

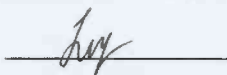
Міністерство освіти і науки України
Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки

ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи бакалавра
на тему:

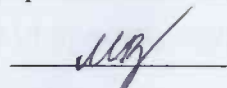
«Проект on-premise інфраструктури системи керування аудитами із
застосуванням методологій DevOps, SRE та Platform Engineering»

Виконав: студент 4-го курсу групи КІ-22-1
спеціальності 123 «Комп'ютерна інженерія»



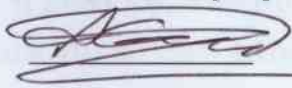
Назарій ТИШКЕВИЧ

Керівник: старший викладач кафедри КІ та КБ



Вадим МИКОЛАЙЧУК

Рецензент: завідувач кафедри КІ та КБ, к.т.н., доцент



Андрій ЄФІМЕНКО

Державний університет «Житомирська політехніка»
Факультет інформаційно-комп'ютерних технологій
Кафедра комп'ютерної інженерії та кібербезпеки
Спеціальність 123 «Комп'ютерна інженерія»
Освітня програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ

Завідувач

кафедри комп'ютерної

інженерії та кібербезпеки

 Андрій Єфіменко

2 лютого 2026 р.

ЗАВДАННЯ

на випускню кваліфікаційну роботу бакалавра

Студент: Тишкевич Назарій Юрійович

Тема роботи: «Проект on-premise інфраструктури системи керування аудитами
із застосуванням методологій DevOps, SRE та Platform Engineering»

затверджена Наказом університету від 2 лютого 2026 р. № 97/с

Термін здачі студентом закінченої роботи 26 червня 2026 р.

Вихідні дані роботи:

- процеси проєктування, розгортання та забезпечення надійної роботи локальної (on-premise) інфраструктури для функціонування системи керування аудитами;
- принципи та методології DevOps, Site Reliability Engineering (SRE) та Platform Engineering, а також технології автоматизації інфраструктури (IaC), контейнеризації, оркестрації та підходи GitOps до неперервного розгортання.

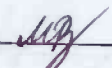
Обладнання та програмне забезпечення:

1. Proxmox VE 8.6
2. Terraform 1.15.5
3. Ansible 2.9
4. RKE2 v1.36.1
5. Rancher 2.13
6. MariaDB Galera Cluster 4 10.6
7. ArgoCD 3.4.4
8. Helm 4.2.1
9. Django 6.0.6
10. Bootstrap 5.3
11. Docker 29.5.3

Консультанти з випускної кваліфікаційної роботи із зазначенням розділів, що їх стосуються

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
1	Головня О. С.	14.03.2026	 14.03.2026
2	Головня О. С.	07.04.2026	 07.04.2026
3	Головня О. С.	27.05.2026	 27.05.2026

Студент  Назарій ТИШКЕВИЧ

Керівник  Вадим МИКОЛАЙЧУК

Календарний план

№ з/п	Назви етапів випускної кваліфікаційної роботи	Термін виконання етапів роботи	Примітка
1	Постановка завдання	02.05.2026	Виконано
2	Аналіз можливостей	05.05.2026	Виконано
3	Збір літератури	08.05.2026	Виконано
4	Аналіз архітектури систем Інфраструктури як Коду	11.05.2026	Виконано
5	Аналіз рішень для платформи віртуалізації.	22.05.2026	Виконано
6	Аналіз рішень контейнеризації	25.05.2026	Виконано
7	Проектування автоматизації для налаштування платформи контейнеризації	29.05.2026	Виконано
8	Реалізація механізмів спостережуваності	04.06.2026	Виконано
9	Тестування, оцінка ефективності та підготовка висновків	15.06.2026	Виконано
10	Захист кваліфікаційної роботи	26.06.2026	Виконано

Студент Назарій Назарій ТИШКЕВИЧ
 Керівник Вадим Вадим МИКОЛАЙЧУК

АНОТАЦІЯ

Кваліфікаційна робота присвячена проєктуванню та реалізації on-premise інфраструктури для системи керування аудитами із застосуванням методологій DevOps, SRE та Platform Engineering. Пояснювальна записка викладена на 110 сторінках і містить 42 рисунки, 23 таблиці та 20 найменувань у списку використаних джерел, розміщеному на 2 сторінках.

Метою кваліфікаційної роботи є проєктування та реалізація відтворюваної, автоматизованої та відмовостійкої on-premise інфраструктури для надійної експлуатації системи керування аудитами на основі сучасних методологій управління інфраструктурою.

У роботі визначено основні вимоги до on-premise інфраструктури — надійність, доступність та контроль над даними; проаналізовано еволюцію підходів до системного адміністрування від класичної моделі до методологій DevOps, SRE та Platform Engineering; розглянуто особливості захищеного дистрибутиву Kubernetes RKE2 та відмовостійкого кластера баз даних Galera; обґрунтовано технологічний стек та спроектовано багаторівневу архітектуру інфраструктури з відмовостійкістю на кожному критичному рівні; показано розробку автоматизованого розгортання інфраструктури засобами Terraform та Ansible, а також моделі самообслуговування на основі підходу GitOps; наведено результати перевірки відмовостійкості та ефективності реалізованої інфраструктури.

Ключові слова: ON-PREMISE ІНФРАСТРУКТУРА, DEVOPS, SRE, PLATFORM ENGINEERING, ІНФРАСТРУКТУРА ЯК КОД, KUBERNETES, RKE2, GALERA, TERRAFORM, ANSIBLE, ВІДМОВОСТІЙКІСТЬ.

					ККІКБ.КР.Б – 123 – 15 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Проект on-premise інфраструктури системи керування аудитами із застосуванням методологій DevOps, SRE та Platform Engineering	Лім.	Арк.	Аркуші
Розроб.		Тишкевич. Н. Ю.		26.06.26				
Керівник		Миколайчук В. В.		26.06.26			5	119
Реценз.		Єфіменко А.		26.06.26		ФІКТ, Гр. КІ-22-1		
Н. контр.		Миколайчук В. В.		26.06.26				
Затверд.		Єфіменко А.		26.06.26				

ANNOTATION

The qualification work is devoted to the design and implementation of an on-premise infrastructure for an audit management system using DevOps, SRE, and Platform Engineering methodologies. The explanatory note consists of 110 pages, contains 42 images, 23 tables, and 20 references listed on 2 pages.

The aim of the qualification work is to design and implement a reproducible, automated, and fault-tolerant on-premise infrastructure for the reliable operation of an audit management system based on modern infrastructure management methodologies.

The paper defines the main requirements for on-premise infrastructure — reliability, availability, and data control. It analyzes the evolution of system administration approaches from the classical model to DevOps, SRE, and Platform Engineering methodologies. The study considers the features of the secure Kubernetes RKE2 distribution and the fault-tolerant Galera database cluster. The technology stack is justified, and a multi-tier infrastructure architecture with fault tolerance at every critical level is designed. Furthermore, the work demonstrates the development of automated infrastructure deployment using Terraform and Ansible, alongside a self-service model based on the GitOps approach. Finally, the results of testing the fault tolerance and efficiency of the implemented infrastructure are presented.

Keywords: ON-PREMISE INFRASTRUCTURE, DEVOPS, SRE, PLATFORM ENGINEERING, INFRASTRUCTURE AS CODE, KUBERNETES, RKE2, GALERA, TERRAFORM, ANSIBLE, FAULT TOLERANCE.

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				6
Змн.	Арк.	№ докум.	Підпис	Дата		

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	8
ВСТУП.....	10
РОЗДІЛ 1. АНАЛІЗ ПІДХОДІВ ДО ПОБУДОВИ ІНФРАСТРУКТУРИ.....	11
1.1 Вимоги до сучасної On-premise інфраструктури.....	11
1.2 Особливості класичних методів адміністрування.....	15
1.3 Передумови переходу до DevOps	19
1.4 Site Reliability Engineering як інженерний підхід до надійності	24
1.5 Platform Engineering як еволюція операційної моделі	28
РОЗДІЛ 2. ПРОЄКТУВАННЯ ON-PREMISE ІНФРАСТРУКТУРИ	40
2.1 Загальна архітектура інфраструктури та вимоги до неї	40
2.2 Проєктування середовища віртуалізації на Proxmox VE	44
2.3 Проєктування підсистеми ІаС та конфігураційного менеджменту	48
2.4 Проєктування кластера оркестрації RKE2	54
2.5 Проєктування відмовостійкого рівня даних на основі Galera.....	57
2.6 Проєктування підсистеми доставки навантаження	61
2.7 Проєктування підсистеми спостережуваності.....	66
2.8 Проєктування моделі самообслуговування	70
РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ПЕРЕВІРКА ІНФРАСТРУКТУРИ	76
3.1 Розгортання базового середовища віртуалізації.....	76
3.2 Розгортання відмовостійкого кластера Galera.....	79
3.3 Розгортання кластера оркестрації RKE2	85
3.4 Масштабування кластера засобами інфраструктури як код	90
3.5 Розгортання балансувальника навантаження	93
3.6 Контейнеризація та розгортання цільового навантаження	96
3.7 Впровадження підсистеми спостережуваності.....	99
3.8 Тестування відмовостійкості та оцінка ефективності інфраструктури	102
ВИСНОВКИ.....	108
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	109
ДОДАТКИ.....	111

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

AGPLv3 — GNU Affero General Public License version 3

API — Application Programming Interface

CapEx — Capital Expenditures

CI/CD — Continuous Integration / Continuous Delivery

CIS — Center for Internet Security

CNI — Container Network Interface

CSV — Cluster Shared Volume

DC — Cluster Shared Volumes

DevOps — Development and Operations

DNS — Domain Name System

DRS — Distributed Resource Scheduler

ESXi — Elastic Sky X Integrated

FIPS — Federal Information Processing Standard

GDPR — General Data Protection Regulation

GNU — GNU's Not Unix

HA — High Availability

HCL — HashiCorp Configuration Language

HTTP — HyperText Transfer Protocol

HTTPS — HyperText Transfer Protocol Secure

IaC — Infrastructure as Code

IDP — Internal Developer Platform

IEC — International Electrotechnical Commission

iSCSI — Internet Small Computer Systems Interface

ISO — International Organization of Standardization

IST — Incremental State Transfer

KVM — Kernel-based Virtual Machine

LVM — Logical Volume Manager

LXC — Linux Containers

MTTR — Mean Time To Recovery

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				8
Змн.	Арк.	№ докум.	Підпис	Дата		

MVP — Minimum Viable Product
 NFS — Network File System
 OpEx — Operational Expenditure
 QEMU — Quick Emulator
 RBAC — Role-Based Access Control
 REST — Representational State Transfer
 RKE2 — Rancher Kubernetes Engine 2
 SELinux — Security-Enhanced Linux
 SLA — Service Level Agreement
 SLI — Service Level Indicator
 SLO — Service Level Objective
 SMB — Server Message Block
 SRE — Site Reliability Engineering
 SSH — Secure Shell
 SST — State Snapshot Transfer
 TCO — Total Cost of Ownership
 TCP — Transmission Control Protocol
 TLS — Transport Layer Security
 TVP — Thinnest Viable Platform
 UDP — User Datagram Protocol
 VE — Virtual Environment
 VIP — Virtual IP
 VM — Virtual Machine
 VMFS — Virtual Machine File System
 VRRP — Virtual Router Redundancy Protocol
 vSAN — Virtual Storage Area Network
 WMI — Windows Management Instrumentation
 XFS — Xtreme File System
 YAML — YAML Ain't Markup Language
 ZFS — Zettabyte File System

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				9
Змн.	Арк.	№ докум.	Підпис	Дата		

ВСТУП

Актуальність теми. Сучасні інформаційні системи, що оперують чутливими та регульованими даними, зокрема системи керування аудитами, висувають підвищені вимоги до інфраструктури — надійності, доступності та контролю над даними. Водночас класичні підходи до системного адміністрування, засновані на ручному конфігуруванні не здатні забезпечити належний рівень відтворюваності, масштабованості та керованості інфраструктури, що призводить до дрейфу конфігурації, крихкості систем і високої залежності від ручної зміни. Застосування сучасних методологій управління інфраструктурою — DevOps, Site Reliability Engineering та Platform Engineering — дає змогу подолати ці обмеження, проте їх комплексне впровадження саме в умовах on-premise розгортання, де організація зберігає повний контроль над даними, є нетривіальним інженерним завданням. Це й зумовлює актуальність теми, у межах якої вирішується проблема побудови відтворюваної, автоматизованої та відмовостійкої on-premise інфраструктури для надійної експлуатації застосунків зі станом.

Метою кваліфікаційної роботи є проектування та реалізація відтворюваної, автоматизованої та відмовостійкої on-premise інфраструктури для системи керування аудитами на основі методологій DevOps, SRE та Platform Engineering.

Об'єктом дослідження є процес побудови та експлуатації on-premise інфраструктури для інформаційних систем зі станом.

Предметом дослідження є методи; інструменти та архітектурні рішення, з використанням методології DevOps, SRE та Platform Engineering і засоби їх реалізації — інфраструктура як код (Terraform, Ansible), оркестрація контейнерів (RKE2), відмовостійке зберігання даних (Galera) та спостережуваність (Prometheus, Grafana, Loki).

У процесі виконання кваліфікаційної роботи було розроблено проєкт on-premise інфраструктури для системи керування аудитами

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				10
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1. АНАЛІЗ ПІДХОДІВ ДО ПОБУДОВИ ІНФРАСТРУКТУРИ

1.1 Вимоги до сучасної On-premise інфраструктури

Сучасні інформаційні системи, що оперують чутливими або регульованими даними, висувають підвищені вимоги до інфраструктури, на якій вони функціонують. Система керування аудитами належить саме до такого класу застосунків: вона накопичує відомості про перевірки, виявлені невідповідності, докази та результати їх усунення — тобто дані, цілісність і конфіденційність яких безпосередньо впливають на довіру до результатів аудиту. З огляду на це вибір моделі розміщення інфраструктури та формування вимог до неї є відправною точкою проєктування.

On-premise (локальна) модель передбачає розміщення обчислювальних ресурсів, систем зберігання та програмного забезпечення в межах власної фізичної інфраструктури організації, на противагу хмарним моделям, де ці ресурси орендуються у зовнішнього постачальника. Ключова відмінність полягає не лише у фізичному розташуванні обладнання, а у тому, хто володіє рівнем керування (control plane): за локальної моделі організація повністю контролює інфраструктуру, дані та політики доступу, тоді як у хмарі частина відповідальності та контролю залишається за провайдером [3].

Така модель має зворотний бік: організація самостійно несе відповідальність за розгортання, обслуговування, оновлення та безпеку всіх компонентів, а також за капітальні витрати на обладнання, і поступається хмарі в еластичності масштабування. Проте саме повнота контролю робить on-premise підхід доцільним для систем, де вимоги до збереження даних усередині периметра організації переважають над перевагами хмарної гнучкості.

Надійність як характеристику якості програмних систем формалізує міжнародний стандарт ISO/IEC 25010, що відносить до неї чотири підхарактеристики: зрілість (відсутність відмов за штатних умов), доступність, відмовостійкість та відновлюваність. У межах цієї моделі доступність визначає, наскільки система є працездатною та досяжною в момент, коли вона потрібна

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				11
Змн.	Арк.	№ докум.	Підпис	Дата		

користувачеві, і на практиці виражається через угоди про рівень обслуговування (SLA) як частка часу безвідмовної роботи за певний період [1].

Інженерний погляд на надійність розвиває методологія Site Reliability Engineering, у межах якої надійність розглядається як найважливіша властивість системи, адже непрацездатний сервіс не приносить користі незалежно від його функціональних можливостей. SRE пропонує керувати надійністю кількісно — через цільові показники та механізм «бюджету помилок», що дає змогу свідомо балансувати між темпом упровадження змін і стабільністю експлуатації [2]. Для системи керування аудитами ці вимоги мають прикладне значення: простій у період проведення перевірки блокує роботу аудиторів, а втрата накопичених даних означає втрату доказової бази, тому підхарактеристики відмовостійкості та відновлюваності набувають критичної ваги [3].

On-premise модель забезпечує найвищий рівень контролю над даними: організація точно знає, де фізично зберігаються дані, і самостійно визначає, хто має до них доступ. Це безпосередньо пов'язано з поняттям суверенітету даних і спрощує дотримання регуляторних вимог, зокрема GDPR, оскільки дані не залишають периметр організації. Додатковими перевагами локального розміщення є простіше ведення та збереження журналів аудиту (audit trails), можливість роботи в ізольованому (air-gapped) середовищі та незалежність від доступності зовнішнього постачальника [4].

Для системи керування аудитами ця вимога є подвійно значущою: інструмент, призначений для забезпечення прозорості та підзвітності, сам повинен функціонувати в середовищі з контрольованим доступом і повним володінням даними, інакше довіра до його результатів знижується. Таким чином, сформульовані вимоги — надійність, доступність і контроль над даними — утворюють критерії, відповідно до яких у подальших підрозділах аналізуються методологічні підходи (DevOps, SRE та Platform Engineering) і обґрунтовується вибір технологічного стеку.

Сформульовані вище критерії доцільно деталізувати, оскільки самі лише категорії «надійність», «доступність» і «контроль над даними» є надто загальними

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				12
Змн.	Арк.	№ докум.	Підпис	Дата		

для безпосереднього використання як технічного завдання. На практиці вимоги до інфраструктури поділяють на дві групи — функціональні, що визначають, які можливості повинна надавати система, та нефункціональні, що визначають якісні характеристики її роботи. Для системи керування аудитами функціональні вимоги є відносно типовими для вебзастосунку корпоративного класу (автентифікація користувачів, зберігання структурованих та неструктурованих даних, формування звітності), тоді як саме нефункціональні вимоги — до відмовостійкості, спостережуваності, безпеки та відтворюваності розгортання — визначають архітектурні рішення, що розглядаються в цій роботі.

Окремої уваги потребує економічний вимір вибору моделі розміщення. Витрати на інфраструктуру традиційно поділяють на капітальні (CapEX) — одноразові витрати на придатне для постійної служби обладнання, та операційні (OpEX) — поточні витрати на електроенергію, обслуговування й персонал. Хмарна модель переносить більшість витрат у категорію OpEX, перетворюючи їх на регулярні платежі, пропорційні фактичному використанню ресурсів, тоді як on-premise модель вимагає істотних капітальних інвестицій на початковому етапі, які надалі амортизуються протягом терміну служби обладнання. Для організації, що планує тривалу та передбачувану експлуатацію системи зі стабільним, наперед відомим профілем навантаження — а саме таким є типовий профіль використання системи керування аудитами, що не має різких сезонних піків, властивих, наприклад, роздрібній торгівлі — модель капітальних витрат з подальшою низькою вартістю володіння є економічно обґрунтованою альтернативою постійним хмарним платежам [3].

Регуляторний контекст для систем класу «керування аудитами» формує додатковий шар вимог, що виходить за межі загальних міркувань про якість програмного забезпечення. Такі системи зазвичай підпадають під дію галузевих стандартів управління інформаційною безпекою, зокрема серії ISO/IEC 27001, що вимагають документованого контролю над тим, де саме зберігаються дані, хто має до них доступ та яким чином забезпечується їх цілісність протягом усього життєвого циклу. У контексті обробки персональних даних додається вимога

Загального регламенту захисту даних (GDPR) щодо локалізації обробки та можливості повного контролю над передаванням даних третім сторонам. On-premise розміщення дозволяє організації документально підтвердити відповідність цим вимогам безпосередньо через фізичний контроль над інфраструктурою, без необхідності покладатися на договірні гарантії хмарного провайдера щодо місця обробки даних [3].

З погляду моделі загроз система керування аудитами становить привабливу ціль для зловмисників саме через цінність даних, які вона накопичує: інформація про виявлені невідповідності та вразливості організації, якщо вона потрапить до третіх осіб, може бути використана для шантажу, конкурентної розвідки або підготовки цілеспрямованої атаки. Це обумовлює необхідність застосування принципу ешелонованої безпеки (defense in depth), за яким захист не покладається на єдиний механізм, а реалізується одночасно на кількох рівнях — мережевому, рівні операційної системи, рівні застосунку та рівні даних. У межах подальшого проєктування цей принцип знаходить відображення в розмежуванні рівнів інфраструктури, застосуванні захищеного дистрибутива Kubernetes з увімкненими механізмами контролю доступу та ізоляції чутливих параметрів конфігурації від образів контейнерів.

Варто зазначити, що жодна з розглянутих моделей не є універсально кращою — обрання моделі розміщення є рішенням, що балансує між контролем, вартістю володіння та операційною складністю відповідно до конкретного профілю ризиків та вимог організації. Для системи, що обробляє чутливі дані аудиту, потребує передбачуваної відмовостійкості та має стабільний профіль навантаження, ваги цього балансу схиляються на бік локального розміщення, що й зумовлює вибір моделі для подальшого проєктування в цій роботі.

Ці вимоги є наскрізними для всієї подальшої роботи: кожен з наступних підрозділів цього розділу розглядає один із методологічних підходів (класичне адміністрування, DevOps, SRE, Platform Engineering) саме крізь призму того, наскільки він здатний задовольнити сформульовані вище вимоги, а Розділ 2 формує на їх основі конкретні архітектурні рішення.

1.2 Особливості класичних методів адміністрування

До масового поширення хмарних технологій і практик автоматизації управління інфраструктурою спиралося переважно на ручну працю системного адміністратора, який безпосередньо взаємодіяв із кожним сервером. Цей підхід, що домінував на початку 2010-х років, ґрунтувався на трьох характерних практиках — ручному конфігуруванні, скриптингу та монолітному розгортанні застосунків. Розгляд цих практик і притаманних їм обмежень є необхідним для розуміння передумов появи сучасних методологій, оскільки саме недоліки класичної моделі визначили напрям подальшої еволюції.

За класичної моделі налаштування Linux-сервера виконувалося адміністратором інтерактивно: він підключався до машини за протоколом SSH і вносив зміни безпосередньо в її конфігурацію — встановлював пакети, редагував конфігураційні файли, ініціалізував базу даних. Приклад ручного конфігурування показано на рисунках 1.1 та 1.2.

```
C:\Users\local-admin>ssh rocky@192.168.2.186
rocky@192.168.2.186's password:
Last login: Sun May 31 11:58:03 2026
[rocky@cl01-m01 ~]$ vi server-conf
```

Рисунок 1.1 — Приклад ручного конфігурування серверу адміністратором

```
apt-get update && apt-get install -y nginx php-fpm mariadb-server
vim /etc/nginx/sites-available/audit-app      # правки вносяться руками
vim /etc/php/8.1/fpm/php.ini                  # тюнінг параметрів вручну
systemctl restart nginx php8.1-fpm
mysql -u root -p < schema.sql                 # схема БД накочується вручну
```

Рисунок 1.2 — Приклад ручного конфігурування із проблемою відтворюваності

Головна проблема такого підходу полягає в тому, що внесені зміни ніде не фіксуються у відтворюваній формі: фактичний стан сервера існує в конфігураційних файлах, історії оболнки та у пам'яті адміністратора. Накопичення таких ручних правок призводить до явища дрейфу конфігурації (configuration drift) — неконтрольованого розходження налаштувань між машинами, що мали б бути однаковими. Граничним проявом цього стає так

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				15
Змн.	Арк.	№ докум.	Підпис	Дата		

званий snowflake-сервер — унікальна машина, яку через накопичені «особливі» налаштування практично неможливо відтворити з нуля, а отже, і безпечно замінити у разі відмови. З такими серверами поводяться «як з домашніми улюбленцями, а не як з худобою»: кожен є незамінним і потребує індивідуального догляду.

Частковою відповіддю на рутинність ручних операцій став скриптинг — оформлення повторюваних дій у вигляді сценаріїв, найчастіше мовою оболонки. Приклад автоматизації адміністрування скриптингом зображено на рис. 1.3.

```
#!/bin/bash
set -e

scp ./build.tar.gz deploy@10.0.0.5:/tmp/

ssh deploy@10.0.0.5 _<< 'EOF'
systemctl stop audit-app
rm -rf /var/www/audit-app
mkdir -p /var/www/audit-app
tar -xzf /tmp/build.tar.gz -C /var/www/audit-app
systemctl start audit-app
EOF
```

Рисунок 1.3 — Приклад автоматизації адміністрування

Скриптинг частково знижував обсяг ручної праці, проте не усував її принципових вад.

По-перше, подібні сценарії є імперативними та не ідемпотентними: вони описують послідовність команд, а не бажаний кінцевий стан системи, тож їх повторний запуск або виконання на машині з дещо іншим станом може призвести до непередбачуваного результату.

По-друге, такий сценарій передбачає короточасну зупинку служб серверів на час оновлення, що безпосередньо погіршує доступність, яка є важливим показником при виборі платформи для системи контролю аудитів. По-третє, така ручна, повторювана й позбавлена тривалої цінності операційна робота в методології SRE отримала назву «toil»; її ключова проблема полягає в тому, що її обсяг зростає лінійно з масштабом інфраструктури й не масштабується [2]. Поєднання крихкості сценаріїв із недовірою до них породжує цикл страху автоматизації (automation fear): чим більше адміністратор побоюється, що

інструмент щось зламає, тим частіше він вносить зміни вручну, що, своєю чергою, робить інфраструктуру ще менш передбачуваною.

Архітектурно класична модель тяжіла до монолітного розгортання, коли застосунок разом із залежностями, а нерідко й база даних, розміщувалися на одному або кількох довгоживучих серверах і розгорталися як єдине ціле. Масштабування за такої моделі було переважно вертикальним — нарощуванням ресурсів окремої машини, а не додаванням додаткових вузлів. Унаслідок цього сервер ставав єдиною точкою відмови, а будь-яка, навіть незначна, зміна потребувала перерозгортання всього застосунку із супутнім простом. Сукупно описані практики формували крихку інфраструктуру (fragile infrastructure) — середовище, у якому зміни є ризикованими, а відновлення після збою залежить від наявності конкретних фахівців та їхніх неформалізованих знань [5]. Саме ці обмеження — нездатність відтворювати середовища, дрейф конфігурації, погана масштабованість і залежність від ручної праці — стали безпосередніми передумовами переходу до методології DevOps, що детально розглядається в наступному підрозділі.

Окрім суто технічних обмежень, класична модель адміністрування мала й глибші системні наслідки, які варто розглянути докладніше, оскільки саме вони, а не окремі незручності, зрештою зробили цей підхід непридатним для систем із підвищеними вимогами до надійності.

Перший і найкритичніший наслідок стосується надійності в розумінні стандарту ISO/IEC 25010. Неможливість відтворити сервер з нуля прямо суперечить підхарактеристиці відновлюваності — здатності системи відновити працездатний стан після відмови за прийнятний час. Якщо єдиний екземпляр snowflake-сервера виходить з ладу — через апаратну несправність, помилку адміністратора чи пошкодження файлової системи — відновлення перетворюється на тривалу процедуру ручного відтворення накопичених за роки налаштувань, точний перелік яких часто ніде не зафіксований. Для системи керування аудитами це означало б, що відмова сервера в період активної перевірки могла б призвести не лише до тривалого простою, а й до безповоротної

втрати доказової бази, якщо резервне копіювання було налаштоване так само неформально, як і решта інфраструктури. Таким чином, проблема відтворюваності з суто операційної незручності перетворюється на прямий ризик для цілісності даних та безперервності бізнес-процесів.

Другий наслідок— безпековий. За класичної моделі, де зміни вносяться інтерактивно через SSH, фактичний стан системи безпеки кожного сервера залишається непрозорим: неможливо швидко й достовірно відповісти на питання, які саме пакети встановлені, які порти відкриті, які користувачі мають доступ і чи застосовані актуальні оновлення безпеки на всіх машинах одночасно. Дрейф конфігурації призводить до того, що сервери, які мали б бути ідентичними, з часом починають відрізнятися й за параметрами безпеки — на одній машині оновлення встановлено, на іншій його забули застосувати. Для інфраструктури, що обробляє чутливі дані, така непрозорість є неприйнятною, оскільки унеможливорює систематичний аудит власної захищеності та збільшує поверхню атаки за рахунок неконтрольованих розбіжностей між вузлами.

Третій наслідок має організаційний характер і безпосередньо пов'язаний із залежністю від конкретних фахівців. Коли знання про стан інфраструктури існує переважно в пам'яті окремих адміністраторів, а не у формалізованому й версіонованому вигляді, організація потрапляє в стан критичної залежності від цих людей. Відхід ключового фахівця означає втрату частини знань про інфраструктуру, а передавання справ новому співробітникові перетворюється на тривалий і ненадійний процес. У термінах управління ризиками це формує так званий «фактор автобуса» (bus factor) — кількість співробітників, раптова недоступність яких паралізує роботу системи; за класичної моделі цей показник часто дорівнює одиниці, що є неприйнятно високим ризиком для будь-якої критичної системи.

Нарешті, варто кількісно осмислити вартість простою, до якого призводили описані практики. Кожне оновлення монолітного застосунку, що вимагало зупинки служб, прямо зменшувало показник доступності, який, як зазначено вище, формалізується через угоди про рівень обслуговування. Якщо процедура

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				18
Змн.	Арк.	№ докум.	Підпис	Дата		

оновлення з ручним втручанням триває, наприклад, тридцять хвилин і виконується щотижня, це саме по собі формує близько двадцяти шести годин планового простою на рік — і це без урахування незапланованих простоїв через помилки в самому процесі оновлення. Для системи рівня «дев'ять вісімок» (99,9% доступності) річний бюджет простою становить лише близько восьми годин сорока хвилин, тобто навіть планові оновлення за класичною моделлю самотужки вичерпували б його кількаразово. Це наочно демонструє, що класичний підхід був несумісний не лише зі строгими, а й із помірними вимогами до доступності.

