

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проектами
Кафедра інформаційних управляючих систем та технологій

"Допущений до захисту"

Завідувач кафедри ІУСТ



к.т.н, доц. Михелєв І.Л.

" ____ " _____ 2024 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти "магістр"

**на тему: "Дослідження та аналіз автоматизованих систем розгортання
вебзастосунків у хмарних середовищах"**

Виконала: студент групи 6144 м

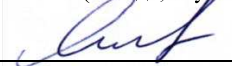

(підпис)

Міняйлов О.О.

Керівник роботи:

доцент кафедри ІУСТ, к.т.н.,

(посада, науковий ступінь вчене звання)


(підпис)

Михелєв І.Л.

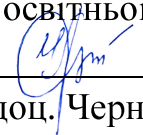
м. Миколаїв – 2024 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова
Навчально-науковий інститут комп'ютерних наук та управління проектами
Кафедра інформаційних управляючих систем та технологій
Спеціальність 126 "Інформаційні системи та технології"
Освітня програма "Інформаційні системи та технології"

"ЗАТВЕРДЖУЮ"

Гарант освітньої програми


к.т.н, доц. Чернова ЛБ.С.

" ____ " ____ 2024 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти "магістр"**

Студенту _____ Міняйлову Олегу Олексійовичу
(прізвище, ім'я, по батькові)

1. Тема роботи: Дослідження та аналіз автоматизованих систем розгортання вебзастосунків у хмарних середовищах

Керівник роботи Михелєв І.Л.

Затверджені наказом ректора № 1055-уч. від "08" жовтня 2024 року.

2. Термін подання роботи: 15 грудня 2024 р.

3. Вихідні дані до проекту (роботи) ДСТУ щодо обробки інформації, літературні джерела, технічна документація на існуючі аналоги систем, матеріали наукового стажування

4. Перелік питань, що належать до розробки (найменування розділів)

5. Перелік презентаційних матеріалів _____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Михелєв І.Л.	05.09.2024	22.12.2024
2	Михелєв І.Л.	23.12.2024	15.06.2024
3	Михелєв І.Л.	15.06.2024	25.09.2024
4	Михелєв І.Л.	25.09.2024	14.11.2024
5	Михелєв І.Л.	25.09.2024	14.11.2024
6	Михелєв І.Л.	25.09.2024	14.11.2024

7. Дата видачі завдання 05 вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів роботи	Примітка
1	Дослідження та аналіз предметної галузі	22.12.2024	
2	Розробка концепції	15.03.2024	
3	Розділ з розробки проектних рішень по системі (та її частинах)	15.06.2024	
4	Реалізація проекту та розробка робочої документації	25.09.2024	
5	Обчислення економічної ефективності	14.11.2024	
6	Вирішення питань охорони праці та навколишнього середовища	14.11.2024	
7	Оформлення пояснювальної записки	14.11.2024	
8	Подання магістерської роботи на перевірку на плагіат	16.11.2024	
9	Подання магістерської роботи рецензенту	20.11.2024	
10	Подання магістерської роботи на державний захист	15.12.2024	

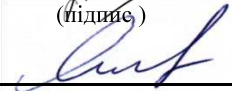
Студент


(підпис)

Міняйлов О.О.

(ПІБ)

Керівник роботи


(підпис)

Михелєв І.Л.

(ПІБ)

АНОТАЦІЯ

У роботі “Дослідження та аналіз автоматизованих систем розгортання вебзастосунків у хмарних середовищах” розглядається автоматизація процесів розгортання вебзастосунків у хмарних середовищах як ключовий аспект сучасної інфраструктури програмного забезпечення.

У дослідженні проаналізовано різні підходи до автоматизації, такі як інфраструктура як код (IaC), контейнеризація та оркестрація контейнерів.

Дипломна робота виконана на 105 сторінках, містить 13 рисунків, 13 таблиць, 1 додаток та список використаної літератури з 16 найменувань.

Ключові слова: розгортання вебзастосунків; Continious Integration/Continious Delivery (CI/CD); контейнеризація; оркестрація; хмарне середовище; пайплайн.

ABSTRACT

“Research and analysis of automated systems for deploying web applications in cloud environments” examines the automation of web application deployment processes in cloud environments as a significant aspect of modern software infrastructure.

The study analyses a range of automation approaches, including Infrastructure as Code (IaC), containerisation and container orchestration.

The thesis is executed on 105 pages, contains 13 figures, 13 tables, 1 appendice, and a list of references with 16 titles.

Keywords. web applications deployment; Continious Integration/Continious Delivery (CI/CD); containerisation; container orchestration; cloud environment; pipeline.

ЗМІСТ

ВСТУП

1 Аналіз існуючих рішень.....	3
1.1 Огляд інструментів контейнеризації.....	3
1.1.1 Docker.....	3
1.1.2 Podman.....	4
1.1.3 LXC (Linux Containers).....	4
1.1.4 Порівняння інструментів контейнеризації.....	5
1.2 Огляд інструментів для оркестрації.....	8
1.2.1 Kubernetes.....	8
1.2.2 Docker Swarm.....	9
1.2.3 OpenShift.....	11
1.2.4 Nomad.....	13
1.2.5 Порівняння інструментів оркестрації.....	14
1.3 Впровадження практик DevSecOps.....	15
1.3.1 Основні принципи DevSecOps.....	16
1.3.2 Статичний аналіз коду (SAST).....	16
1.3.3 Динамічний аналіз безпеки (DAST).....	18
1.3.4 Сканування контейнерних образів.....	19
1.3.5 Мережеві політики у Kubernetes.....	19
1.3.6 Секретний менеджмент.....	20
2 Структура системи.....	22
2.1 Вибір середовища розробки.....	22
2.1.1 PyCharm.....	22
2.1.2 IntelliJ IDEA.....	22
2.1.3 Visual Studio Code.....	23
2.2 Вибір хмарного середовища.....	23
2.2.1 Amazon Web Services (AWS).....	24
2.2.2 Google Cloud Platform (GCP).....	25

2.2.3 Microsoft Azure.....	25
2.2.4 DigitalOcean.....	26
2.2.5 Heroku.....	26
2.2.6 Порівняння хмарних платформ.....	27
2.3 Вибір інструментів безпеки.....	28
2.3.1 Основні критерії вибору інструментів.....	29
2.3.2 Огляд інструментів безпеки.....	29
2.3.3 Порівняння інструментів безпеки.....	31
2.4 Вибір інструментів контейнеризації.....	33
2.5 Вибір інструментів оркестрації.....	36
2.6 Вибір інструментів автоматизації.....	39
2.6.1 Огляд інструментів автоматизації.....	39
2.6.2 Порівняння інструментів автоматизації.....	41
2.7 Вибір інструментів моніторингу.....	43
2.7.1 Prometheus.....	43
2.7.2 Grafana.....	44
2.7.3 Використання Prometheus та Grafana.....	44
2.8 Інструменти та їх взаємодія.....	45
3 Функціональна модель і архітектура.....	48
3.1 Змістовна постановка задачі.....	48
3.2 Опис процесу діяльності.....	51
3.3 Опис функціональної моделі.....	51
3.4 Архітектура інфраструктури.....	54
3.4.1 GitHub Repository.....	56
3.4.2 GitHub Actions.....	57
3.4.3 Google Kubernetes Engine (GKE).....	58
3.4.4 Google Cloud DNS.....	59
3.4.5 Google Container Registry (GCR).....	59
3.5 Діаграма компонентів.....	60
3.6 Діаграма станів.....	62

3.7	Діаграма послідовності.....	65
3.8	Діаграма потоків даних.....	66
4	Технічний розділ.....	69
4.1	Керівництво користувача.....	69
4.1.1	Створення нової гілки за допомогою веб-інтерфейсу GitHub.....	69
4.1.2	Створення нової гілки за допомогою CLI.....	71
4.1.3	Запуск розгортання через GitHub Actions.....	72
4.2	Випробування системи.....	73
4.2.1	Мета випробувань.....	73
4.2.2	Системне тестування.....	73
4.3	Системні обмеження.....	75
5	Охорона праці.....	80
5.1	Загальні положення.....	80
5.2	Вимоги до робочого місця.....	80
5.3	Електробезпека.....	81
5.4	Пожежна безпека.....	82
5.5	Профілактика професійних захворювань.....	82
5.6	Екологічні аспекти.....	83
5.7	Рівень шуму.....	84

ВСТУП

Сучасні інформаційні технології значно змінили підходи до розробки, розгортання та обслуговування вебзастосунків. Зі зростанням складності систем і зростаючими вимогами до їхньої продуктивності, надійності та безпеки, виникає потреба в інноваційних рішеннях, які дозволяють автоматизувати ключові процеси, знижувати ризики та підвищувати ефективність. Контейнеризація, оркестрація та інтеграція безпеки в кожному етапі розробки стали важливими елементами сучасної інфраструктури DevOps.

Використання контейнеризації та систем оркестрації дозволяє створювати гнучкі, масштабовані та відмовостійкі системи. У той же час інтеграція принципів DevSecOps забезпечує безпеку на всіх етапах життєвого циклу застосунків: від написання коду до його розгортання у виробниче середовище. Комбінація цих технологій відкриває нові можливості для створення потужних і безпечних систем, що є особливо актуальним в умовах швидкого зростання кіберзагроз та постійних змін у вимогах до програмного забезпечення.

Мета роботи. Розробити автоматизовану систему розгортання вебзастосунків із використанням технологій контейнеризації, оркестрації та DevSecOps, яка забезпечує гнучкість, масштабованість та надійність, а також інтегрує механізми безпеки у весь процес CI/CD.

Предмет дослідження. Процеси контейнеризації, оркестрації та впровадження практик DevSecOps у системах автоматизованого розгортання.

Об'єкт дослідження. Інструменти контейнеризації (Docker), оркестрації (Kubernetes) та системи автоматизації CI/CD з інтеграцією елементів DevSecOps.

Завдання дослідження.

1. Розробити та реалізувати систему контейнеризації вебзастосунків із використанням Docker.
2. Інтегрувати Kubernetes для автоматизації оркестрації контейнерів, масштабування та забезпечення безвідмовної роботи.

3. Впровадити елементи DevSecOps для забезпечення безпеки на етапах розгортання:
 - a) Аналіз вихідного коду (SAST).
 - b) Сканування контейнерів на вразливості.
 - c) Налаштування мережевих політик у Kubernetes.
4. Налаштувати CI/CD пайплайн із автоматизованими етапами безпеки.
5. Провести тестування системи у різних сценаріях роботи та проаналізувати її продуктивність.

Результати роботи можуть бути використані для створення та управління вебзастосунками, що потребують високого рівня надійності, гнучкості та безпеки. Система підходить для застосування у великомасштабних інфраструктурах, де ключовими є швидкість і безпека розгортання.

У роботі будуть розглянуті сучасні підходи до контейнеризації та оркестрації, а також розроблено рішення, що забезпечує автоматизовану інтеграцію елементів безпеки в процеси розгортання вебзастосунків.

РОЗДІЛ 1. Аналіз існуючих рішень та технологій

Успішна реалізація інформаційної системи залежить від ретельного аналізу існуючих рішень, вибору оптимальних технологій та чіткого формулювання вимог до системи. У сучасних умовах контейнеризація, оркестрація та інтеграція DevSecOps стали невід’ємними складовими процесів автоматизації розгортання програмного забезпечення, що дозволяють досягти високої продуктивності, гнучкості та безпеки.

У цьому розділі буде розглянуто існуючі інструменти для контейнеризації та оркестрації, а також практики DevSecOps, які використовуються для інтеграції безпеки у процеси CI/CD. На основі аналізу сучасних технологій сформульовано концепцію інформаційної системи, яка враховує актуальні тенденції у розробці програмного забезпечення, такі як автоматизація процесів розгортання, забезпечення безпеки та масштабованості.

1.1 Огляд інструментів контейнеризації

Контейнеризація стала однією з ключових технологій у сучасній розробці програмного забезпечення. Вона дозволяє ізолювати застосунки та їхні залежності в легковагових контейнерах, що забезпечує їхню портативність і консистентність у різних середовищах. У цьому пункті розглянуто основні інструменти контейнеризації, їхній функціонал, переваги та недоліки.

1.1.1 Docker

Docker є провідним інструментом контейнеризації, який використовується для створення, розгортання та запуску контейнерів. Він став стандартом у цій галузі завдяки простоті використання та широким можливостям для ізоляції застосунків [2, 15].

Docker забезпечує такі ключові функціональні можливості:

- **Docker Images:** Docker використовує образи як базовий шаблон для створення контейнерів. Образи містять усі залежності, необхідні для запуску застосунків, включаючи операційну систему, бібліотеки та сам застосунок. Це дозволяє створювати стандартизовані середовища для розгортання.
- **Контейнери:** Контейнери Docker — це запущені екземпляри образів. Вони забезпечують ізоляцію застосунків від операційної системи та інших контейнерів, що дозволяє уникнути конфліктів залежностей.
- **Docker Hub:** Це хмарний реєстр для пошуку, завантаження та спільного використання образів. Він містить безліч готових образів, які можна використовувати у проєктах.
- **Docker Compose:** Інструмент для налаштування та управління багатоконтейнерними застосунками. Використовуючи YAML-файли, можна описати залежності та зв'язки між контейнерами.
- **Мережеві можливості:** Docker дозволяє створювати ізольовані мережі для зв'язку між контейнерами.

Переваги Docker:

1. Простота у використанні завдяки CLI-інструментам.
2. Портативність: Контейнери можуть працювати на будь-якій платформі, яка підтримує Docker Engine.
3. Широка екосистема: Docker Hub забезпечує доступ до тисяч готових образів.

Недоліки Docker:

1. Залежність від демон-процесу (dockerd), що може створювати ризики для безпеки.
2. Відсутність потужних вбудованих засобів для оркестрації контейнерів (для цього використовується Kubernetes або Docker Swarm).

1.1.2 Podman

Podman (POD Manager) — це сучасний інструмент для контейнеризації, який є бездемонною (daemonless) альтернативою Docker. Він був розроблений із наголосом на безпеку, простоту та сумісність з існуючими стандартами Docker.

Podman забезпечує такі ключові функціональні можливості:

- **Daemonless Architecture:** Podman працює без фонових процесів, які керують контейнерами, на відміну від Docker. Це знижує ризик експлуатації привілейованих процесів.
- **Сумісність із Docker:** Подібність команд Podman CLI до Docker CLI дозволяє розробникам легко переходити між цими інструментами.
- **Rootless Containers:** Podman дозволяє запускати контейнери без root-доступу, що підвищує безпеку.
- **Підтримка подів:** Podman дозволяє групувати контейнери у поди, що є аналогом Kubernetes Pod. Це спрощує управління застосунками, які мають залежності між контейнерами.

Переваги Podman:

1. Відсутність демонів, що знижує ризики безпеки.
2. Підтримка роботи без root-доступу.
3. Сумісність із стандартами OCI (Open Container Initiative).

Недоліки Podman:

1. Менша екосистема та відсутність потужного репозиторію, як Docker Hub.
2. Відсутність вбудованих інструментів оркестрації (Docker Swarm або Kubernetes).

1.1.3 LXC (Linux Containers)

LXC (Linux Containers) — це технологія контейнеризації на рівні операційної системи, яка забезпечує ізоляцію робочого середовища без

використання віртуалізації. LXC дозволяє створювати легковагові контейнери, які працюють безпосередньо на ядрі Linux, що робить їх продуктивними та ефективними.

LXC забезпечує такі ключові функціональні можливості:

- Системна контейнеризація: На відміну від Docker, який фокусується на застосунках, LXC забезпечує ізоляцію повноцінних системних середовищ. Контейнери LXC мають власні файлові системи, мережі та процеси, що дозволяє запускати повноцінні ОС у контейнері.
- Прямий доступ до ресурсів: LXC працює без шару віртуалізації, тому контейнери використовують ресурси хостової системи без додаткової накладної.
- Гнучке управління: LXC підтримує детальну конфігурацію контейнерів через текстові файли. Це дозволяє точно налаштовувати доступ до ресурсів, мережі та ізоляцію.
- Сумісність із ядром Linux: LXC інтегрується з інструментами Linux, такими як cgroups (контроль використання ресурсів) і namespaces (ізоляція процесів і мереж).

Переваги LXC:

1. Висока продуктивність: Контейнери LXC мають мінімальну накладну, оскільки вони працюють безпосередньо на ядрі хостової системи.
2. Повна ізоляція: Кожен контейнер має власне середовище, ізольоване від інших контейнерів і хоста.
3. Гнучкість у використанні: Можливість налаштовувати контейнери для широкого спектра завдань, від тестування до створення середовищ розробки.

Недоліки LXC:

1. Складність налаштування: У порівнянні з Docker, налаштування контейнерів LXC вимагає більше знань і часу.

2. Обмежена екосистема: LXC не має власного реєстру образів, як Docker Hub, і менш популярний у DevOps-спільноті.

3. Відсутність оркестрації: Для управління великою кількістю контейнерів потрібне додаткове програмне забезпечення, оскільки LXC не має вбудованих засобів для оркестрації.

LXC є корисним для створення легковагових середовищ ізольованої розробки або тестування. Проте через складність налаштування і відсутність вбудованих засобів оркестрації його роль обмежується специфічними сценаріями, де потрібна висока продуктивність і контроль над системним середовищем.

1.1.4 Порівняння інструментів контейнеризації

Для визначення найбільш підходящого інструменту контейнеризації для магістерської роботи проведено їх порівняння за ключовими параметрами.

Таблиця 1.1 - порівняння інструментів контейнеризації

Інструмент	Переваги	Недоліки	Рекомендація
Docker	Простота, популярність, екосистема Docker Hub	Залежність від демона, обмежений Swarm	Основний інструмент роботи
Podman	Безпека, відсутність демонів	Менша екосистема, відсутність оркестрації	Альтернатива для окремих проєктів
LXC	Висока продуктивність, гнучкість	Складність, обмежена підтримка сучасних пайплайнів	Для специфічних сценаріїв

На основі проведеного аналізу для реалізації магістерської роботи обрано Docker. Його переваги, зокрема простота використання, широка підтримка спільноти та інтеграція з сучасними інструментами оркестрації, роблять його

найкращим вибором для контейнеризації вебзастосунків у рамках даного проєкту.

