Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

СИСТЕМА РОЗГОРТАННЯ ТА МОНІТОРИНГУ ПРИКЛАДНИХ СЕРВІСІВ ІЗ ІНТЕГРАЦІЄЮ DEVOPS-ПРАКТИК

Виконав студент 2 курсу, групи 1КІ-24м
спеціальності 123 – Комп'ютерна інженерія
_____ Заяц В.В.
Керівник к.т.н., доц. каф. ОТ
_____ Савицька Л.А
"15" 12 2025 р.
Опонент к.т.н., доц. каф. П.З. Рейда О.М.
"15" 12 2025 р. Рейда О.М.

Допущено до захисту
Завідувач кафедри ОТ
д.т.н. проф. Азаров О.Д.
"18" 12 2025 р.

ВНТУ 2025

ВІННИЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет інформаційних технологій та комп'ютерної інженерії

Кафедра обчислювальної техніки

Галузь знань — Інформаційні технології

Освітній рівень — магістр

Спеціальність — 123 Комп'ютерна інженерія

Освітньо-професійна програма — Комп'ютерна інженерія

ЗАТВЕРДЖУЮ

Завідувач кафедри обчислювальної техніки

проф., д.т.н. О.Д. Азаров

"25" 09. 2025 р.

З А В Д А Н Н Я

НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ

студенту **Зайцю Владиславу Валерійовичу**

1 Тема роботи «Система розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик» керівник роботи к.т.н., доц. каф. ОТ Савицька Л. А., затверджено наказом вищого навчального закладу від 24.09.2025 року № 313.

2 Строк подання студентом роботи 04.12.2025р.

3 Вихідні дані до роботи: виділений вузол, операційна система Linux, API-сервіс для розгортання, Kubernetes, Docker-артефакт, Grafana для візуалізації даних.

4 Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити): вступ, аналіз методів розгортання та моніторингу сучасних прикладних сервісів, обґрунтування вибору технологій для розгортання та моніторингу прикладних сервісів, реалізація системи розгортання та моніторингу прикладного сервісу, економічна частина, перелік джерел посилань.

5 Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень): Блок-схема алгоритму роботи сервісу, структурна схема розгортання прикладного сервісу, структурна схема кластеру.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1— Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
1 – 3	Савицька Л.А., к.т.н., проф. каф. ОТ		
4	Ратушняк О.Г., к.т.н., доц. каф. ЕПВМ		
Нормоконтроль	Асистент Швець С.І.		

7 Дата видачі завдання 25.09.2025.

8 Календарний план виконання МКР приведений в таблиці 2.

Таблиця 2 — Календарний план

№ з/п	Назва етапів виконання магістерської роботи	Строк виконання етапів роботи	Примітки
1	Постановка задачі роботи	26.09.2025	виконано
2	Аналіз предметної області	27.09.2025 01.10.2025	виконано
3	Проєктування системи розгортання прикладних сервісів	02.10.2025 12.10.2025	виконано
4	Реалізація системи розгортання та моніторингу прикладного сервісу	13.10.2025 27.10.2025	виконано
5	Розрахунок економічної частини	29.11.2025	виконано
6	Оформлення матеріалів до захисту МКР	01.11.2025	виконано
7	Перевірка якості виконання магістерської роботи	04.11.2025	виконано
8	Підписи супроводжувальних документів у нормконтролера, керівника, опонента	07.11.2025	виконано
9	Перевірка на антиплагіат та ШІ	10.11.2025	виконано
10	Попередній захист роботи	11.11.2025	виконано

Студент _____ Заяц В.І

Керівник роботи _____ Савицька Л.А

АНОТАЦІЯ

УДК 004.45

Заяц В.В. Система розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик. Магістерська кваліфікаційна робота зі спеціальності 123 — Комп'ютерна Інженерія, Вінниця: ВНТУ, 2025 — 113с.

На укр. мові. Бібліогр.: 35; рис.: 15; табл.: 5.

У роботі розглянуто створення системи автоматизованого розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик. Система забезпечує автоматизацію процесів конфігурації серверів, розгортання контейнеризованих застосунків, організацію CI/CD-конвеєра та безперервний моніторинг інфраструктури. Розроблена система підвищує ефективність розгортання, зменшує вплив людського фактора та забезпечує надійність серверних сервісів у продукційному середовищі.

Ключові слова: автоматизація розгортання, моніторинг, CI/CD, DevOps, Terraform, Ansible, Kubernetes, Prometheus.

ABSTRACT

Zayats V. V. System for Deployment and Monitoring of Application Services with Integration of DevOps Practices: Master's Qualification Thesis in Specialty 123 – Computer Engineering. Vinnytsia: VNTU, 2025. – 113 p.

In Ukrainian. Bibliography: 35; fig.: 15; tables: 5.

The work considers the creation of a system of automated deployment and monitoring of applied services with the integration of the Devops practitioner. The system provides automation of server configuration processes, the deployment of containerized applications, the CI/CD conveyor organization and continuous infrastructure monitoring. The developed system increases the efficiency of deployment, reduces the impact of the human factor and ensures the reliability of server services in the product environment.

Keywords: deployment automation, monitoring, CI/CD, Devops, Terraform, Ansible, Kubernetes, Prometheus

ЗМІСТ

ВСТУП

Актуальність теми дослідження полягає в тому, що в умовах стрімкого розвитку цифрових технологій і глобальної інформатизації сучасні організації дедалі частіше стикаються з потребою швидкого, надійного та відтворюваного розгортання прикладних сервісів. Зростання складності програмних систем, широке використання мікросервісної архітектури, контейнеризації, хмарних технологій та розподілених інфраструктур призводить до необхідності автоматизації всіх етапів життєвого циклу програмного забезпечення — від розробки до експлуатації. У цьому контексті набуває особливої важливості впровадження DevOps-практик, які поєднують процеси розробки (Development) та експлуатації (Operations) в єдиний інтегрований цикл безперервної інтеграції, доставки та моніторингу.

Традиційні методи розгортання систем передбачали ручне налаштування серверів, встановлення залежностей, конфігурування мережевих параметрів і підтримку інфраструктури адміністраторами. Такий підхід був ефективним лише у невеликих масштабах і не відповідав вимогам сучасного середовища, яке потребує гнучкості, швидкості оновлення та мінімізації людського фактору. Впровадження концепцій Infrastructure as Code (IaC), Continuous Integration/Continuous Deployment (CI/CD) та автоматизованого моніторингу дозволяє значно підвищити ефективність управління інфраструктурою, забезпечити стабільність і прогнозованість роботи сервісів.

Інтеграція DevOps-практик забезпечує низку переваг: прискорення доставки нових версій програмного забезпечення, підвищення надійності систем за рахунок автоматичного тестування та розгортання, гнучке масштабування інфраструктури, а також зменшення часу на усунення інцидентів завдяки системам моніторингу та сповіщень. Водночас, побудова цілісної системи DevOps вимагає глибокого розуміння архітектури, принципів автоматизації та використання широкого спектра інструментів — таких як Terraform, Ansible, Docker, Kubernetes, Jenkins, Prometheus та Grafana.

Метою роботи є вдосконалення системи розгортання та моніторингу прикладних сервісів за рахунок автоматизації із інтеграцією DevOps-практик, що забезпечує прискорення процесів створення інфраструктури, конфігурації серверів, розгортання контейнеризованих застосунків, а також безперервний моніторинг і виявлення відмов у системі.

Для досягнення поставленої мети необхідно вирішити такі **задачі:**

— провести аналітичний огляд сучасних DevOps-технологій і методів автоматизації розгортання та моніторингу сервісів;

— обґрунтувати вибір технологій для розгортання і моніторингу прикладних сервісів;

— розробити програмну реалізацію api-сервіса як прикладного сервісу для розгортання і моніторингу;

— реалізувати систему розгортання та моніторингу прикладного api-сервіса

— розрахувати витрати в економічній частині.

В роботі використовуються такі **методи дослідження:** методи системного аналізу, методи моделювання програмно-апаратних систем, методи об’єктно-орієнтованого та модульного програмування

Об’єктом дослідження є процеси автоматизованого розгортання, управління та моніторингу прикладних сервісів у серверних і контейнерних середовищах.

Предметом дослідження є методи, інструменти та технології інтеграції DevOps-практик у процеси розробки, розгортання та супроводу прикладних сервісів.

Наукова новизна роботи полягає у вдосконаленій комплексній моделі системи розгортання та моніторингу за рахунок автоматизації з інтеграцією DevOps практик, яка поєднує підходи IaC, CI/CD і моніторинг у єдиній автоматизованій архітектурі. Розроблена система дозволяє автоматизувати конфігурації інфраструктури, забезпечує повторюваність і масштабованість розгортання, а також надає механізми оперативного виявлення відмов і збоїв. В

роботі запропоновано інтеграційний підхід, який може бути використаний для побудови ефективних DevOps-рішень у реальних виробничих середовищах.

Практичне значення роботи полягає у можливості використання розробленої системи для автоматизації процесів керування прикладними сервісами в корпоративних інфраструктурах. Вона може бути впроваджена в діяльність DevOps-команд, системних адміністраторів і розробників для підвищення надійності сервісів, скорочення часу розгортання та покращення контролю за роботою серверів. Отримані результати можуть бути застосовані як у виробничих середовищах малого та середнього бізнесу, так і в освітньому процесі для підготовки фахівців у галузі комп'ютерної інженерії, системного адміністрування та DevOps-інженерії.

Апробація результатів роботи була здійснена на міжнародних науково-практичних інтернет-конференціях «Молодь в науці: дослідження, проблеми, перспективи – 2025,2026»

Публікації за результатами досліджень опубліковано тези доповідей на науково-технічній конференції:

Заяц В.В., Савицька Л.А., Система розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик Конференція ВНТУ: Молодь в науці: дослідження, проблеми, перспективи (МН-2026) [1]. Режим доступу: https://conferences.vntu.edu.ua/index.php/mn/mn2026/paper/view/26836.

1 АНАЛІЗ МЕТОДІВ РОЗГОРТАННЯ ТА МОНІТОРИНГУ СУЧАСНИХ ПРИКЛАДНИХ СЕРВІСІВ

DevOps — це набір практик, інструментів та культурної філософії, яка автоматизує та інтегрує процеси між розробкою програмного забезпечення та IT командами. DevOps-підхід — інтеграція методологій Development (розробки) та Operations (експлуатації), спрямована на скорочення циклу доставки програмного забезпечення, підвищення стабільності та контрольованості середовища. Основна ідея DevOps полягає в тому, щоб забезпечити безперервність усіх етапів життєвого циклу програмного продукту — від написання коду до його розгортання в продуктивному середовищі. Цей підхід ґрунтується на принципах автоматизації, відтворюваності та моніторингу, що реалізуються за допомогою спеціалізованих інструментів і технологій. Основними цілями DevOps є:

— скорочення часу виходу продукту на ринок;

— зниження частоти відмов нових релізів;

— мінімізація часу, необхідного для усунення помилок;

— зменшення тривалості відновлення системи у випадку збоїв чи відключень [2].

Впровадження DevOps сприяє автоматизації та програмуванню рутинних процесів, що забезпечує підвищення передбачуваності, ефективності, безпеки та ремонтопридатності операційної діяльності інформаційних систем. Інтеграція DevOps передбачає безперервну доставку програмного забезпечення, постійне тестування, контроль якості, розробку нових функцій та впровадження оновлень обслуговування з метою підвищення стабільності та безпеки систем. Важливою складовою цього підходу є тісна взаємодія між командами розробки та експлуатації, що дозволяє оперативно реагувати на зміни вимог і швидко усувати виявлені дефекти. Такий підхід забезпечує коротший цикл розробки та розгортання програмних продуктів, зменшує ризики помилок під час впровадження змін і є ключовим чинником конкурентоспроможності сучасних IT-компаній.

Абстрактне представлення життєвого циклу DevOps зображено на рисунку 1.1.

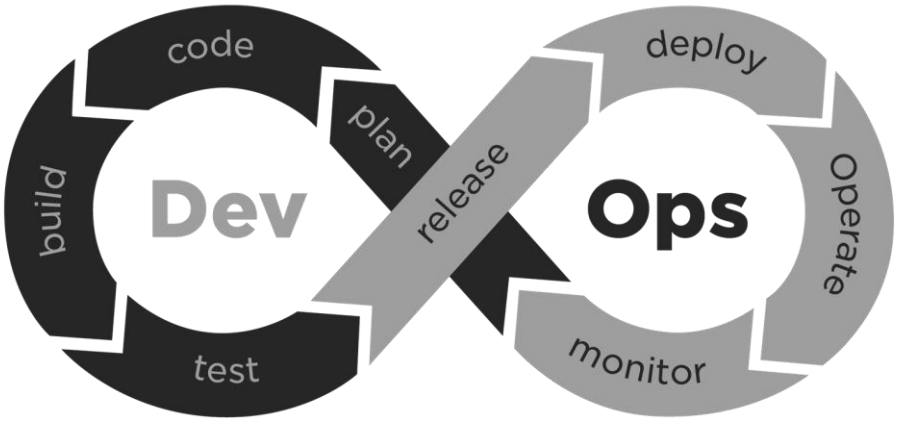

Рисунок 1.1 — Абстарктне представлення життєвого циклу DevOps

DevOps сприяє стандартизації середовища розробки та управління випусками програмного забезпечення, що дає можливість ефективно відстежувати події, документувати процеси та формувати детальні звіти. Завдяки цьому розробники отримують більше контролю над середовищем, а інфраструктура стає більш прозорою, керованою та орієнтованою на розуміння [3].

1.1 Аналіз підходу інфраструктури як коду

Одним із базових напрямів DevOps є концепція інфраструктури як коду (IaC), яка дозволяє описувати серверні ресурси, мережеві компоненти та середовище виконання у вигляді конфігураційних файлів.

Інфраструктура як код (Infrastructure as Code, IaC) — це підхід до управління обчислювальними ресурсами, який передбачає створення, налаштування та підтримку інфраструктури за допомогою коду, а не шляхом ручного налаштування та конфігурації. Будь-яке середовище для роботи програмного забезпечення потребує низки інфраструктурних компонентів, таких як операційні системи, бази даних, мережеві з’єднання, системи зберігання даних тощо. Розробники постійно налаштовують, оновлюють та обслуговують цю інфраструктуру для забезпечення процесів розроблення, тестування та розгортання застосунків [6].

Традиційне ручне управління інфраструктурою є трудомістким і схильним до помилок, особливо у випадках масштабного керування системами. Підхід IaC дозволяє описати бажаний стан інфраструктури без необхідності визначати кожен

крок досягнення цього стану. Таким чином, управління інфраструктурою автоматизується, що дає змогу розробникам зосередитися на вдосконаленні програмних продуктів, а не на налаштуванні середовищ. Використання IaC допомагає організаціям контролювати витрати, зменшувати ризики та швидше реагувати на нові бізнес-можливості.

Автоматизація є ключовою метою будь-якого обчислювального середовища. IaC застосовується для автоматизованого створення та управління інфраструктурними середовищами, насамперед у розробці програмного забезпечення для побудови, тестування та розгортання застосунків. У минулому системні адміністратори використовували поєднання скриптів і ручних дій для створення середовищ, що було складним і тривалим процесом. Завдяки IaC сьогодні можна автоматично розгорнути повноцінне середовище за лічені хвилини та ефективніше його підтримувати. Основні переваги IaC полягають у наступному:

— легка реплікація середовищ;

— зменшення кількості помилок у конфігураціях;

— ітеративний розвиток найкращих практик.

Подібно до того, як програмний код описує функціональність застосунку, IaC описує архітектуру системи — сервери, мережі, операційні системи, сховища тощо. Цей підхід дає змогу керувати віртуалізованими ресурсами у повторюваний і контрольований спосіб, використовуючи конфігураційні файли, що зберігаються як вихідний код.

IaC-інструменти підтримують різні мови опису інфраструктури. Код можна створювати у звичних середовищах розробки (IDE), із підтримкою перевірки синтаксису, керування версіями та внесення змін через систему контролю версій. Файли IaC інтегруються у загальну кодову базу проєкту, забезпечуючи прозорість та узгодженість усіх компонентів.

Існують два основні підходи до реалізації інфраструктури як коду:

— декларативний підхід;

— імперативний підхід.

При використанні декларативного підходу розробник описує кінцевий бажаний стан інфраструктури — які ресурси та налаштування мають бути доступними. Система автоматично створює це середовище на основі опису. Декларативний підхід є простішим, якщо відомо, яких саме компонентів потребує застосунок;

При використанні імперативного підходу розробник описує послідовність кроків, необхідних для досягнення бажаного стану інфраструктури. Імперативний IaC є складнішим, однак дає більше контролю у складних сценаріях, де критичним є порядок виконання дій [8].

Інтеграція IaC у процеси CI/CD (Continuous Integration / Continuous Deployment) дозволяє автоматично застосовувати зміни інфраструктури разом із оновленням програмного коду. Це забезпечує узгодженість середовищ, прискорює розгортання і підвищує стабільність релізів.

DevOps-команди використовують IaC для таких завдань:

— швидке створення повних середовищ — від розробницьких до продукційних;

— забезпечення відтворюваності конфігурацій між різними етапами життєвого циклу ПЗ;

— інтеграції з хмарними провайдерами та ефективного масштабування ресурсів залежно від навантаження;

— забезпечення спільної мови для розробників і адміністраторів, що підвищує прозорість змін і сприяє співпраці в межах DevOps-культури.

1.2 Управління конфігураціями

Управління конфігураціями (Configuration Management) — це процес забезпечення того, щоб конфігурації серверів, застосунків та інших компонентів інформаційної системи залишалися відомими, узгодженими та надійними протягом усього життєвого циклу системи. Будь-яка IT-система має певні конфігураційні параметри, пов’язані з версіями програмного забезпечення, налаштуваннями безпеки, мережевими характеристиками та іншими аспектами, що визначають її

оптимальне функціонування. Управління конфігураціями забезпечує відстеження, оновлення та підтримку цих параметрів, щоб система працювала відповідно до встановленого базового рівня продуктивності та залишалася захищеною від можливих збоїв чи несанкціонованих змін. Цей процес спрямований на підтримання узгодженості між фактичним станом системи, її функціональними характеристиками, вимогами проєктування та експлуатаційною документацією.

Конфігураційні дані — це набір параметрів, які визначають спосіб функціонування інформаційної системи. До розгортання системи адміністратор задає низку конфігураційних параметрів, таких як розподіл пам’яті, апаратні ресурси, права доступу користувачів, мережеві налаштування тощо, з метою забезпечення оптимальної роботи системи.

Однак, у динамічному бізнес-середовищі вимоги до систем постійно змінюються. Це зумовлює потребу в ефективному механізмі управління конфігураціями, який дозволяє оперативно адаптувати систему до нових умов, зберігаючи при цьому її стабільність, безпеку та узгодженість між усіма середовищами — розробки, тестування та продуктивного розгортання.

Оновлення конфігураційних параметрів є складним завданням через наявність численних взаємозалежностей між компонентами системи, широке використання мікросервісної архітектури та складні вимоги до управління даними. За відсутності централізованої системи контролю швидко виникає явище, відоме як “дрейф конфігурації” (configuration drift) — ситуація, коли поточні налаштування системи перестають відповідати встановленим бізнес-вимогам або іншим середовищам.

Наприклад, якщо в середовищі розробки було змінено певні параметри, ці самі зміни повинні бути точно відтворені у тестовому та продукційному середовищах. Пропуск або некоректне виконання цього процесу призводить до розбіжностей у конфігураціях, що може спричинити збої, проблеми з безпекою або непередбачувану поведінку системи [9].

Управління конфігураціями дає змогу централізовано контролювати, відстежувати та оновлювати конфігураційні дані, які використовуються

програмним забезпеченням упродовж усього його життєвого циклу. Такий підхід забезпечує єдину базу даних конфігурацій, що виступає еталонним станом (baseline) для всієї системи.

Замість ручного редагування конфігураційних файлів на окремих апаратних або програмних платформах, адміністратори можуть застосовувати інструменти управління конфігураціями (configuration management tools), які дозволяють безпечно, послідовно та поступово впроваджувати зміни у всіх середовищах.Крім того, такі системи автоматично документують усі внесені зміни, що створює передумови для аналізу їхнього впливу на продуктивність та стабільність системи. Це підвищує прозорість процесів, спрощує аудит змін і забезпечує збереження історії конфігурацій.

Організації впроваджують керування конфігурацією з метою уніфікації параметрів налаштувань у різних IT-системах. Нижче наведено основні переваги, які забезпечує керування конфігурацією.

Інженери-програмісти використовують інструменти керування конфігурацією для створення базової (еталонної) конфігурації функціонального програмного забезпечення. У процесі впровадження нових функцій деякі параметри можуть змінюватися, що впливає на стабільність системи. Завдяки керуванню конфігурацією можна запобігти порушенню роботи сервісу, оперативно відновлюючи або повертаючи попередні стабільні значення конфігурації;

Неправильні налаштування програмного забезпечення часто призводять до збоїв у роботі сервісів, що негативно позначається на працівниках і клієнтах, які залежать від цих IT-систем. Системи керування конфігурацією документують і зберігають усі зміни в централізованому сховищі. Це дозволяє відтворити середовище, де було виявлено помилку, для подальшого аналізу та усунення причин несправності, що знижує ризик дороговартісних зупинок системи;

Комп’ютерні системи працюють із десятками параметрів конфігурації, які змінюються з часом. Інструменти керування конфігурацією дають змогу відстежувати причини внесення змін, аналізувати їхній вплив на роботу програмного забезпечення та визначати, хто саме відповідає за конкретні

оновлення. Наявність прозорого журналу змін полегшує командну роботу, забезпечує комунікацію між розробниками й підтримує контроль над якістю оновлень.

Під час розроблення програмного забезпечення команди витрачають значний час на тестування різних варіантів конфігурацій. Завдяки системам керування конфігурацією ці процеси можна автоматизувати, спостерігати за змінами та аналізувати їх у тестових і продуктивних середовищах. Це дозволяє легко імітувати роботу системи в реальних умовах, змінюючи параметри без порушення базових значень.

Керування конфігурацією забезпечує гнучке застосування параметрів на різних етапах життєвого циклу програмного забезпечення. Наприклад, під час розроблення програми на фізичних машинах ресурси можуть бути обмеженими, однак у продуктивному середовищі вони повинні відповідати реальним вимогам. Система керування конфігурацією дозволяє легко переходити між різними версіями налаштувань, не вдаючись до ручного редагування.

Керування конфігурацією здійснюється шляхом написання сценаріїв (скриптів) або програмного коду, які автоматизують налаштування всіх компонентів системи. Це дає змогу фіксувати, уніфікувати й оптимізувати метадані, такі як параметри апаратних ресурсів, ключі API, точки доступу до баз даних тощо.

Основні етапи процесу такі:

— інженери систем збирають дані з різних апаратних і програмних середовищ за допомогою інструментів керування конфігурацією;

— отримані дані зберігаються в централізованому сховищі — базі даних керування конфігурацією (Configuration Management Database, CMDB), до якої мають доступ усі зацікавлені сторони;

— конфігураційні дані перевіряються фахівцями для забезпечення їхньої оптимальності та відповідності робочим вимогам системи;

— за допомогою автоматизованих інструментів створюється наскрізний конвеєр (pipeline) керування конфігурацією, який використовується під час

розгортання навантажень у приватних, публічних або гібридних хмарних середовищах.

1.3 Контейнеризація та оркестрація прикладних систем

Контейнеризація — це процес розгортання програмного забезпечення, який полягає в об’єднанні коду застосунку з усіма необхідними для його роботи бібліотеками, конфігураційними файлами та залежностями у єдиний ізольований пакет — контейнер. Такий контейнер може бути виконаний на будь-якій обчислювальній інфраструктурі, незалежно від операційної системи чи апаратного середовища.

Традиційно для запуску програмного забезпечення на певному комп’ютері необхідно було встановити його версію, сумісну з операційною системою цього пристрою. Наприклад, версію програми для Windows слід було інсталювати лише на комп’ютерах під керуванням Windows. Натомість контейнеризація дозволяє створювати універсальні пакети, здатні функціонувати на різних платформах без необхідності адаптації або повторної компіляції коду [12].

Розробники програмного забезпечення активно застосовують контейнеризацію для створення, тестування та розгортання сучасних застосунків завдяки її численним перевагам.

Однією з головних переваг контейнеризації є портативність. Програмісти можуть створювати застосунок один раз і розгортати його у різних середовищах без переписування коду. Той самий контейнер може бути виконаний як у середовищі Linux, так і у Windows. Це спрощує перенесення програм між локальними серверами, віртуальними машинами та хмарними інфраструктурами.Крім того, контейнеризація дає змогу модернізувати застарілі застосунки, переносячи їх у контейнерне середовище без повного переписування програмного коду.

Контейнери є легковаговими програмними одиницями, які ефективно використовують ресурси системи. На відміну від віртуальних машин, контейнер не потребує окремого завантаження операційної системи, тому його запуск

відбувається набагато швидше. Це дозволяє масштабувати застосунок горизонтально — розгортати декілька контейнерів із тим самим застосунком на одному фізичному сервері або в межах кластеру. Всі контейнери спільно використовують ресурси операційної системи, але працюють ізольовано, що запобігає взаємному впливу між ними.