Сукупність цих чотирьох вимірів — ризик для відновлюваності та цілісності даних, непрозорість стану безпеки, організаційна залежність від окремих фахівців та неприйнятна вартість простою — пояснює, чому індустрія не обмежилася частковими покращеннями на кшталт скриптингу, а перейшла до принципово іншої парадигми. Ця парадигма, що оформилася як методологія DevOps, запропонувала розглядати інфраструктуру не як сукупність унікальних серверів, які налаштовують вручну, а як код, що описує бажаний стан системи у відтворюваній формі, — підхід, який детально розглядається в наступному підрозділі.

1.3 Передумови переходу до DevOps

Розглянуті практики класичного адміністрування не були довільними помилками окремих фахівців — вони закономірно впливали з наявних на той час інструментів і організаційних структур. Проте сукупність притаманних їм обмежень із певного моменту почала суперечити вимогам бізнесу до швидкості та надійності постачання програмного забезпечення. Систематизація цих обмежень — як технічних, так і організаційних — дає змогу зрозуміти, що поява методології DevOps була не появою чергової «моди», а закономірною відповіддю на структурну кризу класичної моделі.

Описані раніше практики породжували взаємопов'язаний комплекс технічних проблем. По-перше, це нездатність до відтворення середовищ: стан сервера існував лише на ньому самому, тож точне повторення конфігурації на

іншій машині було практично неможливим. По-друге, дрейф конфігурації та snowflake-сервери унеможливлювали уніфіковане керування парком машин, оскільки кожна з них поступово ставала унікальною [5]. По-третє, монолітне розгортання та вертикальне масштабування створювали єдині точки відмови й робили будь-яку зміну ризикованою. По-четверте, операційна діяльність трималася на ручній, повторюваній праці (toil), обсяг якої зростав пропорційно до масштабу інфраструктури й тому не масштабувався [2]. Підсумковим наслідком ставала крихка інфраструктура, у якій зростання кількості систем не пришвидшувало, а сповільнювало роботу. Показово, що ці технічні обмеження прямо суперечили сформульованим у підрозділі 1.1 вимогам до надійності й доступності: середовище, яке неможливо відтворити й безпечно змінити, за визначенням не може бути ні відмовостійким, ні стабільно доступним.

Технічні обмеження були лише видимою частиною глибшої, організаційної проблеми. У більшості ІТ-організацій існував структурний конфлікт між командами розробки (Development) та експлуатації (Operations), зумовлений протилежними системами мотивації: розробників заохочували за швидке впровадження змін і нових функцій, тоді як експлуатацію — за стабільність і безперебійність роботи, тобто фактично за відсутність змін. У праці «The DevOps Handbook» цей феномен, услід за теорією обмежень Е. Голдратта, названо «ключовим хронічним конфліктом», що запускає низхідну спіраль: крихкі системи спричиняють збої, збої посилюють взаємні звинувачення, а накопичення технічного боргу робить кожну наступну зміну ще ризикованішою.

Організаційним проявом цього конфлікту стала так звана «стіна нерозуміння» (wall of confusion) — метафора, за якою розробка «перекидає» готовий код через умовну стіну на бік експлуатації, не переймаючись подальшою долею застосунку, а експлуатація отримує систему, яку не проєктувала й погано розуміє. Саме розділення відповідальності між ізольованими підрозділами (silos), за якого локальні цілі кожної команди перешкоджали досягненню спільної мети організації, і було коренем проблеми, що його суто технічними засобами усунути неможливо [6].

Подолання описаної кризи стало можливим завдяки збігу кількох передумов наприкінці 2000-х років. Поширення віртуалізації та хмарних платформ суттєво знизило бар'єр для програмного керування інфраструктурою, ідеї гнучкої розробки (Agile) почали застосовувати не лише до коду, а й до інфраструктури, а рух Velocity привернув увагу до питань продуктивності та експлуатації веброзвитку. Каталізатором об'єднання цих ідей стала перша конференція DevOpsDays, організована Патріком Дебуа у місті Гент у 2009 році; від назви цього заходу й походить власне термін «DevOps» [6].

За своєю суттю DevOps постав не як набір інструментів, а як організаційно-культурна відповідь на ключовий хронічний конфлікт: усунення «стіни» між розробкою та експлуатацією шляхом запровадження спільної відповідальності за весь життєвий цикл продукту, від написання коду до його експлуатації у промисловому середовищі. Конкретні інженерні практики й інструментарій, за допомогою яких ця ідеологія втілюється на практиці, є предметом розгляду наступного підрозділу.

Як було показано раніше, DevOps за своєю природою є передусім організаційно-культурним підходом. Проте його впровадження неможливе без конкретних інженерних практик та інструментів, які операціоналізують його принципи. Методологічною основою цих практик є концепція «трьох шляхів» (The Three Ways), сформульована у праці: перший шлях забезпечує швидкий потік роботи зліва направо — від розробки до експлуатації; другий — налагоджує зворотний зв'язок у протилежному напрямі; третій — формує культуру безперервного навчання та експериментування. Розглянуті нижче практики є технічним втіленням цих принципів і водночас прямою відповіддю на обмеження класичної моделі, систематизовані в підрозділі 1.2.

Інфраструктура як код (Infrastructure as Code, IaC) — це підхід, за якого обчислювальні ресурси, мережі та конфігурації описуються у машинозчитуваних файлах визначень і керуються тими самими методами, що й програмний код, тобто з обов'язковим версіонуванням, тестуванням і автоматизованим застосуванням. Ключовою є відмова від імперативного опису послідовності дій на

користь декларативного опису бажаного кінцевого стану системи. Це забезпечує дотримання основних принципів ІаС — відтворюваності, узгодженості та одноразовості ресурсів, що дає змогу поводитися із серверами «як з худобою, а не з домашніми улюбленцями» [5]. На відміну від наведеного у підрозділі 1.2 ручного конфігурування, той самий результат у декларативній парадигмі описується ідемпотентно: Приклад інфраструктури як код зображено на рис. 1.4.

```
# Фрагмент Ansible-плейбука: бажаний стан сервера застосунку
- hosts: app_servers
  become: true
  tasks:
    - name: Install nginx, php-fpm, and mariadb-server
      apt:
        name: [nginx, php-fpm, mariadb-server]
        state: present
        update_cache: true

    - name: Deploy nginx configuration from the template
      template:
        src: audit-app.conf.j2
        dest: /etc/nginx/sites-available/audit-app
        notify: restart nginx
```

Рисунок 1.4 — Приклад інфраструктури як код (Infrastructure as Code)

Такий опис можна безпечно застосовувати багаторазово: інструмент сам приводить систему до заданого стану незалежно від поточного, що усуває дрейф конфігурації та проблему snowflake-серверів. На практиці завдання ІаС розподіляються між інструментами різного призначення: засоби розгортання (зокрема Terraform) відповідають за створення самих ресурсів — віртуальних машин, мереж, сховищ, — тоді як засоби керування конфігурацією (зокрема Ansible) приводять уже наявні системи до потрібного стану. Безперервна інтеграція та доставка (CI/CD).

Якщо ІаС розв'язує проблему відтворюваності середовищ, то практики безперервної інтеграції та доставки (Continuous Integration/Continuous Delivery, CI/CD) забезпечують швидкий і надійний потік змін — реалізацію першого шляху

DevOps. Безперервна інтеграція передбачає часте злиття змін коду до спільної гілки з автоматичним складанням і тестуванням, що дає змогу виявляти дефекти на ранніх етапах. Її продовженням є концепція конвеєра розгортання (deployment pipeline) — автоматизованого процесу проведення кожної зміни від моменту фіксації у системі контролю версій до випуску в експлуатацію. Завдяки автоматизації складання, тестування та розгортання команди отримують змогу випускати зміни за години чи навіть хвилини, а малий розмір кожної зміни знижує ризик і спрощує виявлення причин відмов [7]. Це безпосередньо протиставляється монолітному розгортанню класичної моделі з його тривалими, ризикованими й рідкісними релізами.

Окремою практикою, що радикально вплинула на спосіб постачання застосунків, стала контейнеризація — пакування застосунку разом з усіма його залежностями у стандартизований незмінний образ на основі віртуалізації рівня операційної системи. Контейнери забезпечують ізоляцію ресурсів та середовища виконання і усувають класичну проблему розбіжності середовищ (відому як «працює на моїй машині»), оскільки той самий образ однаково виконується на будь-якому вузлі. Незмінність (immutability) контейнерних образів узгоджується з принципами ІаС: оновлення відбувається не правкою наявного контейнера, а розгортанням нового [8]. Найпоширенішим інструментом контейнеризації став Docker. Хоча базові концепції ізоляції ресурсів на рівні ядра операційної системи розвивалися понад десятиліття, саме вихід Docker у 2013 році забезпечив стандартизацію формату образів та масове впровадження цієї технології в архітектуру сучасних інформаційних систем.

Зі зростанням кількості контейнерів ручне керування ними стає неможливим, що зумовлює потребу в оркестрації. Системи оркестрації автоматизують розгортання, масштабування, балансування навантаження та самовідновлення контейнеризованих застосунків на кластері машин на основі декларативно заданого бажаного стану. Провідним рішенням у цій галузі є Kubernetes — система з відкритим кодом, що є прямим спадкоємцем розробленого в Google диспетчера кластерів Borg і успадкувала від нього такі ключові

механізми, як планування завдань, мінімізація часу відновлення після відмов та зниження ймовірності корельованих збоїв. Саме Kubernetes (а точніше, його захищений дистрибутив, обґрунтований у подальших розділах) є основою практичної частини цієї роботи.

Вибір Kubernetes на противагу простішим інструментам оркестрації (наприклад, Docker Compose) зумовлений тим, що Kubernetes дозволяє чітко розмежувати зони відповідальності між системними адміністраторами (Ops) та розробниками (Dev) у межах єдиної платформи. Адміністратори фокусуються на підготовці інфраструктури — налаштуванні просторів імен (Namespaces) для ізоляції середовищ та виділенні постійних сховищ (Persistent Volumes). Своєю чергою, розробники відповідають безпосередньо за розгортання застосунків. Оскільки управління всіма ресурсами здійснюється через декларативні маніфести у форматі YAML, це формує єдину зрозумілу мову взаємодії для обох команд, що цілком відповідає парадигмі DevOps.

Сукупно розглянуті практики впроваджують ідеологію DevOps і усувають ключові обмеження класичного підходу. Водночас вони зосереджені переважно на швидкості та надійності постачання змін, тоді як питання надійної експлуатації систем у промисловому середовищі потребує окремого інженерного підходу, що його пропонує методологія Site Reliability Engineering.

1.4 Site Reliability Engineering як інженерний підхід до надійності

Методологія DevOps зосереджена насамперед на швидкому й надійному постачанні змін. Проте вона не дає вичерпної відповіді на питання, як саме вимірювати надійність експлуатованих систем і керувати нею. Цю прогалину заповнює Site Reliability Engineering (SRE) — підхід, що виник у компанії Google і, за визначенням його засновника Бена Трейнора, є результатом застосування методів програмної інженерії до задач, які традиційно належали до сфери експлуатації [2]. SRE прийнято розглядати як конкретну інженерну реалізацію принципів DevOps: якщо DevOps описує, що і чому потрібно робити, то SRE пропонує вимірювані, як саме це робити.

В основі SRE лежить теза: надійність є найважливішою характеристикою системи, оскільки непрацездатний сервіс не приносить користі незалежно від багатства його функціональних можливостей. Водночас SRE відмовляється від наївної мети стовідсоткової безвідмовності, визнаючи, що понад певний поріг витрати на підвищення надійності зростають непропорційно до вигоди, а сама ця надійність стає невідчутною для користувача. Тому надійність розглядається як свідомо керований компроміс між стабільністю та темпом змін. Окремим принципом є системна боротьба з toil — ручною, повторюваною та автоматизовуваною операційною працею, яка не створює тривалої цінності й обсяг якої зростає пропорційно до масштабу сервісу; SRE прагне обмежувати частку такої праці й усувати її засобами автоматизації, вивільняючи час інженерів на розробку [2].

Ключовим внеском SRE є переведення поняття надійності з якісної у кількісну площину за допомогою трьох пов'язаних понять. Індикатор рівня обслуговування (Service Level Indicator, SLI) — це кількісна міра певного аспекту якості сервісу, наприклад частки успішних запитів або затримки відповіді. Ціль рівня обслуговування (Service Level Objective, SLO) — це цільове значення або діапазон, у якому має перебувати відповідний SLI. На відміну від них, угода про рівень обслуговування (Service Level Agreement, SLA) є зовнішнім договором із користувачем, що передбачає наслідки за недотримання погоджених показників.

На основі SLO визначається бюджет помилок (error budget) — допустима частка ненадійності, що обчислюється як різниця між стовідсотковою та цільовою надійністю. Цей механізм має принципове значення, оскільки безпосередньо розв'язує хронічний конфлікт між розробкою та експлуатацією: доки бюджет помилок не вичерпано, команда може вільно впроваджувати зміни; щойно він вичерпується, пріоритет автоматично зміщується на підвищення стабільності [2]. У такий спосіб суб'єктивне протистояння «швидкість проти надійності» замінюється об'єктивним, узгодженим обома сторонами числовим критерієм.

Кількісне керування надійністю неможливе без здатності спостерігати за станом системи. Класичний моніторинг у SRE рекомендують будувати навколо

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		25

«чотирьох золотих сигналів» — затримки, трафіку, помилок і насиченості ресурсів, що дають достатню картину стану сервісу для користувача. Розвитком моніторингу є концепція спостережуваності (observability) — здатності робити висновки про внутрішній стан системи на основі даних, які вона виробляє назовні. Спостережуваність традиційно ґрунтується на «трьох стовпах» телеметрії: метриках (числових вимірюваннях у часі), логах (записах подій) і трасуваннях (відстеженні проходження запиту крізь розподілену систему) [9].

Невіддільною складовою SRE є структуроване керування інцидентами, що передбачає чіткий розподіл ролей під час реагування, а після усунення збою — проведення «безвинного» (blameless) розбору (postmortem), орієнтованого не на пошук винних, а на виявлення системних причин відмови та запобігання її повторенню [2]. Такий підхід формує культуру безперервного навчання, узгоджену з третім шляхом DevOps.

Підсумовуючи, SRE надає інженерно строгий, кількісний апарат для забезпечення надійності експлуатації. Проте зі зростанням масштабу організації та різноманіття інструментів на інженерів лягає дедалі більше когнітивне навантаження, пов'язане з опануванням усього технологічного стеку.

Окремої уваги заслуговує практичне застосування механізму бюджету помилок, оскільки саме він перетворює абстрактні принципи SRE на робочий інструмент управління. Розглянемо це на прикладі, релевантному для системи керування аудитами. Якщо для такої системи встановлено ціль рівня обслуговування у 99,9% доступності за квартал, то бюджет помилок становить 0,1% від загального часу, що еквівалентно приблизно двом годинам тридцяти хвилинам припустимого простою за квартал. Цей бюджет є спільним ресурсом команди розробки та експлуатації: кожне планове оновлення, кожен збій і кожне профілактичне обслуговування «витрачають» частину бюджету. Доки він не вичерпаний, команда має свободу впроваджувати нові функції та зміни з прийнятним рівнем ризику. Якщо ж серія інцидентів вичерпала бюджет достроково, подальше впровадження змін призупиняється на користь робіт зі стабілізації — і це рішення є не предметом суперечки між командами, а

автоматичним наслідком об'єктивного показника. Для системи аудиту, де простій під час активної перевірки має високу ціну, такий механізм дає змогу свідомо й прозоро балансувати між потребою в нових можливостях та вимогою стабільності.

Важливим аспектом, що відрізняє SRE від класичної експлуатації, є ставлення до автоматизації як до способу боротьби з toil. У класичній моделі адміністрування, розглянутій у попередніх підрозділах, реакція на зростання навантаження зазвичай полягала в найманні додаткового персоналу — обсяг ручної праці зростав лінійно разом із масштабом інфраструктури. SRE натомість встановлює явне обмеження на частку часу, яку інженер може витратити на toil (за рекомендацією Google — не більше 50%), а решту часу спрямовує на інженерну роботу, що усуває першопричини ручної праці. Цей принцип має прямий зв'язок із темою цієї роботи: інфраструктура як код, декларативне розгортання та механізми самовідновлення, які реалізуються в подальших розділах, є саме тими інженерними засобами, що дають змогу обслуговувати інфраструктуру зі сталою кількістю інженерів незалежно від зростання кількості вузлів. Без такого підходу обслуговування кластера з багатьох вузлів вимагало б пропорційного збільшення штату, що є економічно невиправданим.

Концепція спостережуваності, згадана вище, також потребує детальнішого розгляду в контексті її практичної відмінності від традиційного моніторингу. Класичний моніторинг побудований навколо наперед визначених питань: інженер заздалегідь знає, які показники можуть указувати на проблему, і налаштовує відповідні перевірки та сповіщення. Такий підхід ефективний для відомих, передбачуваних режимів відмови, проте виявляється недостатнім для складних розподілених систем, у яких відмови часто виникають унаслідок непередбачуваної взаємодії компонентів. Спостережуваність розв'язує цю проблему, забезпечуючи достатню повноту телеметрії для дослідження станів, які не були передбачені заздалегідь, — тобто для відповіді на питання, які інженер ще не знав, що поставить. Саме тому в підсистемі спостережуваності, що проєктується в цій роботі, поєднано збирання як метрик, так і централізованих

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		27

логів: метрики дають змогу швидко виявити відхилення в поведінці системи, а логи — дослідити його першопричину, забезпечуючи перехід від констатації факту відмови до розуміння її механізму.

Нарешті, варто відзначити організаційний вимір SRE, який часто залишається поза увагою при суто технічному розгляді. Практика безвинного розбору інцидентів ґрунтується на засадничому припущенні, що люди не помиляються через недбалість, а діють раціонально з огляду на наявну в них інформацію та наявні інструменти; отже, причиною інциденту слід вважати не конкретну дію людини, а недосконалість системи, що допустила цю дію та її наслідки. Такий підхід формує культуру, у якій інженери не приховують помилок зі страху покарання, а відкрито аналізують їх для запобігання повторенню, що в кінцевому підсумку підвищує надійність системи в цілому.

Цей принцип тісно пов'язаний із концепцією відтворюваності інфраструктури: коли стан системи описано як код у системі контролю версій, будь-яка зміна є простежуваною, а наслідки помилкової зміни — відтворюваними та оборотними, що зменшує ціну помилки й узгоджується з культурою безперервного навчання. Таким чином, SRE постає не лише як набір технічних практик, а як цілісна інженерна культура, що поєднує кількісне керування надійністю з організаційними механізмами навчання на досвіді.

1.5 Platform Engineering як еволюція операційної моделі

Як було зазначено наприкінці попереднього підрозділу, послідовне впровадження практик DevOps і SRE має побічний наслідок: на команди розробки лягає дедалі більший обсяг суміжних обов'язків — опанування інфраструктури як коду, конвеєрів CI/CD, оркестрації, безпеки та спостережуваності. Коли продуктова команда змушена самотійно володіти всім цим стеком, її здатність зосередитися на основній меті — створенні цінності для користувача — знижується. Відповіддю на цей виклик стала методологія Platform Engineering, що виникла наприкінці 2010-х років і ґрунтується на ідеях, систематизованих у праці «Team Topologies» [10].

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		28

Центральним поняттям, що обґрунтовує потребу в інженерії платформ, є когнітивне навантаження — сукупний обсяг інформації, який команда має утримувати в полі уваги для виконання своєї роботи. У праці надмірне когнітивне навантаження визначено як одну з головних причин уповільнення постачання програмного забезпечення. Зростання складності сучасної інфраструктури призвело до того, що від розробників почали очікувати одночасного володіння розробкою застосунку, конфігуруванням інфраструктури, конвеєрами доставки, питаннями безпеки, відповідності та спостережуваності. Таке навантаження не лише сповільнює розробку, а й підвищує ймовірність помилок. Принциповою тезою Platform Engineering є те, що цю складність не можна просто перекласти на продуктові команди — її потрібно інкапсулювати за чітко визначеним інтерфейсом, надавши командам відповідні «огорожі» (guardrails), що убезпечують від помилок, не обмежуючи самостійності

Технічним втіленням цього підходу є внутрішня платформа розробника (Internal Developer Platform, IDP) — шар самообслуговуваних інструментів, інтерфейсів та автоматизації, побудований поверх наявної інфраструктури. Згідно з моделлю взаємодії команд, платформна команда надає такий шар іншим командам за принципом «як послугу» (X-as-a-Service): продуктова команда споживає можливості платформи через стабільний інтерфейс, не вникаючи у внутрішню реалізацію. Ключовим елементом IDP є так звані «золоті стежки» (golden paths) — рекомендовані, добре задокументовані та підтримувані платформною командою шляхи виконання типових завдань, що звужують кількість рішень, які доводиться ухвалювати розробникові. Модель самообслуговування усуває потребу у зверненнях до окремого підрозділу експлуатації через систему заявок (ticket-driven підхід) і тим самим ліквідує ті самі організаційні «передавання через стіну», але вже на новому технологічному рівні.

Принципова відмінність Platform Engineering від простого надання набору інструментів полягає у ставленні до платформи як до продукту, а не як до внутрішнього проєкту чи «фабрики із виконання заявок». Це означає, що платформа має власних користувачів (внутрішніх розробників), розвивається на

основі їхнього зворотного зв'язку, має дорожню карту й орієнтована на якість досвіду розробника (Developer Experience, DevEx). Щоб уникнути створення надмірно складної та незатребуваної платформи, у праці запропоновано концепцію «найтоншої життєздатної платформи» (Thinnest Viable Platform, TVP) — за аналогією з мінімально життєздатним продуктом (MVP) вона передбачає реалізацію лише найменшого набору інтерфейсів, документації та інструментів, що дає вимірюване прискорення роботи команд[10]. Такий підхід убезпечує від типової помилки — побудови громіздкої платформи з функціями, яких користувачі не потребують.

Отже, Platform Engineering не заперечує і не замінює DevOps та SRE, а надбудовується над ними, упаковуючи інфраструктуру як код, конвеєри доставки, оркестрацію та спостережуваність у цілісний самообслуговуваний продукт. Це завершує простежену в розділі еволюцію операційної моделі: від ручного класичного адміністрування — через культурно-процесну трансформацію DevOps і кількісну інженерію надійності SRE — до інкапсуляції складності засобами Platform Engineering. Саме синергія цих трьох методологій становить методологічну основу проєктування, що здійснюється у наступному розділі. Обґрунтування конкретного технологічного стеку для реалізації цієї моделі та постановка технічного завдання наведені в підрозділі 1.7.

Проведений у першому розділі аналіз дає змогу перейти від теоретичних засад до формування інженерних рішень. У ньому здійснено синтез розглянутих методологій у систему принципів проєктування, на їх основі обґрунтовано вибір конкретного технологічного стеку для on-premise реалізації та сформульовано технічне завдання.

Узагальнення проаналізованих підходів дозволяє виокремити такі принципи, яким має відповідати проєктована інфраструктура. По-перше, декларативність і відтворюваність: усі компоненти інфраструктури описуються як код, що усуває дрейф конфігурації та робить середовища відтворюваними [5]. По-друге, незмінність та автоматизація: зміни вносяться шляхом перерозгортання, а ручна операційна праця (toil) системно мінімізується [2]. По-третє,

відмовостійкість на кожному рівні: інфраструктура не повинна містити єдиних точок відмови, що впливає з вимог до надійності та доступності, сформульованих у підрозділі 1.1. По-четверте, кількісно вимірювана надійність та спостережуваність: стан системи має бути доступним для спостереження, а її надійність — піддаватися кількісній оцінці через SLI/SLO [2]. По-п'яте, безпека та відповідність за замовчуванням, що є критичним для on-premise системи, яка обробляє чутливі дані аудиту [4]. По-шосте, самообслуговування та зниження когнітивного навантаження як кінцева мета операційної моделі [10].

Вибір кожного компонента стеку зумовлений наведеними принципами та специфікою on-premise розгортання.

Платформа віртуалізації — Proxmox VE. Як базовий рівень обрано Proxmox VE — відкриту (ліцензія AGPLv3) платформу серверної віртуалізації на основі Debian GNU/Linux, що поєднує дві технології: повну апаратну віртуалізацію на базі KVM/QEMU для віртуальних машин і контейнери рівня операційної системи LXC [13].

Вибір Proxmox VE як базової платформи зумовлений глибокою кризою та архітектурними недоліками традиційних комерційних рішень для on-premise розгортання — насамперед VMware vSphere та Microsoft Hyper-V.

Незважаючи на історичне домінування на ринку, використання платформ VMware наразі пов'язане з критичними бізнес-ризиками. Після поглинання компанії корпорацією Broadcom відбулася радикальна та агресивна зміна ліцензійної політики. Повна ліквідація безкоштовних версій гіпервізора (ESXi Free), примусовий перехід на модель дорогих пакетних підписок та штучне обмеження функціоналу призвели до непрогнозованого зростання загальної вартості володіння (TCO) у кілька разів. Microsoft фактично згорнула розвиток автономного безкоштовного продукту Hyper-V Server, змушуючи користувачів купувати ліцензії Windows Server. Порівняння платформ віртуалізації зроблено в таблиці А.1

Для цілей цієї роботи принциповим є саме режим повної віртуалізації KVM, оскільки вузли кластера RKE2 потребують ізольованих віртуальних машин із

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		31

власним ядром, а не контейнерів. Вибір Proxmox зумовлений не лише його відкритістю та відповідністю on-premise моделі (повний контроль над інфраструктурою та даними згідно з підрозділом 1.1), а й трьома властивостями, що безпосередньо підтримують обрані методології.

По-перше, наявність REST API та відповідного провайдера для Terraform робить платформу придатною до декларативного керування — віртуальні машини можна розгорнути як код, реалізуючи принцип IaC.

По-друге, вбудовані механізми кластеризації (на основі Corosync), менеджер високої доступності з автоматичним перенесенням ВМ у разі відмови вузла та «жива» міграція (live migration) забезпечують відмовостійкість уже на рівні віртуалізації, доповнюючи відмовостійкість на рівнях оркестрації та даних і втілюючи принцип ешелонованої надійності.

По-третє, підтримка програмно-визначуваних сховищ (ZFS, Ceph, LVM) забезпечує надійне та надлишкове зберігання даних для застосунків зі станом [13]. Сукупно це робить Proxmox не просто гіпервізором, а інтегрованим фундаментом, властивості якого узгоджені з вимогами проєктованої інфраструктури.

Інфраструктура як код — Terraform та Ansible. Завдання IaC розподілено між двома інструментами відповідно до їх призначення: Terraform у декларативний спосіб розгортає віртуальні машини на Proxmox, забезпечуючи відтворюване створення ресурсів, тоді як Ansible виконує керування конфігурацією вже наявних систем — початкове налаштування, посилення безпеки операційної системи та розгортання компонентів. Така комбінація реалізує принципи декларативності та автоматизації.

Хоча Terraform може частково виконувати функції управління конфігураціями за допомогою local-exec та remote-exec, а для Ansible існують модулі для розгортання інфраструктури, в нашому проєкті найкращою практикою залишається їх комбіноване використання, де кожен інструмент вирішує свої профільні завдання. Подальші підрозділи цього розділу розглядають застосування цієї комбінації для проєктування кожного з рівнів інфраструктури — від

середовища віртуалізації до підсистеми спостережуваності. Порівняння Terraform та Ansible наведено в таблиці 1.1

Таблиця 1.1

Порівняльний аналіз інструментів інфраструктури як коду

Критерій	Terraform	Ansible
Призначення	Розгортання інфраструктури	Керування конфігурацією наявних систем
Підхід	Декларативний	Декларативний / процедурний
Мова опису	HCL (HashiCorp Configuration Language)	YAML (playbooks + roles)
Агент на хості	Не потрібен	Не потрібен (з'єднання через SSH)
Управління станом	Файл стану (terraform.tfstate)	Без стану

Оркестрація контейнерів — RKE2. Як платформу оркестрації обрано RKE2 — дистрибутив Kubernetes від SUSE, орієнтований на безпеку та відповідність. Цей вибір прямо зумовлений темою роботи: RKE2 за замовчуванням відповідає вимогам бенчмарку безпеки CIS Kubernetes Benchmark [11]. Такий рівень захищеності «з коробки» узгоджується з вимогами до системи керування аудитами значно краще, ніж самостійне посилення ванільного Kubernetes або менш захищені за замовчуванням дистрибутиви. Відмовостійкість на рівні оркестрації забезпечується розгортанням кількох керувальних вузлів (control plane) із вбудованим сховищем etcd.

Доцільність такого вибору стає очевиднішою при порівнянні RKE2 з альтернативними варіантами постачання Kubernetes. Хоча всі дистрибутиви реалізують єдиний програмний інтерфейс оркестрації та є сертифікованими на відповідність специфікації Kubernetes, вони суттєво різняться за компромісом між споживанням ресурсів, простотою розгортання та рівнем захищеності за замовчуванням. Ванільний Kubernetes, що розгортається утилітою kubectl, надає максимальну гнучкість, проте перекладає весь тягар посилення безпеки,

налаштування мережових політик та шифрування на інженера, що суперечить принципу «безпеки за замовчуванням» і збільшує ймовірність помилок конфігурування. Полегшений дистрибутив K3s, навпаки, оптимізований передусім під середовища з обмеженими ресурсами та периферійні розгортання, через що частина механізмів захисту в ньому спрощена, а типове використання вбудованої бази SQLite замість etcd погіршує характеристики відмовостійкості рівня керування. RKE2 посідає проміжну позицію, поєднуючи просте розгортання з єдиного бінарного файлу, властиве K3s, із промисловим рівнем безпеки та відмовостійким сховищем стану на основі etcd. Зведене порівняння трьох розглянутих варіантів за ключовими для цієї роботи показниками наведено в таблиці 1.2.

