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

**Факультет інформатики та обчислювальної техніки**

**Кафедра автоматики та управління в технічних системах**

«На правах рукопису»
УДК 004.453

До захисту допущено:

Завідувач кафедри

________ Олександр РОЛІК

«___»____________20__ р.

# Магістерська дисертація

**на здобуття ступеня магістра**
**за освітньо-професійною програмою «Інтегровані інформаційні системи»**
**зі спеціальності 126 «Інформаційні системи та технології»**
**на тему: «Автоматизована система ініціалізації та управління веб-додатками у Kubernetes кластерах»**

Виконав:
студент VI курсу, групи ІА-92мп
Татарін Валерій Вячеславович ___________

Керівник:
д.т.н, професор, завідувач кафедри АУТС
Ролік Олександр Іванович ___________

Рецензент:
_______________________________________________________
_______________________________________________________ ___________

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

Київ – 2020 року

**Національний технічний університет України**
**«Київський політехнічний інститут імені Ігоря Сікорського»**
**Факультет інформатики та обчислювальної техніки**
**Кафедра автоматики та управління в технічних системах**

Рівень вищої освіти – другий (магістерський)

Спеціальність – 126 «Інформаційні системи та технології»

Освітньо-професійна програма «Інтегровані інформаційні системи»

ЗАТВЕРДЖУЮ
Завідувач кафедри
_______ Олександр РОЛІК
«___»____________20__ р.

**ЗАВДАННЯ**
**на магістерську дисертацію студенту**
**Татаріну Валерію Вячеславовичу**

1. Тема дисертації «Автоматизована система ініціалізації та управління веб-додатками у Kubernetes кластерах», науковий керівник дисертації Ролік Олександр Іванович, д.т.н.,_професор, завідувач кафедри АУТС, затверджені наказом по університету від «26» 10 2020 р. №3132-с

2. Термін подання студентом дисертації ________________________________

3. Об'єкт дослідження: процес розробки та експлуатації веб-додатків у Kubernetes кластерах

4. Вихідні дані: технічна література з розробки програм мовою Go, технічна література з паралельного програмування, документація Kubernetes, документація операційної системи Linux, матеріали мережі Інтернет, що стосуються теми роботи

5. Перелік завдання, які потрібно розробити: огляд існуючих рішень, аналіз архітектури Kubernetes, аргументація вибору інструментів створення системи, аналіз ефективності роботи системи, автоматизація розгортання тестових кластерів, графічний матеріал

6. Орієнтовний перелік графічного (ілюстративного) матеріалу: структурна схема системи, діаграма використання системи, діаграма розгортання, діаграма послідовності системи, діаграма залежностей ресурсів Terraform

7. Дата видачі завдання: ____________________

Календарний план

| № з/п | Назва етапів виконання дипломного проекту | Строк виконання етапів проекту | Примітка |
|---|---|---|---|
| 1. | Огляд існуючих рішень | 2.09.2020 р. | |
| 2. | Аналіз архітектури Kubernetes | 21.09.2020 р. | |
| 3. | Вибір інструментів для побудови системи | 27.09.2020 р. | |
| 4. | Розробка системи | 12.10.2020 р. | |
| 5. | Розгортання тестового кластеру | 9.11.2020 р. | |
| 6. | Тестування розробленої системи | 12.11.2020 р. | |
| 7. | Розробка стартап – проекту | 16.11.2020 р. | |
| 8. | Оформлення текстової документації | 28.11.2020 р. | |
| 9. | Подання готової роботи | 03.12.2020 р. | |

Студент Валерій ТАТАРІН

Науковий керівник дисертації Олександр РОЛІК

# РЕФЕРАТ

Татарін В. В. Автоматизована система ініціалізації та управління веб-додатками у Kubernetes кластерах. КПІ ім. Ігоря Сікорського, Київ, 2020.

Робота містить 141 с. тексту, 27 рисунків, 23 таблиць, 34 джерела та 3 додатки.

Зважаючи на стрімкий ріст популярності технології контейнеризації та широке використання мікросервісної архітектури, вирішення проблем що спричиняють дані технології є актуальним.

Об’єктом розробки є автоматизована система ініціалізації та управління веб-додатками у Kubernetes кластерах.

Метою магістерської дисертації є підвищення ефективності, безпеки та простоти у користуванні ініціалізації веб-додатків та пришвидшення процесу розробки програмного забезпечення.

Предметом дослідження є ініціалізація та управління веб-додатками у Kubernetes кластерах.

Розроблено підсистему для ініціалізації веб-додатків мовою Go та підсистему управління веб-додатками з використанням фреймворку Kudo.

Розроблена система має потенціал для подальшого розвитку та може застосовуватися у сферах розробки та експлуатації програмного забезпечення, особливо при використанні мікросервісної архітектури. На основі результатів розробки було створено план для стартап-проекту.

Ключові слова: Kubernetes, мікросервісна архітектура, хмарні обчислення, управляння додатками, ініціалізація додатків.

# ABSTRACT

Tatarin V. Automated system for initialization and management of web applications in Kubernetes clusters. Igor Sikorsky Kyiv Polytechnic Institute, Kyiv, 2020.

The work contains 141 p. of text, 27 figures, 23 tables, 34 references and 3 additions.

According to the rapid growth of popularity of containerization technology and wide use of microservice architecture, resolution of challenges caused by such approaches is important.

The object of development is the automated system for initialization and management of web applications in Kubernetes clusters.

The master's dessert is aimed to improve effectivity, security and simplicity of use of applications initialization and to speed up software development lifecycle.

The subject of the research is initialization and management of web applications in Kubernetes clusters.

A subsystem for applications initialization is developed in Golang and a subsystem for applications management is built with the Kudo framework.

The resulting system has the potential for further development and can be used in software development and operations fields, especially in the case of microservice architecture implementation. Based on the results, a plan for a startup project was created.

Keywords: Kubernetes, microservice architecture, cloud computing, applications management, applications initialization.

# ЗМІСТ

ВСТУП ........................................................................................................ 10

1 ПРИЗНАЧЕННЯ І ГАЛУЗЬ ЗАСТОСУВАННЯ ..................................................... 12

2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ ........................................................................... 13

2.1 Ініціалізація додатків у середовищі Kubernetes ............................................ 13

2.1.1 Файли конфігурацій Kubernetes ............................................................... 14

2.1.2 Контролер конфігурацій Berglas ............................................................... 16

2.2 Управління веб-додатками .......................................................................... 19

2.3 Висновки до розділу ................................................................................... 23

3 АНАЛІЗ ОСОБЛИВОСТЕЙ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА KUBERNETES ........................................................................................................ 24

3.1 Мікросервісна архітектура .......................................................................... 25

3.1.1 Аналіз мікросервісного підходу ................................................................ 25

3.1.2 Взаємодія між мікросервісами .................................................................. 27

3.1.3 Тестування платформ на базі мікросервісної архітектури ........................... 29

3.1.4 Особливості розгортання додатків у хмарних середовищах ........................ 31

3.1.5 Контейнеризація за допомогою Docker ...................................................... 33

3.2 Оркестратори контейнерів ........................................................................... 37

3.2.1 Огляд особливостей використання Kubernetes .......................................... 42

3.2.2 Огляд основних компонентів Kubernetes ................................................... 43

3.2.3 Аналіз типів оточень запуску контейнерів ................................................. 52

3.2.4 Поди у Kubernetes ...................................................................................... 54

3.3 Висновки до розділу ................................................................................... 58

4 РОЗРОБЛЕННЯ СИСТЕМИ ІНІЦІАЛІЗАЦІЇ ТА УПРАВЛІННЯ ВЕБ-ДОДАТКАМИ ..... 59
4.1 Аналіз вимог до робочого оточення ..... 59
4.2 Аналіз вимог до системи ..... 60
4.3 Розробка структурної схеми системи ..... 62
4.3.1 Простори імен веб-додатків ..... 62
4.3.2 Система контролю доступу на основі ролей ..... 63
4.4 Сценарії використання системи ..... 68
4.5 Вибір і обґрунтування засобів програмної реалізації ..... 69
4.5.1 Вибір мови програмування ..... 70
4.5.2 Custom Resource Definition та Kudo ..... 73
4.5.3 Вибір середовища розробки ..... 75
4.5.4 Засоби для автоматизації розробки ..... 77
4.5.5 Засоби тестування ПЗ ..... 78
4.5.6 Автоматизація розгортання Kubernetes кластера у GCP ..... 79
4.6 Короткий опис програми ініціалізації додатків ..... 81
4.7 Короткий опис системи управління додатками ..... 83
4.8 Висновки до розділу ..... 84
5 АНАЛІЗ ЕФЕКТИВНОСТІ РОБОТИ СИСТЕМИ ..... 86
5.1 Ініціалізація додатків ..... 86
5.2 Управління додатками ..... 87
5.3 Висновки до розділу ..... 88
6 РОЗРОБКА СТАРТАП ПРОЕКТУ ..... 89
6.1 Опис ідеї проекту ..... 89
6.2 Технологічний аудит ідеї проекту ..... 91

6.3 Аналіз ринкових можливостей стартап-проекту ................................................ 92

6.4 Ринкова стратегія ........................................................................................ 102

6.5 Розроблення маркетингової програми стартап-проекту .................................. 106

6.6 Висновки до розділу ...................................................................................... 110

ВИСНОВКИ ......................................................................................................... 112

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ................................................................... 114

Додаток А – Лістинг програми ініціалізації веб-додатків .........**Ошибка! Закладка не определена.**

Додаток Б – Лістинг програми управління веб-додатками........**Ошибка! Закладка не определена.**

Додаток В – Лістинг програми розгортання GKE кластеру ......**Ошибка! Закладка не определена.**

## ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧЕНЬ І ТЕРМІНІВ

ABAC – Attribute Based Access Control

API – Aplication Programming Interface

AWS – Amazon Web Services

CRD – Custom Resource Definition

CRUD – Create Read Update Delete

DNS – Domain Name System

EC2 – Elastic Compute Cloud

ELK – Elasticsearch Logstash Kibana

EKS – Elastic Kubernetes Service

EFK – Elasticsearch Fluentd Kibana

GA – Generally Available

GCP – Google Cloud Platform

GKE – Google Kubernetes Engine

HDD – Hard Disk Drive

IaaS – Infrastructure as a Service

IAM – Identity and Access Management

LTS – Long Term Support

NAT – Network Address Translation

NFS – Network File System

PaaS – Platform as a Service

SaaS – Software as a Service

SPOF – Single Point of Failure

SSD – Solid State Drive

БД – База даних

ВМ – Віртуальна машина

ОС – Операційна система

ЦОД – Центр обробки даних

## ВСТУП

Швидкий розвиток технологій і величезна кількість користувачів – одні з основних складностей при побудові складних сучасних інформаційних систем. Залишаючи програмний продукт єдиним цілим в умовах сучасного світу, досить важко змінювати його разом з потребами примхливого ринку. Саме тому слід розподіляти окремі самостійні функції систем за різними потребами та вимогами на різні компоненти. Створення програмного забезпечення з використанням мікросервісної архітектури стає все більш актуальним підходом, порівнюючи з монолітною архітектурою. Проте, досить складно контролювати велику кількість сервісів одночасно. Таким чином, в процесі розробки та підтримки програмного забезпечення з'являється новий клас проблем, серед яких такі як оновлення та узгодження версій мікросервісів, контроль за їх розгортанням та ініціалізацією, проблеми розмежування рівнів доступу до інфраструктурних ресурсів.

Віртуалізація у сучасному світі є невід'ємною складовою побудови будь-якої платформи, адже дозволяє найоптимальнішим способом використовувати фізичні обчислювальні ресурси, гарантуючи безпеку та простоту у використанні. Проте, віртуалізатори використовують доволі велику кількість обчислювальних ресурсів для емуляції повного ядра ОС, процесору та інших фізичних складових ПК. Контейнеризація ж покладається на вбудовані у ОС механізми ізоляції робочих оточень процесів. Кожен контейнер є окремим процесом з відділеною частиною пам'яті та процесорного часу.

Контейнеризація або віртуалізація на рівні операційної системи – метод віртуалізації, при якому ядро операційної системи підтримує кілька ізольованих примірників простору користувача замість одного. Ці екземпляри або контейнер з точки зору користувача повністю ідентичні окремому екземпляру операційної

системи. Для систем на базі Unix ця технологія схожа на вдосконалену реалізацію механізму chroot. Ядро забезпечує повну ізольованість контейнерів, тому програми з різних контейнерів не можуть впливати один на одного.

Однією з відмінностей між апаратною віртуалізацією, при якій повністю емулюється апаратне оточення, є те, що в контейнері може бути запущена операційна система тільки з тим ядром, що і у ОС хостової машини. Контейнеризація також не витрачає час на емуляцію віртуального обладнання, тому час на створення контейнера є значно меншим ніж віртуальної машини.

Docker – програмне забезпечення для автоматизації розгортання і управління додатками в середовищах з підтримкою контейнеризації. У своєму ядрі Docker дозволяє запускати практично будь-який додаток, безпечно ізольований у контейнері. Безпечна ізоляція дозволяє запускати на одному сервері багато контейнерів одночасно з оптимальним використанням обчислювальних ресурсів.

Для побудови цілісної платформи з використанням мікросервісної архітектури необхідним є механізм, котрий контролюватиме сотні, а іноді й тисячі контейнерів одночасно. Саме таку задачу вирішують оркестратори, найбільш розвинутим та популярним з яких є Kubernetes від корпорації Google.

Проте, контроль веб-додактів у Kubernetes кластерах є досить складним з точки зору часу та ресурсів завданням. Окрім того, етап ініціалізації додатків, що включає в себе їх запуск та конфігурування часто є складною частиною автоматизації керування платформою. Саме розробка системи для вирішення цих проблем й є завданням магістерської дисертації.

# 1 ПРИЗНАЧЕННЯ І ГАЛУЗЬ ЗАСТОСУВАННЯ

Система, що розробляється в даній магістерській дисертації призначена для підвищення ефективності, безпеки та простоти у користуванні ініціалізації веб-додатків та пришвидшення процесу розробки програмного забезпечення, а також тестування та розгортання платформ, що побудовані на базі мікросервісної архітектури. Використання системи можливе у сфері розробки та експлуатації програмного забезпечення.

У разі використання мікросервісного підходу, розроблюваний функціонал програмного запеченння може забезпечуватися з використанням великої кількості незалежних компонент (сервісів). Кожен з сервісів потребує забезпечення власного способу ініціалізації та створення певного набору інфраструктурних компонент. Управління великою кількістю конфігураційних та інфраструктурних залежностей є досить клопіткою та складною задачею. Розроблювана система покликана автоматизувати життєвий цикл даних елементів, що пришвидшить процес тестування програмного забезпечення та забезпечить вищий рівень надійності та безпеки розроблюваних програмних платформ.

Система повинна використовувати актуальні та широко розповсюджені технічні засоби реалізації для того, щоб забезпечити якомога просту та швидку інтеграцію системи у різних середовищах розробки програмного забезпечення.

# 2 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ

Для початку, оглянуто існуючі рішення щодо ініціалізації робочих навантажень Kubernetes. Аналіз комерційного ринку та рішень з відкритим кодом показав, що є досить велика кількість прогрманих комплексів, що вирішують певний перелік існуючих проблем та навіть частина з них може бути використана без додаткових компонент, за рахунок використання вбудованих в Kubernetes механізмів.

## 2.1 Ініціалізація додатків у середовищі Kubernetes

Ініціалізація додатків включає в себе завантаження необхідного образу, генерування конфігураційних файлів та, власне, запуск додатку у контейнерів. Kubernetes бере на себе процес завантаження образів, проте конфігурування та запуск додатків можна виконувати одним за багатьох способів.

Переважна більшість додатків не підтримує так зване «гаряче» конфігурування, що дозволяє змінювати параметри налаштувань без перезапуску додатків. У світі Docker перезапуск додадтку означає перезапуск контейнеру, адже через відсутніть менеджеру процесів у контейнерах неможливо жодним чином їх перезавантажувати. Проектуючи дану особливість запуску додатків у контейнерах це визначає, що для перезавантаження конфігурації додатку необхідно перестворити под, що означає перестворити непостійні томи зберігання даних та іншу компоненти, що прив’язані до поду.

Рекомендації з приводу проектування додатків свідчать, що конфігураційні параметри, що можуть змінюватися для запуску ПЗ в, наприклад, тестових та реальних оточеннях, мають бути винесені в конфігураційні файли або змінні оточення, котрі можуть бути змінені без внесення змін до програмного коду, що в даному випадку означає й перестворення образу контейнера.

Далі будуть розглянуті способи конфігурування додатків в робочому оточенні Kubernetes, створеному у середовищі хмарних обчислень GCP.

Процес запуску з може бути виконаний як звичайний запуск додатку, так і більш складними способами. Вони будуть розглянути пізніше.

### 2.1.1 Файли конфігурацій Kubernetes

Kubernetes має два основні типи файлів конфігурацій – це секрети («secret») та файл конфігурації («configMap»).

Секрети – це невеликі об’єкти, що містять конфіденційну інформацію, таку як облікові дані або ключі доступу. Вони зберігаються як відкритий текст у etcd, доступні серверу API Kubernetes, і їх можна монтувати у вигляді файлів у підсистеми (використовуючи виділені секретні томи, які перетворюються на звичайні томи даних), які потребують до них доступу. Один і той же секрет можна вмонтувати в декілька подів. Kubernetes сам створює секрети для своїх компонентів, і ви можете створювати власні секрети. Слід зауважити, що усі значення у секретах зберігаються закодовані у форматі base64 [1] (рисунок 2.1).

Проте, ніякого додаткового рівня захисту секретів Kubernetes не має: будь-хто, хто має доступ до Kubernetes – має доступ і до секретів. Така сама ситуація і при їх монтуванні як томи – будь-то, хто має доступ до файлової системи контейнеру, може прочитати секрети. З секретного у секретів Kubernetes – лише ім’я.

```
- apiVersion: v1
  data:
    ca.crt: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSURDekNDQWZPZ0F3SUJBZ0lRQ3AxOXBtZHBsZUp5R
    054TTBaVUl5ekFOQmdrcWhraUc5dzBC5qRXpNRGMwTWpVeldoY05NalF3TmpFeE1EZzBNalV6V2pBdk1TMHdLd1lE
    VlFRREV5UXdZelJoCll6YzRNeTB5Wx=
    token: ZXlKaGJHY2lPaUpTVXpJMU5pSXNJbXRwWkNJNklpSjkuZXlKcGMzTWlPaUpyZFdKbGNtNWxkR1Z6TDNObG
    NuWnBZMlZoWTJOdmRXNTBJaXdpYTNWaVpYSnVTFhCeWIyUWlMQ0pyZFdKbGNtNWxkR1Z6TG1sdkwzTmxjblpwWTJW
    aFkyTnZkVzUwTDNObFkzSmxkQzV1W==
  kind: Secret
  metadata:
    annotations:
      kubernetes.io/service-account.name: default
      kubernetes.io/service-account.uid: 817c69ce-bf6f-11ea-92c2-4201ac10001a
    creationTimestamp: "2020-12-06T10:00:22Z"
    name: default-token-x47p4
    namespace: test-namespace
    selfLink: /api/v1/namespaces/test-server/secrets/default-token-x47p4
    uid: 817eace3-bf6f-11ea-92c2-4201ac10001a
  type: kubernetes.io/service-account-token
```