Контейнеризація сприяє підвищенню надійності та відмовостійкості програмних систем. Розробники використовують кілька контейнерів для реалізації мікросервісної архітектури, у якій кожен контейнер виконує окрему функцію. Оскільки кожен контейнер працює в ізольованому середовищі користувача, збій одного з них не впливає на роботу інших. Це забезпечує високу доступність сервісів та швидке відновлення працездатності у випадку помилки.

Контейнеризовані застосунки функціонують в ізольованих обчислювальних середовищах, що спрощує налагодження, тестування та оновлення програмного коду. Розробники можуть змінювати логіку застосунку без втручання в роботу операційної системи чи інших сервісів. Такий підхід підвищує агільність (agility) процесу розробки, дозволяє скоротити життєвий цикл випуску нових версій і прискорити оновлення системи.

Контейнеризація полягає у створенні самодостатніх програмних пакетів, які забезпечують стабільне виконання застосунку незалежно від середовища. Розробники створюють і розгортають контейнерні образи (container images) — файли, що містять всю необхідну інформацію для запуску застосунку. Ці образи створюються згідно зі специфікацією Open Container Initiative (OCI) — відкритим стандартом для формування контейнерних образів. Контейнерні образи є незмінними (read-only), що гарантує повторюваність і надійність процесу розгортання. У типову архітектуру контейнеризації входять такі шари:

— інфраструктура;

— операційна система;

— контейнерний рушій;

— застосунок і залежності.

Найнижчий рівень контейнерної архітектури складається з апаратного забезпечення або фізичного сервера, на якому працюють контейнери.

На другому рівні розміщується операційна система. Найпопулярнішою платформою для контейнеризації є Linux, однак у хмарних середовищах часто застосовуються сервіси на основі AWS EC2, Google Compute Engine або інших аналогів.

Контейнерний рушій це програмне забезпечення, яке створює контейнери на основі образів і управляє ними. Рушій виступає посередником між операційною системою та контейнерами, забезпечуючи ізоляцію ресурсів. Він дозволяє одночасно запускати декілька контейнерів на одному сервері, розподіляючи між ними ресурси процесора, пам’яті та мережі.

На верхньому рівні знаходиться безпосередньо застосунок, бібліотеки, файли конфігурації та інші необхідні залежності. У деяких випадках контейнер може містити мінімальну гостьову операційну систему, яка виконується поверх базової ОС хоста.

Оркестрація контейнерів — це технологія автоматизованого управління контейнерами у масштабованих середовищах. Вона є критично необхідною для сучасної розробки хмарних застосунків, які можуть містити сотні або навіть тисячі контейнерів, що реалізують мікросервісну архітектуру. Ручне керування такою кількістю контейнерів є практично неможливим, тому використовуються спеціальні системи оркестрації.

Інструменти оркестрації дозволяють автоматично створювати, запускати, масштабувати та керувати контейнерами. Вони забезпечують рівномірний розподіл навантаження, балансування ресурсів, оновлення без простоїв і самовідновлення сервісів у випадку збою.

Команди, що займаються оркестрацією програмного забезпечення, зазвичай використовують інструменти оркестрації контейнерів, такі як Kubernetes або Docker Swarm. Процес починається з опису конфігурації застосунку у спеціальному конфігураційному файлі, який визначає джерела отримання контейнерних образів, а також способи взаємодії та мережевої комунікації між контейнерами. Система

оркестрації на основі цих даних планує розгортання контейнерів у межах кластеру та визначає оптимальний хост для їх розміщення відповідно до заздалегідь встановлених обмежень, таких як мітки (labels) або метадані. Далі інструмент автоматично керує життєвим циклом контейнерів, забезпечуючи їх запуск, масштабування, оновлення та завершення роботи згідно з параметрами, зазначеними у конфігураційному описі.

Основна мета оркестрації полягає у тому, щоб забезпечити централізоване керування великою кількістю контейнерів у складних розподілених системах. Автоматизація оркестрації дозволяє масштабувати застосунки однією командою, швидко створювати нові екземпляри контейнеризованих сервісів для обробки зростаючого трафіку, а також спрощує процес інсталяції та оновлення.

Крім того, автоматизована оркестрація сприяє підвищенню безпеки системи, оскільки забезпечує контрольований розподіл навантаження, ізоляцію контейнерів і централізоване застосування політик доступу. Таким чином, оркестрація контейнерів є невід'ємною складовою сучасних DevOps-практик, що поєднує автоматизацію, надійність і масштабованість у процесах розгортання прикладних сервісів.

Kubernetes — це програмне забезпечення з відкритим кодом для оркестрації контейнерів, яке спрощує керування контейнеризованими застосунками у масштабованих середовищах. Система автоматизує процеси планування, запуску, зупинки та моніторингу контейнерів, а також виконує різноманітні функції адміністрування. Завдяки цьому розробники отримують усі переваги контейнеризації без необхідності виконувати рутинні операції з управління інфраструктурою.

Pod є базовою одиницею розгортання у Kubernetes. Кожен pod може містити один або декілька контейнерів, які спільно використовують системні ресурси, зокрема файлове сховище, мережеві інтерфейси та IP-адресу.Контейнери в межах одного pod'а не є ізольованими один від одного — вони можуть взаємодіяти напряму через спільний мережевий простір і файлову систему. Таким чином, pod можна розглядати як аналог віртуальної машини (VM), у межах якої контейнери

функціонують подібно до окремих застосунків. Pod’и або групи pod’ів можна класифікувати за допомогою міток (labels), що дає змогу логічно структурувати їх за призначенням, наприклад, позначаючи середовище розробки (dev) або промислове середовище (prod).

Node — це обчислювальний вузол, на якому виконуються pod’и. Node може бути фізичним сервером або віртуальною машиною, наприклад, екземпляром Amazon EC2. Кожен вузол містить набір компонентів, необхідних для функціонування Kubernetes:

Kubelet — агент, який відповідає за керування pod’ами, моніторинг їх стану та комунікацію з контролером кластера;

Kube-proxy — мережевий проксі-сервер, який забезпечує маршрутизацію трафіку до pod’ів та підтримує мережеві правила;

Container runtime — рушій контейнерів, необхідний для безпосереднього запуску контейнерів на вузлі. Kubernetes підтримує декілька типів контейнерних рушіїв, зокрема реалізації, сумісні зі специфікацією Container Runtime Interface (CRI).

Окремий pod є самостійним об’єктом і не відновлюється автоматично у разі збою вузла, на якому він працює. Для забезпечення відмовостійкості та масштабування використовується об’єкт ReplicaSet, який гарантує, що певна кількість подібних pod’ів завжди буде запущена у кластері. Це дозволяє системі автоматично створювати нові pod’и у випадку відмови окремого вузла або збільшення навантаження.

Deployment є вищим рівнем абстракції в Kubernetes, що керує ReplicaSet-ами. Він відповідає за розгортання, оновлення та відкат (rollback) версій застосунків без переривання їхньої роботи. Це забезпечує безперервність сервісів (zero-downtime updates) під час оновлення програмного забезпечення.

Для забезпечення доступу до pod’ів через мережу використовується об’єкт Service. Він виступає у ролі мережевої абстракції, яка дозволяє взаємодіяти з pod’ами за єдиною IP-адресою або DNS-іменем, незалежно від їх кількості чи місця розташування в кластері.

Коли потрібно надати доступ до сервісу ззовні, використовується об’єкт Ingress, який визначає правила маршрутизації HTTP/HTTPS-запитів від зовнішнього трафіку до відповідних pod’ів. Ingress зазвичай реалізується за допомогою спеціального контролера (наприклад, NGINX Ingress Controller або Traefik).

Кластер Kubernetes складається з одного або кількох вузлів (nodes), на яких виконуються pod’и. Ключовим елементом архітектури є площина керування (control plane), яка відповідає за координацію роботи всіх компонентів кластера.

Основні компоненти площини керування:

— kube-apiserver — центральний компонент, що забезпечує взаємодію між користувачами, інструментами та внутрішніми елементами Kubernetes через REST API;

— etcd — розподілене сховище ключ-значення, у якому зберігається поточний і бажаний стан кластера;

— kube-scheduler — модуль, який визначає, на якому вузлі буде запущено новий pod, враховуючи ресурси, політики розміщення та обмеження.

— kube-controller-manager — компонент, що містить набір контролерів, відповідальних за підтримку бажаного стану об’єктів (наприклад, контролер вузлів, контролер реплік, контролер завдань);

— cloud-controller-manager — компонент, який забезпечує інтеграцію Kubernetes із хмарними платформами (AWS, Google Cloud, Azure тощо), керуючи специфічними ресурсами, такими як балансувальники навантаження або дискові томи [16].

Оскільки контейнери за своєю природою є ефемерними (тимчасовими) і не мають власного постійного сховища, виникає потреба у механізмах persistent storage для збереження даних, які мають існувати незалежно від життєвого циклу контейнера. Для цього Kubernetes використовує об’єкти PersistentVolume (PV) та PersistentVolumeClaim (PVC).

Постійні томи (PV) додаються до кластера як ресурси зберігання, які можуть бути прив’язані до pod’ів. Вони можуть базуватися на локальних дисках,

мережевих файлових системах (NFS), або хмарних сервісах зберігання (наприклад, Amazon EBS, Google Persistent Disk). Таке рішення дозволяє pod'ам зберігати дані між перезапусками або навіть після повного видалення контейнерів, забезпечуючи стабільність і надійність критичних застосунків.

1.4 Практики безперервної інтеграції та безперервного доставлення

У галузі програмної інженерії CI/CD (або CICD) — це поєднання практик безперервної інтеграції (Continuous Integration, CI) та безперервного доставлення (Continuous Delivery, CD), або, рідше, безперервного розгортання (Continuous Deployment). Ці підходи іноді узагальнено називають безперервною розробкою (Continuous Development) або безперервною розробкою програмного забезпечення (Continuous Software Development).

CI/CD належить до методології DevOps, яка об'єднує команди розробників і адміністраторів (Development + Operations) та поєднує практики безперервної інтеграції (Continuous Integration, CI) і безперервного доставлення (Continuous Delivery, CD).

Основна мета CI/CD полягає в автоматизації більшої частини — або навіть усіх — етапів, які раніше вимагали ручного втручання для перенесення нового коду з репозиторію у виробниче середовище. До таких етапів належать: збирання (build), тестування (включно з модульними, інтеграційними та регресійними тестами), розгортання (deploy) та підготовка інфраструктури (infrastructure provisioning) [18].

За наявності конвеєра CI/CD (CI/CD pipeline) команди розробників можуть вносити зміни до коду, які автоматично проходять тестування і готуються до доставлення у цільове середовище. Правильно налаштований процес CI/CD мінімізує простої, прискорює випуск нових версій та забезпечує високу стабільність програмного продукту.

CI/CD є надзвичайно важливим компонентом сучасної інженерії програмного забезпечення, оскільки:

— прискорює доставлення програмного продукту, забезпечуючи швидший вихід оновлень на ринок;

— зменшує простої системи, завдяки ранньому виявленню помилок;

— підвищує якість продукту, оскільки більшість процесів перевірки і тестування виконується автоматично.

У сучасних високотехнологічних середовищах безперервна інтеграція (CI) та безперервне доставлення (CD) перестали бути лише модними тенденціями — вони стали основою сучасного процесу розробки програмного забезпечення. CI/CD забезпечує автоматизацію всього життєвого циклу розробки — від написання коду до його розгортання. Це дає змогу командам швидше й частіше випускати нові функції та оновлення, підвищуючи гнучкість продукту та його здатність відповідати на потреби користувачів.

Завдяки постійній інтеграції та доставленню помилки виявляються на ранніх етапах, що скорочує простої та підвищує загальну надійність системи. Крім того, CI/CD сприяє формуванню швидких циклів зворотного зв’язку зі стейкхолдерами, допомагаючи командам краще узгоджувати функціональність із очікуваннями користувачів. Таким чином, CI/CD є фундаментальною практикою для будь-якої організації, яка прагне досягти високої швидкості, надійності та якості розробки.

Безперервна інтеграція (CI) забезпечує ранню та регулярну перевірку змін у коді за допомогою автоматизованого збирання та тестування, що скорочує кількість помилок і прискорює розробку. Суть CI полягає у частому об’єднанні змін коду в основну гілку спільного репозиторію. Кожне злиття (commit або merge) супроводжується автоматичним запуском процесів тестування й збирання. Такий підхід дозволяє вчасно виявляти помилки, конфлікти коду чи проблеми безпеки.

Переваги CI:

— виявлення помилок на ранньому етапі, коли їх легше усунути;

— мінімізація конфліктів між розробниками, навіть у великих командах;

— швидке отримання зворотного зв’язку про стан коду.

Типовий процес перевірки коду в CI починається зі статичного аналізу, який визначає якість коду. Якщо він проходить цю перевірку, система CI автоматично збирає і компілює програму, після чого запускає набір автоматичних тестів. CI-процеси завжди повинні працювати у поєднанні із системою контролю версій (Git,

SVN), яка відстежує зміни та дозволяє чітко визначати версію коду, що використовується на певному етапі розробки.

Безперервне доставлення (CD) — це процес автоматичної підготовки протестованого коду таким чином, щоб він у будь-який момент був готовий до розгортання у будь-якому середовищі. Ця практика працює у тісному зв’язку з CI, автоматизуючи процеси підготовки інфраструктури, збирання застосунку та його випуску.

Після того, як код пройшов етапи тестування і збирання у рамках CI, процес CD забезпечує пакування, конфігурацію і доставлення застосунку до цільового середовища — тестового, staging або production. Таким чином, програмне забезпечення завжди підтримується у стані, готовому до миттєвого розгортання. Саме CD гарантує сталість і надійність випусків, зменшуючи ризики, пов’язані з ручним втручанням під час публікації.

Безперервне розгортання (Continuous Deployment) усуває необхідність ручного виконання кроків публікації, дозволяючи автоматично доставляти нові версії програмного забезпечення у виробниче середовище. У цьому випадку команди DevOps заздалегідь визначають критерії якості та стабільності, після чого система самостійно розгортає код у production, якщо всі тести пройдено успішно.

Безперервне розгортання забезпечує:

— значне прискорення випуску функцій;

— зниження кількості помилок через людський фактор;

— оперативну реакцію на зворотний зв’язок від користувачів.

Таким чином, організації можуть бути більш гнучкими й швидше надавати користувачам нові можливості. Варто зазначити, що хоча безперервну інтеграцію можна реалізувати без доставлення або розгортання, CD неможливе без CI, оскільки процес доставлення базується на принципах постійної інтеграції, тестування та збирання невеликих змін.

Конвеєр CI/CD — це автоматизований процес, який використовується командами розробників для оптимізації створення, тестування та розгортання програмних продуктів. CI (Continuous Integration) означає постійне злиття змін у

центральний репозиторій для раннього виявлення проблем. CD (Continuous Delivery / Continuous Deployment) автоматизує випуск застосунку до цільового середовища, забезпечуючи його готовність до використання у будь-який момент.

Конвеєр CI/CD є ключовим інструментом для команд, які прагнуть підвищити якість програмного забезпечення та скороти́ти час його доставлення шляхом регулярних, передбачуваних оновлень. Впровадження CI/CD-конвеєра у процес розробки суттєво знижує ризик помилок під час розгортання, а автоматизація збирання й тестування гарантує раннє виявлення і швидке усунення дефектів, підтримуючи високу стабільність та надійність програмних систем.

Jenkins — це сервер безперервної інтеграції (Continuous Integration, CI) з відкритим вихідним кодом, призначений для автоматизації процесів збирання, тестування, документування, пакування та аналізу програмного забезпечення. Jenkins є одним із найпопулярніших інструментів у DevOps-середовищі, який використовується тисячами команд розробників по всьому світу.

Основна мета Jenkins полягає у централізованому керуванні та контролі різних етапів життєвого циклу розробки програмного забезпечення, що дозволяє зменшити обсяг ручної роботи та підвищити стабільність процесів розгортання.

Jenkins базується на модульній, розподіленій архітектурі, що дозволяє масштабувати процеси побудови і тестування. Основними елементами системи є Jenkins Controller (раніше — Master) та Jenkins Agent (раніше — Slave).

Контролер виконує функції організатора та координатора всієї системи. Він містить:

— центральну конфігурацію Jenkins;

— набір плагінів і засобів керування агентами;

— інтерфейс користувача;

— систему розподілу завдань між вузлами.

Контролер може самостійно виконувати збірки, однак зазвичай він лише розподіляє навантаження між агентами. Його основне завдання — керування процесами CI/CD, моніторинг вузлів, планування задач та взаємодія з репозиторіями коду. Контролер може також самостійно виконувати збірки, однак

у великих середовищах це не є ефективним, оскільки призводить до навантаження на центральний сервер.

Агент Jenkins — це вузол, який під’єднується до контролера та безпосередньо виконує завдання збірки. Агент може бути розгорнутий на фізичній машині, віртуальному сервері, хмарному екземплярі, Docker-контейнері або в кластері Kubernetes. Використання кількох агентів дозволяє розподіляти навантаження між серверами, забезпечувати паралельне виконання збірок, скорочувати час виконання конвеєрів і створювати безпечне середовище ізольоване від контролера.

Усі вузли, як агенти, так і контролери, у термінології Jenkins називаються “nodes” (вузлами). Система постійно моніторить їхній стан, а в разі виявлення відхилень або погіршення продуктивності автоматично переводить вузол у режим офлайн.

Робота Jenkins базується на подіях, що ініціюють процеси автоматизації. Найчастіше Jenkins реагує на зміни в коді, зафіксовані у репозиторіях GitHub, GitLab, або Bitbucket. Коли розробник створює новий commit, Jenkins запускає процес збірки, який може включати компіляцію, тестування, аналіз якості коду та створення артефактів.

У великих проєктах для підвищення ефективності використовується розподілена архітектура Jenkins, де контролер розподіляє навантаження між численними агентами. Кожен агент може виконувати окрему збірку або тестування різних версій програмного продукту, що дозволяє виконувати CI/CD-процеси паралельно. У Jenkins проєкт (Project) або завдання (Job) — це автоматизований процес, створений користувачем Jenkins. Базова дистрибуція пропонує широкий набір типів завдань: збірка застосунку, запуск тестів, створення документації тощо. Додаткові типи завдань можуть бути додані за допомогою плагінів (Plugins).

Jenkins Pipeline — це модель опису автоматизованого конвеєра, що визначає послідовність кроків (build, test, deploy, security scan тощо), які необхідно виконати в процесі CI/CD. Пайплайни можна створювати безпосередньо в інтерфейсі Jenkins або за допомогою файлу Jenkinsfile, який описує конвеєр у вигляді коду. Jenkinsfile використовує Groovy-сумісну мову, що підтримує декларативний або скриптовий

синтаксис, що дозволяє описати як прості, так і складні процеси автоматизації. Таким чином, Jenkins реалізує принцип “Pipeline as Code”, забезпечуючи прозорість, контроль версій і повторюваність процесів розгортання [19].

Переваги Jenkins:

— висока розширюваність — наявність великої кількості плагінів забезпечує підтримку майже всіх сучасних технологій і дозволяє створювати складні конвеєри;

— гнучкість і стабільність — Jenkins довів свою надійність у великих проєктах і може працювати у гібридних та мультихмарних середовищах;

— зрілість екосистеми — проект має багаторічну історію, активну спільноту, розгалужену базу знань і документацію;

— підтримка розподілених збірок — дозволяє паралельно виконувати завдання на різних вузлах;

— підтримка корпоративних технологій — оскільки Jenkins написано мовою Java, він добре інтегрується із корпоративними середовищами, які використовують цю екосистему.

Недоліки Jenkkins:

— архітектурні обмеження — Jenkins використовує монолітну архітектуру з єдиним сервером контролера, що обмежує масштабування та ускладнює керування у великих проєктах;

— jenkins sprawl — явище, коли різні команди створюють численні незалежні інсталяції Jenkins, що призводить до фрагментації та складності адміністрування.

1.5 Моніторинг та візуалізація розгорнутих сервісів

Після розгортання програмного застосунку у промисловому середовищі одним із найважливіших етапів забезпечення його стабільності, надійності та ефективності є моніторинг. Він виконує роль постійного спостереження за станом системи, її продуктивністю, доступністю та поведінкою користувачів, забезпечуючи можливість оперативного виявлення, діагностики та усунення проблем. У сучасній парадигмі DevOps моніторинг є не просто допоміжним

інструментом, а невід'ємною складовою безперервного циклу розробки, розгортання та експлуатації програмного забезпечення, який забезпечує зворотний зв'язок між усіма етапами життєвого циклу продукту.

Основною метою моніторингу є досягнення високої доступності (High Availability), що передбачає мінімізацію часу реагування на інциденти та підтримання безперервної роботи сервісів. Для оцінки ефективності цього процесу використовуються такі показники, як час виявлення проблеми (Time to Detect, TTD), час пом'якшення наслідків (Time to Mitigate, TTM) та час остаточного усунення проблеми (Time to Remediate, TTR). Зменшення цих показників свідчить про зрілість DevOps-команди та її здатність швидко реагувати на непередбачувані події.

Процес моніторингу починається з автоматичного виявлення відхилень у роботі системи за допомогою спеціалізованих інструментів, які збирають аналітичні дані з різних компонентів інфраструктури. У разі виявлення проблеми система надсилає сповіщення відповідальним фахівцям, що дозволяє їм оперативно вжити заходів для відновлення стабільної роботи. Після локалізації інциденту команди проводять аналіз кореневих причин (Root Cause Analysis), розробляють заходи для їх усунення та вдосконалюють механізми запобігання подібним випадкам у майбутньому. Таким чином, ефективний моніторинг є ключем до підвищення стійкості інфраструктури та зменшення простоїв, що напряму впливає на задоволеність користувачів [20].

Другою важливою метою моніторингу є підтримка перевіреного навчання (Validated Learning), що полягає у використанні зібраних даних для ухвалення обґрунтованих рішень під час розвитку продукту. Кожне оновлення або розгортання програмного забезпечення розглядається як експеримент, результати якого мають підтвердити або спростувати гіпотези, висунуті командою розробки. Аналіз показників використання системи, поведінки користувачів і впливу нових функцій на загальну ефективність дозволяє об'єктивно оцінювати результати впроваджених змін. У разі позитивного підтвердження гіпотези команда може продовжити обраний напрямок розвитку, а у випадку невдачі — оперативно

змінити стратегію або повернутися до попереднього стану. Таким чином, моніторинг стає основним інструментом для формування культури прийняття рішень на основі даних (data-driven development), що сприяє постійному вдосконаленню продукту.

Важливою технічною основою моніторингу є телеметрія (Telemetry) — механізм збору, передавання та аналізу даних про стан системи. Телеметрія може реалізовуватись через агенти, встановлені у середовищі виконання, за допомогою SDK, який вбудовується безпосередньо у вихідний код програми, або через централізовані журнали подій (логи). Отримані дані можуть бути класифіковані на два основні потоки: дані реального часу, які використовуються для оперативних сповіщень та побудови дашбордів, і великі обсяги історичних даних, призначених для глибинного аналізу, усунення несправностей та аналітики використання. Ефективне поєднання цих потоків дозволяє досягти балансу між швидкістю реагування та точністю аналітичних висновків.

Для оцінки продуктивності та доступності сервісів застосовуються два підходи: синтетичний моніторинг (Synthetic Monitoring) та моніторинг реальних користувачів (Real User Monitoring, RUM). Синтетичний моніторинг ґрунтується на виконанні передбачуваних тестових транзакцій, які імітують взаємодію користувача із системою. Цей метод дозволяє отримати повторювані результати та порівнювати ефективність роботи сервісу між різними релізами. Натомість RUM фіксує реальні показники з пристроїв кінцевих користувачів — браузерів, мобільних застосунків або робочих станцій. Він враховує умови мережевого середовища, затримки, кешування та маршрутизацію, забезпечуючи більш точне розуміння того, як користувач сприймає роботу системи. Таким чином, поєднання синтетичного моніторингу та RUM дозволяє досягти як відтворюваності вимірювань, так і реалістичності даних.

Моніторинг є також основним інструментом під час тестування у виробничому середовищі (Testing in Production), коли команди DevOps здійснюють спостереження за поведінкою системи безпосередньо після розгортання нових версій. У цьому випадку моніторинг забезпечує потік телеметричних даних у

реальному часі, що дозволяє оперативно виявляти аномалії, аналізувати невідомі раніше проблеми (“unknown unknowns”) та своєчасно вживати заходів для їх усунення. Поєднання систем моніторингу з конвеєром безперервного розгортання (Continuous Deployment Pipeline) створює замкнений цикл спостереження та оптимізації, який підвищує стабільність і надійність програмного забезпечення.