Таблиця 1.2

Порівняння дистрибутивів Kubernetes за фактичними показниками

Критерій	K8s (kubeadm)	K3s	RKE2
Мін. RAM (керувальний вузол)	2 ГБ (рек. 4 ГБ)	512 МБ (рек. 1 ГБ)	2 ГБ (рек. 4 ГБ)
Мін. CPU (керувальний вузол)	2 ядра	1 ядро	1 ядро
Типове споживання RAM (сервер)	~2 ГБ+	~1,2 ГБ	~1,5 ГБ
Споживання RAM (агент)	~700 МБ	~268 МБ	~400 МБ
Розмір дистрибутива	кілька пакетів	<100 МБ, єдиний бінарний файл	єдиний бінарний файл
Сховище стану (типове)	etcd	SQLite	etcd
CIS Kubernetes Benchmark	ручне посилення	не посилений типово	профіль cis вбудований
Шифрування Secrets at rest	вимкнено типово	увімкнено типово	увімкнено типово

Для контейнеризації цільового застосунку розглянуто чотири інструменти роботи з контейнерами, що різняться за призначенням та архітектурою. Їх порівняння за ключовими для цієї роботи критеріями наведено в табл. 2.6.

Інструменти Docker і Podman є повнофункціональними платформами складання та запуску контейнерів, тоді як containerd є низькорівневим середовищем виконання, а LXC реалізує віртуалізацію рівня операційної системи з власним форматом образів. Порівняння рішень контейнеризації наведено в таблиці А.2

Принциповим є те, що Docker і Podman сумісні між собою: обидва використовують той самий формат опису складання (Dockerfile) та той самий стандарт образів OCI, унаслідок чого образ, зібраний одним інструментом, без змін виконується будь-яким OCI-сумісним середовищем. Відмінність між ними полягає переважно в архітектурі: Docker працює через фоновий демон (dockerd) з підвищеними правами, тоді як Podman є бездемонним і за замовчуванням запускається без прав суперкористувача, що зменшує поверхню атаки. З огляду на спрямованість роботи на захищеність, для практичного складання образу застосунку використано Podman як інструмент, сумісний з Docker за форматом, але з безпечнішою за замовчуванням архітектурою.

Отриманий OCI-образ виконується штатним для кластера RKE2 середовищем containerd, тоді як LXC, призначений для системних контейнерів рівня операційної системи, для пакування окремого застосунку не застосовується.

Вибір конкретної реалізації контролера входу має суттєве значення для безпеки, оскільки цей компонент перебуває безпосередньо в шляху проходження зовнішнього трафіку рівня застосунків. Дистрибутив RKE2 надає на вибір кілька реалізацій контролера входу, найпоширенішими з яких є Traefik та ingress-nginx. Порівняння описано в таблиці 1.3.

Таблиця 1.3

Порівняння контролерів Ingress для кластера RKE2

Критерій	Traefik	Ingress NGINX	HAProxy Ingress
Статус підтримки	Активно підтримується	EOL з березня 2026	Активно підтримується
Майбутні виправлення вразливостей	Так	Не випускаються	Так

Закінчення таблиці 1.3

Критерій	Traefik	Ingress NGINX	HAProxy Ingress
Типовий контролер RKE2 від v1.36	Так	Виводиться з обігу	Ні
Проксі-ядро	Власне (Go)	NGINX	HAProxy
Зміна конфігурації	Динамічно, без перезапуску	Перезавантаження ConfigMap	Часткова
Підтримка Gateway API	Так	Заморожено (Ingress API)	Так
Інтеграція Let's Encrypt (ACME)	Вбудована	Через cert-manager	Через cert-manager

З огляду на наведене порівняння контролером входу обрано Traefik. Цей вибір зумовлений тим, що Traefik активно підтримується та визначений як типовий контролер входу для майбутніх версій RKE2, реалізує сучасний стандарт Gateway API, підтримує динамічну зміну конфігурації без перезавантаження та має вбудовану інтеграцію з механізмом автоматичного отримання TLS-сертифікатів. Постачання Traefik у складі дистрибутива RKE2 спрощує операційну модель, оскільки не потребує встановлення стороннього компонента, а його активна підтримка усуває ризик використання непідтримуваного програмного забезпечення в шляху проходження трафіку, що є важливим для відповідності вимогам аудиту захищеності.

Слід враховувати, що проєкт ingress-nginx у листопаді 2025 року оголошено таким, що виводиться з підтримки: після березня 2026 року для нього припиняється випуск оновлень та виправлень вразливостей безпеки, а сам дистрибутив RKE2, починаючи з версії v1.36, переходить на Traefik як типовий контролер. Для системи, що обробляє чутливі дані аудиту та підлягає вимогам відповідності (зокрема ISO/IEC 27001), використання компонента без подальшої підтримки безпеки є небажаним.

Доставка навантаження та керування — ArgoCD, Rancher і cert-manager. Безперервна доставка реалізується за моделлю GitOps засобами ArgoCD —

оператора, що синхронізує стан кластера з декларативним описом у репозиторії [15]. Шар керування та самообслуговування над кластером забезпечує платформа Rancher, що надає централізоване адміністрування, керування доступом і спостереження за станом кластера [16]. Автоматизацію керування TLS-сертифікатами покладено на cert-manager [17].

Рівень даних — кластер Galera. Оскільки цільове навантаження є застосунком зі станом, що використовує реляційну базу даних, рівень даних реалізується на основі MariaDB Galera Cluster. Galera забезпечує синхронну multi-master реплікацію, за якої кожен вузол містить ідентичну копію даних і здатний обслуговувати як читання, так і запис; у разі відмови вузла решта продовжує роботу без ручного перемикавання, а для запобігання ситуації «розщеплення мозку» (split-brain) кластер потребує кворуму з непарної кількості вузлів (щонайменше трьох) [12]. Порівняння з іншими рішеннями баз даних проілюстровано в таблиці 1.4.

Таблиця 1.4

Порівняння технологій баз даних

Критерій	MariaDB Galera Cluster	Oracle Database	MongoDB
Тип реплікації	Синхронна Multi-Master (wsrep API)	Active Data Guard / RAC (синхронна)	Асинхронна / Eventually Consistent
ACID-транзакції	Повна підтримка (InnoDB)	Повна підтримка	Часткова (з 4.0+, обмежено)
Вирішення конфліктів	Зріла сертифікаційна модель	Зріла (Oracle SCN), але централізована архітектура RAC	Не застосовно (документна модель)
RPO (втрата даних)	0 — синхронний коміт на всіх вузлах	0 (Data Guard Sync), але висока вартість	Не гарантовано (залежить від write concern)
Ліцензування	GPLv2 (відкритий код)	Комерційна; від \$25 000+/процесор	SSPL (обмежено відкрита)

Критерій	MariaDB Galera Cluster	Oracle Database	MongoDB
Сумісність з MySQL	Повна (drop-in заміна)	Відсутня (інший діалект SQL)	Відсутня (інша модель даних)
Рефакторинг коду	Не потрібен	Значний (PL/SQL, інший синтаксис)	Фундаментальний (зміна схеми даних)

Спостережуваність — Prometheus, Grafana та Loki. Рівень спостережуваності будується на поєднанні трьох інструментів, що разом охоплюють два з трьох «стовпів» телеметрії, розглянутих у підрозділі 1.5 [9].

Під час вибору платформи для забезпечення спостережуваності було проведено порівняння Grafana Stack (Prometheus, Loki, Grafana) та Elastic Stack (Elasticsearch, Logstash, Kibana). Порівняння платформ моніторингу представлено в таблиці А.3

Рішення на користь Grafana Stack зумовлене його більш ефективною та cloud-native архітектурою для Kubernetes-середовищ. На відміну від Elasticsearch, який створює повнотекстові індекси для всіх логів і потребує значних обчислювальних ресурсів та швидких накопичувачів, Loki індексує лише метадані логів, що суттєво знижує споживання CPU, оперативної пам'яті та витрати на зберігання даних. Крім того, Prometheus є стандартом де-факто в екосистемі Kubernetes і підтримується більшістю сучасних сервісів та інфраструктурних компонентів без додаткових агентів або складних налаштувань.

Висновки до першого розділу

У першому розділі розглянуто підходи до побудови on-premise інфраструктури для систем, що оперують чутливими даними. Проаналізовано обмеження класичних методів адміністрування — дрейф конфігурації, нездатність відтворити середовище після відмови та неприйнятна вартість

планового простою — що унеможливають їх застосування за підвищених вимог до надійності. Розглянуто методології DevOps, SRE та Platform Engineering як взаємодоповнювальні підходи до усунення цих обмежень, а також проведено порівняльний аналіз ключових інструментів і платформ у кожній з категорій. На основі проведеного аналізу сформульовано шість принципів проєктування та обґрунтовано вибір технологічного стеку — Proxmox VE, Terraform, Ansible, RKE2, MariaDB Galera Cluster, Prometheus, Loki і Grafana — як прямої реалізації цих принципів в умовах on-premise розгортання.

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				39
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 2. ПРОЄКТУВАННЯ ON-PREMISE ІНФРАСТРУКТУРИ

2.1 Загальна архітектура інфраструктури та вимоги до неї

У цьому розділі здійснюється перехід від теоретичних засад до проєктних рішень. Цей підрозділ визначає загальну архітектуру інфраструктури, профіль цільового навантаження та топологію вузлів, що деталізуються в подальших підрозділах.

Відповідно до спрямованості роботи, цільовий застосунок — система керування аудитами — розглядається не як об'єкт розроблення, а як профіль навантаження, навколо якого проєктується інфраструктура. Для цілей проєктування суттєвими є лише ті його характеристики, що впливають на інфраструктурні рішення: застосунок є вебзастосунком зі станом (stateful), що взаємодіє з користувачем за протоколом HTTP/HTTPS та зберігає дані у реляційній, сумісній із MySQL/MariaDB базі даних. Така абстракція має методологічну перевагу: спроектована інфраструктура залишається придатною для будь-якого навантаження з аналогічним профілем, а не прив'язується до конкретного застосунку.

Інфраструктуру побудовано за принципом поділу на функціональні рівні, що відповідає принципу ешелонованої надійності, закладеному в підрозділі 1.7. Виокремлено п'ять рівнів. Рівень віртуалізації на основі Proxmox VE утворює фізичний фундамент, на якому розгортаються всі віртуальні машини. Рівень оркестрації на основі кластера RKE2 відповідає за виконання, масштабування та самовідновлення контейнеризованого навантаження. Рівень даних на основі кластера Galera забезпечує відмовостійке зберігання стану застосунку. Два наскрізні рівні доповнюють цю вертикаль: рівень спостережуваності (Prometheus, Grafana, Loki) збирає телеметрію з усіх нижчих рівнів, а рівень доставки та самообслуговування (IaC, CI/CD) автоматизує створення та зміну інфраструктури. Принциповим є те, що відмовостійкість реалізується на кожному рівні окремо: відмова одного вузла на рівні віртуалізації, оркестрації чи даних не призводить до відмови системи в цілому.

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		40

Разом з тим, у межах поточної ітерації розробки повноцінне автоматизоване розгортання (деплой) самого інфраструктурного базису безпосередньо через централізовані конвеєри доставки (CI/CD pipelines) не було остаточно реалізовано. На даному етапі управління конфігурацією та запуск сценаріїв IaC (інструментів Terraform та Ansible) здійснюється у контрольованому режимі безпосередньо з робочої станції адміністратора. Зазначене обмеження архітектури розглядається як свідомий етап розробки та пріоритетний об'єкт для подальшого вдосконалення й модернізації системи в майбутньому.

Для реалізації описаної архітектури спроектовано топологію з віртуальних машин, розгорнутих на Proxmox VE. Рівень оркестрації утворюють три керувальні вузли (RKE2 server) із вбудованим сховищем etcd та два робочі вузли (RKE2 agent). Непарна кількість керувальних вузлів є обов'язковою умовою, оскільки розподілене сховище etcd потребує кворуму для збереження узгодженості, а конфігурація з трьох вузлів забезпечує високодоступний рівень керування, здатний витримати відмову одного з них [11]. Рівень даних утворюють три вузли Galera; вибір саме трьох вузлів зумовлений вимогою кворуму непарної кількості членів (щонайменше трьох) для запобігання ситуації «розщеплення мозку» [12]. Окремий вузол виконує роль балансувальника навантаження (на основі HAProxy) — він надає єдину точку доступу до API Kubernetes та до кластера Galera — і водночас слугує керувальним вузлом, з якого виконуються інструменти автоматизації.

Окремим проєктним рішенням є винесення кластера Galera за межі кластера оркестрації, на виділені віртуальні машини. Хоча технічно можливе розгортання бази даних усередині Kubernetes, винесений варіант обрано свідомо: він спрощує операційну модель, уникаючи складної взаємодії між механізмами забезпечення стану в Kubernetes (StatefulSet, постійні томи) та внутрішнім кворумом Galera, і забезпечує чіткіший поділ відповідальності між рівнем оркестрації та рівнем даних. Контейнеризований додаток, розгорнутий в RKE2, звертається до бази даних як до зовнішнього сервісу через точку доступу балансувальника. Спроектована топологія повністю задовольняє вимогу відмовостійкості з

технічного завдання, оскільки витримує відмову щонайменше одного вузла на кожному з критичних рівнів, і слугує основою для детального проектування підсистем у наступних підрозділах [11].

Запропонований поділ на функціональні рівні потребує докладнішого обґрунтування, оскільки саме він визначає всю подальшу логіку проектування. Принцип розмежування рівнів спирається на ідею ізоляції відмов: кожен рівень має власний, незалежний механізм забезпечення відмовостійкості, тому відмова на одному рівні не поширюється на інші. Рівень віртуалізації забезпечує відмовостійкість через можливість перерозподілу віртуальних машин між фізичними ресурсами; рівень оркестрації — через кворум etcd та механізм перепланування подів; рівень даних — через синхронну реплікацію та кворум Galera. Така багаторівнева ізоляція відповідає принципу ешелонованості, за яким система не покладається на жоден єдиний механізм захисту, а вибудовує кілька незалежних рубежів. Важливою перевагою цього підходу є також модульність: кожен рівень може бути спроектований, реалізований та перевірений окремо, що спрощує як початкове розгортання, так і подальше обслуговування та діагностику несправностей [1].

Профіль цільового навантаження, визначений вище як вебзастосунок зі станом, має кілька наслідків для проектування, які варто розглянути докладніше. Найсуттєвішим є те, що застосунок зберігає стан у зовнішній реляційній базі даних, а не локально, — це означає, що самі екземпляри застосунку є фактично без стану (stateless) і можуть вільно створюватися, знищуватися та переміщуватися між вузлами без втрати даних. Ця властивість є визначальною для відмовостійкості на рівні навантаження: оскільки жоден екземпляр не зберігає унікальних даних, відмова окремого пода не призводить до втрати інформації, а механізм оркестрації може просто створити новий екземпляр на справному вузлі. Саме завдяки чіткому відділенню стану (винесеного в кластер Galera) від обчислень (контейнеризованого застосунку) досягається можливість горизонтального масштабування навантаження — додавання нових екземплярів для обслуговування більшої кількості запитів без зміни архітектури. Винятком є

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		42

завантажені користувачами файли, які потребують окремого механізму збереження стану, що враховується при проектуванні підсистеми зберігання.

Окремо слід обґрунтувати кількісні параметри топології, оскільки вибір саме такої кількості вузлів на кожному рівні не є довільним [11]. Конфігурація з трьох керувальних вузлів рівня оркестрації є мінімальною, що забезпечує високу доступність: розподілене сховище etcd використовує алгоритм консенсусу, який потребує згоди більшості членів для підтвердження запису, тому для збереження працездатності кластера має бути доступною більшість вузлів. За трьох вузлів більшість становить два, отже, кластер витримує відмову одного вузла; за двох вузлів більшість також становить два, тому відмова будь-якого з них вже унеможлиблює досягнення кворуму — саме тому два вузли не забезпечують відмовостійкості, а потрібно щонайменше три.

Аналогічна логіка кворуму застосовується й до кластера даних Galera, де непарна кількість вузлів запобігає ситуації рівного розподілу голосів при розщепленні мережі. Два робочі вузли рівня оркестрації обрано як мінімальну конфігурацію, що дає змогу продемонструвати перепланування навантаження при відмові одного з них, зберігаючи при цьому помірне споживання ресурсів, прийнятне для модельної реалізації.

Рішення про винесення кластера даних за межі оркестрації, згадане вище, заслуговує на додаткове пояснення з огляду на його неочевидність. У сучасній практиці існує тенденція до розміщення баз даних усередині Kubernetes за допомогою спеціалізованих операторів, що автоматизують керування їх життєвим циклом. Проте для цієї роботи свідомо обрано класичніший підхід із розміщенням бази даних на виділених віртуальних машинах, і це рішення має кілька підстав.

По-перше, воно усуває складну взаємодію двох незалежних механізмів забезпечення кворуму — внутрішнього кворуму Galera та механізмів забезпечення стану Kubernetes, — що могли б конфліктувати при одночасній відмові.

По-друге, винесена база даних має передбачуваніший профіль продуктивності, оскільки не конкурує за ресурси з іншим навантаженням кластера.

По-третє, такий поділ відповідає принципу чіткого розмежування відповідальності між рівнями, що спрощує діагностику: проблеми рівня даних локалізуються на відповідних вузлах і не змішуються з проблемами рівня оркестрації. Цей підхід узгоджується із загальною спрямованістю роботи на надійність та операційну прозорість, навіть ціною певної відмови від автоматизації, яку могло б надати розміщення бази даних усередині кластера.

2.2 Проектування середовища віртуалізації на Proxmox VE

Платформа віртуалізації Proxmox VE утворює фундаментальний рівень спроектованої інфраструктури, на якому розгортаються всі віртуальні машини, визначені топологією в підрозділі 2.1. Налаштування віртуальних машин описані в таблиці 2.1. У цьому підрозділі проектується конфігурація хоста віртуалізації, розподіл обчислювальних ресурсів між віртуальними машинами, підсистеми зберігання та мережі, а також механізм підготовки шаблонів, що забезпечує відтворюваність середовища. Спроектоване середовище віртуалізації на основі Proxmox VE зображено на рис. 2.1.

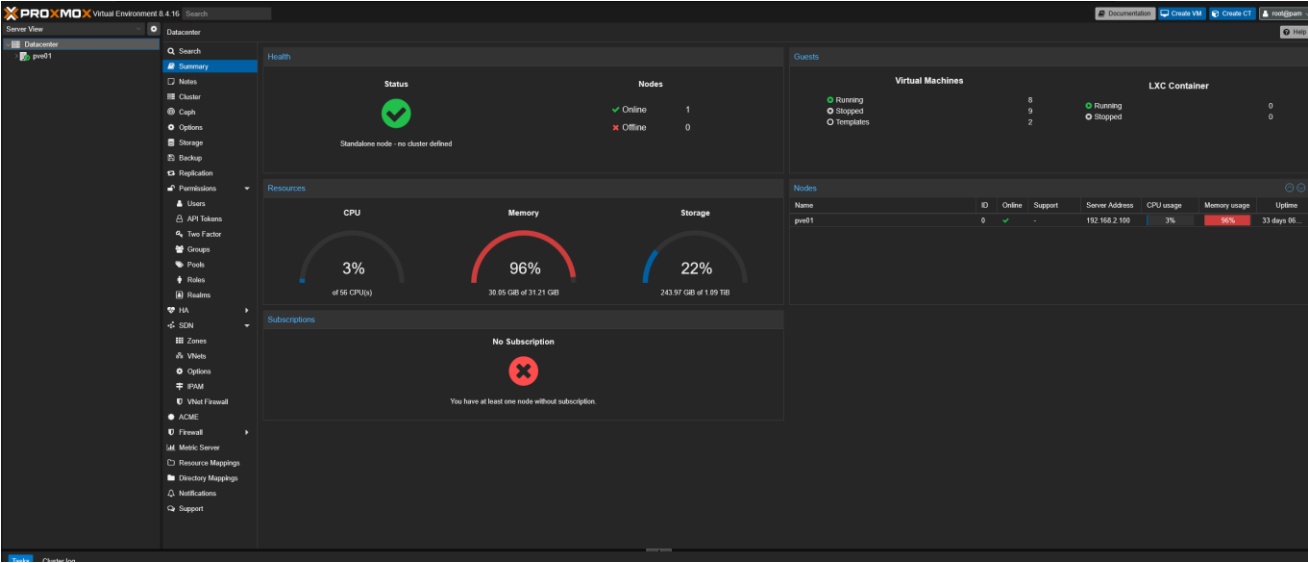


Рисунок 2.1 — Проектування середовища віртуалізації на основі Proxmox VE

Таблиця 2.1

Розподіл обчислювальних ресурсів між віртуальними машинами

Роль ВМ	К-сть	vCPU	ОЗП, ГБ	Диск, ГБ
RKE2 Master	3	8	8	50
RKE2 Worker	2	8	4	100
Galera Cluster	3	8	8	65
HAProxy (балансувальник)	1	2	2	30
Разом	9	58	62	475

Для розгортання модельної інфраструктури передбачено один фізичний вузол під керуванням Proxmox VE. Слід окремо зазначити, що використання єдиного хоста є свідомим обмеженням саме модельної реалізації, продиктованим наявними апаратними ресурсами, а не архітектурним недоліком: рівень віртуалізації залишається єдиним рівнем, на якому в моделі не реалізується апаратна відмовостійкість. У промисловому середовищі це обмеження усувається об'єднанням кількох хостів у кластер Proxmox (на основі Corosync) з увімкненим менеджером високої доступності, що забезпечує автоматичне перенесення віртуальних машин у разі відмови фізичного вузла [13]. Оскільки методологічний фокус роботи зосереджено на відмовостійкості рівнів оркестрації та даних, спроектована конфігурація ВМ навмисно зберігає сумісність із таким кластерним розгортанням без зміни самих машин.

Вибір Proxmox VE як платформи віртуалізації зумовлений сукупністю його характеристик, що відповідають вимогам роботи. Proxmox VE є відкритою платформою віртуалізації корпоративного класу, побудованою на основі операційної системи Debian, що поєднує два механізми віртуалізації: апаратну віртуалізацію на основі KVM для повноцінних віртуальних машин та контейнеризацію на основі LXC для легших системних контейнерів [13]. Для цієї роботи використовується саме механізм KVM, оскільки вузли інфраструктури потребують повної ізоляції на рівні ядра операційної системи та можливості запуску власних ядер, що є необхідним для коректної роботи механізмів

контейнеризації Kubernetes усередині віртуальних машин. Істотною перевагою Proxmox VE є наявність єдиного вебінтерфейсу керування та програмного інтерфейсу (API), через який здійснюється автоматизоване створення віртуальних машин засобами інфраструктури як код, що буде розглянуто в підрозділі 2.3.

Розподіл обчислювальних ресурсів між віртуальними машинами, наведений у табл. 2.1, відображає диференційований підхід до кожної ролі відповідно до її функціонального призначення. Керувальним вузлом кластера оркестрації виділено по 8 віртуальних процесорних ядер та 8 ГБ оперативної пам'яті, оскільки вони одночасно обслуговують програмний інтерфейс Kubernetes, розподілене сховище etcd та системні компоненти кластера, чутливі до затримок. Робочим вузлом, призначеним для виконання прикладного навантаження, виділено по 8 ядер та 4 ГБ пам'яті, що відображає їх роль як обчислювальних ресурсів із меншими вимогами до пам'яті. Вузлом кластера даних виділено по 8 ядер та 8 ГБ пам'яті з огляду на ресурсоемність операцій реляційної бази даних та потребу кешування даних в оперативній пам'яті. Натомість вузол балансувальника навантаження, що виконує виключно проксіювання мережеских з'єднань без збереження стану, потребує мінімальних ресурсів — 2 ядер та 2 ГБ пам'яті. Сумарно інфраструктура використовує 58 віртуальних процесорних ядер, 62 ГБ оперативної пам'яті та 475 ГБ дискового простору, що є прийнятним для розгортання на одному фізичному хості з достатніми ресурсами.

Підсистема зберігання спроектована з урахуванням різних вимог компонентів інфраструктури до дискового простору. Для вузлів кластера даних передбачено окремий віртуальний диск, відмінний від системного, що призначається безпосередньо під дані бази даних. Таке розділення має кілька переваг: воно ізолює дискове навантаження бази даних від операцій системного диска, спрощує адміністрування та резервне копіювання даних, а також дозволяє керувати обсягом сховища бази даних незалежно від системного розділу. На цьому окремому диску засобами менеджера логічних томів (LVM) створюється логічний том, що форматується у файлову систему XFS — обрану через її продуктивність при роботі з великою кількістю операцій введення-виведення,

характерних для навантаження бази даних. Решта вузлів використовує єдиний системний диск, оскільки не має специфічних вимог до окремого сховища стану.

Підсистема мережі побудована на основі мостового з'єднання (bridge) Proxmox VE, що об'єднує всі віртуальні машини в єдину логічну мережу та надає їм доступ до фізичної мережі через мережевий міст хоста віртуалізації. Усім віртуальним машинам призначено статичні IP-адреси в межах єдиної підмережі, що є необхідною умовою для коректної роботи кластерних механізмів: вузли etcd, Galera та компоненти Kubernetes покладаються на стабільність і передбачуваність мережевих адрес для підтримання членства в кластері. Статична адресація задається централізовано на етапі розгортання через механізм cloud-init, що унеможливорює розходження мережевої конфігурації та забезпечує відтворюваність. Використання єдиної пласкої підмережі є свідомим спрощенням для модельної реалізації; у промисловому середовищі мережу зазвичай сегментують на окремі підмережі за функціональним призначенням (мережа керування, мережа даних, мережа зберігання) для підвищення безпеки та ізоляції трафіку.

Ключовим механізмом, що забезпечує відтворюваність середовища віртуалізації, є використання шаблону віртуальної машини на основі хмарного образу (cloud image) операційної системи Rocky Linux 9 із підтримкою cloud-init. Замість встановлення операційної системи на кожен віртуальну машину окремо — процесу тривалого та схильного до розбіжностей — підготовлюється єдиний еталонний шаблон, з якого надалі клонуються всі віртуальні машини інфраструктури. Механізм cloud-init дозволяє при першому запуску кожної клонованої машини автоматично застосувати індивідуальну конфігурацію: задати ім'я вузла, мережеві параметри, облікові записи та початкові налаштування, не змінюючи самого шаблону.

Такий підхід має дві принципові переваги. По-перше, він гарантує, що всі віртуальні машини створюються з ідентичної базової основи, що усуває дрейф конфігурації на рівні операційної системи ще до початку її налаштування. По-друге, він повністю узгоджується з підходом «інфраструктура як код»: шаблон є

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		47

незмінним артефактом, а вся варіативність винесена в декларативний опис, що обробляється засобами автоматизації. Вибір Rocky Linux 9 як базової операційної системи зумовлений її сумісністю з вимогами захищеного дистрибутива RKE2 та підтримкою механізму примусового контролю доступу SELinux, що є важливим для забезпечення захищеності інфраструктури.

Слід ще раз наголосити, що єдиним свідомим обмеженням цього рівня в межах модельної реалізації є відсутність апаратної відмовостійкості через використання одного фізичного хоста, що, як зазначено вище, усувається в промисловому середовищі кластеризацією Proxmox без зміни самих віртуальних машин. На цьому спроектованому фундаменті розгортаються рівні оркестрації та даних, детальне проєктування яких наведено в наступних підрозділах.

2.3 Проєктування підсистеми IaC та конфігураційного менеджменту

Підсистема автоматизації є наскрізним рівнем інфраструктури, що відповідає за створення та конфігурування всіх інших рівнів у відтворюваний спосіб. Згідно з технічним завданням та принципом відтворюваності, жодна дія з налаштування середовища не виконується вручну — натомість усі стани описуються кодом. Порівняння технологій описано в таблиці 2.2.

Таблиця 2.2

Порівняння інструментів розгортання інфраструктури

Критерій	Terraform	OpenTofu	Pulumi
Розробник / супровід	HashiCorp (IBM)	Linux Foundation	Pulumi Corp.
Ліцензія	BSL 1.1	MPL 2.0 (відкрита)	Apache 2.0 (відкрита)
Мова опису	HCL	HCL	Python, TypeScript, Go, C#
Модель	Декларативна	Декларативна	Декларативна + імперативна
Сумісність із Terraform	—	Повна (форк 1.5/1.6)	Через конвертер

Критерій	Terraform	OpenTofu	Pulumi
Кількість провайдерів	4800+	3900+	через місток Terraform
Зберігання стану	Файл .tfstate	Файл .tfstate	Pulumi Cloud
Шифрування стану (вбудоване)	Ні (лише в HCP)	Так (з v1.7)	Так

Усі три інструменти реалізують декларативний підхід до розгортання інфраструктури, проте різняться за ліцензійною моделлю, мовою опису та зрілістю екосистеми. Pulumi вирізняється використанням мов загального призначення (Python, TypeScript, Go) замість спеціалізованої мови, що є перевагою для команд із досвідом розробки, проте має менший набір власних провайдерів.

OpenTofu є відкритим відгалуженням (форком) Terraform під управлінням Linux Foundation, повністю сумісним з ним на рівні мови HCL та формату стану, що виник як відповідь спільноти на зміну ліцензії Terraform на обмежувальну BSL у 2023 році. Для цієї роботи обрано Terraform з огляду на найбільшу та найзрілішу екосистему провайдерів — зокрема наявність якісного провайдера для платформи Proxmox VE (bpg/proxmox), використаного в реалізації, — а також найширшу доступність документації та прикладів. Слід зазначити, що завдяки повній сумісності OpenTofu подальша міграція на цей повністю відкритий інструмент за потреби не вимагатиме змін у написаному коді, що додатково знижує ризик прив'язки до постачальника.

Підсистема керування конфігурацією реалізована засобами Ansible — інструмента, що доповнює Terraform у частині приведення вже створених вузлів до цільового стану. На відміну від інструментів розгортання інфраструктури, розглянутих вище, засоби керування конфігурацією орієнтовані на встановлення

та налаштування програмного забезпечення на наявних системах. Порівняння основних представників цієї категорії наведено в таблиці 2.3.