1.2 Огляд інструментів для оркестрації

Оркестрація контейнерів є критично важливою складовою для управління масштабними застосунками, що складаються з багатьох контейнерів. Інструменти оркестрації забезпечують автоматизацію процесів розгортання, моніторингу, масштабування та відновлення контейнеризованих сервісів. У цьому розділі розглянуто найпопулярніші інструменти оркестрації, їх функціонал, переваги, обмеження та доцільність використання у магістерській роботі.

1.2.1 Kubernetes

Kubernetes (K8s) — це найпоширеніший інструмент оркестрації, створений компанією Google і підтримуваний спільнотою Cloud Native Computing Foundation (CNCF). Kubernetes став стандартом у сфері управління контейнерами завдяки своїй масштабованості, функціональності та гнучкості [3, 15].

Основні функціональні можливості Kubernetes:

- Автоматичне масштабування: Kubernetes підтримує горизонтальне (додавання подів) і вертикальне (збільшення ресурсів для подів) масштабування. Це дозволяє оптимально використовувати ресурси залежно від навантаження[15].
- Відмовостійкість: Kubernetes автоматично перезапускає поди у разі збоїв, переміщує їх на інші вузли та контролює їх стан.
- Мережеві політики: Kubernetes забезпечує налаштування мережевих політик для контролю взаємодії між подами.

- Розгортання оновлень: Kubernetes підтримує rolling updates, які дозволяють оновлювати застосунки без перерв у роботі.

Переваги Kubernetes:

1. Масштабованість: Підтримка великих кластерів і розподілених систем.
2. Гнучкість: Широкі можливості для налаштування мереж, сховищ і політик безпеки.
3. Інтеграція з хмарними платформами (GCP, Google Cloud, Azure).

Недоліки Kubernetes:

1. Складність налаштування: Вимагає глибокого розуміння для коректної роботи.
2. Високі вимоги до ресурсів для роботи кластера.

1.2.2 Docker Swarm

Docker Swarm — це інструмент для оркестрації контейнерів, вбудований у Docker. Він забезпечує управління кластерами Docker-контейнерів, дозволяючи легко масштабувати, балансувати навантаження і забезпечувати високу доступність для контейнеризованих застосунків. На відміну від Kubernetes, Docker Swarm зосереджується на простоті та легкості використання.

Docker Swarm забезпечує такі ключові функціональні можливості:

- Управління кластерами: Docker Swarm дозволяє об'єднувати кілька вузлів у кластер. Один або кілька вузлів стають менеджерами (managers), які керують кластером, тоді як інші вузли виконують роль робітників (workers).
- Автоматичне масштабування: Користувач може легко збільшувати або зменшувати кількість контейнерів (званих сервісами) у кластері за допомогою команди `docker service scale`.

- Балансування навантаження: Docker Swarm автоматично розподіляє вхідні запити між контейнерами, які працюють на різних вузлах, забезпечуючи рівномірний розподіл навантаження.
- Захист і безпека: Docker Swarm забезпечує захищену комунікацію між вузлами через TLS, а також підтримує функцію шифрування даних у кластері.
- Легка інтеграція: Docker Swarm інтегрований у Docker CLI, що дозволяє використовувати знайомі команди Docker для управління оркестрацією.

Переваги Docker Swarm:

1. Простота використання: Інтуїтивно зрозумілий процес налаштування кластера та команд для управління.
2. Легка інтеграція: Працює безпосередньо у Docker, що усуває потребу в додаткових інструментах.
3. Балансування навантаження: Вбудована функція рівномірного розподілу запитів між контейнерами.
4. Мінімальні вимоги до ресурсів: Docker Swarm має менші вимоги до системних ресурсів порівняно з Kubernetes.

Недоліки Docker Swarm:

1. Обмежений функціонал: Docker Swarm пропонує базову оркестрацію, яка менш гнучка у порівнянні з Kubernetes.
2. Менша підтримка спільноти: У порівнянні з Kubernetes, Docker Swarm має меншу базу користувачів і обмежену кількість ресурсів для навчання.
3. Відсутність розширених функцій: Наприклад, складні сценарії розгортання (canary deployment, blue-green deployment) або інтеграція з DevSecOps потребують додаткових налаштувань.

Docker Swarm може бути корисним для невеликих і середніх проєктів, які не вимагають складної оркестрації. Для магістерської роботи Docker Swarm може бути використаний у тестових середовищах або для демонстрації базових можливостей оркестрації, але для виробничих сценаріїв пріоритет надається Kubernetes, який має більш потужний функціонал.

1.2.3 OpenShift

OpenShift — це контейнерна платформа від компанії Red Hat, побудована на основі Kubernetes. Вона надає додаткові інструменти для автоматизації, безпеки та управління контейнеризованими застосунками, орієнтуючись на корпоративні середовища. OpenShift поєднує функціональність Kubernetes із додатковими можливостями для DevOps і DevSecOps.

OpenShift забезпечує такі ключові функціональні можливості:

- Платформа як сервіс (PaaS): OpenShift надає інтегроване середовище для розробки, тестування, розгортання та управління застосунками. Це спрощує роботу розробників та адміністраторів.
- Розширена безпека: OpenShift включає інструменти для забезпечення відповідності стандартам безпеки:
 - Контроль доступу на основі ролей (RBAC).
 - SELinux для ізоляції контейнерів.
 - Підтримка секретного менеджменту для безпечного зберігання конфіденційних даних.
- Інтеграція DevOps та DevSecOps: Платформа надає інструменти для CI/CD (наприклад, OpenShift Pipelines), інтегровані з Kubernetes і Jenkins. Це дозволяє автоматизувати процеси розробки та тестування.
- Підтримка багатохмарності: OpenShift підтримує розгортання у публічних, приватних і гібридних хмарних середовищах, таких як GCP, Google Cloud, Azure, а також у локальних датацентрах.

- Інтеграція з Red Hat екосистемою: OpenShift інтегрується з іншими продуктами Red Hat, такими як Ansible для автоматизації та Red Hat Quay для управління контейнерними образами.

Переваги OpenShift:

1. Повноцінне середовище PaaS: Спрощує управління життєвим циклом застосунків.
2. Розширені інструменти безпеки: Забезпечує високий рівень ізоляції контейнерів та управління доступом.
3. Підтримка корпоративних стандартів: OpenShift адаптований для використання у великих компаніях, які мають складні вимоги до безпеки та відповідності стандартам.
4. Інструменти для автоматизації: Вбудована підтримка CI/CD через OpenShift Pipelines.

Недоліки OpenShift:

1. Висока вартість: OpenShift є комерційним продуктом, і вартість ліцензій може бути значною.
2. Складність налаштування: Вимагає більше часу та ресурсів для розгортання та адміністрування у порівнянні з Kubernetes.
3. Ресурсоємність: OpenShift має вищі вимоги до системних ресурсів через додаткові сервіси та функції.

OpenShift підходить для проєктів, де необхідно забезпечити високий рівень автоматизації, відповідність корпоративним стандартам безпеки та інтеграцію з DevSecOps. Однак через високу вартість і складність налаштування OpenShift менш придатний для проєктів із обмеженим бюджетом або для невеликих команд.

1.2.4 Nomad

Nomad — це універсальний інструмент для оркестрації від компанії HashiCorp, який підтримує управління не лише контейнеризованими застосунками, але й іншими типами робочих завдань, такими як віртуальні машини, скрипти, і бінарні файли. Nomad позиціонується як легковага альтернатива Kubernetes, яка зосереджується на простоті використання та мультиплатформенності.

Nomad забезпечує такі ключові функціональні можливості:

- **Мультиплатформенність:** На відміну від Kubernetes, який обмежується контейнерами, Nomad може працювати з різними типами робочих навантажень, включаючи Docker-контейнери, Java-програми, виконувані файли на Windows/Linux та інші.
- **Простота розгортання:** Nomad має легку архітектуру, що складається з одного бінарного файлу. Це спрощує налаштування та використання навіть у невеликих командах.
- **Гнучке управління ресурсами:** Використовує модель планування на основі загальних ресурсів (CPU, пам'ять, диск), дозволяючи ефективно розподіляти завдання між вузлами.
- **Інтеграція з іншими продуктами HashiCorp:** Nomad безшовно інтегрується з Consul для служби DNS та автоматизації мережі, а також із Vault для управління секретами.
- **Відмовостійкість:** Nomad підтримує роботу в кластерному середовищі з автоматичним переміщенням завдань між вузлами у разі збоїв.
- **Масштабованість:** Nomad може керувати десятками тисяч завдань і забезпечувати балансування навантаження між вузлами.

Переваги Nomad:

1. Простота: Легка у розгортанні архітектура з мінімальними вимогами до інфраструктури.
2. Гнучкість: Підтримка різних типів навантажень, що робить його універсальним для різних завдань.
3. Інтеграція: Глибока інтеграція з іншими продуктами HashiCorp (Consul, Vault).
4. Легковаговість: Порівняно з Kubernetes, Nomad має низькі вимоги до ресурсів.

Недоліки Nomad:

1. Обмежена екосистема: У порівнянні з Kubernetes, Nomad має меншу кількість додаткових модулів і плагінів.
2. Менша популярність: Nomad має обмежену спільноту користувачів, що може ускладнювати пошук рішень для специфічних завдань.
3. Відсутність нативної підтримки CI/CD: Для впровадження CI/CD необхідно використовувати сторонні інструменти.

Nomad може бути корисним для невеликих або середніх проєктів, де потрібна проста оркестрація і мультиплатформенність. Його легковагова архітектура дозволяє швидко розгортати рішення з мінімальними витратами ресурсів. Однак через меншу популярність і обмежену екосистему він менш придатний для складних і масштабних завдань, які потребують розширеної підтримки.

1.2.5 Порівняння інструментів оркестрації

Для визначення найбільш підходящого інструменту оркестрації для магістерської роботи проведено їх порівняння за ключовими параметрами.

Таблиця 1.2 - порівняння інструментів оркестрації

Інструмент	Основні переваги	Обмеження	Рекомендації з використання

Kubernetes	Масштабованість, гнучкість, багатофункціональність	Висока складність, вимоги до ресурсів	Основний вибір для великих проектів
Docker Swarm	Простота, інтеграції	Обмежений функціонал	Для невеликих проектів
OpenShift	Підтримка PaaS, розширена безпека	Висока вартість, складність	Для корпоративних застосунків
Nomad	Мультиплатформеність, інтеграція з HashiCorp	Менша популярність	Для специфічних сценаріїв

Для реалізації магістерської роботи обрано Kubernetes, оскільки він:

- Забезпечує повну гнучкість і масштабованість.
- Має широку підтримку спільноти та інтеграцію з DevSecOps.
- Є стандартом у сучасному управлінні контейнеризованими застосунками.

Обраний інструмент дозволяє реалізувати складні сценарії оркестрації, такі як автоматичне масштабування, забезпечення відмовостійкості та інтеграція з практиками безпеки.

1.3 Впровадження практик DevSecOps

У сучасному світі розробки програмного забезпечення важливо інтегрувати безпеку на всіх етапах життєвого циклу, починаючи від написання коду і закінчуючи його розгортанням у виробничому середовищі. DevSecOps (Development, Security, Operations) — це підхід, який інтегрує безпеку у практики DevOps, забезпечуючи виявлення та усунення вразливостей на ранніх стадіях.

У цьому розділі розглядаються ключові аспекти DevSecOps, інструменти для забезпечення безпеки на етапах CI/CD і кращі практики впровадження цього підходу.

1.3.1 Основні принципи DevSecOps

DevSecOps базується на інтеграції безпеки у всі етапи розробки, використовуючи автоматизовані перевірки коду, сканування контейнерів і налаштування мережевих політик. Цей підхід дозволяє знижувати ризики кіберзагроз [1, 13].

DevSecOps базується на таких ключових принципах:

1. Інтеграція безпеки у всі етапи розробки - забезпечення безпеки не тільки у продакшн-середовищі, а й під час розробки та тестування.
2. Автоматизація перевірок безпеки - використання інструментів, які автоматично сканують код, контейнерні образи та інфраструктуру на наявність вразливостей.
3. Постійне вдосконалення - аналіз вразливостей і розробка політик для їх попередження у майбутньому.

1.3.2 Статичний аналіз коду (SAST)

Статичний аналіз коду (SAST — Static Application Security Testing) є важливим етапом у забезпеченні безпеки програмного забезпечення. Цей підхід дозволяє виявляти вразливості у вихідному коді ще до його виконання, що дає змогу уникнути багатьох проблем безпеки та знижує витрати на виправлення помилок на пізніх етапах розробки.

Основні принципи SAST:

1. Аналіз вихідного коду: SAST перевіряє вихідний код програми або його проміжне представлення (наприклад, байт-код) без виконання програми.
2. Раннє виявлення вразливостей: Аналіз виконується під час розробки або на етапі збирання, що дозволяє виявляти проблеми на ранніх стадіях.
3. Автоматизація процесу: Більшість інструментів SAST інтегруються у CI/CD пайплайни, що забезпечує автоматичний аналіз коду при кожному коміті.

Ключові функціональні можливості SAST:

1. Виявлення поширених вразливостей:

- a) SQL-ін'єкції.
 - b) XSS (Cross-Site Scripting).
 - c) Незахищені змінні.
 - d) Використання застарілих або небезпечних бібліотек.
2. Аналіз відповідності стандартам: Перевірка коду на відповідність стандартам безпеки, таким як OWASP, CERT, NIST.
 3. Генерація звітів: Інструменти SAST надають докладні звіти про знайдені вразливості, включаючи їхній рівень критичності та рекомендації щодо усунення.
 4. Інтеграція з CI/CD: Підтримка автоматизації аналізу коду у процесах безперервної інтеграції (наприклад, Jenkins, GitHub Actions).

Популярні інструменти SAST:

SonarQube

SonarQube — один із найпопулярніших інструментів для статичного аналізу коду. Він підтримує велику кількість мов програмування та інтегрується з багатьма DevOps-інструментами.

Особливості:

- Підтримує понад 25 мов програмування.
- Виявляє вразливості, помилки та дублікацію коду.
- Інтегрується з GitHub, GitLab, Jenkins та іншими системами CI/CD.
- Надає звіти в режимі реального часу через веб-інтерфейс.

Приклад використання:

1. Інтеграція SonarQube у CI/CD пайплайн.
2. Запуск аналізу при кожному пуші в репозиторій.
3. Виведення результатів аналізу у вигляді звітів із деталізацією проблем.

Checkmarx

Checkmarx — це комерційний інструмент SAST, який фокусується на пошуку вразливостей безпеки у великих проєктах.

Особливості:

- Підтримка широкого спектра мов програмування та фреймворків.
- Глибокий аналіз коду, враховуючи залежності між компонентами.
- Генерація звітів про ризики відповідно до стандартів OWASP та NIST.

Приклад використання: використання Checkmarx для перевірки коду на відповідність вимогам безпеки перед виходом у продакшн.

1.3.3 Динамічний аналіз безпеки (DAST)

DAST (Dynamic Application Security Testing) — це метод тестування, який перевіряє застосунок під час його виконання. Основна мета — виявити вразливості, пов'язані з логікою роботи програми, ін'єкціями SQL, XSS та іншими атаками.

Інструменти для DAST:

- OWASP ZAP: Безкоштовний інструмент для автоматизованого сканування вебзастосунків.
- Burp Suite: Потужний комерційний інструмент для аналізу безпеки.

Приклад впровадження:

1. Розгортання тестового середовища вебзастосунку.
2. Використання OWASP ZAP для автоматизованого сканування URL-адрес.
3. Генерація звітів про знайдені вразливості та їх подальше усунення.

1.3.4 Сканування контейнерних образів

Сканування контейнерних образів є необхідною складовою DevSecOps, оскільки воно дозволяє виявляти вразливості на ранніх етапах. Наприклад, Google Cloud Artifact Registry дозволяє зберігати та перевіряти образи на відповідність стандартам безпеки [9].

Контейнерні образи є важливою складовою DevOps-середовища, але вони можуть містити вразливості, якщо базуються на застарілих або небезпечних версіях бібліотек.

Інструменти для сканування:

- Trivy: Легкий і швидкий інструмент для перевірки Docker-образів на вразливості.
- Clair: Інструмент для аналізу контейнерів на основі відомих баз даних CVE.

Приклад впровадження:

1. Інтеграція Trivy у CI/CD пайплайн для автоматичного сканування образів перед розгортанням.
2. Виявлення застарілих бібліотек або компонентів у базових образах.
3. Оновлення небезпечних компонентів до безпечних версій.

1.3.5 Мережеві політики у Kubernetes

Kubernetes дозволяє використовувати Network Policies для ізоляції мережевої взаємодії між подами. Це запобігає несанкціонованому доступу між компонентами системи.

Приклад реалізації:

1. Налаштування політик у YAML-файлі для обмеження доступу між подами.
2. Використання плагінів для мережі (наприклад, Calico) для реалізації правил.

1.3.6 Секретний менеджмент

У DevSecOps важливо забезпечити безпечне зберігання конфіденційної інформації (ключів, токенів, паролів). Використання спеціалізованих інструментів дозволяє запобігти витоку цих даних.

Інструменти для секретного менеджменту:

- HashiCorp Vault: Потужний інструмент для шифрування і безпечного зберігання секретів.
- Kubernetes SGCRets: Легка інтеграція для управління секретами у Kubernetes.

Висновок до розділу

Аналіз існуючих інструментів контейнеризації, оркестрації та практик DevSecOps дозволив визначити ключові технології, необхідні для успішної реалізації сучасних інформаційних систем. Контейнеризація, представлена такими інструментами, як Docker, Podman та LXC, забезпечує ізоляцію, портативність і стандартизацію середовищ для розгортання. Docker був обраний основним інструментом завдяки його простоті використання, широкій екосистемі та підтримці спільноти.

Оркестрація контейнерів, зокрема через Kubernetes, Docker Swarm, OpenShift і Nomad, дозволяє автоматизувати управління застосунками, забезпечуючи масштабованість, відмовостійкість і гнучкість. Kubernetes, завдяки своїй багатofункціональності та інтеграції з DevSecOps, став основним вибором для магістерської роботи.

