

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до випускної кваліфікаційної роботи бакалавра освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)
на тему:
«Вебзастосунок для оркестрації гібридних хмарних інфраструктур»

Виконав студент групи ЗІПЗ 22-1
АНТОНЮК Андрій Русланович

Керівник роботи:
ВАКАЛЮК Тетяна Анатоліївна

Рецензент:
ЧИЖМОТРЯ Олексій Володимирович

Житомир – 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ЗАТВЕРДЖУЮ

Зав. кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«27» лютого 2026 р.

ЗАВДАННЯ

НА ВИПУСКНУ КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА

Здобувач вищої освіти: Антонюк Андрій Русланович

Керівник роботи: Вакалюк Тетяна Анатоліївна

Тема роботи: «Вебзастосунок для оркестрації гібридних хмарних інфраструктур» затверджена наказом № 93/с від 27 лютого 2026р.

Вихідні дані: технічна документація хмарних провайдерів AWS, Azure, GCP; специфікація REST API та WebSocket; стандарти NIST щодо хмарних систем.

Предметом дослідження є методи та засоби розробки вебзастосунків для управління та оркестрації гібридних хмарних інфраструктур. Дослідження спрямоване на вивчення технологій і підходів, що забезпечують централізоване управління ресурсами різних хмарних провайдерів та локальної інфраструктури, моніторинг стану системи, автоматизацію розгортання сервісів і підвищення ефективності адміністрування хмарного середовища.

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Вакалюк Т.А.	27 лютого 2026 р.	27 лютого 2026 р.
2	Вакалюк Т.А.	27 лютого 2026 р.	27 лютого 2026 р.
3	Вакалюк Т.А.	27 лютого 2026 р.	27 лютого 2026 р.

РЕФЕРАТ

Пояснювальна записка до випускної кваліфікаційної роботи бакалавра:
104 стор., 10 рис., 8 табл., 30 джерел, 5 додатків.

Об'єкт дослідження - процеси управління та оркестрації ресурсів гібридних хмарних інфраструктур у корпоративному середовищі.

Предмет дослідження - методи та засоби розробки вебзастосунків для управління гібридними хмарними інфраструктурами на базі стеку React + Node.js + PostgreSQL + Docker.

Мета роботи — розробка вебзастосунку для централізованого моніторингу та оркестрації контейнеризованих навантажень у гібридній інфраструктурі, що поєднує локальні сервери та Docker-хости, із забезпеченням єдиного вебінтерфейсу для спостереження, розгортання й керування ресурсами та архітектурною готовністю до інтеграції з API публічних хмарних провайдерів.

У першому розділі проведено огляд предметної галузі: розглянуто еволюцію хмарних обчислень, класифікацію типів хмарних інфраструктур, поняття оркестрації, а також порівняльний аналіз наявних рішень.

У другому розділі обґрунтовано вибір технологічного стеку розробки: React, Node.js/Express, PostgreSQL, Docker. Проведено порівняльний аналіз альтернативних технологій та описано архітектуру системи.

У третьому розділі детально описано функціональні можливості розробленого застосунку та надано інструкцію користувача для всіх модулів системи: авторизації, дашборду, моніторингу, управління контейнерами, журналу подій та візуалізації топології.

КЛЮЧОВІ СЛОВА: ХМАРНА ІНФРАСТРУКТУРА, ГІБРИДНА ХМАРА, ОРКЕСТРАЦІЯ, DOCKER, КОНТЕЙНЕРИЗАЦІЯ, REACT, NODE.JS, POSTGRESQL, SOCKET.IO, МОНІТОРИНГ, DEVOPS.

					ІІЗ.КР.Б – 121 – 26 – ІІЗ			
<i>Зм.</i>	<i>Арк</i>	<i>№ докум.</i>		<i>Дат</i>	Пояснювальна записка «Вебзастосунок для оркестрації гібридних хмарних інфраструктур»	<i>Літ.</i>	<i>Арк.</i>	<i>Аркушів</i>
<i>Розроб.</i>		Антонюк А.Р.					3	104
<i>Керівник</i>		Вакалюк Т.А.				Житомирська політехніка, Група ЗІПЗ 22-1		
<i>Рецензент.</i>		Чижмотря О.В.						
<i>Зав.кафедр</i>		Вакалюк Т.А.						

ABSTRACT

Explanatory note to the bachelor's qualification thesis: 104 pages, 10 figures, 8 tables, 30 references, 5 appendices.

Object of research – processes of management and orchestration of hybrid cloud infrastructure resources in a corporate environment.

Subject of research – methods and tools for developing web applications for managing hybrid cloud infrastructures based on the React + Node.js + PostgreSQL + Docker stack.

The purpose of the work is to develop a web application for the centralized monitoring and orchestration of containerized workloads across a hybrid infrastructure that combines local servers and Docker hosts, providing a unified web interface for observation, deployment, and resource management, with an architecture ready for integration with public cloud provider APIs.

The first chapter reviews the subject area: the evolution of cloud computing, the classification of cloud infrastructure types, the concept of orchestration, as well as a comparative analysis of existing solutions.

The second chapter substantiates the choice of the development technology stack: React, Node.js/Express, PostgreSQL, Docker. A comparative analysis of alternative technologies is carried out, and the system architecture is described.

The third chapter describes in detail the functional capabilities of the developed application and provides a user guide for all system modules: authorization, dashboard, monitoring, container management, event log, and topology visualization.

KEYWORDS: CLOUD INFRASTRUCTURE, HYBRID CLOUD, ORCHESTRATION, DOCKER, CONTAINERIZATION, REACT, NODE.JS, POSTGRESQL, SOCKET.IO, MONITORING, DEVOPS.

		АНТОНЮК А.Р.			ІІЗ.КР.Б – 121 – 26 – ІІЗ	4
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Зміст	
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ГАЛУЗІ.....	9
1.1 Поняття хмарних обчислень та їх еволюція	9
1.2 Гібридна хмарна інфраструктура та концепція оркестрації	12
1.3 Огляд наявних рішень та обґрунтування необхідності розробки	17
Висновки до розділу 1	21
РОЗДІЛ 2. ОПИС СТЕКУ ТЕХНОЛОГІЙ ТА АРГУМЕНТАЦІЯ ВИБОРУ	23
2.1 Архітектурний підхід та вибір технологічного стеку	23
2.2 Вибір та обґрунтування бекенд-технологій та СУБД.....	28
2.3 Допоміжні технології та структура взаємодії компонентів системи	31
2.4 Проектування та розгортання системи	34
Висновки до розділу 2.....	37
РОЗДІЛ 3. ОПИС ФУНКЦІОНАЛЬНИХ МОЖЛИВОСТЕЙ ТА ІНСТРУКЦІЯ КОРИСТУВАЧА.....	39
3.1 Встановлення, автентифікація користувачів та головний дашборд.....	39
3.2 Управління інфраструктурами, вузлами та моніторинг ресурсів	43
3.3 Логи, сповіщення, топологія та адміністрування	49
3.4 Тестування та забезпечення якості програмного забезпечення	54
Висновки до розділу 3.....	57
ВИСНОВКИ	59
ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	61
ДОДАТОК А.....	64
ДОДАТОК Б.....	65
ДОДАТОК В	68
ДОДАТОК Г.....	85
ДОДАТОК Д.....	98

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

API – Application Programming Interface (інтерфейс програмування застосунків).

CRUD – Create, Read, Update, Delete (створення, читання, оновлення та видалення даних).

JWT – JSON Web Token (токен автентифікації та авторизації користувачів).

RBAC – Role-Based Access Control (рольова модель керування доступом).

REST – Representational State Transfer (архітектурний стиль побудови вебсервісів).

ORM – Object-Relational Mapping (технологія взаємодії об'єктів програми з базою даних).

MVC – Model-View-Controller (архітектурний шаблон програмного забезпечення).

SPA – Single Page Application (односторінковий вебзастосунок).

WebSocket – протокол двостороннього обміну даними в реальному часі.

CORS – Cross-Origin Resource Sharing (механізм контролю доступу до ресурсів між різними доменами).

CI/CD – Continuous Integration / Continuous Delivery (безперервна інтеграція та доставка програмного забезпечення).

СУБД – Система управління базами даних.

БД – База даних.

ПЗ – Програмне забезпечення.

ІТ – Інформаційні технології.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	6
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

ВСТУП

Стрімкий розвиток хмарних обчислень та зростання складності IT-інфраструктур сучасних підприємств висувають нові вимоги до засобів управління та автоматизації. За даними звіту Flexera «State of the Cloud 2024», близько 73% організацій застосовують гібридну хмарну стратегію, а 89% — мультихмарну [26], поєднуючи ресурси публічних хмарних провайдерів та власної локальної інфраструктури.

Управління такими гетерогенними середовищами є складним завданням через відсутність єдиних стандартів, різномірність API хмарних провайдерів та необхідність координації ресурсів у різних середовищах одночасно. Наявні корпоративні рішення, такі як VMware vRealize чи IBM Cloud Pak, є дорогими та складними у впровадженні, що робить їх недоступними для більшості організацій.

Актуальність теми визначається необхідністю створення доступного, функціонального вебзастосунку для оркестрації гібридних хмарних інфраструктур [9], який забезпечить єдиний інтуїтивний інтерфейс для команд DevOps-інженерів та IT-адміністраторів різного рівня технічної підготовки [8]. Зростання потреби у фахівцях, компетентних у хмарних технологіях, підкреслюється й у науково-педагогічних дослідженнях, присвячених методикам їх вивчення та підготовки відповідних спеціалістів [28].

Мета дослідження - розробка вебзастосунку для централізованого управління та оркестрації гібридних хмарних інфраструктур на базі стеку React + Node.js + PostgreSQL + Docker.

Об'єкт дослідження - процеси управління та оркестрації ресурсів гібридних хмарних інфраструктур у корпоративному середовищі.

Предмет дослідження - методи та засоби розробки вебзастосунків для управління гібридними хмарними інфраструктурами.

Для досягнення поставленої мети в роботі вирішуються такі завдання:

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	7
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

- провести аналіз предметної галузі та наявних рішень для управління гібридними хмарними інфраструктурами;
- обґрунтувати вибір технологічного стеку для розробки застосунку;
- спроектувати архітектуру системи та структуру бази даних;
- реалізувати функціональні модулі застосунку: авторизацію, моніторинг, управління контейнерами, журнал подій, систему сповіщень та візуалізацію топології;
- розгорнути застосунок у Docker-контейнерах та опублікувати код у репозиторії GitLab.

Практична значущість роботи полягає у створенні функціонального вебзастосунку, що дозволяє командам DevOps та IT-адміністраторам ефективно керувати гетерогенними хмарними середовищами через єдиний веб-інтерфейс.

Структура роботи. Пояснювальна записка складається зі вступу, трьох розділів, загальних висновків, переліку використаних джерел та додатків. Загальний обсяг роботи - 104 сторінок.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	8
		Вакалюк Т.А.				
Змін.	Арк.	№ докум.	Підпис	Дата		

РОЗДІЛ 1. ОГЛЯД ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Поняття хмарних обчислень та їх еволюція

Хмарні обчислення - це модель надання обчислювальних ресурсів (серверів, сховища даних, мережевих можливостей, програмного забезпечення) через мережу Інтернет за принципом «як сервіс». Замість того, щоб підтримувати власну фізичну інфраструктуру, організації та приватні користувачі отримують доступ до необхідних ресурсів від хмарних провайдерів, сплачуючи лише за фактичне використання.

Національний інститут стандартів і технологій США (NIST) визначає хмарні обчислення як модель [23], що забезпечує зручний мережевий доступ на вимогу до спільного пулу налаштовуваних обчислювальних ресурсів, які можна швидко виділяти та вивільняти з мінімальними зусиллями з управління або взаємодії з постачальником послуг. Це визначення залишається актуальним і лежить в основі сучасного розуміння галузі.

Еволюція хмарних обчислень бере свій початок ще з 1960-х років, коли Джон Маккарті висловив ідею про те, що обчислення можуть надаватися суспільству як комунальна послуга. Проте практична реалізація цієї концепції стала можливою лише на початку 2000-х років із розвитком широкосмугового доступу до Інтернету та технологій віртуалізації.

Наприкінці 1990-х років розвиток технологій віртуалізації створив технологічну базу для появи перших комерційних хмарних платформ.

У 2002 році компанія Amazon запустила Amazon Web Services (AWS) — набір хмарних сервісів, що поклав початок ері комерційних хмарних платформ.

У 2006 році було представлено Amazon EC2 та Amazon S3 які стали першими масово доступними хмарними інфраструктурними сервісами. Це відкрило нову еру в розвитку інформаційних технологій, коли будь-яка компанія могла отримати доступ до потужної обчислювальної інфраструктури без значних капітальних витрат.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	9
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Протягом 2008–2012 років ринок хмарних обчислень стрімко розвивався. Google запустив Google App Engine (2008), Microsoft представив платформу Azure (2010), а Salesforce популяризував концепцію Software as a Service (SaaS) [13]. Саме в цей період сформувалася класична триярусна модель хмарних послуг: IaaS (Infrastructure as a Service), PaaS (Platform as a Service) та SaaS (Software as a Service).

Починаючи з 2013 року, стрімкого розвитку набула технологія контейнеризації. Поява Docker у 2013 році та Kubernetes у 2014 році кардинально змінила підходи до розгортання та управління застосунками в хмарному середовищі [19]. Контейнери забезпечили однорідне середовище виконання незалежно від підлеглої інфраструктури, що значно спростило процеси розробки та розгортання програмного забезпечення.

Сучасний етап розвитку хмарних обчислень характеризується кількома ключовими тенденціями. По-перше, активним поширенням мультихмарних та гібридних стратегій, коли організації використовують ресурси одразу кількох хмарних провайдерів. По-друге, розвитком технологій граничних обчислень (edge computing), що дозволяють обробляти дані ближче до місця їх виникнення. По-третє, широким впровадженням принципів Infrastructure as Code (IaC) та автоматизації управління інфраструктурою.

За даними аналітичної компанії Gartner, обсяг світового ринку хмарних послуг у 2023 році перевищив 590 мільярдів доларів США, а до 2027 року очікується його зростання до 1 трильйона доларів [26]. Це свідчить про те, що хмарні обчислення стали невід'ємною частиною сучасної цифрової інфраструктури підприємств будь-якого масштабу [14].

Розглянувши еволюцію хмарних обчислень, можна перейти до класифікації сучасних хмарних інфраструктур за моделлю розгортання.

Залежно від моделі розгортання та рівня доступу, сучасні хмарні інфраструктури поділяються на чотири основних типи: публічна хмара, приватна хмара, гібридна хмара та мультихмарне середовище (multi-cloud). Кожен із цих

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	10
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

типів має власні характеристики, переваги та обмеження, що визначають їх застосування в конкретних бізнес-контекстах.

Публічна хмара - це хмарне середовище, яким управляє сторонній провайдер і яке надається клієнтам через Інтернет. Ресурси в публічній хмарі є спільними між кількома клієнтами (мультиорендна модель), хоча ізоляція між ними забезпечується засобами віртуалізації. Провідними провайдерами публічної хмари є Amazon Web Services, Microsoft Azure та Google Cloud Platform. Головними перевагами публічної хмари є висока масштабованість, відсутність необхідності управління фізичним обладнанням, оплата за фактичне використання та глобальна доступність.

Приватна хмара - це хмарна інфраструктура, що розгортається виключно для однієї організації та може розміщуватись як на її власних потужностях, так і в дата-центрах третіх сторін. На відміну від публічної хмари, приватна хмара забезпечує повний контроль над інфраструктурою, вищий рівень безпеки та відповідності вимогам регуляторів. Це робить її привабливою для організацій із суворими вимогами до конфіденційності даних, зокрема фінансових установ, медичних організацій та урядових структур. Серед популярних платформ для побудови приватної хмари варто відзначити OpenStack, VMware vSphere та Microsoft Azure Stack.

Мультихмарне середовище передбачає використання послуг двох або більше незалежних хмарних провайдерів для виконання різних завдань. Така стратегія дозволяє уникнути прив'язки до одного постачальника (vendor lock-in), оптимізувати витрати, використовуючи найкращі пропозиції різних провайдерів, а також підвищити стійкість системи до відмов. За даними дослідження HashiCorp State of Cloud Strategy Survey 2023, 76% організацій використовують мультихмарний підхід у своїй діяльності [22].

Гібридна хмара поєднує елементи публічної та приватної хмар, забезпечуючи можливість переміщення даних і застосунків між ними залежно від поточних потреб. Детальніше цей тип інфраструктури буде розглянуто в наступному підрозділі як основний предмет дослідження цієї роботи.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	11
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Узагальнюючи розглянуті типи хмарних інфраструктур, варто зазначити, що Національний інститут стандартів і технологій США (NIST) виділяє п'ять основних характеристик, які відрізняють хмарні обчислення від традиційних моделей надання ІТ-послуг: самообслуговування за вимогою (on-demand self-service), коли користувач може отримати ресурси без участі персоналу провайдера; широкий мережевий доступ (broad network access) через стандартні протоколи з різних типів клієнтських пристроїв; об'єднання ресурсів у пул (resource pooling) з динамічним розподілом між клієнтами за принципом мультиорендності; швидка еластичність (rapid elasticity), що дозволяє масштабувати ресурси практично миттєво відповідно до навантаження; та вимірюваність послуг (measured service), коли використання ресурсів автоматично контролюється й оптимізується на основі метрик.

Вибір конкретної моделі розгортання хмарної інфраструктури - публічної, приватної, гібридної чи мультихмарної - визначається сукупністю критеріїв, серед яких вимоги до конфіденційності та відповідності регуляторним нормам, прогнозованість навантаження, бюджетні обмеження, наявність уже наявної локальної інфраструктури та рівень технічної експертизи команди. Для організацій, що мають значні капітальні інвестиції у власне обладнання та водночас потребують гнучкості публічної хмари для масштабування під час пікових навантажень, гібридна модель є оптимальним компромісом, що пояснює її домінування серед корпоративних стратегій управління ІТ-інфраструктурою.

1.2 Гібридна хмарна інфраструктура та концепція оркестрації

Гібридна хмарна інфраструктура - це інтегроване обчислювальне середовище, що поєднує ресурси приватної хмари або локальних серверів (on-premises) із ресурсами одного або кількох публічних хмарних провайдерів. Ключовою характеристикою гібридної хмари є наявність оркестраційного рівня [1][6][7], що забезпечує єдине управління, переміщення навантаження та обмін даними між різними середовищами.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	12
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Концепція гібридної хмари виникла як відповідь на практичні потреби підприємств, яким необхідно поєднати переваги публічної хмари (гнучкість, масштабованість, економічність) зі збереженням контролю над критично важливими даними та застосунками у власній інфраструктурі. Отже, гібридна хмара не є простою сумою публічної та приватної, а утворює якісно нову архітектуру з власними закономірностями та характеристиками.

Серед ключових переваг гібридної хмарної інфраструктури варто виділити наступні. Гнучкість масштабування дозволяє організаціям зберігати базові навантаження у власній інфраструктурі [10], а у пікові моменти динамічно розширювати ресурси за рахунок публічної хмари - цей підхід отримав назву «хмарний сплеск». Оптимізація витрат досягається завдяки раціональному розподілу навантажень: постійні робочі процеси виконуються на власному обладнанні, а тимчасові або нерегулярні - у публічній хмарі з оплатою за фактичне використання.

Відповідність регуляторним вимогам є критично важливою перевагою для галузей із суворим законодавчим регулюванням. Наприклад, банківські установи можуть зберігати персональні дані клієнтів у приватній хмарі відповідно до вимог GDPR чи вітчизняного законодавства про захист персональних даних, водночас використовуючи публічну хмару для аналітики та розробки. Підвищена відмовостійкість забезпечується можливістю реалізації гео-розподілених резервних копій та аварійного відновлення (disaster recovery) з використанням ресурсів публічної хмари як резервного майданчика.

Водночас гібридна хмарна інфраструктура пов'язана з низкою викликів. Складність управління є головним із них: адміністратори мають справу з гетерогенним середовищем, де різні компоненти можуть використовувати різні API, інструменти управління та моделі безпеки. Забезпечення безпеки та відповідності вимогам у гібридному середовищі також значно складніше, ніж в однорідній інфраструктурі, адже потребує координації заходів безпеки між приватним та публічним компонентами. Мережева інтеграція та забезпечення

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	13
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

надійної і захищеної з'єднаності між локальними та хмарними ресурсами вимагає значних технічних зусиль.

Саме ці виклики зумовлюють нагальну потребу в спеціалізованих інструментах для управління та оркестрації гібридних хмарних інфраструктур, що є предметом дослідження цієї роботи.

Описавши сутність та переваги гібридної хмарної інфраструктури, перейдемо до розгляду концепції оркестрації як ключового механізму управління такими середовищами.