Рисунок 2.1 – Типовий вигляд секрету

Файли конфігурацій схожі на секрети. Вони можуть створюватися та використовуватися у контейнерах однаковими способами. Єдина різниця між ними – це відсутність кодування у форматі base64 (рисунок 2.2). Файли конфігурацій призначені для нечутливих даних – даних конфігурації, таких як конфігураційні файли та змінні середовища, і є чудовим способом внести зміни до конфігурації контейнерів, не створюючи новий образ, наприклад для використання публічно доступних образів.


```yaml
- apiVersion: v1
  data:
    application.properties: |-
      spring.profiles.active=test,redis
      app.metrics.samplingProbability=0.001
  kind: ConfigMap
  metadata:
    annotations:
      kubectl.kubernetes.io/last-applied-configuration: |
        {"apiVersion":"v1","data":{"application.properties":"spring.profiles.active=test,redi
        s\napp.metrics.samplingProbability=0.001"},"kind":"ConfigMap","metadata":{"annotation
        s":{},"labels":{"app":"test-service","environment":"prod"},"name":"test-service-prod"
        ,"namespace":"test-service"}}
    creationTimestamp: "2020-12-02T17:07:29Z"
    labels:
      app: test-service
      environment: prod
    name: test-service-prod
    namespace: test-service
    resourceVersion: "287258135"
    selfLink: /api/v1/namespaces/test-service/configmaps/test-service-prod
    uid: 38985691-1526-11ea-b22f-4201ac100016
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```


Рисунок 2.2 – Типовий вигляд файлу конфігурацій

При такому способі конфігурації самі параметри зберігаються у межах конкретного кластеру (у etcd). Перенесення додатків між кластерами спричинить досить велику кількість проблем, адже не існує зручного способу оптимально вивантажити дані з etcd та перенести їх у сторонній кластер. Також, при виході з ладу кластеру неможливим буде простий перезапуск робочих навантажень на резервному кластері.

Даний спосіб конфігурації може бути використаний у досить обмеженій кількості випадків.

До переваг даного рішення можна віднести наступне:

– є стандартною компонентою Kubernetes, отже не створює додаткових витрат;

– має можливість встановлювати секрети як змінні оточення та окремі файли, що доступні у файловій системі;

– простота у використанні.

У свою чергу, серед недоліків можна виділити наступне:

– об’єкти типу Secret зберігаються у незахищеному вигляді;

– неможливість перенесення секретів між кластерами;

– у разі виникнення проблем з кластером, також втрачаються конфігураційні данні;

– відсутність зручного способу для внесення змін (лише через стандартну утиліту командного рядка kubectl);

– відсутність аудиту доступу;

– додаткове навантаження на системні компоненти Kubernetes.

Отже, зважаючи на вище згадані переваги та недоліки даного рішення, стандартний спосіб конфігурації додатків від Kubernetes не є оптимальним рішенням для використання. Його можливостей може бути достатньо лише для невеликих систем з відсутніми вимогами щодо відповідності певним стандартам безпеки та надійності.

### 2.1.2 Контролер конфігурацій Berglas

Berglas – додаток з відкритим програмним кодом, що дозволяє взаємодіяти зі сховищем даних Google Secret Manager, а саме створювати та отримувати дані з нього. Цей додаток може бути встановлений як незалежне ПЗ, що перед стартом основного ПЗ завантажує конфігураційні файли, що необхідні для запуску основного програмного додатку.

Корпорація Google випустила на ринок GA версію продукту під назвою «Secret Manager» у травні 2020 року [2]. Даний сервіс є PaaS рішенням зі зберігання даних у форматі ключ-значення.

Доступ до ресурсу контролюється через Google IAM політики з підтримкою гранулярності до конкретного ресурсу, що означає, що можна надати доступ до конкретного ключу, або ж до їх списку.

Дане сховище є високонадійним, що гарантує зберігання даних у мінімум як 3 географічних локаціях. Конкретний перелік локацій може бути визначений власноруч розробниками, або ж автоматично від імені Google з урахуванням географічних даних розташування клієнтів, що виконують запити до даного сервісу. Таким чином гарантується висока надійність сервісу (при виході 2 або менше ДЦ різних регіонів дані не втрачаються) та висока швидкість обробки запитів.

Також підтримується версіонування. Дана опція дозволяє зберігати більш ніж 1 версію ключа що дозволяє переглянути його історію у разі необхідності.

Аудит доступу забезпечується за рахунок сервісу Cloud Audit Logs. Таким чином, можна проглянути, які користувачі у який час які виконували запити до даного сервісу. Дана особливість є необхідністю при отриманні сертифікатів з відповідності до вимог безпеки, наприклад ISO/IEC 27001.

Крім того, Secret Manager має зручний веб-інтерфейс, що може бути використаний користувачами для створення або редагування вже існуючих ресурсів. Даний інтерфейс є частиною консолі керування інфраструктурою GCP (рисунок 2.3).

Окрім зручного веб-інтерфейсу може бути використана й утиліта командного рядка для взаємодії з ресурсами GCP під назвою «gcloud», що є досить зручною у використанні через інформативні повідомлення при виконанні та зручний пошук за командами.

API виклики виконуються з використанням протоколу TLS, що гарантує секретність даних при передачі. У самому ж сховищі дані зашифровані за допомогою алгоритму AES-256. Ключ шифрування можуть бути використані як власні, так і надані від Google для конкретного користувача.

Рисунок 2.3 – Веб-інтерфейс Secret Manager

Advanced Encryption Standard (AES) – це симетричний алгоритм шифрування, обраний урядом США для захисту секретної інформації. AES впроваджено в програмне та апаратне забезпечення у всьому світі для шифрування конфіденційних даних. Національний інститут стандартів і технологій (NIST) розпочав розробку AES у 1997 році, коли оголосив про необхідність альтернативи стандарту шифрування даних (DES), який починав ставати вразливим до атак грубої сили. AES був створений для уряду США з додатковим добровільним, безкоштовним використанням у державних або приватних, комерційних або некомерційних програмах, що надають послуги шифрування. Однак на неурядові організації, які вирішили використовувати AES, поширюються обмеження, встановлені експортним контролем США. AES-256, який має довжину ключа 256 біт, підтримує найбільший бітовий розмір і практично не ламається за рахунок алгоритмів підбору з використанням нині доступних обчислювальних потужностей, що робить його найсильнішим стандартом шифрування [2].

Серед недоліків Berglas слід зазначити, що дана утиліта не дозволяє встановлювати завантажені секрети як змінні оточення та за один запуск має змогу завантажити лише один секрет. З таким підходом при додаванні нового секрету

необхідно вносити зміни до образу контейнеру, що не є завжди бажаним підходом для роботи.

Отже, до переваг даного рішення можна віднести наступне:

- висока надійність;
- легке масштабування;
- надійні алгоритми шифрування даних;
- відсутність навантаження на системні компоненти Kubernetes;
- аудит доступу до ресурсів.

У свою чергу, серед недоліків можна виділити наступне:

- додаткові витрати на даний сервіс, хоча й невеликі;
- доступність лише у середовищі GCP;
- неможливість встановлювати секрети як змінні оточення;
- необхідність вносити зміни до образу контейнера при додаванні чи видаленні секретних змінних.

Отже, зважаючи на зазначені переваги та недоліки даного рішення слід зазначити, що даний продукт є безперечним лідером з точки зору надійності та безпеки, проте не є досить зручним у, власне, використанні. Деякі ж недоліки, як наявність накладних витрат на користування сервісом GCP Secret Manager є достатньо незначними у порівнянні з отримуваними перевагами.

## 2.2 Управління веб-додатками

Kubernetes є надзвичайно гнучким та потужним рішенням для створення складних систем. Особливо необхідним він є для платформ, що побудовані з використанням мікросервісної архітектури. Проте, беззаперечною складністю є питання оркестрації та ефективного управління робочими навантаженнями. Всю цю гнучку архітектуру необхідно певним чином версіонувати та стежити за загальним станом системи. На жаль, дана задача не вирішується за допомогою стандартних засобів Kubernetes.

Саме через це й був створений проєкт з відкритим програмним кодом для контролю конфігурацій Kubernetes кластерів Helm.

Helm – це аналог звичних пакетних менеджерів для різноманітних ОС, проте замість системних пакетів Helm вміє версіонувати та стежити за ресурсами Kubernetes. Крім того, Helm у своїй основі використовує досить потужну бібліотеку templates мови програмування Go. Таким чином, однією з підтримуваних функцій є рендер шаблонів, що дозволяє використовувати шаблони.

Helm надає підтримку для кількох важливих сценаріїв використання:

– управління складними абстракціями;
– легке оновлення та версіонування;
– повторне використання кодової бази;
– безпечні відновлення старих версій додатків (rollback) у разі виникнення проблем з новими.

Даний менеджер робочих навантажень дозволяє пакувати усі необхідні для запуску додатку ресурси у окрему одиницю, що носить назву «чарт». Таким чином, окремі додатки, що запускаються у кластері Kubernetes мають свої власні чарти, котрі можуть складатися з набору шаблонів, файлів з переліком параметрів для рендеру шаблонів та інших системних додатків.

Чарти можуть описувати навіть найскладніші програми, забезпечувати повторювану установку додатків та слугувати єдиним джерелом для розгортання великої кількості різнотипних робочих навантажень. Швидкі оновлення та спеціальні маркери дозволяють легко розгортати нові версії додатків. Досить просто ділитися чартами, які можна версіонувати і розміщувати на загальнодоступних або приватних серверах. Коли потрібно відмінити нещодавні оновлення, Helm надає єдину команду, щоб відмінити цілий набір змін, що були внесені до інфраструктури.

Зазначається, що кодова база публічних серверів з додатками є досить обширною. За необхідності, можна знайти публічний чарт майже будь-якого публічного ПЗ. Проте, слід бути обачним, адже як і у випадку з публічним Docker Registry, ніхто не контролює якість розміщеного коду та наявність в ньому шкідливих

компонент, що можуть вивести з ладу будь-яку систему. Саме тому, слід надавати перевагу офіційним чартам, адже шанс наразитися на небезпеку при їх використанні – мінімальний.

Найбільшим публічним сервером чартів є проект вже згаданої раніше спільноти CNCF – Artifact HUB через досить зручний веб-інтерфейс (рисунок 2.4). Проте, кількість пакетів, до доступні для завантаження є шаленою, адже будь-хто з бажаючих має змогу завантажити туди свою версію пакету під довільною назвою. Наприклад, пошук «ingress nginx» для найбільш популярного безкоштовного веб-серверу видає 19 результатів, що, на перший погляд, виглядають майже однаковими та не виключено, що певна частина з цих чартів може навіть не працювати належним чином, що може спричинити досить велику кількість проблем [3].

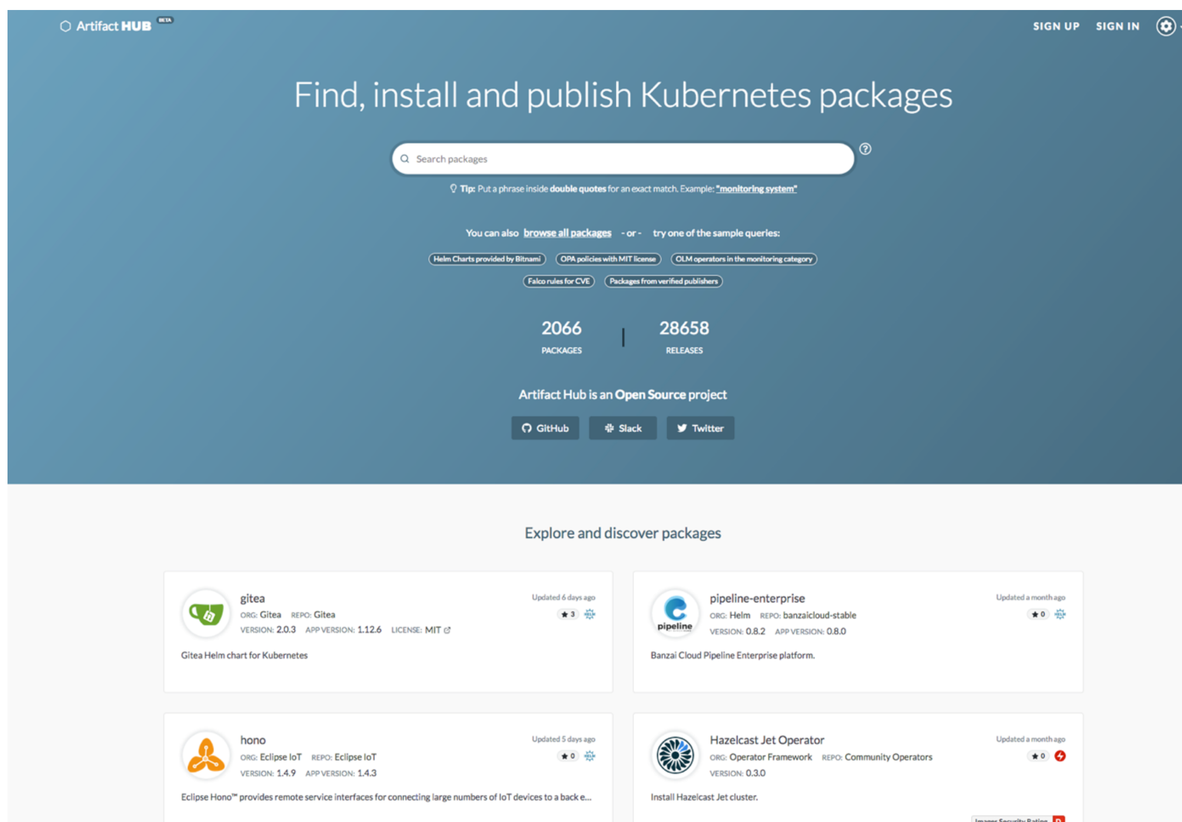


Рисунок 2.4 – Веб-інтерфейс Artifact Hub

На сайті можна з легкістю знайти чарти для більшості популярного безкоштовного ПЗ, наприклад, веб-серверу nginx (рисунок 2.5).

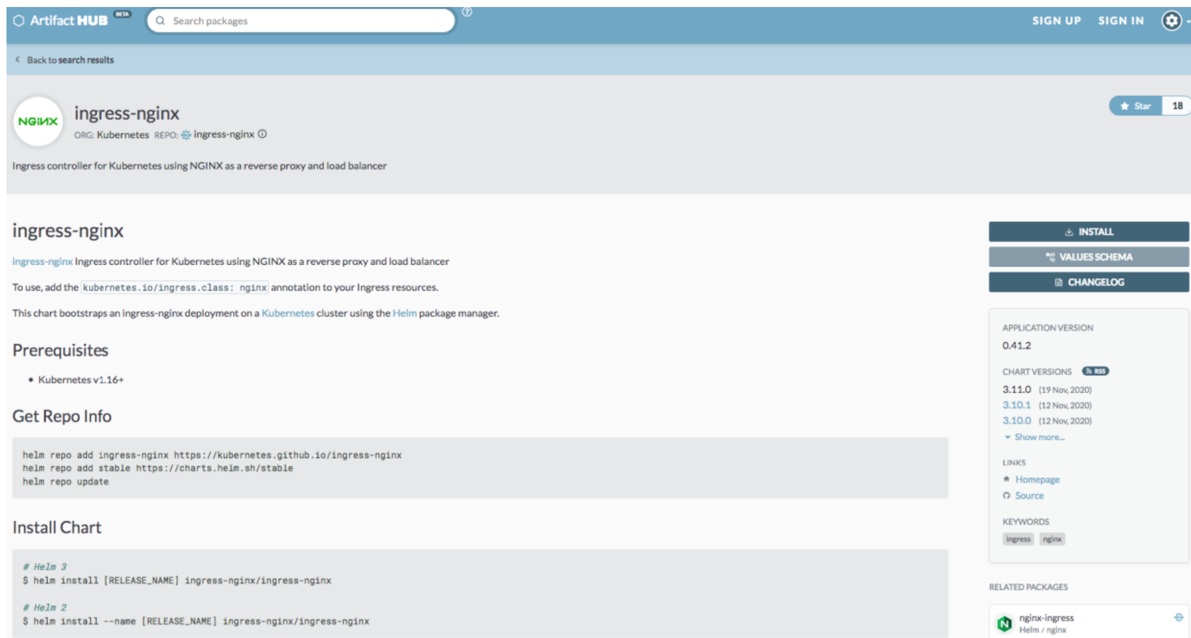


Рисунок 2.5 – Сторінка nginx на Artifact Hub

Також, досить зручною є можливість вказувати залежності між чартами. Так, наприклад, може створюватися додаток, що міститиме серед залежностей сервер певної бази даних і перед розгортанням додатку буде встановлена також база даних, котра буде згодом доступна для використання.

Отже, можна виділити Helm як досить потужне рішення для контролю робочих навантажень Kubernetes. Він має безліч переваг, серед яких [4]:

– потужний механізм шаблонізування;
– наявність відкритої публічної кодової бази;
– підтримка від CNCF;
– пряма взаємодія за API Kubernetes без посередніх механізмів;
– можливість легко відтворювати зміни з попередні версії додатків.

Серед недоліків слід зазначити наступне:

– відсутність аналогів у своєму класі;
– великий ступінь «забрудненості» публічних серверів чартами від невідомих розробників;
– досить рання стадія розвитку рішення, тому версії досить часто не є зворотньо сумісними;

– зазвичай, абсолютно неінформативні повідомленні про помилки, що робить процес пошуку несправностей надзвичайно складним;

– складний механізм шаблонізування;

– велика кількість способів реалізації одних й тих самих задач, що ускладнює пошук оптимального підходу;

– надзвичайно широкий функціонал, що розширює кодову базу додатку.

## 2.3 Висновки до розділу

Розглянуті існуючі рішення для ініціалізації та управління веб-додатками у Kubernetes кластерах мають низку переваг та недоліків. Рішення для ініціалізації додатків є занадто простими та негнучкими, що дозволяє їм задовольняти потреби лише простих систем, що не мають високих вимог щодо надійності даних та масштабування.

Серед рішень ж для управління додатками існує лише один програмний комплекс, що намагається задовольнити потреби більшості користувачів. У таких умовах ПЗ виходить всеохоплюючим, проте досить складним у користуванні та впровадженні.

Зважаючи на особливості оглянутих рішень, розробка системи автоматизації, що вбере в себе переваги вже існуючих рішень та позбавиться від їх недоліків є оптимальним підходом для вирішення проблем ініціалізації та управління веб-додатками у Kubernetes кластерах.

# 3 АНАЛІЗ ОСОБЛИВОСТЕЙ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ТА KUBERNETES

Мікросервісна архітектура – це рішення, яке може оптимізувати та полегшити процес розробки. Наразі найбільш популярною системою оркестрації є Kubernetes. Kubernetes – це система з організацією контейнерів з відкритим кодом для автоматизації розгортання, масштабування та управління. Він довгий час використовувався задля задоволення внутрішніх потреб компанії Google під назвою Borg.

Оркестрація – це процес координації взаємодії декількох контейнерів. Звичайно, можна працювати і без оркестрації – ніхто не забороняє створити контейнер, в якому будуть запущені всі необхідні процеси. Однак в цьому випадку не буде гнучкості, масштабованості, а також виникнуть питання безпеки, оскільки запущені в одному контейнері процеси не будуть ізольовані і можуть впливати друг на друга.

Оркестрація дозволяє створювати інформаційні системи з безліччю контейнерів, кожен з яких відповідає тільки за одну певну задачу, а спілкування здійснюється через мережеві порти і загальні каталоги. При необхідності кожен такий контейнер можна замінити іншим, що дозволяє, наприклад, швидко перейти на іншу версію бази даних при необхідності.

Сервіси – це ще один рівень абстракції над контейнерами. Як і у контейнера, у сервісу буде ім'я, базовий образ, опубліковані порти і томи. На відміну від контейнера, сервісу можна задати вимоги до хосту, на яких його можна запускати. Та й взагалі, сервіс можна масштабувати прямо в момент створення, вказавши, скільки саме контейнерів для нього потрібно запустити [5].

Kubernetes це складний продукт відносно конкурентних, побудова кластера якого потребує хороших знань не тільки самої системи, а й інших фундаментальних речей, як операційні системи, тестування, проектування архітектури застосунків.

Даний оркестратор вимагає має досить складну систему конфігурації робочих навантажень. Серед можливих способів конфігурації – файли у форматах YAML, JSON та інші. Проте, досить складно тримати під контролем десятки, сотні, а іноді й тисячі файлів конфігурації.

Дана магістерська дисертація присвячена створенню системи для ініціалізації та управління веб-додатками у Kubernetes кластерах. Її доцільність була обґрунтована прикладною задачею відносно планування інфраструктури кластера.

Огляд підходів у мікросервісній архітектурі та огляд архітектури Kubernetes буде розглянутий у даному розділі. Розроблювана система досить тісно залежить від особливостей даного оркестратора, адже саме з його об’єктами вона й буде оперувати. Саме тому огляд архітектури Kubernetes є важливим.

## 3.1 Мікросервісна архітектура

Мікросервісна архітектура є найбільш розповсюдженою серед підходів до розробки ПЗ у сучасний час. Даний підрозділ висвітлює переваги та недоліки цього підходу та описує проблеми, котрі породилися разом з даним підходом, оскільки розроблювана система автоматизації є вирішенням саме однієї з даних проблем [6].

### 3.1.1 Аналіз мікросервісного підходу

Довгий час програмне забезпечення розроблювалося з використанням монолітного підходу. Такий підхід зумовлює розробку багатофункціонального продукту як єдиного додатку (єдиної кодової бази, системи контролю залежностей та інше). Даний підхід з’явився історично, оскільки з самого початку програмні додатки не були настільки великими, якими їх уявляють зараз. Тому, розпочинаючи з невеликого простого додатку, він міг еволюціонувати у величезний проєкт. З використанням даного підходу, усі функції системи забезпечуються єдиною програмою. Досить часто, такі додатки мають складну багаторівневу архітектуру,

використовують єдину (нерідко перевантажену) базу даних та є незручними у розвитку та підтримці.

З часом було усвідомлено недоліки даного підходу та почали використовувати антипод монолітній архітектурі – монолітну архітектуру. Дана архітектура зумовлює розбиття системи на незалежні компоненти, кожен з яких відповідає за окрему функцію. Кожен з компонентів зазвичай є окремим незалежним програмним додатком з власною базою даних (якщо вона необхідна), власною систем контролю залежностей та інше [7].

Поява мікросервісів була неймовірно великим кроком у сфері розробки та обслуговування програмного забезпечення. З мікросервісами розроблювана платформа розбивається на окремі компоненти, котрі взаємодіють одне з одним зазвичай через інтерфейс – API.

Кожен з мікросервісів є самодостатнім, має власний шар зберігання даних та може бути незалежно оновленим, що зображено на рисунку 3.1.

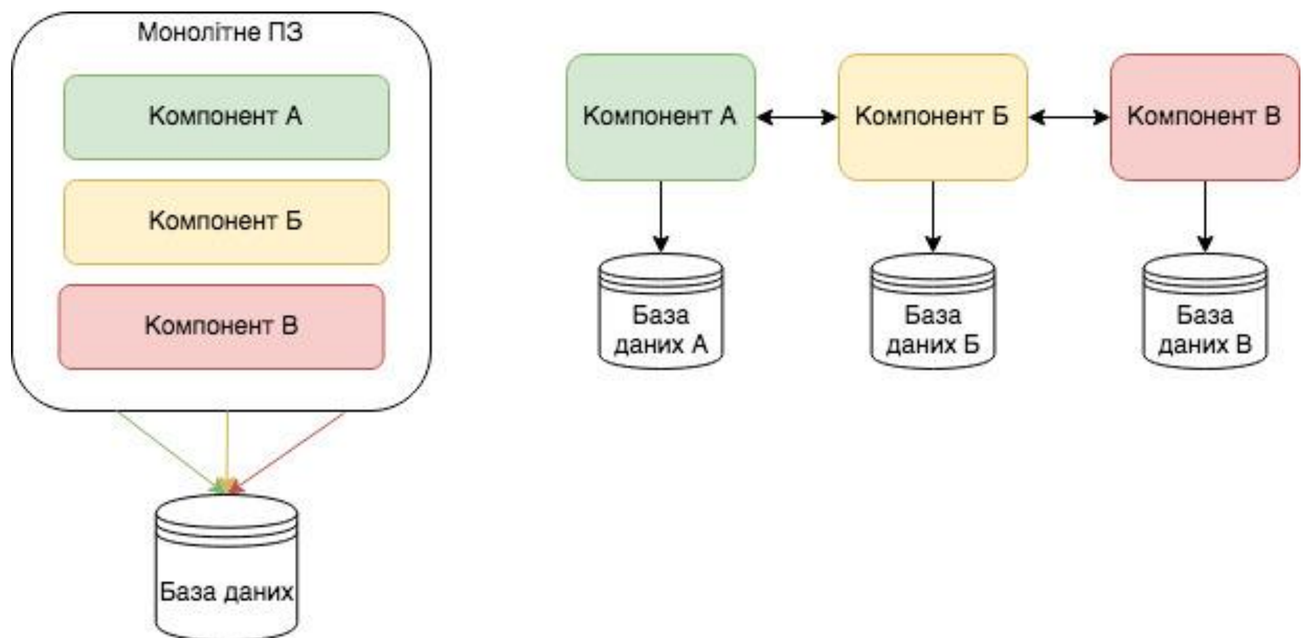


Рисунок 3.1 – Порівняння взаємодії з БД монолітних та мікросервісних додатків

Перехід до даного підходу пришвидшує та спрощує розробку програмних комплексів, адже для старту внесення необхідних змін до продукту, новим

розробникам треба набагато менше часу для того, щоб вникнути в особливості програмного коду.

Окрім того, такі додатки набагато легше розміщувати на декількох серверах. А така особливість дуже добре допомагає відповідати інколи різко змінюючимся потребам ринку. Наприклад, якщо функціонал, що надає певна частина платформи різко набув популярності та ним почало користуватися багато людей, то з легкістю можна розбалансувати навантаження на декілька серверів, що допоможе обробити стрімко збільшену кількість запитів від користувачів.

Мікросервіси – головний чинник цілої низки культурних та інженерних змін, котрі з'явилися після набуття мікросервісним підходом популярності. Методи еластичної розробки ПЗ, переміщення власних дата-центрів до хмарних, поява понять та запровадження неприривної розробки, використання контейнерів – все це наслідки появи та набуття популярності мікросервісною архітектурою, що спричинили революцію у світі розробки програмного забезпечення [7].

Окрім величезної кількості покращень процесу розробки ПЗ, з'явилася й ціла низка проблем. А саме такі:

- підвищення складності моніторингу мікросервісних платформ;
- проблема узгодженості API;
- проблема дуплікації даних;
- поява єдиних проблемних точок (SPOF);
- проблема розгортання комплексів додатків;
- проблема ініціалізації додатків на великих масштабах.

Саме останні дві проблеми й покликана вирішити система, що розроблюється в рамках даної магістерської дисертації.

### 3.1.2 Взаємодія між мікросервісами

Взаємодія між мікросервісами здійснюється за контрактами. Контракт – це умовна домовленість, яку розробники сервісів створюють для взаємодії з

клієнтськими сервісами. Такі сервіси можуть розроблюватися одними й тими самими людьми, а можуть й поставлятися у якості чорного ящику від команд з іншого відділу, або й інших компаній.

Сервіс може надавати у відповідь об'єкти будь-якого змісту, проте контракт декларує саме формат комунікації між сервісами, а не зміст. Внутрішній зміст регулюється лише на рівні бізнес-логіки продукту. Відповідно, контракт диктує зовнішнім користувачам, що вони будуть отримувати у відповідь та як отримувати до них доступ до API [6].

При збільшенні складності цього завдання, коли будуть не десятки мікросервісів, а сотні та тисячі, то традиційні, «класичні» контракти перестають задовольняти потреби. І тоді необхідно модернізувати контракти з оглядом на архітектуру мікросервісів. Розробити схему роботи для розробників, в якій вони та «кінцеві користувачі» мікросервісу обмінюються інформацією – такий підхід називається контактами користувачів.

Вважається, що користувачі можуть створювати контракти, вказувати вимоги та направляти їх до розробників сервісу, який їх виконує. Оскільки кожен сервіс має дуже обмежену кількість користувачів, отримати декілька специфікацій від кожного та впровадити їх набагато простіше, ніж намагатися здогадатися, який саме формат відповіді може бути найбільш зручним у й слід віддавати в кожному випадку і що ще може знадобитися.

Досить часто саме комунікація між командами-розробниками платформи є непростою, адже жодна з команд може не йти на зустріч, адже це може означати необхідність докладання додаткових зусиль при розробці продукту. Неузгодженість контрактів може спричинити вихід з ладу реальної системи та втрата довіри користувачів, або завади на шляху тестувальників ПЗ у процесі тестування нової версії певної зв'язки мікросервісів. Саме тому узгодженість версій контрактів між мікросервісами є надзвичайно важливою [7].

### 3.1.3 Тестування платформ на базі мікросервісної архітектури

Основою автоматизованих фреймворків для тестування є юніт-тести, відповідно до піраміди тестування, що зображена на рисунку 3.2. Такі тести перевіряють базові складові програмного продукту з використання тестових даних. Юніт – частинка у перекладі з англійської. Саме частинки такі тести й перевіряють – частинки коду такі як функції або ж класи. Процес створення юніт тестів для мікросервісів майже не відрізняється від процесу їх створення для монолітів. Проте, до переваг слід додати те, що мікросервіси – програми за набагато меншою кодовою базою, аніж звичні багатьом моноліти. Таким чином, покриття тестами мікросервісів зазвичай потребує меншої кількості часу та зусиль. Мікросервіси закликані вирішувати досить конкретний спектр задач певної бзінес-логіки, отож й можливих тест-кейсів тут дещо менше, ніж зазвичай у монолітах.

Другою невід’ємною частиною автоматизованого тестування мікросервісів є інтеграційні тести. Фаза інтеграційного тестування, в ході якої декілька компонент об’єднується та тестуються як одне ціле.

Рисунок 3.2 – Піраміда тестування

Власне, одним з недоліків мікросервісного підходу є стрімке зростання складності виконання даної фази тестування. Мікросервісна архітектура має на

меті розділення бізнес-логіки платформи на певну кількість незалежних, але тісно пов’язаних компонент. Розмір цих компонент не є чітко визначеним жодним з понять, тому він є досить варійованими у кожному окремому випадку. Таким чином, система може бути розбита на десятки, або й сотні мікросервісів. Зібрати до купи декілька з них задля тестування певної частини функціоналу – досить складна задача [6].

Кількість мікросервісів зростає, але кількість тестувальників залишається не дуже великою. Як правило, є один ручний тестувальник на один компонент загальної бізнес-логіки. Зрозуміло, що досить складно повністю покрити потреби всіх розробників у команді. В середньому це декілька розробників користувацького інтерфейсу, кілька розробників серверної частини та кілька розробників, що займаються автоматичним тестуванням тощо.

Для вирішення цієї проблеми можна почати застосовувати методологію Agile Testing. Суть цієї методології полягає в тому, щоб запобігати помилкам, а не шукати їх. Завжди слід обговорювати тестування певної версії збірки мікросервісів. Це дає змогу відразу визначати, як слід тестувати компонент: чи достатньо її охопити одним тестом, і якщо достатньо одиничної перевірки, яким типом перевірки вона повинна бути. Обговорення проводиться з тестувальником, який визначає, чи буде потрібно додаткове ручне тестування. Як правило, достатньо одного тесту. Крім того, тестувальник може запропонувати й деякі інші додаткові випадки, а також може надати контрольний список, на основі якого можна написати необхідні одиночні або функціональні тести. Проте, всі з вище зазначених способів тестування потребують розгортання конкретних версій мікросервісів разом, налаштування їх взаємодії, конфігурування кожного з них та інше [8].

