій (versions) зі схемами, що дозволяють перевіряти ресурси такого типу, групи та версії на

достовірність, що завантажуються в Kubernetes.

Наприклад, зображена на рис. 27 конфігурація при публікації Kubernetes буде перевірена на достовірність за допомогою схеми, описаної в додатку А.

```
apiVersion: apimanagement.cloud/v1alpha1
kind: APIConfig
metadata:
  name: example-api-config
spec:
  gateway: gateway
  routes:
  - destination:
      address:
        host: service
        port: 8080
      rules:
      - match:
          path: /api/v1/service
          headers:
            - name: X-Header
              value: x-value
          pathRewrite: /
```

Рисунок 6 – Приклад формату конфігурації, що зберігається в Kubernetes.

Маючи опис формату конфігурації для Kubernetes та саму конфігурацію, ми отримуємо можливість зберігати у Kubernetes конфігурації нашого формату. Також тепер можна відстежувати ці конфігурації, причому відстежувати будь-які зміни від видалення і закінчуючи додаванням.

Однак, щоб здійснити вирішення наступної проблеми, а саме відстеження, потрібно налагодити для нашого додатка комунікацію з Kubernetes.

Для цього існує бібліотека [k8s.io/client-go](https://github.com/kubernetes/client-go), вихідний код якої та документація розташовані на ресурсі <https://github.com/kubernetes/client-go>.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: apiconfigs.apimanagement.cloud
spec:
  group: apimanagement.cloud
  names:
    kind: APIConfig
    listKind: APIConfigList
    plural: apiconfigs
    shortNames:
    - apicfg
    singular: apiconfig
  scope: Namespaced
  versions:
  - name: v1alpha1
    schema: ...
    served: true
    storage: true
```

Рисунок 7 - Частина ресурсу, що описує конфігурацію API

Опис використання цієї бібліотеки в цій роботі буде опущено через громіздкість. Однак варто відзначити кроки, які були зроблені, щоб "навчити" програму відстежувати зміни конфігурації за допомогою цієї бібліотеки.

Першим кроком був опис використовуваних структур у форматі мови Golang (див. рис. 8).

```

11 // +resource:patn=apimanagement
12 type APIConfig struct {
13     metav1.TypeMeta `json:",inline"`
14     metav1.ObjectMeta `json:"metadata"`
15     Spec             APIConfigSpec `json:"spec"`
16 }
17
18 // +k8s:deepcopy-gen=true
19 // +k8s:deepcopy-gen:interfaces=k8s.io/apimachiner
20 // +resource:path=apimanagement
21 type APIConfigList struct {
22     metav1.TypeMeta `json:",inline"`
23     metav1.ListMeta `json:"metadata"`
24     Items           []APIConfig `json:"items"`
25 }
26
27 // APIConfigSpec is configuration of routes for AI
28 // +k8s:deepcopy-gen=true
29 type APIConfigSpec struct {
30     Gateway string `json:"gateway"`
31     Routes  []Route `json:"routes"`
32 }

```

Рисунок 8 – Частина коду, що описує структури конфігурації

Далі, на основі цього було згенеровано кілька клієнтів, які дозволяють працювати з Kubernetes API та безпосередньо відстежувати конфігурації.

Робота з клієнтом досить проста, ми вибираємо простір імен, в якому будемо здійснювати спостереження за конфігурацією та отримуємо канал, яким будемо отримувати події.

Для реалізації функції публікації конфігурації було використано бібліотеку github.com/envoyproxy/go-control-plane, яка дозволила використовувати вже готові структури для створення конфігурації.

Для того щоб Envoyproxy міг забрати конфігурацію, що публікується, необхідно запустити сервер, що обслуговує gRPC запити, а також зареєструвати сервіси для обробки запитів на кожну з сутностей.