Оркестрація хмарних інфраструктур - це автоматизоване управління складними взаємопов'язаними хмарними процесами та компонентами, що включає виділення ресурсів, налаштування мережі, розгортання застосунків, моніторинг та масштабування [10]. На відміну від простої автоматизації, яка передбачає виконання окремих завдань, оркестрація координує взаємодію між численними автоматизованими процесами для досягнення складної мети.

Оркестрацію хмарних інфраструктур часто плутають із суміжним поняттям «автоматизація». Якщо автоматизація стосується виконання окремих дій без участі людини (наприклад, автоматичного запуску сервера), то оркестрація забезпечує координацію послідовності взаємопов'язаних дій для виконання складного робочого процесу (наприклад, розгортання повного стека застосунку з усіма залежностями, мережевими правилами та конфігураціями).

Сучасна оркестрація хмарних інфраструктур базується на кількох ключових принципах. Принцип Infrastructure as Code (IaC) передбачає опис бажаного стану інфраструктури у декларативній формі за допомогою коду, що зберігається у системах контролю версій. Це забезпечує відтворюваність, версіювання та аудитабельність інфраструктурних змін. Принцип незмінної інфраструктури (immutable infrastructure) полягає в тому, що замість модифікації наявних компонентів відбувається їх повна заміна новими, що знижує ризик появи непередбачуваних конфігураційних дрейфів.

Декларативний підхід до управління конфігурацією, реалізований у таких інструментах як Terraform та Kubernetes [20], дозволяє описувати бажаний

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	14
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

кінцевий стан системи, залишаючи визначення конкретних кроків для досягнення цього стану на відкуп оркестратору. Це суттєво спрощує управління порівняно з імперативним підходом, де адміністратор має явно описати кожну дію.

Ключовими функціональними можливостями систем оркестрації є: автоматичне виділення та звільнення ресурсів, динамічне масштабування залежно від навантаження [10][11], управління конфігурацією та забезпечення узгодженості стану, оркестрація розгортань (rolling updates, blue-green deployments, canary releases), самовідновлення (self-healing) при виявленні відмов, а також збір метрик та організація спостережуваності (observability).

У контексті гібридних середовищ оркестрація набуває особливої складності, оскільки потребує координації дій в різномірних середовищах з різними API та моделями безпеки. Саме тут виникає потреба в єдиній платформі управління, здатній абстрагувати відмінності між різними хмарними провайдерами та локальною інфраструктурою.

Технічна реалізація з'єднання між приватною та публічною складовими гібридної хмари здійснюється кількома основними способами. Найпростішим є з'єднання через мережу Інтернет із застосуванням віртуальних приватних мереж (VPN) на основі протоколів IPsec або WireGuard, що забезпечує шифрування трафіку при відносно низькій вартості впровадження. Для організацій з вищими вимогами до пропускної здатності та стабільності затримок провідні хмарні провайдери пропонують виділені канали типа AWS Direct Connect, Azure

ExpressRoute або Google Cloud Interconnect, які встановлюють приватне фізичне з'єднання між локальною інфраструктурою та хмарним дата-центром, мінуючи публічний Інтернет. Вибір способу з'єднання безпосередньо впливає на архітектуру оркестраційного рівня, оскільки визначає допустиму латентність між компонентами розподіленої системи.

Інструменти оркестрації хмарних інфраструктур можна умовно розділити за рівнем абстракції на три категорії. Інструменти управління конфігурацією (Ansible, Chef, Puppet) забезпечують приведення окремих серверів до бажаного

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	15
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

стану шляхом виконання визначених сценаріїв. Інструменти оркестрації контейнерів (Kubernetes, Docker Swarm) керують життєвим циклом контейнеризованих застосунків у кластері вузлів, включно з плануванням розміщення, масштабуванням та самовідновленням. Платформи Infrastructure as Code (Terraform, Pulumi, AWS CloudFormation) описують та створюють саму інфраструктуру - віртуальні машини, мережі, диски - у декларативній формі. Розроблений вебзастосунок поєднує елементи другої та третьої категорій, надаючи єдиний інтерфейс для управління як інфраструктурними ресурсами, так і контейнеризованими навантаженнями на них.

Варто також зазначити, що оркестрація гібридних хмарних середовищ суттєво відрізняється від управління однорідними інфраструктурами. Гетерогенність середовища вимагає від оркестраційної платформи підтримки різноманітних API хмарних провайдерів, протоколів автентифікації та моделей безпеки. Наприклад, AWS використовує IAM (Identity and Access Management) з власним синтаксисом політик, тоді як Azure — RBAC на основі Azure Active Directory, а локальна інфраструктура може застосовувати зовсім інші механізми контролю доступу. Ефективна оркестрація повинна абстрагувати ці відмінності, надаючи адміністратору єдиний нормалізований інтерфейс. Саме тому сучасні оркестраційні платформи, такі як Terraform та HashiCorp Vault, реалізують концепцію «провайдерів» — модульних компонентів, що інкапсулюють специфіку кожного зовнішнього сервісу.

Окремої уваги заслуговує питання моніторингу та спостережуваності (observability) в оркестрованих гібридних середовищах. Концепція спостережуваності включає три ключові складові: метрики (кількісні показники стану системи, наприклад завантаження CPU або обсяг вільної пам'яті), журнали подій (логи, що фіксують конкретні події та помилки з часовими мітками) та трасування (distributed tracing, що дозволяє відстежити шлях окремого запиту крізь ланцюжок мікросервісів). У гібридних середовищах ці дані надходять з різноманітних джерел — хмарних провайдерів, контейнерів, віртуальних машин та фізичних серверів — що вимагає централізованої агрегації та нормалізації перед

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	16
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

подальшим аналізом. Популярні стеки моніторингу для гібридних середовищ включають Prometheus для збору метрик, Grafana для їхньої візуалізації та Loki або ELK Stack (Elasticsearch, Logstash, Kibana) для аналізу журналів подій. Розроблений вебзастосунок реалізує власну підсистему моніторингу, що агрегує метрики з підключених вузлів та відображає їх у реальному часі за допомогою технології Socket.IO (поверх протоколу WebSocket).

Важливим аспектом оркестрації є також управління мережевою взаємодією між компонентами розподіленої системи. У гібридних середовищах мережева архітектура ускладнюється необхідністю забезпечення безпечного та надійного каналу зв'язку між приватним та публічним сегментами. Концепція Software-Defined Networking (SDN) та мережевих політик Kubernetes (NetworkPolicy) дозволяють декларативно описувати дозволений трафік між компонентами, мінімізуючи поверхню атаки. Сервісні мережі (service mesh) — зокрема Istio та Linkerd — надбудовуються над базовою мережевою інфраструктурою та забезпечують взаємну автентифікацію сервісів (mTLS), балансування навантаження, автоматичний retry та circuit breaking. Отже, повноцінна оркестрація гібридної інфраструктури охоплює не лише управління обчислювальними ресурсами, але й контроль над мережею, безпекою та спостережуваністю всіх компонентів системи.

1.3 Огляд наявних рішень та обґрунтування необхідності розробки

На сучасному ринку існує значна кількість інструментів та платформ для управління гібридними хмарними інфраструктурами. Аналіз наявних рішень є важливим кроком для визначення особливостей та конкурентних переваг розробленого застосунку.

Terraform від HashiCorp є одним із найпоширеніших інструментів класу Infrastructure as Code для управління хмарними ресурсами [3][20]. Він підтримує понад 1000 провайдерів, включаючи всі провідні публічні хмари та локальні рішення. Terraform використовує власну декларативну мову конфігурації HCL (HashiCorp Configuration Language) і дозволяє описувати, планувати та

		Антонюк А.Р.			ІІЗ.КР.Б – 121 – 26 – ПЗ	17
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

застосовувати зміни інфраструктури. Серед обмежень Terraform варто відзначити відсутність вбудованого веб-інтерфейсу для кінцевих користувачів (Terraform Enterprise та Terraform Cloud вирішують цю проблему частково), значний поріг входу для нетехнічних користувачів та складність управління станом у великих командах.

Kubernetes (K8s) - відкрита платформа оркестрації контейнерів [4], розроблена Google і передана під управління Cloud Native Computing Foundation (CNCF) [21][29]. Kubernetes забезпечує автоматизоване розгортання, масштабування та управління контейнеризованими застосунками [12][2]. Проєкти типу Rancher та OpenShift розширюють Kubernetes до повноцінних платформ управління гібридними середовищами. Головним обмеженням Kubernetes є висока складність адміністрування та значна крива навчання, що ускладнює його використання невеликими командами без спеціалізованої експертизи.

Portainer - відкрита платформа управління контейнерами з графічним веб-інтерфейсом, що підтримує Docker, Docker Swarm та Kubernetes. Portainer значно спрощує управління контейнерними середовищами завдяки інтуїтивному інтерфейсу. Однак його функціональність зосереджена переважно на управлінні контейнерами і не охоплює повний спектр завдань оркестрації гібридної інфраструктури, зокрема управління хмарними ресурсами різних провайдерів.

AWS Management Console, Azure Portal та Google Cloud Console є нативними інструментами управління відповідних хмарних провайдерів. Вони надають зручний веб-інтерфейс для управління ресурсами в межах однієї платформи, проте за своєю природою не підходять для управління гетерогенними гібридними середовищами - кожен із них обмежений екосистемою власного провайдера.

Pulumi - сучасна альтернатива Terraform [20], що дозволяє описувати інфраструктуру за допомогою популярних мов програмування (JavaScript, TypeScript, Python, Go, C#).

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	18
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Це знижує поріг входу для розробників, які вже знайомі з цими мовами. Однак, як і Terraform,

Pulumi є переважно CLI-інструментом без розвиненого вбудованого вебінтерфейсу для кінцевих користувачів.

Аналіз наявних рішень виявляє ряд незаповнених ніш: відсутність єдиної веб-платформи з інтуїтивним інтерфейсом для управління гетерогенними гібридними середовищами в реальному часі, складність для нетехнічних операторів, висока вартість комерційних рішень корпоративного рівня та відсутність вбудованої візуалізації топології гібридної інфраструктури у більшості інструментів.

Таблиця 1.1.

Порівняльний аналіз наявних рішень для управління гібридними хмарними інфраструктурами

Веб-інтерфейс	Частково	Так (Portainer)	Так	Ні	Так
Мульти-провайдер	Так	Так	Ні	Так	Так
Моніторинг в реальному часі	Ні	Так	Частково	Ні	Так
Візуалізація топології	Ні	Частково	Ні	Ні	Так
Управління контейнерами	Ні	Так	Частково	Ні	Так
Вартість	Безкоштовно	Безкоштовно	Платно	Безкоштовно	Безкоштовно
Поріг входу	Високий	Середній	Низький	Середній	Низький

На основі проведеного огляду наявних рішень доцільно обґрунтувати необхідність розробки власного вебзастосунку.

На основі проведеного аналізу предметної галузі та огляду наявних рішень можна сформулювати чітке обґрунтування необхідності розробки спеціалізованого вебзастосунку для оркестрації гібридних хмарних інфраструктур.

Перше це, зростання складності ІТ-інфраструктур підприємств. Сучасні організації середнього та великого бізнесу, як правило, використовують комбінацію локальних серверів, ресурсів приватної хмари та послуг одного або кількох публічних хмарних провайдерів.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	19
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Управління такою гетерогенною інфраструктурою за допомогою окремих нативних інструментів кожного провайдера є надзвичайно неефективним і призводить до операційних помилок.

По-друге, брак доступних рішень для команд середнього розміру. Наявні корпоративні платформи для управління гібридними хмарами є дорогими та складними у впровадженні, що робить їх недоступними для більшості організацій. З іншого боку, безкоштовні інструменти з відкритим кодом потребують значної технічної експертизи для налаштування та підтримки.

По-третє, зростання попиту на DevOps та автоматизацію [8]. Впровадження практик DevOps та Site Reliability Engineering (SRE) в організаціях підвищує попит на зручні інструменти автоматизації розгортання та управління інфраструктурою з веб-інтерфейсом, що дозволяють командам з різними технічними компетенціями ефективно співпрацювати.

Четверте, тренд на візуалізацію інфраструктури. Сучасні DevOps-команди та IT-менеджери дедалі більше потребують наочної візуалізації стану та топології своїх інфраструктур для швидкого розуміння поточної ситуації та прийняття обґрунтованих рішень.

Розроблений вебзастосунок для оркестрації гібридних хмарних інфраструктур покликаний заповнити визначену нішу, надавши командам інтуїтивний, доступний та функціональний інструмент для централізованого управління гетерогенними хмарними середовищами. Застосунок спрощує складні операції з оркестрації до рівня, доступного операторам із різними технічними компетенціями, водночас зберігаючи необхідну глибину функціональності для досвідчених адміністраторів.

Окрім розглянутих рішень, варто також згадати платформи Rancher та Red Hat OpenShift, які надбудовуються над Kubernetes і пропонують розширений веб-інтерфейс для управління кластерами, проте успадковують високий поріг входу базової платформи та орієнтовані переважно на роботу з контейнерами, залишаючи поза увагою управління віртуальними машинами та локальними серверами, що не входять до складу Kubernetes-кластера. Комерційні платформи

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	20
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

управління мультихмарними середовищами, такі як Morpheus Data чи Flexera One, пропонують широкий функціонал, проте їх вартість ліцензування, що нерідко розраховується на основі кількості керованих вузлів, робить їх економічно недоцільними для малих та середніх команд, бюджет яких обмежений кількома тисячами доларів на рік на інфраструктурні інструменти.

Отже, проведений аналіз дозволяє сформулювати чіткий перелік вимог до розробленого вебзастосунку: наявність єдиного веб-інтерфейсу для роботи з різнорідними ресурсами (хмарні провайдери, локальні сервери, Docker-хости); підтримка моніторингу стану ресурсів у реальному часі: безагентного — для Docker-хостів (через Docker Engine API) і через легкий агент `node_exporter` — для решти вузлів; інтуїтивна візуалізація топології інфраструктури; рольова модель доступу, що дозволяє розподіляти повноваження між адміністраторами, операторами та спостерігачами; а також відкритість коду та відсутність ліцензійних платежів, що робить рішення доступним для команд будь-якого масштабу. Саме ці вимоги стали основою для подальшого проектування архітектури та вибору технологічного стеку, що детально розглядається у другому розділі.

Висновки до розділу 1

У першому розділі було проведено комплексний огляд предметної галузі вебзастосунку для оркестрації гібридних хмарних інфраструктур. Розглянуто еволюцію хмарних обчислень від концепції 1960-х років до сучасних рішень, що базуються на контейнеризації та принципах Infrastructure as Code. Проаналізовано основні типи хмарних інфраструктур — публічну, приватну, гібридну та мультихмарну — і визначено ключові характеристики кожного типу.

Встановлено, що гібридна хмарна інфраструктура поєднує переваги публічної та приватної хмар, забезпечуючи гнучкість масштабування, оптимізацію витрат та відповідність регуляторним вимогам, водночас вносячи додаткову складність в управління. Розглянуто концепцію оркестрації як ключового підходу до управління складними хмарними середовищами,

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	21
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

заснованого на принципах ІаС, декларативного управління конфігурацією та автоматизації.

Огляд наявних рішень виявив незаповнену нішу для доступного вебзастосунку з інтуїтивним інтерфейсом для управління гетерогенними гібридними середовищами, що обґрунтовує актуальність та практичну цінність розробленого продукту.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	22
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

РОЗДІЛ 2. ОПИС СТЕКУ ТЕХНОЛОГІЙ ТА АРГУМЕНТАЦІЯ ВИБОРУ

2.1 Архітектурний підхід та вибір технологічного стеку

Перед вибором конкретних технологій необхідно визначити загальний архітектурний підхід до побудови вебзастосунку. Для розробленої системи оркестрації гібридних хмарних інфраструктур було обрано клієнт-серверну архітектуру з чітким розділенням фронтенду та бекенду за принципом Single Page Application (SPA) з REST API.

Архітектура SPA передбачає завантаження єдиної HTML-сторінки та динамічне оновлення її вмісту через JavaScript без перезавантаження сторінки. Це забезпечує плавний інтерфейс користувача, швидкий відгук на дії та можливість роботи зі складними інтерактивними елементами - такими як графі топології інфраструктури, дашборди моніторингу в реальному часі та форми управління конфігурацією.

REST (Representational State Transfer) API забезпечує стандартизований інтерфейс взаємодії між фронтом та бекендом на основі HTTP-протоколу. Використання RESTful підходу дозволяє чітко визначити ресурси системи та операції над ними, забезпечує незалежність клієнта та сервера, спрощує тестування та документування API. Для операцій, що потребують передачі даних у реальному часі (моніторинг метрик, оновлення статусу вузлів), передбачено використання технології подій на основі бібліотеки Socket.IO, яка використовує транспортний протокол WebSocket як основний низькорівневий канал зв'язку.

Запропонована архітектура також полегшує горизонтальне масштабування системи, оскільки серверна частина не зберігає клієнтських сесій на сервері (*stateless*), а синхронізація стану між клієнтами здійснюється через дуплексні Socket.IO-з'єднання з архітектурою підписки на події (*event-driven architecture*). Для забезпечення безперебійної роботи у гетерогенних мережевих середовищах технологія підтримує автоматичне перемикавання на альтернативні протоколи

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	23
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

(наприклад, HTTP Long-Polling) у разі блокування WebSocket-трафіку фаєрволами.

Розгортання застосунку здійснюється у контейнерах Docker, що забезпечує однорідне середовище виконання незалежно від підлеглої платформи, спрощує процес розгортання та масштабування. Така архітектура повністю відповідає принципу, система демонструє застосування власних принципів у практичній реалізації - система оркестрації контейнерів сама розгортається у контейнерах, що є наочною демонстрацією переваг технології.

Зберігання даних реалізується в реляційній базі даних PostgreSQL, що забезпечує надійне та структуроване зберігання конфігурацій, метаданих інфраструктурних ресурсів, журналів подій та даних користувачів. Взаємодія бекенду з базою даних здійснюється через ORM (Object-Relational Mapping) бібліотеку Sequelize, що спрощує роботу з даними та підвищує безпеку завдяки захисту від SQL-ін'єкцій.

Визначившись із загальним архітектурним підходом, перейдемо до обґрунтування вибору конкретних фронтенд-технологій.

Для реалізації клієнтської частини застосунку було розглянуто чотири провідних JavaScript-фреймворки: React, Vue.js, Angular та Svelte. Порівняльний аналіз наведено в таблиці 2.1.

Таблиця 2.1.

Порівняльний аналіз фронтенд-фреймворків

Критерій	React	Vue.js	Angular	Svelte
Популярність (прм/тиж.)	~25 млн	~5 млн	~3 млн	~1 млн
Крива навчання	Середня	Низька	Висока	Низька
Екосистема	Дуже велика	Велика	Велика	Мала
Компонентний підхід	Так	Так	Так	Так
Підтримка TypeScript	Відмінна	Добра	Вбудована	Добра
Virtual DOM	Так	Так	Ні	Ні
Ринок праці (UA)	Дуже великий	Середній	Середній	Малий

На основі проведеного аналізу для розробки фронтенду було обрано React - відкриту JavaScript-бібліотеку для побудови користувацьких інтерфейсів, розроблену компанією Meta [15]. Обґрунтування вибору базується на таких аргументах.

React є найпопулярнішим фронтед-інструментом у 2024 році: за даними Stack Overflow Developer Survey 2024 [27], ним користуються понад 39,5% розробників у всьому світі. Це забезпечує найбільшу екосистему готових компонентів, бібліотек та документації. Компонентна архітектура React дозволяє будувати складний інтерфейс із повторно використовуваних, ізольованих компонентів, що є критично важливим для розробленого застосунку зі складним дашбордом та численними елементами управління.

Virtual DOM забезпечує ефективне оновлення інтерфейсу: React порівнює поточний та попередній стан DOM-дерева та оновлює лише змінені елементи, що є важливим для дашбордів з даними моніторингу, які оновлюються в реальному часі. Hooks API (useState, useEffect, useContext, useReducer) надає потужний та зручний механізм управління станом компонентів без потреби в громіздких class-компонентах.

Для управління глобальним станом застосунку використовується бібліотека Redux Toolkit - офіційний набір інструментів для спрощеного написання Redux-логіки. Для побудови інтерфейсу використовується бібліотека компонентів Ant Design (antd), що надає готовий набір якісних UI-компонентів із власною бізнес-орієнтованою мовою дизайну, оптимізованих для корпоративних застосунків.