Ефективний моніторинг є фундаментальним чинником успішної реалізації DevOps-підходу. Він забезпечує командам можливість працювати у швидкому темпі, зменшуючи час реакції на інциденти, підвищує якість продукту та рівень задоволеності користувачів. Крім того, моніторинг створює умови для розвитку довгострокових відносин із клієнтами, сприяє зростанню лояльності, утриманню користувачів і покращенню бізнес-результатів. Таким чином, у сучасному контексті DevOps моніторинг виступає не лише технічним інструментом, а стратегічним компонентом процесу розробки, який забезпечує стійкість, надійність і конкурентоспроможність програмних систем.

Prometheus та Grafana утворюють потужний інструментарій для збору, збереження, аналізу та візуалізації телеметричних даних, що дозволяє забезпечувати безперервне спостереження за станом систем, сервісів та інфраструктури. Їхнє поєднання стало стандартом у побудові систем моніторингу для контейнеризованих і хмарних середовищ, оскільки воно дозволяє не лише фіксувати технічні показники, але й оперативно реагувати на відхилення, що впливають на стабільність роботи програмного забезпечення.

Prometheus — це система моніторингу та збору метрик з відкритим вихідним кодом, розроблена спочатку компанією SoundCloud і нині підтримувана спільнотою Cloud Native Computing Foundation (CNCF). Вона є одним із базових компонентів екосистеми Kubernetes і широко використовується для моніторингу мікросервісних архітектур. Основна концепція Prometheus полягає у моделі збору даних на основі pull-підходу, за якої Prometheus регулярно опитує визначені цілі (targets) через HTTP-інтерфейс і зберігає отримані метрики у власній time-series базі даних. Така архітектура забезпечує незалежність від агентів та високу гнучкість у налаштуванні системи моніторингу.

Prometheus характеризується наявністю високопродуктивного механізму збереження часових рядів (time series database), який дозволяє ефективно опрацьовувати великі обсяги метрик у реальному часі. Кожна метрика ідентифікується унікальною комбінацією її назви та набору ключів (labels), що дозволяє створювати багатовимірну модель даних. Завдяки цьому Prometheus надає можливість виконувати складні аналітичні запити до даних, використовуючи власну мову запитів PromQL (Prometheus Query Language). PromQL підтримує агрегацію, фільтрацію та обчислення на основі часових рядів, що робить її ефективним засобом для аналізу стану інфраструктури, виявлення тенденцій і прогнозування можливих проблем.

Збір даних у Prometheus здійснюється через експортери (exporters) — спеціальні сервіси, які надають дані у форматі, сумісному з Prometheus. Наприклад, Node Exporter використовується для збору системних метрик із серверів, таких як використання процесора, пам’яті чи файлової системи, тоді як cAdvisor або kube-state-metrics надають інформацію про контейнери та об’єкти Kubernetes. Цей підхід забезпечує високу гнучкість, оскільки дає змогу інтегрувати Prometheus із широким спектром програмного забезпечення, сервісів та інструментів DevOps.

Окрім збору метрик, Prometheus має вбудовану систему оповіщень (Alertmanager), яка відповідає за формування, обробку та маршрутизацію сповіщень про критичні події. Alertmanager дозволяє визначати складні правила сповіщення, групувати подібні події, уникати дублювання повідомлень і надсилати їх через різні канали — електронну пошту, Slack, Telegram або інші системи сповіщення. Таким чином, Prometheus забезпечує не лише моніторинг стану системи, але й активну підтримку процесу реагування на інциденти, що є ключовим аспектом DevOps-культури.

Grafana доповнює Prometheus, забезпечуючи візуалізацію зібраних метрик у вигляді інформативних та інтерактивних дашбордів. Вона виступає фронтенд-компонентом системи моніторингу, яка дозволяє DevOps-командам швидко оцінювати стан інфраструктури, виявляти аномалії та аналізувати історичні дані. Grafana підтримує підключення до численних джерел даних, зокрема Prometheus,

InfluxDB, Loki, Elasticsearch, OpenTSDB та ін., що робить її універсальним інструментом для централізованого спостереження. Візуалізація метрик у Grafana здійснюється за допомогою різних типів віджетів — графіків, таблиць, гістограм, панелей і карт, що дозволяє створювати гнучкі дашборди, адаптовані під конкретні завдання.

Завдяки своїй архітектурі Grafana дозволяє створювати інтерактивні аналітичні панелі, які відображають стан системи в реальному часі та забезпечують можливість фільтрації, порівняння та деталізації даних. У поєднанні з Prometheus вона надає можливість відображати ключові показники ефективності (Key Performance Indicators, KPI), такі як затримки запитів, використання ресурсів, навантаження на сервери, кількість помилок тощо. Крім того, Grafana має механізм Alerting, який дозволяє визначати порогові значення метрик і створювати правила сповіщень, що дублюють або розширюють можливості Prometheus Alertmanager.

Prometheus і Grafana разом формують повноцінну екосистему моніторингу, що відповідає принципам DevOps — безперервному зворотному зв’язку, автоматизації та прозорості. У контексті розподілених систем і мікросервісної архітектури вони дозволяють досягати високого рівня спостережуваності (observability), забезпечуючи не лише моніторинг метрик, але й глибоке розуміння взаємозв’язків між компонентами інфраструктури. Завдяки цьому команди можуть швидко ідентифікувати проблеми, локалізувати їхні причини та впроваджувати коригувальні дії, зменшуючи час простою та підвищуючи надійність системи [22].

Grafana дозволяє вибирати різні вигляді для панелів у дашборді, що дає можливість експерементувати та обирати зручний спосіб подання даних, які виводить користувач.

Наприклад, якщо користувач хоче подати часові дані, тобто такі, які показують певне значення з фіксованим часовим значенням, то такі дані буде краще всього виводити у панель за допомогою графіків, які дозволять відтворити часову картину за рахунок графічного подання та краще аналізувати отримані дані.

Представлення Grafana-дашборду з даними Prometheus показано на рисунку 1.2.

Рисунок 1.2 — Grafana-дашборд з даними Prometheus

У даному розділі було проведено розширений аналіз сучасного стану технологій розгортання, моніторингу та інтеграції DevOps-практик у життєвий цикл розробки програмного забезпечення. Розглянуто концептуальні засади DevOps як сукупності організаційних підходів, інженерних методів і програмно-технічних засобів, спрямованих на тісну інтеграцію процесів розробки, тестування, розгортання та експлуатації програмних систем. Підкреслено, що впровадження DevOps дозволяє суттєво скоротити час виходу програмного продукту на ринок, знизити кількість помилок під час випуску нових версій, підвищити стабільність і надійність сервісів, а також забезпечити безперервну інтеграцію та доставку (CI/CD) програмних рішень.

Окрему увагу приділено ролі автоматизації як ключового чинника ефективності DevOps-процесів. Зокрема, проаналізовано технологію Infrastructure as Code (IaC), яка передбачає опис інфраструктури у вигляді формалізованого коду та її керування за допомогою спеціалізованих інструментів. Такий підхід дозволяє мінімізувати людський фактор, забезпечити відтворюваність середовищ, спростити масштабування та полегшити супровід інформаційних систем упродовж усього життєвого циклу. Визначено основні підходи до реалізації IaC — декларативний, що орієнтований на опис бажаного стану інфраструктури, та імперативний, який базується на послідовному виконанні команд для її створення й модифікації.

Детально розглянуто принципи керування конфігураціями (Configuration Management), яке спрямоване на підтримання стабільності, узгодженості та безпеки IT-систем упродовж усього життєвого циклу. Проаналізовано переваги впровадження систем керування конфігураціями, зокрема підвищення відмовостійкості, спрощення аудиту, скорочення часу на оновлення та зниження ймовірності помилок. Розглянуто основні концепції контейнеризації, принципи роботи систем оркестрації контейнерів, зокрема Kubernetes, а також ключові підходи до побудови безперервних процесів інтеграції та доставлення (CI/CD).

Розглянуто особливості використання Prometheus і Grafana як основних інструментів збору, обробки та візуалізації телеметричних даних, які утворюють комплексну екосистему для побудови систем моніторингу. Визначено їх переваги у контексті реалізації DevOps-підходу, зокрема забезпечення спостережуваності інфраструктури, раннього виявлення відхилень у роботі сервісів та підвищення ефективності процесів реагування на інциденти.

2 ОБҐРУНТУВАННЯ ВИБОРУ ТЕХНОЛОГІЙ ДЛЯ РОЗГОРТАННЯ ТА МОНІТОРИНГУ ПРИКЛАДНИХ СЕРВІСІВ

2.1 Вибір інструменту для побудови інфраструктури за допомогою коду

Сучасна практика побудови інфраструктури в межах DevOps-підходу базується на концепції Infrastructure as Code (IaC), що передбачає опис і керування інфраструктурними ресурсами за допомогою програмного коду, а не ручних дій. Такий підхід забезпечує відтворюваність середовищ, прозорість змін і можливість автоматизації розгортання систем будь-якої складності. Серед великої кількості інструментів IaC — таких як Terraform, AWS CloudFormation, Pulumi, Chef, SaltStack та інші — було обрано саме Terraform від компанії HashiCorp як базовий засіб для побудови інфраструктури системи розгортання прикладних сервісів.

Основною перевагою Terraform є його провайдерна архітектура та універсальність. На відміну від таких інструментів, як AWS CloudFormation, який орієнтований лише на екосистему Amazon Web Services, Terraform підтримує десятки провайдерів, серед яких не лише хмарні сервіси (AWS, Azure, Google Cloud, Linode, DigitalOcean), але й локальні рішення, мережеве обладнання, DNS-системи та контейнери. Завдяки цьому Terraform дозволяє описувати повну інфраструктуру — від мережевої топології та віртуальних машин до балансувальників навантаження та DNS-записів — у єдиній декларативній моделі.

Terraform використовує декларативну мову опису інфраструктури (HCL — HashiCorp Configuration Language), яка забезпечує високу читабельність і зрозумілість коду навіть для нефахівців із програмування. Це важлива перевага порівняно з інструментами на основі скриптів, такими як Pulumi, що використовує загальні мови програмування (Python, Go, TypeScript) і тим самим ускладнює підтримку для команд, які не мають досвіду розробки програмного коду.

HCL дозволяє описати бажаний стан інфраструктури, а Terraform самостійно визначає необхідні дії для приведення поточного стану до бажаного.

Загальну схему роботи IaC інструменту Terraform зображено на рисунку 2.1.

Рисунок 2.1 — Загальна схема роботи Terraform

Такий підхід реалізує модель ідемпотентності, за якої повторне виконання коду не призводить до небажаних змін у вже створеній інфраструктурі. Важливою особливістю Terraform є використання механізму “execution plan” — попереднього плану змін, який показує, які саме дії будуть виконані (створення, зміна чи видалення ресурсів) до моменту їх застосування. Це забезпечує високий рівень контрольованості змін та знижує ризик помилкових або небезпечних операцій у виробничих середовищах. Інструменти типу Ansible або Chef, які також можуть використовуватись для управління інфраструктурою, не мають такого рівня прозорості при попередньому плануванні змін.

Крім того, Terraform має вбудований стан інфраструктури (state file), який зберігає актуальну інформацію про всі створені ресурси. Це дозволяє синхронізувати описану модель із реальною інфраструктурою та уникати розсинхронізації, що є критично важливим для складних систем із багатьма залежностями. Завдяки можливості зберігати state у віддалених сховищах (наприклад, S3 або Git-репозиторіях), Terraform забезпечує командну роботу та контроль версій інфраструктури — аналогічно до системи керування кодом.

Порівняно з іншими IaC-інструментами, Terraform також має більш зрілу екосистему та активну спільноту. Велика кількість модулів із відкритим кодом

дозволяє швидко розгортати типові інфраструктурні сценарії без потреби створювати все з нуля. Це особливо корисно при побудові системи розгортання прикладних сервісів, де важлива швидкість, відтворюваність і стандартизація конфігурацій.

2.2 Інструменти для контролю конфігураціями

DevOps-практики передбачають не лише автоматизоване створення інфраструктури, а й забезпечення її подальшої конфігурації, оновлення та підтримки в узгодженому стані. Ці завдання реалізуються за допомогою підходу Configuration Management, який забезпечує централізоване та відтворюване керування програмним середовищем на великій кількості серверів. Серед найпоширеніших інструментів конфігураційного управління — Ansible, Puppet, Chef та SaltStack — найбільш оптимальним рішенням для побудови системи розгортання прикладних сервісів було обрано саме Ansible, розроблений компанією Red Hat.

Основними принципами та цілями, закладеними у дизайн Ansible, є:

— мінімалізм система керування не повинна створювати додаткових залежностей або ускладнювати середовище, в якому вона працює;

— послідовність Ansible дозволяє створювати та підтримувати стабільні, передбачувані середовища;

— безпека Ansible не використовує агентів, що знижує ризики вразливостей, а для роботи потрібні лише OpenSSH та Python на керованих вузлах;

— надійність при правильному написанні playbook (сценаріїв Ansible) вони є ідемпотентними;

— простота у навчанні Ansible використовує зрозумілу декларативну мову опису на основі YAML і шаблонів Jinja.

Основна перевага Ansible полягає у його безагентій архітектурі. На відміну від Puppet чи Chef, які вимагають встановлення спеціального агента на кожному вузлі, Ansible використовує стандартний SSH-протокол для взаємодії з керованими серверами. Це суттєво спрощує процес розгортання системи, зменшує кількість

залежностей і підвищує безпеку, оскільки виключає необхідність відкривати додаткові порти або підтримувати постійне з’єднання між вузлами. Такий підхід дозволяє інтегрувати Ansible у будь-яке середовище — як локальне, так і хмарне — без потреби змінювати його архітектуру.

Формула ефективності автоматизації розгортання:

$$E_t = \frac{T_{мануал} - T_{авто}}{T_{мануал}} * 100\%, \quad (2.1)$$

де $T_{мануал}$ — час ручного розгортання;

$T_{авто}$ — час автоматизованого розгортання.

Ще однією вагомою перевагою є декларативний стиль опису конфігурацій, який реалізується через Playbooks — YAML-файли, що описують бажаний стан системи. Цей підхід набагато зручніший порівняно з процедурним програмуванням у Chef або Puppet, оскільки дозволяє чітко визначити, що саме потрібно отримати в результаті, без необхідності детально описувати як саме це слід зробити. Завдяки цьому код стає зрозумілим, легким для підтримки та спільної роботи. Вузли, з якими працює Ansible можна розділяти на окремі групи в файлах, які називаються Inventory. Inventory-файл дозволяє вказати вузли, з якими буде відбуватись робота, задати окремі правила, такі як паролі до вузлів, ключі, імена юзерів та розділити вузли на окремі групи, що дозволяє працювати з декількома провайдерами або особливими вузлами заздалегідь вказавші потрібні налаштування для групи. Крім того, Ansible забезпечує ідемпотентність операцій — повторне виконання одного і того ж плейбуку не спричиняє змін, якщо система вже перебуває в необхідному стані. Це особливо важливо для стабільності виробничих середовищ.

З технічного погляду, Ansible має просту та гнучку архітектуру, що складається з модулів, інвентарних файлів та ролей. Модулі реалізують конкретні дії, такі як встановлення пакетів, редагування файлів чи керування сервісами. Інвентарні файли визначають список хостів, якими необхідно керувати, а ролі дозволяють структурувати конфігураційний код і повторно використовувати його для різних середовищ.

Такий підхід відповідає принципам модульності та повторного використання, що є фундаментальними для побудови масштабованих DevOps-систем.

Загальну схему безагентної архітектури інструменту керування конфігураціями Ansible показано на рисунку 2.2.

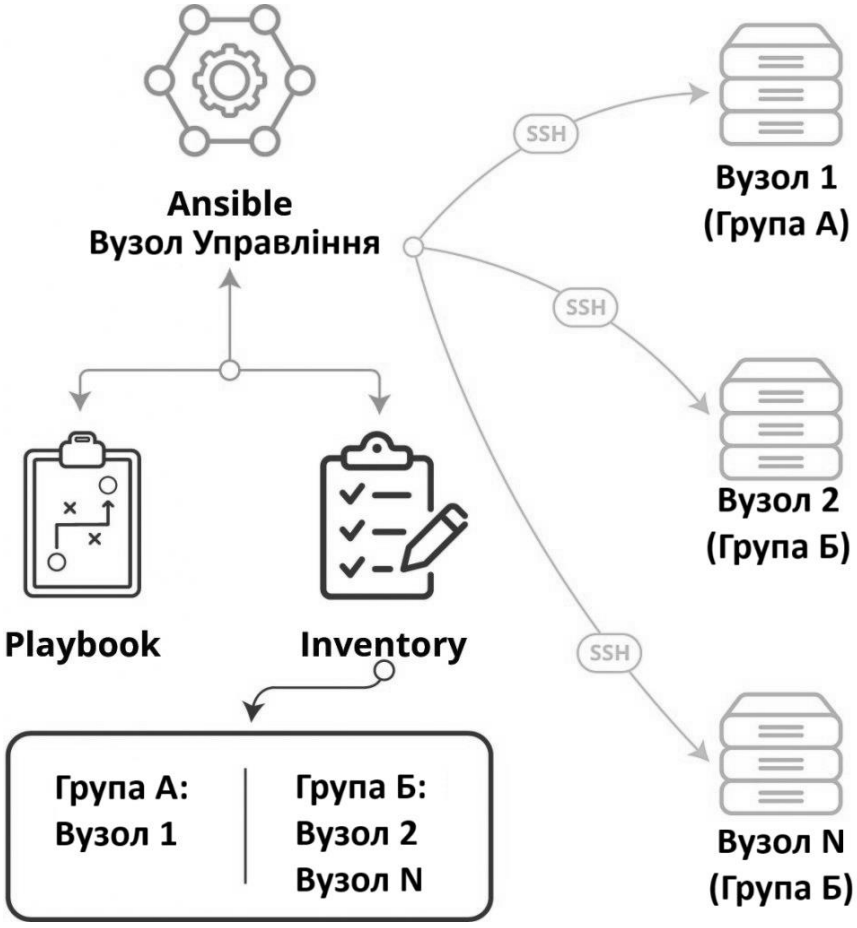

Рисунок 2.2 — Загальна схема безагентної архітектури Ansible

Порівняно з іншими системами конфігураційного управління, Ansible вирізняється низьким порогом входу. Його синтаксис на основі YAML є інтуїтивно зрозумілим, а навчання не потребує глибоких знань програмування. Puppet та Chef, навпаки, використовують спеціальні доменні мови (DSL) або Ruby, що значно ускладнює їх впровадження в командах без досвіду розробки. Простота Ansible також полегшує інтеграцію з іншими інструментами DevOps, зокрема Terraform, Jenkins та Kubernetes, що дозволяє реалізувати повний цикл автоматизованого

розгортання — від створення інфраструктури до налаштування середовища виконання.

Важливою технічною перевагою Ansible є розширюваність та інтеграційна сумісність. Система підтримує підключення через плагіни до хмарних провайдерів (AWS, Azure, GCP, Linode), контейнерних середовищ (Docker, Kubernetes) та платформ CI/CD. Наприклад, після створення інфраструктури за допомогою Terraform, Ansible може автоматично підключатися до новостворених вузлів і виконувати конфігураційні завдання — встановлення пакетів, налаштування файрволу, запуск сервісів і розгортання застосунків. Завдяки цьому досягається повна автоматизація життєвого циклу інфраструктури, що є ключовим принципом DevOps.

Ще однією перевагою є висока масштабованість та стабільність роботи Ansible у великих середовищах. Використання концепції інвентаризації дозволяє централізовано керувати сотнями вузлів без суттєвих витрат на підтримку. Для великих корпоративних систем передбачено можливість застосування Ansible Tower (або AWX) — веб-інтерфейсу з розширеними функціями моніторингу, контролю доступу та журналювання. Це дозволяє застосовувати Ansible не лише як інструмент автоматизації, а як повноцінну платформу централізованого управління конфігураціями.

З точки зору продуктивності, Ansible демонструє оптимальний баланс між швидкістю виконання та стабільністю. Хоча інструменти на основі агентів, такі як SaltStack, можуть забезпечувати швидше виконання команд завдяки постійному підключенню, відсутність агентів в Ansible робить його більш надійним, безпечним і простим у підтримці. Для більшості сценаріїв DevOps, де головним є не миттєва реакція, а стабільність і передбачуваність змін, саме цей підхід є найдоцільнішим.

2.3 Аналіз інструментів оркестрації та контейнеризації

Контейнеризація та оркестрація є ключовими складовими побудови гнучких, масштабованих і відмовостійких систем розгортання прикладних сервісів. Вони забезпечують ефективне використання ресурсів, спрощують процеси деплою,

оновлення та відновлення сервісів, а також сприяють підвищенню стабільності інфраструктури. Серед існуючих технологій контейнеризації та оркестрації — таких як Docker, Podman, containerd, OpenShift, Docker Swarm і Kubernetes — найраціональнішим вибором для побудови системи DevOps є зв'язка Docker + Kubernetes, яка забезпечує повний цикл керування контейнеризованими застосунками — від створення контейнерів до їх масштабованого розгортання та моніторингу.

Docker став стандартом де-факто у сфері контейнеризації завдяки своїй стабільності, зрілості та широкій підтримці спільнотою. Його основна перевага полягає в ізоляції програмного забезпечення разом із усіма необхідними залежностями в рамках єдиного середовища виконання — контейнера. Це дозволяє створювати однакові умови роботи застосунку незалежно від операційної системи чи апаратної архітектури. Порівняно з традиційною віртуалізацією, де кожен віртуальний сервер потребує власного ядра операційної системи, контейнери є значно легшими, швидшими в запуску та ефективнішими з точки зору споживання ресурсів.

Архітектура Docker базується на двох ключових компонентах: Docker Engine — середовищі виконання, яке забезпечує створення, запуск і управління контейнерами, та Docker Images — стандартизованих шаблонах, на основі яких формуються ізольовані середовища виконання. Docker Engine складається з серверної частини (демона), REST API та клієнтського інтерфейсу, що дозволяє автоматизувати керування контейнерами й інтегрувати Docker у різноманітні інструменти безперервної інтеграції та доставки. Кожен контейнер є ізольованим від інших і функціонує поверх спільного ядра операційної системи хоста, що забезпечує як високий рівень безпеки, так і ефективне використання обчислювальних ресурсів у порівнянні з традиційними віртуальними машинами.

Використання образів Docker забезпечує відтворюваність середовища виконання незалежно від апаратної платформи чи операційної системи, на якій здійснюється розгортання. Образи є багатошаровими, що дозволяє повторно використовувати окремі шари та оптимізувати процес зберігання і передачі даних.

Центральну роль у формуванні образу відіграє файл конфігурації Dockerfile, у якому декларативно описується послідовність інструкцій для створення контейнерного середовища, включаючи встановлення залежностей, копіювання вихідного коду та налаштування параметрів запуску застосунку.

Загальна схема роботи Docker контейнерів зображено на рисунку 2.3.

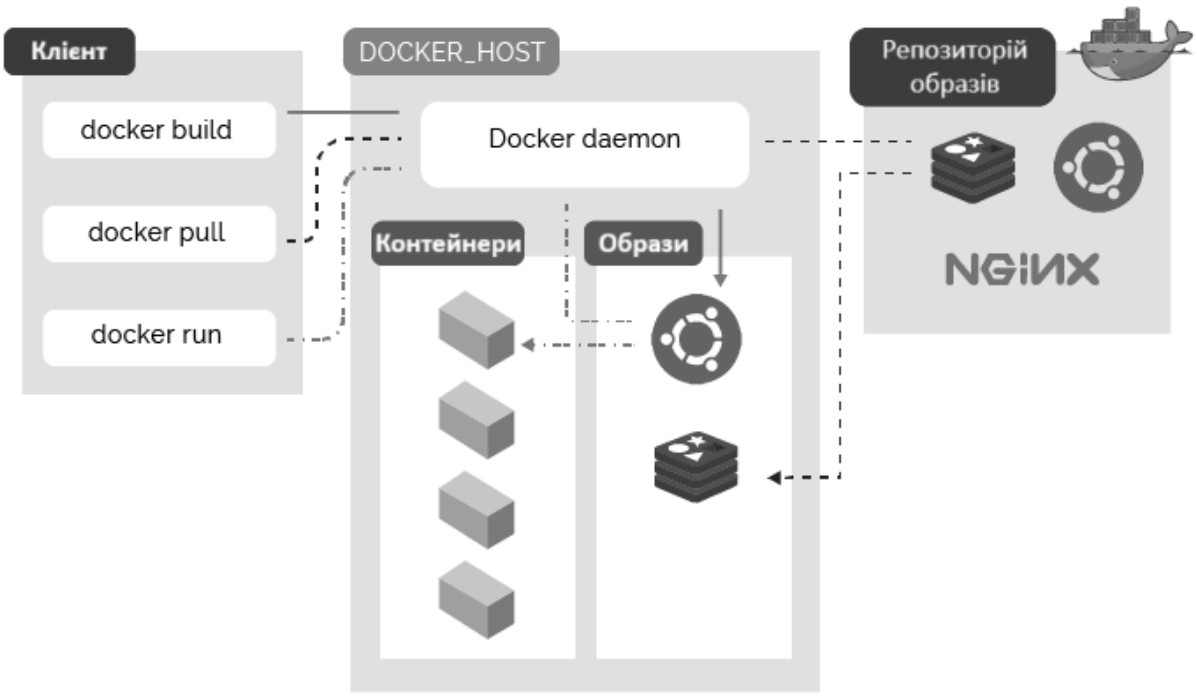

Рисунок 2.3 — Загальна схема роботи Docker контейнерів

Docker надає можливість створення та поширення образів через Docker Hub або приватні реєстри, що забезпечує централізоване управління артефактами розгортання. Такий підхід значно спрощує підтримку різних версій сервісів і сприяє повторюваності процесів у CI/CD-конвеєрі. Порівняно з альтернативами, такими як Podman або LXC, Docker має більш зрілу екосистему, широку інтеграцію з інструментами автоматизації (Ansible, Jenkins, Terraform) та більшу кількість готових рішень для хмарних платформ. Саме тому Docker залишається найбільш оптимальним вибором для контейнеризації прикладних сервісів у сучасних DevOps-архітектурах.