Таблиця 2.3

Порівняння інструментів керування конфігурацією

Критерій	Ansible	Puppet	SaltStack
Агент на хості	Не потрібен (SSH)	Потрібен	Потрібен
Мова опису	YAML	DSL (Ruby)	YAML або Python
Модель	Push	Pull	Push та Pull
Ідемпотентність	Так	Так	Так

Для керування конфігурацією обрано Ansible з огляду на сукупність його характеристик, що найкраще відповідають вимогам проєктованої інфраструктури: відсутність необхідності встановлення агентів на керованих вузлах спрощує операційну модель і зменшує поверхню атаки, низький поріг входження на основі YAML узгоджується з принципом зниження когнітивного навантаження, а push-модель взаємодії через SSH відповідає архітектурі розгортання з робочої станції адміністратора. На відміну від Puppet, що вимагає розгортання агентів і знання власної DSL-мови, та SaltStack, що також потребує інфраструктури агентів, Ansible дає змогу розпочати автоматизацію без попередньої підготовки керованих вузлів, що є критично важливим для відтворюваного розгортання з нуля.

Відповідно до обґрунтованого в підрозділі 1.4 поділу між розгортанням ресурсів і керуванням їх конфігурацією, підсистема автоматизації будується на двох інструментах із чітко розмежованими зонами відповідальності. Terraform виконує розгортання на платформі Proxmox VE шляхом клонування підготовленого шаблону, задаючи їх обчислювальні ресурси, диски, мережеві параметри та початкову конфігурацію [15]. Ansible виконує конфігураційний менеджмент — приводить уже створені машини до цільового стану, встановлюючи та налаштовуючи на них програмне забезпечення (RKE2, MariaDB Galera та супутні компоненти) [16].

Взаємодія між цими інструментами здійснюється для платформи Proxmox через її програмний інтерфейс (REST API) із застосуванням стороннього провайдера [15]. Взаємодію Terraform та Ansible у процесі розгортання зображено на рис. 2.2.

```

1 terraform {
2   required_version = ">= 1.5.0"
3   required_providers {
4     proxmox = {
5       source = "bpg/proxmox"
6       version = "0.66.1"
7     }
8     local = {
9       source = "hashicorp/local"
10      version = "~> 2.0"
11    }
12  }
13 }
14
15 provider "proxmox" {
16   endpoint = var.proxmox_endpoint
17   api_token = var.proxmox_api_token
18   insecure = var.proxmox_insecure
19 }

```

Рисунок 2.2 — Взаємодія Terraform та Ansible у процесі розгортання

Створення віртуальних машин описується декларативно з використанням механізму лічильника (count), розгортання зображено на рис. 2.3.

```

resource "proxmox_virtual_environment_vm" "master_nodes" {
  count = var.master_count
  name = "cl01-m0${count.index + 1}"
  node_name = var.target_node
  vm_id = var.master_vm_id_start + count.index

  clone {
    vm_id = var.clone_vm_id
    full = true
  }

  cpu {
    cores = 8
    type = "host"
  }

  memory { dedicated = 8192 }

  disk {
    datastore_id = var.datastore_id
    interface = "scsi0"
    size = 50
  }

  network_device { bridge = "vmbbr0" }
}

```

Рисунок 2.3 — Ілюстрація створення віртуальних машин з Terraform

Такий опис є прямим практичним протиставленням ручному конфігуруванню з підрозділу 1.2: один параметризований фрагмент

детерміновано породжує весь набір вузлів із послідовними іменами та адресами, унеможливлуючи дрейф конфігурації між ними.

Окремої уваги потребує спосіб зберігання стану Terraform, оскільки цей файл відіграє центральну роль у моделі «інфраструктура як код». Стан Terraform (terraform state) є файлом, що містить відповідність між описаними в коді ресурсами та їх фактичними примірниками в реальній інфраструктурі, включно з їх ідентифікаторами, параметрами та взаємозв'язками.

Саме на підставі цього файлу Terraform визначає, які зміни потрібно внести при кожному застосуванні конфігурації — що створити, змінити чи видалити. У межах модельної реалізації цієї роботи стан зберігається локально, у файлі на робочій станції інженера, що є найпростішим і цілком прийнятним варіантом для одноосібної роботи в лабораторному середовищі, де відсутня потреба в одночасному доступі кількох інженерів до однієї інфраструктури.

Проте для промислової експлуатації локальне зберігання стану є неприйнятним з кількох причин. По-перше, локальний файл стану є єдиною точкою відмови: його втрата або пошкодження унеможливорює подальше коректне керування інфраструктурою засобами Terraform, оскільки інструмент втрачає відомості про відповідність коду й реальних ресурсів. По-друге, за командної роботи кількох інженерів локальний стан унеможливорює узгоджене внесення змін і може призвести до конфліктів або пошкодження інфраструктури при одночасному застосуванні.

Тому в промисловому середовищі стан зберігається у віддаленому сховищі (remote backend) — наприклад, в об'єктному сховищі з підтримкою версіонування, — що забезпечує централізований доступ, автоматичне резервне копіювання та механізм блокування (state locking), який запобігає одночасній зміні стану кількома інженерами.

Крім того, оскільки файл стану може містити чутливі дані у відкритому вигляді, віддалене сховище забезпечує їх шифрування та контроль доступу. Застосування віддаленого зберігання стану з резервним копіюванням визначено

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		52

як необхідний напрям доопрацювання інфраструктури при переході від модельної реалізації до промислового впровадження.

Окремим проєктним рішенням є спосіб передавання інформації від етапу розгортання (Terraform) до етапу конфігурування (Ansible). Оскільки IP-адреси та імена створених віртуальних машин стають відомі лише після їх фактичного створення, статичне формування переліку вузлів для Ansible є неможливим.

Тому передбачено динамічну генерацію інвентарного файлу Ansible [16] засобами самого Terraform: після створення віртуальних машин Terraform формує файл з їх фактичними адресами, групує вузли за ролями (masters, workers).

У такий спосіб джерелом істини про склад інфраструктури залишається стан Terraform, а Ansible лише споживає згенерований ним перелік, що усуває ручне ведення інвентарю та можливі розбіжності між фактичним і описаним складом кластера.

Інструменти автоматизації виконуються не з вузла всередині інфраструктури, а з робочої станції інженера. Відповідно до цього проєктується модель зберігання стану та доступу. Описи інфраструктури зберігаються у системі контролю версій, що забезпечує їх версіонування та можливість відкату. Стан Terraform (terraform state), який містить відповідність між описаними та реально створеними ресурсами, зберігається окремо від керованого середовища; доступ Terraform до платформи віртуалізації здійснюється за токеном програмного інтерфейсу, а доступ Ansible до вузлів — за протоколом SSH [16].

На поточному етапі реалізації проєкту декларативні описи та репозиторії керування інфраструктурним кодом розміщено у репозиторії на платформі GitLab. З метою підвищення загального рівня безпеки, конфіденційності конфігураційних даних та забезпечення повної автономності системи, вагомим об'єктом для подальших покращень у майбутньому є розгортання власної локальної платформи контролю версій (на кшталт GitLab Self-Hosted або приватного Git-сервера) разом із виділеними внутрішніми агентами автоматизації (self-hosted runners) безпосередньо всередині проєктуємого периметра. Таке рішення дозволить

повністю замкнути весь життєвий цикл конвеєрів доставки та управління інфраструктурою в межах приватного контуру безпеки компанії.

Така модель узгоджується з природою підходу «інфраструктура як код»: керованість середовища не залежить від доступності окремого внутрішнього вузла, а сама інфраструктура може бути повністю відтворена з її опису та стану.

Слід зазначити, що в межах кваліфікаційної роботи застосовано низку спрощень, але неприпустимих у промисловій експлуатації: автентифікація на вузлах за паролем замість ключів, зберігання облікових даних у відкритому вигляді та вимкнення перевірки відбитків SSH. У промисловому середовищі ці аспекти вирішуються застосуванням систем керування секретами та віддаленого сховища стану зі спільним доступом і блокуванням. Детальне проєктування рівнів оркестрації та даних, що розгортаються за допомогою описаної підсистеми, наведено в наступних підрозділах.

2.4 Проєктування кластера оркестрації RKE2

Кластер оркестрації RKE2 є рівнем, що відповідає за виконання, масштабування та самовідновлення контейнеризованого навантаження. Згідно з обґрунтуванням у підрозділі 1.7, обрано саме дистрибутив RKE2 з огляду на його захищеність за замовчуванням (відповідність CIS Kubernetes Benchmark, FIPS 140-2, SELinux), що узгоджується зі специфікою системи керування аудитами [11]. У цьому підрозділі проєктуються топологія та схема найменування вузлів кластера.

Кластер розгортається на п'яти віртуальних машинах: трьох керувальних вузлах та двох робочих. Для всіх вузлів інфраструктури запроваджено єдину схему найменування, що відображає їх належність до кластера та роль, а також узгоджені з нею ідентифікатори віртуальних машин (VMID) та IP-адреси. Найменування побудовано за шаблоном [кластер][NN]-[роль][NN], де роль кодується літерою (m — master, w — worker, n — node бази даних). Така схема забезпечує однозначність ідентифікації вузла за його іменем та масштабованість

найменування при розширенні інфраструктури. Повну схему наведено в таблиці 2.4.

Таблиця 2.4

Схема найменування та адресації вузлів інфраструктури

Ім'я вузла	Роль	VMID	IP-адреса
gl01-n01	Galera (вузол БД)	701	192.168.2.181
gl01-n02	Galera (вузол БД)	702	192.168.2.182
gl01-n03	Galera (вузол БД)	703	192.168.2.183
cl01-m01	RKE2 server (control plane)	706	192.168.2.186
cl01-m02	RKE2 server (control plane)	707	192.168.2.187
cl01-m03	RKE2 server (control plane)	708	192.168.2.188
cl01-w01	RKE2 agent (worker)	711	192.168.2.191
cl01-w02	RKE2 agent (worker)	712	192.168.2.192
lb01-n01	HAProxy (балансувальник)	720	192.168.2.180

Усі вузли розміщуються в єдиному мережному сегменті 192.168.2.0/24 зі шлюзом 192.168.2.1, що відповідає мережевій моделі, закладеній у підрозділі 2.2.

Високодоступність рівня керування забезпечується розгортанням трьох керувальних вузлів зі вбудованим розподіленим сховищем etcd, у якому зберігається весь стан кластера Kubernetes. Сховище etcd використовує алгоритм консенсусу, що потребує наявності кворуму — більшості вузлів — для підтвердження запису. За конфігурації з трьох вузлів кворум становить два, завдяки чому кластер зберігає працездатність і узгодженість стану в разі відмови одного керувального вузла; саме це обґрунтовує вибір непарної кількості вузлів, закладений ще в підрозділі 2.1 [11].

Процес формування кластера спроектовано за моделлю «ініціалізація — приєднання». Перший керувальний вузол (cl01-m01) ініціалізує кластер і генерує маркер приєднання (node-token); решта керувальних вузлів та робочі вузли приєднуються до кластера, використовуючи цей маркер.

Оскільки звернення до програмного інтерфейсу кластера (Kubernetes API) можливе до будь-якого з трьох керувальних вузлів, для надання клієнтам єдиної

відмовостійкої точки доступу передбачено вузол-балансувальник (lb01-n01) на основі HAProxy, що розподіляє запити між справними керувальними вузлами. Це усуває залежність доступу до кластера від конкретного керувального вузла й доповнює відмовостійкість на рівні самого сховища etcd.

Внутрішня мережа кластера будується мережевим додатком (CNI), що постачається у складі RKE2 за замовчуванням — Canal, який поєднує Flannel для організації мережі подів та Calico для підтримки мережесхемних політик [11]. Цей рівень створює віртуальну мережу подів і служб, ізольовану від зовнішньої мережі вузлів. Зовнішній доступ до застосунків, розгорнутих у кластері, забезпечується контролером входу (Ingress) на основі Traefik, що постачається у складі дистрибутива RKE2 за замовчуванням; він маршрутизує зовнішні HTTP/HTTPS-запити до відповідних служб усередині кластера на підставі імені вузла призначення. Автоматизацію керування TLS-сертифікатами для цих служб покладено на компонент cert-manager.

Окремої уваги заслуговує вибір мережевого додатка Canal як CNI-рішення для кластера. Canal є композитним рішенням, що поєднує два окремі компоненти з різними зонами відповідальності: Flannel забезпечує базову мережеву зв'язність між подами на різних вузлах через накладену мережу, тоді як Calico реалізує підтримку мережесхемних політик, що дають змогу декларативно обмежувати мережеву взаємодію між компонентами на рівні кластера.

Слід зазначити, що в межах модельної реалізації для зіставлення доменних імен із IP-адресами застосовано службу nip.io, що дає змогу використовувати імена вигляду <сервіс>.<IP>.nip.io без розгортання власного DNS-сервера. Це є прийнятним лабораторним спрощенням; у промисловому середовищі замість нього застосовується повноцінна система доменних імен. Спроектований у такий спосіб кластер є середовищем виконання для контейнеризованого навантаження, а шар керування ним та механізми самообслуговування розглядаються в підрозділі 2.5.

2.5 Проектування відмовостійкого рівня даних на основі Galera

Рівень даних є найкритичнішим з погляду надійності, оскільки саме він зберігає стан системи керування аудитами, втрата або пошкодження якого є неприпустимими. Цей рівень будується на основі кластера MariaDB Galera, що забезпечує синхронну multi-master реплікацію. У цьому підрозділі проектується топологія кластера даних, стратегія доступу до нього через балансувальник та організація зберігання даних на вузлах.

Рівень даних утворюють три виділені віртуальні машини (gl01-n01–gl01-n03), розміщені поза кластером оркестрації згідно з рішенням, обґрунтованим у підрозділі 2.1. Galera забезпечує синхронну multi-master реплікацію, за якої кожен вузол містить ідентичну копію бази даних, а транзакція підтверджується лише після її сертифікації на всіх вузлах; це гарантує узгодженість даних та відсутність затримки реплікації.

Вибір саме трьох вузлів зумовлений вимогою кворуму: для запобігання ситуації «розщеплення мозку» (split-brain), коли роз'єднані частини кластера незалежно приймають записи, кластер повинен містити непарну кількість вузлів, щонайменше три [12].

За такої конфігурації відмова одного вузла не порушує роботу кластера — решта два зберігають кворум (більшість) і продовжують обслуговування, а відновлений вузол автоматично ресинхронізується з кластером.

Формування кластера спроектовано за моделлю «ініціалізація — приєднання», аналогічно до рівня оркестрації: перший вузол ініціалізує кластер (виконує початкове завантаження, bootstrap), після чого решта вузлів приєднуються до нього й отримують повну копію даних механізмом перенесення стану (SST). Узгодженість реплікації на рівні рушія бази даних забезпечується обов'язковими для Galera параметрами — рядковим форматом журналу та відповідним режимом генерації автоінкрементних значень, що запобігає конфліктам при паралельних операціях.

Доступ контейнеризованого навантаження до кластера даних здійснюється через той самий вузол-балансувальник (lb01-n01), що обслуговує і програмний

інтерфейс кластера оркестрації, але через окремий незалежний фронтенд на порту MariaDB (3306). Попри те що Galera технічно дозволяє запис у будь-який вузол, для рівня даних свідомо обрано стратегію «активний вузол — резерв» (active-backup), а не рівномірний розподіл (roundrobin). Це проєктне рішення зумовлене специфікою multi-master реплікації: одночасний запис у різні вузли підвищує ймовірність конфліктів сертифікації транзакцій та пов'язаних із ними взаємних блокувань. За стратегії active-backup балансувальник спрямовує всі з'єднання на один обраний вузол, а на резервні перемикається лише в разі його відмови, що поєднує відмовостійкість із уникненням конфліктів запису.

Для зберігання даних на вузлах кластера спроектовано окремий рівень логічних томів, відокремлений від кореневої файлової системи. На кожному вузлі під дані бази відводиться окремий віртуальний диск, на якому засобами менеджера логічних томів (LVM) створюється виділений том, монтований у каталог даних MariaDB. Таке відокремлення має дві переваги: воно ізолює дані бази від системного розділу, запобігаючи впливу їх зростання на роботу операційної системи, та спрощує подальше масштабування дискового простору без переналаштування системи. Як файлову систему для цього тому обрано XFS, що є рекомендованою для навантажень бази даних з огляду на її продуктивність при роботі з великими файлами та паралельними операціями введення-виведення.

Вибір файлової системи для тому даних бази є окремим проєктним рішенням, що безпосередньо впливає на продуктивність і надійність рівня даних. Порівняння трьох основних файлових систем, що розглядалися як кандидати, наведено в таблиці 2.5.

Таблиця 2.5

Порівняння файлових систем

Критерій	XFS	ZFS (OpenZFS)	ext4
Ліміт розміру одного файлу	8 ЕБ (Екзабайт)	16 ЕБ (Екзабайт)	16 ТБ (Терабайт) — критичне обмеження для великих БД

Критерій	XFS	ZFS (OpenZFS)	ext4
Контроль цілісності даних	CRC32с тільки для метаданих	End-to-End верифікація метаданих та даних (Fletcher4 / SHA-256)	Контрольні суми тільки для журналу
Механізм запису на диск	<i>In-place</i>	<i>Copy-on-Write</i> (CoW)	<i>In-place</i>
Паралелізація вводу-виводу (I/O)	Нативна багатопотоковість через незалежні групи розподілу (<i>Allocation Groups</i>)	Паралелізація на рівні пулів пристроїв (<i>vdevs</i>)	Обмежена — виникають блокування на рівні суперблока та журналу
Специфіка для InnoDB (MariaDB)	Підтримка асинхронного та прямого виводу (<i>Direct I/O</i>) в обхід кешу ОС	Вимагає ручного налаштування <i>recordsize=16k</i> для уникнення <i>Write Amplification</i>	Використовує відкладене виділення пам'яті (<i>Delayed Allocation</i>)

Механізм синхронної реплікації, на якому ґрунтується робота кластера даних, доцільно розглянути докладніше, оскільки саме він визначає надійність зберігання стану. За такої реплікації кожен запис, перш ніж бути остаточно прийнятим, узгоджується між усіма вузлами кластера: транзакція підтверджується лише тоді, коли всі вузли засвідчують можливість її застосування без конфлікту з іншими одночасними змінами. Завдяки цьому всі вузли кластера в будь-який момент містять однаковий набір даних, а відставання між ними фактично відсутнє. Для системи керування аудитами ця властивість має принципове значення, оскільки гарантує, що зафіксований результат перевірки буде однаково й негайно доступним незалежно від того, до якого вузла кластера звернеться застосунок. Це унеможливорює ситуацію, коли застосунок отримав би застарілі дані, що для системи, призначеної забезпечувати достовірність і підзвітність, є неприпустимим.

Окремо слід розглянути поведінку кластера під час відновлення вузла після відмови, оскільки саме вона визначає здатність системи повертатися до повного рівня надійності без втручання адміністратора. Коли вузол, що тимчасово вибув з роботи, знову стає доступним, він автоматично визначає, наскільки його дані відстали від поточного стану кластера, і отримує від справних вузлів усі зміни, яких йому бракує. Якщо відсутність була нетривалою, передаються лише ці зміни; якщо тривалою — вузол отримує повну копію бази даних. Після завершення цього процесу вузол знову стає повноцінним учасником кластера. Така автоматична поведінка означає, що після усунення причини відмови відновлення відбувається самостійно, без ручних операцій, і кластер повертається до початкового рівня відмовостійкості. Це узгоджується із закладеним у роботі принципом зниження обсягу ручної праці з обслуговування інфраструктури.

Слід також зважати на те, що синхронна реплікація має і зворотний бік, який враховано при проєктуванні. Оскільки кожен запис потребує підтвердження від усіх вузлів кластера, час його виконання дещо зростає порівняно з одиночним сервером бази даних. Для системи керування аудитами цей компроміс є виправданим: характер її роботи передбачає переважання операцій читання над операціями запису та не висуває вимог до миттєвої швидкості запису. Натомість перевага, яку дає синхронна реплікація, — гарантована узгодженість даних та збереження їх цілісності у разі відмови вузла — є для такої системи значно важливішою за незначне збільшення часу запису. Таким чином, обрана модель зберігання даних свідомо віддає перевагу надійності та узгодженості над максимальною швидкістю, що повністю відповідає вимогам, сформульованим у технічному завданні.

Узагальнюючи проєктування рівня даних, слід підкреслити, що всі прийняті рішення підпорядковані єдиній меті — забезпеченню надійного та узгодженого зберігання стану системи. Винесення кластера на окремі віртуальні машини відокремлює дані від обчислювального навантаження, синхронна реплікація між трьома вузлами гарантує їх узгодженість і збереження у разі відмови, стратегія «активний вузол — резерв» на балансувальнику запобігає конфліктам запису, а

окремий дисковий том ізолює дані від системного розділу. Кожне з цих рішень посилює надійність на своєму рівні, а разом вони формують рівень даних, здатний витримати відмову одного вузла без втрати працездатності чи цілісності інформації.

Таким чином, спроектований рівень даних задовольняє вимогу відмовостійкості з технічного завдання на найкритичнішому рівні: він зберігає працездатність і цілісність даних у разі відмови одного вузла бази даних. Разом із рівнями віртуалізації та оркестрації він утворює завершену відмовостійку вертикаль, на якій розгортається цільове навантаження, проєктування доставки якого розглядається в наступному підрозділі.

Саме така багатоаспектна надійність є визначальною для системи керування аудитами.

2.6 Проєктування підсистеми доставки навантаження

Підсистема доставки відповідає за процес перенесення контейнеризованого навантаження до кластера оркестрації у керований, автоматизований та відтворюваний спосіб. Згідно з принципами, закладеними в підрозділах 1.4 та 1.6, доставка проєктується не як разова ручна операція, а як безперервний процес, керований декларативним описом бажаного стану. У цьому підрозділі проєктуються модель контейнеризації навантаження, механізм безперервної доставки на основі підходу GitOps та шар керування кластером.

Відповідно до спрямованості роботи, цільовий застосунок розглядається як готове контейнеризоване навантаження, а не як об'єкт розроблення. Це означає, що застосунок постачається у вигляді стандартного образу контейнера, незмінного та самодостатнього, який містить сам застосунок і всі його залежності. Розгортання такого образу в кластері описується декларативно — у вигляді маніфестів Kubernetes (або їх параметризованого пакета, чарту Helm [20]), що задають бажаний стан навантаження: кількість реплік, споживані ресурси, мережеві служби та параметри підключення до зовнішнього рівня даних. Підключення застосунку до бази даних здійснюється через точку доступу

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		61

балансувальника (lb01-n01:3306), тобто кластер Galera виступає для навантаження зовнішнім сервісом, що узгоджується з рішенням про винесення рівня даних за межі кластера (підрозділ 2.1).

Як модель безперервної доставки обрано підхід GitOps [19], що є розвитком практик CI/CD, розглянутих у підрозділі 1.4. Його суть полягає в тому, що єдиним джерелом істини про бажаний стан кластера є репозиторій системи контролю версій: усі маніфести, що описують навантаження, зберігаються в репозиторії, а спеціалізований агент усередині кластера безперервно зіставляє фактичний стан із описаним і автоматично приводить кластер до бажаного стану. Інструментом реалізації цього підходу обрано ArgoCD — оператор GitOps для Kubernetes [19].

Така модель має принципові переваги, що прямо відповідають закладеним методологіям. По-перше, вона забезпечує декларативність і відтворюваність: стан кластера завжди відповідає вмісту репозиторію, а будь-яка зміна відбувається через зафіксовану в історії версій правку. По-друге, вона усуває ручне втручання в кластер: інженер не виконує команди розгортання вручну, а лише змінює опис у репозиторії, що реалізує принцип зниження обсягу ручної праці (підрозділ 1.5). По-третє, вона спрощує відкат: повернення до попереднього стану зводиться до повернення до попередньої версії опису. У такий спосіб GitOps замикає петлю автоматизації, розпочату інфраструктурою як код: якщо Terraform та Ansible декларативно описують інфраструктуру, то ArgoCD декларативно описує навантаження на ній.

Над кластером оркестрації передбачено шар керування на основі платформи Rancher, що надає централізований інтерфейс для адміністрування кластера, спостереження за його станом, керування доступом (RBAC) та розгортання застосунків. Цей компонент є практичним утіленням ідей Platform Engineering, розглянутих у підрозділі 1.5: він інкапсулює складність безпосередньої взаємодії з програмним інтерфейсом Kubernetes за зрозумілим інтерфейсом, знижуючи когнітивне навантаження на користувачів кластера та надаючи їм керовані можливості самообслуговування [17]. Автоматизацію керування TLS-

сертифікатами для компонентів цього шару покладено на cert-manager, що звільняє адміністратора від ручного випуску та поновлення сертифікатів.

Сукупно підсистема доставки об'єднує три рівні автоматизації в єдиний ланцюг: інфраструктура створюється як код (Terraform, Ansible), навантаження доставляється за моделлю GitOps (ArgoCD), а керування та самообслуговування забезпечуються платформним шаром (Rancher). Детальне проектування механізмів самообслуговування, що спираються на цей шар, розглядається в підрозділі 2.8, а підсистема спостережуваності, яка забезпечує контроль стану всіх описаних рівнів, — у підрозділі 2.7.

Модель контейнеризації навантаження потребує детальнішого розгляду, оскільки саме властивості образу контейнера визначають можливість його надійної доставки. Образ контейнера є незмінним артефактом: після складання його вміст не змінюється, а будь-яке оновлення застосунку призводить до створення нового образу з новою версією, а не до зміни наявного. Ця властивість незмінності є фундаментальною для відтворюваності, оскільки гарантує, що той самий образ, перевірений у тестовому середовищі, буде ідентичним до образу, розгорнутого в робочому середовищі, — усувається ціла категорія проблем, пов'язаних із розбіжностями середовищ. Образ будується за принципом самодостатності: він містить не лише код застосунку, а й усі його залежності — інтерпретатор мови, бібліотеки та системні компоненти, необхідні для роботи. Завдяки цьому вузол кластера, на якому виконується образ, не потребує жодного попереднього налаштування під конкретний застосунок, що відповідає принципу одноманітності вузлів та спрощує горизонтальне масштабування навантаження.

Окремим проєктним аспектом є спосіб параметризації розгортання для різних середовищ. Оскільки той самий образ має використовуватися незмінним, уся варіативність конфігурації — параметри підключення до бази даних, режими роботи застосунку, секретні дані — виносяться за межі образу та передається йому ззовні, через змінні оточення та спеціалізовані об'єкти Kubernetes. Чутливі дані, такі як паролі доступу до бази даних, зберігаються в об'єктах типу Secret, відокремлених від основного опису навантаження, що дає змогу керувати ними з

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		63

урахуванням вимог безпеки та не розкривати їх у відкритому вигляді в маніфестах. Такий поділ конфігурації та коду є усталеною практикою розгортання контейнеризованих застосунків і безпосередньо реалізує принцип, за яким один незмінний артефакт придатний для використання в будь-якому середовищі без перескладання.

Підхід GitOps, обраний як модель безперервної доставки, доцільно розглянути в зіставленні з традиційною моделлю CI/CD, щоб підкреслити його відмінність. У класичній моделі безперервного розгортання система автоматизації (наприклад, конвеєр CI/CD) активно надсилає зміни до кластера ззовні — вона має облікові дані для доступу до кластера та виконує команди розгортання за принципом «доставлення» (push). Натомість у моделі GitOps застосовується протилежний принцип «витягування» (pull): агент, розміщений усередині кластера, самостійно зчитує бажаний стан із репозиторію та застосовує його зсередини. Така інверсія має дві суттєві переваги з погляду безпеки. По-перше, кластер не надає зовнішнім системам облікових даних для доступу до себе, що зменшує поверхню атаки. По-друге, агент безперервно зіставляє фактичний стан кластера з описаним і автоматично усуває будь-які відхилення — як випадкові, так і несанкціоновані; це явище, відоме як самовиправлення (self-healing) конфігурації, гарантує, що фактичний стан кластера ніколи не розходиться з описаним у репозиторії. Для системи, що підлягає вимогам аудиту, ця властивість має додаткову цінність, оскільки репозиторій стає повним і незаперечним журналом усіх змін інфраструктури навантаження.

Шар керування на основі платформи Rancher заслуговує на докладніший розгляд у контексті його ролі в загальній архітектурі. Безпосередня взаємодія з кластером Kubernetes здійснюється через його програмний інтерфейс за допомогою інструмента командного рядка, що вимагає від користувача глибокого розуміння внутрішніх абстракцій кластера та відповідної кваліфікації. Платформа Rancher інкапсулює цю складність, надаючи графічний інтерфейс, через який типові операції — перегляд стану навантаження, аналіз журналів, керування доступом користувачів — виконуються без потреби в глибокому знанні

програмного інтерфейсу. Це знижує поріг входження та зменшує когнітивне навантаження, що є центральною ідеєю Platform Engineering: складність не усувається, а приховується за зрозумілим інтерфейсом, лишаючись доступною для досвідчених користувачів за потреби. Додатково Rancher централізує керування доступом на основі ролей (RBAC), даючи змогу гнучко розмежовувати права різних категорій користувачів — від адміністраторів інфраструктури до користувачів, які мають доступ лише до власного навантаження. Така можливість розмежування доступу є особливо значущою для системи керування аудитами, де принцип найменших привілеїв та контроль доступу є не лише технічною, а й регуляторною вимогою.

Слід також зазначити взаємодоповнювальну роль ArgoCD та Rancher у запроєктованій архітектурі, оскільки ці інструменти вирішують різні, але суміжні завдання. ArgoCD відповідає за безперервну синхронізацію стану навантаження з репозиторієм — тобто за відповідь на питання «що має бути розгорнуто і в якому стані». Rancher натомість відповідає за керування самим кластером та доступом до нього — тобто за питання «хто і як взаємодіє з кластером». Разом вони формують повноцінний платформний шар, у якому декларативна доставка поєднується з керованим самообслуговуванням. Така комбінація узгоджується з концепцією внутрішньої платформи розробника (Internal Developer Platform), розглянутою в підрозділі 1.6: інженер, відповідальний за навантаження, отримує можливість керувати його розгортанням через зміну опису в репозиторії, не звертаючись безпосередньо до адміністратора інфраструктури, тоді як адміністратор зберігає централізований контроль над кластером.