Для візуалізації топології інфраструктури у вигляді інтерактивного графу вузлів та зв'язків використовується бібліотека React Flow - спеціалізований інструмент для побудови вузлових діаграм та блок-схем на базі React. Для відображення графіків метрик у реальному часі використовується бібліотека Recharts, побудована на основі D3.js та оптимізована для роботи в React-застосунках.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	25
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Ключовим нефункціональним фактором, що визначив архітектурні рішення, стала масштабованість системи: оскільки кількість підключених вузлів та обсяг метрик, що надходять у реальному часі, можуть суттєво зростати з часом, серверна частина проєктується такою, що не зберігає серверних сесій, а необхідний контекст автентифікації переноситься у JWT-токен, що дозволяє за потреби запускати кілька екземплярів бекенду за балансувальником навантаження без додаткових змін у коді. Безпека також є наскрізною вимогою: усі запити до API проходять через middleware автентифікації та авторизації на основі ролей, паролі користувачів зберігаються виключно у вигляді хешів [5][24][25], а конфіденційні параметри (ключі доступу до хмарних провайдерів, секрети JWT) зберігаються поза кодовою базою у змінних середовища. Продуктивність інтерфейсу забезпечується завдяки Virtual DOM React та використанню Socket.IO для подієво-орієнтованого оновлення даних замість періодичного опитування (polling), що суттєво знижує навантаження на мережу та сервер.

З точки зору середовищ розгортання, архітектура передбачає три типових конфігурації: середовище розробки (development), у якому фронтенд та бекенд запускаються локально в режимі гарячого перезавантаження за допомогою Nodemon та вбудованого dev-сервера React; середовище тестування (staging), що максимально наближене до промислового та використовується для перевірки коректності роботи перед релізом; і промислове середовище (production), що розгортається через Docker Compose з оптимізованою збіркою фронтенду та вимкненим режимом детального логування помилок. Розділення середовищ здійснюється через файли .env з різними наборами змінних, що відповідає принципам методології Twelve-Factor App та дозволяє уникнути жорсткого прив'язування конфігурації до коду.

Для забезпечення безпеки застосунку було реалізовано багаторівневу систему захисту. Автентифікація користувачів базується на JSON Web Token (JWT) — стандарті RFC 7519, що визначає компактний та самодостатній спосіб безпечної передачі інформації між сторонами. JWT-токен містить claims

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	26
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

(твердження) з ідентифікатором користувача, його роллю та часом закінчення дії і підписується секретним ключем за алгоритмом HMAC SHA-256. Час дії токена обмежено 24 годинами, після чого користувач повинен повторно пройти автентифікацію. Усі паролі зберігаються у базі даних виключно у вигляді хешів, обчислених за алгоритмом bcrypt з коефіцієнтом складності 12, що унеможлиблює відновлення оригінальних паролів навіть у разі компрометації бази даних.

Архітектура бази даних спроектована відповідно до принципів нормалізації та з урахуванням вимог до продуктивності. Таблиця infrastructures зберігає метадані хмарних та локальних інфраструктур верхнього рівня. З нею пов'язана таблиця nodes, кожен запис якої відповідає одному фізичному або віртуальному серверу з атрибутами: IP-адреса, тип операційної системи, кількість ядер CPU, обсяг оперативної пам'яті та дискового простору. Таблиця metrics накопичує показники продуктивності вузлів з інтервалом 30 секунд — завантаження CPU, RAM та диску — і є найбільшою за обсягом даних. Для підтримки прийнятної продуктивності запитів до metrics передбачено складений індекс на полях (node_id, recorded_at), що дозволяє ефективно вибирати метрики за конкретний часовий проміжок. Таблиця containers зберігає стан Docker-контейнерів, запущених на кожному вузлі, а таблиці alert_rules та alerts реалізують систему порогових сповіщень з підтримкою налаштування індивідуальних порогів для кожного вузла.

Комунікація між фронтендом та бекендом здійснюється двома каналами: HTTP REST API для операцій читання та запису даних (CRUD) та WebSocket-з'єднання на основі Socket.IO для передачі метрик у реальному часі. REST API організовано за ресурсно-орієнтованим принципом: кожен ресурс (інфраструктура, вузол, контейнер тощо) має власний базовий маршрут, а CRUD-операції відображаються на стандартні HTTP-методи GET, POST, PUT та DELETE. Усі маршрути, окрім автентифікаційних (/api/auth/login та /api/auth/register), захищено middleware authenticate, що перевіряє валідність JWT-токена та завантажує дані поточного користувача. Middleware authorize

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	27
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

приймає перелік дозволених ролей та повертає HTTP 403, якщо роль поточного користувача не входить до переліку. Така архітектура дозволяє гнучко налаштовувати права доступу на рівні окремих маршрутів без дублювання логіки авторизації.

2.2 Вибір та обґрунтування бекенд-технологій та СУБД

Для реалізації серверної частини застосунку розглядалися такі технологічні стеки: Node.js [16] з фреймворком Express.js [17], Python з фреймворком FastAPI, Java зі Spring Boot та мова Go. Порівняльний аналіз представлено в таблиці 2.2.

Таблиця 2.2.

Порівняльний аналіз бекенд-технологій

Критерій	Node.js	Python (FastAPI)	Java (Spring)	Go
Мова програмування	JavaScript	Python	Java	Go
Продуктивність I/O	Дуже висока	Висока	Висока	Дуже висока
Єдина мова (full-stack)	Так	Ні	Ні	Ні
Крива навчання	Низька	Низька	Висока	Середня
WebSocket підтримка	Відмінна	Добра	Добра	Добра
NPM-пакети	2+ млн	PyPI	Maven	pkg.go
Хмарна підтримка	Відмінна	Відмінна	Добра	Добра

Для розробки серверної частини обрано Node.js з фреймворком Express.js [17]. Node.js - це відкрите кросплатформне середовище виконання JavaScript, побудоване на движку V8 від Google Chrome. Головна перевага Node.js полягає в асинхронній, неблокуючій моделі виконання на основі події (event-driven, non-blocking I/O model), що робить його надзвичайно ефективним для обробки великої кількості одночасних підключень та операцій введення-виведення.

Ключовим аргументом на користь Node.js є можливість використання єдиної мови програмування - JavaScript, як на клієнті, так і на сервері. Це дозволяє повторно використовувати код між фронтендом та бекендом, спрощує

командну роботу та знижує когнітивне навантаження на розробника, якому не потрібно постійно переключатися між різними мовними парадигмами.

Express.js [17] - мінімалістичний та гнучкий веб-фреймворк для Node.js, що забезпечує набір надійних функцій для побудови веб-застосунків та API. Middleware-архітектура Express дозволяє легко розширювати функціональність застосунку через підключення додаткових модулів для обробки запитів, автентифікації, логування та інших завдань. За даними npm, Express завантажується понад 30 мільйонів разів на тиждень, що свідчить про його широке використання у промислових застосунках.

Для організації маршрутизації API використовується модульна структура Express Router, що дозволяє логічно розділяти обробники запитів за функціональними групами (авторизація, управління вузлами, моніторинг, логи тощо). Автентифікація та авторизація реалізовані на основі JSON Web Tokens (JWT) - відкритого стандарту (RFC 7519) для безпечної передачі інформації між сторонами у вигляді JSON-об'єкта з цифровим підписом.

Для обробки WebSocket-з'єднань, необхідних для передачі даних моніторингу в реальному часі, використовується бібліотека Socket.IO - найпопулярніше рішення для двосторонньої комунікації в реальному часі між браузером та сервером на платформі Node.js [30]. Socket.IO автоматично обирає оптимальний транспортний протокол (WebSocket, HTTP long-polling) залежно від можливостей клієнта та мережевого середовища.

У розробленому застосунку Socket.IO забезпечує двобічну комунікацію між сервером і браузером для трансляції метрик вузлів, сповіщень про спрацьовані алерти та оновлень статусів контейнерів. Автоматичний вибір транспорту дозволяє системі стабільно працювати навіть у корпоративних мережах із обмеженнями на прямі WebSocket-з'єднання.

Socket.IO базується на подієво-орієнтованій моделі (event-driven), де сервер та клієнт обмінюються іменованими подіями з довільним корисним навантаженням у форматі JSON. У межах розробленого застосунку це дозволило сформувати чіткий протокол подій для оновлення метрик, зміни статусу

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	29
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

контейнерів та спрацювання сповіщень, кожна з яких ініціює відповідну реакцію інтерфейсу без необхідності повного перезавантаження даних.

Окремої уваги потребує обґрунтування вибору системи управління базою даних, яка використовується для зберігання даних застосунку.

Для зберігання даних застосунку розглядалися реляційні (PostgreSQL, MySQL, SQLite) та документоорієнтована (MongoDB) СУБД. Порівняльний аналіз представлено в таблиці 2.3.

Таблиця 2.3.

Порівняльний аналіз систем управління базами даних

Критерій	PostgreSQL	MySQL	MongoDB	SQLite
Тип	Реляційна	Реляційна	Документна	Реляційна
ACID-транзакції	Так	Так	Частково	Так
JSON підтримка	Відмінна	Обмежена	Нативна	Відсутня
Масштабованість	Висока	Середня	Дуже висока	Низька
Складні запити	Відмінно	Добре	Обмежено	Добре
Ліцензія	Open Source	Open Source	SSPL	Public Domain

За результатами порівняльного аналізу для зберігання даних обрано PostgreSQL — потужну відкриту об'єктно-реляційну систему управління базами даних [18], що активно розробляється понад 35 років. PostgreSQL підтримує повний набір SQL-стандартів, ACID-транзакції, зовнішні ключі, складні запити з підзапитами та оконними функціями.

Для розробленого застосунку особливо важливою є підтримка PostgreSQL типу даних JSONB для зберігання гнучких конфігурацій хмарних ресурсів, структура яких може відрізнятися залежно від типу провайдера. JSONB дозволяє зберігати напівструктуровані дані з повною підтримкою індексування та SQL-запитів, поєднуючи переваги реляційної та документо-орієнтованої моделей.

Взаємодія бекенду з PostgreSQL здійснюється через ORM-бібліотеку Sequelize - одну з найпопулярніших Node.js ORM, що підтримує PostgreSQL, MySQL, MariaDB, SQLite та Microsoft SQL Server. Sequelize забезпечує зручне визначення моделей даних у вигляді JavaScript-класів, автоматичну генерацію

SQL-запитів, управління міграціями схеми бази даних та захист від SQL-ін'єкцій через параметризовані запити.

Обробка помилок у застосунку побудована на принципі централізованого middleware: усі помилки, що виникають у контролерах, передаються через функцію `next(error)` до спеціального обробника помилок Express, який формує уніфіковану структуру відповіді з кодом статусу HTTP, текстовим повідомленням та, у середовищі розробки, стеком викликів для діагностики. Це дозволяє уникнути дублювання логіки обробки помилок у кожному контролері та забезпечує клієнту передбачуваний формат повідомлень про помилки незалежно від того, в якому модулі вони виникли - чи то помилка валідації, чи помилка з'єднання з Docker-демоном певного вузла, чи відмова у доступі через недостатній рівень ролі користувача.

2.3 Допоміжні технології та структура взаємодії компонентів системи

Окрім основних технологій стеку, розробка вебзастосунку передбачає використання низки допоміжних інструментів та технологій, що забезпечують якість, безпеку та зручність розробки.

Docker - провідна платформа контейнеризації [12], що дозволяє пакувати застосунок разом із усіма його залежностями в стандартизований контейнер. Docker забезпечує однорідність середовища виконання від локальної розробки до промислового розгортання, усуваючи проблему «працює на моїй машині». Для розгортання повного стеку застосунку (фронтенд, бекенд, база даних) використовується Docker Compose - інструмент для визначення та запуску багатоконтейнерних Docker-застосунків.

JSON Web Token (JWT) - відкритий стандарт для безпечної передачі інформації між сторонами у вигляді JSON-об'єкта. В розробленому застосунку JWT використовується для реалізації безсесійної автентифікації: після успішного входу сервер генерує підписаний токен, який клієнт надсилає в заголовок Authorization при кожному подальшому запиті. Це дозволяє легко масштабувати серверну частину горизонтально без потреби в спільному сховищі

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	31
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

сесій. Водночас, щоб забезпечити можливість миттєвого відкликання доступу (наприклад, блокування користувача), проміжний шар автентифікації додатково звіряє кожен запит із поточним статусом облікового запису в базі даних.

Bcrypt - алгоритм хешування паролів, стійкий до атак грубої сили завдяки навмисно уповільненій функції хешування з налаштовуваним коефіцієнтом складності. Використовується для безпечного зберігання паролів користувачів у базі даних.

dotenv - бібліотека для завантаження змінних середовища з файлу .env у process.env Node.js. Використовується для зберігання чутливих конфігураційних даних (рядки підключення до бази даних, секретні ключі JWT, API-ключі хмарних провайдерів) поза кодовою базою, що є важливою вимогою безпеки відповідно до методології Twelve-Factor App.

Nodemon - утиліта, що автоматично перезапускає Node.js-застосунок при виявленні змін у файлах вихідного коду. Використовується в процесі розробки для підвищення продуктивності, усуваючи потребу в ручному перезапуску сервера після кожної зміни.

Cors (Cross-Origin Resource Sharing) - middleware для Express, що налаштовує заголовки CORS для управління доступом до API з різних доменів. У розробці фронтенд та бекенд запускаються на різних портах, тому правильне налаштування CORS є обов'язковим для коректної роботи застосунку.

Git та GitLab - система контролю версій та платформа для хостингу репозиторіїв відповідно. Весь вихідний код проєкту зберігається в репозиторії на GitLab ЖДТУ, що забезпечує версіювання, можливість відстеження історії змін та командну роботу над кодом.

Розглянувши допоміжні технології, перейдемо до опису загальної структури та схеми взаємодії компонентів розробленої системи.

Розроблений вебзастосунок має трирівневу архітектуру: рівень представлення (фронтенд на React), рівень логіки (бекенд на Node.js/Express) та рівень даних (PostgreSQL). Взаємодія між рівнями відбувається через чітко

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	32
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

визначені інтерфейси: REST API між фронтендом та бекендом, та Sequelize ORM між бекендом та базою даних.

Фронтенд-частина містить наступні ключові модулі: модуль автентифікації (сторінки входу та реєстрації, управління JWT-токеном у localStorage, захищені маршрути), головний дашборд (огляд стану всіх підключених інфраструктур), модуль управління вузлами (перелік, деталі та операції з вузлами інфраструктури), модуль моніторингу (графіки метрик CPU, RAM, диску в реальному часі), модуль управління контейнерами (перелік, запуск, зупинка контейнерів), модуль логів (перегляд та фільтрація журналів подій), модуль сповіщень та модуль візуалізації топології.

Розглянуті технології об'єднуються у багаторівневу архітектуру, де кожен компонент виконує чітко визначену роль, а взаємодія між ними відбувається через стандартизовані інтерфейси. Такий підхід забезпечує слабку зв'язність модулів, спрощує їх незалежне тестування та подальший супровід.

Типовий потік обробки запиту має такий вигляд. Користувач виконує дію у браузері, після чого клієнтський застосунок, побудований на React, формує HTTP-запит за допомогою бібліотеки Axios і надсилає його до REST API серверної частини. На боці сервера запит послідовно проходить ланцюг проміжного програмного забезпечення: спершу налаштування CORS дозволяє міждоменний доступ, далі відбувається розбір тіла запиту у формат JSON, після чого проміжний шар автентифікації перевіряє JWT-токен і звіряє статус облікового запису, а бібліотека express-validator валідує вхідні дані згідно зі схемою відповідного ендпоінту.

Лише після успішного проходження всіх перевірок керування передається відповідному контролеру, що містить бізнес-логіку операції. Контролер звертається до бази даних не напряму, а через об'єктно-реляційний відображувач Sequelize, який абстрагує SQL-запити у виклики методів моделей та захищає від типових вразливостей, зокрема SQL-ін'єкцій. Результат операції серіалізується у формат JSON і повертається клієнту, що оновлює відповідну частину інтерфейсу.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	33
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Окремий канал взаємодії забезпечує передачу даних моніторингу в реальному часі. Фоновий збирач метрик періодично опитує вузли інфраструктури та, отримавши нові значення, не лише зберігає їх у базі даних, а й надсилає подію через двостороннє з'єднання Socket.IO. Клієнтська частина, підписана на відповідні події, миттєво оновлює показники на дашборді та графіки без перезавантаження сторінки чи повторного опитування сервера.

Наскрізним для всіх рівнів є механізм журналювання та обробки помилок. Кожна значуща дія і кожна виняткова ситуація фіксуються структурованим журналом за допомогою бібліотеки Winston, що спрощує діагностику й аудит роботи системи. Помилки перехоплюються на рівні контролерів та повертаються клієнту у вигляді стандартизованих відповідей із відповідним кодом стану, що робить поведінку API передбачуваною і зручною для інтеграції.

Така багаторівнева організація з чітким розмежуванням відповідальності між клієнтом, шаром маршрутизації, проміжним програмним забезпеченням, контролерами та шаром доступу до даних забезпечує масштабованість системи, полегшує її подальший розвиток і дозволяє замінювати окремі компоненти без впливу на решту застосунку.

2.4 Проектування та розгортання системи

На основі сформульованих вимог було виконано проектування системи, результати якого подано у вигляді структурних і поведінкових моделей. Структурні моделі описують організацію програмних компонентів і даних, а поведінкові — взаємодію акторів із системою та послідовність обробки ключових операцій. Окремо в підрозділі описано процес збирання, неперервної інтеграції (CI/CD) та розгортання системи.

Бекенд-частина організована у модульну структуру за принципом MVC (Model-View-Controller). Маршрути (routes) визначають ендпоінти API та делегують обробку контролерам. Контролери (controllers) містять бізнес-логіку обробки запитів. Моделі (models) описують структуру даних через Sequelize-

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	34
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

моделі та забезпечують взаємодію з базою даних. Middleware-шар обробляє автентифікацію, валідацію вхідних даних та логування запитів.

База даних PostgreSQL містить наступні основні таблиці: users (дані користувачів), roles (ролі та дозволи), infrastructures (підключені інфраструктури), nodes (вузли інфраструктур), containers (контейнери та їх метадані), metrics (часові ряди метрик), logs (журнали подій), alert_rules (правила порогових сповіщень) та alerts (зафіксовані спрацювання сповіщень).

Структура бази даних у вигляді ER-діаграми наведена на рисунку 2.1.

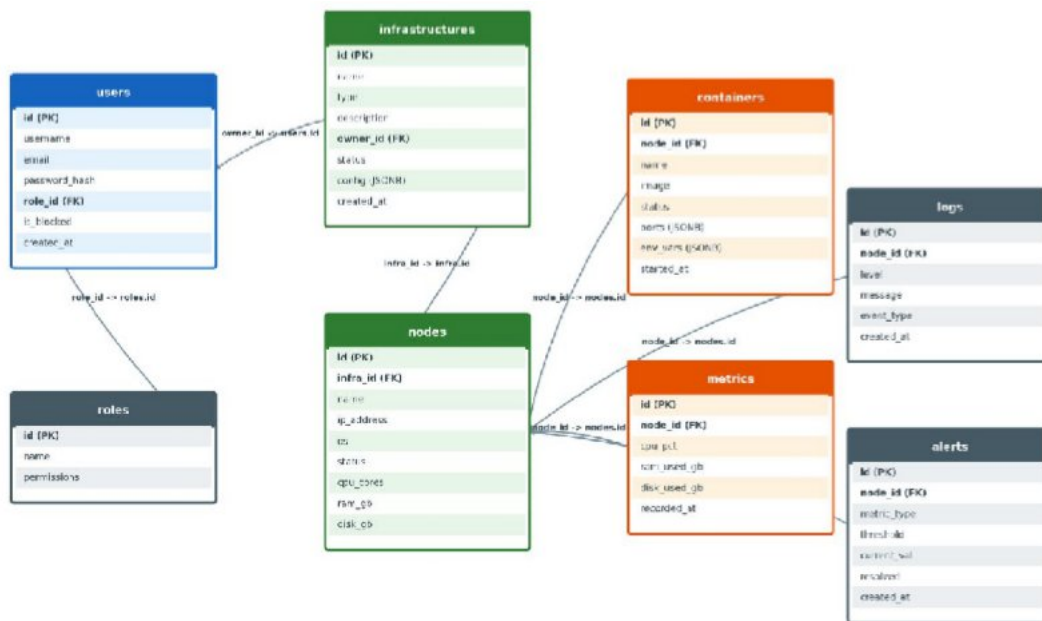


Рис. 2.1 — Структура бази даних (ER-діаграма)

База даних побудована на основі реляційної моделі та налічує дев'ять взаємопов'язаних сутностей. Центральною таблицею є nodes (вузли інфраструктури), яка зберігає інформацію про фізичні та віртуальні сервери — IP-адресу, операційну систему, кількість ядер CPU, обсяг оперативної пам'яті, дисковий простір та поточний статус. Кожен вузол належить до інфраструктури (infrastructures) та може містити запущені контейнери (containers).

Підсистема моніторингу представлена таблицею metrics, у якій з періодичністю 30 секунд накопичуються показники завантаження CPU, RAM та

диска для кожного вузла. На основі цих даних система перевіряє умови з таблиці alert_rules і за необхідності створює записи у таблиці alerts.