Однак із зростанням масштабів інфраструктури та кількості контейнерів виникає потреба у системі, яка могла б автоматично керувати розгортанням,

балансуванням навантаження, відновленням після збоїв і масштабуванням контейнерів. Ці завдання вирішує Kubernetes. Kubernetes дозволяє керувати великою кількістю контейнеризованих сервісів у кластері, забезпечуючи їх автоматичне розподілення між вузлами, відновлення після збоїв і безперервне оновлення без простоїв.

Kubernetes базується на декларативному підході до управління інфраструктурою, за якого бажаний стан системи описується у вигляді YAML-маніфестів, а платформа автоматично забезпечує його досягнення та підтримання незалежно від поточних або проміжних змін у середовищі виконання. Такий підхід дозволяє абстрагувати процес управління від конкретних операцій адміністрування та зосередитися на описі цільової конфігурації, що істотно підвищує відтворюваність і керованість розгортань. Kubernetes постійно порівнює фактичний стан кластера з описаним у маніфестах і, у разі відхилень, виконує необхідні дії для його автоматичного відновлення.

Основними об’єктами Kubernetes є Pod, Node, Deployment, ReplicaSet, Service та Ingress, кожен з яких виконує чітко визначену функцію в межах загальної архітектури кластера. Pod є базовою одиницею розгортання та інкапсулює один або кілька контейнерів, які спільно використовують мережевий простір і файлову систему, що забезпечує ефективну взаємодію між компонентами застосунку. Node представляє собою фізичний або віртуальний сервер, на якому запускаються Pod’и та на якому працюють ключові компоненти Kubernetes, зокрема kubelet і контейнерне середовище виконання.

Для управління життєвим циклом прикладних сервісів використовується об’єкт Deployment, який забезпечує автоматизоване розгортання, масштабування та оновлення застосунків без переривання їхньої доступності. Deployment керує об’єктами ReplicaSet, що відповідають за підтримку заданої кількості реплік Pod’ів і тим самим підвищують надійність та відмовостійкість системи. Мережеву взаємодію всередині кластера та доступ користувачів до сервісів забезпечують об’єкти Service та Ingress. Service абстрагує набір Pod’ів і надає стабільну точку доступу до них, тоді як Ingress дозволяє реалізувати маршрутизацію зовнішнього

HTTP(S)-трафіку, що є особливо важливим для розгортання веборієнтованих прикладних сервісів.

Завдяки такій моделі Kubernetes виступає універсальною платформою для керування контейнеризованими застосунками, забезпечуючи масштабованість, автоматичне відновлення та високий рівень доступності сервісів у межах сучасних DevOps-орієнтованих інфраструктур.

Загальну схему архітектури оркестратора Kubernetes зображено на рисунку 2.4.

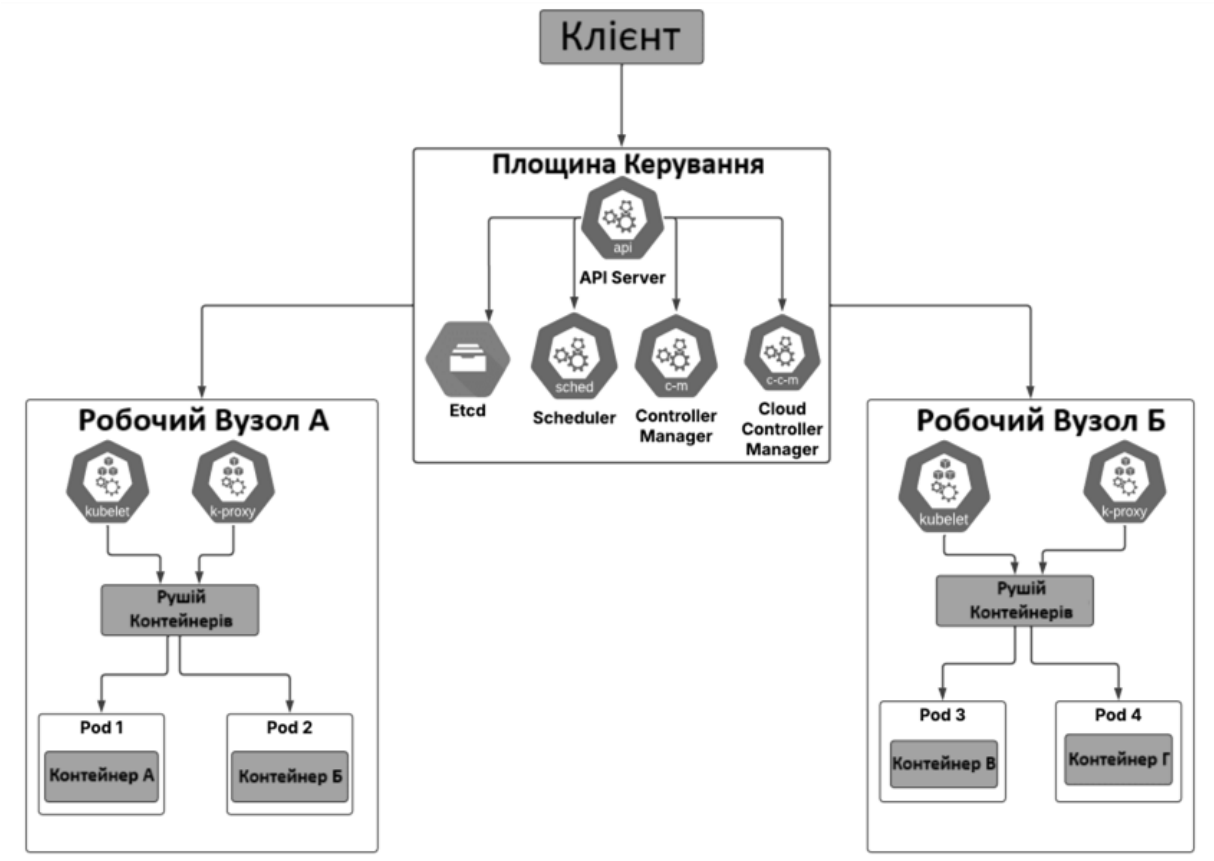

Рисунок 2.4 — Загальна схема архітектури оркестратора Kubernetes

Kubernetes надає потужні механізми для балансування навантаження, автоматичного відновлення (self-healing) та горизонтального масштабування (Horizontal Pod Autoscaling). У разі збою вузла або контейнера система автоматично перезапускає необхідні компоненти, забезпечуючи безперервну доступність сервісів. Це робить Kubernetes критично важливим елементом інфраструктури високої надійності. На відміну від простіших рішень, таких як Docker Swarm,

Kubernetes має більш розвинену архітектуру та гнучкі можливості управління, що дозволяє використовувати його як у локальних, так і в хмарних середовищах.

Інтеграція Docker із Kubernetes забезпечує повний цикл контейнерного управління. Docker використовується для створення й пакування контейнерів, а Kubernetes — для їх розгортання, масштабування та моніторингу. Завдяки стандартам Open Container Initiative (OCI) обидва інструменти є сумісними на рівні форматів контейнерів, що гарантує стабільність та переносимість рішень між різними середовищами. Така архітектура ідеально підходить для DevOps-систем, оскільки дозволяє автоматизувати весь життєвий цикл застосунку: від його побудови в CI/CD-конвеєрі (наприклад, через Jenkins) до розгортання й моніторингу в кластері Kubernetes із використанням Prometheus та Grafana.

Додатковою перевагою Kubernetes є широка екосистема допоміжних інструментів, серед яких виділяються Helm — менеджер пакетів для Kubernetes, який спрощує управління складними конфігураціями, та kubectl — утиліта командного рядка для адміністрування кластерів. Це робить Kubernetes не просто оркестратором, а платформою для побудови масштабованих хмарних рішень, що забезпечує ефективну інтеграцію з CI/CD-процесами, моніторингом і системами логування.

2.4 Розгляд рішень для автоматизації доставки коду

Процеси безперервної інтеграції (Continuous Integration, CI) та безперервного доставлення або розгортання (Continuous Delivery / Continuous Deployment, CD) використовуються для досягнення високої швидкості та надійності випуску програмного забезпечення. Вони забезпечують автоматизацію усіх етапів життєвого циклу застосунку — від коміту коду до його потрапляння у production-середовище. Для реалізації такого підходу на практиці використовується низка інструментів, серед яких найпоширенішими є Jenkins, GitLab CI/CD, GitHub Actions, CircleCI, Travis CI, TeamCity тощо. Проте саме Jenkins було обрано як базовий інструмент для реалізації CI/CD у системі розгортання прикладних

сервісів, оскільки він поєднує гнучкість, розширюваність, зрілість екосистеми та можливість інтеграції з будь-якими компонентами DevOps-інфраструктури.

Jenkins — це відкрите та безкоштовне рішення для автоматизації процесів розробки програмного забезпечення, яке було створене на основі Java і має одну з найпотужніших екосистем плагінів серед CI/CD-платформ. Його основна концепція полягає у побудові конвеєрів (pipelines), що автоматизують послідовність дій — від збірки та тестування до розгортання та моніторингу. Jenkins може взаємодіяти з репозиторіями коду (GitHub, GitLab, Bitbucket), системами керування конфігураціями (Ansible, Chef, Puppet), контейнерними середовищами (Docker, Kubernetes), а також із системами моніторингу, як-от Prometheus чи Grafana.

Перевагою Jenkins є його архітектурна гнучкість. Він підтримує як класичний підхід до автоматизації через окремі завдання (jobs), так і декларативний підхід через Jenkinsfile, у якому процес CI/CD описується у вигляді коду (Pipeline as Code). Такий підхід є логічним продовженням принципів Infrastructure as Code (IaC) і забезпечує повторюваність, контроль версій та зручність інтеграції з системами керування кодом. Це дозволяє легко документувати процеси CI/CD, відслідковувати зміни в конвеєрі та розгортати їх на різних серверах чи у хмарних середовищах.

Jenkins має модульну архітектуру, що складається з Controller (раніше Master) та Agent вузлів. Контролер відповідає за керування конвеєрами, координацію завдань, зберігання конфігурацій та розподіл навантаження між агентами. Агенти, у свою чергу, виконують безпосередньо завдання збірки, тестування чи деплою. Такий підхід дозволяє масштабувати систему CI/CD горизонтально, що є надзвичайно важливим при роботі з великими обсягами коду або множинними проєктами. Наприклад, у запропонованій системі Jenkins Controller може бути розміщений на окремому сервері, тоді як агенти виконуватимуть окремі етапи — побудову Docker-образів, розгортання на кластері Kubernetes чи перевірку коректності конфігурацій Ansible. На відміну від багатьох конкурентних рішень, Jenkins вирізняється високим рівнем кастомізації.

Схема архітектури CI/CD інструменту Jenkins зображена на рисунку 2.5.

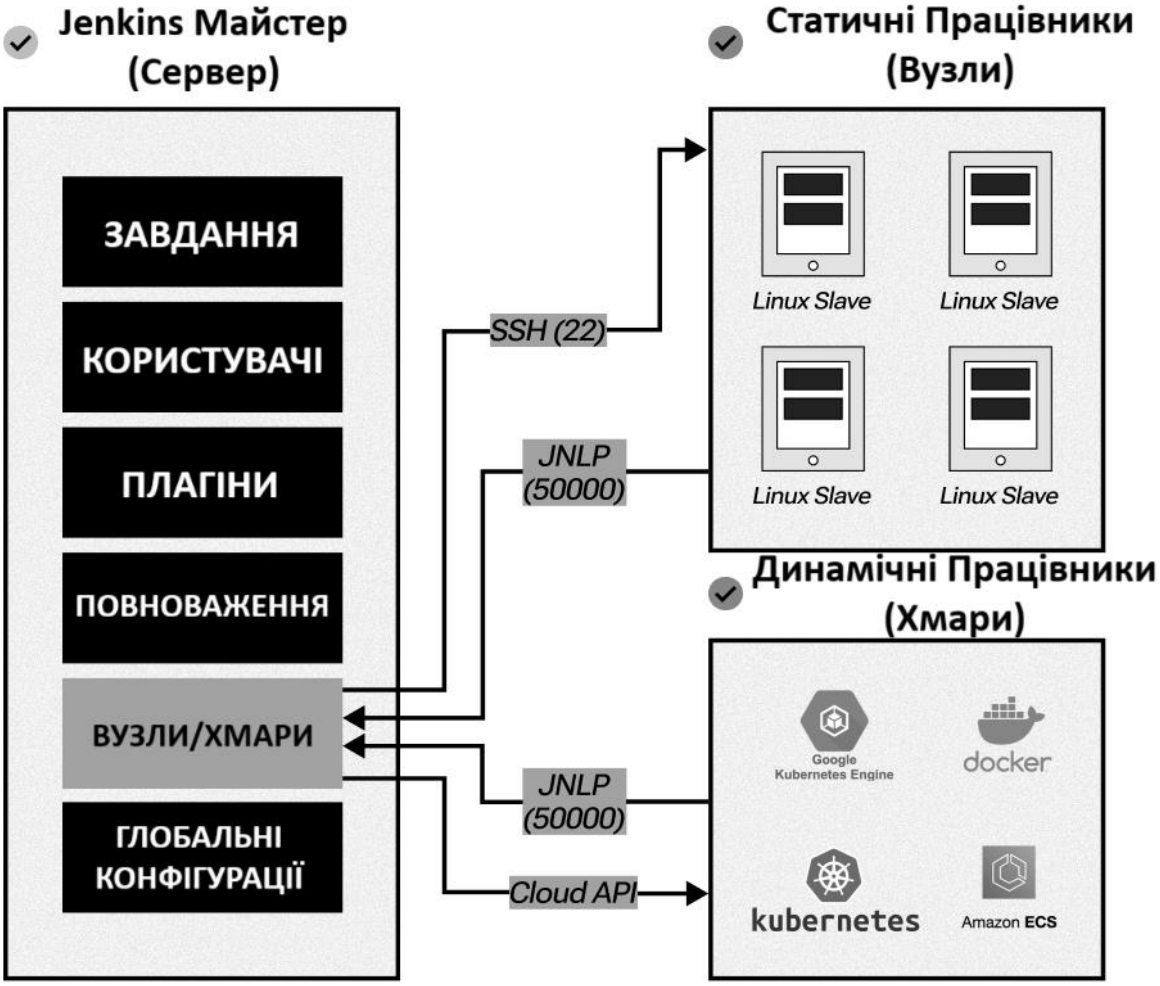

Рисунок 2.5 — Схема архітектури CI/CD інструменту Jenkins

Його екосистема налічує понад дві тисячі плагінів, які розширюють функціональність платформи практично в будь-якому напрямку — від інтеграції з SCM-системами до взаємодії з хмарними провайдерами. Наприклад, плагіни Kubernetes Plugin або Docker Pipeline дозволяють динамічно створювати агентів у контейнерах, що забезпечує еластичність і скорочує час на розгортання середовищ для тестування. Це робить Jenkins не просто інструментом CI/CD, а потужною платформою автоматизації, яку можна адаптувати під будь-який сценарій.

Важливим фактором вибору Jenkins є його сумісність з різними середовищами — локальними, хмарними або гібридними. Його можна розгорнути як на фізичних серверах, так і у контейнерах або в Kubernetes-кластері, що

забезпечує узгодженість з іншими компонентами інфраструктури, описаними у попередньому розділі (Terraform, Ansible, Docker, Kubernetes). Таким чином, Jenkins може стати центральним елементом автоматизації, який координує усі етапи DevOps-конвеєра: створення інфраструктури, розгортання середовищ, тестування, публікацію артефактів та оновлення застосунків у production.

Jenkins може працювати у двох режимах:

— push-модель, коли репозиторій або зовнішній інструмент надсилає повідомлення про зміну;

— pull-модель, коли Jenkins періодично перевіряє репозиторій на наявність нових комітів.

Після успішного проходження збірки Jenkins може:

— розгорнути застосунок на тестовому сервері;

— запустити автоматизовані тести;

— надіслати повідомлення розробникам у разі помилки;

— автоматично виконати розгортання у виробничому середовищі, якщо всі перевірки пройдено успішно.

Порівняно з альтернативами, такими як GitLab CI/CD або GitHub Actions, Jenkins має кілька стратегічних переваг. GitLab CI/CD є тісно інтегрованим із власним репозиторієм і підходить для екосистеми GitLab, проте обмежений у можливостях налаштування зовнішніх інтеграцій. GitHub Actions є зручним інструментом для хмарної автоматизації, однак має меншу гнучкість у побудові складних конвеєрів та розгортанні у приватних інфраструктурах. Натомість Jenkins може функціонувати повністю автономно, без прив’язки до конкретного хостингу або постачальника, що забезпечує повний контроль над середовищем CI/CD і підвищує рівень безпеки, оскільки конфіденційні дані не передаються у сторонні сервіси. Ще однією перевагою Jenkins є стабільність та зрілість платформи. Інструмент активно розвивається з 2011 року, має потужну спільноту розробників та докладну технічну документацію. Завдяки цьому Jenkins широко використовується у корпоративному секторі, а його архітектура перевірена в масштабних виробничих середовищах. Саме тому Jenkins залишається еталонним

інструментом для побудови CI/CD-конвеєрів, на основі якого розробляються власні рішення в таких компаніях, як IBM, Red Hat, Netflix та Amazon.

В більшості сучасних систем моніторинг є не просто допоміжним процесом, а ключовим елементом життєвого циклу розробки та експлуатації програмного забезпечення. Безперервне спостереження за станом інфраструктури, сервісів і застосунків дозволяють оперативно виявляти, аналізувати й усувати проблеми, а також прогнозувати потенційні збої. Для досягнення цих цілей необхідно використовувати системи, здатні працювати в умовах динамічно змінюваних середовищ, таких як контейнеризовані застосунки чи Kubernetes-кластери. Серед великої кількості існуючих рішень — Nagios, Zabbix, Datadog, New Relic, Elastic Stack — найбільш ефективним та гнучким рішенням для моніторингу сучасних DevOps-орієнтованих систем є поєднання Prometheus і Grafana.

Prometheus — це відкрита система моніторингу та збору метрик, розроблена спеціально для хмарних і контейнеризованих середовищ. Її основна перевага полягає у моделі збору даних Pull-based, коли Prometheus самостійно опитує цільові ендпоінти (exporters), що забезпечує незалежність від зовнішніх агентів і спрощує конфігурацію. Такий підхід особливо ефективний у Kubernetes-середовищах, де кількість сервісів може динамічно змінюватися — Prometheus автоматично виявляє нові цілі (targets) завдяки механізму service discovery. Це зменшує потребу у ручному налаштуванні та підвищує стабільність системи моніторингу навіть при масштабуванні.

Архітектура Prometheus побудована з урахуванням принципів надійності, розподіленості та автономності. Система може зберігати часові ряди даних локально без потреби у зовнішній базі даних, що мінімізує залежності та підвищує швидкість обробки запитів. Для побудови запитів використовується власна мова — PromQL (Prometheus Query Language), яка дозволяє гнучко агрегувати, порівнювати та обчислювати метрики у реальному часі. Цей інструмент забезпечує високий рівень деталізації при аналізі системних ресурсів — завантаження CPU, використання пам’яті, дискових операцій, мережевої активності тощо.

Ще однією перевагою Prometheus є його модульність і розширюваність. Для збору метрик з різних компонентів системи використовуються так звані exporters — легкі процеси, які перетворюють дані у формат, що розуміє Prometheus. Наприклад, Node Exporter застосовується для збору метрик із серверів, Blackbox Exporter — для перевірки доступності сервісів, cAdvisor — для моніторингу контейнерів, а Kube-State-Metrics — для збору інформації про стан об’єктів Kubernetes. Така гнучкість дозволяє централізовано збирати дані з різнорідних джерел і створювати єдину систему моніторингу для всієї інфраструктури.

У поєднанні з Prometheus найчастіше використовується Grafana — потужна платформа для візуалізації метрик, створення дашбордів та аналітики. Grafana є універсальним інструментом, який може підключатися до десятків джерел даних (Prometheus, InfluxDB, Loki, Elasticsearch, PostgreSQL тощо) і надавати зручний інтерфейс для побудови інтерактивних графіків, таблиць і панелей моніторингу. Саме Grafana забезпечує візуальний рівень сприйняття даних, який дозволяє швидко оцінити стан системи, виявити аномалії чи тренди, а також отримувати автоматичні сповіщення при досягненні критичних значень метрик.

Важливим аспектом є інтеграція між Prometheus та Grafana, яка реалізується природним чином — Grafana має вбудований драйвер для Prometheus, що дозволяє напряму виконувати PromQL-запити та відображати результати у графічному вигляді. Це забезпечує оперативний моніторинг усіх компонентів системи, від фізичних серверів до контейнерів і мікросервісів, без потреби у складних проміжних прошарках. Крім того, Grafana підтримує систему алертингу, яка дозволяє надсилати сповіщення у Slack, Telegram, електронну пошту чи інші канали при порушенні визначених порогових значень. Таким чином, команда DevOps може швидко реагувати на проблеми ще до того, як вони вплинуть на користувачів.

Якщо порівнювати Prometheus та Grafana з іншими системами моніторингу, такими як Zabbix, Nagios або Datadog, їх переваги стають очевидними. Zabbix і Nagios, хоч і залишаються потужними класичними рішеннями, мають агентно-

орієнтовану модель збору даних, що ускладнює розгортання в динамічних контейнеризованих середовищах

Загальну схема інтеграції Prometheus та Grafana в кластері зображено на рис. 2.6.

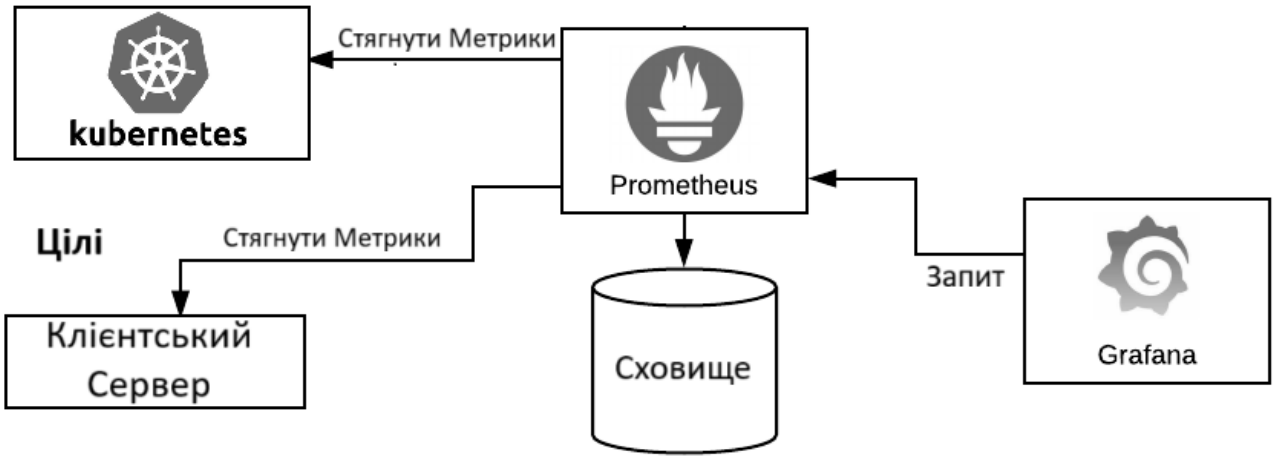

Рисунок 2.6 — Загальна схема інтеграції Prometheus та Grafana в кластері

Вони також гірше масштабуються та мають менш гнучкий механізм запитів до метрик. Datadog, у свою чергу, є комерційним SaaS-продуктом, який має зручний інтерфейс, але не надає повного контролю над збереженими даними та вимагає додаткових витрат. Натомість Prometheus і Grafana є повністю відкритими та самодостатніми рішеннями, які можна розгорнути на власних серверах або в хмарі, зберігаючи повний контроль над усією аналітичною інфраструктурою

У контексті розроблюваної системи розгортання прикладних сервісів вибір саме Prometheus і Grafana є оптимальним. Prometheus забезпечує гнучке та масштабоване збирання метрик із різних джерел — серверів, контейнерів, кластерів Kubernetes, мережевих інтерфейсів. Grafana, своєю чергою, перетворює ці метрики у наочні дашборди, що дозволяють в режимі реального часу оцінювати продуктивність системи, навантаження, стан окремих компонентів та тенденції змін. Разом вони створюють цілісну екосистему моніторингу, що відповідає принципам DevOps — прозорість, автоматизація, аналітика та швидкий зворотний зв’язок.