Інтеграція DevSecOps забезпечує автоматизацію безпекових процесів на всіх етапах життєвого циклу програмного забезпечення. Практики, такі як статичний (SAST) і динамічний (DAST) аналіз безпеки, сканування контейнерів, мережеві політики та секретний менеджмент, дозволяють підвищити рівень безпеки системи.

Обрані інструменти та підходи забезпечують гнучкість, масштабованість і безпеку процесів автоматизації розгортання, відповідаючи сучасним вимогам

до хмарних середовищ. Це створює надійну основу для реалізації концепції інформаційної системи у рамках проекту.

РОЗДІЛ 2. СТРУКТУРА СИСТЕМИ

Розробка сучасних інформаційних систем потребує чітко визначеної архітектури, яка забезпечує продуктивність, гнучкість, масштабованість та безпеку. У цьому розділі описано структуру системи, її основні компоненти, взаємодію інструментів.

2.1 Вибір середовища розробки

Одними з найпопулярніших середовищ розробки наразі є Visual Studio Code, PyCharm та IntelliJ IDEA. Далі буде розглянуто кожне з них.

2.1.1 PyCharm

PyCharm - це інтегроване середовище розробки (IDE), спеціально розроблене для роботи з мовою програмування Python. Воно надає розширені можливості розробки та підтримує весь життєвий цикл розробки Python-програм.

2.1.2 IntelliJ IDEA

IntelliJ IDEA - це потужне інтегроване середовище розробки (IDE), призначене для роботи з різними мовами програмування, зокрема Java, Kotlin, Scala, Groovy та іншими мовами, що працюють на платформі Java Virtual Machine (JVM).

2.1.3 Visual Studio Code

Visual Studio Code (VSCode) - це безкоштовний, легкий у використанні та розширюваний текстовий редактор, розроблений компанією Microsoft. Він призначений для розробки різних типів програмного забезпечення та підтримує різні мови програмування. VSCode було обрано серед інших середовищ розробки, оскільки це, на відміну від PyCharm та IntelliJ IDEA, розширюваний

текстовий редактор, а не повноцінне середовище розробки, з чого випливають наступні особливості Visual Studio Code:

- VSCode має вбудовану підтримку систем контролю версій, таких як Git. Що дозволяє легко працювати з репозиторіями, переглядати зміни, виконувати коміти та злиття, а також розв'язувати конфлікти, не виходячи з редактора.
- VSCode відомий своєю швидкістю та продуктивністю. Він має мінімальний запуск та завантаження часу, швидку відгуки на введення та ефективну роботу з великими проєктами.

VSCode має широкий набір розширень, що дозволяють налаштувати та розширити його функціональність. Розширення доступні для підтримки різних мов програмування, фреймворків, систем контролю версій, інструментів розробки та багатьох інших сценаріїв. Саме такі розширення допоможуть полегшити роботу з GitHub, GitHub Actions та Docker.

2.2 Вибір хмарного середовища

У розробці сучасних інформаційних систем хмарні середовища стали ключовим інструментом для забезпечення продуктивності, масштабованості та безпеки. Обрання оптимальної хмарної платформи є важливим рішенням, що впливає на ефективність функціонування системи. У цьому розділі буде розглянуто основні хмарні платформи та обґрунтовано вибір середовища для магістерської роботи.

2.2.1 Amazon Web Services (AWS)

AWS — це найбільша хмарна платформа, яка пропонує понад 200 послуг для обчислень, зберігання даних, штучного інтелекту, аналітики, безпеки тощо. AWS використовується багатьма провідними компаніями світу завдяки своїй потужності та гнучкості [5].

Основні послуги AWS для проєкту:

- Amazon EC2 — обчислювальні інстанси для запуску застосунків із високою продуктивністю.
- Amazon S3 — хмарне сховище для зберігання великих обсягів даних.
- Amazon RDS — керовані бази даних, зокрема MySQL, PostgreSQL, SQL Server.
- Amazon EKS — сервіс для оркестрації контейнерів на основі Kubernetes.
- AWS CloudFormation — інструмент для автоматизації налаштування інфраструктури.

Переваги:

1. Широкий спектр сервісів: Найбільша кількість інтегрованих рішень для будь-яких завдань.
2. Глобальна доступність: Сервери у понад 30 регіонах по всьому світу.
3. Інструменти безпеки: AWS IAM, KMS, CloudTrail, які забезпечують надійний захист даних.
4. Масштабованість: Автоматичне збільшення або зменшення ресурсів залежно від навантаження.

Недоліки:

1. Висока складність для новачків через широкий вибір сервісів.
2. Може бути дорожчим, ніж інші платформи, для невеликих проєктів.

2.2.2 Google Cloud Platform (GCP)

GCP — це хмарна платформа, яка надає сервіси для розробників із акцентом на швидкість і продуктивність. Вона має одну з найшвидших мережевих інфраструктур [4].

Основні послуги GCP для проєкту:

- Google Kubernetes Engine (GKE): Один із найкращих сервісів для управління контейнерами.
- App Engine: Платформа для автоматизації розгортання застосунків.
- BigQuery: Аналітика великих даних.

Переваги:

1. Інтеграція з Kubernetes: GKE забезпечує високий рівень автоматизації та зручності [3, 10].

2. Оптимізована мережа: Надшвидкі сервери та низька затримка.

3. Просте ціноутворення: Гнучкі тарифи, включаючи безкоштовний план.

Недоліки:

1. Обмежений вибір сервісів порівняно з GCP.

2. Менша популярність серед великих організацій.

2.2.3 Microsoft Azure

Microsoft Azure — це хмарна платформа з великою кількістю послуг для корпоративних застосунків [6]. Вона інтегрується з продуктами Microsoft, такими як Active Directory та Microsoft 365.

Основні послуги Azure для проєкту:

- Azure Kubernetes Service (AKS): Сервіс для оркестрації контейнерів.

- Azure DevOps: Інтегрована платформа для CI/CD.

- Azure Blob Storage: Масштабоване хмарне сховище.

Переваги:

1. Гібридна інфраструктура: Підтримка локальних і хмарних середовищ.

2. Розширені засоби безпеки: Інтеграція з Azure Active Directory.

3. Підтримка Windows: Ідеально для проєктів, побудованих на технологіях Microsoft.

Недоліки:

1. Вища вартість для складних конфігурацій.

2. Складність налаштування для малих команд.

2.2.4 DigitalOcean

DigitalOcean — проста хмарна платформа, яка орієнтована на розробників і пропонує зрозумілий інтерфейс для швидкого запуску проєктів.

Основні послуги DigitalOcean для проєкту:

- Droplets: Віртуальні сервери для обчислень.
- Kubernetes Service: Просте налаштування кластерів Kubernetes.
- Spaces: Масштабоване сховище для даних.

Переваги:

1. Простота у використанні: Інтуїтивно зрозумілий інтерфейс.
2. Прозора модель ціноутворення: Зручна для невеликих бюджетів.
3. Швидкий запуск середовища: Легкість у налаштуванні.

Недоліки:

1. Обмежений набір сервісів для великих проєктів.
2. Менша підтримка корпоративних сценаріїв.

2.2.5 Heroku

Heroku — це хмарна платформа, яка дозволяє швидко запускати та розгортати вебзастосунки без необхідності адміністрування інфраструктури.

Основні послуги Heroku для проєкту:

- Dynos: Віртуальні контейнери для виконання застосунків [7].
- Add-ons: Плагіни для додаткових функцій, таких як бази даних.

Переваги:

1. Простота використання: Мінімальні вимоги до налаштування.
2. Інтеграція з GitHub: Зручність у роботі з CI/CD.

Недоліки:

1. Висока вартість при масштабуванні.
2. Обмежена кастомізація.

2.2.6 Порівняння хмарних платформ

Для ефективного вибору хмарного середовища було проаналізовано п'ять провідних платформ: Amazon Web Services (AWS), Google Cloud Platform (GCP),

Microsoft Azure, DigitalOcean та Heroku. Кожна з них має свої унікальні переваги, які роблять їх придатними для різних типів проєктів. Далі здійснено порівняння платформ за основними критеріями, включаючи функціональність, масштабованість, безпеку, вартість та інтеграцію з DevOps.

Таблиця 2.1 — Порівняльна таблиця хмарних платформ

Платформа	Переваги	Недоліки	Рекомендація
GCP	Інтеграція з Kubernetes, глобальна доступність, потужні інструменти безпеки	Складність налаштування,	Для масштабованих і комплексних проєктів
GCP	Інтеграція з Kubernetes, швидкість	Обмежений вибір сервісів	Для проєктів з орієнтацією на Kubernetes
Azure	Гібридна інфраструктура, корпоративна безпека	Висока вартість	Для корпоративних середовищ
DigitalOcean	Простота, прозоре ціноутворення	Обмежені можливості для великих проєктів	Для невеликих і середніх проєктів
Heroku	Легкість у використанні	Висока вартість при масштабуванні	Для малих або демонстраційних проєктів

На основі аналізу, для реалізації магістерської роботи обрано Amazon Web Services (GCP). Основними причинами цього вибору є:

1. Широка екосистема сервісів: GCP забезпечує всі необхідні інструменти для розгортання, моніторингу, масштабування та управління застосунками.

2. Глобальна доступність: GCP має найбільшу кількість датацентрів у світі, що гарантує мінімальну затримку та високу доступність.
3. Масштабованість: Автоматичне розподілення ресурсів дозволяє ефективно працювати в умовах змінного навантаження.
4. Інструменти безпеки: GCP пропонує потужні рішення для управління доступом і захисту даних.
5. Підтримка Kubernetes: Amazon GKE дозволяє легко інтегрувати систему з Kubernetes.

2.3 Вибір інструментів безпеки

Забезпечення безпеки на всіх етапах життєвого циклу програмного забезпечення є важливим аспектом розробки сучасних інформаційних систем. Інтеграція інструментів безпеки в процеси DevOps (DevSecOps) дозволяє автоматично виявляти вразливості, знижувати ризики та забезпечувати відповідність стандартам безпеки. У цьому розділі буде обрано основні інструменти для статичного (SAST) і динамічного (DAST) аналізу, сканування контейнерних образів, управління секретами та забезпечення мережевої безпеки.

2.3.1 Основні критерії вибору інструментів

При виборі інструментів безпеки враховувалися такі критерії:

1. Інтеграція з DevOps-пайплайнами — підтримка CI/CD і автоматизації процесів.

2. Можливість статичного та динамічного аналізу — виявлення вразливостей у коді та під час виконання застосунку.

3. Сканування контейнерів — аналіз базових образів і контейнерів на наявність вразливих компонентів.

4. Управління секретами — безпечне зберігання конфіденційних даних (паролів, ключів тощо).

5. Простота впровадження — легкість налаштування та інтеграції в існуючу інфраструктуру.

2.3.2 Огляд інструментів безпеки

1. SonarQube

SonarQube — це платформа, яка допомагає розробникам знаходити помилки, вразливості та покращувати якість коду. Вона підтримує багато мов програмування, інтегрується з популярними інструментами DevOps, такими як Jenkins чи GitHub Actions, і автоматично перевіряє код у CI/CD пайплайнах. Основні її можливості включають пошук поширених вразливостей, наприклад SQL-ін'єкцій, відповідність стандартам безпеки та аналіз якості коду. SonarQube надає зручний веб-інтерфейс для перегляду звітів, але потребує серверного середовища для роботи. Це потужний інструмент для підвищення надійності програмного забезпечення.

2. OWASP ZAP

OWASP ZAP — це безкоштовний інструмент для тестування безпеки вебзастосунків. Він дозволяє автоматично виявляти вразливості, такі як SQL-ін'єкції чи XSS, шляхом сканування вебсайтів під час їх роботи. ZAP зручний у використанні, інтегрується в CI/CD пайплайни і може працювати як у ручному, так і в автоматичному режимі. Це робить його популярним вибором для розробників і команд безпеки. Хоча він не такий потужний, як комерційні

інструменти, його безкоштовність і можливості роблять його ідеальним для невеликих і середніх проєктів.

3. Trivy

Trivy — це легкий та швидкий інструмент для сканування контейнерних образів, сховищ і залежностей на наявність вразливостей. Він аналізує базові образи, бібліотеки та конфігурації на основі відомих баз даних CVE. Trivy легко інтегрується в CI/CD пайплайни, що дозволяє автоматично перевіряти безпеку контейнерів перед розгортанням. Простота використання та швидкість роботи роблять його популярним серед розробників. Однак для великих і складних проєктів може знадобитися додатковий інструментарій через обмежені можливості деталізації аналізу.

4. HashiCorp Vault

HashiCorp Vault — це інструмент для безпечного зберігання та управління секретами, такими як токени доступу, паролі, ключі шифрування та конфіденційні дані. Він забезпечує централізоване зберігання з високим рівнем шифрування, динамічну генерацію секретів для баз даних і сервісів, а також контроль доступу на основі ролей (RBAC). Vault інтегрується з багатьма інструментами, хмарними платформами та Kubernetes, що робить його ідеальним для використання в середовищах DevSecOps. Хоча Vault забезпечує високий рівень безпеки, його налаштування може бути складним для невеликих команд або проєктів з обмеженими ресурсами.

5. Kubernetes Network Policies

Kubernetes Network Policies — це механізм для управління мережевою взаємодією між подами у кластері Kubernetes [3]. Вони дозволяють визначати правила доступу на основі джерела трафіку, типу протоколу (TCP, UDP) і

портів, забезпечуючи ізоляцію компонентів системи та захист від несанкціонованого доступу.

Network Policies надають:

- Контроль мережових потоків: Можливість дозволяти чи блокувати трафік між подами або зовнішніми ресурсами.
- Гнучкість: Детальне налаштування доступу залежно від застосунку, простору імен або міток подів.
- Інтеграція з мережевими плагінами: Наприклад, Calico, Cilium або Weave Net.

Ці політики вимагають налаштування YAML-файлів, де описуються правила доступу. Хоча їх впровадження підвищує безпеку, воно може бути складним у великих кластерах через необхідність детального планування мережевої архітектури. Kubernetes Network Policies ідеально підходять для захисту мікросервісів у кластері.

2.3.3 Порівняння інструментів безпеки

Для реалізації практик DevSecOps та інтеграції безпеки на всіх етапах життєвого циклу розробки програмного забезпечення було розглянуто кілька популярних інструментів. У цьому пункті наводиться порівняння інструментів безпеки з точки зору їх функціональності, переваг, обмежень та доцільності використання у межах магістерської роботи.

Таблиця 2.2 — Порівняльна таблиця інструментів безпеки

Інструмент	Переваги	Недоліки	Рекомендація
SonarQube	Статичний аналіз коду, інтеграція з CI/CD	Вимога до серверного середовища	Для статичного аналізу
OWASP ZAP	Безкоштовний	Обмеження у	Для динамічного

	DAST, інтеграція з пайплайнами	функціоналі	аналізу
Trivy	Швидке сканування контейнерів, інтеграція з CI/CD	Обмежена ефективність для складних систем	Для перевірки контейнерних образів
HashiCorp Vault	Безпечне управління секретами	Складність налаштування	Для управління секретами
Kubernetes Policies	Ізоляція компонентів, мережевий захист	Вимога до налаштування	Для мережевої безпеки

Для забезпечення комплексної безпеки в магістерській роботі обрано такі інструменти:

1. SonarQube для статичного аналізу коду (SAST), що дозволяє автоматично виявляти вразливості у вихідному коді.
2. OWASP ZAP для динамічного аналізу безпеки (DAST) з метою перевірки вебзастосунку під час виконання.
3. Trivy для сканування контейнерних образів на вразливості перед розгортанням.
4. HashiCorp Vault для безпечного управління конфіденційними даними.
5. Kubernetes Network Policies для забезпечення мережевої ізоляції між компонентами системи.

Обрані інструменти відповідають сучасним стандартам безпеки та легко інтегруються у CI/CD процеси, що забезпечує високий рівень автоматизації та захисту.

Висновок

Забезпечення безпеки є важливим аспектом у розробці інформаційних систем. Використання інструментів для статичного й динамічного аналізу, сканування контейнерів, управління секретами та мережевої ізоляції дозволяє значно знизити ризики та підвищити стійкість системи до атак. Обрані інструменти забезпечать інтеграцію безпеки на всіх етапах життєвого циклу проєкту, відповідаючи вимогам DevSecOps та сучасним стандартам.

2.4 Вибір інструментів контейнеризації

Для реалізації магістерської роботи важливо обрати інструмент контейнеризації, який забезпечить ефективність, гнучкість та відповідність сучасним вимогам. Аналіз інструментів показав, що Docker є найоптимальнішим вибором, враховуючи його можливості, популярність та інтеграцію з сучасними платформами та інструментами. У цьому підрозділі детально розглянемо основні аргументи на користь вибору Docker.

1. Простота використання

Однією з головних переваг Docker є його простота. Інтуїтивно зрозумілий інтерфейс та доступність великої кількості навчальних матеріалів дозволяють швидко опанувати цей інструмент навіть розробникам із мінімальним досвідом роботи з контейнерами.

Ключові аспекти простоти Docker:

- Командний рядок (CLI): Docker CLI забезпечує легке управління контейнерами за допомогою простих і зрозумілих команд (наприклад, ``docker run``, ``docker build``, ``docker ps``).
- Широка документація: Docker має багатий набір офіційних документів, а також численні ресурси від спільноти, що спрощує процес впровадження.

2. Популярність і підтримка спільноти

Docker є найбільш популярним інструментом контейнеризації, що робить його вибір вигідним завдяки активній підтримці спільноти та широкій інтеграції з іншими технологіями. Велика кількість розробників і користувачів Docker гарантує швидке вирішення проблем через форуми, GitHub, Stack Overflow тощо. Команда розробників регулярно випускає оновлення, додаючи нові функції та покращуючи безпеку й продуктивність. Docker використовується в багатьох компаніях і організаціях, що підтверджує його надійність і відповідність сучасним вимогам.