Окремий блок утворюють сутності управління доступом: користувачі (users) пов'язані з ролями (roles), що визначає набір дозволених дій. Усі суттєві події системи фіксуються у журналі (logs) із зазначенням рівня важливості, джерела та часу виникнення. Така структура забезпечує цілісність даних, нормалізацію 3НФ та зручність розширення системи у майбутньому.

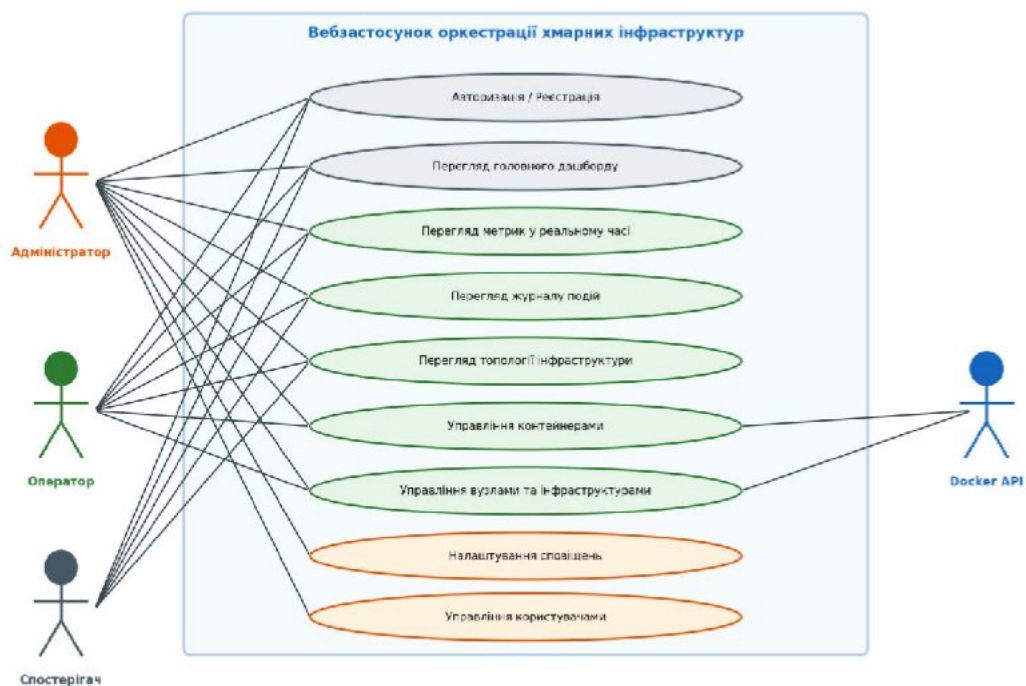


Рис. 2.2 — Діаграма варіантів використання

Діаграма варіантів використання (рисунок 2.2) відображає взаємодію між акторами — Адміністратором, Оператором, Спостерігачем та Docker API — і функціональними можливостями вебзастосунку.

Процес безперервної інтеграції та доставки (CI/CD) розробленого застосунку автоматизовано засобами GitLab CI/CD: при кожному push у репозиторій запускається конвеєр (pipeline), що складається з кількох послідовних етапів - встановлення залежностей, запуск літера ESLint для перевірки стилю коду, виконання модульних тестів бекенду та фронтенду, а також збірка Docker-образів. У разі успішного проходження всіх етапів на

основній гілці репозиторію образи автоматично публікуються у внутрішньому реєстрі контейнерів GitLab Container Registry, звідки вони можуть бути розгорнуті на цільовому сервері командою `docker-compose pull` та `docker-compose up -d`. Такий підхід суттєво знижує ризик потрапляння до продакшн-середовища коду, що не пройшов перевірку якості, та забезпечує відтворюваність процесу розгортання.

Для журналювання подій бекенду застосовується бібліотека Winston, що дозволяє записувати повідомлення різних рівнів важливості (error, warn, info, debug) як у консоль під час розробки, так і у файли журналів у промисловому середовищі, з автоматичною ротацією файлів за датою для запобігання надмірному зростанню обсягу логів на диску. Додатково middleware Morgan фіксує інформацію про кожен HTTP-запит до API - метод, шлях, код відповіді та час обробки - що є корисним для діагностики продуктивності окремих ендпоінтів. Зібрані таким способом журнали бекенду є джерелом даних для підсистеми журналу подій застосунку, описаної у третьому розділі, що дозволяє користувачам переглядати системні події безпосередньо через веб-інтерфейс без необхідності доступу до файлової системи сервера.

Висновки до розділу 2

У другому розділі було здійснено комплексний огляд та аргументований вибір технологічного стеку для розробки вебзастосунку оркестрації гібридних хмарних інфраструктур. Визначено загальний архітектурний підхід — клієнт-серверна SPA-архітектура з REST API та технологією Socket.IO (поверх протоколу WebSocket) для передачі даних у реальному часі.

На основі порівняльного аналізу альтернативних рішень обрано технологічний стек: React з бібліотеками Redux Toolkit, Ant Design, React Flow та Recharts для фронтенду; Node.js з Express.js та Socket.IO для бекенду; PostgreSQL з Sequelize ORM для зберігання даних; Docker та Docker Compose для контейнеризації та розгортання.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	37
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Кожне технологічне рішення обґрунтоване з урахуванням функціональних вимог системи, зрілості технології, якості екосистеми та доступності на ринку праці.

Описана трирівнева архітектура застосунку забезпечує чітке розділення відповідальності між компонентами, полегшує підтримку та масштабування системи, а також відповідає сучасним стандартам розробки корпоративних вебзастосунків.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	38
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

РОЗДІЛ 3. ОПИС ФУНКЦІОНАЛЬНИХ МОЖЛИВОСТЕЙ ТА ІНСТРУКЦІЯ КОРИСТУВАЧА

3.1 Встановлення, автентифікація користувачів та головний дашборд

Для коректної роботи вебзастосунку необхідно забезпечити наявність наступного програмного забезпечення на цільовій машині: Node.js версії 18.0 або вище, npm версії 9.0 або вище, Docker версії 24.0 або вище, Docker Compose версії 2.0 або вище, а також PostgreSQL версії 14.0 або вище (якщо не використовується Docker-контейнер для бази даних).

Процес розгортання застосунку через Docker Compose є рекомендованим способом, оскільки автоматично налаштовує всі необхідні компоненти та залежності. Для запуску необхідно виконати наступну послідовність дій.

- Клонувати репозиторій проєкту з GitLab
- Перейти до кореневої директорії проєкту: `cd <назва-репозиторію>`
- Створити файл `.env` на основі шаблону `.env.example` та заповнити необхідні змінні середовища (рядок підключення до бази даних, секретний ключ JWT, порти сервісів).
- Запустити всі сервіси командою: `docker-compose up -d`
- Після успішного запуску застосунок буде доступний у браузері за локальною адресою на порту 3000.

Для розгортання без Docker необхідно встановити залежності та запустити сервіси окремо. У директорії `backend` виконати `npm install` для встановлення залежностей, `npm run migrate` для застосування міграцій бази даних та `npm start` для запуску сервера. У директорії `frontend` виконати `npm install` та `npm start` для запуску застосунку у режимі розробки або `npm run build` для збірки продакшн-версії.

Перевірку успішності запуску можна здійснити, звернувшись до ендпоінту перевірки стану сервера: `GET /api/health`. У разі успішного запуску сервер повертає статус 200 та JSON-об'єкт зі статусом «ok» та поточним часом сервера.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	39
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Після встановлення та запуску застосунку розглянемо реалізовану в ньому систему реєстрації та автентифікації користувачів.

Система розмежування доступу реалізована на основі рольової моделі (Role-Based Access Control, RBAC). В системі передбачено три ролі користувачів, перелік яких наведено в таблиці 3.1.

Таблиця 3.1.

Ролі користувачів та їх повноваження

Роль	Назва	Доступні функції
admin	Адміністратор	Повний доступ: управління користувачами, всіма інфраструктурами, налаштування системи
operator	Оператор	Управління інфраструктурами, вузлами, контейнерами, перегляд логів та метрик
viewer	Спостерігач	Перегляд стану інфраструктур, метрик та логів без можливості внесення змін

Розглянемо детальніше процес реєстрації нового користувача.

Для реєстрації нового облікового запису необхідно перейти на сторінку реєстрації за адресою /register або натиснути посилання «Зареєструватись» на сторінці входу. Форма реєстрації містить наступні поля: ім'я користувача (username) — унікальний ідентифікатор від 3 до 50 символів; електронна пошта (email) — дійсна адреса електронної пошти; пароль (password) — мінімум 8 символів, обов'язково містить літери та цифри; підтвердження паролю (confirm password) — повторне введення паролю для перевірки.

Після заповнення форми та натискання кнопки «Зареєструватись» система виконує валідацію введених даних. У разі успіху створюється новий обліковий запис із роллю «viewer» за замовчуванням, і користувач автоматично перенаправляється до головного дашборду. Адміністратор системи може змінити роль користувача через панель управління.

Після реєстрації облікового запису користувач отримує можливість входу до системи.

Для входу до системи необхідно перейти на сторінку авторизації за адресою /login. Форма входу містить поля «Електронна пошта» та «Пароль». Після введення коректних облікових даних та натискання кнопки «Увійти» сервер генерує JWT-токен з терміном дії 24 години та повертає його клієнту. Токен зберігається в localStorage браузера та автоматично додається до заголовка Authorization кожного наступного запиту до API.

У разі введення некоректних облікових даних система відображає повідомлення про помилку. Після трьох невдалих спроб входу поспіль обліковий запис тимчасово блокується на 15 хвилин для захисту від атак перебору паролів. Для виходу із системи необхідно натиснути кнопку «Вийти» в правому верхньому куті навігаційної панелі — при цьому JWT-токен видаляється з localStorage.

Після успішної автентифікації користувач потрапляє на головний дашборд застосунку.

Головний дашборд є центральним елементом інтерфейсу застосунку та надає зведений огляд стану всіх підключених інфраструктур. Він відкривається автоматично після успішного входу до системи та доступний за маршрутом /dashboard.

Верхня частина дашборду містить чотири ключові показники (KPI-картки): загальна кількість підключених інфраструктур; кількість активних вузлів (зелений індикатор) та вузлів із проблемами (червоний або жовтий індикатор); кількість запущених контейнерів; кількість активних сповіщень, що потребують уваги оператора.

Центральна частина дашборду містить зведені графіки використання ресурсів по всіх інфраструктурах: графік завантаження CPU за останні 24 години, графік використання оперативної пам'яті та діаграму використання дискового простору. Графіки оновлюються в реальному часі через Socket.IO-з'єднання з частотою оновлення 30 секунд.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	41
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Нижня частина дашборду містить таблицю останніх подій (логів) та список активних сповіщень із можливістю швидкого переходу до детальної інформації про відповідний вузол або інфраструктуру.

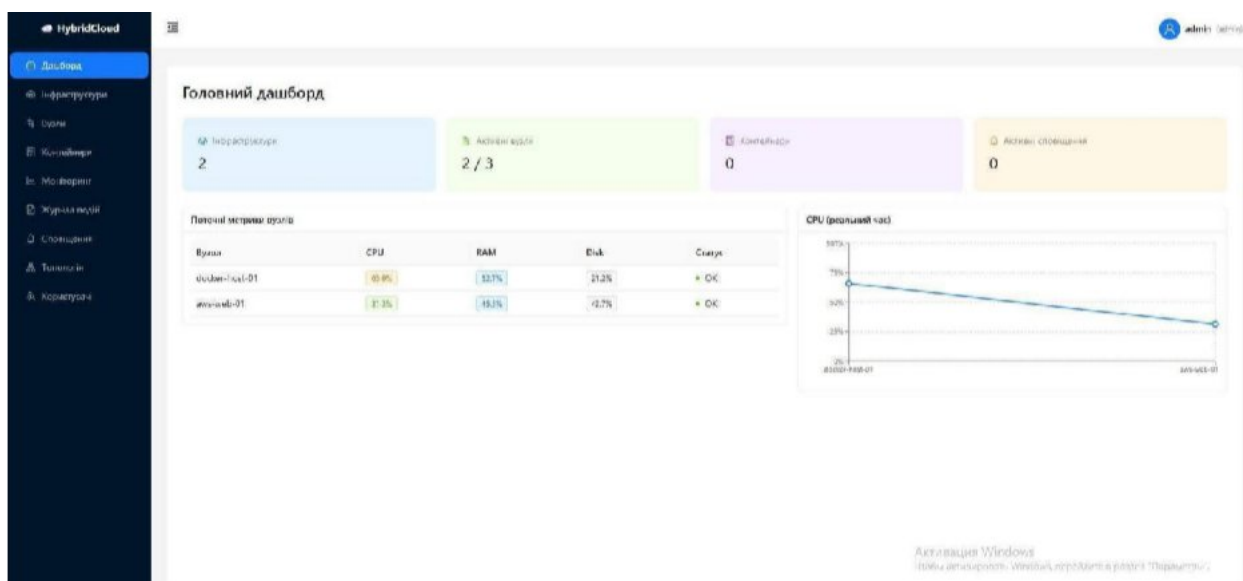


Рис. 3.1 — Головний дашборд застосунку

Окрім перегляду метрик окремого вузла, інтерфейс моніторингу підтримує групування вузлів за інфраструктурами та фільтрацію за статусом, що дозволяє оператору швидко локалізувати проблемні ділянки у великій розподіленій системі, що складається з десятків вузлів у кількох хмарних середовищах одночасно. Для вузлів, що належать до однієї інфраструктури, передбачено агреговане відображення середнього навантаження, що допомагає оцінити загальний стан кластера без необхідності перегляду кожного вузла окремо. Історія метрик зберігається в таблиці metrics з періодичністю запису 30 секунд, що забезпечує достатню деталізацію для аналізу короткострокових піків навантаження, водночас обсяг накопичених даних контролюється політикою архівації застарілих записів старших за 30 днів для запобігання неконтрольованому зростанню розміру бази даних.

Порогові значення сповіщень, наведені раніше, є початковими (за замовчуванням) і обрані на основі загальноприйнятих рекомендацій галузі щодо завантаження серверних ресурсів: перевищення 80% завантаження CPU протягом тривалого часу зазвичай свідчить про наближення до межі

продуктивності вузла, тоді як використання пам'яті понад 85% підвищує ризик активації механізму swar, що різко знижує швидкодію. Адміністратор може скоригувати ці значення індивідуально для кожного вузла залежно від його ролі: наприклад, для вузлів баз даних доцільно встановлювати нижчий порог використання пам'яті, оскільки СУБД активно використовують кешування, а різке зростання цього показника може сигналізувати про витік пам'яті в одному з запущених процесів.

3.2 Управління інфраструктурами, вузлами та моніторинг ресурсів

Для підключення нової інфраструктури до системи необхідно перейти до розділу «Інфраструктури» в лівому навігаційному меню та натиснути кнопку «Додати інфраструктуру». Відкриється форма з наступними полями: назва інфраструктури (довільна назва для ідентифікації), тип інфраструктури (локальний сервер, Docker-хост або хмарний акаунт — AWS, Azure, GCP — як ознака категоризації обліку, без прямої інтеграції з SDK провайдера на поточному етапі), опис (необов'язкове текстове поле).

Для хмарних провайдерів додатково вказуються облікові дані доступу (API-ключі, регіон розгортання).

Після збереження система виконує перевірку підключення до вказаної інфраструктури та відображає статус перевірки. У разі успішного підключення інфраструктура з'є у загальному переліку зі статусом «активна». Тестове підключення можна повторити у будь-який момент через кнопку «Перевірити підключення» у картці інфраструктури.

Після підключення інфраструктури розглянемо управління окремими вузлами цієї інфраструктури.

Після вибору конкретної інфраструктури зі списку відкривається сторінка її деталей із переліком усіх вузлів (серверів, VM, контейнерних хостів). Для кожного вузла відображається: назва вузла, IP-адреса, поточний статус (online/offline/warning), операційна система, поточне завантаження CPU та RAM у вигляді прогрес-барів, час останнього оновлення метрик.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	43
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

При натисканні на картку вузла відкривається детальна сторінка з повною інформацією: технічні характеристики (кількість ядер CPU, обсяг RAM, розмір диску), графіки метрик за різні часові періоди (1 година, 24 години, 7 днів), перелік запущених контейнерів, журнал подій вузла. Для вузлів із типом «Docker» додатково відображається перелік Docker-образів та можливість запуску нових контейнерів.

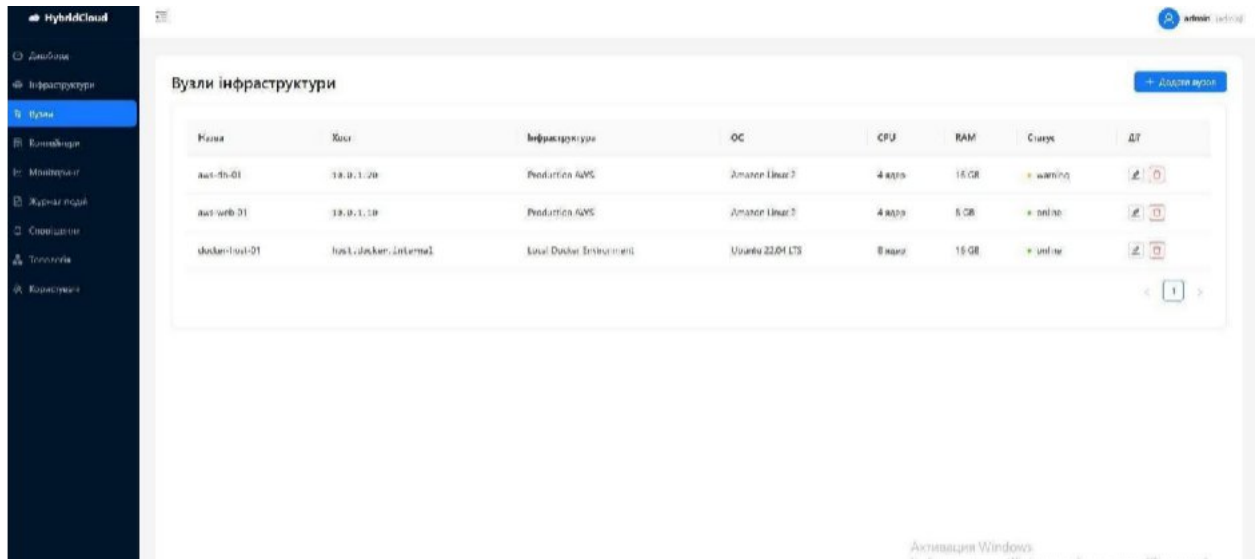


Рис. 3.2 — Перелік вузлів інфраструктури

Поряд з управлінням вузлами важливим є моніторинг ресурсів у реальному часі.

Розділ моніторингу доступний через пункт «Моніторинг» у навігаційному меню або через кнопку «Детальний моніторинг» на картці будь-якого вузла. Система збирає та відображає три ключові типи метрик: завантаження процесора (CPU utilization, %), використання оперативної пам'яті (RAM usage, GB та %), використання дискового простору (Disk usage, GB та %).

Метрики відображаються у вигляді інтерактивних графіків на основі бібліотеки Recharts. Користувач може обирати часовий діапазон для відображення: остання година, останні 24 години, 7 днів або 30 днів. При наведенні курсора на будь-яку точку графіка відображається спливаюча підказка з точним значенням метрики та часом вимірювання.

Для порівняльного аналізу кількох вузлів одночасно передбачена функція накладання графіків: користувач може обрати до п'яти вузлів для відображення їх метрик на одному графіку різними кольорами. Це дозволяє швидко виявити вузли з нестандартним поведінковим патерном або нерівномірний розподіл навантаження в кластері.

Дані моніторингу оновлюються в реальному часі через Socket.IO-з'єднання. Індикатор підключення у верхньому правому куті сторінки моніторингу показує статус Socket.IO-з'єднання: зелений — підключено та отримуються дані в реальному часі, жовтий — відбувається повторне підключення, червоний — з'єднання відсутнє, відображаються кешовані дані.



Рис. 3.3 — Моніторинг ресурсів у реальному часі

Послідовність дій під час розгортання контейнера — від запиту оператора до відображення результату в інтерфейсі — наведена на рисунку 3.4 у вигляді діаграми діяльності. Представлений алгоритм відображає повний життєвий цикл обробки запиту, починаючи з первинної валідації даних форми на стороні клієнта та надсилання POST-запиту на ендпоінт `/api/containers/deploy`. На рівні бекенду система виконує обов'язкову автентифікацію користувача через JWT-токен та перевіряє наявність прав оператора для виконання операцій

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	45
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

розгортання. У разі успішної авторизації ініціюється виклик Docker Remote API на цільовому вузлі, де перевіряється наявність необхідного образу в локальному сховищі або здійснюється його автоматичне завантаження з Docker Registry.