У даному розділі було здійснено комплексне технічне обґрунтування вибору інструментів, що формують архітектуру системи розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик. На основі аналізу сучасних підходів до автоматизації життєвого циклу програмного забезпечення було визначено оптимальний набір технологічних засобів, які забезпечують узгоджену, відтворювану та масштабовану інфраструктуру, здатну відповідати вимогам безперервної інтеграції, безперервної доставки та комплексної спостережуваності системи. Особливу увагу приділено мінімізації ручних операцій, стандартизації процесів та забезпеченню прозорості управління інфраструктурою на всіх етапах її експлуатації.

Обраний набір технологій — Terraform, Ansible, Docker, Kubernetes, Jenkins, Prometheus і Grafana — формує цілісну та взаємопов’язану екосистему DevOps-інструментів, що охоплює повний життєвий цикл прикладних сервісів: від автоматизованого створення та конфігурації інфраструктурних ресурсів до контейнеризації застосунків, організації конвеєрів CI/CD, збору телеметричних даних і їх подальшої аналітичної візуалізації. Terraform забезпечує декларативне управління інфраструктурою в хмарному середовищі, Ansible — ідемпотентну підготовку серверів та налаштування системного програмного забезпечення, Docker — уніфікацію середовища виконання прикладних сервісів, а Kubernetes — їх оркестрацію та керування життєвим циклом контейнерів.

Інструмент Jenkins відіграє ключову роль у реалізації процесів безперервної інтеграції та доставки, автоматизуючи збірку, тестування та розгортання прикладних сервісів у кластерному середовищі. Система моніторингу Prometheus у поєднанні з Grafana забезпечує збір, зберігання та візуалізацію метрик стану як інфраструктурних компонентів, так і прикладних сервісів, що дозволяє своєчасно виявляти відхилення в роботі системи та підвищує її експлуатаційну надійність.

3 РЕАЛІЗАЦІЯ СИСТЕМИ РОЗГОРТАННЯ ТА МОНІТОРИНГУ ПРИКЛАДНОГО СЕРВІСУ

3.1 Обґрунтування вибору мови програмування

Розглянемо реалізацію системи розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик, розробленої на основі попередньо обґрунтованих технологій. Як базовий приклад прикладного сервісу для практичної демонстрації обрано власний вебсервіс типу “ToDo List API”, реалізований мовою програмування Go (Golang). Даний сервіс має просту, але показову архітектуру, що включає типові CRUD-операції (Create, Read, Update, Delete), що дозволяє ефективно відтворити ключові аспекти процесів розгортання, масштабування, моніторингу та забезпечення доступності сучасних мікросервісних застосунків.

Мова програмування Go (Golang), розроблена інженерами компанії Google є однією з найвпливовіших сучасних технологій у сфері створення високопродуктивних серверних та хмарних застосунків. Вона поєднує у собі простоту синтаксису, притаманну мовам високого рівня, з ефективністю, властивою низькорівневим системним мовам, що робить її надзвичайно придатною для розроблення прикладних сервісів, які мають працювати з мінімальними затримками та високим рівнем надійності.

Go притаманна компіляція у машинний код безпосередньо, без проміжних інтерпретаторів або віртуальних машин, що забезпечує надзвичайно швидкий час виконання. На відміну від мов, які працюють через інтерпретатор (як Python або JavaScript), або потребують додаткових середовищ виконання (як JVM у Java), Go формує єдиний виконуваний бінарний файл, який можна легко розгортати на будь-якому сервері без зовнішніх залежностей. Такий підхід суттєво спрощує процес контейнеризації та безсерверного розгортання (serverless deployment), оскільки контейнер із Go-додатком містить лише сам виконуваний файл і не потребує інсталяції додаткових бібліотек чи рантаймів.

Вбудована підтримка конкурентності (concurrency) через механізм goroutine і канали (channels) дозволяє ефективно використовувати багатоядерні процесори та

виконувати численні операції одночасно, що є критично важливим для сучасних вебсерверів і мікросервісів, які обробляють велику кількість паралельних запитів. На відміну від традиційних потоків (threads) у таких мовах, як Java або C++, goroutine є набагато легшими — тисячі конкурентних процесів можуть працювати одночасно без помітного навантаження на систему. Такий підхід ідеально підходить для реалізації API-сервісів, які повинні забезпечувати швидку обробку запитів користувачів, масштабованість і низьку затримку відповіді. Формула середнього часу відповіді API-сервісу:

$$T_{сер} = \frac{1}{n}\sum_{i=1}^{n} T_I, \quad (3.1)$$

де T_I — час відповіді на і-й HTTP-запит;

n — кількість запитів.

Важливою характеристикою Go є її мінімалізм та однозначність. У мові відсутні складні парадигми, такі як спадкування або перевантаження операторів, що часто ускладнюють читання та супровід коду в інших об’єктно-орієнтованих мовах. Натомість Go базується на композиції та інтерфейсах, що забезпечує більш гнучку архітектуру програм і сприяє створенню чистих, зрозумілих та передбачуваних структур коду.

Go має вбудований інструментарій для розробки та автоматизації, що відповідає філософії DevOps. Мова постачається з власною системою керування пакетами (go mod), утилітами для форматування (go fmt), тестування (go test), профілювання (pprof) та збірки (go build). Такий підхід дозволяє створювати стандартизовані, керовані процеси збірки, що легко інтегруються у CI/CD-пайплайни, зокрема з такими інструментами, як Jenkins, GitHub Actions або GitLab CI. Зібраний бінарний файл Go легко розміщується в контейнері Docker, що спрощує подальше розгортання у Kubernetes або хмарному середовищі, такому як Google Cloud, AWS чи Azure.

З погляду продуктивності, Go займає проміжне місце між C/C++ і Java, забезпечуючи достатню швидкість виконання для більшості серверних задач при

значно нижчій складності розробки. Завдяки збірці сміття (garbage collector), розробникам не потрібно вручну керувати пам’яттю, що зменшує кількість критичних помилок, пов’язаних із витоками ресурсів. Водночас, Go GC оптимізовано для мінімізації затримок під час роботи в реальному часі, що робить її ефективною для високонавантажених API-сервісів.

Серед недоліків Go можна відзначити відсутність деяких високорівневих можливостей, характерних для динамічних мов, таких як узагальнене програмування (generics), яке було офіційно додано лише у пізніших версіях (починаючи з Go 1.18). Це частково обмежувало повторне використання коду та створення універсальних бібліотек. Також Go не підтримує класичне спадкування, що може ускладнювати розробку об’єктно-орієнтованих систем. Ще однією потенційною проблемою є невеликий розмір стандартної бібліотеки — хоча вона достатня для більшості системних задач, для складних вебпроєктів часто доводиться підключати сторонні пакети. Однак, ці обмеження одночасно сприяють стабільності та передбачуваності мови, що є цінним для промислових застосунків.

Вибір Go для розроблення прикладного сервісу ToDo List API у межах дослідження обґрунтовується саме її технічними характеристиками, орієнтованими на створення легких, швидких і стабільних мікросервісів, здатних ефективно інтегруватися у DevOps-інфраструктуру. Використання Go дозволяє побудувати застосунок, який споживає мінімум ресурсів, швидко компілюється, не потребує складних залежностей і готовий до контейнеризації відразу після збірки. У поєднанні з такими інструментами, як Docker та Kubernetes, застосунки на Go легко масштабуються та забезпечують надійне виконання навіть за високих навантажень.

3.2 Реалізація TODO API-сервісу

У представленому фрагменті коду реалізовано прикладний вебсервіс ToDo List API, який є типовим мікросервісом для управління задачами (tasks) у форматі RESTful API. Цей сервіс створено мовою Go (Golang) із використанням стандартної бібліотеки net/http та інтеграції з бібліотекою Prometheus Client для експорту метрик моніторингу. Структура та архітектурні рішення в цьому коді відповідають

принципам сучасної розробки прикладних сервісів, орієнтованих на контейнеризацію, оркестрацію та автоматизацію процесів розгортання.

Блок-схему прикладного API-сервісу показано на рисунку 3.1.

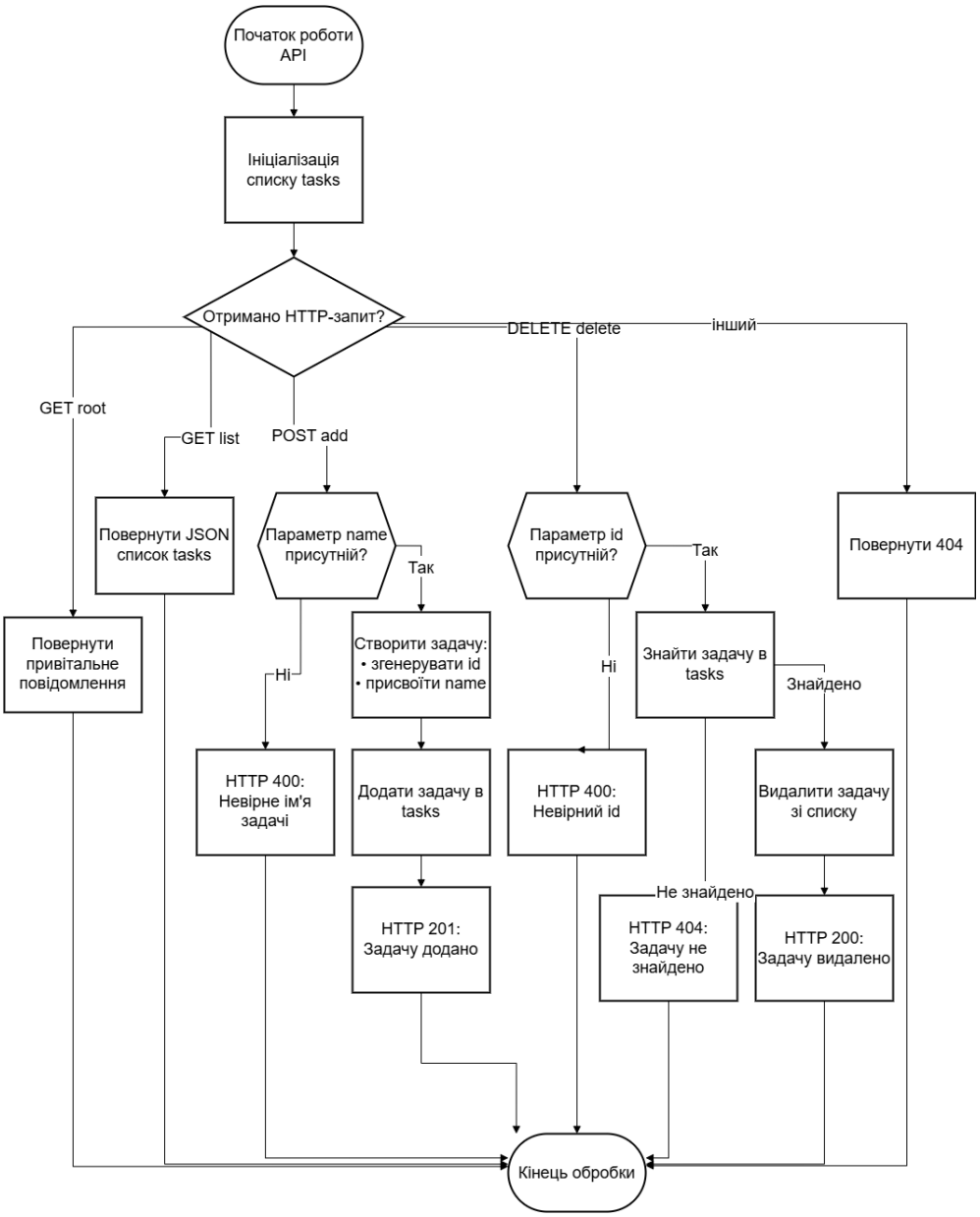

Рисунок 3.1 - Блок-схема прикладного API-сервісу

У основі сервісу лежить структура даних Task, яка описує окрему задачу зі списку:

```
type Task struct {
        ID   int    `json:"id"`
        Name string `json:"name"`
```

```
        Done bool   `json:"done"`
}
```

Кожен об’єкт Task містить унікальний ідентифікатор ID, текстову назву Name та булевий атрибут Done, що позначає стан виконання. JSON-теги забезпечують коректну серіалізацію даних у форматі, який використовується для передачі через HTTP.

Для управління колекцією задач використовується спеціальна структура TaskManager, що включає в себе вбудований м’ютекс (sync.RWMutex) для забезпечення потокобезпечного доступу до спільних даних у багатопоточному середовищі:

```
type TaskManager struct {
        sync.RWMutex
        tasks []Task
}
```

Цей підхід дозволяє уникнути станів гонки (data races) під час одночасного виконання операцій зчитування та модифікації задач, що є особливо важливим для вебсерверів, які обробляють кілька запитів паралельно. Змінна manager є глобальним екземпляром цієї структури і виступає центральним сховищем для всіх задач під час роботи сервісу. Зберігання даних відбувається в оперативній пам’яті, що є типовим рішенням для легковагових демонстраційних API-сервісів, де відсутня необхідність у персистентності даних.

Сервер реалізує низку HTTP-ендпоінтів, кожен з яких відповідає певній функції у межах CRUD-парадигми (Create, Read, Update, Delete). Реєстрація маршрутів відбувається через стандартний маршрутизатор http.NewServeMux(). Основними ендпоінтами є:

— tasks — універсальний ендпоінт для взаємодії зі списком задач;

— healthz — службовий ендпоінт для перевірки працездатності сервісу;

— metrics — ендпоінт для експорту метрик у форматі, сумісному з Prometheus.

Функції обробки запитів (handlers) побудовані з урахуванням принципів коректної обробки HTTP-протоколу, включаючи коди статусів, перевірку вхідних даних і типів запитів.

Функція listTasks здійснює безпечне зчитування колекції задач із блокуванням RLock() (режим лише для читання) і повертає результат у форматі JSON. У заголовках встановлюється Content-Type: application/json, а статус відповіді — 200 OK.

```
func listTasks(w http.ResponseWriter, r *http.Request) {
    totalRequests.WithLabelValues("/tasks", r.Method).Inc()
    manager.RLock()
    defer manager.RUnlock()
    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(manager.tasks)
}
```

Функція addTask реалізує створення нової задачі через POST-запит. Тіло запиту очікується у форматі JSON. Після валідації даних нова задача додається до колекції з автоматичним присвоєнням унікального ID. Повертається серіалізований об’єкт задачі та статус 201 Created.

```
func addTask(w http.ResponseWriter, r *http.Request) {
    totalRequests.WithLabelValues("/tasks", r.Method).Inc()
    if r.Method != http.MethodPost {
        http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
        return
    }
    var newTask Task
    if err := json.NewDecoder(r.Body).Decode(&newTask); err != nil {
        http.Error(w, "Invalid JSON format", http.StatusBadRequest)
        return
    }
    defer r.Body.Close()
    if newTask.Name == "" {
        http.Error(w, "Task name cannot be empty", http.StatusBadRequest)
        return
    }
    ...
}
```

Функція deleteTask виконує видалення задачі за ідентифікатором, переданим у параметрах запиту (?id=1). Використовується перевірка типу методу (DELETE) і валідація параметра. У разі успішного видалення клієнт отримує JSON-відповідь із повідомленням "Task deleted successfully". Якщо задача не знайдена — повертається статус 404 Not Found.

Функція healthHandler повертає базову інформацію про стан застосунку, що дозволяє автоматизованим системам моніторингу переконатися у стабільності роботи процесу.

```
func healthHandler(w http.ResponseWriter, r *http.Request) {
	totalRequests.WithLabelValues("/healthz", r.Method).Inc()
	w.WriteHeader(http.StatusOK)
	w.Write([]byte("OK"))
}
```

У коді інтегровано базові Prometheus-метрики за допомогою пакета github.com/prometheus/client_golang/prometheus. Для кожного HTTP-запиту інкрементується лічильник totalRequests, який підраховує кількість звернень до API за маршрутом і методом:

```
var (
	totalRequests = prometheus.NewCounterVec(
		prometheus.CounterOpts{
			Name: "todo_api_http_requests_total",
			Help: "Загальна кількість HTTP-запитів за ендпоінтом і методом.",
		},
		[]string{"path", "method"},
	)
)
```

Цей показник використовується для подальшого аналізу навантаження на сервіс, виявлення аномалій у трафіку або неочікуваного збільшення кількості запитів певного типу. Експортування відбувається через окремий маршрут /metrics, який обслуговує вбудований HTTP-хендлер Prometheus (promhttp.Handler()). Порт, на якому працює сервер, визначається через змінну середовища PORT, що відповідає DevOps-принципу “configuration via environment” і полегшує

контейнеризацію сервісу. Якщо змінна не задана, використовується порт 8080 за замовчуванням. Для забезпечення стабільності під час зупинки або оновлення контейнера реалізовано механізм Graceful Shutdown. Сервіс реагує на сигнали SIGINT та SIGTERM, після чого коректно завершує активні з'єднання та звільняє ресурси протягом встановленого таймауту (5 секунд).

Структура та логіка роботи сервісу відповідають вимогам до сучасних мікросервісних архітектур, що використовуються у DevOps-середовищах:

— сервіс має чіткий RESTful API і не містить зовнішніх залежностей;

— підтримує health-check та метрики, необхідні для Kubernetes і Prometheus;

— конфігурація виконується через змінні середовища, що забезпечує гнучкість при розгортанні у Docker чи Kubernetes;

— реалізовано потокобезпечну роботу через механізм sync.RWMutex;

— забезпечено Graceful Shutdown, що дозволяє проводити оновлення без переривання сервісу.

У даному розділі було представлено процес реалізації прикладного сервісу у вигляді ToDo List API, створеного мовою програмування Go, а також здійснено обґрунтування вибору відповідних технологій і середовищ для його розгортання в рамках системи DevOps. На основі проведеного аналізу було встановлено, що Go є оптимальним рішенням для розроблення мікросервісних застосунків завдяки своїй високій продуктивності, мінімалістичному синтаксису, ефективному управлінню конкурентністю та легкій інтеграції у процеси контейнеризації та автоматизованого розгортання. Особливістю розробленого API є його модульна структура, що дозволяє забезпечити чітке розділення логіки взаємодії з користувачем, обробки запитів та керування даними, що в сукупності сприяє підвищенню стабільності, масштабованості та зрозумілості коду.

Розроблений ToDo List API виконує роль демонстраційного прикладного сервісу, який відображає практичне застосування сучасних підходів до автоматизованого розгортання, моніторингу та підтримки сервісів у хмарному середовищі. Використання мови Go в даному контексті не лише сприяє побудові високопродуктивної архітектури, але й демонструє її доцільність для побудови

масштабованих, надійних і керованих систем, що відповідають вимогам сучасної DevOps-парадигми. Таким чином, обраний технологічний стек та реалізовані рішення формують основу для подальшої інтеграції інструментів моніторингу та візуалізації, що забезпечить повноцінне функціонування системи розгортання та контролю прикладних сервісів.

3.3 Розгортання та підготовка серверної інфраструктури за допомогою Terraform та Ansible

Розгортання системи здійснюється на віртуальному сервері, розміщеному в інфраструктурі Google Cloud Platform (GCP) — одному з провідних хмарних середовищ, що надає широкі можливості для створення, масштабування та моніторингу програмних систем. Google Cloud характеризується високим рівнем надійності, продуктивності та безпеки, забезпечуючи SLA до 99,95% доступності, що є критично важливим для демонстрації стійкості системи розгортання. Використання GCP також дозволяє застосовувати гнучке керування ресурсами через Google Compute Engine (GCE) — сервіс для створення та адміністрування віртуальних машин. Це забезпечує ізольоване середовище, у якому можливо автоматизовано створити інфраструктуру за допомогою Terraform, налаштувати її з використанням Ansible та розгорнути контейнери за допомогою Kubernetes.

Google Cloud надає всі необхідні сервіси для підтримки DevOps-процесів, включаючи Cloud Storage, VPC (Virtual Private Cloud), IAM (Identity and Access Management), а також можливість інтеграції з зовнішніми інструментами CI/CD, такими як Jenkins. Завдяки цьому середовище GCP стає універсальною платформою для розгортання як окремих контейнерних сервісів, так і повноцінних кластерних систем, орієнтованих на масштабованість та безперервну доставку оновлень.

У межах реалізації системи розгортання та моніторингу прикладного сервісу важливим етапом є формування керованого та відтворюваного середовища виконання. Для досягнення цього застосовано принципи Infrastructure as Code (IaC), що дозволяють опчисати інфраструктуру декларативним способом і отримати

стабільне середовище з чіткою фіксацією змін та можливістю повного відтворення. Центральним інструментом для автоматизації створення інфраструктури було обрано Terraform, який забезпечує універсальний механізм роботи з провайдерами хмарних сервісів, зокрема Google Cloud Platform (GCP). Наведена конфігурація спрямована на створення мінімалістичного, але повністю функціонального серверного середовища, достатнього для виконання завдань дипломної роботи.

Першим елементом конфігурації є визначення провайдера GCP, що задає параметри аутентифікації та географічні характеристики середовища. Оголошення

```
provider "google" {
  project = var.project_id
  region  = "us-central1"
}
```

Вказує Terraform, що всі наступні ресурси мають створюватися в межах визначеного проєкту GCP та у регіоні us-central1. Обраний регіон є одним з економічно оптимальних для розгортання віртуальних машин класу E2, що дає змогу зменшити витрати без втрати продуктивності та доступності.

Наступним важливим елементом конфігурації є створення статичних зовнішніх IP-адрес для майбутніх інстансів:

```
resource "google_compute_address" "static_ip" {
  name   = "diploma-static-ip"
  region   = "us-central1"
}
```

Статична IP-адреса необхідна для стабільного доступу до розгорнутого сервера, оскільки вона забезпечує незмінність точки входу для SSH-доступу, CI/CD-процесів, моніторингових систем та викликів API.

Створення основних обчислювальних ресурсів здійснюється через опис інстансів:

```
resource "google_compute_instance" "custom_e2_instance" {
  name         = "diploma-vm"
  machine_type = "e2-custom-8-16384"
  zone         = "us-central1-a"
```

У конфігурації визначено інстанси типу e2-custom, які базуються на параметрах 8 vCPU та 16 GB RAM. Такий обсяг ресурсів обрано як оптимальний для одночасного розгортання прикладного сервісу, допоміжних компонентів і можливих навантажень, пов’язаних із тестуванням, CI/CD-процесами та моніторингом. Вибір зони us-central1-a забезпечує низьку латентність, високу доступність та стабільність.

Блок boot_disk задає конфігурацію системного диска інстанса:

```
image = "ubuntu-os-cloud/ubuntu-2204-lts"
size  = 225
type  = "pd-standard"
```

Використання Ubuntu 22.04 LTS забезпечує стабільне, підтримуване та сумісне середовище для DevOps-інструментів і контейнеризації. Розмір у 225 ГБ обрано з урахуванням вимог до розміщення логів, кешів, артефактів CI/CD та потенційного використання Docker-образів. Стандартний тип диска (pd-standard) є економічним рішенням, достатнім для більшості сценаріїв навантаження.

Мережева конфігурація визначається блоком:

```
network_interface {
  network = "default"
  access_config {
    nat_ip = google_compute_address.static_ip.address
  }
}
```

Цей фрагмент забезпечує прив’язку попередньо створеної статичної IP-адреси до інстанса. Використання стандартної VPC спрощує інтеграцію з інтерфейсами управління та тестуванням.

Для забезпечення безпечного доступу та автоматичної конфігурації під час запуску використано блок метаданих:

```
metadata = {
  ssh-keys       = "${var.instance_user}:${var.public_ssh_key}"
  startup-script = templatefile("scripts/setup-user.sh", { INSTANCE_USER = var.instance_user })
}
```

SSH-ключі гарантують безпечний доступ до інстанса згідно з рекомендаціями щодо кращих практик хмарної безпеки. Стартовий скрипт дозволяє автоматизувати початкове налаштування системи: створення користувача, встановлення основних пакунків, підготовку середовища до подальшого конфігураційного управління за допомогою Ansible.

У підсумку дана Terraform-конфігурація формує легковагову, економічну й водночас функціонально повноцінну інфраструктуру, яка повністю відповідає вимогам роботи. Вона забезпечує швидке розгортання сервера, можливість автоматизованого налаштування та інтеграцію зі всіма подальшими модулями системи — Ansible, Kubernetes, CI/CD-процесами та комплексним моніторингом.

Структурну схему розгортання та підготовки серверної інфраструктури показано на рисунку 3.2.