Підсумовуючи проектування підсистеми доставки, її доцільно розглянути як єдиний наскрізний ланцюг автоматизації, що охоплює всі рівні інфраструктури. Цей ланцюг починається на рівні фізичної інфраструктури, де Terraform описує створення віртуальних машин, продовжується на рівні операційної системи, де Ansible приводить ці машини до цільового стану, і завершується на рівні навантаження, де ArgoCD описує застосунки, що виконуються в кластері. Принциповим є те, що на кожному з цих рівнів застосовується той самий базовий

принцип: бажаний стан описується кодом у репозиторії, а відповідний інструмент приводить фактичний стан у відповідність до описаного. Завдяки цій узгодженості вся інфраструктура — від фізичних машин до прикладного навантаження — стає відтворюваною з її опису, версіонованою та простежуваною.

Саме така наскрізна відтворюваність є кінцевою метою застосування методологій DevOps, SRE та Platform Engineering у цій роботі: інфраструктура перестає бути сукупністю вручну налаштованих компонентів і перетворюється на передбачуваний результат виконання коду, що може бути відтворений, перевірений та змінений у керований спосіб. Запроєктована в такий спосіб підсистема доставки повністю відповідає вимогам технічного завдання щодо автоматизованості та відтворюваності й слугує сполучною ланкою між інфраструктурними рівнями та прикладним навантаженням.

2.7 Проєктування підсистеми спостережуваності

Підсистема спостережуваності є наскрізним рівнем, що забезпечує контроль стану всіх інших рівнів інфраструктури та є практичним утіленням підходу SRE, розглянутого в підрозділі 1.5. Без неї неможливе ні кількісне оцінювання надійності, ні своєчасне виявлення відмов, ні перевірка відповідності інфраструктури вимогам технічного завдання. У цьому підрозділі проєктуються архітектура збирання телеметрії, охоплення двох стовпів спостережуваності — метрик і логів — та модель візуалізації й оповіщення.

Підсистему спостережуваності спроєктовано на основі стеку Prometheus, Grafana та Loki, обґрунтованого в підрозділі 1.7. Її розгортання передбачено всередині кластера оркестрації, на робочих вузлах, що дає змогу використовувати штатні механізми Kubernetes для виявлення цілей моніторингу. Центральним компонентом збирання метрик є Prometheus, що працює за моделлю опитування (pull): він періодично звертається до кінцевих точок метрик, які експонують компоненти інфраструктури та навантаження, і зберігає отримані числові часові ряди. Виявлення цілей опитування здійснюється автоматично через механізм

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		66

service discovery Kubernetes, завдяки чому додавання нового навантаження до кластера не потребує ручного переналаштування системи моніторингу — нові цілі виявляються й починають опитуватися автоматично, що узгоджується з принципом зниження ручної праці.

Для джерел, які не експонують метрики самостійно, передбачено проміжні агенти-експортери. Зокрема, метрики операційної системи вузлів збираються експортером рівня вузла (node exporter), а стан кластера даних — спеціалізованим експортером, що перетворює внутрішні показники Galera (зокрема розмір кластера та стан синхронізації вузлів) на формат, придатний для опитування Prometheus. Це забезпечує охоплення моніторингом усіх рівнів вертикалі — від віртуальних машин до рівня даних, винесеного за межі кластера.

Відповідно до моделі «трьох стовпів», розглянутої в підрозділі 1.5, спроектована підсистема охоплює два з них. Стовп метрик реалізується зв'язкою Prometheus, що дає змогу відстежувати «чотири золоті сигнали» SRE — затримку, трафік, помилки та насиченість ресурсів — і на їх основі обчислювати індикатори рівня обслуговування (SLI) та контролювати дотримання цільових показників (SLO). Стовп логів реалізується системою Grafana Loki, що агрегує логи з усіх вузлів і навантаження; на відміну від систем повнотекстового індексування, Loki індексує лише мітки потоків логів, що робить його ресурсоощадним і придатним для розгортання в межах модельної інфраструктури [14]. Збирання логів із вузлів та подів кластера здійснюється агентом, що передає їх до Loki.

Поза охопленням свідомо залишено третій стовп — розподілене трасування, оскільки воно є доцільним переважно для складних мікросервісних архітектур із численними міжсервісними викликами, тоді як профіль цільового навантаження цього не вимагає. Цей вибір зафіксовано як обґрунтоване обмеження обсягу, а не як прогалину, і окреслює напрям можливого подальшого розвитку системи спостережуваності.

Об'єднувальним компонентом підсистеми є Grafana, що слугує єдиним інтерфейсом візуалізації для обох джерел даних — метрик Prometheus і логів Loki. Така єдина точка спостереження дає змогу співвідносити числові показники з

відповідними подіями в логах, що суттєво прискорює діагностику причин відмов. Візуалізацію спроектовано у вигляді інформаційних панелей (dashboards), згрупованих за рівнями інфраструктури: стан вузлів віртуалізації, стан кластера оркестрації, стан кластера даних та показники цільового навантаження.

Окрім візуалізації, передбачено механізм оповіщення (alerting), що автоматично сповіщає про порушення заданих умов — наприклад, про зменшення розміру кластера Galera нижче кворуму, недоступність вузла або вичерпання бюджету помилок за визначеним SLO. Це переводить підсистему зі стану пасивного спостереження у стан активного контролю надійності, замикаючи петлю зворотного зв'язку, передбачену методологією SRE. Спроектована в такий спосіб підсистема спостережуваності забезпечує дані, необхідні для перевірки ефективності інфраструктури в Розділі 3, а також є основою для моделі самообслуговування, що розглядається в наступному підрозділі.

Для забезпечення гнучкості та автоматизації процесів конфігурування збирання метрик у межах кластера оркестрації розгортання центрального компонента підсистеми спостережуваності Prometheus реалізовано на основі шаблону оператора (Prometheus Operator). Такий підхід дає змогу відійти від традиційного статичного керування конфігураційними файлами на користь декларативного опису цілей моніторингу через механізми розширення Kubernetes API, зокрема за допомогою користувацьких ресурсів типу ServiceMonitor та PodMonitor. Функціонування архітектури автоматичного виявлення цілей (Service Discovery) базується на безперервній взаємодії Prometheus із сервером API кластера, що дозволяє динамічно відстежувати зміни у життєвому циклі об'єктів інфраструктури. Під час появи нового поду або сервісу, що відповідають заданим селекторам міток, конвеєр збирання телеметрії автоматично ініціює процедуру релейблінгу (relabeling), під час якої внутрішні метадані оркестратора трансформуються в постійні мітки часових рядів, забезпечуючи уніфікацію та збереження контексту походження даних. Опитування кінцевих точок виконується з фіксованим інтервалом через HTTP-запити до ендпоінтів метрик, після чого отримані числові значення оптимізуються та записуються до

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		68

спеціалізованого часобазового сховища (Time Series Database). Для запобігання критичному навантаженню на оперативну пам'ять архітектура сховища передбачає блочний запис даних із використанням механізму випереджувального логування (Write-Ahead Logging), що гарантує високу стійкість до збоїв та швидке відновлення стану системи у разі перезапуску компонентів.

Враховуючи необхідність повного вертикального охоплення моніторингом усіх архітектурних рівнів, для компонентів, які не мають вбудованої підтримки експорту метрик у форматі Prometheus, інтегровано спеціалізовані агенти-транслятори. На рівні віртуальних машин операційної системи розгортається сервіс Node Exporter, який функціонує як системна служба і збирає низькорівневі показники ядра Linux, включаючи утилізацію процесорного часу, інтенсивність дискових операцій введення-виведення, стан віртуальної пам'яті та мережеву активність інтерфейсів. На рівні збереження даних, який винесено за межі кластера оркестрації з міркувань надійності та збереження стану, контроль працездатності здійснюється за допомогою Galera Cluster Exporter. Цей агент здійснює періодичне низькорівневе підключення до реплікованої бази даних, виконує опитування глобальних змінних стану кластера та трансформує текстові та чисельні параметри синхронізації у формат тимчасових рядів. Це дозволяє безперервно оцінювати розмір активного кворуму, поточний статус реплікації вузлів та чергу сертифікації транзакцій, мінімізуючи ризик виникнення несинхронізованих станів або розділення мережі.

Паралельно із шаром збирання чисельних метрик спроектовано архітектуру обробки текстової телеметрії на базі технології Grafana Loki, що реалізує другий стовп спостережуваності — логування. На відміну від традиційних систем повнотекстового індексування, які створюють значне обчислювальне навантаження та вимагають великих обсягів дискового простору, Loki використовує підхід індексування виключно метаданих, що дозволяє суттєво оптимізувати витрату ресурсів у межах модельної інфраструктури. Процес агрегації логів починається на рівні робочих вузлів кластера, де у вигляді системного демона DaemonSet функціонує агент збирання Promtail або Grafana

Alloy. Цей агент отримує доступ до локальних каталогів розміщення системних журналів контейнерів, перехоплює стандартні потоки виведення і збагачує кожен рядок логу контекстними мітками, отриманими від локального API оркестратора, такими як ідентифікатор простору імен, назва поду та конкретного контейнера.. Оптимізація довгострокового зберігання в Loki досягається шляхом стиснення рядків логів за допомогою ефективних алгоритмів компресії та їх запису у вигляді блоків в об'єктне сховище або на постійні томи з чітко визначеною політикою ротації та видалення за терміном давності, що забезпечує економічну доцільність та високу швидкість обробки аналітичних запитів.

Фінальним інтеграційним рівнем підсистеми спостережуваності є аналітична платформа Grafana, що об'єднує потоки чисельних метрик із Prometheus та текстові журнали з Loki в єдиному користувацькому інтерфейсі візуалізації. Ієрархічна структура інформаційних панелей (dashboards) відповідає логічним рівням інфраструктури, розділяючи моніторинг на стан віртуалізації, працездатність кластера оркестрації, продуктивність розподіленої бази даних та бізнес-показники цільового навантаження додатка. Ключовою архітектурною перевагою такої консолідації є реалізація механізму наскрізних контекстних посилань.

2.8 Проєктування моделі самообслуговування

Модель самообслуговування є завершальним рівнем спроектованої операційної моделі та практичним утіленням методології Platform Engineering, розглянутої в підрозділі. Її призначення — інкапсулювати складність нижчих рівнів інфраструктури за зрозумілими інтерфейсами й надати користувачам керовані можливості самостійного розгортання та керування навантаженням, знижуючи їх когнітивне навантаження. У цьому підрозділі проєктуються модель самообслуговування на основі вже наявних компонентів, реалізація «золотих стежок» та межі побудованої платформи.

Модель самообслуговування будується не як окремий новий компонент, а як узгоджене поєднання вже спроектованих у попередніх підрозділах

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		70

інструментів, кожен з яких закриває свою частину взаємодії користувача з інфраструктурою. Платформа Rancher надає графічний інтерфейс самообслуговування для операцій із кластером — перегляду стану навантаження, керування доступом та розгортання застосунків із каталогу — без потреби у безпосередній роботі з програмним інтерфейсом Kubernetes через командний рядок. Оператор ArgoCD надає декларативний, керований через репозиторій спосіб самообслуговування для доставки навантаження. Підсистема спостережуваності (підрозділ 2.7) доповнює цю модель, надаючи користувачеві самостійний доступ до стану його навантаження через інформаційні панелі Grafana. Сукупно ці компоненти утворюють шар, через який користувач взаємодіє з інфраструктурою, не вникаючи в реалізацію нижчих рівнів — віртуалізації, оркестрації та даних.

Центральним елементом моделі самообслуговування є проєктування «золотих стежок» (golden paths) — рекомендованих, наперед визначених та підтримуваних шляхів виконання типових завдань, що звужують кількість рішень, які доводиться ухвалювати користувачеві [10]. У спроєктованій інфраструктурі основною золотою стежкою є розгортання навантаження за моделлю GitOps: користувач не виконує жодних ручних дій із кластером, а лише вносить зміну до декларативного опису в репозиторії, після чого ArgoCD автоматично приводить кластер до описаного стану. Цей шлях є наперед прокладеним — користувачеві не потрібно ухвалювати рішення щодо способу доставки, послідовності команд чи доступу до вузлів, оскільки всі ці аспекти вже інкапсульовані платформою. Таким чином безпосередньо реалізується ключовий принцип Platform Engineering — заміна множини самостійних технічних рішень користувача на проходження прокладеного й безпечного маршруту, що водночас знижує когнітивне навантаження та ймовірність помилок.

Спроєктовану модель самообслуговування свідомо обмежено обсягом, що відповідає концепції «найтоншої життєздатної платформи» (Thinnest Viable Platform, TVP), розглянутій у підрозділі 1.6 [10]. Згідно з цією концепцією, платформа реалізує лише той мінімальний набір можливостей, що дає вимірюване

прискорення роботи, уникаючи побудови надлишкової функціональності. Відповідно, у межах цієї роботи реалізовано базову модель самообслуговування на основі готових платформних компонентів (Rancher, ArgoCD, Grafana) та однієї основної золотої стежки розгортання, тоді як побудова повноцінної внутрішньої платформи розробника з єдиним порталом самообслуговування, каталогом шаблонів і автоматизованим розгортанням середовищ на вимогу винесена за межі моделі.

Таке свідоме обмеження є методологічно обґрунтованим: воно відповідає не лише наявним ресурсам, а й самому принципу TVP, що застерігає від передчасної побудови громіздкої платформи [18]. Окреслений напрям — розвиток моделі до повноцінної внутрішньої платформи розробника — визначається як перспектива подальшого вдосконалення. Спроектowana в такий спосіб модель самообслуговування завершує проєктування операційної моделі: інфраструктура створюється як код, навантаження доставляється за моделлю GitOps [19], її стан є спостережуваним, а взаємодія користувача з нею здійснюється через інкапсульований шар самообслуговування. Детальна реалізація та перевірка всіх спроектованих підсистем на відповідність вимогам технічного завдання наведені в наступному розділі.

Концепцію золотих стежок доцільно розглянути докладніше, оскільки вона є центральною для розуміння того, як платформа знижує когнітивне навантаження на користувача. У середовищі без платформного шару користувач, що бажає розгорнути навантаження, змушений самотійно ухвалювати десятки технічних рішень: яким чином зібрати образ, де його зберігати, як описати маніфести розгортання, як налаштувати доступ до бази даних, як забезпечити зовнішній доступ до застосунку.

Кожне з цих рішень потребує знань і несе ризик помилки, а їх сукупність формує високий поріг входження. Золота стежка усуває цю складність, пропонуючи єдиний рекомендований шлях, у якому всі ці рішення вже ухвалено та перевірено. Користувачеві лишається пройти прокладеним маршрутом, не відволікаючись на деталі реалізації. Важливо, що золота стежка не забороняє

відхилення від неї для досвідчених користувачів, які мають специфічні потреби, — вона лише пропонує найпростіший і найбезпечніший шлях за замовчуванням, що покриває переважну більшість типових випадків. Саме в цьому полягає відмінність платформного підходу від жорсткого регламентування: платформа спрямовує, а не примушує.

Окремої уваги заслуговує роль декларативного опису в репозиторії як основи моделі самообслуговування. Оскільки бажаний стан навантаження повністю описується в репозиторії, а його застосування виконується автоматично, користувач отримує можливість керувати своїм навантаженням, не маючи безпосереднього доступу до кластера. Це має кілька важливих наслідків. По-перше, будь-яка зміна стану навантаження проходить через репозиторій, а отже, фіксується в історії змін із зазначенням автора та часу, що забезпечує повну простежуваність дій. По-друге, така модель природно поєднується з механізмами перевірки та погодження змін, прийнятими в системах контролю версій: зміна може бути попередньо переглянута іншим учасником перед застосуванням, що підвищує якість і безпеку. По-третє, відсутність потреби в безпосередньому доступі до кластера зменшує кількість користувачів, які мають привілейований доступ до інфраструктури, що знижує поверхню можливих помилок та зловживань. Для системи керування аудитами ця властивість є особливо цінною, оскільки вимога простежуваності та контролю змін є для неї не лише організаційною зручністю, а й регуляторною необхідністю.

Розмежування ролей між адміністратором інфраструктури та користувачем навантаження, що його забезпечує модель самообслуговування, потребує окремого розгляду, оскільки саме воно визначає практичну цінність платформи в організаційному вимірі. У класичній моделі, де відсутній платформний шар, будь-яка операція з розгортання навантаження потребує залучення адміністратора, що володіє повним доступом до інфраструктури. Це створює два типи проблем. З одного боку, адміністратор стає вузьким місцем: усі запити на розгортання чи зміну навантаження проходять через нього, що сповільнює роботу та збільшує його завантаженість рутинними завданнями. З іншого боку, така модель або

змушує надавати широкий доступ багатьом користувачам, що небезпечно, або концентрує всю роботу на кількох адміністраторах, що неефективно. Модель самообслуговування розв'язує цю суперечність, чітко розділяючи зони відповідальності: адміністратор відповідає за платформу та її політики, а користувач самостійно керує власним навантаженням у межах, дозволених платформою. Завдяки цьому адміністратор звільняється від рутинних запитів і зосереджується на розвитку платформи, а користувач отримує можливість працювати самостійно, не очікуючи на втручання адміністратора. Таке розмежування безпосередньо реалізує організаційну ідею методології Platform Engineering, за якою платформна команда створює та підтримує середовище, а команди розробки користуються ним самостійно.

Нарешті, слід детальніше обґрунтувати свідоме обмеження обсягу спроектованої платформи, оскільки межі рішення є не менш важливими за його зміст. Спокуса побудувати всеохопну платформу, що покриває всі можливі сценарії, є поширеною помилкою, яка призводить до надмірних витрат на створення функціональності, що не має реального попиту. Принцип найтоншої життєздатної платформи застерігає саме від цього: платформа має починатися з мінімального набору можливостей, що дають вимірювану користь, і розвиватися лише у відповідь на реальні потреби користувачів, а не на припущення про них. У межах цієї роботи такий мінімальний набір складають готові платформні компоненти для керування кластером, доставки навантаження та спостереження за ним, а також одна основна золота стежка розгортання. Цей набір є достатнім для демонстрації принципів самообслуговування й водночас не перевантажений надлишковою функціональністю. Подальший розвиток платформи — додавання єдиного порталу самообслуговування, каталогу типових шаблонів навантаження та можливості створення тимчасових середовищ на вимогу — визначається як напрям перспективного вдосконалення, який доцільно реалізовувати поетапно, керуючись фактичними потребами користувачів, а не прагненням до повноти заради повноти.

Висновки до другого розділу

У другому розділі проведено проектування on-premise інфраструктури для системи керування аудитами. Визначено загальну архітектуру з поділом на п'ять функціональних рівнів — віртуалізації, оркестрації, даних, спостережуваності та доставки — та обґрунтовано топологію з дев'яти віртуальних машин. Спроектовано середовище віртуалізації на основі Proxmox VE із відтворюваним механізмом підготовки шаблонів та статичною адресацією вузлів. Розроблено підсистему IaC на основі Terraform та Ansible із чітким розмежуванням зон відповідальності між розгортанням ресурсів та керуванням їх конфігурацією. Спроектовано відмовостійкий кластер оркестрації RKE2 з трьома керувальними вузлами та розподіленим сховищем etcd, а також кластер даних MariaDB Galera з синхронною multi-master реплікацією на трьох вузлах. Розроблено підсистему доставки навантаження за моделлю GitOps на основі ArgoCD та шар керування на основі Rancher. Спроектовано підсистему спостережуваності на основі Prometheus, Loki та Grafana, що охоплює два стовпи телеметрії — метрики та логи — на всіх рівнях інфраструктури. Завершено проектування моделлю самообслуговування, що реалізує принцип «найтоншої життєздатної платформи» та золоту стежку розгортання навантаження через репозиторій.

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				75
Змн.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ПЕРЕВІРКА ІНФРАСТРУКТУРИ

3.1 Розгортання базового середовища віртуалізації

У цьому розділі описано практичну реалізацію інфраструктури, спроектованої в Розділі 2, її налаштування та перевірку на відповідність вимогам технічного завдання. Виклад побудовано в порядку фактичного розгортання компонентів — від нижчих рівнів до вищих: спочатку базове середовище віртуалізації, далі відмовостійкий рівень даних, рівень оркестрації та балансувальник навантаження, після чого — цільове навантаження, підсистема спостережуваності та перевірка відмовостійкості. Такий порядок відображає реальну послідовність дій інженера та відповідає принципу нарощування інфраструктури знизу вгору.

Базовим рівнем інфраструктури є платформа віртуалізації Proxmox VE, розгорнута на фізичному сервері згідно з конфігурацією, визначеною в підрозділі 2.2. На платформі підготовлено сховище даних на основі ZFS для розміщення дисків віртуальних машин, а також налаштовано мережевий міст (bridge vmbr0), що об'єднує всі віртуальні машини в єдиний сегмент 192.168.2.0/24.

Базовим рівнем і фундаментом для побудови всієї інфраструктури є гіперконвергентна платформа віртуалізації Proxmox VE, розгорнута на фізичному сервері згідно з конфігурацією, визначеною в підрозділі 2.2. На етапі підготовки платформи особливу увагу було приділено організації дискової підсистеми, оскільки ефективність роботи всіх віртуальних машин безпосередньо залежить від швидкодії та надійності сховища даних. Для розв'язання цієї задачі безпосередньо на платформі було підготовлено та оптимізовано сховище даних на основі файлової системи ZFS. Вибір цієї технології зумовлений її здатністю забезпечувати високий рівень цілісності інформації на блочному рівні, стійкістю до прихованого пошкодження даних та наявністю вбудованих механізмів інтелектуального кешування операцій введення-виведення.

Для забезпечення можливості автоматизованого керування платформою та впровадження сучасних підходів до розгортання інфраструктури за допомогою

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		76

інструменту Terraform, було здійснено налаштування безпечного програмного доступу до гіпервізора. У системі керування Proxmox VE було створено окремий службовий обліковий запис із мінімально необхідним набором привілеїв, для якого згенеровано відповідний маркер доступу до програмного інтерфейсу (API token).

Використання цього токена замість традиційних паролів адміністратора дозволяє реалізувати концепцію декларативного керування ресурсами через програмний код. Інструмент Terraform, взаємодіючи з API платформи через цей маркер, отримує можливість в автоматичному режимі створювати, конфігурувати та змінювати параметри обчислювальних ресурсів, повністю виключаючи людський фактор із процесу рутинного адміністрування та гарантуючи абсолютну точність виконання заданих конфігураційних файлів.

Ключовим елементом підготовки середовища є створення базового шаблону віртуальної машини на основі хмарного образу операційної системи Rocky Linux 9 з підтримкою механізму cloud-init. Саме з цього шаблону надалі клонуються всі віртуальні машини інфраструктури, що забезпечує єдність і відтворюваність їх початкового стану. Підготовлене в такий спосіб середовище є основою, на якій розгортаються всі подальші компоненти. Базовий шаблон віртуальної машини на основі cloud-init зображено на рис. 3.1. та рис. 3.2. Перелік розгорнутих віртуальних машин інфраструктури в інтерфейсі Proxmox VE наведено на рис. 3.3.

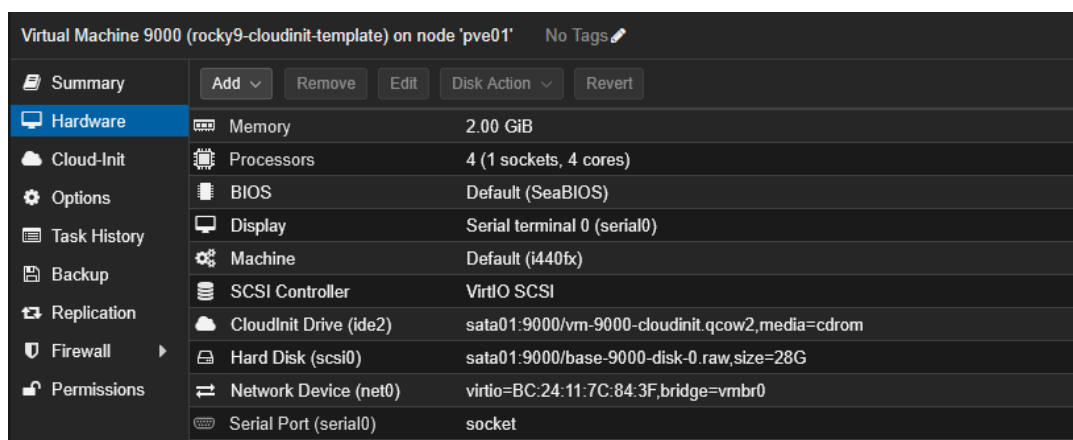


Рисунок 3.1 — Базовий шаблон віртуальної машини на основі cloud-init

User	rocky
Password	*****
DNS domain	use host settings
DNS servers	use host settings
SSH public key	none
Upgrade packages	Yes
IP Config (net0)	ip=dhcp

Рисунок 3.2 — Процес налаштування базового шаблону в Proxmox VE

Type ↑	Description	Disk usage...	Memory us...	CPU usage	Uptime	Host CPU ...	Host Mem...	Tags
qemu	701 (gl01-n01)	0.0 %	6.1 %	0.1% of 8 ...	19 days 11:5...	0.0% of 56 ...	1.6 %	
qemu	702 (gl01-n02)	0.0 %	4.7 %	0.1% of 8 ...	19 days 11:5...	0.0% of 56 ...	1.2 %	
qemu	703 (gl01-n03)	0.0 %	4.2 %	0.1% of 8 ...	19 days 11:5...	0.0% of 56 ...	1.1 %	
qemu	706 (cl01-m01)	0.0 %	83.9 %	11.8% of 8 ...	9 days 08:15...	1.7% of 56 ...	21.5 %	
qemu	707 (cl01-m02)	0.0 %	85.8 %	11.1% of 8 ...	9 days 08:12...	1.6% of 56 ...	22.0 %	
qemu	708 (cl01-m03)	0.0 %	66.2 %	5.3% of 8 ...	9 days 06:56...	0.8% of 56 ...	17.0 %	
qemu	711 (cl01-w01)	0.0 %	65.8 %	2.1% of 8 ...	9 days 08:15...	0.3% of 56 ...	8.4 %	
qemu	712 (cl01-w02)	0.0 %	71.3 %	8.3% of 8 ...	9 days 08:15...	1.2% of 56 ...	9.1 %	
qemu	720 (lb01-n01)	0.0 %	18.4 %	0.2% of 2 ...	9 days 08:15...	0.0% of 56 ...	1.2 %	

Рисунок 3.3 — Інтерфейс Proxmox VE зі списком розгорнутих машин

Ключовим елементом підготовки обчислювального середовища, що безпосередньо впливає на швидкість, стандартизацію та відтворюваність усього процесу розгортання, є створення базового шаблону віртуальної машини. Цей шаблон розроблено на основі офіційного хмарного образу операційної системи Rocky Linux 9, яка була обрана як стабільна, надійна та орієнтована на корпоративне використання платформа з високим рівнем безпеки. Головною особливістю підготовленого шаблону є повна інтеграція та налаштування вбудованого механізму cloud-init. Дана технологія виступає в ролі інструменту первинної ініціалізації та конфігурування операційної системи під час її найпершого запуску після клонування. Саме з цього єдиного базового шаблону надалі здійснюється масове клонування всіх віртуальних машин інфраструктури, що розглядаються в наступних підрозділах.

Використання методології спільного шаблону у зв'язці з cloud-init гарантує абсолютну єдність, передбачуваність та відтворюваність початкового стану кожного новоствореного вузла системи. Під час розгортання чергової віртуальної машини механізм cloud-init в автоматичному режимі, без втручання адміністратора, виконує завдання базової адаптації операційної системи під

конкретну роль: динамічно задає мережеві параметри (IP-адресу, маску, шлюз), присвоює унікальне ім'я хоста (hostname), створює необхідні системні облікові записи користувачів та імпортує ключі безпеки для авторизації. Це дозволяє перетворити рутинний процес інсталяції та первинного налаштування операційних систем на конвеєрну операцію, що триває лічені секунди.

3.2 Розгортання відмовостійкого кластера Galera

Реалізацію рівня даних виконано першою, оскільки він є незалежним нижнім рівнем інфраструктури, від якого залежить цільове навантаження. Розгортання здійснюється у два етапи: розгортання віртуальних машин засобами Terraform та їх конфігурування засобами Ansible згідно з моделлю, описаною в підрозділі 2.3.