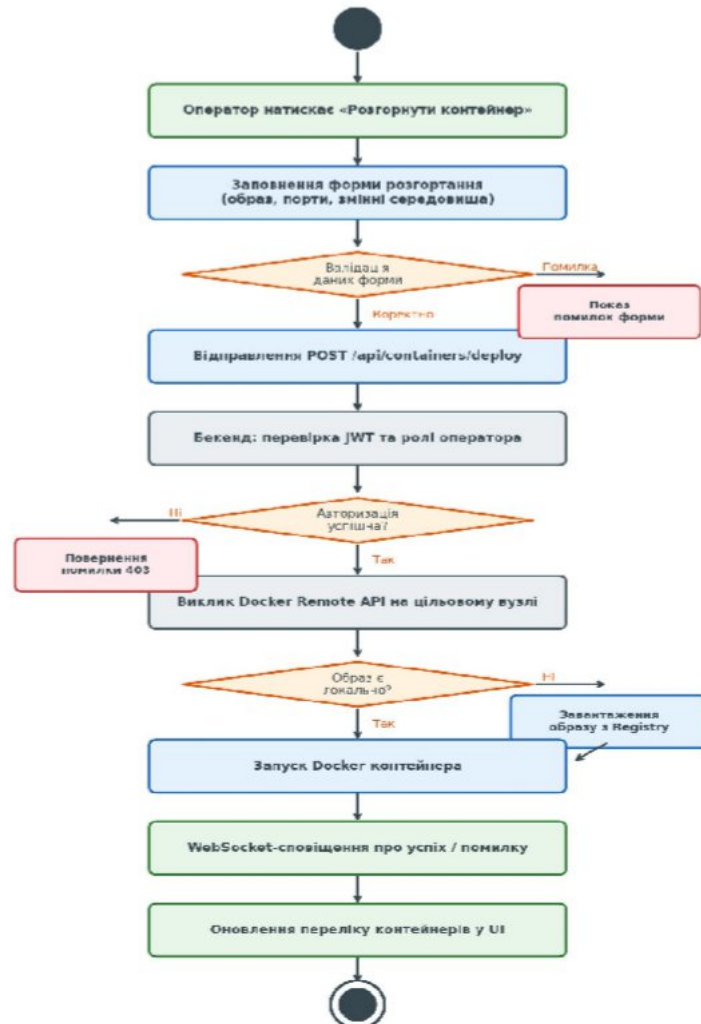


Рис. 3.4 — Діаграма діяльності розгортання контейнера

Окрім моніторингу ресурсів, застосунок надає розвинені засоби управління контейнерами.

Розділ управління контейнерами надає можливість переглядати, запускати, зупиняти та видаляти Docker-контейнери на підключених вузлах інфраструктури. Доступ до розділу здійснюється через пункт «Контейнери» в навігаційному меню.

Головна сторінка розділу відображає таблицю всіх контейнерів у системі з можливістю фільтрації за вузлом, статусом (running, stopped, error) та ім'ям образу. Для кожного контейнера в таблиці відображається: ID контейнера (скорочений), ім'я контейнера, Docker-образ та тег, статус, час запуску, прив'язані порти, вузол розгортання.

Розглянемо детальніше процес розгортання нового контейнера.

Для розгортання нового контейнера необхідно натиснути кнопку «Розгорнути контейнер» та заповнити форму розгортання. Форма містить наступні поля: Docker-образ (наприклад, nginx:latest або node:18-alpine), ім'я контейнера (необов'язково, якщо не вказано — генерується автоматично), вузол для розгортання (вибір зі списку активних вузлів), прив'язка портів у форматі хост:контейнер (наприклад, 8080:80), змінні середовища у форматі ключ=значення, команда запуску (необов'язково, для перевизначення CMD образу).

Після підтвердження форма відправляє запит на бекенд, який передає команду розгортання до Docker-демону відповідного вузла через Docker Remote API. Прогрес завантаження образу та запуску контейнера відображається в реальному часі через Socket.IO. Після успішного запуску контейнер з'являється в загальному переліку.

Після розгортання контейнера користувачу доступні операції управління наявними контейнерами.

Для кожного контейнера доступні наступні операції: зупинка (stop) — зупиняє запущений контейнер із збереженням його стану; запуск (start) — запускає зупинений контейнер; перезапуск (restart) — перезапускає контейнер; видалення (remove) — незворотно видаляє контейнер (доступно лише для зупинених контейнерів); перегляд логів — відкриває сторінку перегляду stdout/stderr контейнера.

Операції зупинки та видалення потребують підтвердження через діалогове вікно з попередженням про наслідки дії. Всі операції над контейнерами

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	47
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

фіксуються в журналі подій системи із зазначенням користувача, що виконав дію, та часу виконання.



Рис. 3.5 — Сторінка управління контейнерами

Розділ моніторингу надає користувачеві гнучкі засоби для аналізу стану інфраструктури. Користувач може обрати один або кілька вузлів для одночасного порівняння їхніх показників на спільних графіках, що зручно для виявлення аномалій та порівняння навантаження між серверами. Для кожного типу метрики — завантаження процесора, використання оперативної пам'яті та дискового простору — будується окремий графік із можливістю вибору часового діапазону відображення (година, доба, тиждень).

Збір метрик реалізовано на основі поєднання агентного та безагентного підходів залежно від типу вузла. Для серверів під керуванням операційних систем на базі Linux показники отримуються від агента `node_exporter`, для вузлів під керуванням Windows — від `windows_exporter`, а для контейнерних хостів застосовується безагентний підхід через Docker Engine API. Збирач метрик опитує лише вузли зі статусом «онлайн» із фіксованим інтервалом, обчислює відсоткові значення завантаження на основі лічильників експортера та зберігає їх у базі даних.

Збережені значення метрик формують історію спостережень, на основі якої будуються графіки за обраний період. Для запобігання надмірному зростанню обсягу бази даних застосовано автоматичне очищення застарілих записів метрик після завершення періоду їх зберігання. Такий механізм дозволяє

поєднати можливість ретроспективного аналізу навантаження з ефективним використанням дискового простору.

Управління окремими вузлами не обмежується переглядом їхнього стану. Адміністратор може додавати нові вузли до інфраструктури, редагувати їхні параметри, тимчасово переводити вузол у стан технічного обслуговування або вилучати його. Зміна статусу вузла безпосередньо впливає на роботу збирача метрик, який автоматично припиняє опитування вимкнених вузлів, що зменшує зайве навантаження на систему та підтримує журнал подій у чистоті.

Поєднання періодичного збору метрик, їх збереження для історичного аналізу та миттєвого оновлення показників у реальному часі через двостороннє з'єднання забезпечує адміністратору актуальну й водночас ретроспективну картину стану кожного вузла гетерогенної інфраструктури.

3.3 Логи, сповіщення, топологія та адміністрування

Розділ «Логи» надає централізований доступ до журналів подій усієї інфраструктури. Системні події поділяються на чотири рівні важливості: DEBUG (детальна діагностична інформація), INFO (звичайні інформаційні повідомлення про події системи), WARNING (попередження про потенційні проблеми, що не впливають на роботу), ERROR (помилки, що потребують уваги оператора).

Для зручної роботи з великою кількістю подій передбачена система фільтрів. Фільтрація може здійснюватися за наступними параметрами: часовий діапазон (від/до з вибором дати та часу), рівень важливості (мультивибір із чекбоксами), інфраструктура та вузол-джерело, тип події (системна, мережева, подія контейнера, подія безпеки), текстовий пошук у повідомленні події.

Результати фільтрації відображаються у вигляді таблиці з пагінацією (50 записів на сторінку за замовчуванням, до 200 — налаштовується). Повідомлення кольорово кодуються відповідно до рівня важливості: сірий для DEBUG, синій для INFO, жовтий для WARNING, червоний для ERROR. При натисканні на

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	49
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

рядок події відкривається детальний перегляд з повним текстом повідомлення, метаданими та пов'язаними подіями.

Передбачена функція експорту журналу подій у форматах CSV та JSON для подальшого аналізу в зовнішніх інструментах. Експорт застосовується до поточних результатів фільтрації, а не до всього журналу.



Рис. 3.6 — Журнал подій

Окрім журналу подій, застосунок реалізує систему сповіщень про аномалії та перевищення порогових значень метрик.

Система сповіщень автоматично відстежує метрики всіх вузлів та генерує сповіщення при виявленні аномалій або перевищенні порогових значень.

Параметри сповіщень за замовчуванням наведено в таблиці 3.2.

Таблиця 3.2.

Параметри сповіщень за замовчуванням

Тип метрики	Поріг за замовчуванням	Опис реакції системи
CPU (завантаження процесора)	80%	Сповіщення в панелі + email (якщо налаштовано)
RAM (використання пам'яті)	85%	Сповіщення в панелі + підсвічування вузла
Disk (використання диску)	90%	Критичне сповіщення + зміна кольору індикатора
Статус вузла	offline / error	Негайне сповіщення + позначення в топології

Для налаштування індивідуальних порогів сповіщень необхідно перейти до розділу «Налаштування» та обрати підрозділ «Сповіщення».

Для кожного вузла або групи вузлів можна встановити власні порогові значення та обрати канали доставки сповіщень: інтерфейс застосунку

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	50
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

(відображення в панелі сповіщень з лічильником), електронна пошта (якщо налаштовано SMTP-сервер).

Активні сповіщення відображаються у панелі сповіщень, доступній через значок дзвоника в шапці застосунку. Лічильник на значку показує кількість непрочитаних сповіщень. Кожне сповіщення містить: назву та рівень критичності, вузол та метрику, що спричинили сповіщення, поточне значення метрики та поріг, час виникнення, кнопку «Переглянути деталі» для переходу до відповідного вузла.

Сповіщення автоматично закривається (resolve), коли значення метрики повертається нижче порогового рівня протягом 5 хвилин. Закриті сповіщення переміщуються до розділу «Історія сповіщень» та зберігаються протягом 30 днів.

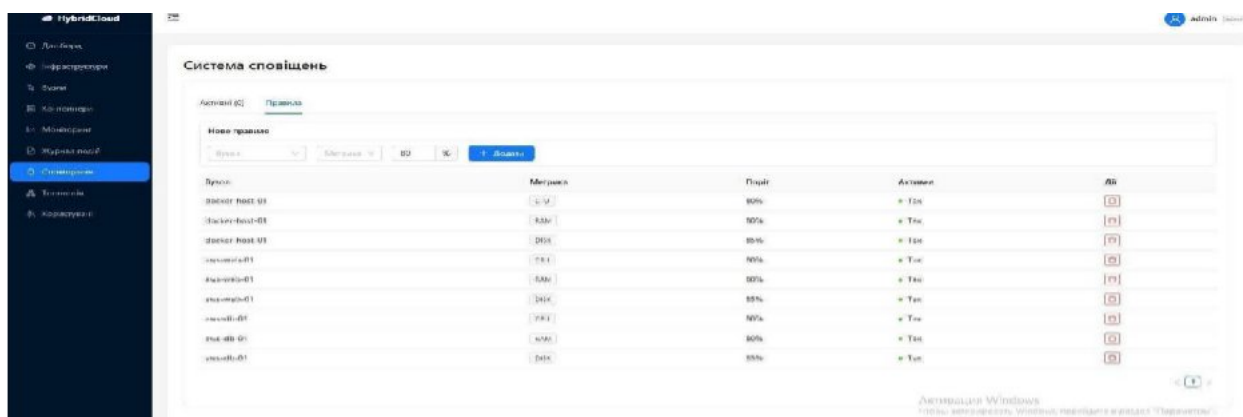


Рис. 3.7 — Налаштування правил сповіщень

Ще одним важливим модулем застосунку є інтерактивна візуалізація топології інфраструктури. Для кожної інфраструктури будується окремий граф, де вузли відповідають серверам та іншим компонентам, а ребра — мережевим з'єднанням між ними. Графова модель підтримує динамічне оновлення стану в реальному часі, що дозволяє користувачу не лише отримати сповіщення про помилку, але й візуально локалізувати місце збою в мережі через підсвічування проблемних вузлів. Завдяки підтримці інтерактивності, користувач може взаємодіяти з графом безпосередньо: змінювати масштаб (zoom), переміщувати вузли для кращого огляду складних ділянок мережі.

Розділ «Топологія» надає інтерактивну графову візуалізацію структури підключених інфраструктур та зв'язків між їх компонентами. Для кожної інфраструктури будується окремий граф, де вузли графу відповідають серверам та компонентам інфраструктури, а ребра — мережевим з'єднанням між ними. Графова модель підтримує динамічне оновлення стану в реальному часі: у разі зміни статусу будь-якого вузла (наприклад, переходу в стан offline), відповідний елемент на схемі миттєво змінює свій колір на червоний.



Рис. 3.8 — Топологія інфраструктури

Реалізацію графу забезпечує бібліотека React Flow, що підтримує перетягування вузлів, масштабування, панорамування та інтерактивний вибір елементів. Вузли графу кольорово кодуються за статусом: зелений — вузол активний та працює нормально; жовтий — є попередження (наближення до порогових значень); червоний — вузол недоступний або має критичні помилки; сірий — вузол відключений або в технічному обслуговуванні.

При натисканні на вузол графу відкривається спливаюча панель з ключовими метриками вузла (CPU, RAM, Disk у вигляді прогрес-барів), IP-адресою, статусом та кількістю активних контейнерів. Подвійне натискання відкриває повну сторінку деталей вузла. Крім того, бібліотека React Flow дозволяє реалізувати механізм «згортання» та «розгортання» (collapsing) підграфів — це допомагає користувачеві структурувати відображення.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	52
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Панель інструментів топології містить: кнопку автоматичного розміщення вузлів (layout), фільтр відображення (показати лише вузли з проблемами), кнопку скидання масштабу та позиції, перемикач між горизонтальним та вертикальним компонуванням, а також кнопку збереження поточного розміщення як шаблону.

Завершальним розглянемо розділ адміністрування системи, доступний користувачам з роллю «admin».

Розділ адміністрування доступний лише для користувачів із роллю «admin». Він містить підрозділи управління користувачами, налаштувань системи та аудиту дій.

На сторінці управління користувачами відображається таблиця всіх зареєстрованих облікових записів із можливістю пошуку та фільтрації.

Для кожного користувача адміністратор може змінити роль (підвищити до оператора або адміністратора, або понизити до спостерігача), тимчасово заблокувати обліковий запис або видалити його. Всі зміни в управлінні користувачами фіксуються в журналі аудиту.

Розділ аудиту дій надає повний журнал всіх значущих дій користувачів у системі: входи та виходи, операції з контейнерами, зміни конфігурацій, операції управління користувачами. Для кожного запису аудиту зберігаються: ім'я користувача, IP-адреса, тип дії, час виконання, результат (успіх або помилка).

Розділ «Налаштування API» дозволяє адміністратору переглянути документацію REST API системи та основні ендпоінти, перелік яких наведено в таблиці 3.3. Документація API дозволяє інтегрувати застосунок із зовнішніми системами моніторингу або автоматизації.

Таблиця 3.3.

Основні ендпоінти REST API

Метод	Ендпоінт	Авторизація	Опис
POST	/api/auth/register	Немає	Реєстрація нового користувача

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	53
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Метод	Ендпоінт	Авторизація	Опис
POST	/api/auth/login	Немає	Вхід та отримання JWT
GET	/api/infrastructures	JWT (operator+)	Список інфраструктур
POST	/api/infrastructures	JWT (operator+)	Додати інфраструктуру
GET	/api/nodes/:id/metrics	JWT (viewer+)	Метрики вузла
GET	/api/nodes/:id/logs	JWT (viewer+)	Логи вузла
POST	/api/containers/deploy	JWT (operator+)	Розгортання контейнера
GET	/api/users	JWT (admin)	Список користувачів

Документація REST API розробленого застосунку реалізована за специфікацією OpenAPI (Swagger) та доступна через вбудований інтерактивний інтерфейс Swagger UI за маршрутом /api-docs, що дозволяє розробникам зовнішніх систем переглядати перелік доступних ендпоінтів, формати запитів і відповідей, а також виконувати тестові запити безпосередньо з браузера, не звертаючись до додаткової документації. З огляду на критичність розділу адміністрування для безпеки всієї системи, окрім стандартної JWT-автентифікації для доступу до нього передбачено додаткову перевірку ролі «admin» на рівні middleware маршрутів, а також ведення окремого, детальнішого журналу для всіх дій, виконаних в межах цього розділу, включно зі зміною ролей користувачів та модифікацією системних налаштувань сповіщень.

3.4 Тестування та забезпечення якості програмного забезпечення

Перевірка коректності роботи вебзастосунку виконувалася на кількох рівнях — модульному, інтеграційному та функційному (системному).

Багаторівневий підхід дозволяє виявляти дефекти якомога раніше: від помилок в окремих функціях до невідповідностей у наскрізних сценаріях роботи системи.

Перевірку якості коду виконано на кількох рівнях. Модульне тестування бекенд-контролерів та допоміжних функцій реалізовано за допомогою фреймворку Jest у поєднанні з бібліотекою Supertest, що дозволяє надсилати HTTP-запити до Express-застосунку безпосередньо в тестовому середовищі без

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	54
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

розгортання повноцінного сервера. На фронтенді для тестування React-компонентів застосовано React Testing Library, яка перевіряє поведінку компонентів з точки зору користувача — наявність елементів на екрані та реакцію на події, — а не деталі їхньої внутрішньої реалізації. Для валідації вхідних даних API використано бібліотеку express-validator, що описує схеми перевірки для кожного ендпоінту та повертає клієнту структуровані повідомлення про помилки ще до того, як запит потрапляє до бізнес-логіки контролера.

Функційне тестування виконувалося вручну за заздалегідь складеними сценаріями, що охоплюють основні варіанти використання системи (рисунки 2.2). Перелік сценаріїв та результати їх виконання наведено в таблиці 3.4.

Таблиця 3.4.

Сценарії та результати функційного тестування

№	Сценарій	Дії / вхідні дані	Очікуваний результат	Результат
1	Реєстрація користувача	Коректне заповнення форми реєстрації	Створення облікового запису з роллю «viewer», перехід на дашборд	Пройдено
2	Валідація реєстрації	Пароль коротший за 8 символів	Відмова, повідомлення про помилку валідації	Пройдено
3	Автентифікація	Правильні логін і пароль	Видача JWT-токена, доступ до системи	Пройдено
4	Захист від підбору пароля	П'ять невдалих спроб входу поспіль	Тимчасове блокування облікового запису	Пройдено
5	Додавання інфраструктури	Заповнення форми та перевірка підключення	Інфраструктуру збережено, статус «активна»	Пройдено
6	Розгортання контейнера	POST /api/containers/deploy з образом і портами	Контейнер запущено, статус «running», подія container:status	Пройдено
7	Дія над контейнером	Зупинка / перезапуск / видалення	Зміна статусу та запис у журнал подій	Пройдено
8	Порогове сповіщення	Перевищення порогу CPU у правилі	Сповіщення рівня «warning/critical», подія alert:new	Пройдено
9	Розмежування прав	Спроба ролі «viewer» розгорнути контейнер	Відмова з кодом 403 (недостатньо прав)	Пройдено
10	Перевірка стану сервера	GET /api/health	Статус 200, JSON зі станом «ok»	Пройдено

За результатами функційного тестування всі перелічені сценарії відпрацьовано успішно. Виявлені під час перевірки дрібні недоліки —

насамперед у валідації форм та форматуванні повідомлень про помилки — було усунуто до фінального складання застосунку.

Окрім ручного функційного тестування, контроль якості автоматизовано в конвеєрі GitLab CI/CD (підрозділ 2.4). На кожному надсиланні змін до репозиторію послідовно запускаються лінтер ESLint для перевірки стилю коду та модульні тести бекенду й фронтенду; лише за умови їх успішного проходження виконується збірка Docker-образів. Такий підхід гарантує, що до основної гілки й до промислового середовища не потрапляє код, який не пройшов перевірки якості.

Окрему увагу приділено модульному тестуванню найвідповідальнішого компонента серверної частини — збирача метрик, оскільки саме коректність обчислення показників завантаження є критичною для достовірності моніторингу. Розроблено набір модульних тестів, що перевіряють функцію розбору експозиції метрик у форматі Prometheus як для агента node_exporter (Linux), так і для windows_exporter (Windows).

Тести подають на вхід еталонні фрагменти експозиції з відомими значеннями лічильників і перевіряють, що обчислені відсоткові показники процесора, оперативної пам'яті та дискового простору збігаються з очікуваними. Зокрема, для тестового набору Linux функція повертає завантаження процесора 20%, пам'яті — 25% та диску — 40%, а для набору Windows — відповідно 10%, 75% та 75%. Окремий тест підтверджує детермінованість демонстраційного режиму: для одного вузла в межах однакового часового проміжку функція повертає відтворювані значення, що виключає використання випадкових чисел.