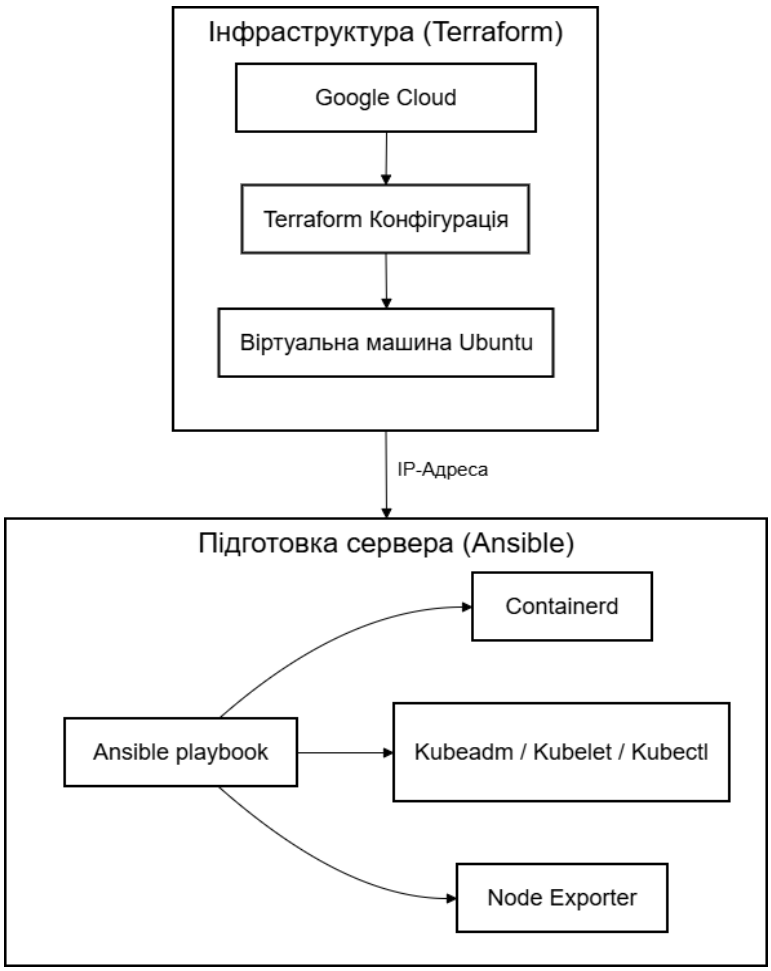

Рисунок 3.2 — Структурна схема розгортання та підготовки серверної інфраструктури

На етапі підготовки інфраструктури до розгортання контейнеризованого прикладного сервісу критично важливим є формування стандартизованого та відтворюваного серверного середовища. У межах цієї роботи процес автоматизації конфігурації серверу реалізовано за допомогою системи керування конфігураціями Ansible, яка дозволяє декларативно описувати стан операційної системи, програмного забезпечення та служб, забезпечуючи повторюваність та контрольованість усіх виконуваних операцій. Розроблений плейбук спрямований на формування повністю функціонального однонодового Kubernetes-вузла, який слугуватиме базовою платформою для подальшого розгортання Go-орієнтованого API-сервісу та інтегрованих компонентів моніторингу.

Першим етапом роботи Ansible є підготовка операційної системи, що включає оновлення списку пакетів, встановлення оновлень системи та додаткових утиліт, необхідних для подальшої роботи контейнерного runtime та Kubernetes-компонентів. У плейбуку це реалізується за допомогою задач:

```
- name: Update apt cache
  apt:
    update_cache: yes
- name: Install required packages
  apt:
    pkg:
      - apt-transport-https
      - ca-certificates
      - curl
```

Цей етап гарантує, що сервер отримує останні патчі безпеки та має коректно сформоване пакувальне середовище для подальшої інсталяції компонентів Docker та Kubernetes.

Наступним ключовим елементом підготовки середовища є встановлення контейнерного runtime, оскільки Kubernetes використовує саме його для створення, запуску та керування контейнерами. У межах роботи було обрано Docker Engine як один із найбільш стабільних, зрілих та добре підтримуваних runtime-ів.

Встановлення Docker починається з додавання відповідного репозиторію та ключа підпису:

```
- name: Add Docker repository
  apt_repository:
    repo: "deb [arch=amd64 signed-by=/usr/share/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu jammy stable"
- name: Install Docker Engine
  apt:
    name:
      - docker-ce
      - docker-ce-cli
      - containerd.io
```

Після інсталяції Docker служба активується як системний демон, що забезпечує її автоматичний запуск після завантаження системи. Це гарантує стабільну роботу контейнеризованих процесів навіть після можливих перезапусків сервера. Надалі виконується встановлення компонентів Kubernetes, необхідних для створення однонодового кластеру. Основними елементами є kubelet, kubeadm та kubectl, кожен з яких виконує окрему системну функцію: kubelet відповідає за локальне управління контейнерами, kubeadm забезпечує процес ініціалізації кластеру, а kubectl слугує основним інтерфейсом комунікації з Kubernetes API. Їх інсталяція відбувається через офіційний репозиторій Kubernetes:

```
- name: Install Kubernetes components
  apt:
    pkg:
      - kubelet
      - kubeadm
      - kubectl
```

Перед ініціалізацією кластеру вимикається свопінг, оскільки Kubernetes вимагає гарантованої передбачуваності використання пам’яті. Цей крок є обов’язковим:

```
- name: Disable swap
  command: swapoff -a
```

Після виконання попередніх завдань плейбук ініціалізує Kubernetes-кластер за допомогою kubeadm init, що формує контрольну площину і налаштовує структуру кластерного управління:

```
- name: Initialize Kubernetes cluster
  shell: kubeadm init --pod-network-cidr=192.168.0.0/16
```

У результаті створюється конфігураційний файл admin.conf, який копіюється до робочого каталогу адміністратора, забезпечуючи можливість використання kubectl для подальших керувальних операцій. Оскільки кластер у межах даної роботи є однонодовим, контрольній площині дозволяється розміщувати контейнеризовані робочі навантаження. Це досягається шляхом видалення taint-мітки. Для забезпечення мережевої взаємодії між подами встановлюється CNI-плагін, у даному випадку — Calico, який забезпечує маршрутизацію, політики мережевої безпеки та масштабованість мережевої архітектури.

```
kubectl taint nodes --all node-role.kubernetes.io/control-plane-
kubectl apply -f https://raw.githubusercontent.com/projectcalico/calico/...
```

Заключним етапом конфігурації є встановлення компонента моніторингу Node Exporter, що забезпечує збір низькорівневих метрик вузла, включно з завантаженням процесора, обсягом вільної пам’яті, I/O-операціями та станом файлової системи. Node Exporter розміщується як окрема служба systemd:

```
- name: Create systemd service for Node Exporter
  copy:
    dest: /etc/systemd/system/node_exporter.service
```

Після цього служба активується та автоматично запускається при кожному старті системи. Розроблений Ansible-плейбук забезпечує комплексну автоматизовану підготовку серверного середовища, включаючи встановлення контейнерного runtime, конфігурацію компонентів Kubernetes, розгортання мережевої інфраструктури та налаштування системи збору телеметричних даних. Це створює повністю відтворювану інфраструктурну основу, необхідну для подальшого розгортання прикладного сервісу, тестування CI/CD-процесів та інтеграції систем моніторингу.

3.4 Налаштування основних компонентів кластеру за допомогою Kubernetes

У процесі розроблення та впровадження архітектури однонодового Kubernetes-кластера ключовим завданням є раціональне структурування всіх компонентів, включаючи прикладний сервіс, інструменти моніторингу, системи візуалізації та засоби безперервної інтеграції. Обрана модель одновузлового розгортання дозволяє поєднати компактність та мінімальні вимоги до обчислювальних ресурсів із повноцінним застосуванням сучасних технологій контейнеризації, оркестрації та експозиції метричних даних. У межах запропонованої інфраструктури центральним елементом є власноруч створений прикладний сервіс — Todo List API, розроблений мовою Go та орієнтований на демонстрацію ключових аспектів розгортання мікросервісних додатків у Kubernetes.

Основу всієї інфраструктури становить ієрархічна структура простору імен Kubernetes, у межах якої кожен логічний компонент системи розміщено в окремому namespace. Такий підхід забезпечує чітку ізоляцію конфігураційних ресурсів, спрощує управління політиками доступу, а також сприяє логічному поділу функціональних підсистем відповідно до їх призначення. Використання просторів імен дозволяє уникнути конфліктів імен ресурсів, підвищує керованість кластера та створює передумови для масштабування системи або її подальшого розширення без порушення вже розгорнутих компонентів.

Для прикладної частини системи створено окремий простір імен todo-app, у межах якого розміщено основні об’єкти розгортання прикладного сервісу, зокрема ресурси типу Deployment та Service. Deployment відповідає за керування життєвим циклом контейнеризованого API-сервісу, забезпечуючи декларативне визначення кількості реплік, політик оновлення та відновлення у разі відмови окремих подів. Service, у свою чергу, реалізує стабільну мережеву абстракцію для доступу до прикладного сервісу, незалежно від динамічних змін у складі подів, що є характерною особливістю оркестрованого середовища.

Структурну схему кластера показано на рисунку 3.3.

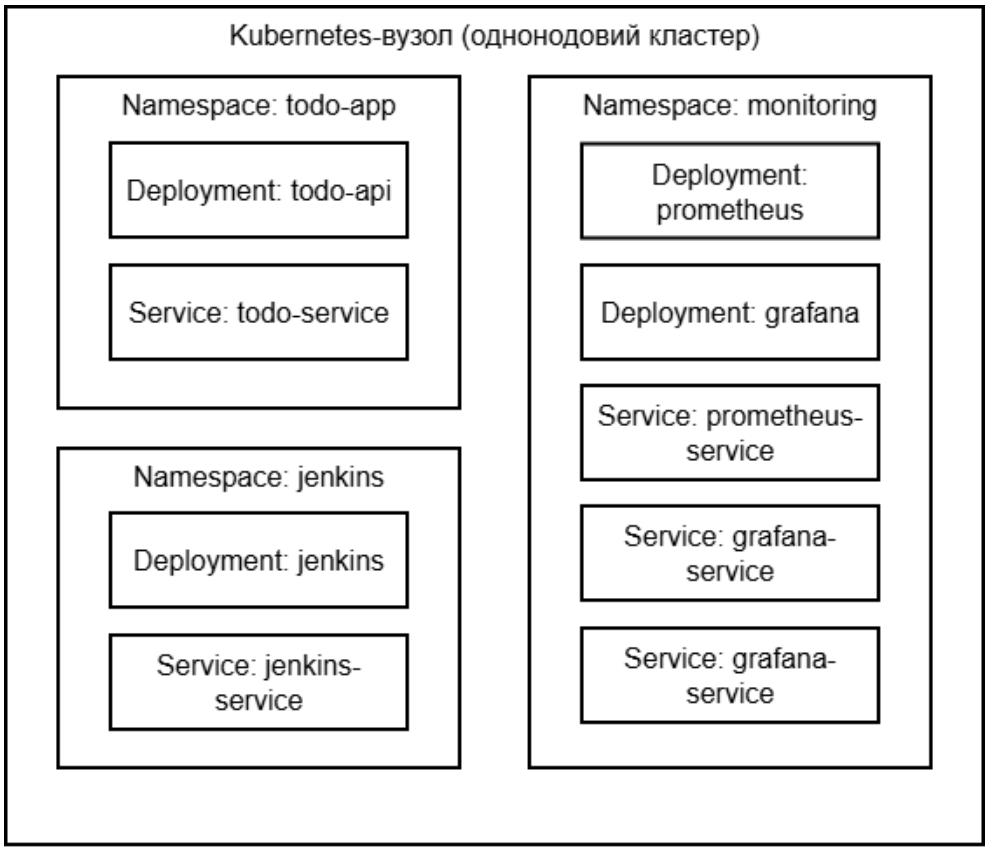

Рисунок 3.3 — Структурна схема кластера

У межах простору імен todo-app основним об’єктом є Deployment, який визначає життєвий цикл контейнеризованого сервісу. Структура Deployment передбачає створення ReplicaSet із однією реплікою, що є цілком виправданим для однонодового навчального кластера, але водночас не суперечить принципам горизонтального масштабування, що можуть бути застосовані у майбутньому. У маніфесті визначено контейнер, побудований на основі оптимізованого Docker-образу, який містить скомпільований Go-бінарник. Важливим елементом конфігурації є секція ports, у якій зазначено експонування порту 8080, що відповідає мережевому інтерфейсу сервісу. Крім того, для гарантування надійності застосовано проби готовності та живучості, задані структурами livenessProbe та readinessProbe, які здійснюють періодичне опитування HTTP-ендпоінтів /healthz сервісу. Це дозволяє Kubernetes визначати працездатність сервісу та автоматично здійснювати його перезапуск у разі виявлення некоректної роботи, фактично забезпечуючи безперервне функціонування без ручного втручання.

```
 readinessProbe:
          httpGet:
            path: /healthz
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 5
          failureThreshold: 3
        livenessProbe:
          httpGet:
            path: /healthz
            port: 8080
          initialDelaySeconds: 15
          periodSeconds: 10
          failureThreshold: 5
```

Комунікацію між подами та зовнішнім середовищем забезпечує об’єкт Service типу ClusterIP, який створює стабільну віртуальну точку доступу до API-сервісу. Завдяки використанню селектора app: todo-api Service динамічно спрямовує трафік на актуальні поди, навіть у разі оновлення або перезапуску контейнерів. Така модель відповідає парадигмі сервіс-орієнтованих систем, у яких поди вважаються ефемерними, натомість сервіс забезпечує сталість доступу.

```
apiVersion: v1
kind: Service
metadata:
  name: todo-service
  namespace: todo
  labels:
    app: todo-api
spec:
  selector:
    app: todo-api
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Розгортання прикладного API-сервісу узгоджується зі стандартними CI/CD-практиками — Docker-образ генерується заздалегідь, надсилається до реєстру, а Kubernetes отримує нову версію шляхом оновлення поля image у Deployment. Це дозволяє забезпечити декларативний підхід до керування версіями сервісу, а також

миттєве оновлення контейнерів без зупинки всього кластера. Важливо, що Deployment передбачає можливість майбутнього масштабування та встановлення стратегій оновлення, але для даної роботи використано конфігурацію за замовчуванням, що демонструє базові механізми Kubernetes без надлишкової складності.

Другим важливим компонентом є підсистема моніторингу, розгорнута у просторі імен monitoring. У межах цього простору на рівні кластера встановлено Prometheus, що виконує функції збору, зберігання та агрегації метричних даних. Конфігурація Prometheus передбачає визначення scrape_config, що вказує на адресу Node Exporter, встановленого на вузлі під час підготовки сервера. Таким чином Prometheus отримує сталі показники стану вузла (процесорне навантаження, використання пам’яті, файлових систем тощо), що дозволяє підтримувати базову інфраструктурну видимість. Complimenting Prometheus, система Grafana розгорнута як окремий Deployment із доступом до сервісу Prometheus через відповідно налаштовані джерела даних. Для цілей роботи Grafana використовується переважно як інструмент візуалізації агрегованих метрик, що дає змогу формувати інформаційні панелі для моніторингу стану вузла та роботи прикладного сервісу.

```
spec:
    containers:
     - name: prometheus
       image: prom/prometheus:v2.44.0
       args:
         - "--config.file=/etc/prometheus/prometheus.yml"
         - "--storage.tsdb.path=/prometheus"
         - "--web.console.libraries=/usr/share/prometheus/console_libraries"
         - "--web.console.templates=/usr/share/prometheus/consoles"
       ports:
         - containerPort: 9090
           name: web
```

Окремим функціональним компонентом є Jenkins, розгорнутий у просторі імен jenkins, що виконує роль системи безперервної інтеграції та доставки. Його конфігурація визначена у стандартизованих Deployment та Service маніфестах, що

забезпечують доступ до Jenkins через веб-інтерфейс та інтеграцію з Kubernetes при виконанні CI/CD конвеєра. Jenkins відповідає за автоматизацію побудови Docker-образів, їх пуш до реєстрів та оновлення розгортань у просторі імен todo-app. Разом з Kubernetes він формує завершений цикл безперервної доставки, повністю інтегрований із контейнеризованою інфраструктурою.

3.5 Тестування системи

Даний розділ в дипломній роботі містить детальний опис процесу перевірки працездатності та ефективності розробленої системи. У цьому розділі будуть представлені основні етапи тестування, які включають в себе перевірку кожного компонента системи та її функціональності в цілому.

Після розгортання всіх необхідних сервісів у кластері потрібно перевірити працездатність прикладного API-сервісу. Перевірка буде проходити за допомогою http-запитів, які дозволять нам з’ясувати стан сервісу, додати нове завдання у систему та переглянути всі існуючі завдання.

Спочатку необхідно перевірити стан сервісу. Для цього необхідно зробити http запит на маршрут /healthz, який відповідає за статус сервісу. Якщо статус “OK” це означає, що сервіс працює.

Перевірка статусу API-сервісу зображено на рисунку 3.4.

```
esc@DESKTOP-C9QHH1R:~/go-todo-api$ curl -X GET api.example.com:8080/healthz && echo ""
OK
```

Рисунок 3.4 — Перевірка статусу API-сервісу

Після перевірки статусу сервісу створимо нове завдання, для перевірки методу створення. Для цього необхідно відправити http запит з методом POST на маршрут /tasks.

Створення нового завдання за допомогою API-сервісу показано на рисунку 3.5.