Подолання даної проблеми є й одне з покликань розроблюваної системи у даній магістерській дисертації.

### 3.1.4 Особливості розгортання додатків у хмарних середовищах

Архітектура програмного забезпечення повинна відповідати конкретним потребам та обмеженням контексту середовища розгортання. Архітектура для програмного забезпечення, розгорнутого у динамічному хмарному середовищі, може надати цю підтримку, базуючись на наступних принципах [9]:

- розділення різних логічних компонент на окремі частини;
- віртуалізація: сервіси мають розгортатися з використанням спільних ресурсів та портативних контейнерів;
- невизначеність: розподілені обчислення, неоднорідність середовища та багатокористувацька;
- адаптивність: можливість швидкого завершення виконання програми та перезапуску.

Наведені особливості відображають основні принципи для розробки та розгортання додактів. Такими додатками можуть бути:

- мікросервіси: гнучка композиція з незалежними, керованими контейнерами;
- безсерверні обчислення (serverless computing): дозволяють динамічно розглядати аспекти невизначеності та адаптувати систему до них;
- додатки з реалізацією зворотнього зв’язку на основі контролера: дозволяє контролеру адаптуватися до змін (наприклад, зміни кількості обчислювальних ресурсів) та керувати ними.

Об’єднуючись, принципи та шаблони проектування пов'язують загальні концепції архітектури програмного забезпечення, такі як мікросервісність, адаптивність або безсерверні обчислення, із особливостями розгортання, такими як віртуалізація та цикли зворотного зв'язку на основі контролера. Результатом комбінації даних підходів й є використання додатків, що загорнуті у контейнери з подальшою оркестрацією контролерами з використанням зворотнього зв’язку у вигляді контролерів Kubernetes або Nomad.

Самоадаптаційна система динамічно пристосовує свою поведінку, щоб зберегти або підвищити показники якості у невизначених робочих умовах. У такому випадку, розвиток мікросервісних додатків як таких самоадаптивних систем все ще залишається недовершеним рішенням. На практиці, наприклад, платформа оркестрації контейнерів Kubernetes полегшує розгортання та управління програмами на базі мікросервісної архітектури, підтримуючи лише автоматичне масштабування (автоматичне зміна кількості екземплярів програмного додатку) та самовідновлення (автоматичний перезапуск невдалих екземплярів програмного додатку). Параметри якості мікросервісу (швидкість надання відповіді на запит, рівень завантаженості ресурсів) можна покращити за допомогою методів планування (визначити найкращу адаптаційну стратегію для кожного мікросервісу), машинного навчання (вивчити нові стратегії адаптації з минулих результатів адаптації), міркувань в умовах невизначеності (впоратися з нечіткими даними моніторингу) [9].

Однак, важливим зауваженням є те, що незалежні та часті розгортання, необхідність високого ступеня автоматизації в контексті кооперації команд розробки та підтримки робочих систем та складні архітектури реалізації роблять мікросервіси ідеальними кандидатами для розгортання як самоадаптивних систем.

Контейнери та мікросервіси – це доволі легкі підходи пристосування до особливостей архітектури та розгортання програмного забезпечення, особливо актуального в умовах хмарних середовищ, які мають на меті забезпечити оптимальні грошові витрати на утримання інфраструктури.

Зазначається, що повністю адаптовані до контейнеризації мікросервіси для, наприклад, платформ потокового передавання даних можуть бути розгорнуті навіть на невеликих, наприклад, одноплатних кластерах пристроїв. Вони можуть обслуговувати випадки використання IoT, для яких обсяг даних зазвичай є не надто великий. Переваги включають гнучке розгортання контейнера та (принаймні) достатню продуктивність та масштабованість, незважаючи на обмеження пристрою. Проблеми все ще існують у динамічному керованому навантаженням розподілі контейнерів у кластерах та проблеми адаптації, які вже зазначалися.

### 3.1.5 Контейнеризація за допомогою Docker

Віртуалізація є невід’ємною й досить фундаментальною технологією для створення сучасних ЦОД та, особливо, хмарних ЦОД. Нещодавно отримала широке застосування така технологія, як контейнеризація, яка також стає все більш популярною та інтегрованою в перелік продуктів хмарних провайдерів.

Насправді, щільна інтеграція контейнерів в основу ядра ОС дозволяє зменшити накладні витрати на програмне забезпечення, що є необхідним для забезпечення функціонування віртуальних машин.

Однак, така щільна інтеграція призводить до збільшення периметру для потенційних атак, викликаючи занепокоєння щодо безпеки. Вже існуючі механізми забезпечення безпеки контейнерів фокусуються головним чином на встановленні безпечного взаємозв'язку між хостовими системами і контейнерами. Зокрема, системи контейнеризації мають вбудовані автоматизовані логічні ланцюги розгортання, призначені для прискорення процесів розгортання коду. Ці ланцюжки часто складаються із сторонніх елементів, що працюють на різних платформах від різних постачальників, що викликає занепокоєння щодо цілісності та безпечності кодової бази. Це може спричинити проникнення в систему численних уразливостей, які можуть бути використані зловмисниками.

Docker – це програмна платформа, яка дозволяє швидко створювати, тестувати та розгортати програми. Docker надає змогу запакувати програмне забезпечення в стандартизовані компоненти, які носять назву контейнери, що містять усе необхідне для запуску програмне забезпечення, включаючи бібліотеки, системні інструменти, код та робоче середовище. За допомогою Docker можна швидко розгорнути та масштабувати програми в майже будь-якому середовищі. У порівнянні зі звичними віртуальними машинами, Docker є більш легковісним рішенням для розподілення обчислювальних ресурсів, аніж традиційні віртуальні машини з віртуалізаторами різних типів (Xen, KVM та інші) – схематично це зображено на рисунку 3.3 [10].

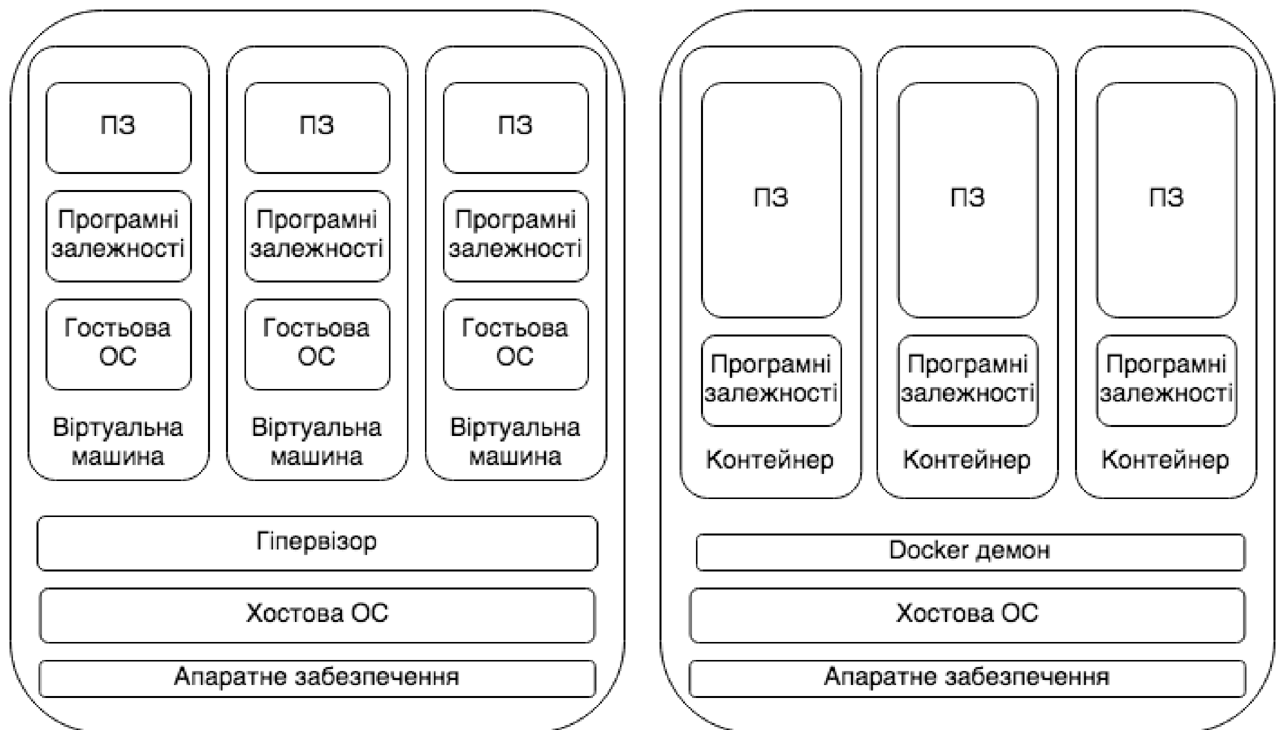


Рисунок 3.3 – Порівняння різноманітних типів віртуалізації та контейнеризації

Проте, Docker – не лише інструмент для створення та запуску контейнерів. Це ціла низка компонент, які разом утворюють екосистему навколо даного програмного продукту.

Даний програмний засіб складається з наступних компонентів:

– Docker-демон – сервер обробки запитів від контейнерів. Демон контролює об'єкти екосистеми та може комунікувати з іншими демонами для контролю власних компонент;

– Docker-клієнт – інтерфейс взаємодії користувача з Docker-демоном. Клієнт може комунікувати з декількома демонами, що, наприклад, можуть бути встановлені на декількох різних машинах;

– Образ – файл, що включає в себе перелік програмних залежностей для запуску певного ПЗ, дані про те, як саме має бути запущене ПЗ, конфігурацію для подальшого розгортання ПЗ та ініціалізації контейнера;

– Контейнер – пакет програмного забезпечення, що був створений за директивами певного образу. Він включає в себе всі необхідні залежності, код ПЗ та інші компоненти, що можуть бути необхідними для запуску ПЗ. Життєвий цикл контейнеризованого додатку зображений на рисунку 3.4;

– Том – емуляція файлової системи для виконання операцій зчитування та запуску у ході виконання ПЗ. Томи створюються автоматично у момент запуску нового контейнеру, проте можуть й бути створені окремо та примонтовані до контейнерів задля забезпечення зберігання даних між перезапусками контейнерів;

– Реєстр – сервер, що використовується для зберігання образів контейнерів. Він є необхідним, адже зазвичай образи є результатом успішного виконання тестів програмного коду та цілої низки автоматичних перевірок. Такі перевірки виконуються на віддалених хостах, що тісно пов’язані с системи контролю версій (Bitbucket, GitLab, GitHub тощо).

Docker успішно став лідером на ринку контейнерів та асоційованої екосистеми DevOps. Зокрема, 92% опитаних людей від ClusterHQ та DevOps.com [10] використовують або планують використовувати Docker у контейнерному середовищі. Безпека є надзвичайно важливим аспектом при аналізі технологій будь-якого спектру. Саме через це команда розробників Docker вже великий час працює над дослідженнями, які дозволяють проводити експерименти та вивчати практичну можливість проведення деяких атак.

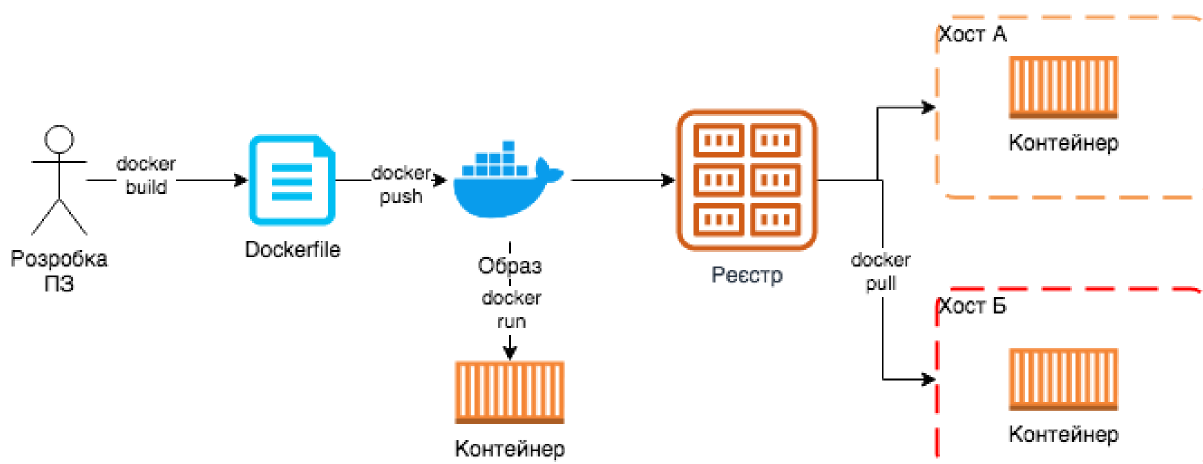

Рисунок 3.4 – Життєвий цикл контейнеризованого додатку

Отже, до переваг даного програмного рішення можна віднести наступне:

– мінімальне використання ресурсів – контейнери не віртуалізують всю гостьову ОС, а використовують ядро хостової системи і ізолюють програму на рівні процесу. Такі процеси потребують набагато менше ресурсів серверу, аніж віртуальна машина;

– швидкісне розгортання – допоміжні компоненти (БД, системи обробки черг повідомлень та інше) не потребують окремого встановлення, а можуть бути з легкістю завантажені з публічних безкоштовних реєстрів. Більш того, не потрібно щоразу завантажувати один й той самий образ декілька разів – файлова система Docker розпізнає шари, з яких складаються образи за їх хеш-сумами та дозволяє перевикористовувати шари з однаковими хеш-сумами;

– зручне приховування процесів – для кожного контейнеру можна використовувати різні методи обробки даних, приховуючи процеси, що виконуються у фоні;

– просте масштабування – контейнери досить легко горизонтально та вертикально масштабуються. Таким чином, можна легко адаптуватися до різко зростаючих навантажень;

– зручний запуск – контейнери можуть бути з легкістю запущені на будь-яких хостах, що мають встановлений Docker демон;

– оптимізація файлової системи – Docker використовує так звану багатошарову файлову систему. Її особливістю є те, що зміни до файлів зберігаються шарами. Такий підхід дозволяє з легкістю використовувати одні й ті самі шари багато разів (наприклад, образ операційної системи може бути перевикористаним для запуску декількох контейнерів з різними додатками, що включають в себе різні списки залежностей, різний програмний код та інше).

## 3.2 Оркестратори контейнерів

Основною задачею контейнерних оркестраторів є розміщення контейнерів на кластерах з машин та слідкування за їх станом, зібрання метрик тощо. Це означає, що оркестратори забезпечують те, що всі контейнери, які виконують різні робочі навантаження, можуть бути запущені на фізичних або віртуальних машинах.

Контейнери мають бути розподілені ефективно, дотримуючись обмежень середовища розгортання та конфігурації кластера. Крім того, оркестратор повинен стежити за всіма запущеними контейнерами та замінювати неактивні (ті, що за певних причин вийшли з ладу та не можуть більше виконувати свої функції), не реагуючі на системні запити або не відповідаючі на запити систем перевірки діяльності контейнери. Всі ці функції та ще багато інших сучасні популярні оркестратори мають змогу виконувати. Окрім того, оркестратори мають змогу розширювати власний набір функцій за допомогою плагінів, що можуть бути створені користувачами та встановлені разом зі стандартним інтерфейсом.

Для того, щоб запускати робочі навантаження, потрібно забезпечити робоче оточення. Сюди входять фактичні фізичні машини з певними обчислювальними можливостями (з різною кількістю фізичних ЦП чи ядер), оперативною пам’яттю та накопичувачами даних (HDD чи SSD). Крім того, необхідним також є спільне постійне сховище даних (наприклад, NFS) та підключення всіх цих машин до спільної мережі, щоб вони могли знаходити одне одного та спілкуватися між собою. На цьому етапі вимогами є лише наявність певної кількості машин (віртуальних або ж фізичних), що мають вільні обчислювальні ресурси та можуть спілкуватися одна з одною. Контейнерні оркестратори можуть бути розгорнуті на реальному кластері (справжнє фізичне обладнання) або на кластері віртуальних машин. Таким чином, оркестратор може організувати контейнери, якими він управляє, безпосередньо на фізичних машинах або на віртуальних машинах [11].

Також кластери для контейнерих оркестраторів можуть складатися з поєднання фізичних обчислювальних машин і віртуальних машин, але це є досить

розповсюдженим підходом. Користувач не отримує багато додаткових витрат при розгортанні великої кількості мікросервісів у контейнерах, як це зазвичай виявляється з віртуальними машинами. Це робить контейнери ідеальними для хмарного розгортання, де виділення цілої віртуальної машини для кожного мікросервісу було б непомітним.

Усі основні хмарні провайдери, такі як AWS, GCP та Azure, надають послуги хостингу контейнерів в наші дні. Деякі з них, такі як GKE від Google, базуються на Kubernetes. Інші, такі як служба контейнерів від Microsoft Azure, базуються на інших рішеннях (Apache Mesos). До речі, AWS має власну систему – ECS (служба розгортання контейнерів через EC2), яка використовує власне рішення для оркестрації. Ще одним досить популярним та потужним оркестратором є Nomad від корпорації Hashicorp. Дана корпорація є виробником великої кількості ПЗ, що тісно пов’язане з хмарними технологіями [11]. Більш детальне порівняння контейнерних оркестраторів наведене у таблиці 3.1.

Таблиця 3.1 – Порівняння популярних контейнерних оркестраторів.

| | Kubernetes | Nomad | Apache Mesos |
|---|---|---|---|
| Складність у використанні | ++ | ++ | +++ |
| Масштабування | ++ | ++ | +++ |
| Мінімальний розмір | 1 мастер + 1 ВМ | 1 сервер | 1 мастер + 1 ВМ |
| Швидкість розгортання | +++ | + | ++ |
| Синтаксис конфігурації | YAML | YAML | JSON |
| Підтримка хмарними провайдерами | AWS, GCP, Azure | AWS, GCP, Azure | AWS, GCP, Azure |
| Відкритий код | Так | Ні | Так |

Важлива відмінність Kubernetes в тому, що його можна і розгорнути у будь-якому середовищі, і придбати як сервіс у більшості хмарних провайдерів. Kubernetes

має інтерфейс для хмарного провайдера, який дозволяє будь-якому хмарному провайдеру інсталювати його та легко інтегрувати у екосистему.

Окрім того, при виборі такого ключового компоненту як оркестратор контейнерів, слід зважати увагу й на те, які з лідерів індустрії використовують технологію. Переважна більшість оркестраторів мають відкритий код і це є їх важливою особливістю, адже це спонукає спільноту користувачів до створення все більшої кількості плагінів, що згодом можуть стати частиною самого оркестратору. Велика кількість спеціалістів мріє співпрацювати з компаніями-лідерами індустрії – FAANG і в ході власного розвитку можуть звертати увагу саме на той набір інструментів, який використовується у конкретній компанії. Переглянути, які з оркестраторів є у використанні у певних компаніях (TODO за даними Stackshare) можна на рисунку 3.5.

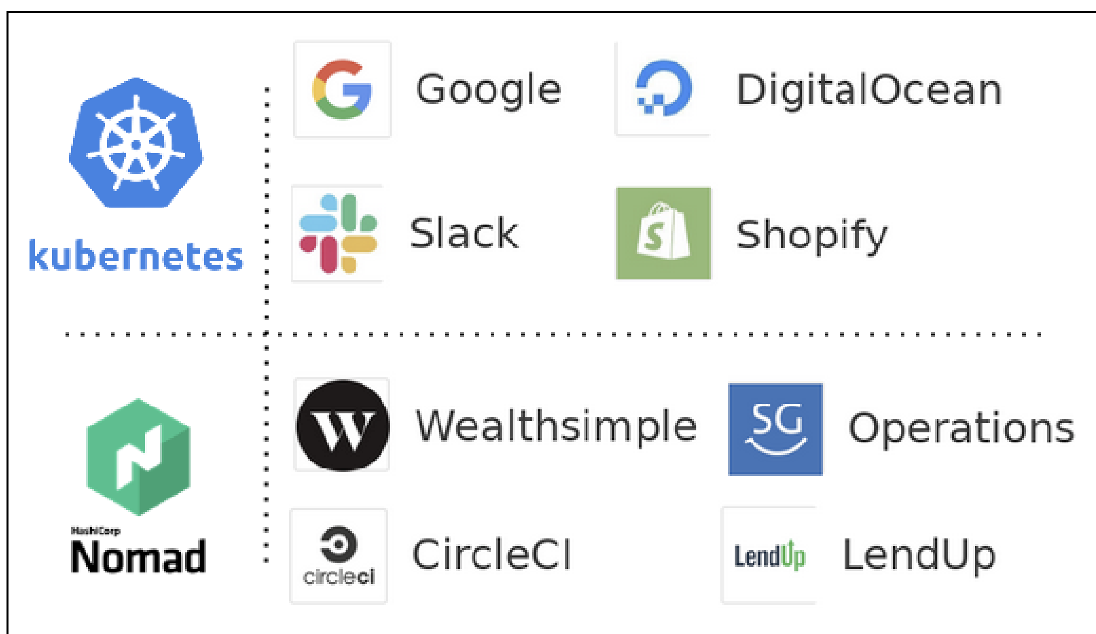


Рисунок 3.5 – Розповсюдженість оркестраторів Nomad та Kubernetes за даними Stackshare

Важлива відмінність Kubernetes в тому, що його можна і розгорнути у будь-якому середовищі, і придбати як сервіс у більшості хмарних провайдерів. Kubernetes має інтерфейс для хмарного провайдера, який дозволяє будь-якому хмарному

провайдеру інсталювати його та легко інтегрувати у екосистему, про що свідчить статистика з рисунку 3.6 [11]. Статистичні дані беззаперечно вказують на широке використання Kubernetes, що пояснюється також й такими причинами як простота у використанні та можливості для інтеграції у вже існуючі інфраструктури. Також, інсталяціія Kubernetes можлива на обчилювальні машини з різноманітними архітектурами процесорів, наприклад, ARM та x86, що робить можливим повторне використання вже навяних серверів у хмарниих ЦОД або ж розміщених у приватних датацентрах.

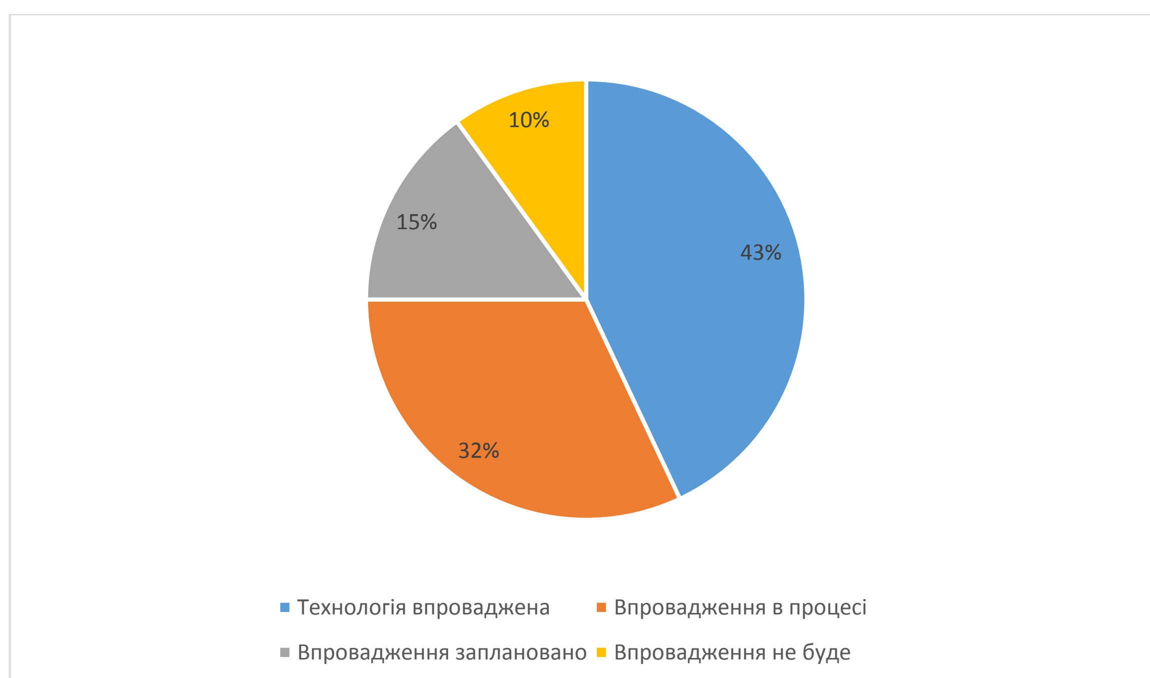


Рисунок 3.6 – Статистика впровадження технології Docker

Окрім того, при виборі такого ключового компоненту як оркестратор контейнерів, слід зважати увагу й на те, які з лідерів індустрії використовують технологію. Переважна більшість оркестраторів мають відкритий код і це є їх важливою особливістю, адже це спонукає спільноту користувачів до створення все більшої кількості плагінів, що згодом можуть стати частиною самого оркестратору.

Оркестратор Kubernetes був створений у стінах компанії Google для задоволення власних потреб з роміщення додатків. Згодом, дане рішення було адаптовано для розгортання також систем, котрими користуються й кінцеві

користувачі компанії (а це сотні, а то й тисячі продуктів по всьому світу). Саме після того, як даний програмний застосунок якнайкраще продемонстрував себе у межах власних продуктів компанії, частина програмного коду була викладена у відкритий доступ під ім'ям «Kubernetes».

Нині Kubernetes використовують тисячі компаній різного масштабу у всіх індустріях. Нові версії додатку виходять щотижнево, що свідчить про стрімкий розвиток технології та надзвичайно широке використання.

Крім того, Kubernetes представлений як PaaS рішення від багатьох вендорів хмарних обчислень. Серед них:

– AWS;
– GCP;
– Azure;
– DIgitalOcean;
– IBM Cloud.

За даними агенції «DZone» саме оркестратором Kubernetes користуються понад 40% команд (рисунок 3.7), що вже впровадили і використовують або ж нині впроваджують технологію Docker [12].

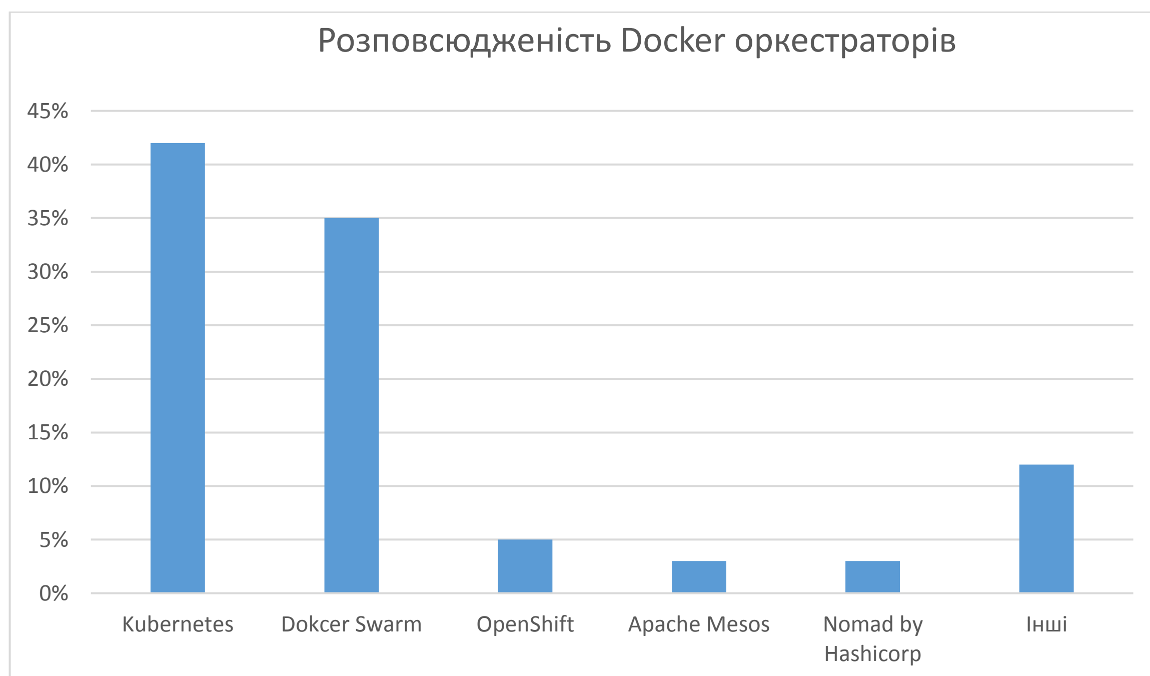


Рисунок 3.7 – Розповсюдженість Docker оркестраторів

Це також підтверджено організацією CNCF – «Cloud Native Computing Foundation». Вона є неофіційним регулятором (офіційного не існує), що встановлює, які з технологій вважаються повністю стандартизованими та адаптованими до використання у хмарному середовищі. Також, CNCF є частиною неприбуткової організації Linux Foundation, що займається підтримкою, розвитком та стандартизації компонент середовища Linux.

Отже, беручи до уваги розповсюдженість технології Kubernetes, динаміку та потенціал її розвитку, відкритий програмний код, та можливості щодо масштабування, можна стверджувати, що вона є беззаперечним лідером серед рішень для оркестрації Docker контейнерів.

Через описані вище особливості та переваги Kubernetes, система автоматизації, розроблювана в даній магістерській дисертації, розробляється саме для взаємодії з оркестратором Kubernetes.

### 3.2.1 Огляд особливостей використання Kubernetes

Зазвичай, коли системи складалися з меншої кількості сервісів та серверів, кожен сервер мав своє статичне ім'я та конфігурацію. Розробники та користувачі точно знали, яке програмне забезпечення працює на кожній машині. У багатьох компаніях, проводилися багатоденні обговорення для обрання імен для парку серверів.

Коли сервер переставав працювати, могла розпочатися велика криза. Усі намагалися з'ясувати, де взяти інший сервер, що навіть було запущено на сервері, що вийшов з ладу, та як змусити все це працювати на новому сервері. Якщо сервер зберігав деякі важливі дані, то, всі сподівались, що була актуальна резервна копія, і, можливо, можна було навіть її відновити.

Очевидно, що такий підхід не масштабується. Коли є кілька десятків чи сотень серверів, слід почати поводитися з ними як з чимось динамічним. Слід зважати на загальний стан системи, а не на параметри кожного окремого серверу чи сервісу.

Kubernetes несе повну відповідальність за розподіл контейнерів на певних машинах. Не потрібно взаємодіяти з окремими вузлами більшу частину часу. Це найкраще підходить для навантажень, що не вимагають збереження поточного стану (від англ. «stateless»), на відміну від баз даних, наприклад.

Що стосується додатків зі станом (від англ. «stateful»), ситуація трохи інша, але Kubernetes пропонує рішення під назвою StatefulSet.

Kubernetes спрямований на управління та спрощення організації, розгортання та підтримки розподілених систем в широкому діапазоні середовищ та хмарних провайдерів. Він надає велику кількість можливостей, які повинні працювати в усьому цьому різноманітті, розвиваючись і залишаючись досить простими для використання звичайними розробниками. Це досить складна мета. Kubernetes досягає цього, дотримуючись безперечно чіткого дизайну високого рівня та продуманої архітектури, що сприяє розширенню та розвитку. Багато частин Kubernetes все ще мають досить жорсткі обмеження або є специфічні для компанії розробника, але тенденція полягає в переробці їх на плагіни та збереженні основної загальної та абстрактної частини [12].

### 3.2.2 Огляд основних компонентів Kubernetes

Серед основних компонентів Kubernetes можна назвати такі як, власне, кластер, etcd, менеджер контролера, планувальник та вузли.

Кластер – це сукупність обчислювальних ресурсів, ресурсів для зберігання даних та мережевих ресурсів, які Kubernetes використовує для запуску різних робочих навантажень, з яких складаються системи. Слід також зауважити, що кластери можуть об’єднуватися, утворюючи федерації, що можуть поєднувати у собі ДЦ, розташовані у різних географічних зонах, або ж різні хмари від декількох провайдерів хмарних послуг.

Головним компонентом управління кластером є панель управління Kubernetes. Вона складається з декількох компонентів, таких як сервер API, планувальник задач та менеджер контролера. Планувальник задач відповідає за глобальне планування

об’єктів на рівні кластера та обробку подій. Зазвичай всі основні компоненти встановлюються на одному хості. Розглядаючи сценарії для забезпечення високої доступності або дуже великі кластери, можливе й розділення даних компонент по різним хостам, або ж реплікація перелічених компонент на декількох хостах.

У кластері Kubernetes є кілька основних компонентів, що використовуються для управління кластером, а також компоненти вузлів, які працюють на кожному вузлі кластера.

Основні компоненти, як правило, працюють на одному вузлі, але у високодоступному або дуже великому кластері вони можуть бути розподілені по декількох вузлах. Діаграма взаємодії компонентів наведена на діаграмі 1.

Сервер API може легко масштабуватися по горизонталі, оскільки не має чіткого поточного стану («stateless») і зберігає всі дані в кластері etcd. Сервер API – це реалізація панелі управління Kubernetes.

Etcd – це високонадійне розподілене сховище даних. Kubernetes використовує його для зберігання стану всього кластера. У кластері невеликих розмірів (десятки вузлів) або ж у такому, що не потребує забезпечення високої надійності один екземпляр etcd може працювати на одному вузлі з усіма іншими основними компонентами. Але для більш значних кластерів досить типово мати 3-вузловий або навіть 5-вузловий etcd-кластер для надмірності та високої доступності [13].

Менеджер контролера – це сукупність різних менеджерів, згорнутих в один програмний компонент. Він містить контролер реплік, контролер контейнерів, контролер сервісів, контролер точок входу та інші. Усі ці менеджери стежать за станом кластера через API, і їх завдання полягає в тому, щоб направити забезпечити знаходження кластеру у бажаному стані [6].

Планувальник відповідає за планування контейнерів у вузлах. Це дуже складне завдання, оскільки воно повинно враховувати безліч взаємодіючих факторів, таких як:

- наявність обмежень щодо розгортання;
- наявність вільних обчислювальних ресурсів;

– доступність вузла;
– оптимальність вибору вузла;
– інше.

Планувальник розгорнутий як звичайний контейнер. Кожен сервіс у кластері (крім так званих «headless») отримує відповідний унікальний запис у DNS. Стручки також можуть отримувати ім'я DNS. Це досить зручно створення систем з автоматичним виявленням компонентів. Наприклад, кластери Elasticsearch можуть бути розгорнуті таким чином досить швидко.

Вузол – це один хост. Це може бути фізична або ж віртуальна машина, наприклад як у хмарі GCP. Конфігурація такої віртуальної машини зображена на рисунку 3.8. Його робота – керувати контейнерами. Кожен вузол Kubernetes запускає кілька компонентів Kubernetes, таких як kubelet та проксі-сервер kube. Вузлами керує мастер Kubernetes. Саме на вузлах Kubernetes й розміщуються робочі навантаження Kubernetes, котрі розробники розгортають. На компонентах інших типів розгортання користувацьких додатків не є можливим.

Системні компоненти Kubernetes займають певну кількість ресурсів на кожному хості. Цей об’єм ресурсів може змінюватися власноруч при мануальному розгортанні Kubernetes та є фіксованим для віртуальних машин кожного розміру на усіх платформах хмарних провайдерів. Таким чином, для віртуальних машин з 2-ядерними процесорами та 8 гігабайтами пам’яті у хмарах від GCP (рисунок 3.9), AWS (рисунок 3.10), та Azure (рисунок 3.11), виробники виділяють таку кількість ресурсів для системних потреб [14].

Вузлам кластера необхідною є пара компонентів для взаємодії з основними компонентами кластера, наприклад, для отримання робочих навантажень для виконання та оновлення кластеру щодо їх стану тощо. Судячи з виділення ресурсів на окремі компоненти можна зробити висновок, що оптимальним з точки зору користувачів є розподіл від AWS.