Алгоритм запуску програми api-publisher досить простий і уміщається на невеликій діаграмі активностей на рисунку 9

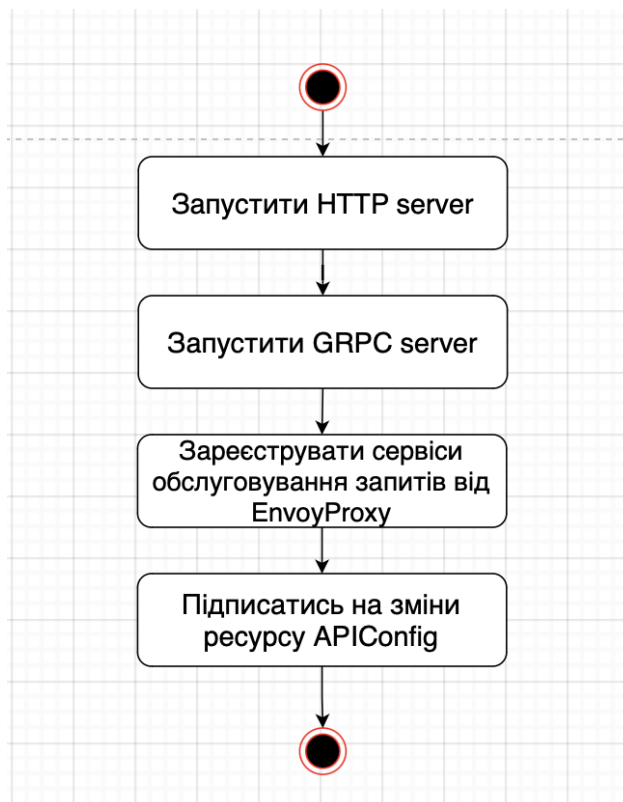


Рисунок 9 – Алгоритм старту компонента api-publisher

З алгоритму стає очевидним, що для роботи компонента необхідний HTTP-сервер та gRPC. Якщо з сервером gRPC все прозоро – він потрібен для обслуговування запитів на отримання конфігурації від EnvoyProxy, то з наявністю HTTP-сервера виникають закономірні питання.

HTTP-сервер необхідний, щоб Kubernetes міг відстежувати стан компонента. Для цього ми не просто запустили сервер, який обслуговує будь-які HTTP запити на порту 8080, а також додатково обробляємо запити, що надходять шляхом /health.

Це для того, щоб повернути актуальний стан запущеного компонента і в разі потреби дати Kubernetes зрозуміти, що з нашим додатком щось не так, коли це дійсно так.

Алгоритм, що обробляє події від Kubernetes, зображений на малюнку 10.



Рисунок 10 – Алгоритм обробника події зміни ресурса

Інша частина логіки компонента `api-publisher`, на якій варто звернути увагу, – обробка події зміни ресурсу. Після того, як Kubernetes сповістить наш додаток у нас є кілька варіантів, як вчинити.

Перше – обробити тип події (додавання, зміна, видалення) і з урахуванням типу вчинити відповідним чином, перетворивши конфігурацію для Envoyпроху. Однак зліпок даних конфігурації для Envoyпроху нам змінювати не можна.

Це пов'язано з обмеженням бібліотеки `go-control-plane`, і навіть специфікою роботи протоколу отримання динамічної конфігурації. Прийнято

рішення ігнорувати тип події та вчинити простіше – діставати на подію всі конфігурації, що існують у певному просторі імен Kubernetes та конвертувати їх у готову конфігурацію Envoyroху.

Потім опублікувати, щоб Envoyroху міг його безперешкодно забрати.