```
esc@DESKTOP-C9QHH1R:~/go-todo-api$ curl -X POST api.example.com:8080/tasks -d "{\"name\":\"New Task\"}"
{"id":1,"name":"New Task","done":false}
```

Рисунок 3.5 — Створення нового завдання за допомогою API-сервісу

Щоб переглянути список всіх створених завдань та їх статус необхідно виконати http запит з методом GET на маршрут /tasks.

Отримання списку всіх завдань за допомогою API-сервісу показано на рисунку 3.6.

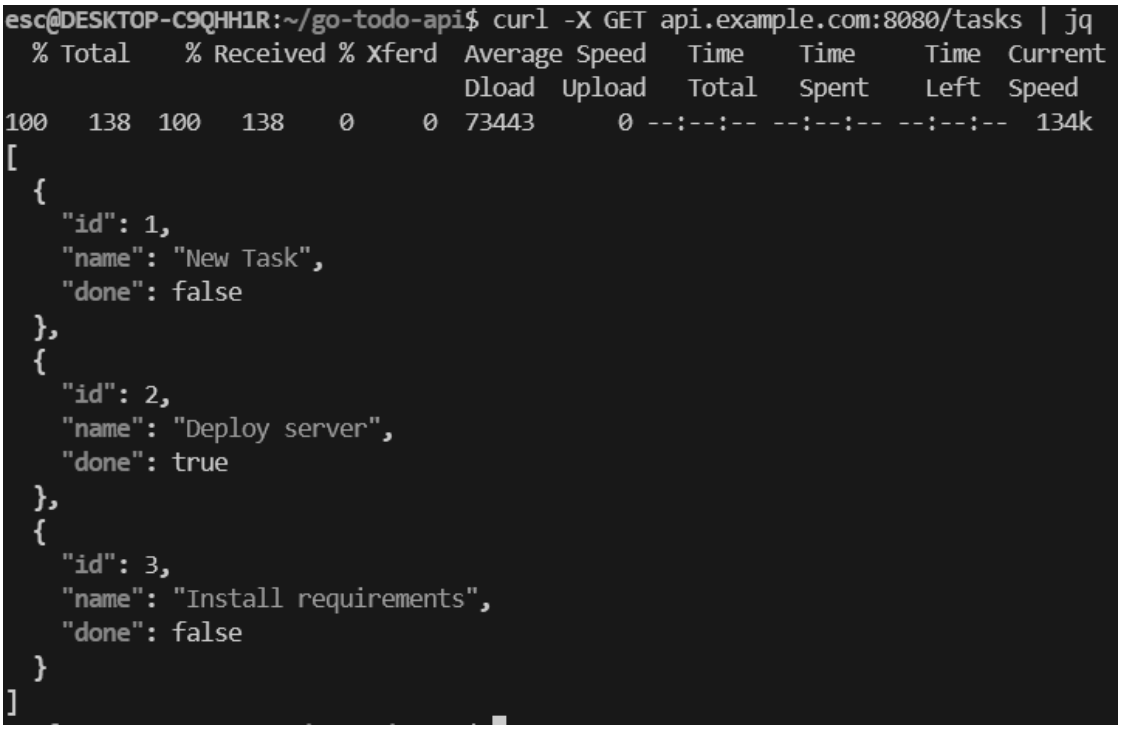

Рисунок 3.6 — Отримання списку всіх завдань за допомогою API-сервісу

Стандартний дашборд Node Exporter у системі Grafana надає можливість здійснювати комплексний моніторинг основних апаратних та системних метрик сервера, на якому розгорнуто прикладний сервіс. Зокрема, за допомогою цього дашборду відображаються показники завантаження центрального процесора, використання оперативної пам’яті, стану файлових систем, дискової підсистеми, мережевої активності, а також параметри, пов’язані з роботою ядра операційної системи.

Серверні метрики розгорнутого серверу показано на рисунку 3.7.

Рисунок 3.8 — Серверні метрики розгорнутого серверу

У підсумку проведеної роботи сформовано комплексне та узгоджене середовище для розгортання, експлуатації й моніторингу прикладного сервісу, що повністю відповідає принципам сучасних DevOps-практик і забезпечує автоматизований життєвий цикл програмного забезпечення. У межах розділу було реалізовано повноцінну інфраструктуру, починаючи з підготовки серверного середовища засобами Terraform та Ansible і завершуючи створенням однонодового Kubernetes-кластера, у якому розгорнуто прикладний сервіс todo-list API, модуль збору метрик Prometheus, систему візуалізації Grafana та платформу автоматизації CI/CD Jenkins. Кожен компонент системи було інтегровано таким чином, щоб забезпечити відмовостійку роботу, передбачуваність процесів розгортання, спостережуваність сервісу та можливість безперервного оновлення.

Описана архітектура демонструє переваги застосування контейнеризації та оркестрації навіть у рамках однонодового середовища, оскільки Kubernetes забезпечує структурованість розгортання, контрольованість станів застосунків і стандартизацію процесів. Інтеграція Prometheus і Grafana надала системі можливість здійснювати точний моніторинг продуктивності, доступності та стану вузла й контейнерів, тоді як Jenkins забезпечив можливість автоматизованої доставки нових версій сервісу, що істотно зменшує ризики людських помилок і скорочує час виходу оновлень.

4 ЕКОНОМІЧНА ЧАСТИНА

4.1 Комерційний та технологічний аудит науково-технічної розробки

Метою даного розділу є проведення технологічного аудиту розроблюваної системи автоматизованого розгортання та моніторингу прикладних сервісів. Особливістю створюваного рішення є підвищення ефективності та швидкості виконання операцій розгортання, оновлення й контролю стану сервісів за рахунок поєднання інфраструктурної автоматизації, контейнеризації, оркестрації та засобів моніторингу. Аналогом подібних рішень можуть виступати комплексні комерційні платформи управління DevOps-процесами, які поєднують CI/CD, інфраструктурне управління та моніторинг. Наприклад, використання GitLab Ultimate або інших платформ корпоративного рівню з DevOps послугами.

Для проведення комерційного та технологічного аудиту системи залучають не менше трьох незалежних експертів. В якості експертів виступили викладачі кафедри ОТ факультету ІТКІ ВНТУ. Перший експерт Кожем'яко А.В. доцент кафедри ОТ, другий експерт Тарновський М.Г. доцент кафедри ОТ, третій ексепрт Дудник О.В. доцент кафедри ОТ. Оцінювання науково-технічного рівня розробки та її комерційного потенціалу здійснюється за п'ятибальною системою за 12 критеріями, у відповідності до табл. 4.1.

Таблиця 4.1 – Рекомендовані критерії оцінювання комерційного потенціалу розробки та їх можлива бальна оцінка

Бали (за 5-ти бальною шкалою)					
Кри-те-рій	0	1	2	3	4
Технічна здійсненність концепції					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено роботоздат-ність продукту в реаль-них умовах
	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому рин-ку	Продукт не має аналогів на великому ринку

Продовження табл. 4.1

Ринкові переваги					
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів
4	Технічні та споживчі властивості продукту значно гірші, ніж в аналогів	Технічні та споживчі властивості продукту трохи гірші, ніж в аналогів	Технічні та споживчі властивості продукту на рівні аналогів	Технічні та споживчі властивості продукту трохи кращі, ніж в аналогів	Технічні та споживчі властивості продукту значно кращі, ніж в аналогів
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційних витрат аналогів	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної динаміки	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною динамікою	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих компаній на ринку	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практик на здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання наявних фахівців	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так із комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування ідеї відсутні	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово-промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві

Продовження табл. 4.1

11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Усі дані по кожному параметру занесено в таблиці 4.2

Таблиця 4.2 – Результати оцінювання комерційного потенціалу розробки

Критерії оцінювання	ПІБ експертів		
	Кожем'яко А.В.	Тарновський М.Г.	Дудник О.В.
	Бали		
Технічна здійсненність концепції	3	4	4
Наявність аналогів на ринку	3	3	4
Цінова політика	4	4	4
Технічні та споживчі властивості виробу	4	3	4
Експлуатаційні витрати	4	4	3
Ринок збуту	4	3	4
Конкурентоспроможність	3	4	3
Фахівці з технічної і комерційної реалізації	4	3	3
Фінансування	4	4	3
Матеріально-технічна база	3	3	3
Термін реалізації ідеї	4	4	3
Супровідна документація	4	3	3
Сума	44	42	41
Середньоарифметична сума балів	(44+42+41) / 3 = 42,33		

За даними таблиці 4.2 можна зробити висновок щодо рівня комерційного потенціалу даної розробки. Для цього доцільно скористатись рекомендаціями, наведеними в таблиці 4.3.

Таблиця 4.3 - Рівні комерційного потенціалу розробки

Середньоарифметична сума балів, розрахована на основі висновків експертів	Рівень комерційного потенціалу розробки
0-10	Низький
11-20	Нижче середнього
21-30	Середній
31-40	Вище середнього
41-48	Високий

Як видно з таблиці, рівень комерційного потенціалу розроблюваного нового програмного продукту є високим, що досягається за рахунок підвищення захищеності інформаційно комунікаційних систем шляхом адаптації методів управління ризиками інформаційної безпеки для визначення оптимального підходу до оцінки ризиків для підприємств в межах розробки комп'ютеризованої системи моніторингу безпеки об'єктів.

4.2 Прогнозування витрат на виконання науково-дослідної (дослідно-конструкторської) роботи

4.2.1 Основна заробітна плата розробників, яка розраховується за формулою:

$$З_о = \frac{М}{Т_р} \cdot t, \qquad (4.1)$$

де М – місячний посадовий оклад конкретного розробника (дослідника), грн.;

$Т_р$ – число робочих днів за місяць, 23 днів;

t – число днів роботи розробника (дослідника).

Результати розрахунків зведемо до таблиці 4.4.

Таблиця 4.4 — Основна заробітна плата розробників

Найменування посади	Місячний посадовий оклад, грн.	Оплата за робочий день, грн.	Число днів роботи	Витрати на заробітну плату, грн.
Керівник проекту	40000	1739,13	40	69565,217
Програміст	38000	1652,17	40	66086,957
Всього				135652,174

Так як в даному випадку розробляється програмний продукт, то розробник виступає одночасно і основним робітником, і тестувальником розроблюваного програмного продукту.

4.2.2 Додаткова заробітна плата розробників, які брати участь в розробці обладнання/програмного продукту.

Додаткову заробітну плату прийнято розраховувати як 11,5 % від основної заробітної плати розробників та робітників:

$$З_Д = З_О \cdot 11{,}5\% \,/\, 100\% \qquad (4.2)$$

$$Зд = (135652{,}174 \cdot 11{,}5\% \,/\, 100\%) = 15600{,}00 \text{ (грн.)}$$

4.2.3 Нарахування на заробітну плату розробників.

Згідно діючого законодавства нарахування на заробітну плату складають 22 % від суми основної та додаткової заробітної плати.

$$Н_З = (З_О + З_Д) \cdot 22\% \,/\, 100\% \qquad (4.3)$$

$$Н_з = (135652{,}174 + 15600{,}00) \cdot 22\% \,/\, 100\% = 33275{,}48 \text{ (грн.)}$$

4.2.4 Витрати на матеріали та комплектуючі.

Оскільки для розроблювального пристрою не потрібно витрачати матеріали та комплектуючі, то витрати на матеріали і комплектуючі дорівнюють нулю.

4.2.5 Амортизація обладнання, яке використовувалось для проведення розробки.

Амортизація обладнання, що використовувалось для розробки в спрощеному вигляді розраховується за формулою:

$$А = \frac{Ц}{Т_в} \cdot \frac{t_{вик}}{12} \text{ [грн.]}. \quad (4.4)$$

де Ц – балансова вартість обладнання, грн.;

Т – термін корисного використання обладнання згідно податкового законодавства, років;

$t_{вик}$ – термін використання під час розробки, місяців.

Розрахуємо, для прикладу, амортизаційні витрати на комп'ютер балансова вартість якого становить 30000 грн., термін його корисного використання згідно податкового законодавства – 2 роки, а термін його фактичного використання – 1,74 міс.

$$А_{обл} = \frac{30000}{2} \times \frac{1{,}74}{12} = 2174 \text{ грн.}$$

Аналогічно визначаємо амортизаційні витрати на інше обладнання та приміщення. Розрахунки заносимо до таблиці 4.5. Так як вартість ліцензійної ОС та спеціалізованих ліцензійних нематеріальних ресурсів менше 20000 грн, то даний нематеріальний актив не амортизується, а його вартість включається у вартість розробки повністю, $В_{нем.ак.}$ = 9000 грн.

Таблиця 4.5 – Амортизаційні відрахування на матеріальні та нематеріальні ресурси для розробників

Найменування обладнання	Балансова вартість, грн.	Строк корисного використання, років	Термін використання обладнання, місяців	Амортиза-ційні відрахуван-ня, грн.
Комп'ютер та комп'ютерна периферія	30000	2	1,74	2173,913
Офісне обладнання (меблі)	26000	4	1,74	942,029
Приміщення	1700000	20	1,74	12318,841
Всього				15434,783

4.2.6 Розрахунок тарифів на елетроенергію

Тарифи на електроенергію для непобутових споживачів (промислових підприємств) відрізняються від тарифів на електроенергію для населення. При цьому тарифи на розподіл електроенергії у різних постачальників (енергорозподільних компаній), будуть різними. Крім того, розмір тарифу залежить від класу напруги (1-й або 2-й клас). Тарифи на розподіл електроенергії для всіх енергорозподільних компаній встановлює Національна комісія з регулювання енергетики і комунальних послуг (НКРЕКП). Витрати на силову електроенергію розраховуються за формулою:

$$В_е = В \cdot П \cdot Ф \cdot К_п, \quad (4.5)$$

де В – вартість 1 кВт-години електроенергії для 1 класу підприємства з ПДВ в 2025 році для Вінницької області за даними Minfin.com, В = 12,24 грн./кВт;

П – встановлена потужність обладнання, кВт. П = 0,3 кВт;

Ф – фактична кількість годин роботи обладнання, годин;

$К_п$ – коефіцієнт використання потужності, $К_п$ = 0,9;

$$В_е = 0,9 \cdot 0,3 \cdot 8 \cdot 40 \cdot 12,24 = 1057,536 \text{ (грн.)}$$

4.2.7 Інші витрати та загальновиробничі витрати.

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками. Витрати за статтею «Інші витрати» розраховуються як 50…100% від суми основної заробітної плати дослідників:

$$I_в = (З_о + З_р) \cdot \frac{Н_{ів}}{100\%}, \quad (4.6)$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати».

$$I_в = 135652{,}174 * 65\% / 100\% = 88173{,}913 \text{ (грн.)}$$

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов’язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов’язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов’язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін. Витрати за статтею «Накладні (загальновиробничі) витрати» розраховуються як 100…150% від суми основної заробітної плати дослідників:

$$Н_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \quad (4.7)$$

де $Н_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати».

$$Н_{нзв} = 135652{,}174 * 125\% / 100\ \% = 169565{,}218 \text{ (грн.)}$$

4.2.9 Витрати на проведення науково-дослідної роботи.

Сума всіх попередніх статей витрат дає загальні витрати на проведення науково-дослідної роботи:

$$В_{заг} = 135652{,}174 + 15600{,}00 + 33275{,}48 + 15434{,}783 + 9000 + 1057{,}536 + 88173{,}913 + 169565{,}218 = 467759{,}104 \text{ грн.}$$

4.2.10 Розрахунок загальних витрат на науково-дослідну (науково-технічну) роботу та оформлення її результатів.

Загальні витрати на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів розраховуються за формулою:

$$ЗВ = \frac{В_{заг}}{\eta} \text{ (грн)}, \quad (4.8)$$

де η – коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи.

Так, якщо науково-технічна розробка знаходиться на стадії: науково-дослідних робіт, то η=0,1; технічного проектування, то η=0,2; розробки конструкторської документації, то η=0,3; розробки технологій, то η=0,4; розробки дослідного зразка, то η=0,5; розробки промислового зразка, то η=0,7; впровадження, то η=0,9. Оберемо η = 0,5, так як розробка, на даний момент, знаходиться на стадії дослідного зразка:

$$ЗВ = 467759{,}104 / 0{,}5 = 935518{,}208 \text{ грн.}$$

4.3 Розрахунок економічної ефективності науково-технічної розробки за її можливої комерціалізації потенційним інвестором

В ринкових умовах узагальнювальним позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів тієї чи іншої науково-технічної розробки, є збільшення у потенційного інвестора величини чистого прибутку. Саме зростання чистого прибутку забезпечить потенційному інвестору надходження додаткових коштів, дозволить покращити

фінансові результати його діяльності, підвищить конкурентоспроможність та може позитивно вплинути на ухвалення рішення щодо комерціалізації цієї розробки.

Для того, щоб розрахувати можливе зростання чистого прибутку у потенційного інвестора від можливого впровадження науково-технічної розробки необхідно:

— вказати, з якого часу можуть бути впроваджені результати науково-технічної розробки;

— зазначити, протягом скількох років після впровадження цієї науково-технічної розробки очікуються основні позитивні результати для потенційного інвестора (наприклад, протягом 3-х років після її впровадження);

— кількісно оцінити величину існуючого та майбутнього попиту на цю або аналогічні чи подібні науково-технічні розробки та назвати основних суб'єктів (зацікавлених осіб) цього попиту;

— визначити ціну реалізації на ринку науково-технічних розробок з аналогічними чи подібними функціями.

При розрахунку економічної ефективності потрібно обов'язково враховувати зміну вартості грошей у часі, оскільки від вкладення інвестицій до отримання прибутку минає чимало часу. При оцінюванні ефективності інноваційних проектів передбачається розрахунок таких важливих показників:

— абсолютного економічного ефекту (чистого дисконтованого доходу);

— внутрішньої економічної дохідності (внутрішньої норми дохідності);

— терміну окупності (дисконтованого терміну окупності).

Аналізуючи напрямки проведення науково-технічних розробок, розрахунок економічної ефективності науково-технічної розробки за її можливої комерціалізації потенційним інвестором можна об'єднати, враховуючи визначені ситуації з відповідними умовами.

4.3.1 Розробка інформаційної системи (web-сайт, консолідований ресурс тощо) на основі нових алгоритмів, програмних або технічних засобів.

В цьому випадку основу майбутнього економічного ефекту буде формувати: ΔN – збільшення кількості споживачів, яким надається відповідна інформаційна послуга в аналізовані періоди часу; N– кількість споживачів, яким надавалась відповідна інформаційна послуга у році до впровадження результатів нової науково-технічної розробки; $Ц_б$ – вартість послуги у році до впровадження інформаційної системи; ±ΔЦ – зміна вартості послуги (зростання чи зниження) від впровадження результатів науково-технічної розробки в аналізовані періоди часу.

Для наведеного випадку можливе збільшення чистого прибутку у потенційного інвестора $\Delta П_i$ для кожного із років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховується за формулою:

$$\Delta П_i = (\pm \Delta Ц_0 \cdot N + Ц_0 \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\vartheta}{100}), \quad (4.9)$$

де $\pm\Delta Ц_о$ – зміна вартості програмного продукту (зростання чи зниження) від впровадження результатів науково-технічної розробки в аналізовані періоди часу;

N – кількість споживачів які використовували аналогічний продукт у році до впровадження результатів нової науково-технічної розробки;

$Ц_о$ – основний оціночний показник, який визначає діяльність підприємства у даному році після впровадження результатів наукової розробки, $Ц_о = Ц_б \pm \Delta Ц_о$;

Цб – вартість програмного продукту у році до впровадження результатів розробки;

ΔN – збільшення кількості споживачів продукту, в аналізовані періоди часу, від покращення його певних характеристик;

λ – коефіцієнт, який враховує сплату податку на додану вартість. Ставка податку на додану вартість дорівнює 20%, а коефіцієнт λ = 0,8333.

р – коефіцієнт, який враховує рентабельність продукту, р=0,3

ϑ – ставка податку на прибуток, у 2025 році ϑ =18%.

Припустимо, що при прогнозованій ціні 5000 грн. за одиницю виробу, термін збільшення прибутку складе 3 роки. Після завершення розробки і її вдосконалення, можна буде підняти її ціну на 250 грн. Кількість одиниць реалізованої продукції також збільшиться: протягом першого року – на 4000 шт., протягом другого року – на 3000 шт., протягом третього року на 2000 шт. До моменту впровадження результатів наукової розробки реалізації продукту не було:

$$\Delta П_1 = (0 * 250 + (5000 + 250) * 4000 * 0{,}8333 * 0{,}3) * (1 - 0{,}18) = 4304827{,}8 \text{ грн.}$$

$$\Delta П_2 = (0 * 250 + (5000 + 250) * (4000 + 3000) * 0{,}8333 * 0{,}3) * (1 - 0{,}18) = 7533448{,}65 \text{ грн.}$$

$$\Delta П_3 = (0 * 250 + (5000 + 250) * (4000 + 3000 + 2000) * 0{,}8333 * 0{,}3) * (1 - 0{,}18) = 9685862{,}55 \text{ грн.}$$

Отже, комерційний ефект від реалізації результатів розробки за три роки складе 21524139 грн.

4.3.2 Розрахунок ефективності вкладених інвестицій та періоду їх окупності.

Розраховуємо приведену вартість збільшення всіх чистих прибутків *ПП*, що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки:

$$ПП = \sum_1^T \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.10)$$

де $\Delta П_i$ –збільшення чистого прибутку у кожному із років, протягом яких виявляються результати виконаної та впровадженої науково-дослідної (науково-технічної) роботи, грн;

T – період часу, протягом якою виявляються результати впровадженої науково-дослідної (науково-технічної) роботи, роки;

τ –ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, $\tau = 0{,}05 \ldots 0{,}15$;

t – період часу (в роках).

Збільшення прибутку ми отримаємо, починаючи з першого року:

$$ПП = (4304827{,}8/(1+0{,}1)^1) + (7533448{,}65/(1+0{,}1)^2) + (9685862{,}55/(1+0{,}1)^3) = 3913479.818 + 6225990.62 + 7277131.893 = 17416602.331 \text{ грн}$$

Далі розраховують величину початкових інвестицій *PV*, які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки. Для цього можна використати формулу:

$$PV = k_{інв} * ЗВ, \quad (4.11)$$

де $k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію, це можуть бути витрати на підготовку приміщень, розробку технологій, навчання персоналу, маркетингові заходи тощо; зазвичай $k_{інв}$=2...5, але може бути і більшим;

ЗВ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, грн.

$$PV = 2 * 935518{,}208 = 1871036.416 \text{ грн.}$$

Тоді абсолютний економічний ефект $E_{абс}$ або чистий приведений дохід (NPV, Net Present Value) для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$E_{абс} = ПП - PV, \quad (4.12)$$

$$E_{абс} = 17416602.331 - 1871036.416 = 15545565.915 \text{ грн.}$$

Оскільки $E_{абс} > 0$ то вкладання коштів на виконання та впровадження результатів даної науково-дослідної (науково-технічної) роботи може бути доцільним.

Для остаточного прийняття рішення з цього питання необхідно розра-хувати внутрішню економічну дохідність або показник внутрішньої но-рми дохідності (IRR, Internal Rate of Return) вкладених інвестицій та порівняти її з так званою бар'єрною ставкою дисконтування, яка визначає ту мінімальну внутрішню економічну дохідність, нижче якої інвестиції в будь-яку науково-технічну розробку вкладати буде економічно недоцільно.

Розрахуємо відносну (щорічну) ефективність вкладених в наукову розробку інвестицій $E_в$. Для цього використаємо формулу:

$$E_в = \sqrt[Т_ж]{1 + \frac{E_{абс}}{PV}} - 1, \quad (4.13)$$

де $Т_ж$ −життєвий цикл наукової розробки, роки.

$$E_в = \sqrt[з]{1 + \frac{15545565.915}{1871036.416}} - 1 = 1{,}104$$

Визначимо мінімальну ставку дисконтування, яка у загальному вигляді визначається за формулою:

$$r = d + f \quad (4.14)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2024 році в Україні d = (0,09…0,14);

f−показник, що характеризує ризикованість вкладень; зазвичай, величина f = (0,05...0,5).

$$\tau_{min}.$$

Так як $E_в > \tau_{min}$, то інвестор може бути зацікавлений у фінансуванні даної наукової розробки.

Розрахуємо термін окупності вкладених у реалізацію наукового проекту інвестицій за формулою:

$$T_{ок} = \frac{1}{E_в}, \quad (4.15)$$

$$T_{ок} = 1 / 1{,}104 = 0{,}9 \text{ р.}$$

Оскільки $T_{ок}$ < 3-х років, а саме термін окупності рівний 0.9 роки, то фінансування даної наукової розробки є доцільним.

Висновки до розділу: економічна частина даної роботи містить розрахунок витрат на розробку нового програмного продукту, сума яких складає 935518,208 гривень. Було спрогнозовано орієнтовану величину витрат по кожній з статей витрат. Також розраховано чистий прибуток, який може отримати виробник від реалізації нового технічного рішення, розраховано період окупності витрат для інвестора та економічний ефект при використанні даної розробки. В результаті аналізу розрахунків можна зробити висновок, що розроблений програмний продукт за ціною дешевший за аналог і є висококонкурентоспроможним. Період окупності складе близько 0,9 роки.

ВИСНОВКИ

У процесі виконання магістерської кваліфікаційної роботи було здійснено комплексне дослідження та розробку автоматизованої інформаційної системи для розгортання, супроводу та моніторингу прикладних сервісів із використанням сучасних DevOps-практик. У першому розділі роботи проведено аналіз предметної області, зокрема розглянуто сучасні підходи до побудови хмарної та контейнеризованої інфраструктури, принципи безперервної інтеграції та доставки програмного забезпечення, а також засоби централізованого моніторингу та спостережуваності. На основі аналізу було виявлено основні проблеми традиційного адміністрування сервісів, пов’язані з високими витратами часу, складністю масштабування та підвищеними ризиками помилок ручного керування.

У другому розділі було розроблено архітектурні рішення та алгоритми функціонування системи, обґрунтовано вибір технологічного стеку та засобів реалізації. Зокрема, було обрано контейнерну оркестрацію Kubernetes, інструменти автоматизації інфраструктури Terraform і Ansible, систему безперервної інтеграції Jenkins, а також засоби моніторингу Prometheus і Grafana.

У третьому розділі виконано практичну реалізацію прототипу системи в однонодовому кластері Kubernetes. Було розроблено прикладний API-сервіс, підготовлено необхідні Kubernetes-маніфести, налаштовано процеси CI/CD, а також реалізовано збір і візуалізацію метрик стану кластера та сервісів. Проведене тестування підтвердило коректність роботи системи, зменшення часу розгортання сервісів та підвищення стабільності їх функціонування порівняно з традиційними підходами ручного адміністрування.

У четвертому розділі здійснено економічне обґрунтування розробки, проведено оцінку витрат на створення та впровадження системи, а також розраховано показники економічної ефективності. Отримані результати свідчать про доцільність впровадження запропонованого рішення, зокрема за рахунок скорочення витрат на оплату праці персоналу, зменшення часу виконання операцій розгортання та супроводу сервісів, а також підвищення продуктивності інформаційно-технічної діяльності підприємства.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1. Заяц В.В., Савицька Л.А., Система розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик Конференція ВНТУ: Молодь в науці: дослідження, проблеми, перспективи (МН-2026) [Едктронниц ресурс] — Режим доступу до ресурсу: https://conferences.vntu.edu.ua/index.php/mn/mn2026/paper/view/26836

2. Азаров О. Д. Комп'ютерні мережі / О. Д. Азаров, С. М. Захарченко, О. В. Кадук та ін. — Вінниця : ВНТУ, 2013. — 370 с. ISBN 978-966-641-543-4

3. Азаров О. Д., Захарченко С. М., Яремчук Ю. Є., Дубінін В. М.. Основи роботи та адміністрування мережних операційних систем. Основи роботи та адміністрування мережних операційних систем [Текст] : навчальний посібник з дисципліни "Корпоративні та загальнодоступні мережі" / О. Д. Азаров, С. М. Захарченко, Є. В. Яремчук, В. М. Дубінін. — Вінниця : ВДТУ. — 2001. — 114 с.

4. What is DevOps? | Atlassian [Електронний ресурс] - Режим доступу до ресурсу: https://www.atlassian.com/devops

5. Devops Principles | Atlassian [Електронний ресурс] - Режим доступу до ресурсу: https://www.atlassian.com/devops

6. What is Infrastructure as Code? - IaC Explained - AWS [Електронний ресурс] - Режим доступу до ресурсу: https://aws.amazon.com/what-is/iac/

7. Terraform | HashiCorp Developer [Електронний ресурс] - Режим доступу до ресурсу: https://developer.hashicorp.com/terraform

8. Infrastructure as Code: базові принципи vs інструменти, що еволюціонують [Електронний ресурс] - Режим доступу до ресурсу: https://dou.ua/lenta/articles/infrastructure-as-code/

9. What is Configuration Management? - Software Configuration Management Explained - AWS [Електронний ресурс] - Режим доступу до ресурсу: https://aws.amazon.com/what-is/configuration-management/

10. Ansible Documentation [Електронний ресурс] - Режим доступу до ресурсу: https://docs.ansible.com/

11. Chef vs Puppet vs Ansible: Top DevOps Tools Comparison [Електронний ресурс] - Режим доступу до ресурсу: https://www.veritis.com/blog/chef-vs-puppet-vs-ansible-comparison-of-devops-management-tools/

12. What is Containerization? - Containerization Explained - AWS - Updated 2025 [Електронний ресурс] - Режим доступу до ресурсу: https://aws.amazon.com/what-is/containerization/

13. What is Docker | AWS [Електронний ресурс] - Режим доступу до ресурсу: https://aws.amazon.com/docker/

14. Docker Docs [Електронний ресурс] - Режим доступу до ресурсу: https://docs.docker.com/

15. Orchestration: Automating Data Pipelines | Databricks [Електронний ресурс] - Режим доступу до ресурсу: https://www.databricks.com/glossary/orchestration

16. Cluster Architecture [Електронний ресурс] - Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/architecture/

17. Kubernetes Documentation [Електронний ресурс] - Режим доступу до ресурсу: https://kubernetes.io/docs/home/

18. What is CI/CD? [Електронний ресурс] - Режим доступу до ресурсу: https://about.gitlab.com/topics/ci-cd/

19. Jenkins User Documentation [Електронний ресурс] - Режим доступу до ресурсу: https://www.jenkins.io/doc/

20. Monitoring and Visualization for Release 1.2 [Електронний ресурс] - Режим доступу до ресурсу: https://docs.oracle.com/en/operating-systems/olcne/1.2/monitor/

21. Overview | Prometheus [Електронний ресурс] - Режим доступу до ресурсу: https://prometheus.io/docs/introduction/overview/

22. Technical documentation | Grafana Labs [Електронний ресурс] - Режим доступу до ресурсу: https://grafana.com/docs/

23. What is DevOps? [Електронний ресурс] Режим доступу до ресурсу: https://about.gitlab.com/topics/devops/

24. The History and Evolution of DevOps | Tom Geraghty [Електронний ресурс] Режим доступу до ресурсу: https://tomgeraghty.co.uk/index.php/the-history-and-evolution-of-devops/

25. What is platform engineering? [Електронний ресурс] Режим доступу до ресурсу: https://www.redhat.com/en/topics/devops/platform-engineering

26. Analyzing the DNA of DevOps [Електронний ресурс] Режим доступу до ресурсу: https://opensource.com/article/18/11/analyzing-devops

27. What is GitOps? [Електронний ресурс] Режим доступу до ресурсу: https://www.redhat.com/en/topics/devops/what-is-gitops

28. Continuous Integration: Infrastructure as Code in DevOps [Електронний ресурс] Режим доступу до ресурсу: https://web.archive.org/web/20160206165308/http://info.easydynamics.com/blog/continuous-integration-infrastructure-as-code

29. Moving from Infrastructure Automation to True DevOps [Електронний ресурс] ¬ Режим доступу до ресурсу: http://devops.com/2015/05/14/moving-from-infrastructure-automation-to-true-devops/

30. Borg, Omega, and Kubernetes - ACM Queue DevOps [Електронний ресурс] ¬ Режим доступу до ресурсу: http://queue.acm.org/detail.cfm?id=2898444

31. API Conventions [Електронний ресурс] ¬ Режим доступу до ресурсу: https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#metadata

32. Role Based Access Control Good Practices [Електронний ресурс] ¬ Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/security/rbac-good-practices/

33. Pod Security Standards [Електронний ресурс] ¬ Режим доступу до ресурсу: https://kubernetes.io/docs/concepts/security/pod-security-standards/

34. "Introduction - The Cluster API Book [Електронний ресурс] ¬ Режим доступу до ресурсу: https://cluster-api.sigs.k8s.io/

35. Querying Prometheus [Електронний ресурс] ¬ Режим доступу до ресурсу: https://prometheus.io/docs/prometheus/latest/querying/basics/

ДОДАТОК А

Технічне завдання

Міністерство освіти і науки України
Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ
проф., д.т.н. О.Д. Азаров
«03» 10 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання магістерської кваліфікаційної роботи
Система розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик

Керівник роботи: к.т.н., доц. каф. ОТ:
_____ Савицька Л.А.

Виконав: студент гр. 1КІ-24м
_____Заяц В.В.

ВНТУ 2025

1 Підстава для виконання бакалаврського дипломного проекту (МКР)

1.1 Актуальність теми дослідження полягає в тому, що в умовах стрімкого розвитку цифрових технологій і глобальної інформатизації сучасні організації дедалі частіше стикаються з потребою швидкого, надійного та відтворюваного розгортання прикладних сервісів У цьому контексті набуває особливої важливості впровадження DevOps-практик, які поєднують процеси розробки (Development) та експлуатації (Operations) в єдиний інтегрований цикл безперервної інтеграції, доставки та моніторингу.;

1.2 Наказ про затвердження теми МКР № 313 від 24.09.2025 року

2 Мета і призначення МКР

2.1 Метою роботи є автоматизація системи розгортання та моніторингу прикладного сервісів із інтеграцією DevOps-практик;

2.2 Призначення розробки — створення та впровадження автоматизованої інфраструктурної системи для розгортання, безперервної інтеграції, доставки та моніторингу прикладного API-сервісу з використанням контейнерних технологій та DevOps-підходів.

3 Вихідні дані для виконання МКР

Вихідні дані: виділений вузол, операційна система Linux, API- сервіс для розгортання, Kubernetes, Docker-артефакт, Grafana для візуалізації даних..

4 Технічні вимоги до виконання МКР

Технічні вимоги:

— аналіз сучасних методів розгортання прикладних сервісів;

— пошук та аналіз технологій для розгортання прикладних сервісів з інтеграцією DevOps практик;

— створення прикладного API-сервісу для розгортання;

— налаштування системи розгортання прикладного API- сервсіу;

— тестування працездатності системи.

5 Етапи МКР та очікувані результати

Етапи роботи та очікувані результати приведено в Таблиці А.1

Таблиця А.1 — Етапи МКР

№ з/п	Назва етапів дипломної роботи	Термін виконання		Очікувані результати
		початок	закінчення	
1	Постановка задачі роботи	26.09.2025	26.09.2025	Задачі дослідження
2	Аналіз предметної області	27.09.2025	01.10.2025	Розділ 1
3	Проєктування системи розгортання прикладних сервісів	02.10.2025	12.10.2025	Розділ 2
4	Реалізація системи розгортання та моніторингу прикладного сервісу	13.10.2025	27.10.2025	Розділ 3
5	Розрахунок економічної частини	29.11.2025	29.11.2025	Розділ 4
6	Оформлення матеріалів до захисту МКР	01.11.2025	01.11.2025	ПЗ, графічний матеріал і презентація
7	Перевірка якості виконання магістерської роботи	04.11.2025	04.11.2025	Оформлені документи
9	Підписи супроводжувальних документів у нормконтролера, керівника, опонента	07.11.2025	07.11.2025	Оформлені документи
10	Перевірка на антиплагіат та ШІ	10.11.2025	10.11.2025	Оформлені документи

6 Матеріали, що подаються до захисту МКР

До захисту МКР подається: пояснювальна записка МКР, графічні та ілюстративні матеріали, протокол попереднього захисту роботи на кафедрі, відзив наукового керівника, рецензія рецензента, анотації до МДП українською та іноземною мовами, нормоконтроль про відповідність оформлення МКР діючим вимогам.

7 Порядок контролю виконання та захисту МКР

Виконання етапів графічної та розрахункової документації МКР контролюється науковим керівником згідно зі встановленими термінами. Захист МКР відбувається на засіданні Державної екзаменаційної комісії, затвердженою

наказом ректора.

8 Вимоги до оформлення МКР

8.1 При оформлювання МКР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— міждержавний ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

— методичні вказівки до виконання магістерських кваліфікаційних робіт зі спеціальності 123 — «Комп’ютерна інженерія»;

— документами на які посилаються у вище вказаних.

8.2 Порядок виконання МКР викладено в «Положення про кваліфікаційні роботи на другому (магістерському) рівні вищої освіти СУЯ ВНТУ–03.02.02 П.001.01:21.

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ КВАЛІФІКАЦІЙНОЇ РОБОТИ НА НАЯВНІСТЬ ТЕКСТОВИХ ЗАПОЗИЧЕНЬ

Назва роботи: Система розгортання та моніторингу прикладних сервісів із інтеграцією DevOps-практик

Тип роботи: магістерська кваліфікаційна робота

(бакалаврська кваліфікаційна робота / магістерська кваліфікаційна робота)

Підрозділ: Факультет інформаційних технологій та комп'ютерної інженерії, кафедра обчислювальної техніки, група 1КІ-24м

(кафедра, факультет, навчальна група)

Коефіцієнт подібності текстових запозичень, виявлених у роботі системою StrikePlagiarism (КП1) 1 %

Висновок щодо перевірки кваліфікаційної роботи (відмітити потрібне)

☐ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак академічного плагіату, фабрикації, фальсифікації. Роботу прийняти до захисту.

☐ У роботі не виявлено ознак плагіату, фабрикації, фальсифікації, але надмірна кількість текстових запозичень та/або наявність типових розрахунків не дозволяють прийняти рішення про оригінальність та самостійність її виконання. Роботу направити на доопрацювання.

☐ У роботі виявлено ознаки академічного плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень. Робота до захисту не приймається.

Експертна комісія:

_____ _____

(прізвище, ініціали, посада) (підпис)

_____ _____

(прізвище, ініціали, посада) (підпис)

Особа, відповідальна за перевірку _____ Захарченко С.М.

(підпис) (прізвище, ініціали)

З висновком експертної комісії ознайомлений(-на)

Керівник _____ к.т.н., доц. каф. ОТ Савицька Л. А.

(підпис) (прізвище, ініціали, посада)

Здобувач _____ Заяц В.В.

(підпис) (прізвище, ініціали)

ДОДАТОК В

Блок-схема алгоритму роботи сервісу

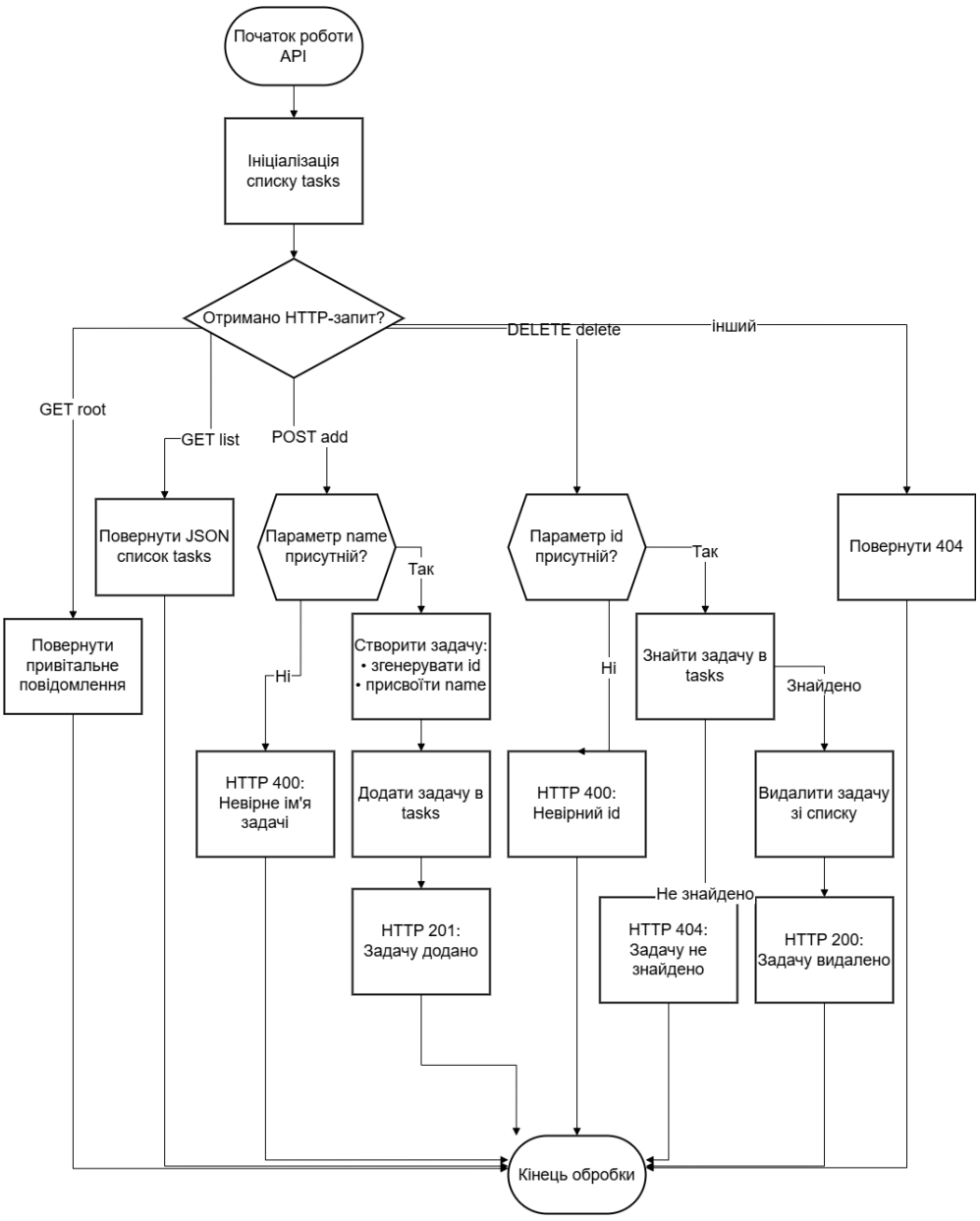

Рисунок В.1 — Блок-схема прикладного API-сервісу

ДОДАТОК Г

Структурна схема розгортання прикладного сервісу

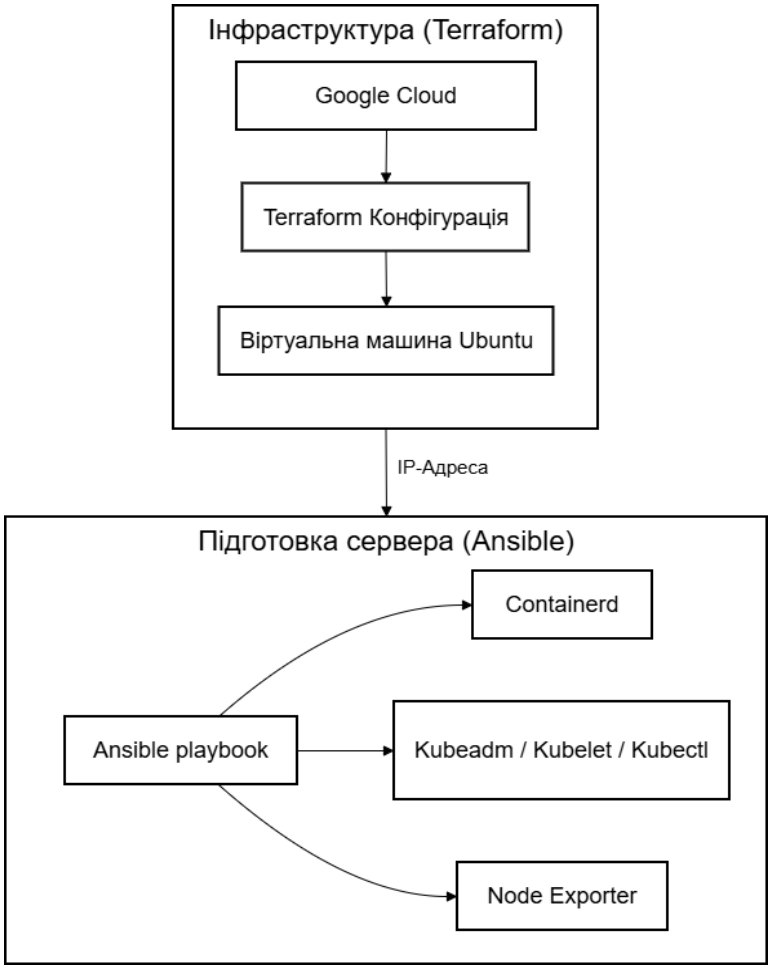

Рисунок Г.1 — Структурна схема розгортання та підготовки серверної інфраструктури

ДОДАТОК Д

Структурна схема кластеру

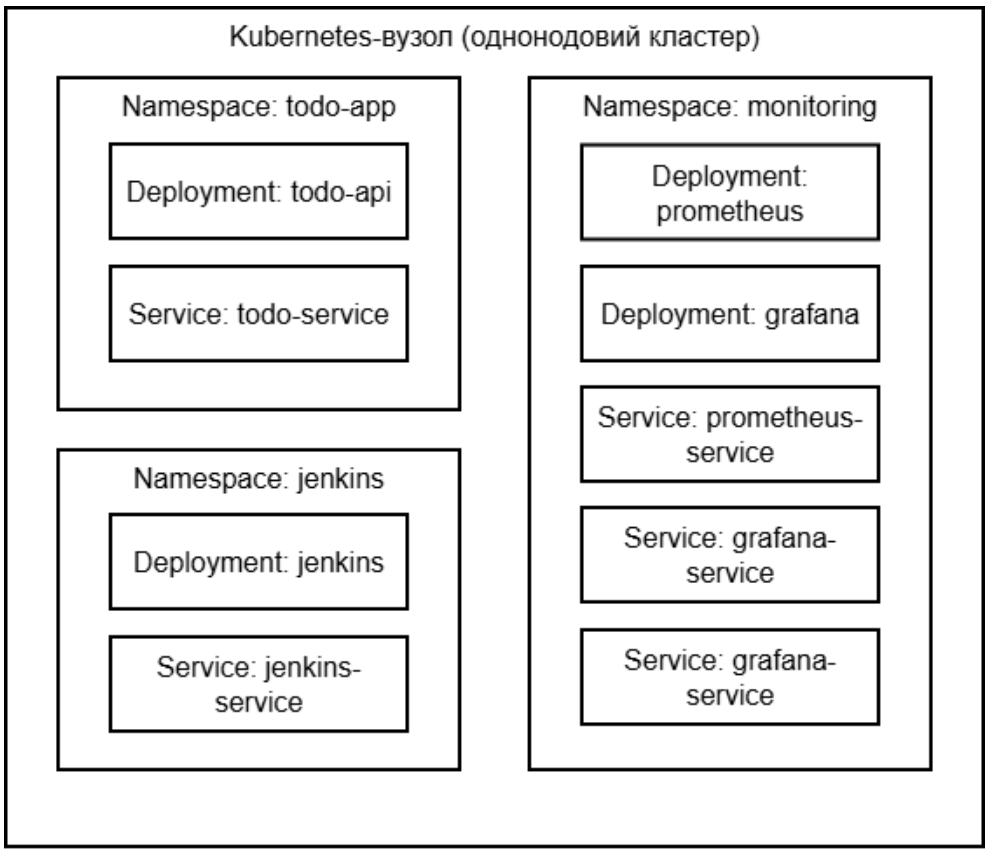

Рисунок Д.1 — Структурна схема кластеру

ДОДАТОК Е

Лістинг API-сервісу

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "os"
    "os/signal"
    "strconv"
    "sync"
    "syscall"
    "time"

    "github.com/prometheus/client_golang/prometheus"
    "github.com/prometheus/client_golang/prometheus/promhttp"
)

// Task — базова структура задачі
type Task struct {
    ID   int    `json:"id"`
    Name string `json:"name"`
    Done bool   `json:"done"`
}

// TaskManager — потокобезпечне сховище задач
type TaskManager struct {
    sync.RWMutex
    tasks []Task
}

// Глобальний менеджер задач
var manager = &TaskManager{tasks: make([]Task, 0)}

// Метрики Prometheus
var (
    totalRequests = prometheus.NewCounterVec(
        prometheus.CounterOpts{
            Name: "todo_api_http_requests_total",
            Help: "Загальна кількість HTTP-запитів за ендпоінтом і методом.",
        },
```

```
        []string{"path", "method"},
    )
)

func init() {
    prometheus.MustRegister(totalRequests)
}

// Health check
func healthHandler(w http.ResponseWriter, r *http.Request) {
    totalRequests.WithLabelValues("/healthz", r.Method).Inc()
    w.WriteHeader(http.StatusOK)
    w.Write([]byte("OK"))
}

// Повертає всі задачі
func listTasks(w http.ResponseWriter, r *http.Request) {
    totalRequests.WithLabelValues("/tasks", r.Method).Inc()
    manager.RLock()
    defer manager.RUnlock()

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusOK)
    json.NewEncoder(w).Encode(manager.tasks)
}

// Додає нову задачу
func addTask(w http.ResponseWriter, r *http.Request) {
    totalRequests.WithLabelValues("/tasks", r.Method).Inc()

    if r.Method != http.MethodPost {
        http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
        return
    }

    var newTask Task
    if err := json.NewDecoder(r.Body).Decode(&newTask); err != nil {
        http.Error(w, "Invalid JSON format", http.StatusBadRequest)
        return
    }
    defer r.Body.Close()

    if newTask.Name == "" {
        http.Error(w, "Task name cannot be empty", http.StatusBadRequest)
        return
```

```
    }

    manager.Lock()
    newTask.ID = len(manager.tasks) + 1
    manager.tasks = append(manager.tasks, newTask)
    manager.Unlock()

    w.Header().Set("Content-Type", "application/json")
    w.WriteHeader(http.StatusCreated)
    json.NewEncoder(w).Encode(newTask)
}

// Видаляє задачу за ID
func deleteTask(w http.ResponseWriter, r *http.Request) {
    totalRequests.WithLabelValues("/tasks", r.Method).Inc()

    if r.Method != http.MethodDelete {
        http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
        return
    }

    idStr := r.URL.Query().Get("id")
    if idStr == "" {
        http.Error(w, "Missing task ID", http.StatusBadRequest)
        return
    }

    id, err := strconv.Atoi(idStr)
    if err != nil || id <= 0 {
        http.Error(w, "Invalid task ID", http.StatusBadRequest)
        return
    }

    manager.Lock()
    defer manager.Unlock()

    for i, task := range manager.tasks {
        if task.ID == id {
            manager.tasks = append(manager.tasks[:i], manager.tasks[i+1:]...)
            w.WriteHeader(http.StatusOK)
            json.NewEncoder(w).Encode(map[string]string{"message": "Task deleted
successfully"})
            return
        }
    }
```

```
    http.Error(w, "Task not found", http.StatusNotFound)
}

// Основна точка входу
func main() {
    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    mux := http.NewServeMux()
    mux.HandleFunc("/healthz", healthHandler)
    mux.HandleFunc("/tasks", func(w http.ResponseWriter, r *http.Request) {
        switch r.Method {
        case http.MethodGet:
            listTasks(w, r)
        case http.MethodPost:
            addTask(w, r)
        case http.MethodDelete:
            deleteTask(w, r)
        default:
            http.Error(w, "Method Not Allowed", http.StatusMethodNotAllowed)
        }
    })
    mux.Handle("/metrics", promhttp.Handler())

    server := &http.Server{
        Addr:         ":" + port,
        Handler:      mux,
        ReadTimeout:  10 * time.Second,
        WriteTimeout: 10 * time.Second,
        IdleTimeout:  60 * time.Second,
    }

    // Graceful shutdown
    stop := make(chan os.Signal, 1)
    signal.Notify(stop, syscall.SIGINT, syscall.SIGTERM)

    go func() {
        log.Printf("ToDo API server is running on port %s\n", port)
        if err := server.ListenAndServe(); err != nil && err != http.ErrServerClosed {
            log.Fatalf("Server error: %v", err)
        }
    }()
```

```
    <-stop
    log.Println("Shutting down server gracefully...")

    ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
    defer cancel()

    if err := server.Shutdown(ctx); err != nil {
        log.Fatalf("Server shutdown failed: %v", err)
    }

    log.Println("Server stopped successfully.")
}
```