```yaml
- apiVersion: v1
  kind: Node
  metadata:
    annotations:
      container.googleapis.com/instance_id: "12323432461"
      node.alpha.kubernetes.io/ttl: "0"
      node.gke.io/last-applied-node-labels: cloud.google.com/
      gke-netd-ready=true,cloud.google.com/gke-nodepool=gke-pool-2-prod-2,cloud.google.com/
      gke-os-distribution=cos,environment=prod-2,iam.gke.io/
      gke-metadata-server-enabled=true,service=gke
      volumes.kubernetes.io/controller-managed-attach-detach: "true"
    creationTimestamp: "2020-12-03T04:17:21Z"
    labels:
      cloud.google.com/gke-netd-ready: "true"
      cloud.google.com/gke-nodepool: gke-pool-2-prod-2
      cloud.google.com/gke-os-distribution: cos
      kubernetes.io/arch: amd64
      kubernetes.io/hostname: gke-gke-pool-2-prod-2-5d8521c7-pmzc
      kubernetes.io/os: linux
      service: gke
    name: gke-gke-pool-2-prod-2-5d8521c7-pmzc
    resourceVersion: "313488842"
    selfLink: /api/v1/nodes/gke-gke-pool-2-prod-2-5d8521c7-pmzc
    uid: c1685007-61bd-4784-bf4b-0b74d33b0ea4
  spec:
    podCIDR: 10.32.13.0/24
    podCIDRs:
    - 10.32.13.0/24
    providerID: gce://test/europe-west1-d/gke-gke-pool-2-prod-2-5ds5z1c7-pmzc
  status:
    addresses:
    - address: 10.132.0.33
      type: InternalIP
    - address: ""
      type: ExternalIP
    - address: gke-gke-pool-2-prod-2-5d8521c7-pmzc.c.test.internal
      type: InternalDNS
    - address: gke-gke-pool-2-prod-2-5d8521c7-pmzc.c.test.internal
      type: Hostname
    allocatable:
      attachable-volumes-gce-pd: "127"
      cpu: 15890m
      ephemeral-storage: "47093746742"
      hugepages-2Mi: "0"
      memory: 56287912Ki
      pods: "110"
    capacity:
      attachable-volumes-gce-pd: "127"
      cpu: "16"
      ephemeral-storage: 98868448Ki
      hugepages-2Mi: "0"
      memory: 61844136Ki
```


Рисунок 3.8 – Типова конфігурація вузлів у GCP

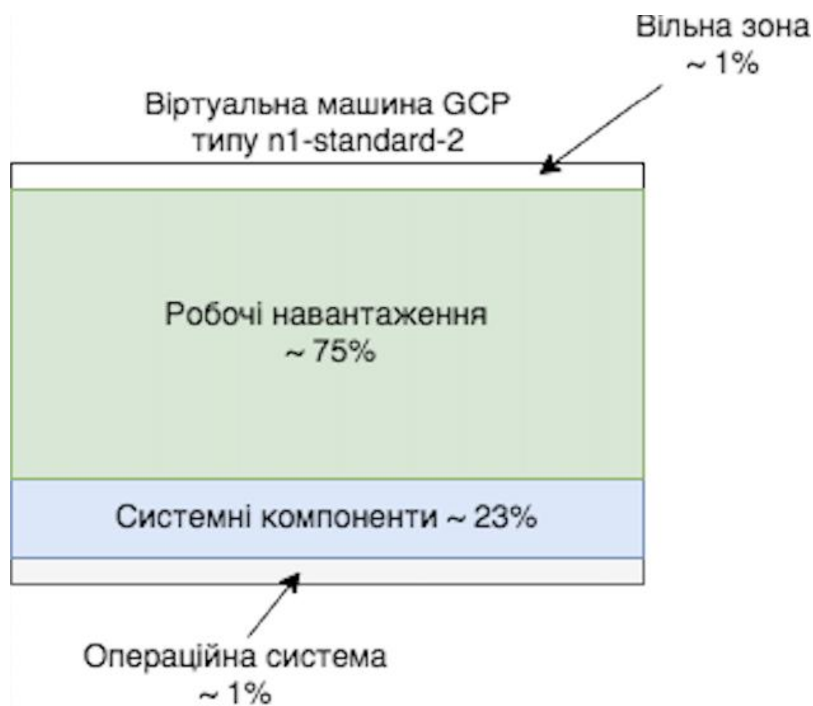


Рисунок 3.9 – Розподіл ресурсів на вузлах Kubernetes у GCP

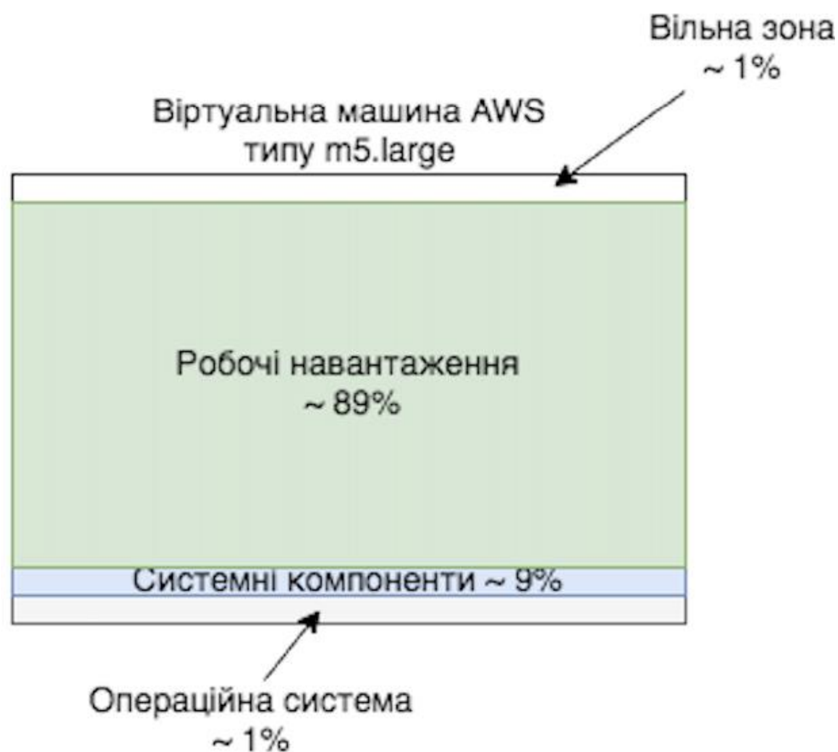


Рисунок 3.10 – Розподіл ресурсів на вузлах Kubernetes у AWS

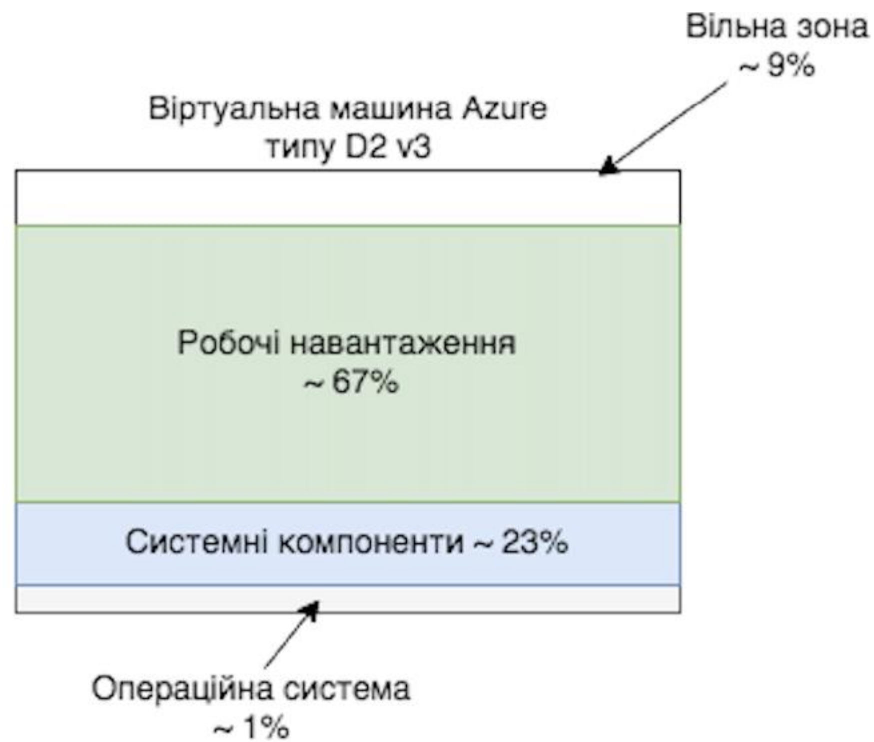


Рисунок 3.11 – Розподіл ресурсів на вузлах Kubernetes у Azure

Проте, як свідчать деякі тести, інколи системні компоненти Kubernetes мають досить нестабільну поведінку в умовах виділення малої кількості ресурсів та великої кількості досить малих контейнерів, що є розміщеними на 1 хості. Беручи до уваги цей факт, слід зауважити, що AWS також виділяє найменшу кількість ресурсів для системних потреб, тому такий розподіл хоч і є найоптимальнішим з точки зору користувачів, та може призвести до нестабільної роботи під навантаженнями. Тому, слід вважати пропорції розподілу ресурсів від Google оптимальними, що робить PaaS рішення GKE від Google найкращим серед конкурентів. Порівняння розподілу ресурсів на різних платформах можна побачити на рисунку 3.12.

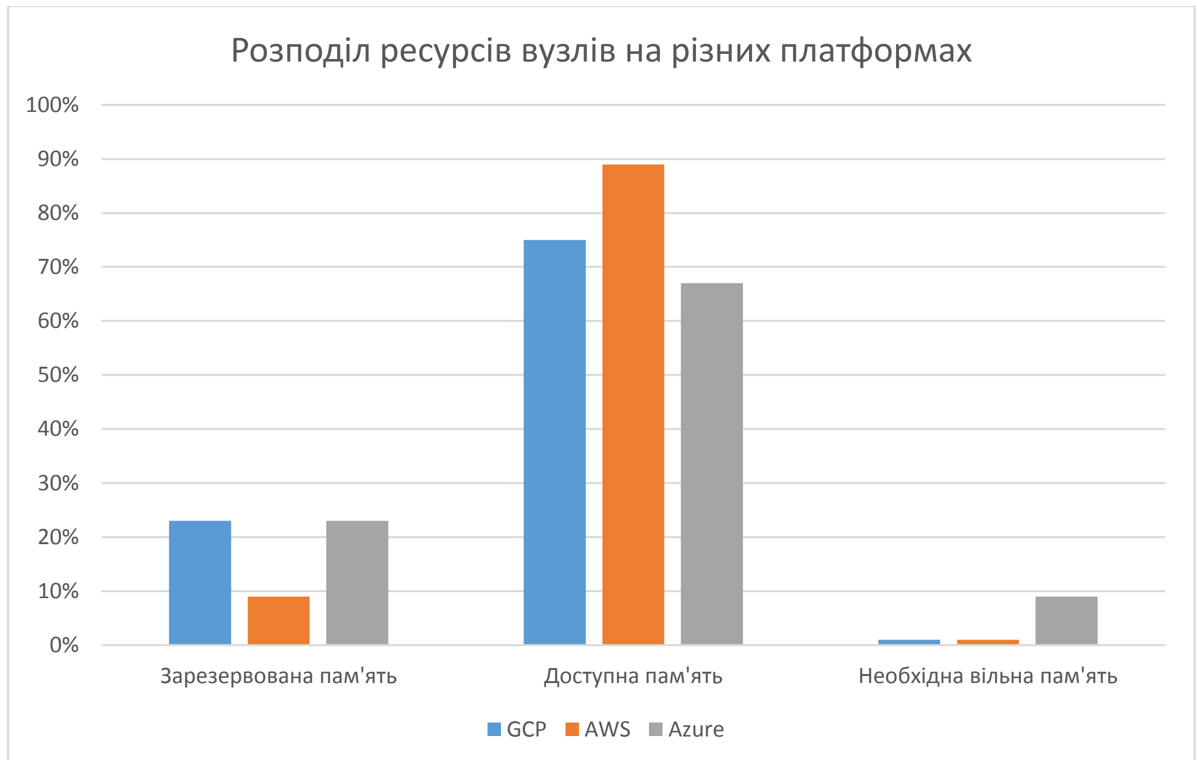


Рисунок 3.12 – Порівняння розподілу ресурсів на вузлах Kubernetes у різних хмарах

Kube-проксі виконує низькорівневі системні операції на кожному вузлі. Він відображає служби Kubernetes локально і може здійснювати переадресацію протоколів передачі даним TCP та UDP. Він знаходить IP-адреси кластера за допомогою змінних середовища або DNS [15].

Наступним компонентом вузла є kubelet, що контролює взаємодію з головними компонентами та керує запущеними додатками. Це включає в себе наступне:

– завантаження секретів додатків із сервера API;
– монтування зовнішніх томів для зберігання даних;
– запуск контейнера додатку (Docker або Rkt);
– повідомлення про стан вузла та кожного додатку;
– запуск перевірок коректної роботи контейнера.

Окрім того, в основі Kubernetes лежать декілька основних архітектурних концепцій запуску контейнерів, зокрема:

- допоміжний контейнер (sidecar);
- амбасадор контейнер;
- адаптер контейнер;
- мультиконтейнерна концепція.

Архітектура допоміжного контейнеру базується на основі розміщення ще одного контейнера у додаток до основного контейнера програми. Основний контейнер додатку нічого не знає про допоміжний контейнер і просто продовжує виконувати свої безпосередні функції. Чудовим прикладом є агент для збору лог-повідомлень. Основний контейнер може просто писати в потік виводу stdout чи stderr, а допоміжний контейнер може перехоплювати вивід та відправляти дані до центрального сховища зберігання логів (наприклад, як у разі з EFK/ELK стеками).

Переваги використання допоміжних контейнерів порівняно з додаванням системи централізованого збору логів до основного контейнера додатків величезні.

По-перше, додатки більше не обтяжені централізованими системами та масивною логікою збору логів, що може спричиняти проблеми. Якщо виникає потреба оновити або змінити політику логування або перейти на абсолютно нове рішення, просто потрібно оновити допоміжний контейнер та розгорнути його. Жоден з основних контейнерів додатку не змінюється, тому унеможливлюються випадкові виходи з ладу основної системи.

Контейнер амбасадор – це можливість представлення віддаленої служби, як якщо б вона була локальною, у поєднанні з застосуванням певної політики. Хорошим прикладом такого підходу є ситуація, коли існує кластер Redis з одним мастером для записів і багатьма репліками для читання. Локальний контейнер може служити проксі-сервером і представляти Redis для основних контейнерів додатку на localhost інтерфейсі. Основний контейнер додатків просто підключається до Redis на localhost: 6379 (порт Redis за замовчуванням), але в такому разі він підключається до амбасадору, що працює в тому самому кластері, який фільтрує запити, і надсилає запити на запит справжньому мастеру Redis і читає запити випадковим чином з одного з виділених реплік для читання. Так само, як і з допоміжним контейнером,

основний додаток має змоги зрозуміти, що саме відбувається, що додає шар абстракції в разі необхідності. Такий підхід може допомогти при тестуванні з використанням справжнього локального кластеру Redis. Крім того, якщо конфігурація кластера Redis змінюється, потрібно змінити лише конфігурацію амбасадору, а основна програма залишається повністю без змін [15].

Контейнер адаптер зазвичай використовується для стандартизації виводу з основного контейнера програми. Розглянемо випадок служби, яка розгортається поступово: вона може генерувати відповіді у форматі, який не відповідає попередній версії. Інші служби та додатки, які обробляють ці відповіді, ще не модернізовані. Контейнер адаптера можна розгорнути в одному модулі з новим контейнером додатку і форматувати їх вивід таким чином, щоб відповідати старій версії, поки всі сусідні сервіси не будуть оновлені. Контейнер адаптера спільно використовує файлову систему з основним контейнером програми, тому він може спостерігати за локальною файловою системою, і коли новий додаток щось пише, він негайно адаптує вивід.

Мультиконтейнерна концепція є досить широко використовуваною. Особливо це часто використовується у системах, що мають на меті збереження поточного стану даних, наприклад, кластеризовані бази даних, наприклад:

– Cassandra;
– Elasticsearch;
– Redis;
– Hadoop;
– Kafka;
– MySQL;
– MongoDB;
– Apache HBase.

Концепції одноконтейнерих додатків підтримуються безпосередньо Kubernetes через розгортання. Багатоконтейнерні підходи з такими операціями як вибори лідера, черги на роботу та розкидання-збір повідомлень, не підтримуються безпосередньо, але можуть бути розгорнуті з використанням стандартних компонентів Kubernetes або

ж за допомогою так званих операторів. Концепт операторів буде розглянутий у наступних розділах, адже він є досить важливим компонентом для всієї екосистеми Kubernetes.

### 3.2.3 Аналіз типів оточень запуску контейнерів

Kubernetes спочатку підтримував лише Docker як механізм запуску виконання контейнерів. Але це вже не так. Rkt – ще один підтримуваний механізм виконання, і є цікаві спроби роботи з контейнерами Hyper.sh через Hypernetes. Основною політикою проектування є те, що сам Kubernetes повинен бути повністю відокремлений від конкретних режимів роботи.

Взаємодія між Kubernetes та середовищем виконання відбувається за допомогою доволі узагальненого інтерфейсу, який повинні реалізовувати середовища виконання. Більша частина спілкування використовується за допомогою концепцій pod, контейнера та операцій, які можна виконати над контейнером. Кожен механізм виконання відповідає за реалізацію інтерфейсу виконання Kubernetes, щоб бути сумісним з оркестратором.

Інтерфейс виконання для контейнерів вказаний у проекті Kubernetes на GitHub. Kubernetes має відкритий програмний код, тому можна переглянути опис інтерфейсу, що відповідає за взаємодію з оточенням запуску контейнеру і побачити, що набір функцій є досить тривіальним без чогось специфічного.

Це означає, що при реалізації інтерфейсу виконання, також повинні реалізовуватися методи перелічених інтерфейсів. Інтерфейси визначені в одному файлі. Просто назви інтерфейсів надають багато інформації про те, що вони роблять. Очевидно, що Kubernetes потрібно запускати команди в контейнерах, і йому потрібно приєднувати контейнери до своїх додатків і витягувати образи контейнерів.

Docker – це, звичайно, де-факто стандарт в світі контейнеризації. Kubernetes спочатку був розроблений для управління лише контейнерами Docker. Можливість виконання у декількох оточеннях була вперше представлена у версії Kubernetes 1.3. До того часу Kubernetes міг керувати лише контейнерами Docker.

Docker користується величезною популярністю та зростанням, але також є багато критики щодо нього. Критики часто згадують такі занепокоєння [16]:

– безпека;
– труднощі з налаштуванням багатоконтейнерних програм (зокрема, мережеві);
– розробка, моніторинг та ведення лог-журналів;
– обмеження контейнерів Docker, що мають виконувати лише однин процес.

У Docker було вирішено деякі з цих проблем. Зокрема, було створено новий оркестратор. Docker Swarm – це рішення для оркестрації, яке використовується для Docker і конкурує з Kubernetes. Він простіший у використанні, ніж Kubernetes, але він не такий потужний та зрілий. Починаючи з Docker 1.12, режим Swarm включений в Docker Daemon, що створює ряд недоліків, зокрема: збільшення розміру коду Docker демону та потенційні проблеми. Це, у свою чергу, змусило більше людей звернутися до CoreOS RKT як до альтернативного рішення.

Починаючи з Docker 1.11, випущеного у квітні 2016 року, у Docker змінено спосіб керування контейнерами. Середовище виконання використовує containerd та runC для запуску зображень Open Container Initiative (OCI) у контейнерах. Дана архітектура зображена на рисунку 3.13.

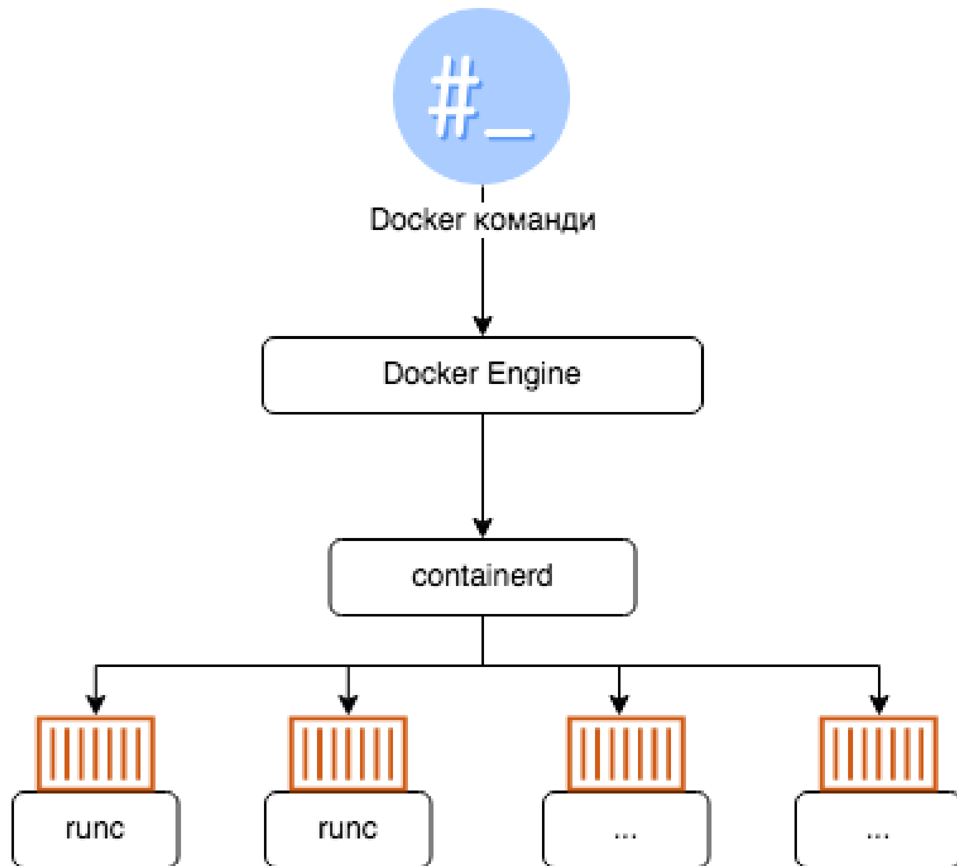


Рисунок 3.13 – Архітектура Docker

Таким чином, OCI забезпечує підтримку декількох оточень для контейнерів, що задовольняє ще більше потреб кінцевих користувачів, що могли б розглядати rkt як альтернативу для міграції та дає змогу користувачам продовжувати використовувати Docker ще довгий час.

### 3.2.4 Поди у Kubernetes

Под – це найменша одиниця розгортання робочого навантаження в Kubernetes. Кожен под містить один або кілька контейнерів – це можуть бути як допоміжні контейнери, так і додатки, що складаються з декількох контейнерів. Усі контейнери в поді мають однакову IP-адресу та простір портів; вони можуть спілкуватися за допомогою localhost інтерфейсу або обміном інформації через спільну файлову систему.

Крім того, усі контейнери в модулі можуть мати доступ до спільного локального сховища на вузлі, де розміщений под. В такому разі, спільне сховище буде доступним кожному контейнеру. Поди є важливим компонентом Kubernetes. Можна запустити декілька програм всередині одного контейнера Docker, маючи щось на зразок супервізора як основну програму Docker, яка запускає декілька процесів, але на цю практику часто нарікають з наступних причин [17]:

– прозорість: робота контейнерів у рамках поду робить усі контейнери видимими для інфраструктури та дозволяє інфраструктурі надавати послуги цим контейнерам, наприклад, управління процесами та моніторинг ресурсів. Це полегшує вирішення цілої ланки задач для кінцевих користувачів;

– розподіл залежностей програмного забезпечення: Окремі контейнери можуть бути незалежно перезібрані з різними версіями залежностей ПЗ, з різними архітектурами ОС чи навіть наявними чи відсутніми певними компонентами. Можливо, Kubernetes зможе навіть підтримувати оновлення контейнерів у майбутньому;

– простота використання: користувачам не потрібно запускати власні менеджери процесів, турбуватися про обробку системних сигналів та кодів виходу тощо;

– ефективність: оскільки інфраструктурні компоненти беруть на себе більшу частину відповідальності за належне виконання додатків, контейнери з ПЗ можуть бути досить нестійкими (перезавантажуватися, виходити з ладу та інше).

Поди – це чудове рішення для управління групами тісно пов’язаних контейнерів, які залежать один від одного і потребують запуску на одному хості для коректного виконання програмного коду. Важливо пам’ятати, що поди вважаються ефемерними, такими що можуть досить легко зникати та відроджуватися на інших вузлах або ж бути видозміненими. Будь-яке сховище подів знищується разом із самим подом, таким чином за умови збереження файлової системи, слід використовувати рішення для зовнішнього збереження даних. Кожен под отримує унікальний

ідентифікатор (UID), тому вони можуть бути з легкістю розрізненими в разі необхідності, що представлено на рисунку 3.14.

```yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubectl.kubernetes.io/restartedAt: "2020-09-17T16:32:34+03:00"
  creationTimestamp: "2020-11-17T19:40:34Z"
  generateName: serviceA-prod-1231332-4qwqs
  labels:
    app: serviceA
    environment: prod
    pod-template-hash: 1231332
  name: serviceA-prod-1231332-4qwqs
  namespace: serviceA
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: serviceA-prod-1231332
    uid: 82d814d1-2fb8-4195-a830-81c5c6bf3d1c
  resourceVersion: "296613202"
  selfLink: /api/v1/namespaces/serviceA/pods/serviceA-prod-1231332-4qwqs
  uid: 7d4f46c0-1a3b-42e8-a271-5e1bc835dd78
spec:
  containers:
  - env:
    - name: KUBERNETES_NAMESPACE
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.namespace
    - name: KUBERNETES_POD_NAME
      valueFrom:
        fieldRef:
          apiVersion: v1
          fieldPath: metadata.name
    - name: JAVA_OPTS
      value: -Xms1400M -Xmx1400M -Xss1024K
    image: eu.gcr.io/registry/serviceA:v2.2.2
    imagePullPolicy: IfNotPresent
    livenessProbe:
      exec:
        command:
        - /usr/local/bin/grpc-health-probe
        - -addr=:6565
        - -service=liveness
      failureThreshold: 5
      initialDelaySeconds: 90
      periodSeconds: 15
      successThreshold: 1
      timeoutSeconds: 1
    name: serviceA
    ports:
    - containerPort: 6565
      name: service-port
      protocol: TCP
```

Рисунок 3.14 – Типова конфігурація поду

Окрім того, Kubernetes оперує своїми ресурсами (зокрема, подами) за допомогою міток. Мітка – це пара ключ-значення, яка використовуються для групування наборів об'єктів. Це важливо для кількох інших концепцій, таких як контролер реплікації, набори реплік та служби, які працюють на динамічних групах об'єктів і потребують ідентифікації членів групи. Між об'єктами та мітками існує зв'язок NxN. Кожен об’єкт може мати кілька міток, і кожна мітка може застосовуватися до різних об’єктів. Існують певні обмеження щодо дизайну міток. Кожна мітка на об’єкті повинна мати унікальний ключ. Ключ мітки повинен дотримуватися суворого синтаксису. Він має дві частини: префікс та ім’я. Префікс необов’язковий. Якщо він існує, то відокремлюється від імені косою рискою (/), і це повинен бути дійсний піддомен DNS. Префікс повинен містити не більше 253 символів. Назва є обов’язковою та має містити не більше 63 символів. Імена повинні починатися та закінчуватися буквено-цифровим символом (a-z, A-Z, 0-9) і містити лише буквено-цифрові символи, крапки, тире та підкреслення. Значення повинні відповідати тому самому набору обмежень, що і імена. Слід також звернути увагу, що мітки призначені для ідентифікації об’єктів, а не для прикріплення до них довільних метаданих. Для цього призначені анотації.

Анотації дозволяють асоціювати довільні метадані з об’єктами Kubernetes. Kubernetes просто зберігає анотації та робить доступними їх метадані. На відміну від міток, вони не мають жорстких обмежень щодо дозволених символів та обмежень щодо розміру. Зазвичай, досить корисними завжди є такі метадані, як, наприклад, час створення об’єкту або ж його автор, для складних систем, і досить зручно, що Kubernetes визнає цю потребу та надає її як частину стандартного функціоналу, тому не потрібно створювати власний сховище метаданих та зіставляти об'єкт з їх метаданими [18].

## 3.3 Висновки до розділу

У даному розділі були проаналізовані переваги та недоліки мікросервісної архітектури. Були визначені проблеми, котрі такий тип проектування систем вирішує та породжує

Як результат розповсюдження мікросервісного підходу, було визначено, яким чином можна оптимально розгортати програмне забезпечення у робочих оточеннях та виявлено, що найбільш широко використовуваним підходом є використання системи контейнеризації на базі Docker.

Короткий огляд екосистеми Docker виявив, які переваги надає його використання та які проблеми породжує. Зокрема, було розглянуто контейнерні оркестратори. Як показав аналіз, найбільш розповсюдженим оркестратором є продукт компанії Google – Kubernetes.

Короткий огляд архітектурних особливостей Kubernetes показав, що примітивів, якими він оперує за замовчуванням є недостатньо для розгортання комплексних систем. Зокрема, процес ініціалізації додатків досить часто налаштовується невірно.

Розглянуті архітектурні особливості та особливості використання усіх компонент дозволяють зрозуміти спектр існуючих проблем у повному обсязі та сформувати вимоги щодо розроблюваної системи якомога чіткіше.

# 4 РОЗРОБЛЕННЯ СИСТЕМИ ІНІЦІАЛІЗАЦІЇ ТА УПРАВЛІННЯ ВЕБ-ДОДАТКАМИ

Цей розділ описує власне розробку програмного продукту – автоматизованої системи ініціалізації та управління веб-додатками у Kubernetes кластерах. А саме, включає формування вимог до розробки, обґрунтування вибору засобів реалізації та короткий опис роботи.

## 4.1 Аналіз вимог до робочого оточення

Обмеження щодо робочого оточення, в котрому може працювати система, наведені у даному розділі.

Компонент для ініціалізації додатків має бути сумісним усіма актуальними нині підтримуваними (LTS) версіями дистрибутивів ОС сімейства Linux, список яких включає в себе наступні:

– Alpine Linux 3.12 або новіше;
– Debian Buster;
– Ubuntu 18.04 Bionic або новіше;
– CentOS 7 або новіше;
– Red Hat Enterprise Linux (RHEL) 7 або новіше;
– Google Container-Optimised OS 81 або новіше;
– Fedora 31 або новіше;
– Flatcar 2605.8.0 або новіше.

Робоче оточення також має містити ключ доступу (у JSON форматі) до ресурсів GCP для авторизації додатку та доступу до Secret Manager. Також, для коректної роботи додатку має бути встановлена змінна оточення з іменем «GOOGLE_APPLICATION_CREDENTIALS» [19], що має у своєму значення містити повний шлях до JSON ключу, наприклад «/home/users/gcp-sa.json». Даний файл має бути доступний для читання користувачеві від імені котрого запускається утиліта.

Альтернативним способом авторизації може слугувати сервісний акаунт Kubernetes, що має серед своїх міток мітку під назвою «iam.gke.io/gcp-service-account», значення котрої має містити ім’я сервісного акаунту GCP IAM.

Обов’язковими правами доступу, котрі мають бути у вище зазначений ключів є такі:

– roles/iam.workloadIdentityUser;
– roles/secretmanager.viewer;
– roles/secretmanager.secretAccessor з доступом до усіх необхідних для використання ресурсів.

Компонента для управління додатками у свою чергу має бути доступна за допомогою стандартного клієнту для Kubernetes – утиліти командного рядка «kubectl». Підтримувані версії кластеру Kubernetes – 1.16 та новіші. Середовище розгортання самого кластеру не має значення, а саме підтримуються усі можливі нині варіанти розгортання:

– Google Kubernetes Engine;
– AWS Elastic Kubernetes Service;
– Azure Kubernetes Service;
– DigitalOcean Kubernetes;
– Bare Metal Kubernetes;
– Minikube;
– інші.

## 4.2 Аналіз вимог до системи

Визначення вимог щодо розробки програмного продукту є невід’ємною частиною життєвого циклу будь-якого ПЗ. Цей етап є надзвичайно важливим, оскільки саме в ході нього визначаються точний функціонал, котрим володітиме продукт, хоча нерідко зміни до продукту можуть вноситися й у процесі розробки. Інколи такі зміни є досить суттєвими, що може майже повністю змінити сам продукт.

До вимог компоненту для ініціалізації додатків слід віднести переваги розглянутих аналогічних рішень та звернути увагу при розробці на їх недоліки, а саме такі:

– можливість додавати параметри конфігурації як змінні оточення;
– можливість додавати параметри конфігурації як файли;
– належне логування;
– можливість додавати чи видаляти параметри конфігурації без внесення змін до кодової бази додатку та змін до образу контейнера;
– підтримка функціоналу GCP IAM Service Account Binding [20];
– інтеграція з Google Secret Manager;
– не створювати додаткового навантаження на системні компоненти Kubernetes;
– можливість використання як незалежної утиліти, що викликається з командного рядку, так і як допоміжний контейнер;
– простота у впровадженні для вже існуючих додатків та нових.

Втілення вище згаданих вимог зробить розроблюваний продукт беззаперечним лідером серед розглянутих доступних аналогів на ринку.

Компонент для управління додатків має відповідати наступним вимогам:

– надавати зручний інтерфейс для версіонування додатків;
– вдало контролювати залежності між додатками;
– належним чином обробляти системні помилки та виводити відповідні повідомлення;
– взаємодіяти з API Kubernetes без жодного виду проксі та інших компонент;
– простий та зрозумілий інтерфейс.

Вище наведеним вимогам відповідає система, розробка структурної схеми котрої описана у наступному підрозділі.

### 4.3 Розробка структурної схеми системи

Розроблена структурна схема системи наведена у Додатку 1. Вона складається з двох основних підсистем, котрі й управляють веб-додатками, що запускаються у кластері:

- підсистема ініціалізації додатків;
- підсистема управління Custom Resource Definition.

Підсистема ініціалізації додатків складається з контролеру конфігурацій й контролеру запусків додатків. Конфігураційні паратери дані контролери отримують з середовища зберігання даних Google Secret Manager. Адміністратор має змогу контролювати параметри конфігурації у середрвищі зберігання даних за допомогою утиліти командного рядка gcloud.

#### 4.3.1 Простори імен веб-додатків

Кожен з веб додатків функціонує у власному просторі імен. Простір імен – це віртуальний кластер. Можна мати єдиний фізичний кластер, який містить кілька віртуальних кластерів, розділених просторами імен. Типова конфігурація простору імен зображена на рисунку 4.1. Кожен віртуальний кластер повністю ізольований від інших віртуальних кластерів, і вони можуть спілкуватися лише через загальнодоступні інтерфейси (наприклад, веб-сокети). Слід зауважити, що деякі об’єкти, наприклад, вузли та томи для постійного зберігання даних не живуть у просторах імен. Kubernetes може планувати виконання подів з різних просторів імен на одному вузлі (проте, можна закріпити вузли або ж окремі групи вузлів для виконання певних визначених типів робочих навантажень). Подібно до цього, поди з різних просторів імен можуть використовувати одне і те ж постійне сховище даних. При використанні просторів імен слід враховувати мережеві політики та квоти ресурсів, щоб забезпечити належний доступ та розподіл ресурсів фізичного кластера.

```yaml
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","kind":"Namespace","metadata":{"annotations":{},"name":"serviceA"}}
  creationTimestamp: "2019-12-03T12:12:08Z"
  name: serviceA
  resourceVersion: "88299531"
  selfLink: /api/v1/namespaces/serviceA
  uid: 20c29079-15c6-11ea-b22f-4201ac100016
spec:
  finalizers:
  - kubernetes
status:
  phase: Active
```


Рисунок 4.1 – Типова конфігурація простору імен

Також, більшість ресурсів, котрі автогенеруються контрольною панеллю Kubernetes, мають у своєму імені або ж серед певних інших конфігураційних директив зазначеним конкретний простір імен.

Підсистема управління Custom Resource Definiton реалізована з використанням фреймворку Kudo, котрий дозволяє за домопогою власних абстракцій оперувати об’єктами конфігурації Kubernetes, що спрощує написання програмного коду, адже Kudo має більш зручний та гнучкий інтерфейс.

Адміністратор має змогу взаємодіяти з підсистемою управління Custom Resource Definition через kubectl – стандартну утиліту командного рядка для роботи з Kubernetes. Усі команди, що виконуються з використанням kubectl контролюються компонентом контролю доступу, що використовує механізм RBAC, котрий більш детально описаний у наступному підрозділі.

### 4.3.2 Система контролю доступу на основі ролей

Система контролю доступу до ресурсів у Kubernetes може бути реалізована за подомогою RBAC (від англ. «Role Based Access Control» – система контролю доступу на основі ролей). Це забезпечило альтернативний механізм автентифікації вже існуючому, але важкому для управління та розуміння, досить давно використовуючимося контролю доступу на основі атрибутів (ABAC). Даний механізм

є надзвичайно потужним, проте складним у розумінні, що все ще є його досить помітним недоліком. Одним з простих прикладів складності механізму є використання Helm: на той час, звичайні підходи до роботи з даною утилітою просто перестали працювати, адже не вистачало сервісних акаунтів та прав для них. Незважаючи на це, ніхто не може заперечити величезний крок, який був зробленим з впровадженням RBAC, розглядаючи Kubernetes як готову для використання у реальних оточеннях складну компоненту багатьох платформ. Оскільки більшість із користувачів використовують Kubernetes з повними привілеями адміністратора, слід зазначити, що в оточенні, що пов’язане за справжніми користувачами потрібно [21]:

– мати декількох користувачів з різними рівнями доступу, встановлюючи належний механізм автентифікації;

– мати повний контроль над тим, які операції може виконувати кожен користувач або група користувачів;

– мати повний контроль над тим, які операції може виконувати кожен процес усередині кожного програмного модуля;

– обмежити видимість певних ресурсів у важливих просторах імен.

У цьому сенсі RBAC є ключовим елементом для забезпечення всіх цих важливих функцій.

Для того, щоб повністю зрозуміти ідею RBAC, необхідно зрозуміти, що задіяні основні три типи абстракцій:

– об’єкти: сукупність користувачів та процесів, які хочуть отримати доступ до API Kubernetes.

– ресурси: набір об’єктів API Kubernetes, доступних у кластері. Прикладами є поди, розгортання, сервіси, вузли та томи зберігання даних та інші.

– дієслова: набір операцій, які можна виконати з наведеними вище ресурсами. Доступні різні дієслова (наприклад, отримати, переглянути, створити, видалити тощо), але зрештою всі вони є операціями створення, читання, оновлення або видалення (CRUD).

Маючи на увазі ці три елементи, ключовою ідеєю RBAC є наступне: необхідно пов'язати об'єкти, ресурси API та операції. Іншими словами, необхідно вказати, враховуючи користувача, які операції можна виконувати над певним набором ресурсів. Дані види ресурсів описуються у декларативному форматі YAML, як можна побачити на рисунку 4.2.

```yaml
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev
rules:
- apiGroups: ["", "extensions", "apps", "autoscaling", "batch"]
  resources: ["*"]
  verbs: ["*"]
- apiGroups: ["networking.k8s.io"]
  resources: ["ingresses"]
  verbs: ["*"]
- apiGroups: ["policy"]
  resources: ["poddisruptionbudgets"]
  verbs: ["*"]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: dev
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: test.user
```

Рисунок 4.2 – Приклад типових ClusterRole та ClusterRoleBinding

Сам RBAC має в основі декілька концепцій:

– ролі (Role): зв'язують ресурси API та дієслова. Їх можна використовувати повторно для різних об'єктів. Вони прив'язані до одного простору імен (не можна використовувати узагальнюючі символи, щоб представити більше одного, але можна

створити один і той же об’єкт ролі в різних просторах імен). Якщо необхідно, щоб роль застосовувалася для кластера, еквівалентний ресурс називається ClusterRoles. Такі ресурси можуть використовуватися у декількох просторах імен одночасно, що дозволяє уникнути повторення схожих об’єктів;

– прив’язка ролей (RoleBinding): з’єднує решту об’єктів-суб’єктів, що видно з рисунку 4.3. Враховуючи роль, яка вже пов’язує об’єкти API та дієслова, прив’язка встановлює, які суб’єкти можуть її використовувати. Для еквівалента на рівні кластера, що не має простору імен, існують ClusterRoleBindings [22].

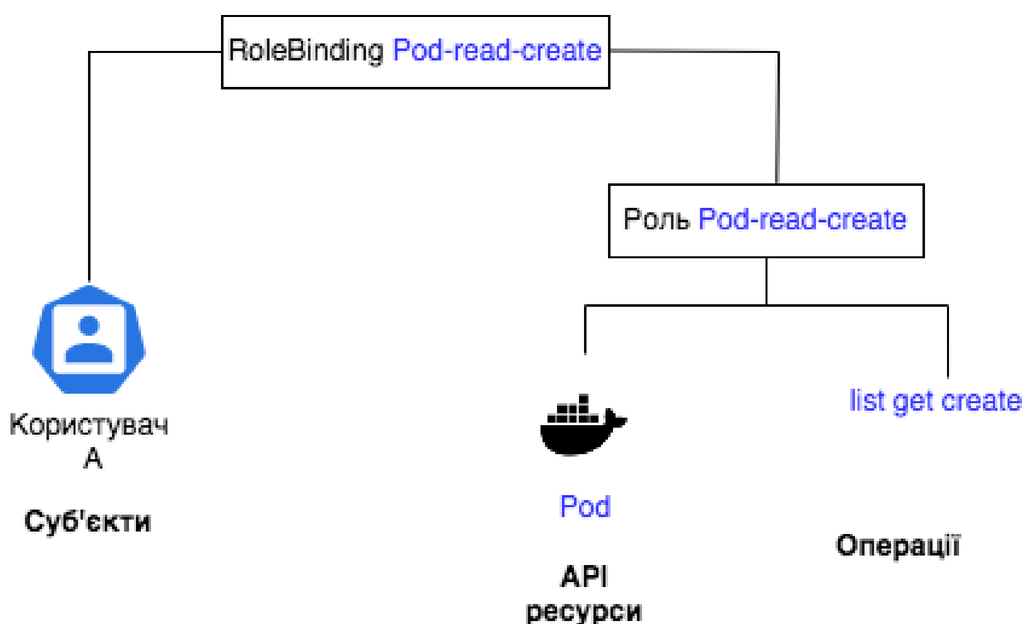


Рисунок 4.3 – Схема взаємодії RBAC ресурсів

Взаємодія всіх вище описаних ресурсів дозволяє забезпечити рівень безпеки, що задовольнить навіть найбільш вимогливих аудиторів з безпеки та дозволить отримати найбільш складні сертифікати відповідності з безпеки, що робить Kubernetes зручним інструментом для використання навіть у сферах, що досить сильно врегульовані законодавчо з точки зору безпеки (наприклад, медицина чи банківська справа).

У Kubernetes є звичайні користувачі, якими керують люди за межами кластера при підключенні до кластера (наприклад, за допомогою команди kubectl). Для ідентифікації ж окремих сервісів, що бажають взаємодіяти з API Kubernetes, існують

такі ресурси як сервісні акаунти («ServiceAccount»). Звичайні користувачі є глобальними і можуть отримати доступ до декількох просторів імен у кластері. Облікові записи сервісів обмежені одним простором імен. Це забезпечує ізоляцію простору імен, оскільки кожного разу, коли сервер API отримує запит від pod, його облікові дані застосовуватимуться лише до його власного простору імен. Kubernetes управляє обліковими записами сервісів від імені подів. Кожного разу, коли Kubernetes створює под, він присвоює йому сервісний акаунт. Обліковий запис сервісу ідентифікує всі процеси підсистеми, коли вони взаємодіють із сервером API. Кожен обліковий запис сервісу має набір облікових даних, змонтованих у секретному томі.

Кожен простір імен має обліковий запис служби за замовчуванням, який називається default. Коли створюється под, йому автоматично призначається обліковий запис служби за замовчуванням, якщо не був вказаний інший обліковий запис служби [23]. Приклад сервісного акаунту за замовчуванням можна побачити на рисунку 4.4.

```yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-dev
  namespace: dev
  labels:
    app: app
    env: dev
  annotations:
    iam.gke.io/gcp-service-account: "app-dev@test-project.iam.gserviceaccount.com"
```

Рисунок 4.4 – Приклад сервісного акаунту

Проте, можна й створити додатковий сервісний акаунт, котрому можна надати довільне ім’я або ж відредагувати список міток. Такі сервісні акаунти можуть бути присвоєні певним об’єктам таким як под, розгортання або ж сервіс (рисунок 4.5) [24].

Проте, можна й створити додатковий сервісний акаунт, котрому можна надати довільне ім’я або ж відредагувати список міток. Такі сервісні акаунти можуть бути присвоєні певним об’єктам таким як под, розгортання або ж сервіс.

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-dev
  namespace: dev
  labels:
    app: app
    env: dev
spec:
  replicas: 3
  selector:
    matchLabels:
      app: app
      env: dev
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 0
      maxSurge: 1
  template:
    metadata:
      labels:
        app: app
        env: dev
    spec:
      serviceAccountName: app-dev
```

Рисунок 4.5 – Присвоєння сервісного акаунту до сервісу

## 4.4 Сценарії використання системи

У Додатку 2 наведена діаграма використання системи. Дана діаграма описує можливі сценарії використання системи у різних ситуаціях. Слід зазначити, що діаграма використання описує саме можливості використання системи, не вдаючись до особливостей реалізації.

Можна розглянути кілька можливих сценаріїв використання системи:

– впровадження мікросервісної архітектури лише починається в умовах міграції з монолітного підходу на мікросервісний або ж розробка продукту ведеться

з самого початку та необхідно налагодити ініціалізацію та керування декількома сервісами в умовах використання у кластерах Kubernetes;

– компанія знаходиться готується до отримання сертифікатів з безпеки щодо архітектури ПЗ. В такому випадку конфігураційні файли мають бути захищені належним чином та доступ до них має бути чітко проконтрольованим;

– поточний підхід щодо розгортання та, особливо, тестування систем виявляється неоптимальним. В такому разі використання розроблюваної системи управління є надзвичайно виправданим кроком;

– з'являється необхідність швидко й легко створювати нові тестові оточення системи для, наприклад, надання до таких систем доступу для нових потенційних клієнтів або ж тестування нового функціоналу;

– у разі скорочення витрат на інфраструктуру, впроваджуються так звані «динамічні» тестові оточення, що можуть бути відключеними або ж видаленими у, наприклад, нічний (неробочий) час команди розробки.

Діаграма послідовностей системи наведена у Додатку 4.

## 4.5 Вибір і обґрунтування засобів програмної реалізації

Для реалізації поставлених задач необхідно обрати засоби, що мають активну підтримку нині та демонструють гарний потенціал для розробки і використання у майбутньому. Це є досить важливим, оскільки досить часто застарілі програмні комплекси (з англ. «legacy») створюють велику кількість проблем для розробників та відділів експлуатації програмного забезпечення. Такі проблеми можуть включати в себе як загрози безпеці, так і неможливість використовувати застаріле ПЗ на нових версіях ОС [6].

Окрім того, при використанні сервісів від різноманітних хмарних провайдерів загрозу становить й потенційна неможливість міграції платформи між різними хмарами. Зазвичай, більшість хмарних провайдерів (AWS, GCP, Azure та інші) мають серед своїх продуктів спектр послуг, що реалізує усі потреби більшості користувачів.

Відрізнятися такі послуги від різних вендорів можуть інколи лише назвами, особливостями використання та ціною. До можливих причин для міграції між хмарними середовищами можна віднести наступне:

- підвищення тарифів на послуги;
- відсутність сервісів, що б задовольнили певний спектр потреб;
- неналежні сертифікати щодо безпеки інфраструктури.

### 4.5.1 Вибір мови програмування

Вибір мови програмування для реалізації ПЗ є надзвичайно важливим фактором. Нині існує велика кількість мов, що мають різний функціонал, призначення та популярність.

Проте, слід почати вибір мови для реалізації проекту враховуючи існуючі обмежень. У якості середовища для зберігання параметрів конфігурації додатків планується для використання Google Secret Manager. Даний сервіс має клієнтські бібліотеки для таких мов програмування:

- C#;
- Go;
- Java;
- Node.js;
- PHP;
- Python;
- Ruby.

C# все ще є не найкращим вибором для реалізації продуктів, котрі плануються для використання на системах з ОС Linux, незважаючи на досить вдалий розвиток продукту .Net Core, що призваний відкрити двері світу Linux для мов програмування від корпорації Microsoft.

Java зазвичай використовується для створення біль складних додатків, аніж невеликі компоненти для автоматизації. Окрім того, середовище JVM, що ж

необхідним для запуску програм, створених мовою програмування Java потребує великої кількості обчислювальних ресурсів та займає багато місця при завантаженні, що є її безперечним недоліком.

Серед альтернатив, що залишилися – Go, Python та Ruby усі мови є нині досить актуальними, їх розробка активно ведеться та перспективи розвитку вказують на чудове майбутнє. Беззаперечним лідером щодо популярності серед розробників, що займаються аспектами експлуатації ПЗ є мова Go. Крім того, корпорація Google, котра стоїть за розробкою, власне Kubernetes (котрий також створений на мові Go) приймає активну участь у розробці даної мови. Також, корпорація Hashicorp, що є надзвичайно відомою завдяки своїм програмним комплексам для платформ на базі мікросервісної архітектури, використовує дану мову програмування для побудови усіх своїх рішень. А зважаючи на те, що більшість з даного ПЗ має відкритий програмний код, велика кількість інженерів також бере участь у розробці даного ПЗ на добровільних засадах.

Синтаксис мови Go є досить стислим, лаконічним та ефективним. Дана мова відноситься до тих, що компілюються перед виконанням. Програмний код не лише швидко компілюється у машинний код, але також має зручні механізми збору сміття (Garbage Collection) [25].

Однак швидкий час компіляції не є єдиною перевагою даної мови. Вбудовані механізми паралельних обчислень є досить простими у використанні та ефективно використовують багатоядерні комп’ютери та системи з розподіленими обчисленнями. У той же час також є підтримка гнучкої та модульної структури побудови програми. Пакет «Go Modules» дозволяє з легкістю розбивати програмний код, що може бути легко перевикористаним у інших додатках або ж бути заміненим на аналогічні реалізації, наприклад для інших хмарних оточень (для заміни Google Secret Manager).

В основі мови Go лежить простота. Це означає, що програмний код швидко компілюється, швидко працює, а також може бути швидко опанованим в разі необхідності. Крім того, щоб пришвидшити це, немає спадкування типів і класів. Це

полегшує розробку продукту, готового для використання за рекордні терміни. Плюс його простота у використанні дозволяє швидко і просто обслуговувати таке ПЗ [26].

Але переваги продуктивності не лише на цьому закінчуються. Оскільки Golang не має ВМ, він може компілювати безпосередньо до машинного коду, якщо ви можете виключити посередницьку збірку Go, що надає йому ще більшу перевагу в швидкості. Крім того, ПЗ, розроблюване даною мовою програмування є кросплатформенним.

Це не означає, що у порівнянні за швидкістю Go буде першою мовою серед усіх, однак вона точно посяде місце серед п'яти найшвидших.

Паралельність – це коли програма здатна виконуватися частинами, які можуть працювати незалежно одна від одної. Це дозволяє виконувати завдання не у строго визначеному порядку, проте результат залишатиметься таким же, як якщо б вони виконувались у строгому порядку. Іншими словами, це дозволяє алгоритму або програмі одночасно виконувати більше одного завдання. Ця концепція подібна до концепції паралельної обробки, але натомість є багато незалежних завдань, які виконують різні дії одночасно, а не виконують один і той самий алгоритм. На відміну від мови Python, паралельна обробка великої кількості конфігурування дозволить зменшити в рази час виконання одного й того ж завдання мовами Go чи Python [27].

Багатопоточні програми завжди було важко написати лише тому, що управління незалежними потоками вимагає координації декількох ресурсів одночасно. Завдяки тому, що нові багатоядерні процесори стають все потужнішими та доступнішими, а також команда, що стоїть за Go, що робить паралельність одним з головних пріоритетів розвитку – це робить Go дуже придатним для використання всієї обчислювальної потужності сучасних комп'ютерів.

Зазначається, що використання таких вбудованих інструментів, як goroutines, досить просто і чудово працює. Крім того, в Go є інші інструменти, спрямовані на широке використання паралельності. Це одна з головних причин, чому Go так добре справляється й зі створеннями серверів HTTP.

Крім того, кожному розробнику необхідний хороший механізм збору сміття. Вся ця дуже складна та необхідна робота автоматично виконується за замовчуванням, не вимагаючи складного налаштування чи реалізації.

У Go є дуже швидкий збирач сміття. Це допомагає у розробці програм, що створюються для швидкого виконання, переконуючись, що пам'ять системи не переповнена сміттям. Це значно покращує продуктивність будь-якої програми.

Кросплатформенність і простота у користуванні також є перевагами Go. Незалежно від обраної платформи – Windows, Linux, OS X чи інші. Go має вбудовані функції, які спеціально враховують сумісність між платформами. Відсутність складної системи залежностей також є перевагою. На виході, програмний продукт являє собою єдиний двійковий файл, що може бути запущений простою командою без необхідності встановлення будь-яких сторонніх залежностей у системі чи віртуальної машини.

Також, Go також має кілька функцій, які допомагають підтримувати чистий код. Go fmt форматує код і допомагає з його читабельністю.

Отже, Усі перелічені особливості роблять мову Go ідеальним вибором для створення розроблюваного ПЗ.

### 4.5.2 Custom Resource Definition та Kudo

Власної моделі робочих навантажень Kubernetes, такої як розгортання, сервіси та демони інколи є недостатньо для моделювання складних робочих навантажень. Проблеми зручності використання викликають величезні перешкоди на шляху до адаптації Kubernetes. Наприклад, розробники сервісів скаржаться на необхідність копіювати однотипні шаблони від сервісу до сервісу. Також деякі користувачі шаблонізації для створення сотень копій одного й того ж самого сервісу, що закінчується кошмаром для відділу налагодження [28].

Підтримка робочих навантажень під час виконання також розвивається, тому надзвичайно важко підтримувати різні версії в одному кластері Kubernetes.

Контролери CRD також керують життєвим циклом для власних ресурсів Kubernetes і обробляють запити щодо їх створення та редагування. Це включає також узгодження бажаної специфікації та фактичної конфігурації, оновлення стану CRD та запису подій. Без CRD розробники програм повинні керувати набагато більшим набором ресурсів, і цей процес досить часто виявляється досить схильним до помилок.

Розробники взаємодіють з кластером Kubernetes через інтерфейс командного рядка. Інструменти командного рядка отримують файли YAML конфігурації робочого процесу та інші властивості збірки (наприклад, ідентифікатор версії) та надсилають їх до служби подання завдань. Це гарантує, що лише перевірені та справжні робочі навантаження будуть надсилатися до кластеру Kubernetes. Тут відбувається автентифікація користувача, дотримання квот та часткова перевірка конфігурації CRD [29].

Після того, як CRD проходить перевірку, він надсилається далі в API Kubernetes. Контролер CRD спостерігає за подіями на всіх спеціальних ресурсах. Він перетворює CR у власні ресурси Kubernetes, додає необхідні ресурси у визначені користувачем підсистеми, встановлює відповідні змінні середовища та виконує інші необхідні роботи з контролю, щоб забезпечити повну підтримку інфраструктури в контейнерах програм розробників. Потім контролер CRD записує отримані власні ресурси назад в API Kubernetes, щоб вони могли бути сплановані планувальником і почати працювати.

Коли підпрограмний додаток запускається на Kubernetes, він автоматично отримує тимчасовий набір ключів для ідентифікації. Ці ключі використовується для доступу до API Kubernetes та ресурсів GCP (таких як секретів для взаємодії з іншими службами через). Тим часом контейнери ініціалізації керування конфігурацією та демон забезпечують усі необхідні залежності, завантажені перед запуском контейнера програми. Коли контейнер програми буде готовий, транспортний контролер трафіку та демон зареєструють нову IP-адресу, щоб зробити його

доступним для клієнтів. Мережевий демон налаштовує мережу для поду, ще до того, як він починає повноцінну роботу.

У якості контролера CRD був обраний Kudo. Це ПЗ з відкритим програмним кодом, що слугує зручним конструктором для побудови CRD. Реалізація фреймворку CRD є одною з найбільш вдалих у Kudo [30].

За допомогою конфігураційних файлів доволі зручного та зрозумілого формату Kudo дозволяє створити CRD контролер, котрий розгортається у середині кластеру Kubernetes та готовий до роботи. Також, Kudo має публічні репозиторії, котрі можна використовувати у якості прикладу для створення власних контролерів.

Особливу увагу у Kudo приділено логуванню та тестуванню. Створення CRD є досить клопіткою справою, тому для зручності їх налаштування, Kudo виводить інформативні та оптимальні за обсягом повідомлення, що є досить зручним при роботі.

### 4.5.3 Вибір середовища розробки

У якості середовища розробки був обраний додаток VS Code. Це розробка компанії Microsoft, що є надзвичайно популярною та широко використовується. Даний редактор доступний на усіх популярних платформах, а саме:

- Windows;
- Linux;
- OS X.

Цей текстовий редактор має зручний та зрозумілий інтерфейс, його зображено на рисунку 4.6. Його особливістю є надзвичайно потужний менеджер плагінів. За замовчуванням, даний додаток є лише текстовий редактором.

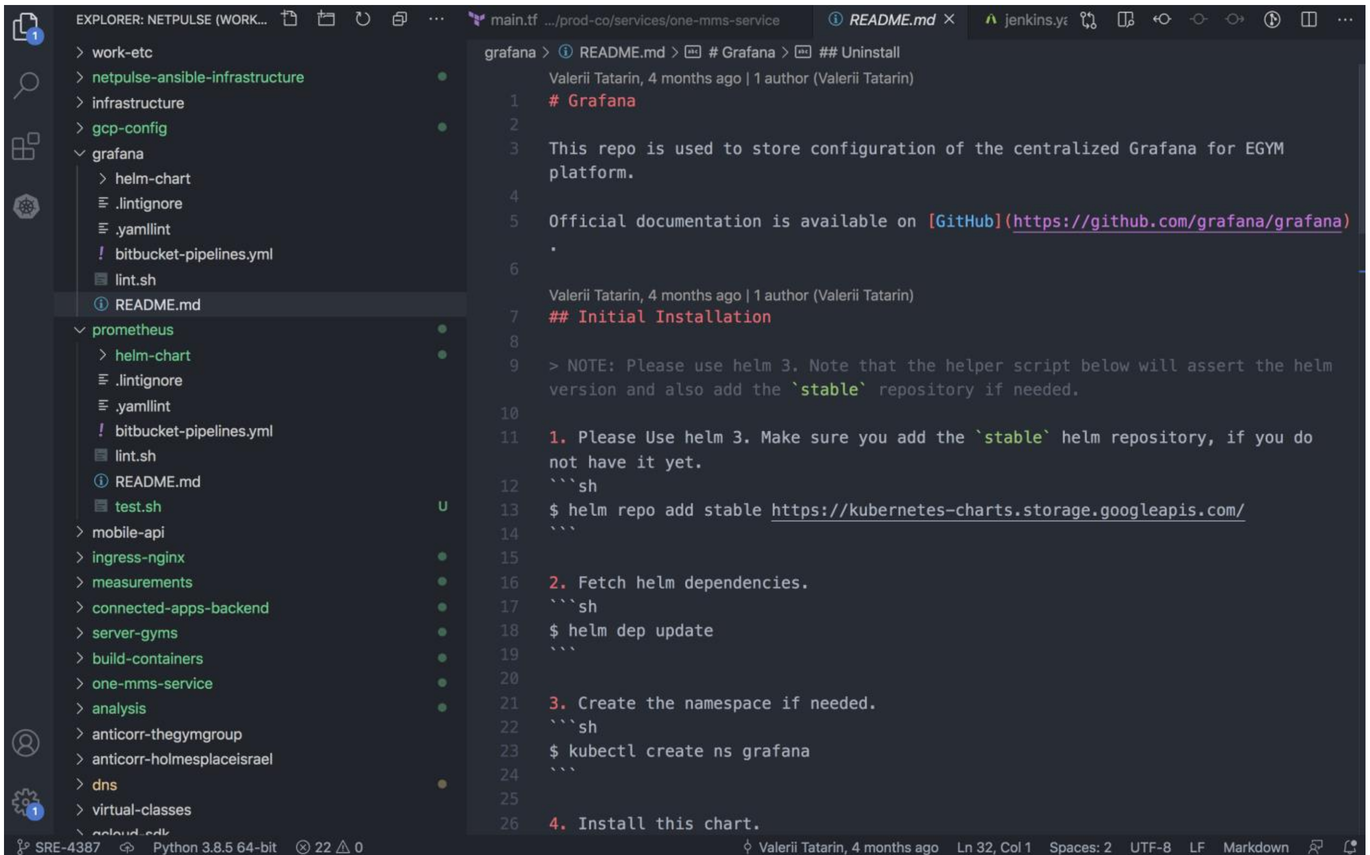


Рисунок 4.6 – Інтерфейс Visual Studio Code

Проте, кожен розробник має змогу персоналізувати його під власні потреби, встановивши будь-які з тисяч доступних плагінів, приклади яких видно на рисунку 4.7. Плагіни можуть змінювати дизайн редактору, додавати підтримку автокорекції синтаксису різних мов програмування, інтегрувати редактор з середовищами розгортання, або ж навіть й кластерами Kubernetes.

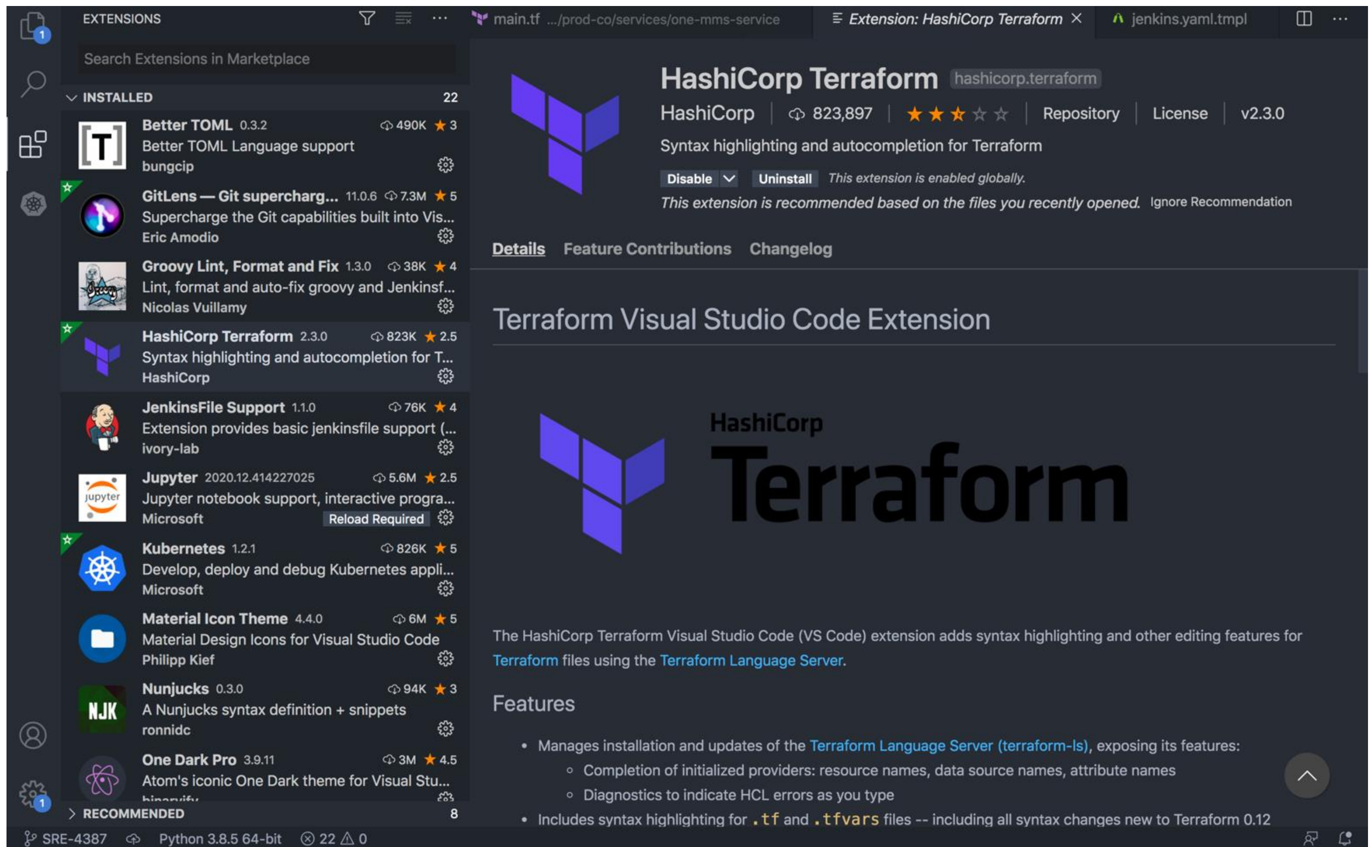

Рисунок 4.7 – Менеджер плагінів Visual Studio Code

### 4.5.4 Засоби для автоматизації розробки

Для пришвидшення процесу розробки ПЗ був використаний інструмент для побудови процесу неперервної розробки Bitbucket Pipelines та система контролю версій Bitbucket. Дані рішення доступні як SaaS продукти та не потребують витрат на утримання та налаштування. Для безкоштовних репозиторіїв надається безкоштовний план Bitbucket Pipelines, котрого було цілком достатньо для задоволення потреб.

Bitbucket має зручний веб-інтерфейс, котрий зображено на рисунку 4.8, що дозволяє перевіряти статус поточний завдань навіть з мобільного телефону або будь-якого іншого пристрою. Також, підтримується інтеграція з Google GSuite.

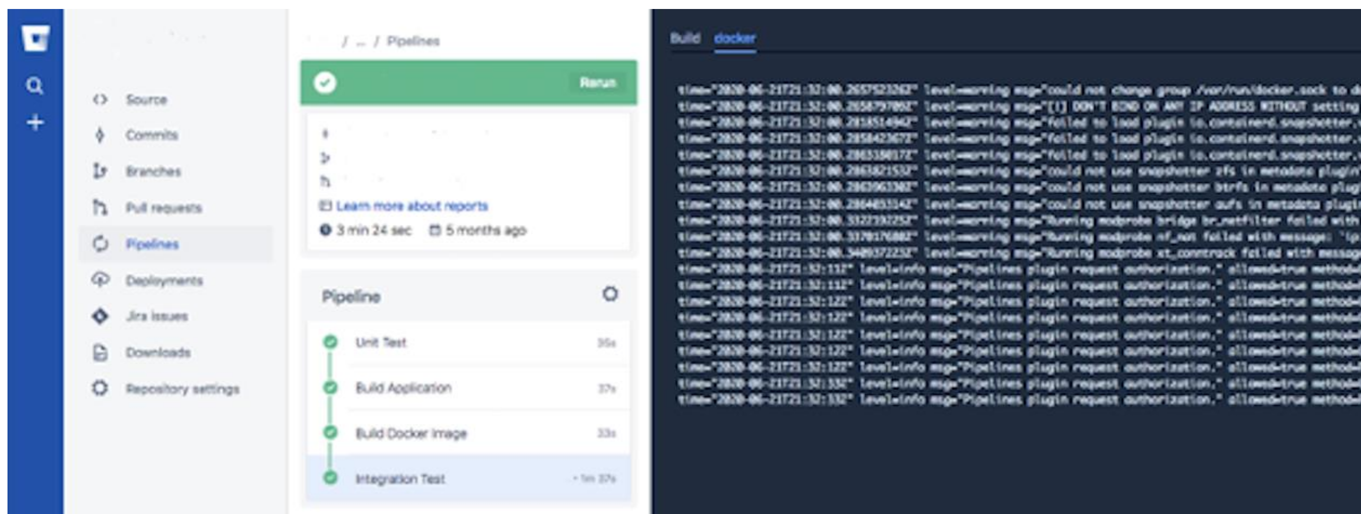


Рисунок 4.8 – Веб-інтерфейс Bitbucket Pipelines

Bitbucket Pipelines конфігурується за допомогою єдиного файлу у YAML форматі. За допомогою зручного синтаксису можна описати довільний план для тестування або ж збірки ПЗ.