На першому етапі засобами Terraform створюються три віртуальні машини для вузлів кластера Galera шляхом клонування підготовленого шаблону. Кожна машина отримує два диски відповідно до проєкту (підрозділ 2.5.3): системний та окремий диск для даних бази. Опис ресурсу використовує механізм лічильника, що з єдиного блоку породжує всі три однотипні вузли з послідовними іменами та адресами. Фрагмент Terraform-конфігурації для розгортання вузлів кластера даних наведено на рис. 3.4.

```

1 resource "proxmox_virtual_environment_vm" "galera_nodes" {
2   count      = var.galera_count
3   name       = "gl01-n0${count.index + 1}"
4   node_name  = var.target_node
5   vm_id      = var.galera_vmid_start + count.index
6 }

```

Рисунок 3.4 — Фрагмент Terraform-конфігурації для розгортання вузлів кластера Galera

Самі машини не встановлюються «з нуля», а клонуються з підготовленого шаблону (підрозділ 3.1), що значно пришвидшує розгортання й забезпечує єдність початкового стану.

```
clone {
  vm_id = var.clone_vm_id
  full  = true
}

cpu {
  cores = 8
  type  = "host"
}

memory { dedicated = 8192 }
```

Рисунок 3.5 — Клонування віртуальних машин із підготовленого шаблону

Відповідно до проєкту рівня даних (підрозділ 2.5) кожній машині призначається два диски — системний та окремий диск під дані бази, який згодом використає LVM-том на етапі конфігурування. Конфігурацію дисків вузла кластера даних наведено на рис. 3.6.

```
disk {
  datastore_id = var.datastore_id
  interface    = "scsi0"
  size         = 50
}

disk {
  datastore_id = var.datastore_id
  interface    = "scsi1"
  size         = 15
}
```

Рисунок 3.6 — Конфігурація дисків вузла кластера даних у Terraform

Початкова мережева конфігурація задається механізмом cloud-init: Terraform передає у машину статичну IP-адресу, шлюз та DNS, які застосовуються при першому запуску без ручного втручання. Адреса формується з базової підмережі та індексу вузла. Налаштування мережевого інтерфейсу через cloud-init показано на рис. 3.7.

```

initialization {
  datastore_id = var.datastore_id
  user_data_file_id = "${var.datastore_id}:snippets/rocky-pw-fix.yaml"

  ip_config {
    ipv4 {
      address = "${var.network_base}${var.galera_ip_start + count.index}/24"
      gateway = var.network_gateway
    }
  }
  dns { servers = [var.network_gateway, "8.8.8.8"] }
}

```

Рисунок 3.7 — Налаштування мережевого інтерфейсу через механізм cloud-init

Окремо слід відзначити блок керування життєвим циклом ресурсу, що ігнорує подальші зміни окремих параметрів.

Це свідоме інженерне рішення: воно запобігає небажаному перестворенню або перезавантаженню вже працюючих вузлів бази даних при повторних запусках Terraform, що є критичним для рівня даних зі станом. Блок керування життєвим циклом ресурсу Terraform наведено на рис. 3.8.

```

agent { enabled = true }
serial_device {}
lifecycle {
  ignore_changes = [
    clone,
    initialization,
    disk
  ]
}

```

Рисунок 3.8 — Блок керування життєвим циклом ресурсу Terraform

Окрім створення машин, Terraform автоматично генерує інвентарний файл для Ansible, наповнюючи його фактичними адресами створених вузлів — у такий спосіб реалізовано передавання стану між етапами, описане в підрозділі 2.3.2. Автоматично згенерований інвентарний файл для Ansible показано на рис. 3.9.

```

1 resource "local_file" "ansible_inventory" {
2   filename = "${path.module}/inventory.ini"
3
4   content = <<-EOT
5   [galera-nodes]
6   %{ for vm in proxmox_virtual_environment_vm.galera_nodes ~}
7   ${vm.name} ansible_host=${split("/", vm.initialization[0].ip_config[0].ipv4[0].address)[0]} ansible_user=rocky ansible_password=rocky
8   %{ endfor ~}
9
10  [cluster:children]
11  galera-nodes
12
13  [cluster:vars]
14  # Add any global variables for this group here
15  ansible_ssh_common_args='-o StrictHostKeyChecking=no'
16  EOT

```

Рисунок 3.9 — Автоматично згенерований інвентарний файл для Ansible

Цей фрагмент є практичним утіленням принципу єдиного джерела істини: перелік вузлів для конфігурування не ведеться вручну, а похідний від фактичного стану інфраструктури, що унеможливорює розбіжності між реально створеними та конфігурованими машинами.

Описаний модуль застосовується з робочої станції інженера, що звертається до програмного інтерфейсу Proxmox ззовні, відповідно до моделі доступу, визначеної в підрозділі 2.3.3. Робочий цикл Terraform складається з трьох послідовних команд, кожна з яких відповідає окремій фазі керування інфраструктурою.

Першою виконується ініціалізація робочого каталогу командою `terraform init`. На цьому етапі Terraform завантажує та встановлює оголошені у блоці `required_providers` провайдери — зокрема провайдер `brg/proxmox` для взаємодії з платформою віртуалізації — і готує локальне середовище до роботи. Ініціалізацію робочого каталогу командою `terraform init` показано на рис. 3.10.

```

[ntyshkevych-local@frm01 nodes-deploy-tf-proxmox]$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/local from the dependency lock file
- Reusing previous version of bpg/proxmox from the dependency lock file
- Using previously-installed hashicorp/local v2.8.0
- Using previously-installed bpg/proxmox v0.66.1

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[ntyshkevych-local@frm01 nodes-deploy-tf-proxmox]$ █

```

Рисунок 3.10 — Ініціалізація робочого каталогу командою terraform init

Другою фазою є планування командою terraform plan. Terraform зіставляє бажаний стан, описаний у коді, з поточним станом інфраструктури, зафіксованим у файлі стану, і формує план змін — перелік ресурсів, які буде створено, змінено або видалено. На цьому етапі жодних фактичних змін не вноситься; план є засобом попереднього контролю, що дає змогу інженерові переконатися в коректності намічених дій перед їх застосуванням.

Для розгортання кластера даних план відображає створення трьох віртуальних машин та супутнього інвентарного файлу. Результат планування змін командою terraform plan наведено на рис. 3.11.

```
# proxmox_virtual_environment_vm.master_nodes[0] will be created
+ resource "proxmox_virtual_environment_vm" "master_nodes" {
+   acpi              = true
+   bios              = "seabios"
+   id                = (known after apply)
+   ipv4_addresses    = (known after apply)
+   ipv6_addresses    = (known after apply)
+   keyboard_layout    = "en-us"
+   mac_addresses     = (known after apply)
+   migrate           = false
+   name              = "cl01-m01"
+   network_interface_names = (known after apply)
+   node_name          = "pve01"
+   on_boot            = true
+   protection         = false
+   reboot             = false
+   scsi_hardware      = "virtio-scsi-pci"
+   started            = true
+   stop_on_destroy    = false
+   tablet_device      = true
+   template           = false
+   timeout_clone      = 1800
}
```

Рисунок 3.11 — Результат планування змін командою terraform plan

Третьою, завершальною фазою є застосування командою terraform apply. Terraform повторно формує план і запитує в інженера явне підтвердження перед внесенням змін; лише після підтвердження (yes) інструмент звертається до програмного інтерфейсу Proxmox і фактично створює описані ресурси — клонує шаблон, налаштовує диски та мережу, після чого генерує інвентарний файл. По завершенні Terraform виводить підсумок виконаних дій та оголошені вихідні значення (output) — зокрема фактичні IP-адреси створених вузлів. Застосування конфігурації командою terraform apply показано на рис. 3.12.

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

master_ips = [
  "192.168.2.186",
]
worker_ips = [
  "192.168.2.191",
  "192.168.2.192",
]
[ntyshkevych-local@fr01 nodes-deploy-tf-proxmox]$ cd ..
[ntyshkevych-local@fr01 new]$ cd
```

Рисунок 3.12 — Застосування конфігурації командою terraform apply

Слід відзначити, що описаний цикл є ідемпотентним: повторне виконання terraform apply без змін у коді не призведе до жодних дій, оскільки фактичний стан інфраструктури вже відповідає бажаному.

Це фундаментально відрізняє декларативний підхід від імперативних сценаріїв класичного адміністрування (підрозділ 1.2), повторний запуск яких міг призводити до непередбачуваних результатів. Після успішного завершення цього етапу інфраструктура рівня даних створена й готова до конфігурування, що розглядається далі.

3.3 Розгортання кластера оркестрації RKE2

Після розгортання рівня даних реалізовано рівень оркестрації — кластер RKE2. Розгортання виконано за тією самою двоетапною моделлю (Terraform для розгортання, Ansible для конфігурування), проте етап конфігурування тут суттєво ширший: окрім формування самого кластера, він автоматизує розгортання платформного шару керування та доставки.

Засобами Terraform створюються віртуальні машини кластера: три керувальні вузли (control plane) та два робочі вузли (workers). На відміну від кластера даних, тут описано два окремі набори ресурсів — для керувальних і робочих вузлів, — оскільки вони відрізняються за призначенням, кількістю та параметрами. Розгортання вузлів кластера RKE2 засобами Terraform наведено на рис. 3.13. та рис. 3.14.

```
1 resource "proxmox_virtual_environment_vm" "master_nodes" {
2   count = var.master_count
3   name   = "cl01-m0${count.index + 1}"
4   node_name = var.target_node
5   vm_id    = var.master_vm_id_start + count.index
6
7   clone {
8     vm_id = var.clone_vm_id
9     full = true
10  }
11
12  cpu {
13    cores = 8
14    type = "host"
15  }
16
17  memory { dedicated = 8192 }
18
19  disk {
20    datastore_id = var.datastore_id
21    interface = "scsi0"
22    size = 50
23  }
24
25  network_device { bridge = "vmb0" }
```

Рисунок 3.13 — Розгортання вузлів кластера RKE2 засобами Terraform

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		85

```

resource "proxmox_virtual_environment_vm" "worker_nodes" {
  count      = var.worker_count
  name       = "cl01-w0${count.index + 1}"
  node_name  = var.target_node
  vm_id      = var.worker_vmid_start + count.index

  clone {
    vm_id = var.clone_vm_id
    full  = true
  }

  cpu {
    cores = 8
    type  = "host"
  }

  memory { dedicated = 4096 }

  disk {
    datastore_id = var.datastore_id
    interface    = "scsi0"
    size         = 100
  }
}

```

Рисунок 3.14 — Принципи конфігурування кластера оркестрації RKE2

Принципи розгортання (клонування шаблону, призначення ресурсів, мережева конфігурація через cloud-init, керування життєвим циклом) аналогічні розглянутим у підрозділі 3.2.1. Після створення машин Terraform генерує інвентарний файл, але, на відміну від кластера даних, групує вузли за двома ролями — masters та workers, — що згодом використовує Ansible для розмежування логіки їх конфігурування. Ansible-роль для розгортання кластера оркестрації показано на рис. 3.15.

```

resource "local_file" "ansible_inventory" {
  filename = "${path.module}/inventory.ini"

  content = <<-EOT
[masters]
%{ for vm in proxmox_virtual_environment_vm.master_nodes ~}
${vm.name} ansible_host=${split("/", vm.initialization[0].ip_config[0].ipv4[0].address)[0]} ansible_user=rocky
%{ endfor ~}

[workers]
%{ for vm in proxmox_virtual_environment_vm.worker_nodes ~}
${vm.name} ansible_host=${split("/", vm.initialization[0].ip_config[0].ipv4[0].address)[0]} ansible_user=rocky
%{ endfor ~}

[cluster:children]
masters
workers

[cluster:vars]
# Add any global variables for this group here
ansible_ssh_common_args='-o StrictHostKeyChecking=no'
EOT
}

```

Рисунок 3.15 — Ansible-роль для розгортання RKE2

Конфігурування виконує Ansible-роль rke2, що автоматизує формування кластера за моделлю «ініціалізація — приєднання», описаною в підрозділі 2.4.2. Роль реалізує три різні сценарії залежно від ролі вузла, розмежовані умовами when.

Під час виконання роль забезпечує послідовне розгортання компонентів кластера та координацію взаємодії між вузлами. Для цього використовуються змінні інвентарю та факти Ansible, які дають змогу передавати параметри конфігурації між етапами розгортання. Зокрема, після створення початкового вузла формується службова інформація, необхідна для підключення інших вузлів до кластера, що дозволяє автоматизувати процес масштабування та уникнути ручного налаштування кожного елемента інфраструктури.

Перший керувальний вузол ініціалізує кластер: на ньому встановлюється компонент RKE2 у режимі сервера, після чого зчитується згенерований маркер приєднання (node-token), який зберігається як факт Ansible для подальшого передавання іншим вузлам. Саме цей вузол стає точкою, навколо якої формується кластер. Принципи конфігурування кластера оркестрації наведено на рис. 3.16.

```
- name: Template First Master config.yaml
  template:
    src: config.yaml.j2
    dest: /etc/rancher/rke2/config.yaml

- name: Install RKE2 Server (First Master)
  shell: curl -sfl https://get.rke2.io | INSTALL_RKE2_TYPE="server" sh -
  args:
    creates: /usr/local/bin/rke2

- name: Enable and Start RKE2 Server (This takes 1-2 minutes)
  systemd:
    name: rke2-server
    state: started
    enabled: yes

- name: Wait for RKE2 node-token to be generated
  wait_for:
    path: /var/lib/rancher/rke2/server/node-token
    state: present
    timeout: 300

- name: Read RKE2 node-token
  slurp:
    src: /var/lib/rancher/rke2/server/node-token
  register: rke2_token_b64

- name: Store token as an Ansible Fact
  set_fact:
    rke2_token: "{{ rke2_token_b64.content | b64decode | trim }}"

when: inventory_hostname == groups['masters'][0]
```

Рисунок 3.16 — Ініціалізація першого керувального вузла кластера RKE2

Решта керувальних вузлів приєднуються до кластера також у режимі сервера, але використовуючи отриманий маркер та адресу першого вузла, що забезпечує формування високодоступного рівня керування з трьох вузлів та спільного сховища etcd. Робочі вузли встановлюють RKE2 у режимі агента й приєднуються до кластера для виконання навантаження. Таким чином, єдина роль автоматизує формування всього кластера, диференціюючи дії за роллю вузла без дублювання коду.

Реалізований у такий спосіб підхід гарантує суворе дотримання принципу ідемпотентності в інженерії автоматизації: повторний запуск сценаріїв конфігурування не порушує роботу вже функціонуючого середовища, а лише перевіряє його відповідність задекларованому еталону. Це створює надійний фундамент для подальшого безперебійного горизонтального масштабування системи шляхом динамічного додавання нових обчислювальних потужностей без простою сервісів.

Після формування кластера роль автоматизує отримання доступу до нього: файл конфігурації доступу (kubeconfig) копіюється на робочу станцію, а адреса програмного інтерфейсу в ньому замінюється з локальної на фактичну адресу керувального вузла, що уможливлює керування кластером ззовні.

Завершальна й найбільша частина ролі автоматизує розгортання платформного шару поверх кластера за допомогою менеджера пакетів Helm. Послідовно встановлюються: cert-manager (для автоматизації керування TLS-сертифікатами), платформа керування Rancher та оператор GitOps ArgoCD. Кожен компонент розгортається декларативно через Helm із заданням параметрів доступу (зокрема імен вузлів у форматі <сервіс>.<IP>.nip.io), а роль очікує на готовність кожного компонента перед переходом до наступного.

Тим самим увесь платформний шар, спроектований у підрозділах 2.6 та 2.8, піднімається автоматично, без жодної ручної дії в інтерфейсі кластера.

Успішне виконання ролі підтверджується підсумковим звітом без помилок (failed=0) та діагностичними повідомленнями з адресами доступу до розгорнутих платформних компонентів.

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		88

Доступність платформного шару засвідчується вебінтерфейсами Rancher та ArgoCD, що стають доступними за відповідними адресами після завершення розгортання. Підсумок виконання Ansible без помилок наведено на рис. 3.17.

```
TASK [rke2 : Print Rancher Access Information] *****
skipping: [c101-m02]
ok: [c101-m01] => {
  "msg": [
    "=====",
    "✅ Rancher has been successfully deployed!",
    "🌐 URL: https://rancher.192.168.2.151.nip.io",
    "👤 Username: admin",
    "🔑 Password: RancherAdmin123!",
    "⏰ Note: It may take 2-3 minutes for the web interface",
    "   to become fully accessible while pods start up.",
    "=====",
  ]
}
skipping: [c101-m03]
skipping: [c101-w01]
skipping: [c101-w02]

PLAY RECAP *****
c101-m01      : ok=26   changed=5    unreachable=0    failed=0    skipped=6    rescued=0    ignored=1
c101-m02      : ok=8     changed=1    unreachable=0    failed=0    skipped=24    rescued=0    ignored=1
c101-m03      : ok=8     changed=1    unreachable=0    failed=0    skipped=24    rescued=0    ignored=1
c101-w01      : ok=8     changed=1    unreachable=0    failed=0    skipped=24    rescued=0    ignored=1
c101-w02      : ok=8     changed=1    unreachable=0    failed=0    skipped=24    rescued=0    ignored=1
```

Рисунок 3.17 — Підсумок виконання Ansible (PLAY RECAP) без помилок

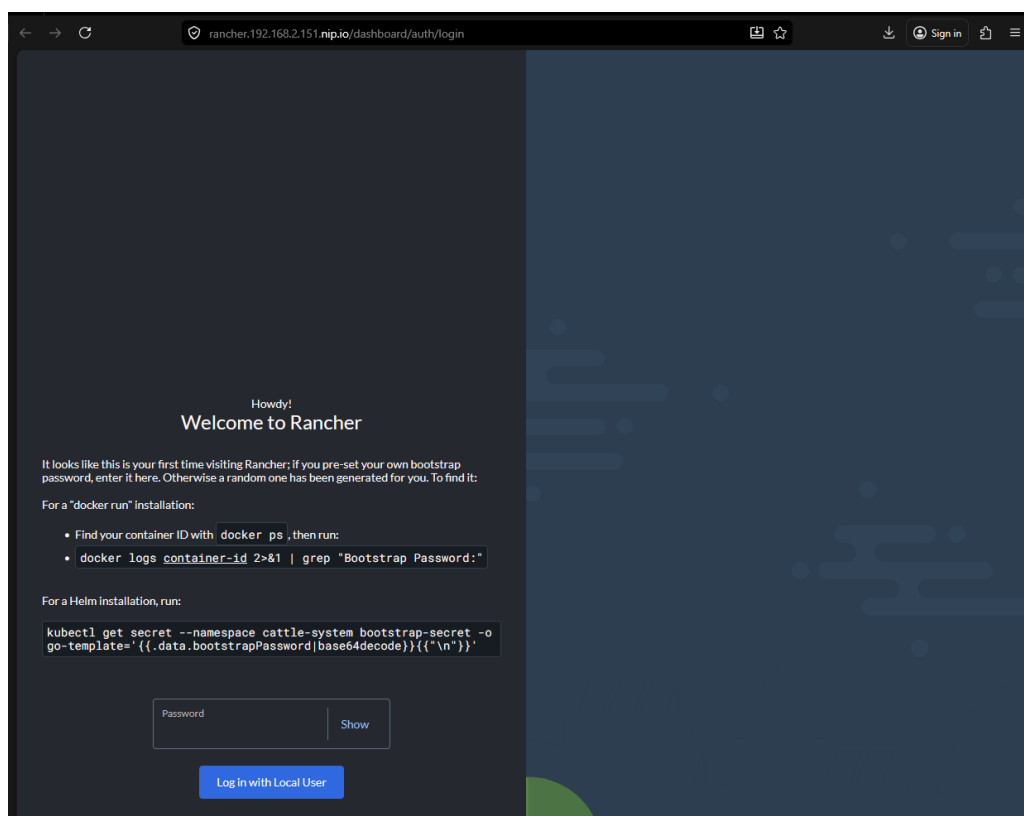


Рисунок 3.18 — Вікно доступу до платформи Rancher

Таким чином, рівень оркестрації реалізовано відповідно до проєкту: розгорнуто кластер RKE2 з високодоступним рівнем керування та автоматично піднятим платформним шаром (Rancher, ArgoCD). Демонстрацію масштабування цього кластера засобами інфраструктури як код наведено в наступному підрозділі.

3.4 Масштабування кластера засобами інфраструктури як код

Окремим етапом реалізації є демонстрація масштабування кластера оркестрації, що наочно ілюструє переваги підходу «інфраструктура як код» порівняно з класичним адмініструванням. Якщо в попередньому підрозділі кластер було розгорнуто з одним керувальним вузлом, то в цьому підрозділі його розширено до трьох керувальних вузлів задля забезпечення високодоступного рівня керування з кворумом etcd, спроектованого в підрозділі 2.4.2. Принципово важливо, що це розширення не потребує ані ручного створення віртуальних машин, ані ручного переналаштування інвентарю.

Уся операція масштабування на рівні наміру інженера зводиться до зміни значення єдиної змінної — бажаної кількості керувальних вузлів — у файлі параметрів. Оновлений декларативний опис інфраструктури для масштабування наведено на рис. 3.19.

```
proxmox_insecure = true

master_count = 3
worker_count = 2

target_node = "pve01"
clone_vm_id = 9000
datastore_id = "sata01"
```

Рисунок 3.19 — Оновлений декларативний опис інфраструктури для масштабування кластера

Ця зміна не є імперативною інструкцією «створити дві машини», а декларацією нового бажаного стану — «кластер повинен мати три керувальні вузли». Узгодження фактичного стану з новим бажаним повністю покладається на інструментарій, що відповідає принципу декларативності, закладеному в основу всієї роботи.

Варто також підкреслити роль ідемпотентності як ключової властивості реалізованої автоматизації, що робить її безпечною для повторного застосування. На відміну від імперативних сценаріїв класичного адміністрування, повторний запуск яких міг призводити до непередбачуваних наслідків, повторне виконання

як Terraform, так і Ansible-ролі на вже налаштованому середовищі не вносять жодних змін — інструменти лише перевіряють відповідність фактичного стану бажаному і завершують роботу без дій. Саме ця властивість робить можливим безпечне масштабування через повторний запуск наявної автоматизації: інженер не ризикує порушити роботу вже функціонуючих вузлів, застосовуючи ту саму автоматизацію до розширеного складу кластера.

У реалізованій моделі єдиним джерелом істини є стан Terraform: інвентар Ansible є його похідною, а не самостійним документом, що потребує ручного супроводу. Завдяки цьому будь-яка зміна складу інфраструктури — додавання чи видалення вузла — автоматично відображається в інструменті конфігурування без жодного ручного втручання, що усуває цілу категорію помилок, характерних для класичного підходу.

Під час наступного виконання команди планування Terraform зіставляє новий бажаний стан із поточним і визначає, що бракує двох керувальних вузлів. Сформований план відображає створення рівно цих двох машин (c101-m02, c101-m03) без жодного впливу на вже наявні вузли — як керувальний, так і робочі залишаються незмінними. Список вузлів у графічному інтерфейсі Rancher показано на рис. 3.20.

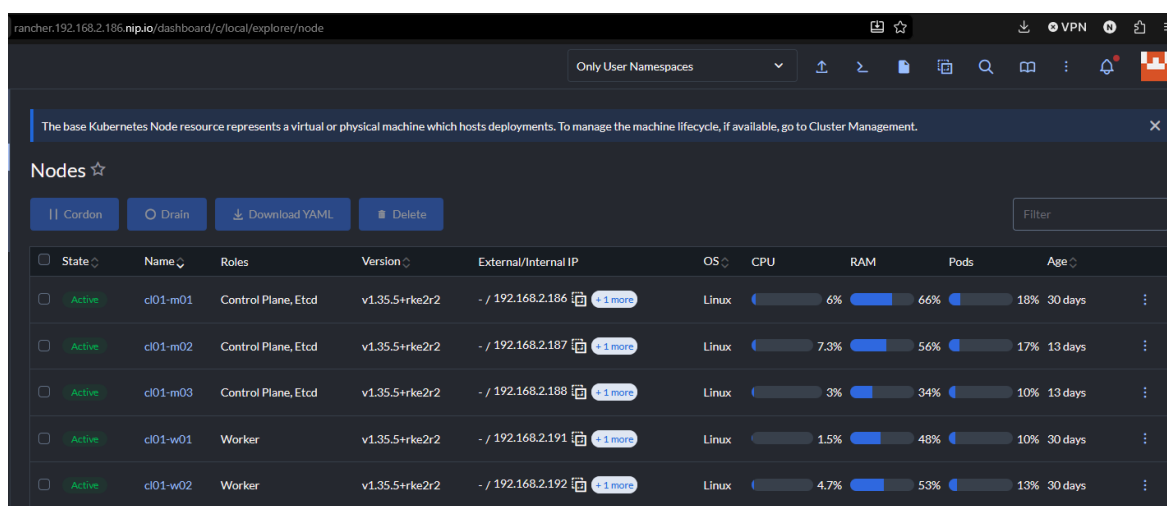


Рисунок 3.20 — Список вузлів в графічному інтерфейсі Rancher

Те, що план містить «0 to change, 0 to destroy», є практичним підтвердженням безпеки операції: масштабування не зачіпає працюючі вузли й не призводить до простою наявних. Завдяки механізму лічильника нові машини

автоматично отримують наступні за порядком імена, ідентифікатори та IP-адреси, узгоджені зі схемою найменування .

Після підтвердження застосування Terraform створює дві нові віртуальні машини та, що особливо важливо, автоматично регенерує інвентарний файл Ansible, додаючи до групи masters нові вузли. Інженерів не потрібно вручну вписувати їх адреси — інвентар є похідним від фактичного стану інфраструктури й оновлюється разом із ним. Це усуває типове джерело помилок класичного підходу, коли фактичний склад інфраструктури й документ, що його описує, поступово розходяться.

Після розгортання нових машин повторно запускається Ansible-роль rke2. Завдяки ідемпотентності ролі вона не виконує повторно дій на вже налаштованих вузлах, а застосовує сценарій приєднання керувального вузла лише до нових машин: на них встановлюється RKE2 у режимі сервера, і вони приєднуються до наявного кластера, використовуючи маркер приєднання першого вузла. У результаті рівень керування розширюється з одного до трьох вузлів, а сховище etcd набуває кворуму, що робить кластер стійким до відмови одного керувального вузла.

Описана демонстрація унаочнює фундаментальну відмінність реалізованого підходу від класичного. За класичного адміністрування розширення кластера на два вузли вимагало б ручного створення віртуальних машин, встановлення на них операційної системи, налаштування мережі, встановлення та конфігурування компонентів кластера й ручного оновлення документації — тобто послідовності ручних, схильних до помилок операцій, розглянутих у підрозділі 1.2. У реалізованій інфраструктурі та сама мета досягається зміною одного числа в декларативному описі та повторним запуском уже наявної автоматизації. Це є практичним підтвердженням ефективності методологій, покладених в основу роботи: масштабованості, відтворюваності та зниження обсягу ручної праці.

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				
Змн.	Арк.	№ докум.	Підпис	Дата		92

3.5 Розгортання балансувальника навантаження

Після розгортання кластерів оркестрації та даних реалізовано єдину точку доступу до обох — вузол-балансувальник lb01-n01 на основі HAProxy. Як обґрунтовано в підрозділі 2.5.2, один балансувальник обслуговує два незалежні фронти на різних портах, що дає змогу уникнути надлишкової складності без втрати функціональності в межах модельної реалізації.

Віртуальна машина балансувальника (lb01-n01, 192.168.2.180) створюється засобами Terraform за тією самою моделлю, що й інші вузли інфраструктури, — клонуванням шаблону Rocky Linux 9 з конфігурацією через cloud-init. З огляду на характер навантаження — виключно проксіювання TCP-з'єднань без збереження стану — вузлу виділено мінімально достатні ресурси: 2 vCPU та 2 ГБ оперативної пам'яті.

Конфігурування виконує Ansible-роль haproxy, що автоматизує встановлення HAProxy, налаштування підсистем безпеки (firewalld, SELinux) та розгортання конфігураційного файлу. Ключовою особливістю є те, що перелік бекендів у конфігурації не задається статично, а формується динамічно з даних інвентарю Ansible: адреси керувальних вузлів та вузлів бази даних підставляються автоматично через шаблон, що унеможливлює розбіжності між фактичним складом кластерів і конфігурацією балансувальника. Фрагмент шаблону, що формує бекенд кластера даних, ілюструє цей механізм. Конфігурацію HAProxy для балансування навантаження наведено на рис. 3.21.