3. Сумісність із хмарними платформами та оркестраторами

Docker забезпечує безшовну інтеграцію з популярними хмарними платформами (AWS, GCP, Azure) та системами оркестрації (Kubernetes, Docker Swarm). Docker підтримує створення контейнерів за стандартами OCI (Open Container Initiative), що робить його ідеальним для використання в кластерах Kubernetes. Docker може бути легко інтегрований із сервісами, такими як Amazon ECS, Google Kubernetes Engine та Azure Kubernetes Service, для автоматизації розгортання контейнерів у хмарному середовищі. Крім того, наявність офіційного реєстру контейнерів дозволяє швидко отримати доступ до готових образів, оптимізуючи процес розробки.

4. Екосистема та модульність

Docker має широку екосистему, яка включає інструменти для розгортання, управління, моніторингу та тестування контейнерів [2]:

- Docker Hub: Це хмарний реєстр образів, який надає доступ до тисяч офіційних і користувацьких образів. Це значно прискорює розробку, зменшуючи необхідність створення базових образів з нуля.

- Docker Desktop: Простий у використанні інструмент для локального тестування та розгортання контейнерів, який підтримує інтеграцію з хмарними платформами.

- Підтримка модульності: Docker дозволяє розробникам ізолювати різні сервіси у вигляді контейнерів, забезпечуючи легке оновлення, заміну та тестування окремих компонентів.

5. Масштабованість і продуктивність

Docker дозволяє ефективно масштабувати застосунки за рахунок швидкого створення та розгортання контейнерів. Серед переваг маємо:

- Швидкість розгортання: Завдяки використанню контейнерів Docker забезпечує мінімальні затримки при розгортанні нових версій застосунку.

- Економія ресурсів: Контейнери Docker працюють безпосередньо на ядрі хостової системи, що забезпечує високу продуктивність із мінімальними накладними витратами.

- Масштабованість: Docker інтегрується з оркестраторами, такими як Kubernetes, що дозволяє автоматизувати масштабування в залежності від навантаження.

6. Безпека

Хоча Docker критикують за залежність від демон-процесу, він має розвинені механізми забезпечення безпеки:

- Ізоляція контейнерів: Кожен контейнер працює у власному просторі, що мінімізує ризик впливу на інші компоненти.

- Сканування контейнерних образів: Інтеграція з інструментами, такими як Trivy, дозволяє виявляти вразливості в базових образах.

- Підтримка секретного менеджменту: Docker SGCerts дозволяє зберігати конфіденційну інформацію, таку як паролі або ключі, у безпечному середовищі.

2.5 Вибір інструментів оркестрації

Для реалізації магістерської роботи необхідно обрати платформу оркестрації, яка дозволить забезпечити надійне управління контейнеризованими застосунками, їх масштабування, моніторинг та відмовостійкість. На основі аналізу популярних інструментів оркестрації було прийнято рішення використовувати Kubernetes як основний інструмент для управління контейнерами. Нижче наведено детальний опис причин цього вибору.

1. Висока масштабованість

Однією з ключових переваг Kubernetes є його здатність до масштабування контейнеризованих застосунків відповідно до потреб проєкту, а саме:

- Горизонтальне масштабування: Kubernetes дозволяє додавати або зменшувати кількість подів (запущених екземплярів контейнерів) у кластері на основі навантаження.
- Автоматичне масштабування: Kubernetes підтримує автоскейлінг (Horizontal Pod Autoscaler), який дозволяє автоматично змінювати кількість подів у залежності від метрик використання ресурсів, таких як CPU або пам'ять.
- Підтримка великих кластерів: Kubernetes може керувати тисячами контейнерів у великомасштабних системах, забезпечуючи їхню узгоджену роботу.

2. Відмовостійкість і висока доступність

Kubernetes забезпечує надійну роботу застосунків завдяки вбудованим механізмам відновлення та балансування навантаження, таким як:

- Перезапуск подів: У разі збоїв Kubernetes автоматично перезапускає поди, переміщує їх на інші вузли або замінює їх на нові.
- Балансування навантаження: Kubernetes розподіляє запити між подами у кластері, забезпечуючи рівномірний розподіл навантаження.
- Self-healing: Kubernetes автоматично видаляє поди, які працюють некоректно, і створює нові для підтримки заданого стану системи.

3. Підтримка мікросервісної архітектури

Kubernetes ідеально підходить для реалізації мікросервісної архітектури завдяки своїй гнучкості та можливостям ізоляції:

- Підтримка подів: Kubernetes дозволяє групувати контейнери в поди, які працюють разом і мають спільні мережеві ресурси.
- Мережеві політики: Використовуючи Network Policies, можна ізолювати мікросервіси один від одного, забезпечуючи їхню безпеку та контроль доступу.
- Сервіс-дискавері: Kubernetes надає механізми автоматичного виявлення сервісів через DNS і дозволяє легко маршрутизувати трафік між подами.

4. Інтеграція з хмарними платформами

Kubernetes підтримує інтеграцію з усіма основними хмарними платформами (GCP, GCP, Azure), що забезпечує додаткову гнучкість у виборі інфраструктури. Amazon Elastic Kubernetes Service (EKS) надає повністю керовану Kubernetes-інфраструктуру на GCP із автоматизацією оновлень, масштабування та інтеграцією з іншими сервісами GCP. Google Kubernetes Engine (GKE) дозволяє легко розгортати та управляти Kubernetes-кластерами з інтеграцією з Google Cloud. Azure Kubernetes Service (AKS) пропонує повністю керований Kubernetes із підтримкою DevOps-інструментів та автоматизації.

5. Розширюваність та екосистема

Kubernetes має велику екосистему інструментів і розширень, що дозволяє адаптувати його до різних потреб проєкту, такі як:

- Helm: Інструмент для управління пакетами Kubernetes, який спрощує встановлення та управління складними застосунками.
- Ingress: Інструмент для управління HTTP(S) трафіком, що дозволяє легко створювати маршрути між сервісами.
- Моніторинг: Kubernetes інтегрується з інструментами, такими як Prometheus і Grafana, для моніторингу метрик продуктивності та стану кластера.

6. Забезпечення безпеки

Kubernetes має вбудовані механізми для забезпечення безпеки контейнеризованих застосунків, а саме:

- Контроль доступу: Role-Based Access Control (RBAC) дозволяє налаштувати доступ до ресурсів Kubernetes на основі ролей.
- Секрети: Kubernetes SGCRets забезпечують безпечне зберігання конфіденційної інформації, такої як паролі, ключі доступу або токени.
- Мережеві політики: Використання Network Policies дозволяє обмежувати мережеву взаємодію між подами для запобігання несанкціонованому доступу.
- Сканування контейнерів: Kubernetes легко інтегрується з інструментами для перевірки контейнерних образів, такими як Trivy, для виявлення вразливостей.

7. Підтримка DevSecOps

Kubernetes є ключовим компонентом у впровадженні DevSecOps практик.

Kubernetes дозволяє автоматизувати розгортання через інтеграцію з GitHub Actions, Jenkins, GitLab CI/CD тощо. Kubernetes YAML-файли дозволяють описувати інфраструктуру, забезпечуючи відтворюваність і контроль версій. А інтеграція з інструментами для сканування безпеки, управління секретами та моніторингу забезпечує безпеку на всіх етапах розробки.

Kubernetes було обрано для магістерської роботи завдяки його масштабованості, відмовостійкості, підтримці мікросервісної архітектури та інтеграції з хмарними платформами. Його розширюваність, безпека та відповідність сучасним стандартам DevOps роблять Kubernetes оптимальним інструментом для управління контейнеризованими застосунками. Це рішення дозволить забезпечити стабільність, продуктивність і безпеку системи, а також інтеграцію з іншими компонентами інфраструктури.

2.6 Вибір інструментів автоматизації

У сучасній розробці програмного забезпечення автоматизація процесів розгортання, тестування та забезпечення безпеки є невід'ємною складовою ефективного життєвого циклу системи. Вибір інструментів автоматизації має значний вплив на швидкість впровадження, масштабованість та стабільність проєкту. У цьому розділі розглянуто найпопулярніші інструменти для автоматизації, їх переваги та обмеження, а також обґрунтовано вибір оптимальних рішень для реалізації магістерської роботи.

2.6.1 Огляд інструментів автоматизації

1. GitHub Actions

GitHub Actions - це інтегрована система Continuous Integration (CI) та Continuous Deployment (CD), розроблена GitHub [1]. Вона дозволяє автоматизувати процеси збирання, тестування та розгортання програмного забезпечення безпосередньо на платформі GitHub, через що вибір інструмента автоматизації був на користь GitHub Actions.

Аналізуючи GitHub Actions, можна визначити, що цей сервіс є потужним та зручним інструментом для автоматизації CI/CD процесів. Він надає гнучкість, інтегрованість з GitHub та широкі можливості кастомізації. GitHub Actions є привабливим варіантом для проєктів, що базуються на платформі GitHub та потребують ефективного та зручного розгортання застосунків.

2. Jenkins

Jenkins - це відкрите програмне забезпечення для автоматизації процесів розробки та розгортання програмного забезпечення. Він є одним з найпопулярніших інструментів Continuous Integration (CI) та Continuous Deployment (CD). Він забезпечує широкі можливості для автоматизації процесів збирання, тестування та розгортання додатків.

Jenkins є потужним та розширюваним інструментом для автоматизації процесів CI/CD. Завдяки своїм можливостям управління, масштабованості та широкому спектру плагінів, Jenkins підходить для різних типів проєктів і розробників. Враховуючи потреби проєкту, Jenkins може бути відмінним вибором для створення ефективного та автоматизованого процесу розгортання та постачання програмного забезпечення.

3. GitLab CI/CD

GitLab CI/CD - це інтегрована система Continuous Integration (CI) та Continuous Deployment (CD), яка надає можливість автоматизувати процеси збирання, тестування та розгортання програмного забезпечення на платформі GitLab.

Цей сервіс є потужним інструментом для автоматичного розгортання застосунків та CI/CD процесів. Він надає гнучкість, простоту використання та широкі можливості конфігурації, що відповідають потребам багатьох проєктів різних розмірів та складності. Залежно від вимог проєкту, GitLab CI/CD може бути відмінним варіантом для автоматизації розгортання та постачання програмного забезпечення.

4. TravisCI

Travis CI - це хмарна платформа Continuous Integration (CI), яка дозволяє автоматизувати процеси компіляції, тестування та розгортання програмного забезпечення. Він є популярним інструментом серед розробників для побудови й постійного оновлення їх проєктів.

Цей сервіс надає просту та ефективну платформу для автоматичного розгортання застосунків та тестування коду. Він підходить для проєктів різної складності та масштабу, забезпечуючи широкий вибір мов програмування та можливостей налаштування. Travis CI є відмінним варіантом для реалізації CI/CD процесу та забезпечення якості програмного забезпечення.

2.6.2 Порівняння інструментів автоматизації

На сьогоднішній день існує багато інструментів неперервної інтеграції та автоматизації, наприклад такі як GitLab CI/CD, Jenkins, TravisCI та GitHub Actions. У порівняльній таблиці 2.3 можна побачити деякі відмінності інструментів автоматизації, наведених нижче.

Таблиця 2.3 — Порівняльна таблиця інструментів автоматизації

Критерій	GitLab CI/CD	Travis CI	Jenkins	GitHub Actions
Інтеграція з репозиторієм	Вбудовано в GitLab, надає безперервну інтеграцію та доставку з мінімальними налаштуваннями	Інтегрується з GitHub і Bitbucket	Потребує ручної інтеграції з репозиторієм	Вбудовано в GitHub, надає безперервну інтеграцію та доставку без додаткових налаштувань
Мови програмування	Підтримує багато мов програмування	Підтримує багато мов програмування	Підтримує багато мов через плагіни	Підтримує багато мов програмування
Інфраструктура	Заснований на Docker, працює на власній інфраструктурі і GitLab	Працює на власній інфраструктурі і Travis CI	Вимагає власної інфраструктури, яку потрібно розгортати та налаштовувати	Працює на інфраструктурі і GitHub, або власноруч налаштована інфраструктура
Паралельність	Так, з планом GitLab	Так, з планом Travis CI	Так, через плагіни	Так, включено за замовчуванням
Вартість	Безкоштовний для відкритих проєктів, платні плани для приватних проєктів	Безкоштовний для відкритих проєктів, платні плани для приватних проєктів	Безкоштовний, але потребує власної інфраструктури та обслуговування	Безкоштовний для відкритих проєктів, платні плани для приватних проєктів

Налаштування	Використовує .gitlab-ci.yml для налаштування	Використовує .travis.yml для налаштування .	Налаштування через веб-інтерфейс або Jenkinsfile.	Використовує файли yml у .github/workflows для налаштування
Додаткові функції	Інтегровані функції збірки, тестування та розгортання	Проста інтеграція з GitHub	Велика кількість плагінів та гнучкість в налаштуваннях	Інтеграція з GitHub, простота налаштування та використання

Для реалізації магістерської роботи обрано GitHub Actions як основний інструмент автоматизації. Основними причинами цього вибору є:

1. Інтеграція з GitHub: Використання GitHub як репозиторію дозволяє легко налаштувати CI/CD пайплайни без додаткових інтеграцій.
2. Готові рішення: Наявність великої кількості готових дій (actions) спрощує налаштування процесів автоматизації.
3. Підтримка контейнеризації та оркестрації: Інтеграція з Docker і Kubernetes відповідає вимогам до системи.
4. Простота у використанні: Інтуїтивний інтерфейс і легкість налаштування дозволяють швидко впровадити автоматизацію.

Автоматизація процесів CI/CD є критично важливою для успішної реалізації інформаційної системи. Обраний інструмент, GitHub Actions, забезпечує інтеграцію з репозиторіями, підтримку сучасних DevOps-практик, гнучкість і простоту використання. Це дозволяє оптимізувати розробку, тестування та розгортання, задовольняючи вимоги до продуктивності та надійності системи.

2.7 Вибір інструментів моніторингу

Моніторинг є ключовим компонентом для забезпечення стабільності, продуктивності та швидкого реагування на проблеми в системі. У магістерській роботі передбачається використання контейнеризованих застосунків у кластері Kubernetes, що вимагає інтегрованого рішення для збору метрик і візуалізації даних. У цьому розділі детально розглянуто вибрані інструменти — Prometheus та Grafana, які дозволяють ефективно вирішувати завдання моніторингу.

2.7.1 Prometheus

Prometheus — це популярна система моніторингу з відкритим кодом, яка використовується для збору й аналізу метрик, особливо у динамічних середовищах, таких як Kubernetes. Вона працює за моделлю pull, тобто самостійно збирає дані з різних джерел, як-от поди, вузли чи зовнішні сервіси. Завдяки мові запитів PromQL можна створювати складні запити для аналізу даних і налаштовувати алерти. Для автоматизації оповіщень Prometheus інтегрується з Alertmanager, що дозволяє швидко реагувати на аномалії. Інтеграція з Kubernetes є простою завдяки експортерам, таким як kube-state-metrics, які забезпечують збір метрик із кластерів.

Prometheus вирізняється високою продуктивністю, масштабованістю та багатою екосистемою, яка включає численні експортери для збору метрик. Він легко інтегрується з іншими DevOps-інструментами, але має певні обмеження. Наприклад, Prometheus не працює з логами та не має вбудованої візуалізації — для цього зазвичай використовується Grafana.

У рамках магістерської роботи Prometheus стане ключовим інструментом для моніторингу. Він забезпечить збір метрик про продуктивність системи, стан подів і використання ресурсів у кластері Kubernetes, створюючи основу для ефективного управління та аналізу.

2.7.2 Grafana

Grafana — це потужна платформа для візуалізації даних і створення інтерактивних дашбордів. Вона дозволяє аналізувати інформацію з різних джерел, таких як Prometheus, Elasticsearch чи InfluxDB, і надає широкий інструментарій для налаштування графіків, таблиць і діаграм. Grafana також підтримує інтеграцію з різними системами моніторингу, що робить її універсальним інструментом для збору та аналізу метрик.

Користувачі цінують Grafana за її зручний інтерфейс, який дозволяє легко створювати дашборди навіть без глибоких технічних знань. Завдяки підтримці багатьох джерел даних, можна поєднувати інформацію з різних систем для побудови комплексного моніторингу. Крім цього, платформа підтримує плагіни, що розширюють її функціональність. Проте є і певні недоліки: Grafana сама по собі не збирає дані, а працює лише з існуючими системами, і налаштування оповіщень може бути складним для новачків.

У магістерській роботі Grafana відіграє важливу роль у візуалізації метрик, отриманих із Prometheus. Вона дозволить створювати дашборди для моніторингу продуктивності системи та налаштовувати автоматичні оповіщення про критичні події, забезпечуючи ефективне управління станом системи.

2.7.3 Використання Prometheus та Grafana

Об'єднання Prometheus і Grafana забезпечує комплексний підхід до моніторингу:

1. Збір метрик: Prometheus буде відповідати за регулярний збір даних із подів Kubernetes, вузлів і сервісів.
2. Аналіз і алерти: За допомогою PromQL у Prometheus будуть налаштовані правила для відстеження аномалій та автоматичного інформування.
3. Візуалізація: Grafana забезпечить створення кастомних дашбордів для аналізу продуктивності системи, використання ресурсів і стану компонентів.

4. Інтеграція з DevOps-процесами: Налаштований моніторинг допоможе швидко виявляти проблеми у CI/CD пайплайнах, а також оптимізувати роботу кластера.

Для магістерської роботи було обрано два ключових інструменти моніторингу — Prometheus та Grafana. Prometheus забезпечить збір і аналіз метрик, тоді як Grafana буде використовуватися для їх візуалізації та створення інтерактивних дашбордів. Разом ці інструменти нададуть повний спектр можливостей для моніторингу системи, дозволяючи оперативно реагувати на проблеми та забезпечувати стабільну роботу застосунків у кластері Kubernetes.

2.8 Інструменти та їх взаємодія

У процесі створення автоматизованої системи розгортання та моніторингу вебзастосунків у хмарному середовищі ключовим завданням є інтеграція різних інструментів, які забезпечують контейнеризацію, оркестрацію, моніторинг, безпеку та автоматизацію. Для цього обрано сучасні DevOps-інструменти, кожен із яких виконує окрему роль, але при цьому інтегрується з іншими для забезпечення безперервного циклу розробки та підтримки.