3. Розробка компонента operator

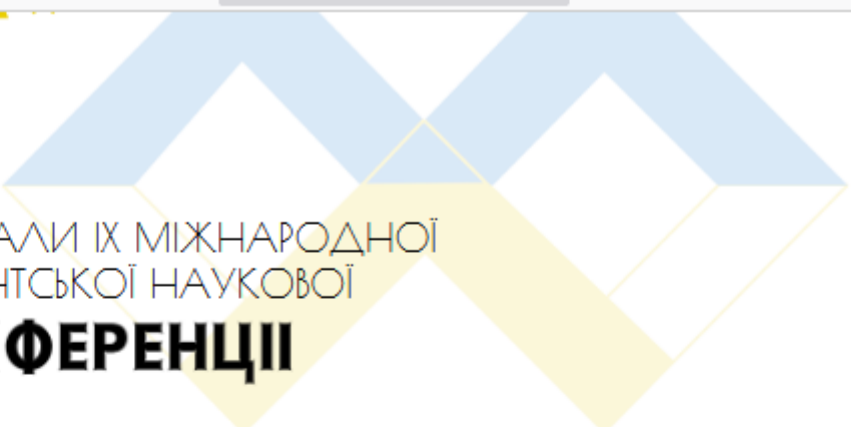
За основу компонента operator ми візьмемо реалізацію shell-operator, проте допишемо своїм функціоналом.

Shell-operator призначений для того, щоб на основі базового механізму спостереження за змінами конфігурацій, що надається Kubernetes, робити свої реалізації шаблону проектування «operator». Основою розширення shell-operator є звані гаки (hook).

Кожен «гачок» — це скрипт, який при виклику з параметром – config повертає конфігурацію, на основі якої shell-operator розуміє, коли основну частину скрипту варто запускати. На поточному етапі проектування достатньо, щоб operator при старті конфігурації розгортання для інших компонентів.

Через масивність цих конфігурацій повна їхня версія в даній роботі наводиться не буде, проте весь вихідний код, а також файли конфігурації та скрипти розгортання та запуску опубліковані за адресою <https://github.com/dmol5E/api-management-app>.

Додаток Б



МАТЕРІАЛИ ІХ МІЖНАРОДНОЇ
СТУДЕНТСЬКОЇ НАУКОВОЇ
КОНФЕРЕНЦІЇ

.....

**АКТУАЛЬНІ ПИТАННЯ ТА
ПЕРСПЕКТИВИ ПРОВЕДЕННЯ
НАУКОВИХ ДОСЛІДЖЕНЬ**

.....

м. Рівне, Україна
28 листопада 2025 рік

Вінниця, Україна
«UKRLOGOS Group»
2025

АРХІТЕКТУРНІ ТА ПРОГРАМНІ ПІДХОДИ ДО ПІДВИЩЕННЯ ЕНЕРГОЕФЕКТИВНОСТІ GPU У СИСТЕМАХ НРС	
Марусиченко О., Науковий керівник: Деркач Т.М.	564
ВАЖЛИВІСТЬ КЛІЄНТ-СЕРВЕРНОГО ЗАСТОСУНКУ В СУЧАСНИХ ІНФОРМАЦІЙНИХ СИСТЕМАХ	
Івасенько П.-І.І., Науковий керівник: Хохлов В.А.	567
ВИКОРИСТАННЯ API ДЛЯ ІНТЕГРАЦІЇ ВЕБ ЗАСТОСУНКІВ ІЗ ЗОВНІШНІМИ СЕРВІСАМИ НА ПРИКЛАДІ ПЛАТФОРМИ CANDIDESK	
Меньков К.В., Науковий керівник: Іванченко М.Г.	569
ВИКОРИСТАННЯ КОНТЕЙНЕРИЗАЦІЇ У РОЗРОБЦІ ТА РОЗГОРТАННІ МІКРОСЕРВІСІВ	
Лапченко І.В.	572
ВИЯВЛЕННЯ ТА АНАЛІЗ РИЗИКІВ ПІД ЧАС РОЗРОБКИ МОБІЛЬНОГО ЗАСТОСУНКУ ДЛЯ ПСИХОЛОГІЧНОЇ ПІДТРИМКИ КОРИСТУВАЧІВ	
Пазушкіна Д.І., Науковий керівник: Сичков В.С.	574
ДЕТАЛІЗАЦІЯ ЗАВДАНЬ ПРОЄКТУ РОЗРОБЛЕННЯ ЕЛЕКТРОННОГО ЩОДЕННИКА ЗА ДОПОМОГОЮ СИСТЕМИ УПРАВЛІННЯ ПРОЄКТАМИ ATlassian JIRA	
Присяжнюк Б.М., Науковий керівник: Сичков В.С.	577
ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ ВИСОКОНАВАНТАЖЕНИХ ПРОЄКТІВ	
Олейник М.В., Науковий керівник: Вечірська І.Д.	581
ЗАСТОСУВАННЯ МЕТОДУ «ШЕСТИ КАПЕЛЮХІВ МИСЛЕННЯ» ДЛЯ ВДОСКОНАЛЕННЯ ПРОГРАМНОЇ СИСТЕМИ УПРАВЛІННЯ ФЕРМЕРСЬКИМ ГОСПОДАРСТВОМ У МЕЖАХ AGILE-РОЗРОБКИ	
Волошенюк В.М., Науковий керівник: Сичков В.С.	583