```
backend galera_backend
  mode tcp
  balance first
  option tcp-check
  {% for host in groups['galera_nodes'] %}
  {% if loop.first %}
    server {{ host }} {{ hostvars[host]['ansible_host'] }}:{{ haproxy_mysql_port }} check
  {% else %}
    server {{ host }} {{ hostvars[host]['ansible_host'] }}:{{ haproxy_mysql_port }} check backup
  {% endif %}
  {% endfor %}
```

Рисунок 3.21 — Конфігурація HAProxy для балансування навантаження

Конфігурацію HAProxy побудовано навколо двох незалежних фронтендів, кожен з яких реалізує власну стратегію балансування відповідно до природи відповідного сервісу.

Перший фронтенд обслуговує програмний інтерфейс Kubernetes (порт 6443) і розподіляє запити між трьома керувальними вузлами за стратегією рівномірного кругового розподілу (roundrobin). Це допустимо, оскільки всі три керувальні вузли рівноцінні та здатні обслуговувати будь-який запит до програмного інтерфейсу кластера.

Другий фронтенд обслуговує кластер даних (порт 3306) за стратегією «активний вузол — резерв» (balance first): усі з'єднання спрямовуються на перший доступний вузол, тоді як решта вузлів позначені як резервні (backup) і починають обслуговувати запити лише в разі відмови активного. Така стратегія забезпечує запис лише в один вузол кластера Galera, що запобігає конфліктам сертифікації транзакцій при паралельному записі в різні вузли.

Коректність роботи балансувальника підтверджується сторінкою статистики HAProxy, доступною на порту 8404. На ній відображено стан обох бекендів у реальному часі.

Бекенд програмного інтерфейсу Kubernetes містить три керувальні вузли (cl01-m01–cl01-m03), усі з яких перебувають у стані UP (позначені зеленим), що відповідає повному кворуму рівня керування (3/3). Бекенд кластера даних відображає реалізовану стратегію «активний вузол — резерв»: вузол gl01-n01 є активним (позначений як Act), а вузли gl01-n02 та gl01-n03 — резервними (позначені як Bck), у повній відповідності до проєкту. Звернення до програмного інтерфейсу Kubernetes через балансувальник показано на рис. 3.21.

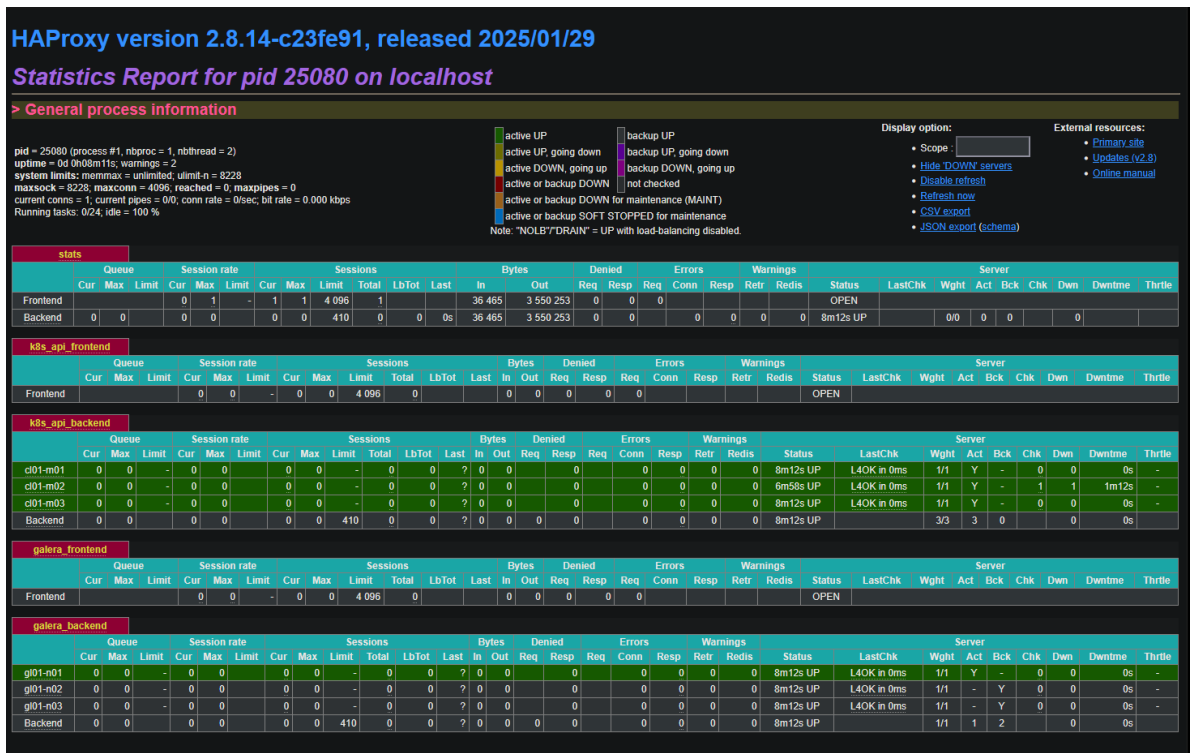


Рисунок 3.22 — Звернення до програмного інтерфейсу Kubernetes через балансувальник

Для остаточної верифікації працездатності спроектованого мережевого рівня було виконано один контрольний запуск комплексної перевіркової команди безпосередньо з робочої станції адміністратора. Одноразове виконання цього тестового запиту дозволило в межах єдиної операційної сесії наочно продемонструвати високу ефективність та безпомилковість функціонування налаштованої схеми балансування. Отриманий результат чітко засвідчив, що пряме звернення до програмного інтерфейсу кластера оркестрації через єдину IP-адресу балансувальника повертає стабільне та коректне підтвердження працездатності. Звернення до програмного інтерфейсу кластера через балансувальник повертає підтвердження працездатності, а запит стану кластера даних — очікуваний розмір кластера з трьох вузлів. Перевірку роботи балансувальника навантаження показано на рис. 3.23.

```
[ntyshkevych-local@frm01 haproxy-ansible-deploy]$ mariadb -h 192.168.2.180 -u root -pubuntu -e "SHOW STATUS LIKE 'wsrep_cluster_size';"
+-----+-----+
| Variable_name | Value |
+-----+-----+
| wsrep_cluster_size | 3 |
+-----+-----+
[ntyshkevych-local@frm01 haproxy-ansible-deploy]$
```

Рисунок 3.23 — Перевірка роботи балансувальника навантаження

3.6 Контейнеризація та розгортання цільового навантаження

Після формування інфраструктурної вертикалі реалізовано розгортання цільового навантаження — системи керування аудитами. Відповідно до спрямованості роботи, застосунок розглядається як корисне навантаження, на якому демонструється робота інфраструктури; його розгортання здійснюється за моделлю GitOps через оператор ArgoCD, що є практичним утіленням «золотої стежки», спроектованої в підрозділі 2.8.

Цільовий застосунок є вебзастосунком на основі фреймворку Django, що взаємодіє з реляційною базою даних. Для його розгортання в кластері створено образ контейнера на основі багатоетапного (multi-stage) складання: на першому етапі встановлюються залежності та збирається статичний вміст, на другому — формується мінімальний образ виконання, що містить лише необхідне для роботи. Такий підхід зменшує розмір кінцевого образу та відповідає принципу незмінності, розглянутому в підрозділі 1.4.

Окремо враховано вимоги захищеного середовища RKE2: образ налаштовано на запуск від непривілейованого користувача (UID 10001), що узгоджується з політиками безпеки CIS, яким відповідає кластер. Конфігурацію застосунку винесено за межі образу — параметри підключення до бази даних та інші чутливі дані передаються через змінні оточення, що дає змогу використовувати той самий незмінний образ у різних середовищах. Принциповим рішенням є те, що застосунок звертається до бази даних не безпосередньо до вузла, а через балансувальник навантаження (192.168.2.180:3306), завдяки чому навантаження є відмовостійким на рівні даних: відмова окремого вузла Galera не порушує його роботу.

Бажаний стан навантаження в кластері описано декларативно у вигляді набору маніфестів Kubernetes, що зберігаються в репозиторії системи контролю версій. Цей набір охоплює: простір імен (namespace) для ізоляції навантаження; секрет (Secret) з параметрами підключення та конфігурацією; розгортання (Deployment) самого застосунку у двох репліках; службу (Service) та контролер

входу (Ingress) для забезпечення зовнішнього доступу; а також запит на постійний том (PersistentVolumeClaim) для зберігання завантажених файлів.

Ключовою особливістю розгортання є застосування ініціалізаційного контейнера (initContainer), що виконує міграції схеми бази даних перед запуском основного застосунку. Завдяки синхронній реплікації кластера Galera зміни схеми, внесені одним екземпляром, автоматично розповсюджуються на всі вузли, тому міграції достатньо виконати один раз. Окрім того, для кожної репліки налаштовано проби готовності (readiness) та життєздатності (liveness), що дають Kubernetes змогу автоматично перезапускати непрацездатні екземпляри та спрямовувати трафік лише на готові до обслуговування, реалізуючи механізм самовідновлення.

Розгортання застосунку здійснюється не ручним застосуванням маніфестів, а реєстрацією відповідного об'єкта Application в ArgoCD, який надалі безперервно синхронізує стан кластера з вмістом репозиторію. Після реєстрації ArgoCD автоматично зчитує маніфести з репозиторію, створює всі описані ресурси й приводить кластер до бажаного стану.

Увімкнений режим автоматичної синхронізації означає, що будь-яка подальша зміна в репозиторії автоматично відображається в кластері без ручного втручання, що повністю реалізує принцип, закладений у підрозділі 2.6.2: репозиторій є єдиним джерелом істини про стан навантаження. Реєстрацію застосунку в ArgoCD за моделлю GitOps показано на рис. 3.24.

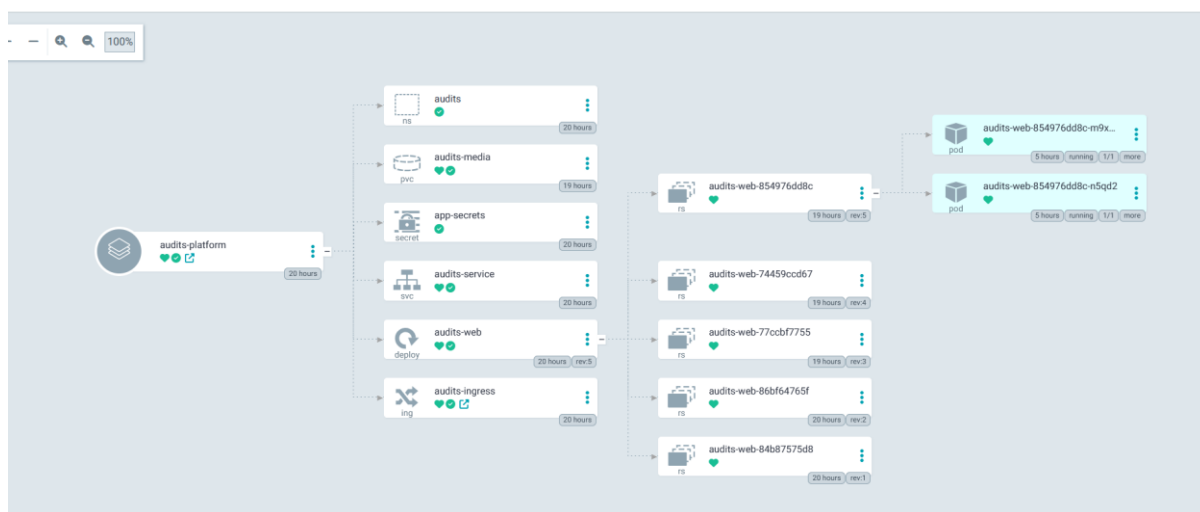


Рисунок 3.24 — Реєстрація застосунку в ArgoCD за моделлю GitOps

Під час розгортання виявлено, що дистрибутив RKE2 не містить наперед налаштованого механізму динамічного виділення постійних томів. Для забезпечення роботи запиту на постійний том розгорнуто розгортувач local-path-provisioner, який створює томи на локальних дисках вузлів кластера.

Додатково, з огляду на ввімкнену в операційній системі Rocky Linux підсистему безпеки SELinux у режимі примусового контролю (Enforcing), для каталогів зберігання томів встановлено відповідний контекст безпеки, без якого контейнери не отримували доступу на запис. Це є практич. ним свідченням того, що розгортання виконувалося в захищеному середовищі, а не з вимкненими механізмами безпеки.

Функціонування оператора ArgoCD у межах обраної парадигми GitOps базується на безперервному циклі узгодження (reconciliation loop), що забезпечує постійне порівняння поточного стану об'єктів у кластері Kubernetes із їхнім еталонним описом у декларативних маніфестах репозиторію. Важливою архітектурною перевагою такої схеми є автоматичне виявлення та нівелювання так званого «дрейфу конфігурації» (configuration drift).

Якщо в процесі експлуатації системи відбудеться несанкціонована або випадкова ручна зміна параметрів розгортання безпосередньо через інтерфейс командного рядка кластера, оператор миттєво зафіксує розбіжність із першоджерелом і автоматично оновить ресурси до задекларованого стану.

Такий підхід не лише мінімізує ризики людського фактора, але й трансформує процес релізу програмного забезпечення з потенційно аварійної процедури на повністю контрольований, документований та безпечний процес, де кожна зміна фіксується історією комітів у системі контролю версій.

Крім того, автоматичне виявлення стану розсинхронізації (Out-of-Sync) та миттєве повернення об'єктів кластера до еталонних налаштувань перетворює репозиторій на надійний інструмент проактивного захисту та протидії несанкціонованим діям усередині периметра системи. У разі виявлення критичних помилок чи аномалій у нових версіях продукту, процедура відновлення працездатності комплексу зводиться до виконання стандартної операції

скасування останнього коміту в Git, що негайно запускає каскадне автоматичне відкочування

3.7 Впровадження підсистеми спостережуваності

Завершальним компонентом інфраструктури є підсистема спостережуваності, що реалізує підхід SRE до кількісного контролю надійності, розглянутий у підрозділах 1.5 та 2.7. Її розгорнуто у вигляді стеку з трьох взаємодоповнювальних інструментів — Prometheus, Grafana та Loki, — що разом охоплюють збирання метрик, їх візуалізацію та агрегацію логів.

Для збирання метрик розгорнуто пакет kube-prometheus-stack за допомогою менеджера пакетів Helm. Цей пакет є комплексним рішенням, що одним розгортанням встановлює повний набір компонентів моніторингу: оператор Prometheus, який керує конфігурацією через власні ресурси Kubernetes; сам сервер Prometheus для збирання та зберігання метрик; агенти node-exporter, що розгортаються на кожному вузлі кластера у вигляді DaemonSet і експонують метрики операційної системи; компонент kube-state-metrics для метрик стану об'єктів Kubernetes; а також Alertmanager для обробки оповіщень. Розгортання здійснено в окремий простір імен monitoring із наданням постійних томів для зберігання даних Prometheus та Alertmanager, що забезпечує збереження історії метрик.

Розгортання підсистеми спостережуваності в реальному кластері виявило низку інженерних особливостей, пов'язаних зі специфікою середовища. По-перше, через відсутність у RKE2 наперед налаштованого динамічного розгортувача для постійних томів та ввімкнену в Rocky Linux підсистему SELinux у режимі примусового контролю, для роботи сховища компонентів моніторингу знадобилося розгорнути local-path та встановити відповідний контекст безпеки на каталогах зберігання на всіх вузлах кластера.

Для реалізації другого стовпа спостережуваності — логів — розгорнуто систему Loki разом з агентом збирання Promtail. Promtail розгортається на кожному вузлі кластера у вигляді DaemonSet, збирає логи всіх контейнерів і

передає їх до централізованого сховища Loki. На відміну від систем повнотекстового індексування, Loki індексує лише мітки потоків логів, що робить його ресурсоощадним і придатним для розгортання в межах модельної інфраструктури, як обґрунтовано в підрозділі 1.7. Завдяки використанню тієї самої моделі міток, що й у Prometheus, логи в Loki можна співвідносити з метриками за спільними мітками простору імен та пода.

Окремим науково-практичним аспектом функціонування розгорнутої системи є забезпечення високої точності дискретизації часових рядів (time-series data) при одночасному збереженні оптимального балансу обчислювальної потужності інфраструктури.

Оскільки підсистема моніторингу безперервно оперує великими масивами динамічних числових показників, що надходять від сотень кінцевих точок, внутрішня архітектура сховища даних Prometheus використовує спеціалізовані алгоритми блочного стиснення. Такий підхід дозволяє суттєво зменшити дисковий слід телеметричної інформації без втрати деталізації історичних даних.

Об'єднувальним компонентом підсистеми є Grafana, що слугує єдиним інтерфейсом для обох джерел даних. Prometheus підключено до Grafana як основне джерело метрик автоматично під час розгортання стеку, а Loki додано як додаткове джерело логів.

Це забезпечує єдину точку спостереження, у якій числові показники стану системи можна співвідносити з відповідними подіями в логах, що суттєво прискорює діагностику. Пакет kube-prometheus-stack постачається з набором наперед налаштованих інформаційних панелей, що одразу після розгортання надають візуалізацію стану кластера на різних рівнях — від споживання ресурсів окремими вузлами до показників навантаження в конкретному просторі імен. Інформаційну панель Grafana зі станом кластера зображено на рис. 3.25 та моніторинг застосунку зображено на рис. 3.26.

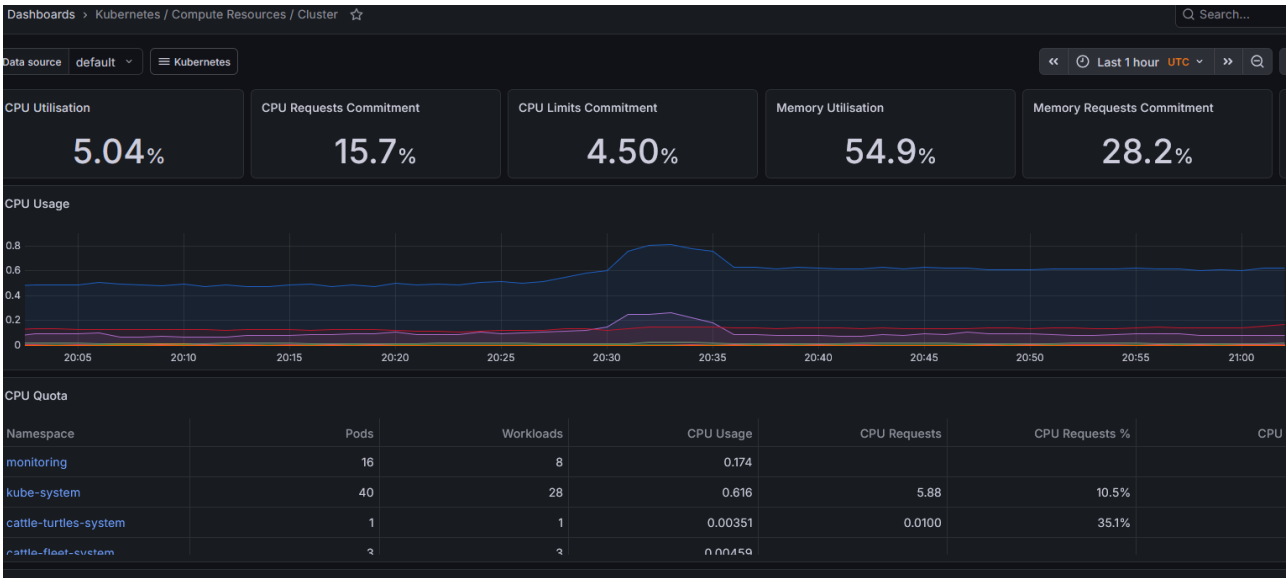


Рисунок 3.25 — Моніторинг з графаною

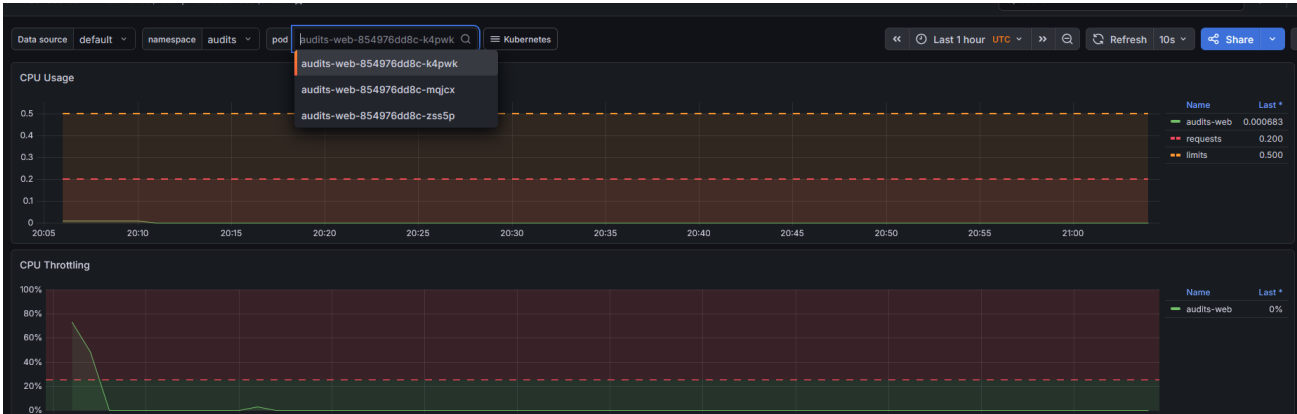


Рисунок 3.26 — Моніторинг додатку аудитів з графаною

По-друге, через прив'язку постійних томів до конкретних вузлів (node affinity), характерну для локального сховища, окремі компоненти потребували явного призначення на робочі вузли. Вирішення цих питань є практичним свідченням того, що розгортання виконувалося в захищеному, наближеному до промислового середовищі, а не в спрощеній конфігурації з вимкненими механізмами безпеки.

Важливим аспектом практичного втілення SRE-орієнтованого підходу в межах розгорнутого стеку є тонке налаштування правил сповіщень (Alerting Rules) у компоненті Alertmanager.

Спроектowana конфігурація передбачає ієрархічну маршрутизацію сигналів тривоги залежно від рівня критичності інциденту (Critical, Warning) та його потенційного впливу на цільові показники надійності (SLO). Завдяки

використанню вбудованих механізмів дедуплікації та групування сповіщень, система дозволяє уникнути явища «втоми від сповіщень» (alert fatigue) серед інженерного персоналу, об'єднуючи каскадні відмови суміжних інфраструктурних компонентів у єдині консолідовані інциденти. Математичні вирази мови запитів PromQL, покладені в основу тригерів, безперервно аналізують швидкість вигорання бюджету помилок (error budget burn rate).

е забезпечує можливість виявлення деградації продуктивності системи або критичного вичерпання апаратних ресурсів ще до моменту, коли це безпосередньо вплине на доступність сервісів для кінцевих користувачів. Завдяки цьому забезпечується перехід від традиційного реактивного адміністрування до проактивного управління інфраструктурою, що дозволяє нівелювати загрози стабільності на ранніх стадіях їхнього зародження. Такий підхід суттєво скорочує середній час виявлення аномалій та мінімізує ймовірність виникнення незапланованих простоїв, які могли б спричинити репутаційні чи операційні втрати. Таким чином, реалізовано повноцінну підсистему спостережуваності, що збирає метрики та логи з усіх рівнів інфраструктури й надає єдиний інтерфейс для контролю її стану. Це завершує побудову інфраструктури; перевірку її відмовостійкості наведено в наступному підрозділі.

3.8 Тестування відмовостійкості та оцінка ефективності інфраструктури

Завершальним етапом роботи є перевірка відмовостійкості реалізованої інфраструктури — її здатності зберігати працездатність у разі відмови окремих компонентів. Перевірку виконано шляхом контрольованої імітації відмов на різних рівнях інфраструктури з наступним спостереженням за поведінкою системи. Відповідно до принципу ешелонованої надійності, закладеного в підрозділі 2.1, перевірку проведено на трьох рівнях: рівні навантаження, рівні даних та рівні оркестрації.

Першим перевірено механізм самовідновлення на рівні навантаження. Імітацію відмови виконано примусовим видаленням одного з двох екземплярів

(подів) застосунку. Примусове видалення пода для імітації відмови показано на рис. 3.27.

```

Error from server (NotFound): pods "audits-web-854976dd8c-mq" not found
[ntyshkevych-local@frm01 new]$ kubectl -n audits get pods
NAME                                READY   STATUS    RESTARTS   AGE
audits-web-854976dd8c-mqjcx         1/1     Running   0           32m
audits-web-854976dd8c-zss5p         1/1     Running   0           33m
[ntyshkevych-local@frm01 new]$ kubectl -n audits delete pod audits-web-854976dd8c-mqjcx
pod "audits-web-854976dd8c-mqjcx" deleted from audits namespace

```

Рисунок 3.27 — Примусове видалення Pod для імітації відмови екземпляра навантаження

У результаті спостерігалось очікуване відновлення: контролер Deployment негайно виявив невідповідність фактичної кількості реплік бажаній і автоматично створив новий под на заміну видаленому, який за короткий час перейшов у стан готовності. Автоматичне відновлення пода контролером Deployment показано на рис. 3.28.

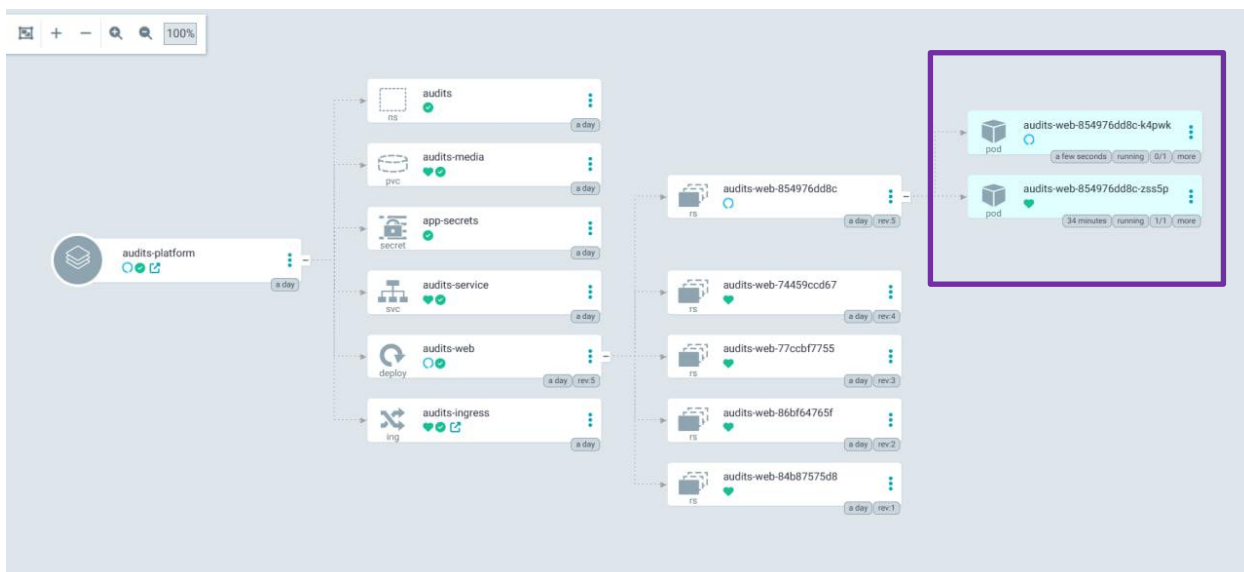


Рисунок 3.28 — Автоматичне відновлення Pod контролером Deployment

Протягом усього процесу застосунок залишався доступним, оскільки другий екземпляр продовжував обслуговувати запити. Це підтверджує коректну роботу механізму самовідновлення Kubernetes та доцільність розгортання навантаження у кількох репліках. Доступність застосунку під час відновлення показано на рис. 3.29. та стан пода після завершення самовідновлення наведено на рис. 3.30.

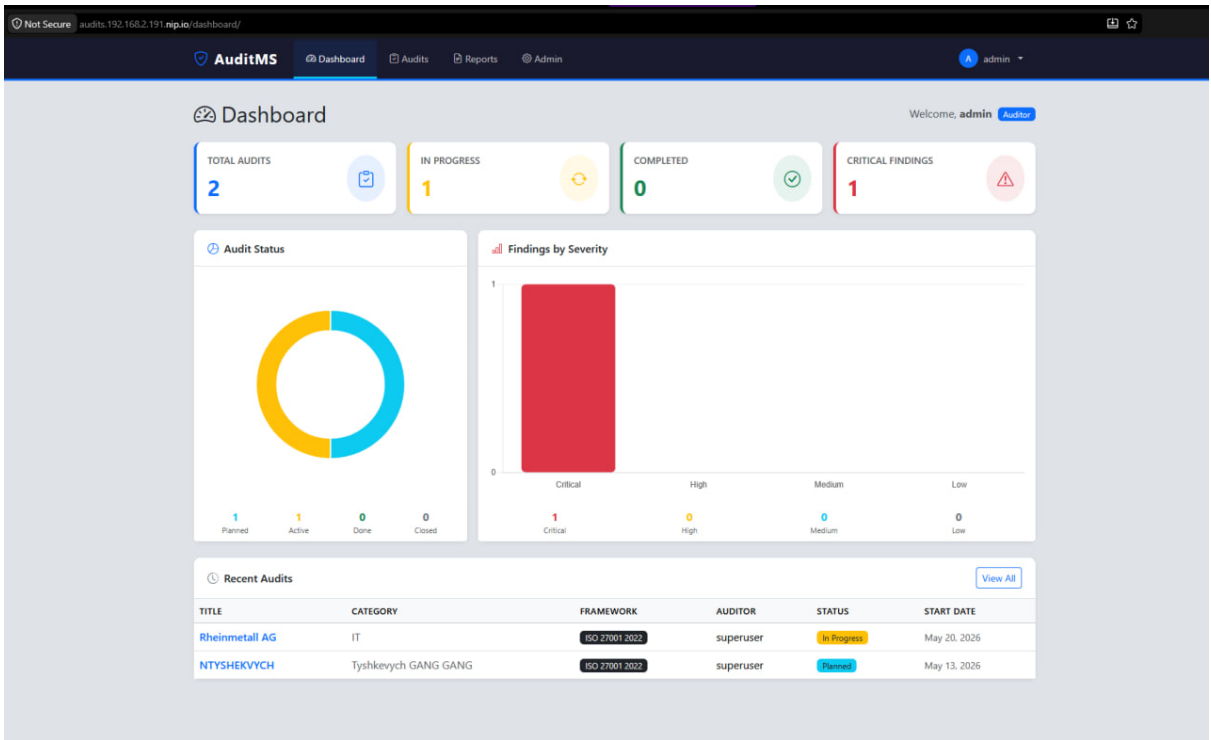


Рисунок 3.29 — Доступність застосунку під час відновлення після відмови

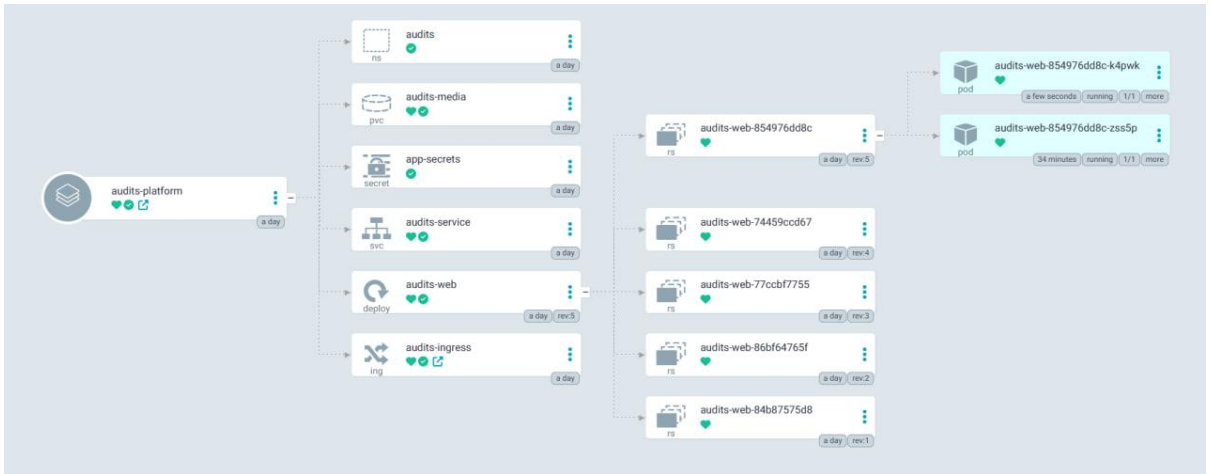


Рисунок 3.30 — Стан Pod після завершення процесу самовідновлення

Другим перевірено найкритичніший рівень — рівень даних. Імітацію відмови виконано зупинкою служби бази даних на одному з трьох вузлів кластера Galera. Зупинку служби бази даних для імітації відмови вузла показано на рис. 3.31 та стан кластера Galera після відмови одного з вузлів наведено на рис. 3.32.