На інтеграційному рівні за допомогою бібліотеки Supertest перевірено коректність роботи маршрутів REST API: доступність службового ендпоінту стану сервера, повернення коду 401 для захищених маршрутів без дійсного токена автентифікації та коду 404 для неіснуючих ресурсів. Окремо протестовано криптографічну складову автентифікації — хешування й перевірку паролів за алгоритмом bcrypt та формування і валідацію JWT-токенів, включно з відхиленням підроблених токенів.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	56
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

Перевірку якості інтегровано в процес безперервної інтеграції: при кожному надходженні змін до репозиторію конвеєр GitLab CI/CD послідовно виконує статичний аналіз коду, запуск автоматизованих тестів та збирання Docker-образів. Складання образу відбувається лише за умови успішного проходження всіх тестів, що запобігає потраплянню дефектного коду на етап розгортання.

Поєднання модульного, інтеграційного та функційного тестування з автоматизованим контролем якості на етапі інтеграції забезпечує надійність розробленого застосунку та дозволяє виявляти регресії на ранніх стадіях розробки.

Висновки до розділу 3

У третьому розділі детально описано функціональні можливості розробленого вебзастосунку для оркестрації гібридних хмарних інфраструктур та надано вичерпну інструкцію для кінцевого користувача. Розглянуто повний цикл роботи з системою: від встановлення та початкового налаштування до щоденного використання всіх функцій.

Окремий підрозділ третього розділу (3.4) присвячено забезпеченню якості програмного забезпечення: визначено рівні тестування (модульне, інтеграційне, функційне), наведено сценарії та результати функційного тестування основних варіантів використання, а також схему автоматизованого контролю якості засобами GitLab CI/CD.

Описано механізм рольової авторизації з трьома рівнями доступу (адміністратор, оператор, спостерігач), що забезпечує гнучке управління правами в командному середовищі. Детально висвітлено функції моніторингу ресурсів у реальному часі, управління контейнерами, перегляду та фільтрації журналів подій, системи сповіщень та інтерактивної візуалізації топології інфраструктури.

Розроблений застосунок охоплює повний спектр завдань, пов'язаних з управлінням гібридними хмарними інфраструктурами, надаючи єдиний зручний

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	57
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

інтерфейс для операторів з різними рівнями технічної підготовки.
Документований REST API забезпечує можливість інтеграції із зовнішніми системами автоматизації та моніторингу.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	58
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

ВИСНОВКИ

Випускна кваліфікаційна робота бакалавра присвячена розробці вебзастосунку для оркестрації гібридних хмарних інфраструктур — актуальної завдання в умовах стрімкого розширення хмарних середовищ та зростання складності IT-інфраструктур сучасних організацій.

Проведений аналіз предметної галузі підтвердив, що гібридні хмарні інфраструктури є домінуючою стратегією для підприємств, які прагнуть поєднати гнучкість публічної хмари з безпекою та контрольованістю приватної. За прогнозами провідних аналітичних компаній, частка організацій, що використовують гібридний підхід, становить 73%, а мультихмарні стратегії застосовують 89% організацій (Flexera, 2024). Водночас управління такими гетерогенними середовищами залишається складним завданням через відсутність єдиних інструментів, різномірність API хмарних провайдерів та високий поріг входу наявних корпоративних рішень.

Огляд наявних рішень — Terraform, Kubernetes, Portainer, нативних консолей хмарних провайдерів — виявив незаповнену нішу для доступного вебзастосунку з інтуїтивним інтерфейсом, що охоплює повний спектр завдань управління гібридною інфраструктурою. Це обґрунтувало актуальність та практичну цінність розробленого продукту.

На основі порівняльного аналізу технологічних альтернатив обґрунтовано вибір стеку React + Node.js/Express + PostgreSQL + Docker. Обраний стек забезпечує оптимальне поєднання продуктивності, масштабованості та зручності розробки, а використання єдиної мови JavaScript на клієнті та сервері знижує складність розробки та підтримки.

Розроблений вебзастосунок реалізує повний набір функцій для управління гібридними хмарними інфраструктурами: рольову систему авторизації з трьома рівнями доступу; підключення та моніторинг вузлів різних хмарних середовищ; відображення метрик CPU, RAM та дискового простору в реальному часі через Socket.IO; управління контейнерами (розгортання, запуск, зупинка);

централізований журнал подій із фільтрацією; систему сповіщень при перевищенні порогових значень; інтерактивну візуалізацію топології інфраструктури на основі React Flow. Застосунок розгортається у Docker-контейнерах, що забезпечує відтворюваність та простоту встановлення.

Практична значущість роботи полягає у створенні функціонального інструменту, який дозволяє командам DevOps та IT-адміністраторам ефективно керувати гетерогенними хмарними середовищами через єдиний веб-інтерфейс без необхідності перемикатися між нативними консолями різних провайдерів. Документований REST API забезпечує можливість інтеграції з іншими системами автоматизації.

Подальший розвиток застосунку може передбачати інтеграцію з провідними хмарними провайдерами через їх офіційні SDK, підтримку Kubernetes як оркестратора контейнерів, реалізацію функціонала Infrastructure as Code через вбудований редактор конфігурацій, а також розширення можливостей аналітики з використанням алгоритмів виявлення аномалій на основі машинного навчання.

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	60
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Burns B., Grant B., Oppenheimer D., Brewer E. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. Queue. 2016. Vol.14, No.1. P.70–93.
2. Arundel J., Domingus J. Cloud Native DevOps with Kubernetes. 2nd ed. O'Reilly Media, 2022. 486 p.
3. Brikman Y. Terraform: Up & Running: Writing Infrastructure as Code. 3rd ed. O'Reilly Media, 2022. 466 p.
4. Poulton N. The Kubernetes Book. 2023 Edition. Self-published, 2023. 280 p.
5. Яровий Р., Улічев О., Склярєнко О., Пашорін В. Моделювання мультиагентних систем захисту інформаційних ресурсів. Herald of Khmelnytskyi National University. Technical Sciences. 2024. Vol.337, No.3(2). С.278–284.
6. Garrison J., Nova K. Cloud Native Infrastructure: Patterns for Scalable Infrastructure. O'Reilly Media, 2021. 166 p.
7. Stiensmeier X., Kanitz A., Krüger J. et al. Hybrid Cloud Architectures for Research Computing: Applications and Use Cases. arXiv:2601.04349. 2026. URL: <https://arxiv.org/abs/2601.04349> (дата звернення: 14.02.2026).
8. Лучкевич М. Інтеграція методології DevOps у професійну підготовку фахівців з інформаційних технологій. Наукові записки УДУ. 2025. URL: <https://enpuir.udu.edu.ua/bitstreams/74df87f0-7d21-4943-8b1d-dc984344b41a/download> (дата звернення: 19.02.2026).
9. Waseem M., Ahmad A., Liang P., Akbar M. A., Khan A. A., Ahmad I., Setälä M., Mikkonen T. Containerization in Multi-Cloud Environment: Roles, Strategies, Challenges, and Solutions for Effective Implementation. Journal of Systems and Software. 2025. Vol.230. Article 112558.
10. Singh S., Chana I., Buyya R. STAR: SLA-aware Autonomic Management of Cloud Resources. IEEE Trans. Cloud Computing. 2020. Vol.8, No.4. DOI: 10.1109/TCC.2017.2648788.

		Антонюк А.Р.			ІІЗ.КР.Б – 121 – 26 – ПЗ	61
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

11. Toka L., Dobreff G., Fodor B., Sonkoly B. Machine Learning-based Scaling Management for Kubernetes Edge Clusters. IEEE Trans. Network Service Mgmt. 2021. Vol.18, No.1. P.958–972.
12. Casalicchio E. Autonomic Orchestration of Containers: Problem Definition and Research Challenges. Proc. 10th EAI Int. Conf. on Performance Evaluation Methodologies and Tools (VALUETOOLS). ACM, 2017. P.287–290. DOI: 10.4108/eai.25-10-2016.2266649.
13. Вінтенко Б., Миронець І., Смірнов О. та ін. Дослідження технологій забезпечення кібербезпеки хмарних сервісів IaaS, PaaS та SaaS. Кібербезпека: освіта, наука, техніка. 2024. URL: <https://csecurity.kubg.edu.ua/index.php/journal/article/view/588> (дата звернення: 27.02.2026).
14. Dua G., Kaur N., Bhatt U. Comparative Analysis of Cloud Computing Service Models. Journal of Cloud Computing. 2023. Vol.12, No.1. Article 45 DOI: 10.1186/s13677-023-00421-3.
15. React Documentation. Meta Open Source. 2024. URL: <https://react.dev/learn> (дата звернення: 05.03.2026).
16. Node.js Documentation. OpenJS Foundation. 2024. URL: <https://nodejs.org/docs/latest-v18.x/api/> (дата звернення: 11.03.2026).
17. Express.js Documentation. OpenJS Foundation. 2024. URL: <https://expressjs.com/en/4x/api.html> (дата звернення: 17.03.2026).
18. PostgreSQL 15 Documentation. PostgreSQL Global Development Group. 2023. URL: <https://www.postgresql.org/docs/15/> (дата звернення: 23.03.2026).
19. Docker Documentation. Docker Inc. 2024. URL: <https://docs.docker.com/get-started/> (дата звернення: 02.04.2026).
20. Terraform Documentation. HashiCorp. 2024. URL: <https://developer.hashicorp.com/terraform/intro> (дата звернення: 08.04.2026).
21. CNCF Annual Survey 2023. Cloud Native Computing Foundation. 2024. URL: <https://www.cncf.io/reports/cncf-annual-survey-2023/> (дата звернення: 15.04.2026).

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	62
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

22. State of Cloud Strategy Survey 2023. HashiCorp Inc. 2023. URL: <https://www.hashicorp.com/state-of-the-cloud> (дата звернення: 21.04.2026).
23. NIST SP 800-204C. Implementation of DevSecOps for a Microservices-based Application. NIST. 2022. URL: <https://csrc.nist.gov/pubs/sp/800/204/c/final> (дата звернення: 28.04.2026).
24. NIST SP 800-210. General Access Control Guidance for Cloud Systems. NIST. 2020. URL: <https://csrc.nist.gov/pubs/sp/800/210/final> (дата звернення: 06.05.2026).
25. NIST Cybersecurity Framework 2.0. NIST. 2024. URL: <https://www.nist.gov/cyberframework> (дата звернення: 13.05.2026).
26. Flexera. State of the Cloud Report 2024. Flexera Software LLC. 2024. URL: <https://www.flexera.com/blog/finops/cloud-computing-trends-flexera-2024-state-of-the-cloud-report> (дата звернення: 19.05.2026).
27. Stack Overflow. Developer Survey 2024. Stack Overflow. 2024. URL: <https://survey.stackoverflow.co/2024/> (дата звернення: 26.05.2026).
28. Oleksiuk V. P., Oleksiuk O. R., Vakaliuk T. A. An Experiment on the Implementation of the Methodology of Teaching Cloud Technologies to Future Computer Science Teachers. Proceedings of the 1st Symposium on Advances in Educational Technology (AET 2020). SciTePress, 2022. Vol.1. P.590–604. DOI: 10.5220/0010926400003364.
29. Cloud Native Survey 2022. Cloud Native Computing Foundation. 2022. URL: <https://www.cncf.io/reports/cncf-annual-survey-2022/> (дата звернення: 03.03.2026).
30. Socket.IO Documentation. OpenJS Foundation. 2024. URL: <https://socket.io/docs/v4/> (дата звернення: 09.03.2026).

ДОДАТОК А

Порівняльна характеристика наявних рішень для управління хмарними інфраструктурами

(до розділу 1)

Таблиця А.1 — Детальний порівняльний аналіз платформ оркестрації

Критерій	Terraform	Kubernetes	Portainer	VMware vRealize	AWS Console	Розроблений застосунок
Тип рішення	IaC-інструмент	Оркестратор контейнерів	UI для Docker	Корпоративна платформа	Нативна консоль	Вебзастосунок
Гібридна хмара	Так	Частково	Ні	Так	Лише AWS	Так
Відкритий код	Так (BSL)	Так (Apache 2.0)	Так / Pro	Ні	Ні	Так
Веб-інтерфейс	Ні (CLI)	Частково	Так	Так	Так	Так
Моніторинг у реальному часі	Ні	Так (з плагінами)	Частково	Так	Так	Так (WebSocket)
Управління Docker	Так (провайдер)	Так	Так	Обмежено	Ні	Так
Журнал подій	Ні	Так	Частково	Так	CloudTrail	Так (централізований)
Складність розгортання	Середня	Висока	Низька	Дуже висока	Хмарна	Низька (Docker Compose)
Вартість	Безкоштовно/платно	Безкоштовно	Безкоштовно/платно	Висока (ліцензія)	Pay-per-use	Безкоштовно
Рольова модель доступу	Обмежена	Так (RBAC)	Так	Розширена	IAM	Так (3 ролі)
Візуалізація топології	Ні	Так (з плагінами)	Ні	Так	Так	Так (React Flow)
Мультипровайдерність	Так	Частково	Ні	Обмежено	Лише AWS	Локальні сервери та Docker-хости (інтеграцію з хмарними API передбачено архітектурою)
Цільова аудиторія	DevOps	DevOps/SRE	Системний адміністратор	Enterprise	AWS-розробники	DevOps / IT-адміністратор

ДОДАТОК Б

Конфігураційні файли розгортання застосунку

(до розділу 2)

Лістинг Б.1 — Файл docker-compose.yml

```
version: "3.9"

services:
  # — PostgreSQL Database —————
  postgres:
    image: postgres:15-alpine
    container_name: hybrid_cloud_db
    restart: unless-stopped
    environment:
      POSTGRES_DB: ${DB_NAME:-hybrid_cloud}
      POSTGRES_USER: ${DB_USER:-postgres}
      POSTGRES_PASSWORD: ${DB_PASSWORD:?DB_PASSWORD is required}
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./backend/migrations/init.sql:/docker-entrypoint-initdb.d/init.sql
    ports:
      - "${DB_PORT:-5432}:5432"
    networks:
      - app_network
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${DB_USER:-postgres}"]
      interval: 10s
      timeout: 5s
      retries: 5

  # — Backend (Node.js / Express) —————
  backend:
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: hybrid_cloud_api
    restart: unless-stopped
    depends_on:
      postgres:
        condition: service_healthy
    environment:
      NODE_ENV: ${NODE_ENV:-production}
      PORT: ${BACKEND_PORT:-5000}
      DB_HOST: postgres
      DB_PORT: 5432
      DB_NAME: ${DB_NAME:-hybrid_cloud}
      DB_USER: ${DB_USER:-postgres}
      DB_PASSWORD: ${DB_PASSWORD:?DB_PASSWORD is required}
      JWT_SECRET: ${JWT_SECRET}
      JWT_EXPIRES_IN: ${JWT_EXPIRES_IN:-24h}
      BCRYPT_ROUNDS: ${BCRYPT_ROUNDS:-12}
      CORS_ORIGIN: ${CORS_ORIGIN:-http://192.168.1.100:3000}
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock:ro
    ports:
      - "${BACKEND_PORT:-5000}:5000"
```

Антонюк А.Р.

Вакалюк Т.А.

ІПЗ.КР.Б – 121 – 26 – ПЗ

65

Змін.

Арх.

№ докум.

Підпис

Дата

```

    networks:
      - app_network

# — Frontend (React) —
frontend:
  build:
    context: ./frontend
    dockerfile: Dockerfile
  args:
    REACT_APP_API_URL: ${REACT_APP_API_URL:-http://192.168.1.100:5000}
    REACT_APP_WS_URL: ${REACT_APP_WS_URL:-ws://192.168.1.100:5000}
  container_name: hybrid_cloud_ui
  restart: unless-stopped
  depends_on:
    - backend
  ports:
    - "${FRONTEND_PORT:-3000}:80"
  networks:
    - app_network

volumes:
  postgres_data:
    driver: local

networks:
  app_network:
    driver: bridge

```

Лістинг Б.2 — Файл .env.example

```

# — Application —
NODE_ENV=development
FRONTEND_PORT=3000
BACKEND_PORT=5000

# — Database —
DB_HOST=db
DB_PORT=5432
DB_NAME=hybrid_cloud
DB_USER=postgres
DB_PASSWORD=your_secure_password_here

# — Authentication —
# Generate with: node -e
"console.log(require('crypto').randomBytes(64).toString('hex'))"
JWT_SECRET=your_jwt_secret_key_minimum_32_chars
JWT_EXPIRES_IN=24h
BCRYPT_ROUNDS=12

# — CORS —
CORS_ORIGIN=http://192.168.1.100:3000

# — React App —
REACT_APP_API_URL=http://192.168.1.100:5000/api
REACT_APP_WS_URL=ws://192.168.1.100:5000

# — Optional: Email Notifications —
SMTP_HOST=smtp.gmail.com
SMTP_PORT=587
SMTP_USER=

```

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	66
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

```
SMTP_PASS=
SMTP_FROM=noreply@hybrid-cloud.local
```

Лістинг Б.3 — Файл backend/Dockerfile

```
FROM node:18-alpine AS builder
WORKDIR /app
COPY package*.json ./
RUN npm ci --only=production
```

```
FROM node:18-alpine
RUN apk add --no-cache dumb-init
WORKDIR /app
ENV NODE_ENV=production
COPY --from=builder /app/node_modules ./node_modules
COPY . .
EXPOSE 5000
USER node
ENTRYPOINT ["dumb-init", "--"]
CMD ["node", "src/server.js"]
```

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	67
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

ДОДАТОК В

Лістинг вихідного коду серверної частини (Node.js/Express)

(до розділу 3)

Лістинг В.1 — Файл src/server.js (точка входу застосунку)

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');
const cors = require('cors');
const dotenv = require('dotenv');
const swaggerUi = require('swagger-ui-express');

const { sequelize } = require('./models');
const routes = require('./routes');
const { swaggerSpec } = require('./config/swagger');
const { initSocketHandlers } = require('./sockets/socketHandlers');
const { startMetricsCollector } = require('./services/metricsCollector');
const logger = require('./utils/logger');

dotenv.config();

const app = express();
const server = http.createServer(app);

// — Socket.IO —
const io = new Server(server, {
  cors: { origin: process.env.CORS_ORIGIN, methods: ['GET', 'POST'] },
  transports: ['websocket', 'polling'],
});
// Доступ до io з контролерів через req.app.get('io')
app.set('io', io);

// — Middleware —
app.use(cors({ origin: process.env.CORS_ORIGIN, credentials: true }));
app.use(express.json({ limit: '10mb' }));
app.use(express.urlencoded({ extended: true }));

// — API documentation (OpenAPI / Swagger UI) —
app.use('/api-docs', swaggerUi.serve, swaggerUi.setup(swaggerSpec));

// — Routes —
app.use('/api', routes);

app.get('/api/health', (req, res) => {
  res.json({ status: 'ok', timestamp: new Date().toISOString(), uptime:
process.uptime() });
});

// — Error Handler —
app.use((err, req, res, next) => {
  logger.error(`${err.status || 500} - ${err.message} - ${req.originalUrl}`);
  res.status(err.status || 500).json({
```

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	68
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

```

        error: process.env.NODE_ENV === 'production' ? 'Internal server error' :
err.message,
    });
});

// — Start —————
const PORT = process.env.PORT || 5000;

sequelize.authenticate()
    .then(() => sequelize.sync({ alter: false }))
    .then(() =>
        { initSocketHandlers(io);
          startMetricsCollector(io);
          server.listen(PORT, () => logger.info(`Server running on port ${PORT}`));
        })
    .catch((err) => { logger.error('DB connection failed:', err);
process.exit(1); });

```

Лістинг В.2 — Файл src/middleware/authMiddleware.js

```

const jwt = require('jsonwebtoken');
const { User, Role } = require('../models');

/**
 * Verify JWT token and attach user to request
 */
const authenticate = async (req, res, next) =>
{
    try {
        const authHeader = req.headers['authorization'];
        if (!authHeader || !authHeader.startsWith('Bearer ')) {
            return res.status(401).json({ error: 'No token provided' });
        }
        const token = authHeader.split(' ')[1];
        const decoded = jwt.verify(token, process.env.JWT_SECRET);

        const user = await User.findByPk(decoded.id,
            { include: [{ model: Role, as: 'role' }],
              attributes: { exclude: ['password'] },
            });

        if (!user || user.isBlocked) {
            return res.status(401).json({ error: 'User not found or blocked' });
        }

        req.user = user;
        next();
    } catch (err) {
        if (err.name === 'TokenExpiredError') {
            return res.status(401).json({ error: 'Token expired' });
        }
        return res.status(401).json({ error: 'Invalid token' });
    }
};

/**
 * Role-based access control middleware factory
 * @param {...string} roles - allowed roles (admin, operator, viewer)
 */
const authorize = (...roles) => (req, res, next) => {
    if (!req.user || !roles.includes(req.user.role.name)) {
        return res.status(403).json({ error: 'Insufficient permissions' });
    }
};

```

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	69
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    }
    next();
  };