МЕТОДИ ЗМЕНШЕННЯ РИЗИКІВ ПІД ЧАС РОЗРОБЛЕННЯ ПРОГРАМНОЇ СИСТЕМИ БРОНЮВАННЯ ГОТЕЛЬНИХ НОМЕРІВ	
Лепко Д.І., Науковий керівник: Сичков В.С.	586

Лапченко Іван Володимирович, здобувач вищої освіти факультету економіки та менеджменту
Сумський Національний Аграрний Університет, Україна

ВИКОРИСТАННЯ КОНТЕЙНЕРИЗАЦІЇ У РОЗРОБЦІ ТА РОЗГОРТАННІ МІКРОСЕРВІСІВ

У сучасних умовах стрімкого розвитку інформаційних технологій та збільшення попиту на високонадійні, масштабовані та легко адаптивні програмні рішення мікросервіси стають все більш складними, але в міру ускладнення системи виникає проблема ефективного управління розробкою, тестуванням та розгортанням цих мікросервісів. Контейнеризація, яка реалізована за допомогою таких технологій, як Docker, стала однією з найбільш перспективних та популярних практик у цій галузі.

Оскільки контейнери забезпечують окреме середовище для кожного мікросервісу, інтеграція значно спрощується, зменшується залежність від середовищ розробки та розгортання, а також підвищує продуктивність командної роботи. Це робить контейнеризацію ключовим інструментом для вирішення сучасних викликів в IT-індустрії, зокрема у розробці програмних продуктів із використанням мікросервісної архітектури.

Дослідження використання контейнеризації при розробці та розгортанні мікросервісів є важливим і необхідним для вдосконалення процесу розробки програмного забезпечення та підвищення його якості та ефективності.

Контейнеризація дозволяє значно підвищити ефективність розробки, тестування та розгортання програмних рішень. Зокрема, технологія Docker забезпечує можливість упаковувати додатки разом з усіма їх залежностями у стандартизовані образи, які можуть бути запущені в будь-якому середовищі, незалежно від його особливостей. Це забезпечує ізоляцію мікросервісів один від одного, зменшує ризики конфліктів між середовищами, а також прискорює процеси CI/CD (Continuous Integration/Continuous Deployment) [4].

Архітектура мікросервісів складається з набору незалежних сервісів, кожен з яких виконує окрему бізнес-логіку [5]. Є важливим інструментом для простого розширення послуг, швидкого створення тестових середовищ та спрощення процесу застосування оновлень. Практичне застосування контейнеризації включає автоматизацію створення Docker-образів, оркестрацію контейнерів за допомогою платформ на кшталт Kubernetes, а також інтеграцію з системами моніторингу та логування [2].