Для контейнеризації обрано Docker, який забезпечує ізоляцію застосунків і їх залежностей у контейнерах. Docker дозволяє стандартизувати середовище розгортання, забезпечуючи портативність і консистентність незалежно від платформи. Контейнери, створені за допомогою Docker, стають базовим компонентом для подальшої оркестрації.

Для управління контейнеризованими застосунками використовується Kubernetes, який забезпечує автоматизацію розгортання, масштабування та управління контейнерами. Kubernetes взаємодіє з Docker для запуску контейнерів, а також підтримує інтеграцію з хмарною платформою GCP через сервіс GKE (Elastic Kubernetes Service). Оркестрація забезпечує високу доступність, балансування навантаження та відмовостійкість.

Для автоматизації процесів CI/CD обрано GitHub Actions, що інтегрується з репозиторієм коду і забезпечує автоматичне збирання, тестування та розгортання застосунків. GitHub Actions взаємодіє з Kubernetes для автоматизованого оновлення кластерів і розгортання нових версій програмного забезпечення.

Моніторинг системи здійснюється за допомогою Prometheus для збору метрик і Grafana для їх візуалізації. Prometheus отримує метрики із компонентів Kubernetes, таких як поди, вузли та сервіси, а Grafana дозволяє створювати кастомні дашборди для аналізу даних. Ці інструменти взаємодіють через Prometheus як основне джерело даних для Grafana.

Для забезпечення безпеки використовуються кілька підходів:

- Trivy для сканування контейнерних образів на вразливості ще до розгортання.
- HashiCorp Vault для безпечного зберігання конфіденційної інформації, такої як ключі доступу та токени.
- Kubernetes Network Policies для налаштування мережевої ізоляції між компонентами системи.

Інструменти системи працюють у тісній взаємодії:

1. Docker створює контейнерні образи, які розгортаються Kubernetes.
2. Kubernetes керує подами, використовуючи метрики від Prometheus для автоматичного масштабування.
3. GitHub Actions забезпечує автоматичне оновлення образів і їх розгортання через інтеграцію з Kubernetes.
4. Prometheus і Grafana відстежують стан системи та відображають метрики для аналізу продуктивності та використання ресурсів.
5. Trivy і HashiCorp Vault інтегруються у пайплайн CI/CD для перевірки безпеки контейнерів і управління секретами.

Висновок

Розробка ефективної структури системи є ключовим етапом у створенні автоматизованого середовища розгортання та управління вебзастосунками. У цьому розділі було проаналізовано та визначено основні компоненти системи, інструменти для їх реалізації, а також описано їхню взаємодію.

Обрані інструменти, такі як Docker для контейнеризації, Kubernetes для оркестрації, GitHub Actions для автоматизації CI/CD, Prometheus і Grafana для моніторингу, а також HashiCorp Vault і Trivy для забезпечення безпеки, створюють узгоджену та гнучку інфраструктуру. Взаємодія між цими компонентами дозволяє реалізувати повний цикл DevOps-процесів із дотриманням практик DevSecOps.

Запропонована структура системи забезпечує високу продуктивність, гнучкість і безпеку в управлінні контейнеризованими застосунками в хмарному середовищі. Вона дозволяє ефективно масштабувати ресурси, автоматизувати процеси розгортання, здійснювати моніторинг стану системи й оперативно реагувати на потенційні загрози. Це створює надійну основу для успішної реалізації магістерської роботи.

РОЗДІЛ 3. ФУНКЦІОНАЛЬНА МОДЕЛЬ І АРХІТЕКТУРА

Розробка ефективної та надійної інформаційної системи вимагає чіткого визначення її функціональної моделі та архітектури. Цей розділ присвячений побудові системи, яка забезпечує автоматизацію розгортання вебзастосунків у хмарному середовищі з використанням сучасних DevOps і DevSecOps практик.

У рамках даного розділу буде сформульовано змістовну постановку задачі, описано процеси діяльності та розроблено функціональну модель системи. Це включає вибір ключових компонентів, таких як інструменти контейнеризації, оркестрації, CI/CD, моніторингу та безпеки, а також їх інтеграцію в єдину архітектуру. Окрему увагу приділено створенню діаграм, які відображають основні компоненти системи, їх взаємодію, стани, послідовності та потоки даних.

Розроблена модель та архітектура забезпечують гнучкість, масштабованість і відмовостійкість системи. Вона відповідає сучасним вимогам до автоматизації та безпеки, дозволяючи ефективно управляти всіма етапами життєвого циклу вебзастосунків — від розробки до підтримки в продуктивному середовищі.

3.1 Змістовна постановка задачі

Розробка автоматизованої системи для розгортання вебзастосунків у хмарному середовищі є складним завданням, яке вимагає поєднання сучасних технологій контейнеризації, оркестрації, моніторингу та управління безпекою. Система повинна забезпечувати високу продуктивність, гнучкість і відмовостійкість, що є критичними вимогами в сучасній розробці програмного забезпечення.

Основна задача полягає у створенні інтегрованого середовища, яке дозволяє автоматизувати всі ключові етапи життєвого циклу вебзастосунку: від розробки та тестування до розгортання та підтримки в реальному часі. Це передбачає використання DevOps- і DevSecOps-практик, що забезпечують

швидкість впровадження змін, стабільність системи та мінімізацію ризиків безпеки.

Мета проєкту — створити інфраструктуру, яка автоматизує процеси розгортання вебзастосунків, оптимізує управління ресурсами та інтегрує інструменти для забезпечення безпеки та моніторингу. Система повинна бути здатна:

1. Контейнеризувати застосунки для забезпечення їх ізоляції та портативності.
2. Автоматично оркеструвати розгортання з урахуванням динамічного масштабування та балансування навантаження.
3. Інтегрувати CI/CD-процеси, забезпечуючи безперервну інтеграцію, тестування та доставку програмного забезпечення.
4. Здійснювати моніторинг продуктивності та стану компонентів у режимі реального часу.
5. Гарантувати безпеку через перевірку контейнерних образів, управління конфіденційними даними та контроль доступу.

Основними завданнями розробки системи є забезпечення її ефективності, гнучкості та безпеки. Починаючи з контейнеризації, важливо створити стандартизовані образи для застосунків, які забезпечують ізоляцію залежностей і портативність. Це дозволяє запускати застосунки на різних платформах без додаткових змін. Оркестрація додає ще більше можливостей: автоматичне масштабування, балансування навантаження та відновлення компонентів у разі збоїв стають основою стабільності системи.

Ще одним важливим аспектом є автоматизація процесів. За допомогою GitHub Actions реалізується безперервна інтеграція, тестування та доставка (CI/CD). Це дозволяє мінімізувати ризики людських помилок і прискорити робочі процеси. Моніторинг, який здійснюється через Prometheus і Grafana, дає можливість у реальному часі відстежувати стан системи, аналізувати її продуктивність і оперативно реагувати на потенційні проблеми.

Безпека — ключовий елемент, що пронизує всі рівні системи. Перевірка контейнерних образів на вразливості через Trivy, управління конфіденційними даними через HashiCorp Vault і застосування мережових політик Kubernetes забезпечують комплексний підхід до захисту системи.

Система має відповідати ряду вимог, таких як масштабованість для адаптації до змін у навантаженні, відмовостійкість для швидкого відновлення у разі збоїв, автоматизація для зменшення ручного втручання, гнучкість для інтеграції нових сервісів та інструментів, простота моніторингу і висока безпека. Це дозволить досягти очікуваних результатів: автоматизованого розгортання, оптимального використання ресурсів, точного моніторингу, високого рівня безпеки та адаптованості системи до різних потреб.

У підсумку, створена система дозволить не лише оптимізувати робочі процеси, але й забезпечить стабільність та безпеку вебзастосунків у хмарному середовищі.

Змістовна постановка задачі визначає основні напрямки роботи над системою, які спрямовані на створення автоматизованої, гнучкої та безпечної платформи для розгортання вебзастосунків у хмарному середовищі. Виконання цих завдань дозволить забезпечити стабільність і продуктивність системи відповідно до сучасних підходів DevOps і DevSecOps.

3.2 Опис процесу діяльності

Основним процесами діяльності є:

- 1) Написання програмного коду
- 2) Завантаження коду у репозиторій GitHub
- 3) Контейнеризація та розгортання застосунку у хмарному середовищі

В деяких випадках, користувачі не мають достатньо знань та навичок для роботи з інструментами автоматизації, CI/CD інтеграції, контейнеризації та розгортання застосунків.

За допомогою автоматизованої системи розгортання вебзастосунків зазначені вище процеси не потребують безпосереднього втручання користувача, що значно спрощує процес конфігурації, автоматизації та розгортання застосунку.

3.3 Опис функціональної моделі

З системою взаємодіє два актори – користувач і DevOps спеціаліст. Користувач виступає у ролі виконавця, що завантажує свій код у шаблон репозиторію на GitHub, а DevOps спеціаліст у ролі адміністратора, що займається підтримкою системи.

Визначимо дії, які можуть виконувати актори системи та представимо їх у таблиці 3.1.

Таблиця 3.1 – Типи варіантів використання

Актор	Варіант використання	Опис дії варіанта використання
Виконавець	Створення своєї гілки репозиторію	Користувач вказує назву своєї гілки та натискає кнопку “Create new branch”.
Виконавець	Завантажити код застосунку у шаблон	Користувач у своїй гілці додає файли коду та пушить зміни до репозиторію.
Виконавець	Переглянути результат розгортання	Користувач у виводі пайплайну може знайти посилання на свій застосунок, перейти за посиланням та переглянути результат.
Виконавець	Видалення вебзастосунку з хмарного середовища	Користувач запускає пайплайн з видаленням його застосунку з хмарного середовища.
Адміністратор	Створення шаблону	Адміністратор обирає технологію і фреймворк, що хоче додати. Конфігурує пайплайн для нової

		технології та пушить зміни
Адміністратор	Редагування шаблону	Адміністратор обирає пайплайн з необхідною технологією або фреймворком та редагує параметри розгортання
Адміністратор	Управління документацією	Адміністратор може написати нові або редагувати старі інструкції користування системою

На діаграмах (рисунки 3.1 – 3.2) можна ознайомитися з усіма можливими діями для користувача та DevOps спеціаліста.

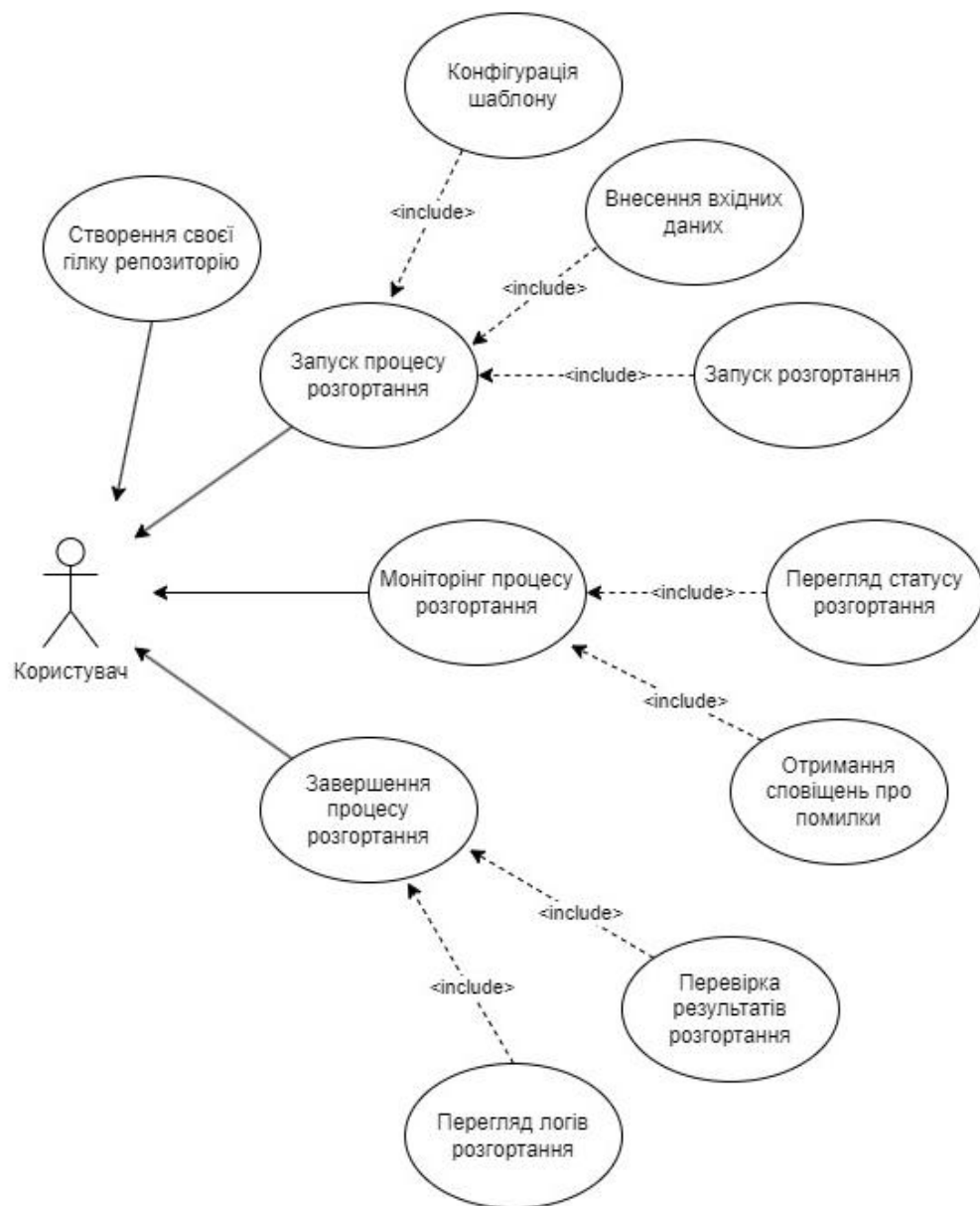


Рисунок 3.1 – можливості користувача

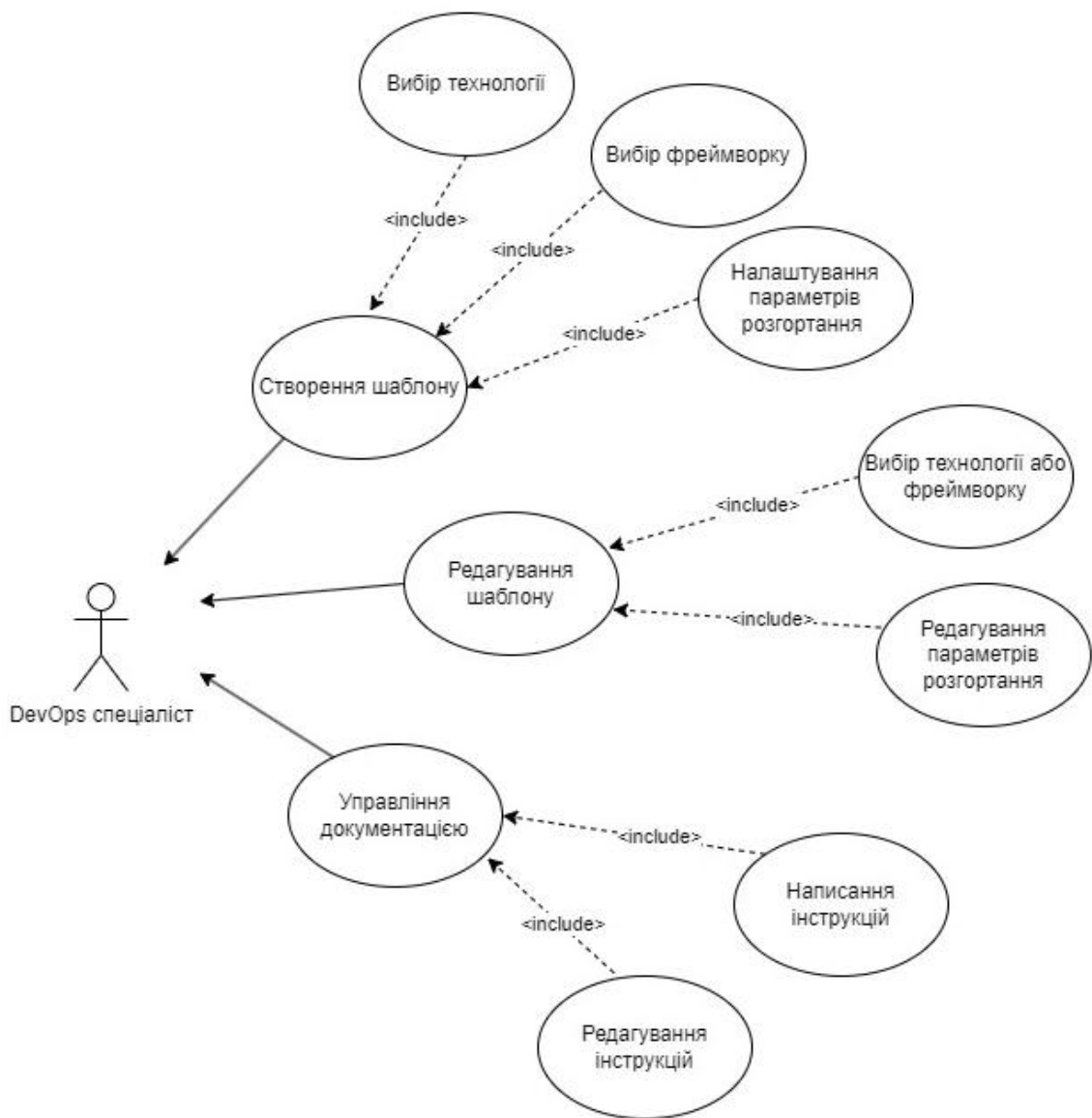


Рисунок 3.2 – можливості DevOps спеціаліста

З цих діаграм можна побачити основні можливості користувача, такі як створення своєї гілки, запуск та моніторинг процесу розгортання, а також можливості DevOps спеціаліста, такі як створення та редагування шаблону, а також управління документацією.

3.4 Архітектура інфраструктури

Архітектура системи для автоматичного розгортання застосунків побудована на основі принципів мікросервісної архітектури та неперервної

інтеграції/неперервної доставки (CI/CD). Цей підхід дозволяє ефективно розгортати, оновлювати та масштабувати застосунки в хмарному середовищі, забезпечуючи стабільність та гнучкість системи.