```
new]$ ssh rocky@192.168.2.181 "sudo systemctl stop mariadb"
```

Рисунок 3.31 — Зупинка служби бази даних для імітації відмови вузла кластера

```
[rocky@localhost ~]$ sudo mariadb -e "SHOW STATUS LIKE 'wsrep_cluster_size';"
+-----+-----+
| Variable_name | Value |
+-----+-----+
| wsrep_cluster_size | 2 |
+-----+-----+
```

Рисунок 3.32 — Стан кластера Galera після відмови одного з вузлів

Після зупинки вузла перевірено стан кластера через системну змінну розміру кластера. Розмір кластера зменшився з трьох до двох вузлів, проте кластер зберіг працездатність, оскільки два справні вузли утворюють кворум (більшість). Балансувальник навантаження, відповідно до спроектованої стратегії «активний вузол — резерв», автоматично спрямував з'єднання на справний вузол, завдяки чому застосунок продовжив роботу з базою даних без переривання. Після повернення вузла до роботи він автоматично ресинхронізувався з кластером, відновивши його розмір до трьох. Відновлення кластера даних після повторного запуску вузла показано на рис. 3.33.

The screenshot shows a web-based security checklist interface. At the top, there's a 'Recommendation' section with a text input and a '+ Add Finding' button. Below it, a 'Checklist' section shows a summary: 'All Compliant' (green) and 'All Non-Compliant' (red) with a count of 15. The main part of the interface is a table with columns: STATUS, QUESTION, NOTES, and TOGGLE. The table lists several security items, most of which are marked as 'Compliant' (green checkmark). One item, '[A.8.1] Asset management', is marked as 'Non-Compliant' (red X). Each item has a corresponding note and a toggle button with checkmark, X, and question mark icons.

STATUS	QUESTION	NOTES	TOGGLE
✓	[A.5.1] Policies for information security	Check policy documents are signed off by management and reviewed in last 12 months.	✓ X ?
✓	[A.6.1] Internal organisation	Verify org chart shows named security roles. Check job descriptions include security duties.	✓ X ?
✓	[A.7.2] Information classification	Look for a classification scheme (Public/Internal/Confidential/Secret). Sample 5 documents.	✓ X ?
✗	[A.8.1] Asset management	Request the asset register. Check it includes owner, classification, and location.	✓ X ?
✓	[A.9.1] Access control policy	Check access is granted on least-privilege. Review user access rights quarterly.	✓ X ?
✓	[A.9.2] User access management	Verify onboarding/offboarding process. Check for any orphaned accounts in AD.	✓ X ?
✓	[A.10.1] Cryptographic controls	Check data at rest and in transit encryption. Verify key management procedure exists.	✓ X ?
✓	[A.11.1] Physical security	Inspect server room access logs. Check CCTV coverage and visitor sign-in records.	✓ X ?

Рисунок 3.33 — Відновлення кластера даних після повторного запуску вузла

Цей тест є практичним підтвердженням відмовостійкості рівня даних, спроектованої в підрозділі 2.5: відмова окремого вузла бази даних не призводить до втрати даних чи недоступності системи.

Третім перевірено відмовостійкість рівня оркестрації. Імітацію відмови виконано зупинкою служби RKE2 на робочому вузлі кластера: Відмову вузла кластера оркестрації та перепризначення навантаження показано на рис. 3.34.

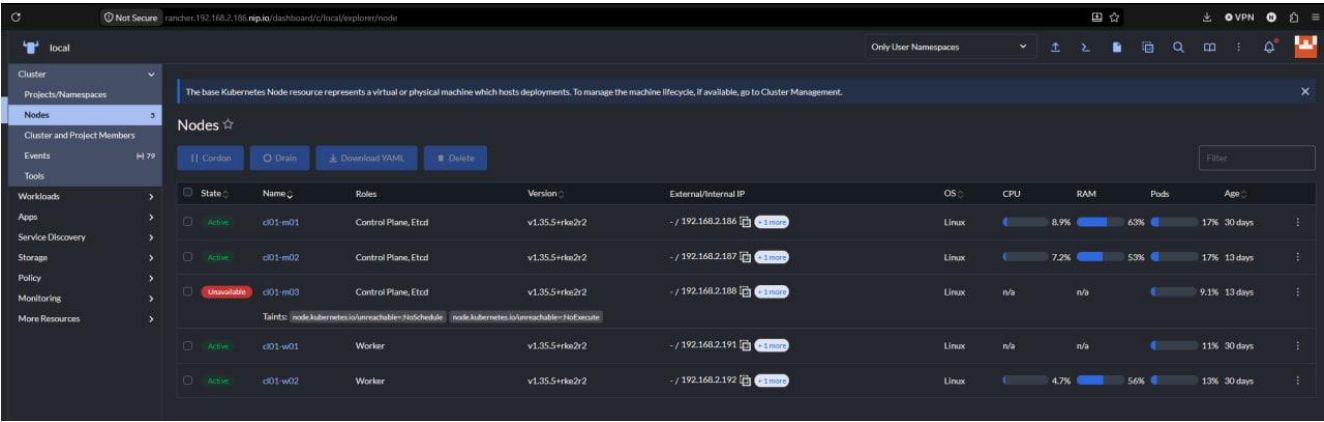


Рисунок 3.34 — Відмова вузла кластера оркестрації та перепризначення навантаження

Планувальник Kubernetes виявив недоступність вузла, позначив його відповідними мітками (taint) як недосяжний і перепланував розміщені на ньому поди на інший справний робочий вузол. Це підтверджує здатність кластера автоматично перерозподіляти навантаження в разі відмови вузла. Після повернення вузла до роботи він знову приєднався до кластера й став доступним для планування.

Окремо слід зазначити, що під час експлуатації інфраструктури спостерігалася й відмова на рівні керування — тимчасова недоступність керувального вузла. Завдяки високодоступній конфігурації з трьох керувальних вузлів та кворуму etcd кластер зберіг працездатність, а після відновлення вузла автоматично повернувся до повністю справного стану. Це є практичним підтвердженням відмовостійкості рівня керування. Стан застосунку показано на рис. 3.35.

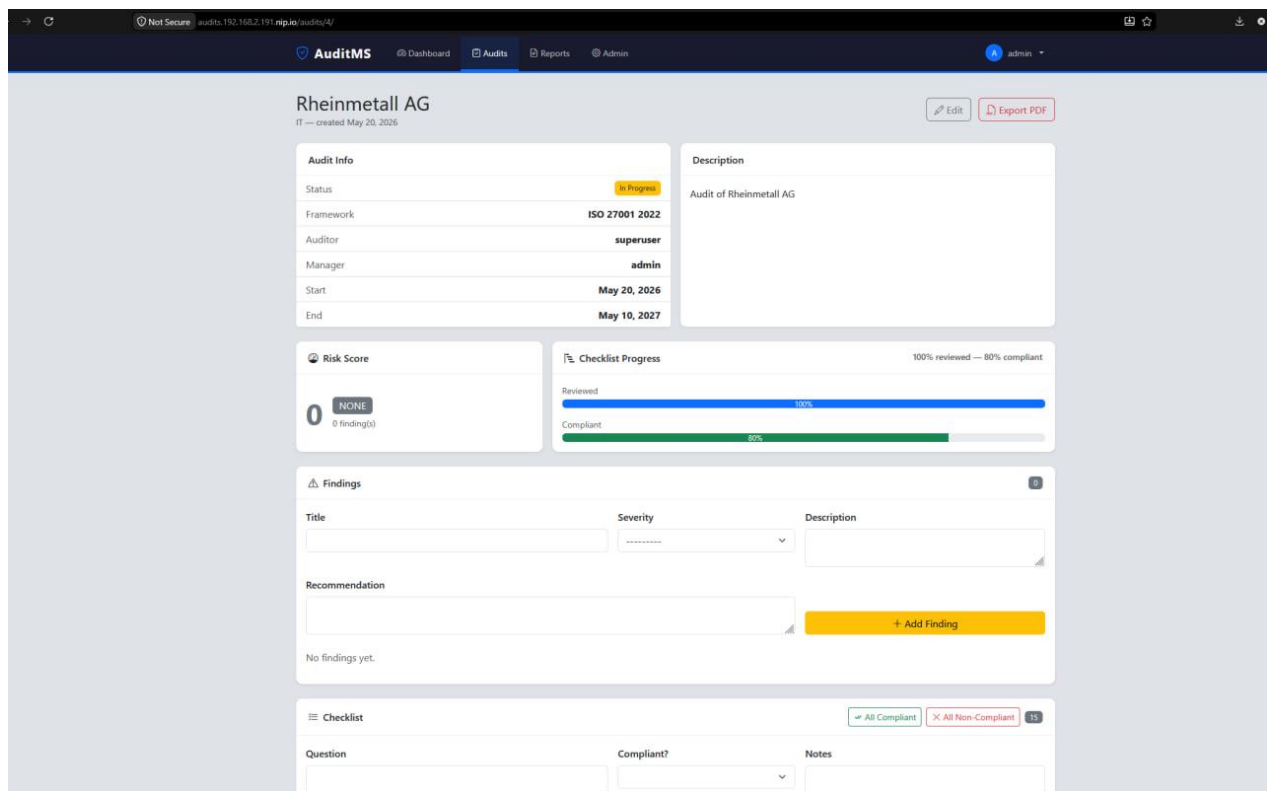


Рисунок 3.35 — Перенесення Pod планувальником Kubernetes на доступний вузол

Висновки до третього розділу

У третьому розділі описано практичну реалізацію спроектованої інфраструктури та перевірено її відповідність вимогам технічного завдання. Розгорнуто базове середовище віртуалізації на основі Proxmox VE із підготовленим шаблоном Rocky Linux 9 та механізмом cloud-init, що забезпечує єдність початкового стану всіх вузлів. Засобами Terraform автоматизовано розгортання віртуальних машин кластерів Galera та RKE2, а засобами Ansible — їх конфігурування; наскрізна передача стану між етапами через автоматично згенерований інвентарний файл усунула потребу в ручному веденні переліку вузлів. Відмовостійкість інфраструктури підтверджено імітацією відмов на трьох рівнях — навантаження, даних та оркестрації, — за результатами яких система щоразу зберігала працездатність і автоматично відновлювалася без ручного втручання.

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи розв'язано актуальне завдання проєктування та реалізації on-premise інфраструктури для системи керування аудитами із застосуванням методологій DevOps, SRE та Platform Engineering.

У процесі було проведено аналіз сучасних підходів до побудови інфраструктури, розглянуто обмеження класичного системного адміністрування та обґрунтовано доцільність застосування декларативних методологій. Виконано порівняльний аналіз інструментів і платформ у кожній категорії, за результатами якого сформовано технологічний стек реалізації та сформульовано технічне завдання.

У межах практичної частини було спроектовано інфраструктуру за принципом ешелонованої надійності з виділенням п'яти функціональних рівнів та топологією з дев'яти віртуальних машин. Засобами Terraform та Ansible автоматизовано розгортання і конфігурування всіх компонентів — кластера Galera, кластера оркестрації RKE2 з платформним шаром та балансувальника навантаження. Реалізовано модель GitOps через ArgoCD та впроваджено підсистему спостережуваності на основі Prometheus, Grafana і Loki.

Проведене експериментальне тестування підтвердило працездатність розробленої інфраструктури та її здатність зберігати працездатність при імітації відмов на рівнях навантаження, даних та оркестрації без ручного втручання.

Перспективними напрямками подальшого розвитку є забезпечення високої доступності балансувальника навантаження шляхом розгортання кластера HAProxy віртуальною IP-адресою, що усуне єдину точку відмови на рівні балансування; перехід з Terraform на OpenTofu як повністю відкриту альтернативу під управлінням Linux Foundation впровадження повноцінних CI/CD-конвеєрів для автоматизованого застосування інфраструктурного коду без ручного запуску з робочої станції адміністратора; перенесення стану Terraform до віддаленого сховища з підтримкою версіонування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. ISO/IEC 25010:2023. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Product quality model. Geneva : International Organization for Standardization, 2023.
2. Beyer B., Jones C., Petoff J., Murphy N. R. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol : O'Reilly Media, 2016. 552 с.
3. Data Sovereignty vs. Data Residency. IBM Think : вебсайт. URL: <https://www.ibm.com/think/topics/data-sovereignty-vs-data-residency> (дата звернення: 16.05.2026).
4. A Switch (Back) to On-Prem Solutions to Maximize Control & Security. Cyber Defense Magazine. 2025. URL: <https://www.cyberdefensemagazine.com/a-switch-back-to-on-prem-solutions-to-maximize-control-security/> (дата звернення: 17.05.2026).
5. Morris K. Infrastructure as Code: Managing Servers in the Cloud. Sebastopol : O'Reilly Media, 2016. 362 с.
6. Kim G., Humble J., Debois P., Willis J. The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations. Portland : IT Revolution Press, 2016. 480 с.
7. Humble J., Farley D. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston : Addison-Wesley, 2010. 512 с.
8. Verma A., Pedrosa L., Korupolu M., Oppenheimer D., Tune E., Wilkes J. Large-scale cluster management at Google with Borg. Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15). New York : ACM, 2015. С. 1–17.
9. Majors C., Fong-Jones L., Miranda G. Observability Engineering: Achieving Production Excellence. Sebastopol : O'Reilly Media, 2022. 318 с.
10. Skelton M., Pais M. Team Topologies: Organizing Business and Technology Teams for Fast Flow. Portland : IT Revolution Press, 2019. 240 с.
11. RKE2 — Security and Hardening Guide. Rancher (SUSE) : офіційна документація. URL: https://docs.rke2.io/security/hardening_guide (дата звернення: 22.05.2026).
12. Galera Cluster. MariaDB Documentation. URL: <https://mariadb.com/kb/en/galera-cluster/> (дата звернення: 31.05.2026).
13. Proxmox VE Documentation. Proxmox Server Solutions GmbH. URL: <https://pve.proxmox.com/pve-docs/> (дата звернення: 31.05.2026).

14. Grafana Loki Documentation. Grafana Labs. URL: <https://grafana.com/docs/loki/> (дата звернення: 24.05.2026).
15. Terraform Documentation. Meta-arguments: count. HashiCorp: офіційна документація. URL: <https://developer.hashicorp.com/terraform/language/meta-arguments/count> (дата звернення: 27.05.2026).
16. Ansible Documentation. Working with dynamic inventory. Red Hat: офіційна документація. URL: https://docs.ansible.com/ansible/latest/inventory_guide/intro_dynamic_inventory.html (дата звернення: 27.05.2026).
17. Rancher Documentation. Setting up Kubernetes Clusters in Rancher. SUSE: офіційна документація. URL: <https://ranchermanager.docs.rancher.com/pages-for-subchapters/kubernetes-clusters-in-rancher-setup> (дата звернення: 28.05.2026).
18. Platform Engineering: A Guide for Technical, Product, and People Leaders. Gartner Research, 2024. URL: <https://www.gartner.com/en/software-engineering/topics/platform-engineering>.
19. Argo CD Documentation. Getting Started. CNCF: офіційна документація. URL: https://argo-cd.readthedocs.io/en/stable/getting_started/ (дата звернення: 28.05.2026).
20. Helm Documentation. Charts. CNCF: офіційна документація. URL: <https://helm.sh/docs/topics/charts/> (дата звернення: 31.05.2026).

ДОДАТКИ

		Тишкевич Н.Ю.			ККІКБ.КР.Б – 123 – 15 – ПЗ	Арк.
		Миколайчук В.В.				111
Змн.	Арк.	№ докум.	Підпис	Дата		

Описові таблиці

Таблиця А.1

Порівняльний аналіз платформ віртуалізації

Критерій	Proxmox VE	VMware vSphere	Microsoft Hyper-V
Ліцензування	AGPLv3 (відкритий код)	Комерційна (Broadcom)	Комерційна (Microsoft)
Вартість	Безкоштовно / від €399/рік на вузол (підтримка)	від \$11 000+/сокет/рік (підписка Broadcom, 2024)	Входить у Windows Server (\$6 155+ ліцензія DC)
Гіпервізор + контейнери	KVM (Type-1) + LXC нативно	ESXi (Type-1); контейнери через Tanzu (платно)	Hyper-V (Type-1); Windows Containers (обмежено)
Кластеризація та міграція	Вбудована кластеризація до 32 вузлів; Live Migration безкоштовно	HA/DRS + vMotion — потребує vCenter Enterprise+	Failover Cluster; Live Migration потребує Windows DC
Сховище	Ceph, ZFS, NFS, iSCSI	VMFS, vSAN (дорого), NFS, iSCSI	CSV, SMB 3.0, iSCSI, Storage Spaces Direct
Резервне копіювання	Proxmox Backup Server	Veeam, Commvault	Azure Backup або Veeam (стороннє)
Відкритий код	Повністю відкритий;	Закритий код;	Закритий код;
Автоматизація	REST API, Terraform, Ansible (офіційні)	REST API, PowerCLI (зріла екосистема)	PowerShell / WMI;

Таблиця А.2

Порівняння інструментів контейнеризації

Критерій	Docker	Podman	containerd	LXC
Тип інструмента	Платформа складання та запуску	Платформа складання та запуску	Середовище виконання (runtime)	OS-рівнева віртуалізація
Архітектура	Фоновий демон dockerd	Бездемонна (fork-exec)	Системний демон	Системний (lxc)
Запуск без прав root	Опційно, потребує налаштування	Типово, «з коробки»	Через rootlesskit	Підтримується
Складання образу	docker build	podman build	BuildKit + nerdctl	Не передбачено
Опис складання	Dockerfile	Dockerfile	Dockerfile	Шаблони rootfs
Формат образів	OCI	OCI	OCI	Власний (LXD)
Інтеграція SELinux	Часткова	Вбудована	Через оркестратор	Підтримується
Сумісність команд	базова	повна з Docker	через nerdctl	—
Роль у проєкті	Технологія контейнеризації	Складання образу (фактично)	Виконання в кластері RKE2	Не застосовується

Таблиця А.3

Порівняння технологій моніторингу.

Критерій	Grafana Stack (Prometheus + Grafana + Loki + Tempo)	Elastic Stack (Elasticsearch + Logstash + Kibana)	SigNoz (OpenTelemetry + ClickHouse)
Призначення	Метрики, логи, трейси спеціалізованих компоненти	Єдина платформа для логів, метрик і пошуку	Єдина уніфікована платформа для метрик, логів.

Критерій	Grafana Stack (Prometheus + Grafana + Loki + Tempo)	Elastic Stack (Elasticsearch + Logstash + Kibana)	SigNoz (OpenTelemetry + ClickHouse)
Ліцензування	Apache 2.0 / AGPLv3	SSPL + Elastic License	MIT
Збір метрик	Prometheus	Metricbeat / Elastic Agent	OpenTelemetry Collector
Збір логів	Loki (індексує лише мітки, зберігає сирі логи)	Elasticsearch (повнотекстовий індекс усіх логів)	OpenTelemetry Collector / Fluent
Трейсинг	Tempo (OpenTelemetry-сумісний, нативно)	APM Server (платні функції в хмарі)	Нативно та безшовно через OpenTelemetry
Споживання ресурсів	Loki не індексує вміст логів. 1 GB мінімально, 2 GB рекомендовано	Elasticsearch індексує все, потребує значної RAM 4 GB мінімально, 16 GB рекомендовано	2 GB мінімально, 4 GB рекомендовано
Мова запитів	PromQL (метрики), LogQL (логи) — прості та потужні	Lucene / KQL — складніший синтаксис	ClickHouse SQL
Масштабування	Горизонтальне через Thanos / Mimir (опційно)	Шардинг, replica sets Elasticsearch	Горизонтальне
Вартість	Безкоштовно для self-hosted; Grafana Cloud — є безкоштовний tier	Self-hosted безкоштовно; розширені функції — платна підписка	Повністю безкоштовно для self-hosted

Лістинги коду

Таблиця Б.1

Підключення Terraform до платформи віртуалізації через її програмний інтерфейс

```
terraform {
  required_version = ">= 1.5.0"
  required_providers {
    proxmox = {
      source = "bpg/proxmox"
      version = "0.66.1"
    }
    local = {
      source = "hashicorp/local"
      version = "~> 2.0"
    }
  }
}

provider "proxmox" {
  endpoint = var.proxmox_endpoint
  api_token = var.proxmox_api_token
  insecure = var.proxmox_insecure
}
```

Таблиця Б.2

Опис ресурсу для створення трьох вузлів кластера даних шляхом клонування підготовленого шаблону.

```
resource "proxmox_virtual_environment_vm" "galera_nodes" {
  count      = var.galera_count
  name       = "gl01-n0${count.index + 1}"
  node_name  = var.target_node
  vm_id      = var.galera_vmid_start + count.index

  clone {
    vm_id = var.clone_vm_id
    full  = true
  }

  cpu { cores = 8, type = "host" }
  memory { dedicated = 8192 }

  disk {
    datastore_id = var.datastore_id
    interface    = "scsi0"
    size         = 50
  }

  disk {
    datastore_id = var.datastore_id
    interface    = "scsi1"
    size         = 15 # окремий диск під дані Galera (LVM)
  }

  network_device { bridge = "vmbr0" }
```

Опис ресурсів для створення трьох керувальних та двох робочих вузлів кластера

```

resource "proxmox_virtual_environment_vm" "master_nodes" {
  count      = var.master_count          # 3 – кворум etcd
  name      = "cl01-m0${count.index + 1}"
  vm_id     = var.master_vmid_start + count.index
  clone     { vm_id = var.clone_vm_id, full = true }
  cpu       { cores = 8, type = "host" }
  memory    { dedicated = 8192 }
  disk      { datastore_id = var.datastore_id, interface = "scsi0", size = 50 }
  network_device { bridge = "vmbr0" }
  initialization {
    ip_config {
      ipv4 {
        address = "${var.network_base}${var.master_ip_start + count.index}/24"
        gateway = var.network_gateway
      }
    }
  }
  lifecycle { ignore_changes = [clone, initialization, disk] }
}

resource "proxmox_virtual_environment_vm" "worker_nodes" {
  count      = var.worker_count          # 2 – робочі вузли
  name      = "cl01-w0${count.index + 1}"
  vm_id     = var.worker_vmid_start + count.index
  clone     { vm_id = var.clone_vm_id, full = true }
  cpu       { cores = 8, type = "host" }
  memory    { dedicated = 4096 }
  disk      { datastore_id = var.datastore_id, interface = "scsi0", size = 100 }
  network_device { bridge = "vmbr0" }
  initialization {
    ip_config {
      ipv4 {
        address = "${var.network_base}${var.worker_ip_start + count.index}/24"
        gateway = var.network_gateway
      }
    }
  }
  lifecycle { ignore_changes = [clone, initialization, disk] }
}

```

Таблиця Б.4

Механізм автоматичного формування інвентарного файлу для Ansible

```

resource "local_file" "ansible_inventory" {
  filename = "${path.module}/inventory.ini"
  content = <<-EOT
    [masters]
    %{ for vm in proxmox_virtual_environment_vm.master_nodes ~}
    ${vm.name} ansible_host=${split("/", vm.initialization[0].ip_config[0].ipv4[0].address)[0]}
    ansible_user=rocky
    %{ endfor ~}
    [workers]
    %{ for vm in proxmox_virtual_environment_vm.worker_nodes ~}
    ${vm.name} ansible_host=${split("/", vm.initialization[0].ip_config[0].ipv4[0].address)[0]}
    ansible_user=rocky
    %{ endfor ~}
    [cluster:children]
    masters
    workers
  EOT
}

```

Автоматизація виділення логічного тому, створення файлової системи та формування кластера.

```
- name: Create Logical Volume for Galera
  community.general.lvol:
    vg: galera-vg
    lv: galera-lv
    size: 10G

- name: Create XFS filesystem on galera-lv
  community.general.filesystem:
    fstype: xfs
    dev: /dev/galera-vg/galera-lv

- name: Mount galera-lv to MariaDB data directory
  ansible.posix.mount:
    path: /var/lib/mysql
    src: /dev/galera-vg/galera-lv
    fstype: xfs
    state: mounted

- name: Bootstrap the first MariaDB Cluster node
  ansible.builtin.command: galera_new_cluster
  when: inventory_hostname == groups[mariadb_cluster_group_name][0]

- name: Start MariaDB on joiner nodes
  ansible.builtin.service: { name: mariadb, state: started, enabled: true }
  when: inventory_hostname != groups[mariadb_cluster_group_name][0]
```

Шаблон конфігураційного файлу, що вмикає синхронну реплікацію та формує перелік вузлів кластера.

```
[mariadb]
wsrep_on                = ON
wsrep_provider           = /usr/lib64/galera-4/libgalera_smm.so
wsrep_cluster_name       = "{{ mariadb_cluster_name }}"
wsrep_cluster_address    = "gcomm://{{ groups[mariadb_cluster_group_name]
                             | map('extract', hostvars, 'ansible_host')
                             | join(',') }}"
wsrep_node_address       = "{{ ansible_host }}"
wsrep_sst_method          = rsync
bind-address             = 0.0.0.0

[mysqld]
binlog_format            = row
default_storage_engine   = InnoDB
innodb_autoinc_lock_mode = 2
```

Таблиця Б.7

```

# Перший керувальний вузол – ініціалізація кластера
- name: Install RKE2 Server (First Master)
  shell: curl -sL https://get.rke2.io | INSTALL_RKE2_TYPE="server" sh -
  args: { creates: /usr/local/bin/rke2 }
  when: inventory_hostname == groups['masters'][0]

- name: Read RKE2 node-token
  slurp: { src: /var/lib/rancher/rke2/server/node-token }
  register: rke2_token_b64
  when: inventory_hostname == groups['masters'][0]

# Робочі вузли – приєднання як агент
- name: Install RKE2 Agent (Workers)
  shell: curl -sL https://get.rke2.io | INSTALL_RKE2_TYPE="agent" sh -
  args: { creates: /usr/local/bin/rke2 }
  when: inventory_hostname in groups['workers']

```

Таблиця Б.8

Шаблон конфігурації балансувальника з двома незалежними фронтедами.

```

# Kubernetes API – рівномірний розподіл між керувальними вузлами
backend k8s_api_backend
  mode tcp
  balance roundrobin
  option tcp-check
{% for host in groups['masters'] %}
  server {{ host }} {{ hostvars[host]['ansible_host'] }}:6443 check
{% endfor %}

# Galera – стратегія "активний вузол – резерв" (один writer)
backend galera_backend
  mode tcp
  balance first
{% for host in groups['galera_nodes'] %}
{% if loop.first %}
  server {{ host }} {{ hostvars[host]['ansible_host'] }}:3306 check
{% else %}
  server {{ host }} {{ hostvars[host]['ansible_host'] }}:3306 check backup
{% endif %}
{% endfor %}

```

Таблиця Б.9

Dockerfile, що формує образ додатку

```

# --- Етап складання ---
FROM python:3.11-slim AS builder
WORKDIR /build
RUN apt-get update && apt-get install -y --no-install-recommends \
    gcc default-libmysqlclient-dev pkg-config \
    libpango-1.0-0 libcairo2 && rm -rf /var/lib/apt/lists/*
COPY requirements.txt .
RUN pip install --no-cache-dir --prefix=/install -r requirements.txt
COPY app/ /build/
RUN PYTHONPATH=/install/lib/python3.11/site-packages \
    DJANGO_SECRET_KEY=build-time-only-key \
    python manage.py collectstatic --noinput

```

```
FROM python:3.11-slim
WORKDIR /app
RUN apt-get update && apt-get install -y --no-install-recommends \
    default-mysql-client libmariadb3 libpango-1.0-0 libcairo2 \
    && rm -rf /var/lib/apt/lists/*
COPY --from=builder /install /usr/local
COPY --from=builder /build /app
RUN useradd -u 10001 django-user && chown -R django-user:django-user /app
USER 10001 # CIS: запуск від непривілейованого користувача
EXPOSE 8000
CMD ["gunicorn", "--bind", "0.0.0.0:8000", "--workers", "2", "myproject.wsgi:application"]
```

Таблиця Б.10

Маніфест розгортання застосунку у двох репліках.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: audits-web
  namespace: audits
spec:
  replicas: 2
  template:
    spec:
      # Міграції БД виконуються перед стартом основного контейнера
      initContainers:
        - name: migrate
          image: registry.gitlab.com/development2026/audits-platform-nt:latest
          command: ["python", "manage.py", "migrate", "--noinput"]
          envFrom: [{ secretRef: { name: app-secrets } }]
      containers:
        - name: audits-web
          image: registry.gitlab.com/development2026/audits-platform-nt:latest
          ports: [{ containerPort: 8000 }]
          envFrom: [{ secretRef: { name: app-secrets } }]
          securityContext:
            runAsNonRoot: true
            runAsUser: 10001
            capabilities: { drop: ["ALL"] }
```

Таблиця Б.11

Декларативний опис застосунку для оператора.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: audits-platform
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://gitlab.com/development2026/audits-platform-nt.git
    targetRevision: main
    path: k8s
  destination:
    server: https://kubernetes.default.svc
    namespace: audits
  syncPolicy:
    automated:
      prune: true # видаляти ресурси, прибрані з Git
      selfHeal: true # повертати кластер до стану з Git
```