Контейнеризація також відіграє важливу роль у забезпеченні незалежності мікросервісів, дозволяючи розробникам працювати автономно над окремими компонентами системи, не впливаючи на інші. Це значно підвищує продуктивність і прискорює розробку складних програмних систем:

1. Docker як базовий інструмент контейнеризації: Docker є найпопулярнішою платформою для створення, управління та розгортання контейнерів. Його використання дозволяє стандартизувати процеси розробки, усунути проблеми несумісності між середовищами і значно прискорити життєвий цикл програмного забезпечення.

2. Створення Docker-образів для мікросервісів: Створення власних Docker-образів для кожного мікросервісу із включенням усіх необхідних бібліотек, залежностей та налаштувань. Це забезпечує стабільність роботи додатку в різних середовищах, таких як розробка, тестування [1].

3. Контейнеризація як інструмент забезпечення інтеграції середовища: використання контейнерів дозволяє створювати однорідні середовища для інтеграційного тестування, що значно спрощує виявлення помилок і конфліктів між мікросервісами ще на ранніх етапах розробки.

Контейнеризація виступає ключовою технологією, що дозволяє вирішити низку проблем сучасної мікросервісної архітектури, таких як масштабованість, ізолюваність, простота управління та розгортання. Її використання сприяє значному підвищенню ефективності розробки програмного забезпечення, скорочує час на налаштування середовищ і забезпечує стабільність роботи мікросервісів у різних умовах [3].

Технологія Docker, як основний інструмент контейнеризації, стала невід'ємною частиною процесів DevOps, забезпечуючи автоматизацію та спрощення управління інфраструктурою. Подальший розвиток контейнеризації та її інтеграція з оркестраційними платформами, такими як Kubernetes, дозволяє впроваджувати масштабовані та гнучкі рішення, що відповідають сучасним вимогам IT-індустрії.

Таким чином, контейнеризація є невід'ємною частиною інноваційного підходу до розробки, який забезпечує успішне впровадження та підтримку програмних рішень у світі швидких змін.

Список використаних джерел:

1. Docker. *Docker*. URL: <https://www.jenkins.io/doc/book/installing/docker/> (дата звернення: 20.11.2024).
2. Kubernetes – що це, як працює популярна система оркестрації контейнерів і коли вона потрібна. *blog.colobridge.net*. URL: <http://surl.li/mqdodm> (дата звернення: 20.11.2024).
3. Production-Grade container orchestration. *Kubernetes*. URL: <https://kubernetes.io/> (дата звернення: 20.11.2024).
4. What is CI/CD?. *Red Hat - We make open source technologies for the enterprise*. URL: <https://www.redhat.com/en/topics/devops/what-is-ci-cd> (дата звернення: 20.11.2024).
5. Zmerzlyi I. Мікросервісна архітектура. *Medium*. URL: <https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d> (дата звернення: 20.11.2024).

Додаток В

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ В. Н. КАРАЗІНА

ПРОЄКТ