- **GitHub Repository:** Репозиторій GitHub використовується для зберігання вихідного коду застосунку та конфігураційних файлів. Це точка взаємодії розробників, де створюються гілки, відслідковуються зміни та керується версіями коду. Усі оновлення автоматично інтегруються в CI/CD пайплайн [14].

- **GitHub Actions:** Для автоматизації процесів CI/CD використовується GitHub Actions. Кожен пуш до репозиторію запускає пайплайн, який автоматично виконує тести, створює Docker-образи та розгортає їх у кластері Amazon Elastic Kubernetes Service (EKS).

- **Google Kubernetes Engine (GKE):** GKE — це керована хмарна платформа для розгортання та управління Kubernetes-кластерами. Вона забезпечує автоматичне масштабування, балансування навантаження, оркестрацію контейнерів і відмовостійкість. Завдяки GKE можна ефективно керувати життєвим циклом застосунків.

- **Google Cloud DNS:** Cloud DNS забезпечує налаштування DNS для з'єднання користувачів із розгорнутими застосунками. Використання Cloud DNS дозволяє налаштовувати дружні URL-адреси для доступу до застосунків і забезпечує надійний маршрутизаційний сервіс [11].

- **Google Container Registry (GCR):** GCR слугує сховищем для Docker-образів, які створюються у процесі CI/CD. Образи зберігаються у приватному репозиторії, забезпечуючи їх безпеку та легкий доступ для подальшого використання в кластері GKE.

Система інтегрує всі ці компоненти для забезпечення автоматичного розгортання, моніторингу та масштабування застосунків у хмарному середовищі GCP. На рисунку 3.3 нижче наведено схему архітектури системи.

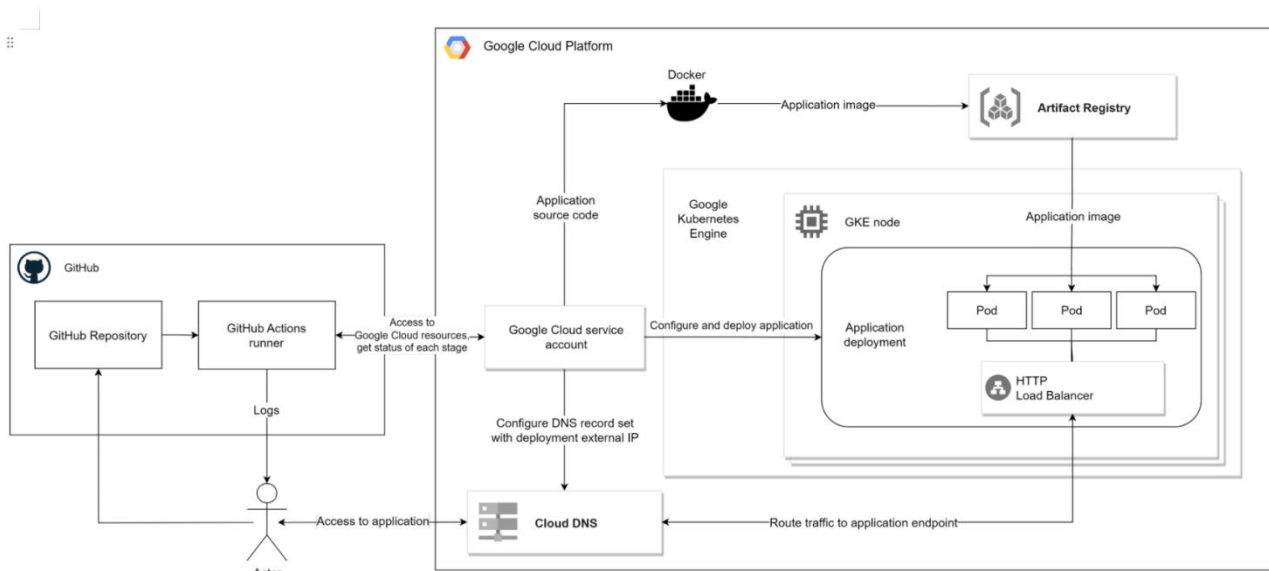


Рисунок 3.3 – Схема архітектури системи

Ця архітектура дозволяє виконувати швидкі та надійні розгортання застосунків у хмарі за допомогою сервіс акаунту [12]. З детальною структурною схемою можна ознайомитись у графічному матеріалі Д1.

3.4.1 GitHub Repository

GitHub Repository є ключовим елементом системи, який об'єднує всі компоненти проєкту, зокрема вихідний код, конфігураційні файли для розгортання, файл Dockerfile для створення контейнерів, а також декларації для автоматизації через GitHub Actions. Репозиторій забезпечує централізоване управління проєктом, підтримуючи процес розробки, тестування та розгортання.

У цій системі репозиторій налаштований для інтеграції з GitHub Actions, що дозволяє автоматизувати збирання та розгортання застосунку після внесення змін. Усі налаштування та інструкції для CI/CD зберігаються у файлах конфігурації, які визначають послідовність дій, необхідних для автоматизації процесів.

Розробники працюють із репозиторієм, створюючи гілки, вносячи зміни до коду або конфігурацій, та пушуючи їх назад. Ці зміни можуть охоплювати додавання нових функцій, виправлення помилок або оновлення конфігурацій.

Як тільки зміни потрапляють у репозиторій, GitHub Actions автоматично запускає пайплайн.

Пайплайн включає збирання нового Docker-образу на основі Dockerfile, який знаходиться в репозиторії. Зібраний образ завантажується до GCR за допомогою облікових даних сервісного аккаунту. Після цього образ стає доступним для використання в кластері GKE.

GitHub Repository виступає центральним вузлом у процесі CI/CD, інтегруючись із іншими компонентами системи для автоматизації, моніторингу та управління вебзастосунками. Це забезпечує швидке й надійне розгортання, дозволяючи розробникам фокусуватися на створенні нових функцій і оптимізації застосунків.

3.4.2 GitHub Actions

GitHub Actions — це ключовий інструмент для автоматизації процесів розробки та розгортання застосунків, який забезпечує безперервну інтеграцію (CI) та доставку (CD). У поточній роботі GitHub Actions використовується для автоматизації збирання, тестування, пушу Docker-образів у Google Container Registry (GCR) та розгортання цих образів у кластері Kubernetes на базі GKE.

Всі дії, які виконує GitHub Actions, описуються в YAML-файлах, що зберігаються в репозиторії. Ці файли визначають, які події в репозиторії (наприклад, пуш у певну гілку) запускають виконання конкретних сценаріїв. Наприклад, при кожному оновленні коду GitHub Actions автоматично збирає Docker-образ застосунку, завантажує його в GCR та ініціалізує розгортання в GKE, використовуючи надані конфігураційні файли Kubernetes.

GitHub Actions також інтегрується з іншими компонентами системи. Він використовує Dockerfile із репозиторію для створення контейнерних образів, облікові дані GCP IAM для доступу до GCR і GKE, а також файли конфігурації Kubernetes для визначення специфіки розгортання застосунку [8]. Така

взаємодія дозволяє автоматизувати весь процес CI/CD, забезпечуючи швидкість, гнучкість і точність у реалізації кожного етапу.

Отже, GitHub Actions виступає важливим елементом екосистеми автоматизації, полегшуючи роботу з хмарними платформами, оптимізуючи розгортання та забезпечуючи надійність і узгодженість у процесах CI/CD.

3.4.3 Google Kubernetes Engine (GKE)

Google Kubernetes Engine (GKE) — це керований сервіс Kubernetes, який надається GCP. Він забезпечує автоматизоване розгортання, масштабування та управління контейнеризованими застосунками в хмарному середовищі. У межах цього проєкту GKE використовується для розгортання та керування застосунками, які збираються та пушаються в Google Artifact Registry за допомогою GitHub Actions.

GKE взаємодіє з іншими компонентами системи наступним чином:

- **GitHub Actions:** Після збирання Docker-образу застосунку та його завантаження в GCR, GitHub Actions автоматично розгортає цей образ у кластері GKE. Це включає налаштування параметрів розгортання, таких як вибір образу, конфігурація контейнерів і управління ресурсами.
- **Google Container Registry (GCR):** GKE витягує Docker-образи з GCR для створення контейнерів у кластері. Облікові дані для доступу до GCR надаються через GCP IAM, налаштований у GitHub Actions.
- **GitHub Repository:** Конфігураційні файли Kubernetes, такі як маніфести подів, сервісів та інші специфікації розгортання, зберігаються в репозиторії GitHub і використовуються GitHub Actions для ініціалізації розгортання в GKE.

Таким чином, GKE є основою для управління контейнеризованими застосунками, інтегруючись із GitHub Actions, GCR і GitHub Repository для забезпечення безперебійної роботи системи.

3.4.4 Google Cloud DNS

Google Cloud DNS — це високопродуктивна, надійна та безпечна послуга DNS, що надається GCP. Google Cloud DNS перетворює (резолює) доменні імена на IP-адреси, що дозволяє користувачам і сервісам звертатися до застосунків та ресурсів через зручні для людини доменні імена замість IP-адрес.

У контексті цього проєкту Google Cloud DNS використовується для налаштування DNS-записів, які спрямовують трафік до застосунків, розгорнутих у кластері Google Kubernetes Engine (GKE).

Google Cloud DNS взаємодіє з іншими компонентами системи наступним чином:

- Google Kubernetes Engine (GKE): DNS-записи в Google Cloud DNS налаштовуються таким чином, щоб спрямовувати трафік до балансувальників навантаження, які обслуговують сервіси в кластері GKE.
- GitHub Actions: У процесі CI/CD GitHub Actions може автоматично оновлювати DNS-записи в Google Cloud DNS. Наприклад, якщо IP-адреса балансувальника навантаження змінюється після нового розгортання в GKE, GitHub Actions забезпечує оновлення відповідного DNS-запису, щоб підтримувати безперебійний доступ до застосунків.

Google Cloud DNS виконує ключову функцію в системі, забезпечуючи стабільний і надійний доступ до застосунків, розгорнутих у кластері GKE, за допомогою зручних DNS-імен. Цей інструмент гарантує, що користувачі зможуть легко та швидко отримати доступ до необхідних ресурсів навіть за умови динамічних змін у конфігурації системи.

3.4.5 Google Container Registry (GCR)

Google Container Registry (GCR) — це обслуговувана служба GCP для управління контейнерними образами, яка спрощує автоматизацію процесів розробки та доставки програмного забезпечення. GCR підтримує зберігання,

керування та розгортання Docker-образів, які використовуються в цьому проєкті.

У межах поточного проєкту GCR використовується для зберігання Docker-образів, які потім розгортаються на Google Kubernetes Engine (GKE).

GCR взаємодіє з іншими компонентами системи наступним чином:

- **GitHub Repository:** Код додатка, що зберігається в репозиторії GitHub, використовується для створення Docker-образів. Ці образи завантажуються (push) в GCR як частина процесу автоматизації CI/CD.
- **GitHub Actions:** GitHub Actions автоматизує створення Docker-образів із вихідного коду додатка та завантажує їх у GCR. Це є важливою складовою процесу безперервної інтеграції та доставки.
- **Google Kubernetes Engine (GKE):** Під час розгортання додатків у кластері GKE необхідні Docker-образи завантажуються (pull) з GCR і використовуються для створення подів і сервісів.

Таким чином, Google Container Registry (GCR) є ключовим компонентом системи, який забезпечує надійне зберігання контейнерних образів, а також інтегрує процеси з GitHub Repository, GitHub Actions і GKE. GCR створює основу для узгодженого та ефективного процесу автоматизації, від створення образів до їх розгортання в хмарному середовищі.

3.5 Діаграма компонентів

Даний розділ візуалізує основні компоненти системи та їх взаємодію, описуючи ключові елементи, які виконують певні функції в інфраструктурі. Кожен компонент відповідає за конкретний аспект системи, що забезпечує її ефективність, масштабованість і надійність.

Діаграма компонентів допомагає чітко уявити структуру системи, показуючи, як її окремі елементи взаємодіють між собою. Це спрощує процес розробки, тестування та подальшого управління системою.

У поточному проєкті основними компонентами є GitHub Repository, GitHub Actions, Google Kubernetes Engine (GKE), Google Container Registry (GCR) та Google Cloud DNS.

- GitHub Repository використовується для зберігання вихідного коду проєкту, відстеження змін та управління версіями. Він є основним джерелом даних для інших компонентів системи.
- GitHub Actions автоматизує процеси збирання, тестування та розгортання застосунків, забезпечуючи безперервну інтеграцію (CI) та доставку (CD).
- Google Kubernetes Engine (GKE) використовується для розгортання та управління контейнеризованими застосунками у масштабованому хмарному середовищі.
- Google Container Registry (GCR) служить для зберігання Docker-образів, які створюються GitHub Actions і використовуються GKE для розгортання.
- Google Cloud DNS забезпечує управління DNS-записами, які спрямовують користувачів до застосунків, розгорнутих у кластері GKE.

На рисунку 3.4 зображено діаграму компонентів системи.

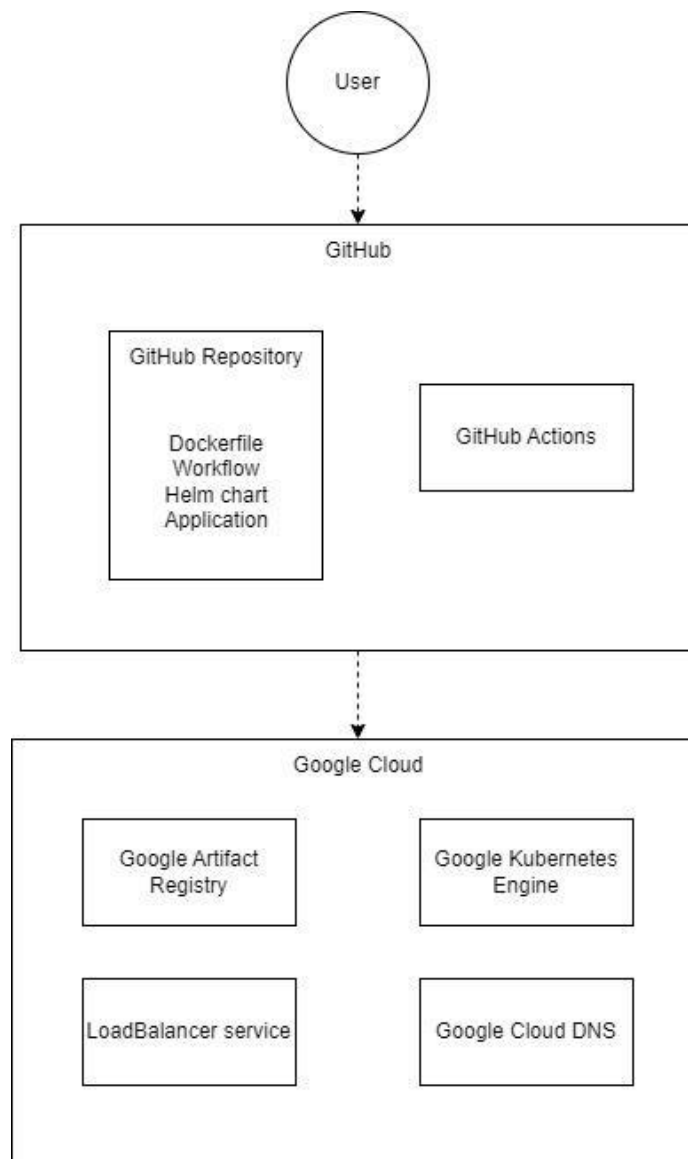


Рисунок 3.4 – Діаграма компонентів

Кожен із цих компонентів відіграє важливу роль у функціонуванні системи, створюючи взаємопов’язану інфраструктуру для ефективного управління життєвим циклом застосунків.

3.6 Діаграма станів

Даний розділ присвячений візуалізації можливих станів, через які проходить проєкт у своєму життєвому циклі, а також переходів між цими станами. Діаграма станів є корисним інструментом для моделювання динаміки поведінки системи, допомагаючи зрозуміти, як вона змінюється залежно від виконуваних дій та подій.

Для поточного проєкту діаграма станів охоплює ключові етапи автоматизації CI/CD процесу з використанням GKE. Основні стани включають:

1. Створення репозиторію: Початковий стан, коли GitHub Repository створено, але він ще не містить усіх необхідних компонентів, таких як Dockerfile, конфігураційні файли Kubernetes та налаштування для GitHub Actions.
2. Розробка: У цьому стані розробники додають і змінюють код застосунку в репозиторії, створюючи нові функції або виправляючи помилки.
3. Збірка образу: За допомогою GitHub Actions створюється Docker-образ із застосунком. Після успішного виконання цього кроку образ завантажується в Google Container Registry (GCR).
4. Розгортання: Якщо збірка пройшла успішно, Docker-образ із GCR розгортається у кластері Google Kubernetes Engine (GKE). Використовуються конфігураційні файли Kubernetes для визначення структури розгортання.
5. Використання: Після успішного розгортання застосунок доступний для кінцевих користувачів через DNS-записи, налаштовані в Google Cloud DNS.

На діаграмі станів (рисунок 3.5) показано, як ці стани взаємопов'язані. Перехід між станами відбувається через події, такі як коміт у GitHub, завершення збірки Docker-образу або успішне розгортання в кластері. Діаграма станів дозволяє візуалізувати динаміку системи, демонструючи її готовність адаптуватися до змін і забезпечувати безперебійне функціонування протягом усього життєвого циклу проєкту.



Рисунок 3.5 – Діаграма станів

На цій діаграмі зображено основні стани системи під час роботи: створення гілки репозиторію, розробка, збірка образу, розгортання та використання застосунку користувачем.

3.7 Діаграма послідовності

Розділ спрямований на візуалізацію взаємодії між компонентами системи в хронологічному порядку. Ця діаграма відображає, як система реагує на окремі події, представляючи послідовність дій, що відбуваються в системі відповідно до часової осі.

Діаграма послідовності є важливою частиною документації проєкту, оскільки вона допомагає розуміти, як різні компоненти системи співпрацюють один з одним у визначений момент часу. Така візуалізація дозволяє аналізувати та оптимізувати взаємодію компонентів системи, що веде до покращення продуктивності системи.