```

```
module.exports = { authenticate, authorize };
```

Лістинг В.3 — Файл `src/controllers/authController.js`

```

const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');
const { User, Role } = require('../models');
const logger = require('../utils/logger');

const BCRYPT_ROUNDS = parseInt(process.env.BCRYPT_ROUNDS) || 12;
const LOCK_THRESHOLD = 3;
const LOCK_MINUTES = 15;

exports.register = async (req, res) =>
{
  try {
    const { username, email, password } = req.body;

    const exists = await User.findOne({ where: { email } });
    if (exists) return res.status(409).json({ error: 'Email already in use' });

    const viewerRole = await Role.findOne({ where: { name: 'viewer' } });
    const hash = await bcrypt.hash(password, BCRYPT_ROUNDS);

    const user = await User.create({
      username, email, password: hash, roleId: viewerRole.id,
    });

    const token = jwt.sign(
      { id: user.id, role: viewerRole.name },
      process.env.JWT_SECRET, { expiresIn: process.env.JWT_EXPIRES_IN || '24h' }
    );

    logger.info(`New user registered: ${email}`);
    res.status(201).json({ token, user: { id: user.id, username, email, role: 'viewer' } });
  } catch (err) {
    logger.error('Registration error:', err);
    res.status(500).json({ error: 'Registration failed' });
  }
};

exports.login = async (req, res) =>
{
  try {
    const { email, password } = req.body;

    const user = await User.findOne({
      where: { email },
      include: [{ model: Role, as: 'role' }],
    });

    if (!user) return res.status(401).json({ error: 'Invalid credentials' });

    // Check account lock

```

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	70
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    if (user.lockedUntil && user.lockedUntil > new Date()) {
      const remaining = Math.ceil((user.lockedUntil - new Date()) / 60000);
      return res.status(429).json({ error: `Account locked. Try again in
${remaining} min` });
    }

    const valid = await bcrypt.compare(password, user.password);
    if (!valid) {
      const attempts = (user.failedAttempts || 0) + 1;
      const update = { failedAttempts: attempts };
      if (attempts >= LOCK_THRESHOLD) {
        update.lockedUntil = new Date(Date.now() + LOCK_MINUTES * 60000);
        update.failedAttempts = 0;
      }
      await user.update(update);
      return res.status(401).json({ error: 'Invalid credentials' });
    }

    await user.update({ failedAttempts: 0, lockedUntil: null, lastLogin: new
Date() });

    const token = jwt.sign(
      { id: user.id, role: user.role.name },
      process.env.JWT_SECRET, { expiresIn: process.env.JWT_EXPIRES_IN || '24h' }
    );

    res.json({ token, user: {
      id: user.id, username: user.username,
      email: user.email, role: user.role.name,
    }});
  } catch (err) {
    logger.error('Login error:', err);
    res.status(500).json({ error: 'Login failed' });
  }
};
/** GET /api/auth/me – поточний автентифікований користувач. */
exports.me = async (req, res) => {
  try {
    const user = await User.findByPk(req.user.id,
      { attributes: ['id', 'username', 'email'],
        include: [{ model: Role, as: 'role', attributes: ['name'] }],
      });
    if (!user) return res.status(404).json({ error: 'User not found' });
    res.json(user);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

```

Лістинг В.4 — Файл src/controllers/containerController.js

```

const fs = require('fs');
const Docker = require('dockerode');
const { Container, Node, Log } = require('../models');
const logger = require('../utils/logger');

/**
 * Повертає клієнт Docker для конкретного вузла. Локальний демон
 * адресується через UNIX-сокет, віддалені вузли – через захищений
 * TLS-порт 2376 із взаємною автентифікацією за сертифікатами,
 * шляхи до яких задаються змінними середовища.
 */

```

		Антонюк А.Р.			ІПЗ.КР.Б – 121 – 26 – ПЗ	71
		Вакалюк Т.А.				
Змін.	Арх.	№ докум.	Підпис	Дата		

```

const getDockerClient = (node) => {
  if (node.host === 'host.docker.internal' || node.host === '127.0.0.1')
    { return new Docker({ socketPath: '/var/run/docker.sock' }); }
  return new
    Docker({ host:
      node.host,
      port: node.dockerPort || 2376,
      protocol: 'https',
      ca: fs.readFileSync(process.env.DOCKER_TLS_CA),
      cert: fs.readFileSync(process.env.DOCKER_TLS_CERT),
      key: fs.readFileSync(process.env.DOCKER_TLS_KEY),
    });
};

exports.listContainers = async (req, res) =>
{ try {
  const { nodeId } = req.query;
  const where = nodeId ? { nodeId } : {};
  const containers = await Container.findAll({
    where,
    include: [{ model: Node, as: 'node', attributes: ['name', 'host'] }],
    order: [['createdAt', 'DESC']],
  });
  res.json(containers);
} catch (err) {
  res.status(500).json({ error: err.message });
}
};

exports.deployContainer = async (req, res) =>
{ try {
  const { nodeId, image, name, ports, env, cmd } = req.body;
  const node = await Node.findPk(nodeId);
  if (!node) return res.status(404).json({ error: 'Node not found' });

  const docker = getDockerClient(node);

  await new Promise((resolve, reject) =>
    { docker.pull(image, (err, stream) => {
      if (err) return reject(err);
      docker.modem.followProgress(stream, resolve);
    });
  });

  const PortBindings = {};
  const ExposedPorts = {};
  (ports || []).forEach((p) => {
    const [hostPort, containerPort] = p.split(':');
    ExposedPorts[`${containerPort}/tcp`] = {};
    PortBindings[`${containerPort}/tcp`] = [{ HostPort: hostPort }];
  });

  const container = await
    docker.createContainer({ Image: image,
      name: name || undefined,
      ExposedPorts,
      Env: (env || []).map((e) => `${e.key}=${e.value}`),
      Cmd: cmd ? cmd.split(' ') : undefined,
      HostConfig: { PortBindings, RestartPolicy: { Name: 'unless-stopped' } },
    });
};

```

Антонюк А.Р.

Вакалюк Т.А.

ІІЗ.КР.Б – 121 – 26 – ПЗ

72

Змін. Арк. № докум. Підпис Дата

```

    await container.start();
    const info = await container.inspect();

    const dbContainer = await
      Container.create({ containerId:
        info.Id.slice(0, 12),
        name: info.Name.slice(1),
        image,
        status: 'running',
        ports: JSON.stringify(ports || []),
        nodeId,
      });

    await
      Log.create({ leve
        l: 'INFO',
        message: `Container ${dbContainer.name} deployed from image ${image}`,
        source: 'container',
        nodeId,
        userId: req.user.id,
      });

    req.app.get('io').emit('container:status', {
      nodeId, containerId: dbContainer.containerId, status: 'running',
    });

    res.status(201).json(dbContainer);
  } catch (err) {
    logger.error('Deploy error:', err);
    res.status(500).json({ error: err.message });
  }
};

exports.containerAction = async (req, res) =>
{ try {
  const { id } = req.params;
  const { action } = req.body; // start | stop | restart | remove
  const dbContainer = await Container.findByPk(id, {
    include: [{ model: Node, as: 'node' }],
  });
  if (!dbContainer) return res.status(404).json({ error: 'Container not
found' });

  const docker = getDockerClient(dbContainer.node);
  const container = docker.getContainer(dbContainer.containerId);

  const actionMap = {
    start: () => container.start(),
    stop: () => container.stop(),
    restart: () => container.restart(),
    remove: async () => { await container.stop().catch(() => {}); await
container.remove(); },
  };
  if (!actionMap[action]) return res.status(400).json({ error: 'Invalid
action' });
  await actionMap[action]();

  const statusMap = { start: 'running', stop: 'stopped', restart: 'running',
remove: 'removed' };
  await dbContainer.update({ status: statusMap[action] });
}

```

```
await
  Log.create({ leve
    l: 'INFO',
    message: `Container ${dbContainer.name} action: ${action}`,
    source: 'container',
    nodeId: dbContainer.nodeId,
    userId: req.user.id,
  });

req.app.get('io').emit('container:status', {
  nodeId: dbContainer.nodeId, containerId: dbContainer.containerId, status:
statusMap[action],
});

res.json({ success: true, status: statusMap[action] });
} catch (err) {
  res.status(500).json({ error: err.message });
}
};

/**
 * Повертає останні рядки журналу контейнера безпосередньо з Docker
 * Engine API (еквівалент docker logs --tail).
 */
exports.getContainerLogs = async (req, res) =>
{ try {
  const { id } = req.params;
  const tail = parseInt(req.query.tail, 10) || 200;
  const dbContainer = await Container.findByPk(id, {
    include: [{ model: Node, as: 'node' }],
  });
  if (!dbContainer) return res.status(404).json({ error: 'Container not
found' });

  const docker = getDockerClient(dbContainer.node);
  const buffer = await
    docker.getContainer(dbContainer.containerId).logs({ stdout: true, stderr:
true, tail, timestamps: true,
  });
  res.type('text/plain').send(buffer.toString('utf8'));
} catch (err) {
  res.status(500).json({ error: err.message });
}
};
```

Лістинг B.5 — Файл src/services/metricsCollector.js

```
const http = require('http');
const Docker = require('dockerode');
const { Node, Metric, Alert, AlertRule } = require('../models');
const { Op } = require('sequelize');
const logger = require('../utils/logger');

const COLLECT_INTERVAL = 30 * 1000; // 30 секунд
const AGENT_PORT = parseInt(process.env.NODE_EXPORTER_PORT, 10) || 9100;
const AGENT_TIMEOUT = 4000;

/**
 * Виконує HTTP-запит до агента node_exporter і повертає сирий
 * текст експозиції у форматі Prometheus.
 */
const fetchExporterText = (host) =>
```

		Вакалюк Т.А.			74
Змін.	Арх.	№ докум.	Підпис	Дата	ІІЗ.КР.Б – 121 – 26 – ПЗ

```

new Promise((resolve, reject) =>
{ const req = http.get(
{ host, port: AGENT_PORT, path: '/metrics', timeout: AGENT_TIMEOUT },
(res) => {
if (res.statusCode !== 200)
{ res.resume();
return reject(new Error(`node_exporter HTTP ${res.statusCode}`));
}
let body = '';
res.setEncoding('utf8');
res.on('data', (chunk) => { body += chunk; });
res.on('end', () => resolve(body));
}
);
req.on('timeout', () => req.destroy(new Error('node_exporter timeout')));
req.on('error', reject);
});

/** Сума значень усіх семплів метрики із заданим префіксом імені. */
const sumSamples = (text, name, filter = () => true) => {
let total = 0;
for (const line of text.split('\n')) {
if (line.startsWith('#') || !line.startsWith(name)) continue;
if (!filter(line)) continue;
const value = parseFloat(line.slice(line.lastIndexOf(' ') + 1));
if (!Number.isNaN(value)) total += value;
}
return total;
};

/**
* Розбирає експозицію node_exporter та обчислює відсоткові показники
* завантаження CPU, використання RAM і кореневої файлової системи.
*/
const parseExporterMetrics = (text) => {
const idle = sumSamples(text, 'node_cpu_seconds_total', (l) =>
l.includes('mode=idle'));
const total = sumSamples(text, 'node_cpu_seconds_total');
const cpu = total > 0 ? (1 - idle / total) * 100 : 0;

const memTotal = sumSamples(text, 'node_memory_MemTotal_bytes');
const memAvail = sumSamples(text, 'node_memory_MemAvailable_bytes');
const ram = memTotal > 0 ? ((memTotal - memAvail) / memTotal) * 100 : 0;

const fsSize = sumSamples(text, 'node_filesystem_size_bytes', (l) =>
l.includes('mountpoint="/'));
const fsAvail = sumSamples(text, 'node_filesystem_avail_bytes', (l) =>
l.includes('mountpoint="/'));
const disk = fsSize > 0 ? ((fsSize - fsAvail) / fsSize) * 100 : 0;

return {
cpu: Math.round(cpu * 10) / 10,
ram: Math.round(ram * 10) / 10,
disk: Math.round(disk * 10) / 10,
};
};

/**
* Збирає показники навантаження Docker-хоста через офіційний Engine API.
* CPU й RAM усереднюються за запущеними контейнерами, дисковий показник

```

		Антонюк А.Р.			
--	--	--------------	--	--	--

		Вакалюк Т.А.				75
Змін.	Арх.	№ докум.	Підпис	Дата	ІІЗ.КР.Б – 121 – 26 – ПЗ	

```

* береться з df-зведення демона.
*/
const fetchDockerMetrics = async (node) =>
{
  const docker = new Docker(
    node.host === '127.0.0.1' || node.host === 'host.docker.internal'
      ? { socketPath: '/var/run/docker.sock' }
      : { host: node.host, port: node.dockerPort || 2376, protocol: 'https' }
  );

  const info = await docker.info();
  const list = await docker.listContainers({ all: false });

  let cpuPercent = 0;
  let memUsed = 0;
  for (const c of list) {
    const stats = await docker.getContainer(c.Id).stats({ stream: false });
    const cpuDelta = stats.cpu_stats.cpu_usage.total_usage -
stats.precpu_stats.cpu_usage.total_usage;
    const sysDelta = stats.cpu_stats.system_cpu_usage -
stats.precpu_stats.system_cpu_usage;
    const cores = stats.cpu_stats.online_cpus || 1;
    if (sysDelta > 0 && cpuDelta > 0) cpuPercent += (cpuDelta / sysDelta) *
cores * 100;
    memUsed += stats.memory_stats.usage || 0;
  }

  const memTotal = info.MemTotal || 1;
  const df = await docker.df();
  const layer = (df.LayersSize || 0);

  return {
    cpu: Math.round(Math.min(100, cpuPercent) * 10) / 10,
    ram: Math.round(Math.min(100, (memUsed / memTotal) * 100) * 10) / 10,
    disk: Math.round(Math.min(100, (layer / memTotal) * 100) * 10) / 10,
  };
};

/**
 * Демонстраційний генератор для стендових вузлів без агента.
 * Вмикається лише за явного прапорця node.demoMode і потрібен
 * виключно для показу інтерфейсу без реального парку серверів.
 */
const demoMetrics = (node) => {
  const seed = (node.id * 2654435761) % 2 ** 32;
  const wave = Math.abs(Math.sin(seed + Date.now() / 60000));
  return {
    cpu: Math.round((20 + wave * 55) * 10) / 10,
    ram: Math.round((35 + wave * 40) * 10) / 10,
    disk: Math.round((25 + wave * 20) * 10) / 10,
  };
};

/**
 * Уніфікований збір метрик вузла: спершу агент node_exporter,
 * далі Docker Engine API, і лише як крайній випадок – демо-режим.
 */
const fetchNodeMetrics = async (node) =>
{
  try {
    if (node.type === 'docker') return await fetchDockerMetrics(node);
    const text = await fetchExporterText(node.host);

```

		Антонюк А.Р.				
--	--	--------------	--	--	--	--

		Вакалюк Т.А.			ІПЗ.КР.Б – 121 – 26 – ПЗ	76
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    return parseExporterMetrics(text);
  } catch (err) {
    if (node.demonMode) {
      logger.warn(`Node ${node.host}: агент недоступний, демо-режим
(${err.message})`);
      return demoMetrics(node);
    }
    throw err;
  }
};

const checkAlerts = async (node, metrics, io) => {
  const rules = await AlertRule.findAll({ where: { nodeId: node.id, isActive:
true } });
  for (const rule of rules) {
    const value = metrics[rule.metric];
    if (value >= rule.threshold) {
      const existing = await Alert.findOne({
        where: { nodeId: node.id, metric: rule.metric, resolvedAt: null },
      });
      if (!existing) {
        const alert = await
          Alert.create({ nodeId: node.id,
            metric: rule.metric,
            value,
            threshold: rule.threshold,
            level: value >= rule.threshold + 10 ? 'critical' : 'warning',
          });
        io.emit('alert:new', { nodeId: node.id, alert });
      }
    } else {
      await Alert.update(
        { resolvedAt: new Date() },
        { where: { nodeId: node.id, metric: rule.metric, resolvedAt: null } }
      );
    }
  }
};

exports.startMetricsCollector = (io) =>
  { setInterval(async () => {
    try {
      const nodes = await Node.findAll({ where: { status: 'online' } });
      for (const node of nodes) {
        try {
          const metrics = await fetchNodeMetrics(node);
          await Metric.create({ nodeId: node.id, ...metrics, recordedAt: new
Date() });
          await checkAlerts(node, metrics, io);
          io.emit('metrics:update', { nodeId: node.id, metrics, timestamp: new
Date() });
        } catch (err) {
          await node.update({ status: 'warning' });
          io.emit('node:status', { nodeId: node.id, status: 'warning', reason:
err.message });
          logger.error(`Збір метрик вузла ${node.host} не вдался:
${err.message}`);
        }
      }
      await Metric.destroy({
        where: { recordedAt: { [Op.lt]: new Date(Date.now() - 30 * 24 * 60 * 60
* 1000) } },

```

		Антонюк А.Р.				
--	--	--------------	--	--	--	--

		Вакалюк Т.А.				77
Змін.	Арх.	№ докум.	Підпис	Дата	ІПЗ.КР.Б – 121 – 26 – ПЗ	

```

    });
  } catch (err) {
    logger.error('Metrics collector error:', err);
  }
}, COLLECT_INTERVAL);

logger.info(`Збирач метрик запущено (інтервал: ${COLLECT_INTERVAL / 1000}
с)`);
};

```

Лістинг вихідного коду серверної частини (Node.js/Express)

Лістинг В.6 — Файл src/models/index.js (моделі Sequelize та опис схеми БД)

```

const { Sequelize, DataTypes } = require('sequelize');

const sequelize = new Sequelize({
  host: process.env.DB_HOST || 'localhost',
  port: process.env.DB_PORT || 5432,
  database: process.env.DB_NAME || 'hybrid_cloud',
  username: process.env.DB_USER || 'postgres',
  password: process.env.DB_PASSWORD || 'secret',
  dialect: 'postgres',
  logging: false,
  pool: { max: 10, min: 0, acquire: 30000, idle: 10000 },
});

// — Role
const Role = sequelize.define('Role', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  name: { type: DataTypes.STRING(50), allowNull: false, unique: true },
  permissions: { type: DataTypes.JSONB, defaultValue: {} },
}, { tableName: 'roles', timestamps: false });

// — User
const User = sequelize.define('User', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  username: { type: DataTypes.STRING(100), allowNull: false, unique: true },
  email: { type: DataTypes.STRING(255), allowNull: false, unique: true },
  password: { type: DataTypes.STRING(255), allowNull: false },
  roleId: { type: DataTypes.INTEGER, allowNull: false },
  isBlocked: { type: DataTypes.BOOLEAN, defaultValue: false },
  failedAttempts: { type: DataTypes.INTEGER, defaultValue: 0 },
  lockedUntil: { type: DataTypes.DATE },
  lastLogin: { type: DataTypes.DATE },
}, { tableName: 'users', underscored: true });

// — Infrastructure
const Infrastructure = sequelize.define('Infrastructure', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  name: { type: DataTypes.STRING(255), allowNull: false },
  type: { type: DataTypes.ENUM('aws', 'azure', 'gcp', 'local', 'docker'), defaultValue: 'local' },
  description: { type: DataTypes.TEXT },
  status: { type: DataTypes.ENUM('active', 'inactive', 'error'), defaultValue: 'active' },
  config: { type: DataTypes.JSONB, defaultValue: {} },
  ownerId: { type: DataTypes.INTEGER },
}, { tableName: 'infrastructures', underscored: true });