**ІТ-ПРОСТІР СЬОГОДЕННЯ:
ТЕНДЕНЦІЇ, ІННОВАЦІЇ
ТА ПЕРСПЕКТИВИ РОЗВИТКУ**

Збірник тез доповідей
II Міжнародної науково-практичної студентської
конференції

(15 жовтня 2025 року, м. Харків, Україна)

Електронний ресурс

Харків – 2025

Колісниченко М.В. ВИКОРИСТАННЯ РІВНІВ ДЕТАЛІЗАЦІЇ (LOD) У МЕРЕЖЕВИХ ІГРАХ ДЛЯ ЗНИЖЕННЯ НАВАНТАЖЕННЯ.....	385
Корепанов І.С. PYTHON ТА JUPYTER NOTEBOOK ЯК ІНСТРУМЕНТИ ДЛЯ РОЗВИТКУ STEM-ОСВІТИ.....	387
Косовський С.Б. ВИКОРИСТАННЯ ІГРОВОГО РУШІЮ UNITY ДЛЯ СТВОРЕННЯ AR ТА VR ОСВІТНЬО-ІГРОВИХ ПРОГРАМ.....	389
Кулалаєв Н. А. ДОСЛІДЖЕННЯ ВПЛИВУ ГНУЧКИХ МЕТОДОЛОГІЙ НА ЕФЕКТИВНІСТЬ ПРОЦЕСУ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....	392
Лапченко І. В. ВИСОКОДОСТУПНА АРХІТЕКТУРА МІКРОСЕРВІСІВ У KUBERNETES...	395
Лебідь О.В. ІНДУСТРІАЛЬНІ ЗАСТОСУВАННЯ МАШИННОГО НАВЧАННЯ: АКТУАЛЬНІ ТРЕНДИ ТА ПЕРЕХІД ДО AUTOML У МАСШТАБАХ 2025 – 2030 РОКІВ.....	397
Макарова Г.В., Артемчук А.В. РОЗРОБКА ТА ДОСЛІДЖЕННЯ TELEGRAM-БОТА ДЛЯ ОРГАНІЗАЦІЇ СПОРТИВНИХ ЗАХОДІВ.....	400
Макарова Г.В., Орешко Д.В. ІНТЕЛЕКТУАЛЬНИЙ МОБІЛЬНИЙ ЗАСТОСУНОК ДЛЯ АДАПТИВНИХ ТРЕНУВАНЬ.....	401
Макарова Г.В., Пархоменко О.А. ВИКОРИСТАННЯ СУЧАСНИХ ЦИФРОВИХ ТЕХНОЛОГІЙ У СФЕРІ ОСВІТИ І НАУКИ.....	403
Маніта А.С., Томашук І.В. ВПЛИВ ІННОВАЦІЙ НА ІНВЕСТИЦІЙНУ ПРИВАБЛИВІСТЬ РЕГІОНІВ УКРАЇНИ.....	405
Матвеев Є. Е. ВИКОРИСТАННЯ ТЕХНОЛОГІЇ FACE ID ДЛЯ АВТЕНТИФІКАЦІЇ КОРИСТУВАЧІВ.....	409
Мостова Г.Г. ВПРОВАДЖЕННЯ ЦИФРОВИХ ТЕХНОЛОГІЙ ТА УДОСКОНАЛЕННЯ СИСТЕМ КОНТРОЛЮ І МОНІТОРИНГУ У СФЕРІ ВИКОНАННЯ СУДОВИХ РІШЕНЬ.....	412

ВИСОКОДОСТУПНА АРХІТЕКТУРА МІКРОСЕРВІСІВ У KUBERNETES

У нинішніх умовах цифрової трансформації бізнесу та освіти вимоги до стійкості систем та їхньої доступності різко зросли. Будь-яка помилка у роботі сервісу здатна спричинити фінансові збитки, зниження довіри споживачів і порушення ключових бізнес-процесів. Тому розробка архітектур, які гарантовано підтримуватимуть безперервну роботу навіть при відміні окремих компонентів, стає надзвичайно важливою.

Kubernetes, як платформа для оркестрації контейнерів, надає потужний арсенал засобів для створення високодоступних мікросервісних рішень. Завдяки механізмам реплікації, автоматичного відновлення та географічного резервування можна конструювати системи, що відповідають сучасним стандартам надійності.

У світовій науковій та технічній літературі активно вивчаються підходи до розробки fault-tolerant архітектур у Kubernetes. У працях K. Hightower, B. Burns, J. Beda викладено основи застосування Kubernetes для забезпечення масштабованості та надійності. Приклади від провідних хмарних провайдерів (AWS, Google Cloud, Microsoft Azure) демонструють стратегії використання кластерів у кількох AZ, систем резервного копіювання та планів аварійного відновлення (Disaster Recovery). Проте в українській науковій літературі питання створення катастрофостійких рішень у Kubernetes ще недостатньо розгорнуті, що підкреслює актуальність нашого дослідження.