ДОДАТОК Є

Лістинг Ansible плейбуку

```
- name: Kubernetes Single-Node Cluster Setup
  hosts: all
  become: true
  vars:
    node_exporter_version: "1.7.0"
    kubernetes_version: "1.30"
    calico_manifest_url:
"https://raw.githubusercontent.com/projectcalico/calico/v3.27.0/manifests/calico.yaml"

  tasks:

    # ----------------------- SYSTEM PREPARATION -----------------------
    - name: Update apt cache
      ansible.builtin.apt:
        update_cache: yes

    - name: Upgrade packages
      ansible.builtin.apt:
        upgrade: dist

    - name: Install required packages
      ansible.builtin.apt:
        pkg:
          - apt-transport-https
          - ca-certificates
          - curl
          - gnupg
          - lsb-release
        state: present

    # ----------------------- DOCKER INSTALLATION -----------------------
    - name: Add Docker GPG key
      ansible.builtin.shell: |
        curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/usr/share/keyrings/docker.gpg

    - name: Add Docker repository
      ansible.builtin.apt_repository:
        repo: "deb [arch=amd64 signed-by=/usr/share/keyrings/docker.gpg]
https://download.docker.com/linux/ubuntu jammy stable"

    - name: Install Docker Engine
```

```
    ansible.builtin.apt:
      name:
        - docker-ce
        - docker-ce-cli
        - containerd.io
      state: present

  - name: Enable and start Docker
    ansible.builtin.systemd:
      name: docker
      enabled: yes
      state: started

  # ----------------------- KUBERNETES INSTALLATION -----------------------
  - name: Add Kubernetes signing key
    ansible.builtin.shell: |
      curl -fsSL https://pkgs.k8s.io/core:/stable:/v{{ kubernetes_version
}}/deb/Release.key | \
      gpg --dearmor -o /etc/apt/keyrings/kubernetes.gpg

  - name: Add Kubernetes APT repository
    ansible.builtin.copy:
      dest: /etc/apt/sources.list.d/kubernetes.list
      content: |
        deb [signed-by=/etc/apt/keyrings/kubernetes.gpg]
https://pkgs.k8s.io/core:/stable:/v{{ kubernetes_version }}/deb/ /

  - name: Update apt cache after adding Kubernetes repo
    ansible.builtin.apt:
      update_cache: yes

  - name: Install Kubernetes components
    ansible.builtin.apt:
      pkg:
        - kubelet
        - kubeadm
        - kubectl
      state: present

  - name: Disable swap
    ansible.builtin.command: swapoff -a

  - name: Ensure swap is permanently disabled
    ansible.builtin.replace:
      path: /etc/fstab
```

```
      regexp: '^(.+ swap .+)$'
      replace: '# \1'

  # ----------------------- CLUSTER INITIALIZATION -----------------------
  - name: Pull required Kubernetes images
    ansible.builtin.shell: kubeadm config images pull

  - name: Initialize Kubernetes cluster
    ansible.builtin.shell: |
      kubeadm init --pod-network-cidr=192.168.0.0/16
    args:
      creates: /etc/kubernetes/admin.conf

  - name: Create .kube directory
    ansible.builtin.file:
      path: /root/.kube
      state: directory
      mode: 0755

  - name: Copy Kubernetes admin config
    ansible.builtin.copy:
      src: /etc/kubernetes/admin.conf
      dest: /root/.kube/config
      remote_src: yes

  - name: Allow scheduling pods on control-plane (single-node mode)
    ansible.builtin.shell: |
      kubectl taint nodes --all node-role.kubernetes.io/control-plane-
    ignore_errors: true

  # ----------------------- CNI NETWORK (CALICO) -----------------------
  - name: Install Calico CNI
    ansible.builtin.shell: kubectl apply -f {{ calico_manifest_url }}

  # ----------------------- NODE EXPORTER INSTALLATION -----------------------
  - name: Download Node Exporter
    ansible.builtin.get_url:
      url: "https://github.com/prometheus/node_exporter/releases/download/v{{
node_exporter_version }}/node_exporter-{{ node_exporter_version }}.linux-
amd64.tar.gz"
      dest: /tmp/node_exporter.tar.gz

  - name: Extract Node Exporter
    ansible.builtin.unarchive:
      src: /tmp/node_exporter.tar.gz
```

```
      dest: /tmp/
      remote_src: yes

  - name: Move Node Exporter binary
    ansible.builtin.copy:
      src: "/tmp/node_exporter-{{ node_exporter_version }}.linux-
amd64/node_exporter"
      dest: /usr/local/bin/node_exporter
      mode: '0755'

  - name: Create node_exporter system user
    ansible.builtin.user:
      name: node_exporter
      shell: /usr/sbin/nologin

  - name: Create systemd service for Node Exporter
    ansible.builtin.copy:
      dest: /etc/systemd/system/node_exporter.service
      content: |
        [Unit]
        Description=Node Exporter
        After=network.target

        [Service]
        User=node_exporter
        ExecStart=/usr/local/bin/node_exporter

        [Install]
        WantedBy=default.target

  - name: Enable Node Exporter
    ansible.builtin.systemd:
      name: node_exporter
      daemon_reload: yes
      state: started
      enabled: yes
```