Діаграму послідовності для розроблюваної системи наведено на рисунку 3.6.



Рисунок 3.6 – Діаграма послідовності

В контексті проєкту діаграма послідовності включає в себе такі основні етапи: внесення конфігураційних даних в репозиторій GitHub, виявлення змін за допомогою GitHub Actions, збірка застосунку, збереження зібраного Docker-образу в GCR, розгортання застосунку в GKE і оновлення DNS записів у Google Cloud DNS.

3.8 Діаграма потоків даних

Діаграма потоків даних для проєкту автоматичного розгортання застосунків на базі GitHub репозиторію описує основні компоненти системи та взаємодію між ними. Ця діаграма демонструє, як дані переміщуються між ключовими елементами інфраструктури:

- **GitHub Repository:** Центральне сховище, де зберігається вихідний код застосунку. Вхідними даними є коміти, зроблені розробниками, а вихідними — версії коду, які використовуються для подальших етапів автоматизації.
- **GitHub Actions:** Цей компонент отримує код із репозиторію як вхідні дані й виконує автоматизовані дії, такі як тестування, збирання Docker-образів і завантаження їх у сховище. Вихідними даними є готові до використання Docker-образи.
- **Google Container Registry (GCR):** GCR слугує сховищем для Docker-образів, зібраних GitHub Actions. Вхідними даними є Docker-образи, а вихідними — образи, які використовуються для розгортання в кластері.
- **Google Kubernetes Engine (GKE):** GKE отримує Docker-образи з GCR і розгортає їх як поди в кластері Kubernetes. Вхідними даними є Docker-образи, а вихідними — розгорнуті застосунки.
- **Google Cloud DNS:** забезпечує управління DNS-записами для застосунків, розгорнутих у кластері GKE. Вхідними даними є інформація про IP-адреси сервісів, а вихідними — оновлені DNS-записи, які дозволяють користувачам отримувати доступ до застосунків за доменними іменами.

Діаграма потоків даних візуалізує шлях, яким дані переміщуються через систему, забезпечуючи чітке розуміння процесу автоматизації та взаємодії між компонентами. Ця структура гарантує ефективне використання ресурсів і швидке розгортання застосунків.

На рисунку 3.7 зображено діаграму потоків даних системи.

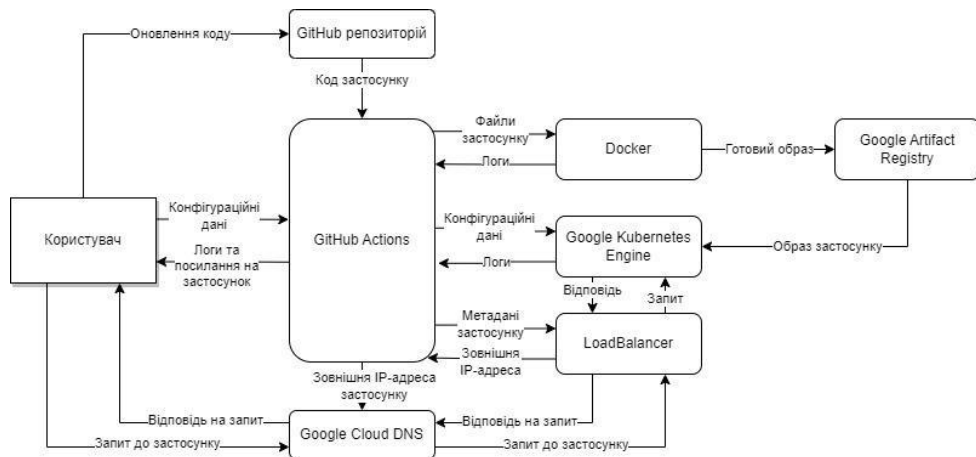


Рисунок 3.7 – Діаграма потоків даних

Висновок

У розділі "Функціональна модель і архітектура" було проведено детальний аналіз функціональних можливостей і архітектурного складу системи. Визначено основні процеси, включаючи розгортання, моніторинг, оновлення та видалення застосунків.

Побудована діаграма функціональних компонентів наочно відображає взаємозв'язки між ключовими елементами системи. Основними компонентами архітектури стали GitHub Repository, GitHub Actions, Google Container Registry (GCR), Google Kubernetes Engine (GKE) та Google Cloud DNS. Кожен із цих елементів виконує критичну роль у забезпеченні функціональності системи.

GitHub Repository використовується для зберігання вихідного коду та конфігураційних файлів, GitHub Actions автоматизує процеси збирання та тестування, Google Container Registry забезпечує зберігання Docker-образів, GKE відповідає за розгортання застосунків у кластері, а Google Cloud DNS управляє DNS-записами для забезпечення доступу до застосунків.

Описана архітектура дозволяє досягти високого рівня автоматизації процесу розгортання застосунків, що є однією з ключових вимог сучасної розробки. Це забезпечує оперативність у внесенні змін, гнучкість, масштабованість і надійність системи.

Таким чином, аналіз функціональної моделі й архітектури підтверджує, що система структурована відповідно до вимог проєкту. Автоматизація процесів розгортання забезпечує високу ефективність, що дозволяє легко управляти й адаптувати систему під нові потреби.

РОЗДІЛ 4. ТЕХНІЧНИЙ РОЗДІЛ

4.1 Керівництво користувача

Посібник користувача надає інструкції для зручного та ефективного використання системи кінцевими користувачами. У ньому детально описано основні функції системи та взаємодію між її компонентами.

Основні кроки роботи з системою:

1. Створення власної гілки в репозиторії:

Користувач створює власну гілку в GitHub репозиторії через веб-інтерфейс або CLI (інтерфейс командного рядка).

2. Розміщення потрібних файлів у відповідних директоріях:

Користувач додає необхідні файли до відповідних директорій у репозиторії.

3. Запуск розгортання через GitHub Actions:

Після додавання файлів, користувач запускає GitHub Actions, що забезпечує автоматичне розгортання застосунку відповідно до конфігураційного файлу репозиторію.

4.1.1 Створення нової гілки за допомогою веб-інтерфейсу GitHub

Відкрити GitHub і перейти до сторінки репозиторію проєкта, як на рисунку 4.1.

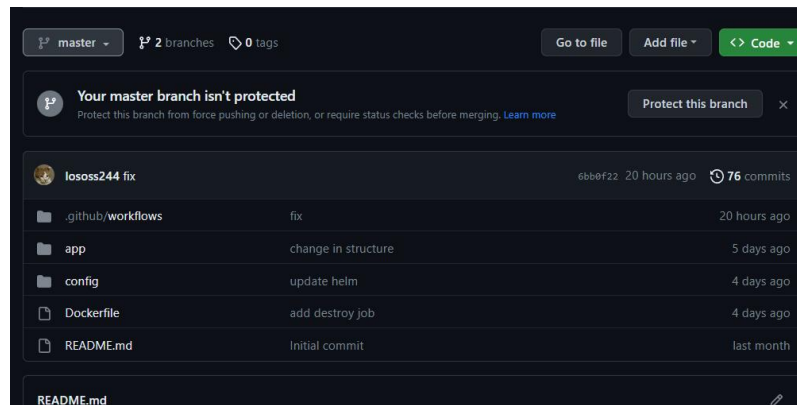


Рис. 4.1 – Сторінка репозиторія

На панелі вгорі списку файлів є поле, яке зазвичай відображає поточну гілку (наприклад, "main" або "master"). Натиснувши на це поле, буде представлений список доступних гілок, як на рисунку 4.2. У цьому списку є текстове поле для введення нового імені гілки. Потрібно ввести ім'я нової гілки і натиснути "Enter". GitHub автоматично створить нову гілку з поточної гілки. Ця нова гілка міститиме всі файли та історію комітів основної гілки.

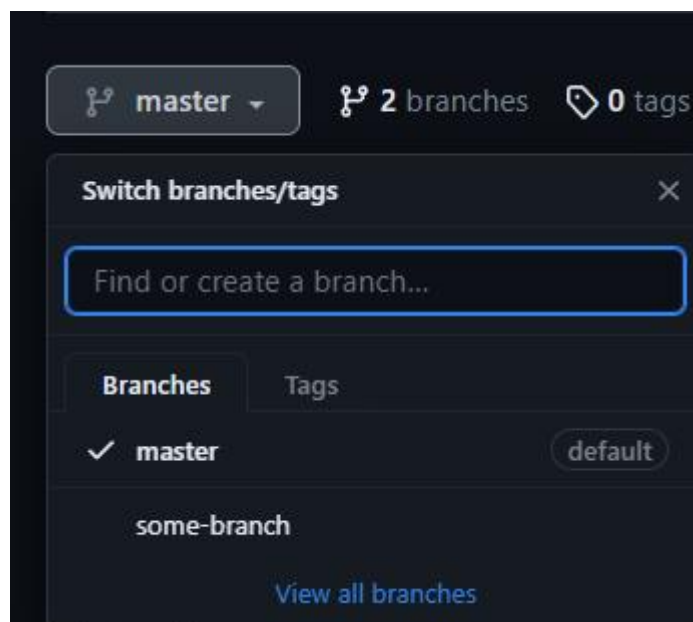


Рис. 4.2 – Список доступних гілок

Слід пам'ятати, що при роботі з гілками у GitHub, коли робляться зміни в новій гілці, ці зміни не впливають на основну гілку, злиття (merge) цих змін. Це дозволяє безпечно експериментувати та працювати над новими функціями, не переймаючись про можливі проблеми, які можуть виникнути з основною гілкою.

4.1.2 Створення нової гілки за допомогою CLI

Створення нової гілки за допомогою командного рядка (CLI) в Git можна здійснити за допомогою наступних кроків:

Відкрити термінал (якщо користувач працює в Linux або MacOS) або командний рядок (якщо користувач працює в Windows). Якщо в користувача немає сконованого репозиторію, то необхідно перейти до директорії, куди потрібно сколонувати репозиторій і ввести команди, які зазначено нижче на рисунку 4.3, замінивши `/path/to/your/repository` на шлях до необхідної директорії.



```
cd /path/to/your/repository  
git clone https://github.com/lososs244/gcp-master.git
```

Рис. 4.3 – Команди для клонування репозиторію

Або одразу навігувати до директорії, де розташований Git-репозиторій за допомогою команди `cd /path/to/your/repository`.

Переконайтесь, що робота йде на актуальній версії основної гілки (часто це "main" або "master"), можна перейшовши до основної гілки за допомогою команди `git checkout`, а потім оновивши її за допомогою `git pull`, як на рисунку 4.4.

```
git checkout main  
git pull
```

Рис. 4.4 – Команди для завантаження актуальної версії коду

Далі необхідно створити нову гілку за допомогою команди на рисунку 4.5, вказавши ім'я нової гілки, замінивши `my-new-branch` на назву нової гілки.

```
git checkout -b my-new-branch
```

Рис. 4.5 – Команда для створення та переходу до нової гілки

4.1.3 Запуск розгортання через GitHub Actions

Для того щоб запустити процес розгортання потрібно перейти до репозиторію на веб-сайті GitHub і натиснути на вкладку "Actions". Далі треба вибрати робочий процес «Deploy to Google Cloud», натиснути на кнопку "Run workflow" справа. Одразу після цього буде запропоновано сконфігурувати розгортання шляхом внесення необхідної інформації як це зображено на рисунку 4.6. Потрібно натиснути "Run workflow" ще раз, щоб запустити робочий процес.

The image shows a configuration form for a GitHub Action workflow. It has a dark theme. At the top, there is a label 'app_name' with a red asterisk, followed by a text input field. Below this are two checkboxes: 'deploy_app' and 'destroy_app', both of which are currently unchecked. Underneath the checkboxes is a label 'framework' followed by a dropdown menu showing 'default'. At the bottom of the form is a green button with the text 'Run workflow'.

Рис. 4.6 – Інтерфейс конфігурації розгортання

Детальніше зі змінними конфігурації можна ознайомитись у таблиці 4.1.

Таблиця 4.1 – Змінні конфігурації

Назва змінної	Тип змінної	Опис змінної	Обов’язкова
App_name	string	Назва застосунку, також ця змінна потім буде використана для створення посилання	так
Deploy_app	bool	Чи потрібно розгортати застосунок у хмарі	ні
Destroy_app	bool	Чи потрібно видалити застосунок з хмари	ні
Framework	choice	Вибір фреймворку	ні

У цій таблиці описано назву, тип та опис змінної, а також її обов’язковість.

4.2 Випробування системи

4.2.1 Мета випробувань

Головною метою проведення випробувань є перевірка чи система працює відповідно до заданих вимог та специфікацій, і готова до використання.

4.2.2 Системне тестування

На цьому етапі перевіряється система в цілому, щоб переконатися, що вона працює правильно та відповідає заданим вимогам та специфікаціям. В ході випробувань, проводилась перевірка всього функціоналу системи, результати записано до таблиць 4.2 – 4.4.

Таблиця 4.2 – Перевірка збірки образу з застосунком

Мета випробування	Перевірити чи працює етап збірки образу з застосунком
Початковий стан	Відкрито вікно конфігурації запуску пайплайна
Випробування	Ввести конфігураційні дані та натиснути кнопку “Run workflow”
Очікуваний результат	Запуск пайплайна, успішна збірка та надсилання образу до Google Artifact Registry.
Стан моделі після випробування	Запуск пайплайна, успішна збірка та пуш образу до Google Artifact Registry.

Отже, з таблиці 4.2 можна побачити, що система правильно обробляє конфігураційні дані, а також успішно збирає та надсилає образ із застосунку до Google Artifact Registry.

Таблиця 4.3 – Перевірка розгортання застосунку у хмарному середовищі

Мета випробування	Перевірити чи працює етап розгортання застосунку у хмарному середовищі
Початковий стан	Успішна збірка та пуш образу до Google Artifact Registry.
Випробування	Збір необхідного образу для подальшого розгортання застосунку, розгортання застосунку у хмарному середовищі
Очікуваний результат	Застосунок розгорнутий у хмарному середовищі

Стан моделі після випробування	Застосунок розгорнутий у хмарному середовищі
--------------------------------	--

Отже, з таблиці 4.3 можна побачити, що система успішно забирає необхідний образ та розгортає застосунок у хмарному середовищі.

Таблиця 4.4 – Перевірка конфігурації Cloud DNS сервісу

Мета випробування	Перевірити чи працює етап конфігурації Cloud DNS сервісу
Початковий стан	Успішне розгортання застосунку у хмарному середовищі
Випробування	Отримання зовнішньої IP-адреси застосунку та налаштування DNS сервісу
Очікуваний результат	Правильно налаштований DNS record set, застосунок доступний через посилання
Стан моделі після випробування	Правильно налаштований DNS record set, застосунок доступний через посилання

Отже, з таблиці 4.4 можна побачити, що система правильно налаштовує DNS record set та застосунок доступний через посилання.

4.3 Системні обмеження

Система наразі працює на одній віртуальній машині, що впливає на загальну кількість виконуваних водночас процесів. Кількість виконуваних водночас задач можна збільшити шляхом збільшення кількості ранерів [13] (серверів, що виконують команди та підтримують зв'язок з GitHub Actions).

Мінімальні вимоги до сервера можна побачити у таблиці 4.5.

Таблиця 4.5 – Системні вимоги до ранера

Назва	Вимога
Операційна система	Linux, macOS, Windows
Процесор	2 vCPU
Оперативна пам'ять	2 ГБ RAM
Дисковий простір	10 ГБ вільного дискового простору
Мережеве з'єднання	Швидке і надійне інтернет-з'єднання для взаємодії з GitHub та іншими сервісами
	Встановлений застосунок GitHub Actions Runner

У таблицях 4.6-4.8 можна побачити приблизний час виконання кожного кроку роботи пайплайну.

Таблиця 4.6 – Час виконання кроків етапу збірки

Назва процесу	Опис процесу	Приблизний час виконання
Set up job	Підготовка сервера до виконання пайплайну	2 секунди
Checkout job	Отримання коду з репозиторію і розміщення його на сервері	1 секунда
Configure Docker	Конфігурація Docker для роботи з інтерфейсом командного рядка Google Cloud	15 секунд

Build and push Docker image	Збірка образу з застосунком та завантаження його до Google Artifact Registry	20 секунд
Complete job	Виведення exit code	Менше секунди

Таблиця 4.7 – Час виконання кроків етапу розгортання

Назва процесу	Опис процесу	Приблизний час виконання
Set up job	Підготовка сервера до виконання пайплайну	2 секунди
Pull Docker image	Завантаження образу із застосунком у разі його відсутності	2 секунд
Get cluster endpoint	Отримання даних для підключення до GKE кластера	15 секунд
Deploy to GKE	Розгортання застосунку у Google Cloud	55 секунд
Configure DNS	Налаштування DNS сервісу	50 секунд
Complete job	Виведення exit code	Менше секунди

Таблиця 4.8 – Час виконання кроків етапу видалення

Назва процесу	Опис процесу	Приблизний час виконання
Set up job	Підготовка сервера до виконання пайплайну	2 секунди
Get cluster endpoint	Отримання даних для підключення до GKE кластера	15 секунд
GKE uninstall	Видалення застосунку з Google Cloud	55 секунд
Delete DNS record set	Видалення DNS запису застосунку	50 секунд
Complete job	Виведення exit code	Менше секунди

Висновок до розділу

В розділі було розглянуто ключові аспекти використання та тестування системного рішення, а також важливі системні обмеження.

Розділ включає в себе інструкції з використання системи. Було розглянуто процес створення гілки за допомогою CLI та веб-інтерфейсу GitHub, процес додавання файлів до репозиторію, а також запуск GitHub Actions для автоматичного розгортання застосунку. Керівництво спрямоване на забезпечення зрозумілості та простоти використання для користувачів, допомагаючи їм ефективно використовувати систему.

У розділі було описано процес системного тестування, надано таблиці з результатами тестів.

У розділі описано основні технічні вимоги та обмеження системи. Це включає мінімальні вимоги до обладнання для використання системи, обмеження зберігання та передачі даних, а також вимоги до мережевого з'єднання.

Проведений аналіз та опис технічного розділу допомагає користувачам та розробникам краще розуміти систему, її функціональність та можливості. Він також виявляє потенційні області для подальшого вдосконалення та оптимізації, що в кінцевому підсумку призведе до створення більш надійного та ефективного продукту.