Високодоступна архітектура мікросервісів у Kubernetes спирається на поєднання декількох основних підходів:

Копіювання сервісів. Для безстанних (stateless) додатків використовують об'єкти Deployment, що створюють кілька однакових копій сервісу. Такий підхід забезпечує автоматичне перенаправлення навантаження на інші поди при відмові одного з них.

У випадку сервісів зі станом (stateful) доцільно застосовувати StatefulSet, який гарантує постійні ідентифікатори і зберігання даних у персистентних томах. Клонування баз даних, кешів або брокерів повідомлень (наприклад, PostgreSQL, Redis, Kafka) дозволяє запобігти втраті даних і зберегти цілісність системи.

Стратегії відновлення після збоїв. Kubernetes включає в себе механізми самовідновлення (self-healing). Коли под завершує роботу з помилкою або вузол перестає бути доступним, система сама перезапускає контейнер або переносить навантаження на інший вузол.

Крім того, Kubernetes підтримує різні підходи до оновлення додатків: Rolling Update (поетапне оновлення без простою) та Blue-Green Deployment (одночасне функціонування старої та нової версій для плавного переходу). Це дозволяє знизити ризики під час розгортання нових релізів.

Забезпечення катастрофостійкості. Для підвищення надійності інфраструктури критично важливо застосовувати кластери в декількох зонах доступності (multi-AZ). Тобто вузли Kubernetes розташовуються у різних дата-центрах або регіонах. При відмові одного дата-центру інші продовжують обслуговувати користувачів.

Для ключових систем впроваджуються стратегії Disaster Recovery (DR), що передбачають регулярне резервне копіювання (backup), використання інструментів типу Velero для відновлення кластерів і розробку сценаріїв відмовостійкості.

Моніторинг і превентивний контроль. Інтеграція з Prometheus та Grafana дає можливість збирати метрики доступності, навантаження та використання ресурсів, що забезпечує швидке виявлення аномалій і оперативне реагування на можливі збої.

Налаштування алертів і SLA (Service Level Agreement) допомагає точно контролювати відповідність роботи сервісу встановленим показникам.

Автоматизація та IaC (Infrastructure as Code). Для ефективного керування інфраструктурою використовують Helm Charts та Terraform, що уніфікують процес розгортання і мінімізують людський фактор. Автоматизація спрощує швидке відновлення кластера у випадку масштабних збоїв.

Досягнення високої доступності мікросервісів у Kubernetes вимагає всебічного підходу, який поєднує реплікацію сервісів, механізми самовідновлення, географічне резервування та автоматизацію управління інфраструктурою. Використання зазначених принципів дозволяє знизити вплив відмов, забезпечити безперервність бізнес-процесів і підвищити довіру користувачів до IT-систем.

Список використаних джерел:

1. Amazon Web Services. Amazon Elastic Kubernetes Service (EKS) URL: <https://docs.aws.amazon.com/eks>
2. Hightower K., Burns B., Beda J. Kubernetes: Up and Running. – 3rd ed. – Sebastopol, CA : O'Reilly Media, 2021. – 350 p.
3. Kubernetes Documentation URL: <https://kubernetes.io>
4. Velero. Disaster recovery for Kubernetes URL: <https://velero.io>

Додаток Г

Схема конфігурації API

```

openAPIV3Schema:
  properties:
    spec:
      properties:
        gateway:
          type: string
        routes:
          items:
            format: object
            properties:
              destination:
                properties:
                  address:
                    properties:
                      host:
                        type: string
                      port:
                        type: integer
                    type: object
                  type: object
            rules:
              items:
                format: object
                properties:
                  match:
                    properties:
                      headers:
                        items:
                          format: object
                          properties:
                            name:
                              type: string
                            value:
                              type: string
                          type: object
                        type: array
                      path:
                        type: string
                      type: object
                  pathRewrite:
                    type: string
                  type: object
              type: array
            type: object
          type: array
      required:
        - gateway
      type: object
    type: object

```