До задач, котрі були винесені до Bitbucket Pipelines можна віднести наступні:

– виконання юніт тестів;
– збірка додатку;
– створення образу Docker для використання додатку як допоміжний контейнер;
– виконання інтеграційні тестів.

### 4.5.5 Засоби тестування ПЗ

Автоматизовані тести є оптимальним підходом, оскільки вони не потребують задіяння у процесі розробки додаткових тест-інженерів, котрі могли б тестувати ПЗ мануальними способами. Саме тому для забезпечення належної якості розроблюваного ПЗ було прийнято рішення покривати розроблюваний код автоматизованими тестами.

Оскільки компонент для ініціалізації веб-додатків був розроблений за допомогою мови програмування GO, то для тестування використовувалась найбільш широко поширена бібліотека для тестування – Go/testing.

Бібліотека Go/testing дозволяє досить зручно покривати розроблюваний програмний код юніт-тестами та запускати їх за допомогою єдиної команди «go test».

Серед розроблених тестів можна виділити дві основні категорії:

– юніт-тести;
– інтеграційні тести.

Підхід юніт-тестування був використаний для перевірки коректної роботи більшості з розроблених функцій додатку. Наприклад, наявні тести на перевірку формату даних, котрі повертаються після виклику певних функцій.

Інтеграційні тести, у свою чергу, виконують перевірки більш високого рівня. Наприклад, наявні тести, що перевіряють, чи були коректно завантажені параметри конфігурації у відповідні файли чи ж змінні оточення.

Повний програмний код мовою програмування Go для інтеграційних та юніт-тестів наведений у Додатку Б.

Також, досить велика кількість перевірок було створена з використанням скриптів на Bash. Оскільки результатом виконання даного програмного забезпечення є файли та змінні конфігурації, Bash є чудовим вибором, адже за його допомогою зручно працювати з файлами у файловій системі ОС Linux чи ж змінними оточення виконання ПЗ.

### 4.5.6 Автоматизація розгортання Kubernetes кластера у GCP

Розроблювана система автоматизації направлена на роботу з кластерами Kubernetes. Розробка проводилася у хмарному оточенні Google Cloud Platform, адже доступні там для розгортання кластери є найбільш сучасними, швидкими та оптимальними для використання.

Інфраструктура як код (англ. Infrastructure as Code (IaC)) — це спосіб постачання та керування обчислювальними та мережевими ресурсами методом їх опису у вигляді програмного коду, на відміну від налаштовування необхідного обладнання власноруч чи з допомогою інтерактивних інструментів [31].

IT інфраструктура, керована таким чином, охоплює як фізичні сервери, так і віртуальні машини, а також пов'язані з ними ресурси. У підході використовуються як виконувані скрипти, так і декларативні визначення, шаблони, які можуть перебувати в системі контролю версій. Термін найчастіше використовується для просування декларативного опису IT інфраструктури.

IaC подається в першу чергу як рішення для платформ хмарних обчислень, які, в свою чергу іноді позиціонуються як IaaS рішення. Iac підтримує IaaS, але їх не слід плутати. Опис інфраструктури тестового кластеру наведений у діаграмі 7.

Terraform – це інструмент для безпечної та ефективної побудови, зміни та модернізації інфраструктури. Terraform може керувати існуючими популярними постачальниками послуг хмарних обчислень, а також індивідуальними приватними рішеннями. Файли конфігурації описують Terraform компоненти, необхідні для запуску однієї програми або всього центру обробки даних. Terraform генерує план виконання, описуючи, що він буде робити для досягнення бажаного стану, а потім виконує його для створення описаної інфраструктури. По мірі зміни конфігурації Terraform може визначати, що змінилося, і створювати додаткові плани виконання, які можна застосувати. Інфраструктура, якою Terraform може керувати, включає компоненти низького рівня, такі як обчислювальні машини, сховище та мережу, а також компоненти високого рівня, такі як записи DNS, функції SaaS тощо. Для опису конфігурацій terraform використовує HCL.

Terraform може працювати з API багатьох популярних хмарних провайдерів або технологій. Серед них:

- AWS;
- GCP;
- Azure;
- Heroku;
- Cloudflare;
- Cisco ASA;
- Oracle Cloud.

Власне, розгортання кластеру Kubernetes у будь-якому з можливих оточень є досить клопіткою та непростою задачею. Саме через це для оптимізації процесу розробки та тестування було прийнято рішення автоматизувати розгортання тестових кластерів та усієї необхідної інфраструктури за допомогою Terraform. Інфраструктура тестового кластеру зображена у Додатку 7. Повний програмний код для розгортання Kubernetes кластерів за допомогою Terraform наведений у Додатку В, а діаграма залежностей ресурсів наведена у Додатку 6.

### 4.6 Короткий опис програми ініціалізації додатків

Даний процес включає в себе етап завантаження параметрів конфігурації та запуску самого додатку.

У якості бази даних для зберігання параметрів конфігурації був обраний Secret Manager. Це сервіс від Google, що надає зрозумілий веб-інтерфейс для користувачів та має зручний програмний інтерфейс.

Для того, щоб забезпечити швидке та легке масштабування параметрів конфігурації, використаємо систему міток у Secret Manager. Таким чином, серед обов’язкових міток для роботи з Secret Manager введемо наступні:

- service;
- environment;
- type;
- inject.

За допомогою міток «service» та «environment» контролюватимемо приналежність параметру конфігурації до певного сервісу, що запущений у певному оточенні. Наприклад, сервіс «application-a» у оточенні «production».

Мітка «inject» вказує на те, чи слід даний параметр конфігурації завантажувати. Таким чином можна досить легко деактивувати секрети, що дозволить не використовувати певну частину конфігурації, не видаляючи її.

Мітка «type» вказуватиме на те, який це тип параметру конфігурації. Чи мусить це бути змінна оточення чи окремий файл. У разі, якщо це файл, то він буде доступний

у поточній директорії запуску з правами лише на читання. Таким чином, навіть якщо зловмисник зможе потрапити до контейнеру, то у разі коректного налаштування образу контейнеру (відсутності прав супер користувача), файл не зможе бути змінений.

У разі ж якщо мітка «type» встановлена у значення «environment», то параметр конфігурації буде завантажений у якості змінної оточення.

Зважаючи на те, що однією з особливостей запуску додатків у Docker контейнерах є відсутність в них менеджеру процесів, для коректної обробки програмним забезпеченням системних сигналів, необхідно правильно запускати послідовні процеси у контейнері.

Розроблюване програмне забезпечення може використовуватися у двох режимах роботи:

- як утиліта, що є вбудованою у образ контейнеру та котра запускається перед запуском основного додатку;
- як допоміжний контейнер (initContainer).

У разі використання першого способу, утиліта має бути запущена як частина передстартового скрипта основного додатку, так званий «Entrypoint» у екосистемі Docker, а основний додаток має бути запущений за допомогою директиви «CMD» екосистеми Docker. При виконанні описаного алгоритму, основний додаток буде запущений з тим самим ідентифікатором процесу («PID»), що й передстартовий скрипт та усі системні сигнали будуть оброблені коректно. Такий результат досягається за допомогою функціоналу ОС Linux, що дозволяє заміщувати дочірній процес на новий, аналогічно до директиви «$@» у середовищі виконання Bash.

Ще однією перевагою такого способу використання утиліти є те, що змінні оточення залишаються доступними лише процесу, котрому належить середовище. Тобто, будь-який користувач, що отримає доступ до контейнеру не зможе переглянути параметри конфігурації додатку. Вони доступні лише самому процесу та у одній з системних директорій («/proc/$PID/environ») [32], котра доступна лише root користувачу (за коректної конфігурації контейнеру, такі права мають бути відсутні за

замовчуванням). Таким чином, безпека параметрів конфігурації, наприклад API ключів від сторонніх сервісів та інше, є на найвищому рівні.

У разі ж використання додатку як допоміжного контейнеру, жодні зміни до основного образу контейнеру не потребуються. Проте, завантаження параметрів конфігурації як змінних оточення у такому разі не є доступним, а може бути використаний лише спосіб з файлами конфігурації.

Детальний алгоритм роботи програми зображений у Додатку 5, а в свою чергу, лістинг програмного коду наведений у Додатку А.

### 4.7 Короткий опис системи управління додатками

Для того, щоб розширити список примітив, з якими може працювати Kubernetes, він підтримує механізм під назвою Custom Resource Definition. За його допомогою, можна «навчити» API Kubernetes працювати з сутностями нових типів, що можуть уособлювати цілком певні мікросервіси, або ж навіть частини функціоналу платформи.

Наприклад, можна створити CRD під назвою «Billing» може представляти усі компоненти, що стосуються підсистеми оплат певної системи. Таким чином, для створення ще одного тестового оточення системи оплат необхідно створити лише один об’єкт типу «Billing», а системні компоненти Kubernetes декомпозують даний об’єкт на десятки або й сотні примітив, котрі будуть пізніше створені.

Для забезпечення більшості потреб середньостатистичної команди розробки було створено наступні об’єкти:

AppService – це сервіс що не має постійного стану, або ж зберігає свої дані у спеціально відведеній БД. Багато базових систем базуються на наборі таких сервісів;

AppJobSet моделює групові завдання, які виконуються та завершаються. Дуже поширеною схемою є те, що кілька завдань запускають однакові контейнери паралельно, кожен захоплює частку робочого навантаження, не залежачи один від одного та завершуються у разі виконання задачі (наприклад, ETL завдання);

AppCronJob широко застосовується для виконання періодичних задач. Це обгортка навколо власного ресурсу Kubernetes з додаванням сервісних акаунтів, політик доступу, мережевих політик, додаткових контейнерів та іншого;

AppDaemon використовуються зазвичай для інфраструктурних задач. Наприклад, може використовуватися утилітами для моніторингу, логування та контролю доступу, що мають бути запущені на кожному з вузлів;

AppTrainerJob створений для розгортання Tensorflow та Pytorch, забезпечуючи той самий рівень підтримки під час виконання, що й усі інші CRD. Оскільки штучний інтелект зараз використовується все ширше, компоненти для його розвинення та пристосування моделей присутні досить часто.

Таким чином, створені CRD перелічених вище типів дозволяють оптимальним способом керувати робочими навантаженнями Kubernetes кластерів, що збільшує загальну швидкість розробки програмний компонент, особливо при використанні мікросервісної архітектури. Діаграма компонентів наведена у Додатку 8.

Лістинг програмного коду для створення CRD та оператору за допомогою фреймворку Kudo наведений у Додатку Б.

## 4.8 Висновки до розділу

В даному розділі були визначені вимоги щодо оточення, в якому може використовуватися розроблюване програмне забезпечення. Також були визначені вимоги до, власне, розроблюваної автоматизованої системи.

Крім того, був проведений аналіз технічних засобів, за допомогою яких може бути реалізована автоматизована система та були обрані оптимальні рішення для реалізації.

Також, у даному розділ були описані засоби автоматизації розробки програмного забезпечення, що використовувалися у ході створення системи автоматизації, такі як Bitbucket та Bitbucket Pipelines та середовище розробки системи, Visual Studio Code.

Оскільки створення кластерів Kubernetes для тестування розроблюваної системи автоматизації є важливим етапом розробки самої системи, був описаний засіб для автоматизації цього процесу. Програмний код для цього наведений у відповідному Додатку В.

Короткий опис роботи системи автоматизації допомагає зрозуміти аспекти функціонування розробленої системи, а програмний код системи наведений також у, відповідно, Додатку А та Додатку Б.

Розроблена система відповідає поставленим вимогам та здатна успішно функціонувати в умовах реальної розробки програмного забезпечення, що буде описано у наступних розділах.

# 5 АНАЛІЗ ЕФЕКТИВНОСТІ РОБОТИ СИСТЕМИ

Дана система була впроваджена на одному з проектів ТОВ «Стар Україна», про що свідчить відповідний акт впровадження, що додається до дисертації. Ефективність системи оцінюється шляхом порівняння продуктивності команди розробників протягом двох однакових за тривалістю циклів розробки (два тижні кожен). Інженери виконували повсякденні задачі з використанням традиційних підходів до ініціалізації та управління веб-додатками протягом першого періоду розробки, потім була впроваджена розроблювана система (діаграма розгортання наведена у Додатку 3), опісля чого команда користувалася даною системою протягом наступного періоду розробки.

## 5.1 Ініціалізація додатків

Після впровадження системи досить ефективним проявило себе те, що системні сигнали почали оброблятися коректно. Через це, сервери, а саме Tomcat, своєчасно отримували попередження про зупинку роботи та мали час завершити обробку поточних викликів, не приймаючи нових запитів від клієнтів.

В той час, як з використанням традиційних способів запуску додатків, зазвичай були допущені помилки при конфігурації образів контейнерів та додатки отримували лише останній сигнал про завершення роботи, що призводило до агресивного відключення серверів, що в свою чергу, призводило до переривання процесу обробки запитів і клієнти отримували відповідні коди помилок, замість очікуваної відповіді.

Даний підхід забезпечив більш коректну роботу тестового оточення, що позитивно вплинуло на стабільність розроблюваної системи. Крім того, дана система автоматизації має гарні шанси на впровадження й у оточеннях, котрі забезпечують й роботу реальних користувачів, а не лише у тестових кластерах проекту.

Крім того, розроблений спосіб завантаження параметрів конфігурації зробив систему більш безпечною, адже у разі потрапляння до робочого контейнеру у кластері Kubernetes, користувач без прав супердоступу не має змоги проглянути значення параметрів конфігурації.

Також, слід зазначити, що кількість запитів на зміну конфігурації оточень до команди експлуатації програмних додатків було зменшено, адже користувачі мали змогу виконувати такі задачі самостійно з використанням зручного веб-інтерфейсу Google Secret Manager.

## 5.2 Управління додатками

Досить суттєвим недоліком мікросервісної архітектури є те, що тестування систем, що побудовані з використанням даного підходу є досить складним.

Розроблена система автоматизації дозволяє розгортати нові оточення системи для виконання певних тестів або ж надання доступу новим потенційним клієнтам до їх тестового оточення на ~30% швидше за традиційний підхід. Таким чином, команда тестувальників мала змогу обробити на ~30% задач більше за другий період розробки, ніж за перший.

Крім того, впровадження динамічних робочих оточень дозволило зекономити близько 30% витрат, що пов’язані з утриманням тестових кластерів Kubernetes. Динамічними оточеннями є ті, котрі здатні знищуватися у період неробочого часу команди тестувальників та відновлюватися перед початком наступного робочого дня. Саме розроблена система дозволяє легко й швидко налаштовувати нові оточення інтегрованої системи.

## 5.3 Висновки до розділу

Отже, розроблена система автоматизації відповідає поставленим критеріям до розробки. Впровадження даної системи у реальному проекті показало підвищенні ефективності у роботі команд. Це демонструється наступними факторами:

- надання розробникам більшої автономності;
- пришвидшення процесу тестування ПЗ;
- зменшення витрат на утримання тестових кластерів;
- підвищення рівня безпеки інфраструктури.

Слід зазначити, що отримані відгуки щодо роботи системи також є позитивними та вказують на правильність обраного вектору розробки автоматизованої системи. Система має потенціал для більш широкого використання.

Також, обрані компоненти для реалізації системи є досить вдалими, оскільки більшість з розробників тестової команди для впровадження мали змогу оцінити програмний код та забажали внести свої корективи щодо роботи системи у майбутньому.

# 6 РОЗРОБКА СТАРТАП ПРОЕКТУ

В даному розділі запропоноване рішення для монетизації продукту, що описаний у даній магістерській дисертації. Після аналізу поточного стану справ на ринку, були сформульовані головні вимоги, запропоновані ідеї щодо монетизації, виділено сильні та слабкі сторони потенційного комерційного продукту.

В більшості підрозділів інформація буде представлена у вигляді таблиць.

## 6.1 Опис ідеї проекту

Основною метою стартап-проекту є розробка системи kube-init, яка може бути використана замовниками у майбутньому на різноманітних проектах (про це йтиме мова далі). Для початку, оглянемо зміст ідеї, можливі сфери застосування, головні переваги продукту у таблиці 6.1.

Таблиця 6.1 – Опис ідеї стартап-проекту

| Основний зміст ідеї | Сфери застосування | Переваги використання системи для користувача |
|---|---|---|
| | Домашні системи «Розумний дім» | Забезпечення надійної роботи програмних компонент комплексів «Розумний дім» |
| | Навчальні заклади/курси | Пришвидшення процесу навчання за рахунок оптимізації процесів поставки ПЗ |
| | Розробка ПЗ | Пришвидшення етапів тестування ПЗ, надання можливості швидкого розгортання тестових оточень (динамічних у разі потреби) |

Огляд ринку виявив, що фактично у розроблюваної системи немає потенційних конкурентів. Аналогом може слугувати поєднання схожих систем, таких як Berglas/Helm. Проте, впровадження альтернативного варіанту є досить складним, документація не є достатньо детальною, а поріг для розуміння зазначених компонент є досить високим. Більш докладно описані слабкі, нейтральні та сильні сторони проектів у таблиці 6.2.

Таблиця 6.2 – Опис сильних, слабких та нейтральних сторін основної ідеї стартап-проекту

| Техніко-економічні ознаки ідеї | (потенційні) товари/концепції конкурентів | | | W (слабка сторона) | N (нейтра-льна сторона) | S (сильна сторона) |
|---|---|---|---|---|---|---|
| | Поточний проект | Berglas | Helm | | | |
| Мова розробки | Go | Go | Go | | + | |
| Архітектура | Мікро-сервісна | Моно-літна | Моно-літна | | | + |
| Тип ліцензії | MIT | Apache | GNU | | | + |
| Спосіб використан-ня | Змінні оточення, файли | Файли | Файли | | | + |
| Підтримува-не хмарне середовище | GCP | GCP | Усі | + | | |
| Забезпечен-ня високої надійність (HA) | Є | Є | Є | | + | |

Продовження таблиці 6.2 – Опис сильних, слабких та нейтральних сторін основної ідеї стартап-проекту

| Техніко-економічні ознаки ідеї | (потенційні) товари/концепції конкурентів | | | W (слабка сторона) | N (нейтра-льна сторона) Поточ-ний проект | S (сильна сторона) Berglas |
|---|---|---|---|---|---|---|
| | Поточний проект | Berglas | Helm | | | |
| Алгоритми захисту даних | AES-256 | AES-256 | Відсутній | | | + |

Наведений вище перелік сильних сторін розроблюваного ПЗ слугує основою для формування конкуренто-спроможної бізнес-моделі для світового ринку розробки ПЗ, адже дані критерії високо цінуються по всьому світу. Основною задачею буде лише збільшення спільноти користувачів, проте з часом ця проблема буде вирішена.

## 6.2 Технологічний аудит ідеї проекту

У таблиці 6.3 наведений технологічний аудит ідеї. Визначений технологічний стек для побудови розроблюваного рішення, проаналізована наявність необхідних технологій у вільному доступі з огляду на актуальні ліцензії.

Ліцензування є неві’ємним та важливим аспектом вибору засобів реалізації проекту, адже деяке ПЗ може розповсюджвуватися з ліцензіями, що забороняють використання у комерційних проєктах.

Також, деякі ліцензії забов’язують розробників надсилати звіти щодо мети використання їх ПЗ

Таблиця 6.3 – Технологічна здійсненність ідеї проекту

| № п/п | Ідея проекту | Технології реалізації | Наявність технологій | Доступність технологій |
|---|---|---|---|---|
| 1 | Ініціалізація додатків | Go | Наявні власні напрацювання | Потрібно доопрацьо-вувати |
| 2 | Управління додатками | Kudo, Terraform | У вільному доступі є велика кількість прикладів реалізації інших систем, котрі можуть слугувати прикладом | Доступні |
| Обрана технологія реалізації ідеї проекту: Розробляється система автоматизація з використанням мови програмування Go та CRD фреймворку Kudo. Дані технології є безкоштовними та у вільному доступі. Єдиним компонентом, котрий потребує витрат – користування GCP Secret Manager, проте навіть безкоштовний ліміт сервісу є достатньо великим. | | | | |

З огляду на наведені вище дані, технічний проект має змогу реалізації з технічної точки зору.

### 6.3 Аналіз ринкових можливостей стартап-проекту

Проаналізовано можливості на даному ринку, які можна запровадити під час запуску стартап-проекту. Також наведені загрози, які потенційно можуть негативно вплинути на проект.

У таблиці 6.4 наведені дані аналізу.

Таблиця 6.4 – Аналіз загроз на ринку для стартап-проекту

| № п/п | Характеристика ринку | Критерій |
|---|---|---|
| 1 | Кількість головних конкурентів, од | ~ 2 |
| 2 | Обсяг продаж, грн/ум.од | Неможливо об'єктивно оцінити |
| 3 | Умови для розвитку | ↑ |
| 4 | Обмеження для старту проекту | - |
| 5 | Проходження сертифікації для запуску | За бажанням |
| 6 | Динаміка ринку | ↑ |
| 7 | Рентабельність, % | Дуже висока, від ~ 88 відсотків за одне рішення |

Виходячи з даних аналізу, можна припустити, що ніша є доволі приваблива для запуску стартап-проекту.

Можливі клієнти, їх характеристики та приблизні вимоги до продукції представлені нижче.

У таблиці 6.5 представлені головні характеристики можливих клієнтів, де ЦА – це цільова аудиторія, ПЦГК – потенційна цільова група клієнтів, ШР – швидкість розгортання.

Таблиця 6.5 – Характеристика можливих клієнтів

| № п/п | Чинник | ЦА | Різниця у поведінці різних ПЦГК | Характеристики, яких потребують користувачі |
|---|---|---|---|---|
| 1 | Контроль оточень | Інженери, які застосовують Kubernetes кластер | Користувачі потребують різних результатів на даному кластері | – простий для сприйняття інтерфейс використання<br>– висока ШР |
| 2 | Швидке проектування кластера на етапі створення проекту | Адміністратори та архітектори проекту | Кластери розгорнуті для різних середовищ використання | – простий для сприйняття інтерфейс використання<br>– висока ШР |

Після аналізу груп можливих клієнтів був проведений аналіз ринкового середовища. Справа в тім, що на різні групи клієнтів можуть впливати різні чинники при використанні продукту.

Наприклад, для різних груп інженерів можуть мати різну пріоритетність такі важливі фактори, як швидкість роботи або зручність у використанні.

Таблиці факторів, які позитивно впливають на запуск проекту на ринку, та факторів, які навпаки негативно впливають наведені в таблицях 6.6 та 6.7.

Таблиця 6.6 – Чинники загроз

| № п/п | Чинник | Аналіз загрози | Рішення |
|---|---|---|---|
| 1 | Конкуренти | Збільшення конкурентів зі схожими проектами на ринку | Розширення функціоналу системи |

Продовження таблиці 6.6 – Чинники загроз

| № п/п | Чинник | Аналіз загрози | Рішення |
|---|---|---|---|
| 2 | Відсутність інвестицій | При наявності інвестицій початок та розробка стартап проекту є більш прискорені | Пошук та залучення іноземних інвесторів |
| 3 | Висока ціна | Висока ціна включає в себе розробку, яка потребує високоякісних спеціалістів, які мають навики в інноваційних рішеннях. | Застосування реклами для популяризації проекту. Збільшення користувачів |

Таблиця 6.7 – Критерії можливостей

| № п/п | Чинник | Рішення | Узгодження |
|---|---|---|---|
| 1 | Вільний ринок в країні | На даний час наявний вільний вхід на ринок | - |
| 2 | Застосування рекламних каналів | Надання пріоритету діджтитал направленню | Запровадження інноваційних маркетингових рішень |

У наведеній нижче таблиці показані дані аналізу, які характеризують загальні ознаки конкуренції на ринку (таблиця 6.8).

Таблиця 6.8 – Ступеневий аналіз конкуренції на ринку

| Особливості конкурентного середовища | В чому проявляється даний критерій | Вплив на діяльність підприємства (можливі дії компанії, щоб бути конкурентоспроможною) |
|---|---|---|
| 1. Вказати тип конкуренції<br>Тип конкуренції: ринок України – монополістична, світовий – чиста | На ринку України відсутні схожі рішення, на світовому – присутні деякі функції проекту | Постійний розвиток за допомогою зворотнього зв'язку від клієнтів. |
| 2. За рівнем конкурентної боротьби – міжнародний | Схожі рішення (не в повному обсязі) представлені на міжнародному ринку | Максимальна масштабність розповсюдження проекту |
| 3. За галузевою ознакою – міжгалузева | Інформаційний ринок присутній майже у кожній галузі | Використання продукту у різних галузях |
| 4. Конкуренція за видами товарів – товарно-родова | Конкуренція на рівні технології задоволення потреб. | Ведення активної маркетингової діяльності |
| 5. За характером конкурентних переваг – нецінова | Користувачі готові платити високу ціну за більш якісну систему | Спрямування більшої частки доходів на постійне вдосконалення та розвиток технологій |
| 6. За інтенсивністю – марочна | Система призначена для спільної цільової аудиторії | Введення активної маркетингової діяльності |

Як видно з проаналізованих даних, конкуренції як такої немає, тобто можна вільно зайняти дану нішу на ринку. Також існують рішення, які частково є готовими рішеннями: деякі потрібні бібліотеки вільні у доступі для того, щоб використовувати їх для розробки певного юніта програми, який ліцензований.