РОЗДІЛ 5. ОХОРОНА ПРАЦІ

Охорона праці є важливим аспектом роботи, особливо під час взаємодії з інформаційними системами та обладнанням. Основна мета – забезпечити безпеку, комфорт та збереження здоров'я користувачів, мінімізуючи ризики, які можуть виникати під час роботи. До цього відносяться організація робочого місця, дотримання електро- та пожежної безпеки, профілактика професійних захворювань, контроль рівня шуму та дотримання екологічних стандартів.

5.1 Загальні положення

Охорона праці під час роботи з інформаційними системами та програмним забезпеченням повинна відповідати державним стандартам та нормативним документам. Основними завданнями охорони праці є:

- Забезпечення безпеки користувачів під час роботи з комп'ютерною технікою.
- Зниження ризиків виникнення аварійних ситуацій.
- Забезпечення належного рівня ергономіки та комфорту робочого місця.
- Профілактика професійних захворювань.

Робоче середовище повинно відповідати чинним санітарно-гігієнічним нормам, щоб забезпечити комфортні умови праці та зменшити негативний вплив на здоров'я користувачів.

5.2 Вимоги до робочого місця

Організація робочого місця:

- Робоче місце має бути обладнане меблями, які відповідають ергономічним вимогам: регульований стілець, стіл з достатнім простором для роботи.
- Екран монітора має розташовуватися на відстані 50-70 см від очей користувача.
- Робоче місце повинно бути оснащено достатнім освітленням. Перевага надається природному освітленню або використанню ламп денного світла.
- Робоче приміщення має бути забезпечене вентиляцією для постійного надходження свіжого повітря.

Санітарно-гігієнічні умови:

- Температура у приміщенні має бути в межах 18–25°C.
- Рівень вологості має бути не нижчим за 40% і не вищим за 60%.
- Шумовий рівень не повинен перевищувати 50 дБ, щоб уникнути дискомфорту та втрати концентрації.

5.3 Електробезпека

Основні правила роботи з електрообладнанням:

- Використовувати лише сертифіковане обладнання з належними засобами захисту.
- Не перевантажувати електромережу шляхом підключення великої кількості пристроїв до одного джерела живлення.
- Регулярно перевіряти справність кабелів, розеток та електропристроїв.

Дії в аварійних ситуаціях:

- У разі виникнення короткого замикання негайно відключити живлення.
- Не використовувати обладнання, яке має видимі пошкодження кабелів чи розеток.
- Повідомити відповідальну особу про інцидент.

5.4 Пожежна безпека

Профілактичні заходи:

- Забороняється залишати увімкнене обладнання без нагляду.
- Не допускається використання електропристроїв із пошкодженою ізоляцією кабелів.
- У приміщенні повинні бути встановлені вогнегасники та інші засоби пожежогасіння, які регулярно перевіряються на придатність до використання.

Дії у разі пожежі:

- Негайно повідомити відповідні служби (ДСНС) за номером 101.
- Відключити електроживлення приміщення, якщо це можливо і не загрожувє життю.
- Організовано евакуювати працівників із приміщення згідно з планом евакуації.

5.5 Профілактика професійних захворювань

Робота з інформаційними системами часто супроводжується тривалим сидінням, використанням клавіатури, миші та монітора, що може призводити

до професійних захворювань (синдром зап'ясткового каналу, проблеми зору тощо).

Рекомендації для профілактики:

- Робити 5–10-хвилинні перерви кожні 50 хвилин роботи за комп'ютером.
- Виконувати вправи для очей: фокусування на об'єктах, розташованих на різній відстані.
- Виконувати фізичні вправи для зняття напруження з м'язів шиї, спини та рук.

Організація режиму роботи:

- Максимальна тривалість роботи за комп'ютером не повинна перевищувати 6 годин на день без перерв.
- Забезпечення ротації завдань для зменшення навантаження на один і той самий тип діяльності.

5.6 Екологічні аспекти

Утилізація обладнання:

- Використане обладнання (комп'ютери, батареї, кабелі) слід утилізувати відповідно до екологічних стандартів.
- Використання сервісів для безпечного перероблення електронних відходів.

Енергозбереження:

- Використовувати енергоефективне обладнання.
- Вимикати пристрої, які не використовуються, щоб зменшити енергоспоживання.

5.7 Рівень шуму

Рівень шуму в приміщенні, де проводиться робота з інформаційними системами, має відповідати нормативним вимогам. Допустимий рівень шуму не повинен перевищувати 50 дБ, щоб забезпечити комфортну атмосферу для роботи. Надмірний шум може викликати втому, зниження концентрації, головний біль та стрес. Для зменшення рівня шуму слід:

- Використовувати шумопоглинаючі матеріали для стін, підлоги та стелі.
- Уникати розміщення шумного обладнання у приміщеннях для роботи.
- Використовувати сучасну техніку з низьким рівнем шуму.
- Забезпечувати технічне обслуговування пристроїв для зменшення шуму, який вони видають.

У разі використання шумного обладнання рекомендується застосовувати захисні засоби, такі як шумозахисні навушники, або встановлювати це обладнання у звукоізованих приміщеннях. Дотримання цих вимог сприяє збереженню концентрації, покращенню загального самопочуття користувачів і підвищенню продуктивності праці.

Висновок до розділу

Дотримання правил охорони праці під час роботи з інформаційною системою є ключовим аспектом створення безпечного та комфортного робочого середовища. Забезпечення відповідності робочого місця санітарно-гігієнічним нормам, контроль рівня шуму, дотримання вимог електро- та пожежної безпеки сприяють зменшенню ризику виникнення аварійних ситуацій та негативного впливу на здоров'я працівників.

Виконання цих вимог забезпечує не лише фізичну безпеку, але й сприяє підвищенню загальної продуктивності праці, мінімізуючи стрес і втому, пов'язані з незручними умовами чи надмірним навантаженням. Профілактичні заходи, такі як перерви у роботі, фізичні вправи, дотримання норм рівня шуму та освітлення, допомагають підтримувати довгострокове здоров'я працівників, попереджуючи розвиток професійних захворювань.

Загалом, системний підхід до охорони праці створює передумови для ефективної роботи, забезпечує захист здоров'я працівників і відповідає високим стандартам безпеки та екологічності. Це не лише покращує якість робочого процесу, а й підвищує репутацію організації, демонструючи турботу про своїх працівників та відповідальне ставлення до навколишнього середовища.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. GitHub Actions Documentation [Електронний ресурс] - Режим доступу до ресурсу: <https://docs.github.com/en/actions>
2. Overview of Docker Build [Електронний ресурс] - Режим доступу до ресурсу: <https://docs.docker.com/build>
3. Kubernetes documentation [Електронний ресурс] - Режим доступу до ресурсу: <https://kubernetes.io/docs/home>
4. Why Google Cloud [Електронний ресурс] - Режим доступу до ресурсу: <https://cloud.google.com/why-google-cloud>
5. AWS Documentation [Електронний ресурс] - Режим доступу до ресурсу: https://docs.aws.amazon.com/?nc2=h_ql_doc_do
6. Get to know Azure [Електронний ресурс] - Режим доступу до ресурсу: <https://azure.microsoft.com/en-us/explore>
7. How Heroku works [Електронний ресурс] - Режим доступу до ресурсу: <https://devcenter.heroku.com/articles/how-heroku-works>
8. Google Cloud IAM overview [Електронний ресурс] - Режим доступу до ресурсу: <https://cloud.google.com/iam/docs/overview>
9. Google Cloud Artifact Registry overview [Електронний ресурс] - Режим доступу до ресурсу: <https://cloud.google.com/artifact-registry>
10. Google Cloud Deploy an app to a GKE cluster [Електронний ресурс] - Режим доступу до ресурсу: <https://cloud.google.com/kubernetes-engine/docs/deploy-app-cluster>
11. Google Cloud DNS documentation [Електронний ресурс] - Режим доступу до ресурсу: <https://cloud.google.com/dns/docs>
12. Google Cloud Service accounts overview [Електронний ресурс] - Режим доступу до ресурсу: <https://cloud.google.com/iam/docs/service-account-overview>
13. About self-hosted runners [Електронний ресурс] - Режим доступу до ресурсу: <https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners>

14. GitHub Repositories documentation [Электронный ресурс] - Режим доступа до ресурсу: <https://docs.github.com/en/repositories>
15. Jones, P. (2019). *Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications*. O'Reilly Media.
16. Bernstein, D. (2014). *Containers and cloud: From LXC to Docker to Kubernetes*. IEEE Cloud Computing

ДОДАТОК А. Конфігураційні маніфести

```
apiVersion: v2
name: gcp-master
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library'
# chart.
#
# Application charts are a collection of templates that
# can be packaged into versioned archives
# to be deployed.
#
# Library charts provide useful utilities or functions for
# the chart developer. They're included as
# a dependency of application charts to inject those
# utilities and functions into the rendering
# pipeline. Library charts do not define any templates and
# therefore cannot be deployed.
type: application

# This is the chart version. This version number should be
# incremented each time you make changes
# to the chart and its templates, including the app
# version.
# Versions are expected to follow Semantic Versioning
# (https://semver.org/)
version: 0.1.0

# This is the version number of the application being
# deployed. This version number should be
# incremented each time you make changes to the
# application. Versions are not expected to
# follow Semantic Versioning. They should reflect the
# version the application is using.
# It is recommended to use it with quotes.
appVersion: "1.16.0"
```

```
app:
  name: "example-app"
  image: "example-image"
  protocol: TCP
```

```

port: 80
resources:
  requests:
    memory: "50Mi"
    cpu: "100m"
  limits:
    memory: "150Mi"
    cpu: "200m"
annotation: {}

service:
  port: 80
  targetPort: 80

serviceAccount:
  create: false

autoscaling:
  enabled: false
  minReplicas: 1
  maxReplicas: 3
  targetCPUUtilizationPercentage: 90
  targetMemoryUtilizationPercentage: 70

ingress:
  enabled: false
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
  hosts:
    - host: example.com
      paths:
        - path: /
          pathType: Prefix
          service:
            name: web-service
            port: 80
    - host: api.example.com
      paths:
        - path: /
          pathType: Prefix
          service:
            name: api-service

```

```

        port: 8080
    tls:
      - hosts:
          - example.com
          - api.example.com
        secretName: example-tls

```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ .Values.app.name | quote }}
  namespace: {{ .Values.app.name | quote }}
  labels:
    app: {{ .Values.app.name | quote }}
spec:
  replicas: 2
  selector:
    matchLabels:
      app: {{ .Values.app.name | quote }}
  template:
    metadata:
      labels:
        app: {{ .Values.app.name | quote }}
    spec:
      containers:
        - name: {{ .Values.app.name | quote }}
          image: {{ .Values.app.image | quote }}
          resources:
            {{- .Values.app.resources | toYaml | nindent
12 }}
          ports:
            - name: http
              containerPort: {{ .Values.app.port }}
              protocol: {{ .Values.app.protocol | quote }}

```

```

{{- if .Values.autoscaling.enabled }}
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: {{ printf "%s-hpa" .Values.app.name | quote }}

```



```

    namespace: {{ .Values.app.name | quote }}
    labels:
      app: {{ .Values.app.name | quote }}
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: {{ .Values.app.name | quote }}
  minReplicas: {{ .Values.autoscaling.minReplicas }}
  maxReplicas: {{ .Values.autoscaling.maxReplicas }}
  metrics:
    {{-
if .Values.autoscaling.targetCPUUtilizationPercentage }}
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization:
{{ .Values.autoscaling.targetCPUUtilizationPercentage }}
        {{- end }}
    {{-
if .Values.autoscaling.targetMemoryUtilizationPercentage }}
    - type: Resource
      resource:
        name: memory
        target:
          type: Utilization
          averageUtilization:
{{ .Values.autoscaling.targetMemoryUtilizationPercentage }}
        {{- end }}
    {{- end }}

```

```

{{- if .Values.ingress.enabled -}}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: {{ .Values.app.name | quote }}
  namespace: {{ .Values.app.name | quote }}
  labels:
    app: {{ .Values.app.name | quote }}
  annotations:

```

```

spec:
  {{- if .Values.ingress.tls }}
  tls:
    {{- range .Values.ingress.tls }}
    - hosts:
        {{- range .hosts }}
        - {{ . | quote }}
        {{- end }}
      secretName: {{ .secretName }}
    {{- end }}
  {{- end }}
  rules:
    {{- range .Values.ingress.hosts }}
    - host: {{ .host | quote }}
      http:
        paths:
          {{- range .paths }}
          - path: {{ .path }}
            {{- with .pathType }}
            pathType: {{ . }}
            {{- end }}
          backend:
            service:
              name: {{ include "gcp-master.fullname" $ }}
              port:
                number: {{ $.Values.service.port }}
            {{- end }}
          {{- end }}
    {{- end }}
  {{- end }}

```

```

apiVersion: v1
kind: Namespace
metadata:
  name: {{ .Values.app.name | quote }}
  labels:
    app: {{ .Values.app.name | quote }}

```

```

apiVersion: v1
kind: Service
metadata:
  name: {{ printf "%s-service" .Values.app.name | quote }}

```

```
namespace: {{ .Values.app.name | quote }}
labels:
  app: {{ .Values.app.name | quote }}
spec:
  type: "LoadBalancer"
  ports:
    - name: http
      port: {{ .Values.service.port }}
      targetPort: {{ .Values.service.targetPort }}
      protocol: TCP
  selector:
    app: {{ .Values.app.name | quote }}
```

```
{{- if .Values.serviceAccount.create -}}
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ printf "%s-sa" .Values.app.name | quote }}
  namespace: {{ .Values.app.name | quote }}
  labels:
    app: {{ .Values.app.name | quote }}
automountServiceAccountToken:
{{ .Values.serviceAccount.automount }}
{{- end }}
```

```
apiVersion: v1
kind: Pod
metadata:
  name: {{ printf "%s-test-connection" .Values.app.name |
quote }}
  labels:
    app: {{ .Values.app.name | quote }}
  annotations:
    "helm.sh/hook": test
spec:
  containers:
    - name: wget
      image: busybox
      command: ['wget']
      args: ['{{ .Values.app.name |
quote }}:{{ .Values.service.port }}']
```

restartPolicy: Never

```
name: Application deployment/destruction
run-name: ${{ github.actor }} deploys on
${{ github.ref_name }}
on:
  push:
    branches: [ "!master" ]
  pull_request:
    branches: [ "master" ]

workflow_dispatch:
  inputs:
    app_name:
      required: true
      type: string
    deploy_app:
      required: false
      type: boolean
    destroy_app:
      required: false
      type: boolean
    framework:
      type: choice
      options:
        - default
        - React
        - Vue.js

jobs:
  build-push:
    name: Setup, Build, and Push
    runs-on: self-hosted
    if: ${{ inputs.deploy_app }}
    steps:
      - name: Checkout Code
        uses: actions/checkout@v2

      - name: Configure Docker
        run: gcloud auth configure-docker
${{ secrets.GCP_ZONE }}-docker.pkg.dev
```

```

- name: Build and Push Docker Image
  run: |
    docker build -t
${{ secrets.GCP_REGISTRY }}/${{ inputs.app_name }} .
    docker push
${{ secrets.GCP_REGISTRY }}/${{ inputs.app_name }}

security_scan:
  runs-on: ubuntu-latest
  name: Image security scan
  needs: [ build-push ]
  steps:
    - name: Checkout Code
      uses: actions/checkout@v3

    - name: Setup Trivy
      run: echo "/usr/local/bin" >> $GITHUB_PATH

    - name: Run Trivy Image Scan
      run: |
        trivy --severity HIGH,CRITICAL image
${{ secrets.GCP_REGISTRY }}/${{ inputs.app_name }}

deploy:
  name: Deploy Application
  runs-on: self-hosted
  needs: [ security_scan ]
  steps:
    - name: Pull Docker image
      run: |
        gcloud auth configure-docker
${{ secrets.GCP_ZONE }}-docker.pkg.dev
        docker pull
'${{ secrets.GCP_REGISTRY }}/${{ inputs.app_name }}'

    - name: Get Cluster Endpoint
      run: gcloud container clusters get-credentials
${{ secrets.GKE_CLUSTER }} --location
${{ secrets.GCP_ZONE }} --project
${{ secrets.GCP_PROJECT_ID }}

    - name: Deploy to GKE
      run: |

```

```

        cd deploy/gcp-master
        helm upgrade --install -f values.yaml
    {{{ inputs.app_name }}} --set
    app.name=${{{ inputs.app_name }}} ,app.image=${{{ secrets.GCP_
    REGISTRY }}}/${{{ inputs.app_name }}} . --wait

    - name: Create new DNS Record Set
      run: |
        export EXTERNAL_IP=$(kubectl get svc
        '{{{ inputs.app_name }}}-service' -n {{{ inputs.app_name }}}
        -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
        gcloud dns record-sets transaction start --
        zone=${{{ secrets.GCP_DNS_ZONE}}}
        gcloud dns record-sets transaction add
        $EXTERNAL_IP \
        --
        name=${{{ inputs.app_name }}}.${{{ secrets.GCP_DNS_DOMAIN }}}
        \
        --type=A \
        --ttl=300 \
        --zone=${{{ secrets.GCP_DNS_ZONE}}}
        gcloud dns record-sets transaction execute --
        zone=${{{ secrets.GCP_DNS_ZONE}}}

    destroy:
      name: Destroy Application
      runs-on: self-hosted
      if: {{{ inputs.destroy_app }}}
      steps:
        - name: Get Cluster Endpoint
          run: gcloud container clusters get-credentials
            {{{ secrets.GKE_CLUSTER }}} --location
            {{{ secrets.GCP_ZONE }}} --project
            {{{ secrets.GCP_PROJECT_ID }}}

        - name: Helm Uninstall
          run: helm uninstall {{{ inputs.app_name }}}

        - name: Delete DNS Record Set
          run: gcloud dns record-sets delete
            {{{ inputs.app_name }}}.${{{ secrets.GCP_DNS_DOMAIN }}}. --
            type="A" --zone=${{{ secrets.GCP_DNS_ZONE }}}

```