Головними перевагами у виборі даного продукту є простота у використанні, великий спектр функцій та якість роботи, яких немає у конкурентів.

Після аналізу конкуренції проводиться більш детальний аналіз умов конкуренції в галузі (за моделлю 5 сил М. Портера). Модель п'яти сил Майкла Портера – це простий, але ефективний інструмент бізнес-аналізу, який використовується для визначення того, чи може стратегія бути вигідною в конкурентному середовищі компанії. Проводячи належним чином, за допомогою відповідних інструментів, аналіз п'яти сил може надати безцінне уявлення про конкуренцію бізнесу та про те, скільки влади можливо отримати на ринку, тому можна скорегувати свою стратегію успіху [33].

Кожна сила в моделі Майкла Портера являє собою окремий рівень конкурентоспроможності товару: ринкова влада покупців, ринкова влада постачальників, загроза вторгнення нових учасників, небезпека появи товарів – замінників, рівень конкурентної боротьби або внутрішньогалузева конкуренція.

Проведення ретельного дослідження конкурентів є важливим аспектом на початку створення свого проекту. Насправді це життєво важлива частина будь-якої успішної маркетингової стратегії. Ознайомлення з тим, що роблять ваші конкуренти, допоможе визначити, якими є їхні сильні та слабкі сторони, даючи краще уявлення про те, чого слід уникати та що слід використовувати як натхнення. Точне знання того, що відбувається у вашій галузі (наприклад, будь-які сучасні тенденції), також дозволить бізнесу бути більш динамічним, розвиваючись відповідно до вимог споживача.

Аналіз п'яти сил Майкла Портера йде на крок далі, вимагаючи від замовника детального вивчення конкретних аспектів ринку, щоб замовник міг приймати більш стратегічні бізнес-рішення. Розуміючи можливості галузі для отримання прибутку та будь-які потенційні загрози для успіху, можна легше визначити сильні сторони

компанії та скористатися ними, або визначити свої слабкі сторони та способи їх вдосконалення [33].

У таблиці 6.9 наведений аналіз даних, з якого випливає що є можливість виходу на ринок є достатньо великою, адже проект пропонує унікальні рішення та техгології, з якими буде дуже важко конкурувати частково існуючим рішенням, а також беручи до уваги те, що система є простою для впровадження, підтримує масштабованість, вона буде задовольняти потреби більшості компаній.

Таблиця 6.9 – Аналіз конкуренції

| | Прямі конкуренти | Потенційні конкуренти | Постачальники | Клієнти | Товари-замінники |
|---|---|---|---|---|---|
| Складові аналізу | Прямі конку-ренти на ринку України відсутні, на світовому присутні часткові рішення | Поява конкурентів на ринку України та світовому ринку | Постачаль-ники відсутні | Обмежені платоспро-можністю | Відсутні |
| Висновки: | Конкуренція несформована | Присутні часткові рішення | Відсутні | Можлива менша плата за не повний пакет функцій послуги | Відсутні |

У таблиці 6.10 наведені дні аналізу конкуренції, не включаючи особливостей ідеї проекту, критерії вибору товару споживачем та факторів маркетингового середовища, формується список конкурентних факторів.

Таблиця 6.10 – Критерії конкурентоспроможності

| № п/п | Критерій конкурентоспроможності | Детальний аналіз чинників |
|---|---|---|
| 1 | Задоволення потреб користувачів у даній сфері | Аналізуючи всю інформація, яка доступна в просторі інтернету щодо схожих рішень конкурентів, можна зробити висновок, що задоволення потреб користувачів повністю не реалізовано і не представлено на ринку. Даний стартап проект буде першим рішенням для повного задоволення потреб користувачів. |
| 2 | Висока взаємодія | |
| 3 | ШР | |
| 4 | Звітність | |

За даними критеріями конкурентоспроможності (таблиця 6.10) проаналізовано переваги та недоліки стартап-проекту (таблиця 6.11).

Таблиця 6.11 – Переваги та недоліки стартап-проекту

| № п/п | Критерій конкурентоспроможності | Оцінка 1-30 | Порівняння існуючих рішень у конкурентів | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | -3 | -2 | -1 | 0 | 1 | 2 | 3 |
| 1 | Швидкість впровадження | 28 | + | | | | | | |
| 2 | Якість результатів роботи проекту | 30 | + | | | | | | |
| 3 | Додаткове налаштування | 25 | | | | + | | | |
| 4 | Звітність | 29 | | | | + | | | |
| 5 | Висока собівартість | 30 | | | | + | | | |

SWOT є останнім етапом аналізу запуску стартап-проекту на ринок, так як SWOT припускає, що сильні та слабкі сторони часто є внутрішніми (таблиця 6.11), тоді як можливості та загрози частіше є зовнішніми (таблиця 6.12). Сильні сторони: характеристики компанії або проекту, які дають їй перевагу над іншими. Слабкі сторони: характеристики, які ставлять компанію чи проект у невигідне становище порівняно з іншими. Можливості: елементи навколишнього середовища, які проект можуть використовувати на свою користь. Загрози: елементи середовища, які можуть створити проблеми для компанії або проекту.

Таблиця 6.12 – SWOT-аналіз

| Плюси: | Мінуси: |
|---|---|
| – висока якість звітності;<br>– легкість у процесі адаптації та налаштуванні продукту;<br>– простота у використанні;<br>– інновація. | – висока ціна продукту;<br>– для розробки потрібні висококооплачувані спеціалісти;<br>– потреба у адаптації і налаштуванні продукту під кожного замовника окремо. |

Продовження таблиці 6.12 – SWOT-аналіз

| Інші перспективи: | Інші ризики: |
| --- | --- |
| – масштабування продукту у різних сферах. | – відносно невеликий круг клієнтів;<br>– можливо з'являться інші компанії зі схожими розробками. |

Аналізуючи дані SWOT-аналізу можна припустити інші способи реалізації стартап проекту для виведення його на ринок та їх орієнтовний термін реалізації з огляду на потенційні проекти конкурентів.

Інші способи реалізації стартап проекту, перспективи отримання ресурсів та їх термін реалізації наведені в таблиці 6.13.

Таблиця 6.13 – Інші способи реалізації стартап проекту на ринку

| № п/п | Інші способи реалізації стартап проекту ринкової поведінки | Перспектива отримання ресурсів | Термін реалізації |
| --- | --- | --- | --- |
| 1 | Збільшення впізнаваності за допомогою маркетинг-відділів | Ймовірно | 3 місяці |
| 2 | Використання безкоштовних рішень для розробки стартап-проекту | Малоймовірно | Протягом розробки проекту |
| 3 | Вихід на ринок зі зменшеною ціною для збільшення кількості користувачів. Можливість використання лише певних функцій на вибір користувача | Малоймовірно | 6 місяців |

Роблячи висновок з даних аналізу, найвигіднішим варіантом для початку є саме 3 пункт таблиці 6.13.

## 6.4 Ринкова стратегія

Для оцінки ринкової діяльності та розробки ринкової політики потрібне вивчення ринкового оточення проекту: ринку, на якому працює компанія розробки, які сегменти ринку обслуговуються, клієнтів, конкурентів ринкового потенціалу суб'єктів ринку і постачальників.

У процесі аналізу ринку в першу чергу повинні бути визначені його продуктові і географічні кордони, суб'єкти ринку (продавці і покупці), оцінена ємність ринку, а також його структура, яка характеризується, зокрема, такими показниками, як частки господарюючих суб'єктів на ринку, кількісні і якісні показники структури ринку. Обов'язковий елемент аналізу ринку – аналіз ринкового потенціалу його суб'єктів.

Опис ЦГ (цільових груп) потенційних споживачів представлені у таблиці 6.14)

Таблиця 6.14 – Вибір ЦГ потенційних користувачів

| № п/п | ЦГ потенційних клієнтів | Зацікавленість введення проекту на ринок для споживача | Приблизний відсоток в межах ЦГ | Конкуренція в сегменті | Легкість входу у сегмент |
|---|---|---|---|---|---|
| 1 | Компанії інформаційних технологій | Висока зацікавленість | 75% | На старті проекту майже відсутня. З часом передбачене ймовірне збільшення | Низька складні-сть |

Продовження таблиці 6.14 – Вибір ЦГ потенційних користувачів

| № п/п | ЦГ потенційних клієнтів | Зацікавленість введення проекту на ринок для споживача | Приблизний відсоток в межах ЦГ | Конкуренція в сегменті | Легкість входу у сегмент |
|---|---|---|---|---|---|
| 2 | Адміністратори/ архітектори | Можлива зацікавленість у ЦГ, що націлені на наукові та офіційні теми. | 75-85% | На старті проекту майже відсутня. З часом передбачене ймовірне збільшення | Середня складні-сть |
| 3 | Офіційні сайти для навчання | Висока зацікавленість | 80% | На старті проекту майже відсутня. З часом передбачене ймовірне збільшення | Середня складність |
| Роблячи аналіз було обрано 1,2 групи | | | | | |

За результатами аналізу потенційних груп споживачів автори ідеї обирають ЦГ, для яких вони пропонуватимуть свій товар, та визначають стратегію охоплення ринку:

1) При стратегії масового (недиференційованого) маркетингу підприємство не враховує відмінності між сегментами і розглядає ринок як єдине ціле.

2) При стратегії диференційованого маркетингу компанія прагне охопити досить велику кількість сегментів ринку зі спеціально для них розробленими товарами (з поліпшеною якістю, специфічними сировинними матеріалами, особливими функціональними властивостями, оригінальним зовнішнім оформленням – дизайном, упаковкою і т.д.) і специфічної маркетинговою політикою.

3) Вибираючи стратегію концентрованого маркетингу, бізнес зосереджує свої зусилля і ресурси на одному сегменті ринку і пропонує товари саме для даної групи покупців. Це стратегія спеціалізації, в якій пропозиція, як правило, оригінально і розраховане «під клієнта», тому підприємство може встановлювати на свій товар досить високі ціни.

Тож для роботи в обраних сегментах ринку необхідно сформувати початкову стратегію розвитку (таблиця 6.15).

Таблиця 6.15 – Початкова стратегія конкурентної поведінки

| № п/п | Альтернатива розвитку | Стратегія охоплення ніші на ринку | Головні конкуренти стартап проекту відповідно до обраної альтернативи | Початкова стратегія розвитку |
|---|---|---|---|---|
| 1 | Вихід на ринок зі зменшеною ціною для збільшення кількості користувачів. Можливість використання лише певних функцій на вибір користувача | Якщо використовувати лише деякі функції, збільшується к-сть, клієнту не завжди потрібний повний пакет функцій. | На даний час не існує головних конкурентів. Дана ніша майже вільна. | Стратегія диференціації |

У таблиці 6.16 наведене визначення початкової стратегії конкурентної поведінки.

Таблиця 6.16 – Початкова стратегії конкурентної поведінки

| № п/п | Чи є проект «першопрохідцем» на ринку? | Чи буде компанія шукати нових споживачів, або забирати існуючих у конкурентів? | Чи буде компанія копіювати основні характеристики товару конкурента, і які? | Стратегія конкурент-ної поведінки |
|---|---|---|---|---|
| 1 | Ні | Буде шукати нових | Можливе вдосконалення існуючих характеристик | Стратегія лідера |

Проводячи аналіз потреб споживачів конкретних сегментів для компанії, що розробила продукт, та для товару (див. таблицю 6.5), а також залежно від обраної початкової стратегії розвитку (таблиця 6.15) та стратегії поведінки конкурентів (таблиця 6.16) розробляється стратегія позиціонування.

Стратегія позиціонування представлена у таблиці 6.17.

Таблиця 6.17 – Визначення стратегії позиціонування

| № п/п | Характеристики товару за вимогами цільової аудиторії | Початкова стратегія розвитку | Головні конкурентоспроможні позиції власного стартап-проекту | Вибір асоціацій, які мають сформувати комплексну позицію власного проекту (три ключових) |
|---|---|---|---|---|
| 1 | Базові знання Kubernetes | Стратегія диференціації | Звітність<br>Можливість додаткового налаштування<br>Простота впровадження<br>Легкість у розгортанні<br>Масштабованість | Управління конфігурацією додатків<br>Проектування систем на базі мікросервісної архітектури<br>Оптимізація витрат на Kubernetes |

## 6.5 Розроблення маркетингової програми стартап-проекту

Маркетингова програма – це розроблений на основі комплексних маркетингових досліджень стратегічний план-рекомендація щодо виробничо-збутової і науково-технічної діяльності фірми на певний період часу, покликаний забезпечити оптимальний варіант її майбутнього розвитку з урахуванням запитів споживачів і відповідно до висунутим цілям і стратегії [34].

У процесі розробки маркетингових програм враховуються множинні умови, перспективи і обмеження як в розвитку ринку, так і у внутрішньофірмовому розвитку, а також дію прямих і зворотних зв'язків з ринком, необхідність пристосування до

мінливим запитам ринку і активного впливу на формування та розширення ринкового попиту.

Сформуємо маркетингову концепцію продукту, який отримає споживач. Визначення головних переваг концепції можливого товару наведено у таблиці 6.18.

Таблиця 6.18 – Визначення головних переваг концепції можливого товару

| № п/п | Чинник | Вигода від товару | Головні переваги перед конкурентами |
|---|---|---|---|
| 1 | Інформативність та достовірність звіту | Висока якість системи | Технології |
| 2 | Зрозумілість та зручність | Легкість у конфігурації та налаштуванні системи | Якість наданих послуг |

Далі було розроблено трирівневу маркетингову модель товару. Детальний опис всіх рівнів моделі товару наведений у таблиці 6.19.

Таблиця 6.19 – Опис трьох рівнів моделі товару

<table>
<tr><th>Рівень</th><th colspan="3">Детальний опис</th></tr>
<tr><td>I. Товар за задумом</td><td colspan="3">Продукт у вигляді автоматизованої системи, у якої головна ціль – це ініціалізація та управління веб-додатками у Kubernetes кластерах. Використання для різних Kubernetes кластерів незалежно від середовища можливе за бажанням.</td></tr>
<tr><td>II. Товар у реальному виконанні</td><td>Властивості/характеристики</td><td>М/Нм</td><td>Вр/Тх /Тл/Е/Ор</td></tr>
</table>

Продовження таблиці 6.19 – Опис трьох рівнів моделі товару

| Рівень | Детальний опис | | |
|---|---|---|---|
| | 1. Період обслуговування з ліцензією | Рік | 1 |
| | 2. Легкість використання | | + |
| | 3. Наявність документації | | + |
| | 4. Кількість параметрів для конфігурації | Шт. | 20+<br>+ |
| | 5. Формування звітів | | + |
| | Стартап проект пройшов всі рівні та типи тестування, повністю відповідає вищезазначеним вимогам, які існують на ринку. | | |
| | Постачається в електронному вигляді на сайті виробника | | |
| | Марка: у розробці. Назва: у розробці | | |
| | Програмне забезпечення | | |
| | Програмне забезпечення | | |
| Потенційна послуга буде захищена від копіювання за допомогою таких функцій як патентування, а створення заявки на отримання патенту на винахід, щоб алгоритм роботи не був скопійований у майбутньому під іншим ім’ям. | | | |

Далі можна визначити цінові межі. Ними користуються при формуванні ціни на потенційний товар. Формування меж ціни представлено у таблиці 6.20.

Таблиця 6.20 – Формування меж встановлення ціни

| № п/п | Рівень цін на товари-замінники | Рівень цін на товари-аналоги | Рівень доходів ЦГ користувачів | Нижня та верхня межі встановлення ціни на послугу |
|---|---|---|---|---|
| 1. | - | 20 000 – 55 000 $ | Від 10 000 $/місяць | 10 000 – 25 000 $ |

У таблиці 6.21 наведено оптимальну СЗ (систему збуту), в межах якої приймаються рішення:

– проводити збут за допомогою внутрішніх ресурсів і каналів або залучати зовнішні ресурси (власна/залучена СЗ);
– дослідження та аналіз оптимальної глибини каналу збуту;
– залучення посередників.

Таблиця 6.21 – Визначення СЗ

| № п/п | Специфіка закупівельної поведінки потенційних клієнтів | Функції збуту, які виконує постачальник | Глибина каналу збуту | Оптимальна система збуту |
|---|---|---|---|---|
| 1. | Замовлення розробки послуги повністю чи частково (окремих функцій), оплата та встановлення | Інформування користувачів | Один рівень | Комбінована |

У таблиці 6.22 наведена концепція маркетингових комунікації.

Таблиця 6.22 – Визначення концепції маркетингових комунікацій

| № п/п | Специфіка поведінки потенційних клієнтів | Канали комунікацій, якими користуються потенційні клієнти | Ключові позиції, обрані для позиціонування | Завдання рекламного повідомлення | Концепція рекламного звернення |
|---|---|---|---|---|---|
| 1. | Сучасні компанії інформаційних технологій | Інтернет | Інноваційне рішення. Поєднання технологій. | Збільшення кількості клієнтів | Популярні SMM канали |

## 6.6 Висновки до розділу

У даному розділі був проведений аналіз потенційної маркетингової стратегії розвитку продукту пов’язаного з автоматизованою системою ініціалізації та розгортання веб-додатків у Kubernetes кластерах доцільність реалізації даного продукту.

Враховуючи результати аналізу стартап проекту, можна стверджувати, що ринкова капіталізація проекту є можливою. Зважаючи на те, що на ринку України немає аналогів, можна розпочинати саме з даної частини світу для старту розвитку. Особливо сприятливою цьому умовою є те, що на ринку України наявна велика кількість компаній у галузі IT, що працюють за аутсорс моделлю. Таким чином, починаючи з України через аутсорсингові компанії продукт має шанси на поширення по всьому світу, адже на світовому ринку кількість аналогів є мізерна.

Рентабельність готової системи є на достатньому рівні, проте не надто висока. Зумовлено це тим, що розробка системи може потребувати коригувань для конкретних замовників у разі доволі специфічних умов використання та наявності відповідних вимог. Крім того, для роботи зі спеціалістами з банківської сфери чи

сфери охорони здоров'я можуть знадобитися відповідні сертифікати безпеки, аудит для котрих є досить коштовним.

У ході розробки маркетингової програми було визначено сильні та слабкі сторони продукту, що дозволяє більш чітко формувати вимоги щодо подальшого розвитку та допоможе запобігти швидкій появі близьких конкурентів у найближчому майбутньому.

Подальша розробка продукту є доцільною з точки зору розвитку галузі галузі загалом, оскільки обране середовище розробки та компоненти є широко використовуваними нині та мають чудові перспективи у майбутньому. Такий вибір також дозволяє досить швидко та просто інтегрувати систему у вже існуючі проекти, що також збільшує потенціал для розвитку продукту.

# ВИСНОВКИ

У даній магістерській дисертації розроблена автоматизована система ініціалізації та управління веб-додатками у кластерах Kubernetes.

Аналіз переваг та недоліків мікросервісного підходу до розробки програмного забезпечення, що набуває ще більше широкого використання упродовж останніх кількох років показав, що питання оркестрації додатками, що запускаються у контейнерах Docker, є актуальним. Цей факт породжує велику потребу у швидких, зручних у використанні та надійних оркестраторах, серед яких Kubernetes є беззаперечним лідером. Проте, аналіз архітектури Kubernetes показав, що стандартний набір примітивів, котрими він оперує не є достатнім для задоволення потреб більшості розробників.

Дана розробка увібрала у себе переваги проаналізованих аналогів та виправила їх недоліки, що робить дану систему досить ефективною та простою у використанні. Зокрема, підсистема ініціалізації веб-додатків забезпечує високий рівень безпеки, адже інтегрована з Google Secret Manager, та виводить досить інформативні повідомлення у ході виконання, що дозволяє шивдко та просто знаходити несправності у роботі. У свою чергу, підсистема управління веб-додатків розширює набір примітив Kubernetes більш високорівневими абстракціями, що дозволяє пришвидшити та полегшити процес розробки контейнеризованих веб-додатків, особливо в умовах використання мікросервісної архітектури для побудови програмних платформ.

Обрані компоненти для реалізації системи, а саме мова програмування Go та фреймворк Kudo для розширення API Kubernetes є доцільним та вдалим, про що свідчить проведений вибір компонентів.

Аналіз комерційного ринку та розробка стартап проекту, пов’язаного з розробленою системою показав, що продовження роботи над розробкою даного продукту є доцільним, адже має гарні шанси на успішну капіталізацію на ринку України та, згодом, світу. Сприяє цьому присутність на ринку України великої

кількості компаній, що працюють за моделлю аутсорс та аутстаф, що допоможе забезпечити плавну та всеохоплюючу популяризацію розробленої системи за межами ринку України.

Розроблена система впроваджена на одному з проектів ТОВ «Стар Україна», що підтверджено відповідним актом впровадження. Акт впровадження свідчить про те, що використання даної системи є доцільним та ефективним. А саме, вона дозволяє більш швидко проводити тестування програмного забезпечення у різноманітних оточеннях та надає більш надійний та безпечний спосіб ініціалізації додатків. Також, названі переваги дозволяють проходити аудити з безпеки, наприклад ISO 27001, більш швидко.

Автоматизована система відповідає сформованим вимогам до розробки. Більш того, вона дозволяє легке внесення покращень та розширення функціоналу в разі потреби у майбутньому.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Secret Manager. Build more secure applications with Secret Manager [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/secret-manager.

2. Advanced Encryption Standart [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Advanced_Encryption_Standard.

3. 8 surprising facts about Docker adoption [Електронний ресурс] – Режим доступу до ресурсу: https://www.datadoghq.com/docker-adoption/.

4. Kubernetes and Container Security and Adoption Trends [Електронний ресурс] – Режим доступу до ресурсу: https://www.stackrox.com/kubernetes-adoption-security-and-market-share-for-containers/.

5. An Architect's Guide to Site Reliability Engineering [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/Arch-SRE.pdf.

6. Ролік О. І. Ролик А.И. Управление корпоративной ИТ-инфраструктурой / О. І. Ролік, С. Ф. Теленик, М. В. Ясочка., 2018. – 576 с. – (Наукова думка).

7. Building Microservices [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devopsasset/devopsreferencematerial/blob/master/Building_Microservices_Nginx.pdf.

8. Microservice Architecture. Aligning principles, practices, and culture [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/CA Technologies - OReilly Microservice Architecture eBook.pdf.

9. Trends in Applied Infrastructure & DevOps [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/451_2019_Trends_in_Applied_Infrastructure_&_DevOps_SLIM.PDF.

10. A Practical Guide to Continuous Delivery [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/A Practical Guide to Continuous Delivery.epub.

11. State of DevOps. Strategies for a New Economy [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/DORA-State of DevOps.pdf.

12. The dzone guide to devops: culture and process [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/9281682-dzone2018-researchguide-devops.pdf.

13. DevOps Handbook [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/Devops Handbook.

14. Mastering Kubernetes. Automating container deployment and management [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/Mastering Kubernetes - Gigi Sayfan.pdf.

15. Site Reliabiity Engineering. How google runs production systems [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devopsreferencematerial/blob/master/OReilly.Site.Reliability.Engineering.2016.3.pdf.

16. The State of Containers and the Docker Ecosystem 2015 [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/State_of_Containers_Ruxit_compressed_V2.pdf.

17. Building Cloud Native Apps Painlessly. The Prescriptive Guide to Kubernetes and Jenkins X [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devopsasset/devopsreferencematerial/blob/master/building_cloud_native_apps_painlessly.pdf.

18. Continuous Security: Implementing the Critical Controls in a DevOps Environment [Електронний ресурс] – Режим доступу до ресурсу: https://github.com/devops-asset/devops-reference-material/blob/master/continuous-security-implementing-critical-controls-devops-environment-36552.pdf.

19. Provider. Providers in the Terraform Registry [Електронний ресурс] – Режим доступу до ресурсу: https://www.terraform.io/docs/providers/index.html.

20. Introducing the Operator Framework: Building Apps on Kubernetes [Електронний ресурс] – Режим доступу до ресурсу: https://coreos.com/blog/introducing-operator-framework.

21. The What, Why, and How of a Microservices Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://medium.com/hashmapinc/the-what-why-and-how-of-a-microservices-architecture-4179579423a9.

22. Microservices. What are microservices [Електронний ресурс] – Режим доступу до ресурсу: https://www.redhat.com/en/topics/microservices/what-are-microservices.

23. Best practices for building containers [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/solutions/best-practices-for-building-containers.

24. GKE best practices: Designing and building highly available clusters [Електронний ресурс] – Режим доступу до ресурсу: https://cloud.google.com/blog/products/containers-kubernetes/best-practices-for-creating-a-highly-available-gke-cluster.

25. Golang basics – writing unit tests [Електронний ресурс] – Режим доступу до ресурсу: https://blog.alexellis.io/golang-writing-unit-tests/.

26. An Overview On Golang Programming Language [Електронний ресурс] – Режим доступу до ресурсу: https://masterofcode.com/blog/an-overview-on-golang-programming-language.

27. About Go Language – An Overview [Електронний ресурс] – Режим доступу до ресурсу: https://blog.learngoprogramming.com/about-go-language-an-overview-f0bee143597c.

28. Creating Custom Kubernetes Operators [Електронний ресурс] – Режим доступу до ресурсу: https://www.magalix.com/blog/creating-custom-kubernetes-operators.

29. Future of CRDs: Structual Schemas [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/blog/2019/06/20/crd-structural-schema/.

30. KUDO. What is KUDO. When would you use KUDO [Електронний ресурс] – Режим доступу до ресурсу: https://kudo.dev/docs/what-is-kudo.html#when-would-you-use-kudo.

31. Software as a service [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Software_as_a_service.

32. Managing Resources for Containers [Електронний ресурс] – Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/.

33. Porter’s five forces analysis [Електронний ресурс]: Режим доступу: https://en.wikipedia.org/wiki/Porter%27s_five_forces_analysis.

34. Маркетингова програма підприємства [Електронний ресурс]: Режим доступу: http://www.ukrainereferat.org/uaref-2263-1.html.