```

		АНТОНЮК А.Р.				
--	--	--------------	--	--	--	--

```
// — Node
const Node = sequelize.define('Node', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  infrald: { type: DataTypes.INTEGER, allowNull: false },
  name: { type: DataTypes.STRING(255), allowNull: false },
  host: { type: DataTypes.STRING(255), allowNull: false },
  dockerPort: { type: DataTypes.INTEGER, defaultValue: 2376 },
  os: { type: DataTypes.STRING(100) },
  status: { type: DataTypes.ENUM('online', 'offline', 'warning'), defaultValue: 'online' },
  cpuCores: { type: DataTypes.INTEGER, defaultValue: 4 },
  ramGb: { type: DataTypes.FLOAT, defaultValue: 8 },
  diskGb: { type: DataTypes.FLOAT, defaultValue: 100 },
}, { tableName: 'nodes', underscored: true });

// — Container
const Container = sequelize.define('Container', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  containerId: { type: DataTypes.STRING(64) },
  name: { type: DataTypes.STRING(255), allowNull: false },
  image: { type: DataTypes.STRING(255), allowNull: false },
  status: { type: DataTypes.ENUM('running', 'stopped', 'error', 'removed'), defaultValue: 'stopped' },
  ports: { type: DataTypes.JSONB, defaultValue: [] },
  nodeId: { type: DataTypes.INTEGER, allowNull: false },
}, { tableName: 'containers', underscored: true });

// — Metric
const Metric = sequelize.define('Metric', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  nodeId: { type: DataTypes.INTEGER, allowNull: false },
  cpu: { type: DataTypes.FLOAT },
  ram: { type: DataTypes.FLOAT },
  disk: { type: DataTypes.FLOAT },
  recordedAt: { type: DataTypes.DATE, defaultValue: DataTypes.NOW },
}, { tableName: 'metrics', underscored: true, timestamps: false });

// — Log
const Log = sequelize.define('Log', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  level: { type: DataTypes.ENUM('DEBUG', 'INFO', 'WARNING', 'ERROR'), defaultValue: 'INFO' },
  message: { type: DataTypes.TEXT, allowNull: false },
  source: { type: DataTypes.STRING(100) },
  nodeId: { type: DataTypes.INTEGER },
  userId: { type: DataTypes.INTEGER },
}, { tableName: 'logs', underscored: true });

// — Alert
const Alert = sequelize.define('Alert', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  nodeId: { type: DataTypes.INTEGER, allowNull: false },
  metric: { type: DataTypes.STRING(50) },
  value: { type: DataTypes.FLOAT },
  threshold: { type: DataTypes.FLOAT },
  level: { type: DataTypes.ENUM('warning', 'critical'), defaultValue: 'warning' },
  resolvedAt: { type: DataTypes.DATE },
}, { tableName: 'alerts', underscored: true });
```

```
// — AlertRule
const AlertRule = sequelize.define('AlertRule', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  nodeId: { type: DataTypes.INTEGER, allowNull: false },
  metric: { type: DataTypes.ENUM('cpu', 'ram', 'disk') },
  threshold: { type: DataTypes.FLOAT, defaultValue: 80 },
  isActive: { type: DataTypes.BOOLEAN, defaultValue: true },
}, { tableName: 'alert_rules', underscored: true });

// — Associations
User.belongsTo(Role, { foreignKey: 'roleId', as: 'role' }); Role.hasMany(User, { foreignKey: 'roleId' });

Infrastructure.belongsTo(User, { foreignKey: 'ownerId', as: 'owner' });
User.hasMany(Infrastructure, { foreignKey: 'ownerId' });

Node.belongsTo(Infrastructure, { foreignKey: 'infraId', as: 'infrastructure' });
Infrastructure.hasMany(Node, { foreignKey: 'infraId', as: 'nodes' });

Container.belongsTo(Node, { foreignKey: 'nodeId', as: 'node' });
Node.hasMany(Container, { foreignKey: 'nodeId', as: 'containers' });

Metric.belongsTo(Node, { foreignKey: 'nodeId', as: 'node' });
Node.hasMany(Metric, { foreignKey: 'nodeId', as: 'metrics' });

Log.belongsTo(Node, { foreignKey: 'nodeId', as: 'node' });
Log.belongsTo(User, { foreignKey: 'userId', as: 'user' });

Alert.belongsTo(Node, { foreignKey: 'nodeId', as: 'node' });
AlertRule.belongsTo(Node, { foreignKey: 'nodeId', as: 'node' });

module.exports = { sequelize, Role, User, Infrastructure, Node, Container, Metric, Log, Alert, AlertRule };
```

Лістинг В.7 — Файл src/routes/index.js (маршрути REST API)

```
const router = require('express').Router();
const { authenticate, authorize } = require('../middleware/authMiddleware');
const authCtrl = require('../controllers/authController');
const infraCtrl = require('../controllers/infrastructureController');
const nodeCtrl = require('../controllers/nodeController');
const containerCtrl = require('../controllers/containerController');
const metricsCtrl = require('../controllers/metricsController');
const logsCtrl = require('../controllers/logsController');
const alertCtrl = require('../controllers/alertController');
const userCtrl = require('../controllers/userController');

// Auth
router.post('/auth/register', authCtrl.register);
router.post('/auth/login', authCtrl.login);
router.get('/auth/me', authenticate, authCtrl.me);

// Dashboard stats
router.get('/stats', authenticate, infraCtrl.stats);

// Infrastructures
router.get('/infrastructures', authenticate, infraCtrl.list);
```

		Вакалюк Т.А.			ІІЗ.КР.Б – 121 – 26 – ПЗ	80
Змін.	Арх.	№ докум.	Підпис	Дата		

```

router.get('/infrastructures/:id', authenticate, infraCtrl.get);
router.post('/infrastructures', authenticate, authorize('admin', 'operator'),
infraCtrl.create);
router.put('/infrastructures/:id', authenticate, authorize('admin', 'operator'),
infraCtrl.update);
router.delete('/infrastructures/:id', authenticate, authorize('admin'),
infraCtrl.remove);

// Nodes
router.get('/nodes', authenticate, nodeCtrl.list);
router.get('/nodes/:id', authenticate, nodeCtrl.get);
router.post('/nodes', authenticate, authorize('admin', 'operator'),
nodeCtrl.create);
router.put('/nodes/:id', authenticate, authorize('admin', 'operator'),
nodeCtrl.update);
router.delete('/nodes/:id', authenticate, authorize('admin'), nodeCtrl.remove);
router.get('/nodes/:id/metrics', authenticate, metricsCtrl.getNodeMetrics);
router.get('/nodes/:id/logs', authenticate, logsCtrl.getNodeLogs);

// Containers – імена обробників збігаються з експортами containerController.js
router.get('/containers', authenticate, containerCtrl.listContainers);
router.post('/containers/deploy', authenticate, authorize('admin', 'operator'),
containerCtrl.deployContainer);
router.post('/containers/:id/action', authenticate, authorize('admin',
'operator'), containerCtrl.containerAction);
router.get('/containers/:id/logs', authenticate,
containerCtrl.getContainerLogs);

// Metrics
router.get('/metrics/latest', authenticate, metricsCtrl.getLatest);

// Logs
router.get('/logs', authenticate, logsCtrl.list);

// Alerts
router.get('/alerts', authenticate, alertCtrl.listAlerts);
router.patch('/alerts/:id/resolve', authenticate, authorize('admin',
'operator'), alertCtrl.resolveAlert);
router.get('/alert-rules', authenticate, alertCtrl.listRules);
router.post('/alert-rules', authenticate, authorize('admin', 'operator'),
alertCtrl.createRule);
router.delete('/alert-rules/:id', authenticate, authorize('admin', 'operator'),
alertCtrl.deleteRule);

// Users (admin only)
router.get('/users', authenticate, authorize('admin'), userCtrl.list);
router.patch('/users/:id/role', authenticate, authorize('admin'),
userCtrl.updateRole);
router.patch('/users/:id/block', authenticate, authorize('admin'),
userCtrl.toggleBlock);
router.delete('/users/:id', authenticate, authorize('admin'), userCtrl.remove);

module.exports = router;

```

Лістинг В.8 — Файл src/controllers/infrastructureController.js

```

const { Infrastructure, Node, Container } = require('../models');
const logger = require('../utils/logger');

exports.list = async (req, res) => {

```

		Антонюк А.Р.				
--	--	--------------	--	--	--	--

		Вакалюк Т.А.			ІІЗ.КР.Б – 121 – 26 – ІІЗ	81
Змін.	Арх.	№ докум.	Підпис	Дата		

```

try {
  const infras = await Infrastructure.findAll({
    include: [{ model: Node, as: 'nodes', attributes: ['id', 'name', 'status'] }], order:
    [['createdAt', 'DESC']],
  });
  res.json(infras);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.get = async (req, res) =>
{ try {
  const infra = await Infrastructure.findByPk(req.params.id, {
    include: [{ model: Node, as: 'nodes', include: [{ model: Container, as: 'containers' } ]}],
  });
  if (!infra) return res.status(404).json({ error: 'Infrastructure not found' }); res.json(infra);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.create = async (req, res) => { try
{
  const { name, type, description, config } = req.body;
  if (!name) return res.status(400).json({ error: 'Name is required' });
  const infra = await Infrastructure.create({ name, type: type || 'local', description, config: config || {}, ownerId: req.user.id });
  logger.info('Infrastructure created: ${name} by ${req.user.email}');
  res.status(201).json(infra);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.update = async (req, res) =>
{ try {
  const infra = await Infrastructure.findByPk(req.params.id); if
  (!infra) return res.status(404).json({ error: 'Not found' }); await
  infra.update(req.body);
  res.json(infra);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.remove = async (req, res) =>
{ try {
  const infra = await Infrastructure.findByPk(req.params.id); if
  (!infra) return res.status(404).json({ error: 'Not found' }); await
  infra.destroy();
  res.json({ success: true });
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.stats = async (req, res) => { try
{
  const { Infrastructure, Node, Container, Alert } = require('../models');
  const { Op } = require('sequelize');
  const [totalInfra, totalNodes, onlineNodes, runningContainers, activeAlerts] = await Promise.all([ Infrastructure.count(),
  Node.count(),
  Node.count({ where: { status: 'online' } }),
  Container.count({ where: { status: 'running' } }),

```

		Антошук А.Р.				
--	--	--------------	--	--	--	--

```

    Alert.count({ where: { resolvedAt: null } }),
  ]);
  res.json({ totalInfra, totalNodes, onlineNodes, runningContainers, activeAlerts });
} catch (err) { res.status(500).json({ error: err.message }); }
};

```

Лістинг В.9 — Файл `src/controllers/nodeController.js`

```

const { Node, Infrastructure, Container, Metric } = require('../models'); const
{ Op } = require('sequelize');

exports.list = async (req, res) =>
{ try {
  const where = req.query.infrald ? { infrald: req.query.infrald } : {}; const
  nodes = await Node.findAll({
    where,
    include: [{ model: Infrastructure, as: 'infrastructure', attributes: ['id', 'name', 'type'] }], order:
    [['createdAt', 'DESC']],
  });
  res.json(nodes);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.get = async (req, res) =>
{ try {
  const node = await Node.findByPk(req.params.id,
    { include: [
      { model: Infrastructure, as: 'infrastructure' },
      { model: Container, as: 'containers' },
    ],
  });
  if (!node) return res.status(404).json({ error: 'Node not found' });

  // Latest metric
  const latestMetric = await
  Metric.findOne({ where: { nodeId: node.id },
    order: [['recordedAt', 'DESC']],
  });
  res.json({ ...node.toJSON(), latestMetric });
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.create = async (req, res) => { try
{
  const { infrald, name, host, os, cpuCores, ramGb, diskGb, dockerPort } = req.body;
  if (!infrald || !name || !host) return res.status(400).json({ error: 'infrald, name, host required' });

  const node = await Node.create({ infrald, name, host, os, cpuCores, ramGb, diskGb, dockerPort });
  res.status(201).json(node);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.update = async (req, res) =>
{ try {
  const node = await Node.findByPk(req.params.id);
  if (!node) return res.status(404).json({ error: 'Not found' });
  await node.update(req.body);
}

```

		Вакалюк Т.А.			ІІЗ.КР.Б – 121 – 26 – ІІЗ	83
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    res.json(node);
  } catch (err) { res.status(500).json({ error: err.message }); }
};

exports.remove = async (req, res) =>
{ try {
  const node = await Node.findByPk(req.params.id);
  if (!node) return res.status(404).json({ error: 'Not found' });
  await node.destroy();
  res.json({ success: true });
} catch (err) { res.status(500).json({ error: err.message }); }
};

```

Лістинг В.10 — Файл src/controllers/alertController.js

```

const { Alert, AlertRule, Node } = require('../models');

exports.listAlerts = async (req, res) =>
{ try {
  const where = {};
  if (req.query.active === 'true') where.resolvedAt = null; const
  alerts = await Alert.findAll({
    where,
    include: [{ model: Node, as: 'node', attributes: ['id', 'name'] }], order:
    [['createdAt', 'DESC']],
    limit: 100,
  });
  res.json(alerts);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.resolveAlert = async (req, res) => { try {
  const alert = await Alert.findByPk(req.params.id);
  if (!alert) return res.status(404).json({ error: 'Alert not found' }); await
  alert.update({ resolvedAt: new Date() });
  res.json(alert);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.listRules = async (req, res) =>
{ try {
  const where = req.query.nodeId ? { nodeId: req.query.nodeId } : {};
  const rules = await AlertRule.findAll({ where, include: [{ model: Node, as: 'node', attributes: ['id', 'name'] }]});
  res.json(rules);
} catch (err) { res.status(500).json({ error: err.message }); }
};

exports.createRule = async (req, res) =>
{ try {
  const { nodeId, metric, threshold } = req.body;
  if (!nodeId || !metric) return res.status(400).json({ error: 'nodeId and metric required' }); const rule
  = await AlertRule.create({ nodeId, metric, threshold: threshold || 80 });
  res.status(201).json(rule);
} catch (err) { res.status(500).json({ error: err.message }); }
};

```

		Антонюк А.Р.				
--	--	--------------	--	--	--	--

```
exports.deleteRule = async (req, res) =>
{
  try {
    const rule = await AlertRule.findByPk(req.params.id);
    if (!rule) return res.status(404).json({ error: 'Not found' });
    await rule.destroy();
    res.json({ success: true });
  } catch (err) { res.status(500).json({ error: err.message }); }
};
```

ДОДАТОК Г

Лістинг вихідного коду клієнтської частини (React)

Лістинг Г.1 — Файл src/App.jsx (точка входу та маршрутизація)

```
import React from 'react';
import { BrowserRouter, Routes, Route, Navigate } from 'react-router-dom'; import
{ useSelector } from 'react-redux';
import AppLayout from './components/Layout';
import LoginPage from './pages/LoginPage'; import
RegisterPage from './pages/RegisterPage';
import DashboardPage from './pages/DashboardPage'; import
InfrastructurePage from './pages/InfrastructurePage'; import
NodesPage from './pages/NodesPage';
import ContainersPage from './pages/ContainersPage';
import MonitoringPage from './pages/MonitoringPage';
import LogsPage from './pages/LogsPage';
import AlertsPage from './pages/AlertsPage'; import
TopologyPage from './pages/TopologyPage'; import
UsersPage from './pages/UsersPage';

const PrivateRoute = ({ children }) => {
  const { isAuthenticated } = useSelector((s) => s.auth);
  return isAuthenticated ? children : <Navigate to="/login" replace />;
};

const AdminRoute = ({ children }) => {
  const { user, isAuthenticated } = useSelector((s) => s.auth); if
(!isAuthenticated) return <Navigate to="/login" replace />;
  if (user?.role !== 'admin') return <Navigate to="/dashboard" replace />;
  return children;
};

export default function App()
{ return (
  <BrowserRouter>
    <Routes>
      <Route path="/login" element={<LoginPage />} />
      <Route path="/register" element={<RegisterPage />} />
      <Route path="/" element={<PrivateRoute><AppLayout /></PrivateRoute>}>
        <Route index element={<Navigate to="/dashboard" replace />} />
        <Route path="dashboard" element={<DashboardPage />} />
        <Route path="infrastructures" element={<InfrastructurePage />} />
        <Route path="nodes" element={<NodesPage />} />
        <Route path="containers" element={<ContainersPage />} />
        <Route path="monitoring" element={<MonitoringPage />} />
        <Route path="logs" element={<LogsPage />} />
        <Route path="alerts" element={<AlertsPage />} />
        <Route path="topology" element={<TopologyPage />} />
        <Route path="users" element={<AdminRoute><UsersPage /></AdminRoute>} />
      </Route>
      <Route path="*" element={<Navigate to="/dashboard" replace />} />
    </Routes>
  </BrowserRouter>
);
}
```

		Вакалюк Т.А.			ІПЗ.КР.Б – 121 – 26 – ПЗ	86
Змін.	Арх.	№ докум.	Підпис	Дата		

Лістинг Г.2 — Файл `src/store/authSlice.js` (Redux: автентифікація)

```
import { createSlice } from '@reduxjs/toolkit';

const token = localStorage.getItem('token');
const user = (() => { try { return JSON.parse(localStorage.getItem('user')); } catch { return null; } })();

const authSlice =
  createSlice({ name: 'auth',
    initialState: { token, user, isAuthenticated: !!token },
    reducers: {
      setCredentials(state, { payload: { token, user } })
        { state.token = token;
          state.user = user;
          state.isAuthenticated = true;
          localStorage.setItem('token', token);
          localStorage.setItem('user', JSON.stringify(user));
        },
      logout(state) { state.token
        = null; state.user = null;
        state.isAuthenticated = false;
        localStorage.removeItem('token');
        localStorage.removeItem('user');
      },
    },
  });

export const { setCredentials, logout } = authSlice.actions;
export default authSlice.reducer;
```

Лістинг Г.3 — Файл `src/api/index.js` (HTTP-клієнт axios)

```
import axios from 'axios';

const api = axios.create({
  baseURL: import.meta.env.VITE_API_URL || '/api',
  timeout: 15000,
});

api.interceptors.request.use((config) => { const
  token = localStorage.getItem('token');
  if (token) config.headers.Authorization = `Bearer ${token}`;
  return config;
});

api.interceptors.response.use(
  (r) => r,
  (err) => {
    if (err.response?.status === 401)
      { localStorage.removeItem('token');
        localStorage.removeItem('user');
        window.location.href = '/login';
      }
    return Promise.reject(err);
  }
);
```

		АНТОНЮК А.Р.				
--	--	--------------	--	--	--	--

);

export default api;

Лістинг Г.4 — Файл src/components/Layout.jsx (основний макет з меню)

```
import React, { useState } from 'react';
import { Outlet, useNavigate, useLocation } from 'react-router-dom';
import { Layout, Menu, Avatar, Dropdown, Badge, Typography, Space, theme } from 'antd';
import {
  DashboardOutlined, CloudServerOutlined, NodeIndexOutlined, ContainerOutlined,
  LineChartOutlined, FileTextOutlined, BellOutlined, ApartmentOutlined,
  TeamOutlined, LogoutOutlined, UserOutlined, MenuFoldOutlined, MenuUnfoldOutlined,
} from '@ant-design/icons';
import { useDispatch, useSelector } from 'react-redux';
import { logout } from '../store/authSlice';

const { Header, Sider, Content } = Layout; const
{ Text } = Typography;

const menuItems = [
  { key: '/dashboard', icon: <DashboardOutlined />, label: 'Дашборд' },
  { key: '/infrastructures', icon: <CloudServerOutlined />, label: 'Інфраструктури' },
  { key: '/nodes', icon: <NodeIndexOutlined />, label: 'Вузли' },
  { key: '/containers', icon: <ContainerOutlined />, label: 'Контейнери' },
  { key: '/monitoring', icon: <LineChartOutlined />, label: 'Моніторинг' },
  { key: '/logs', icon: <FileTextOutlined />, label: 'Журнал подій' },
  { key: '/alerts', icon: <BellOutlined />, label: 'Сповіщення' },
  { key: '/topology', icon: <ApartmentOutlined />, label: 'Топологія' },
  { key: '/users', icon: <TeamOutlined />, label: 'Користувачі', adminOnly: true },
];

export default function AppLayout() {
  const [collapsed, setCollapsed] = useState(false);
  const navigate = useNavigate();
  const location = useLocation();
  const dispatch = useDispatch();
  const { user } = useSelector(s => s.auth);
  const { token: { colorBgContainer, borderRadiusLG } } = theme.useToken();

  const visibleItems = menuItems
    .filter(item => !item.adminOnly || user?.role === 'admin')
    .map(({ key, icon, label }) => ({ key, icon, label }));

  const userMenu = {
    items: [{ key: 'logout', icon: <LogoutOutlined />, label: 'Вийти', danger: true }], onClick:
      ({ key }) => { if (key === 'logout') { dispatch(logout()); navigate('/login'); } },
  };

  return (
    <Layout style={{ minHeight: '100vh' }}>
      <Sider trigger={null} collapsible collapsed={collapsed} width={220}
        style={{ background: '#001529' }}>
        <div style={{ height: 64, display: 'flex', alignItems: 'center', justifyContent: 'center',
          color: '#fff', fontSize: collapsed ? 18 : 16, fontWeight: 700, borderBottom: '1px solid #142a4d' }}>
          {collapsed ? '▲' : '▲ HybridCloud'}
        </div>
        <Menu theme="dark" mode="inline" selectedKeys={[location.pathname]}>

```

```

      items={visibleItems} onClick={({ key }) => navigate(key)} />
    </Slider>
    <Layout>
      <Header style={{ padding: '0 24px', background: colorBgContainer,
        display: 'flex', alignItems: 'center', justifyContent: 'space-between',
        borderBottom: '1px solid #f0f0f0' }}>
        <Space>
          {React.createElement(collapsed ? MenuUnfoldOutlined : MenuFoldOutlined,
            { style: { fontSize: 18, cursor: 'pointer' },
              onClick: () => setCollapsed(!collapsed),
            })}
        </Space>
        <Dropdown menu={userMenu} placement="bottomRight">
          <Space style={{ cursor: 'pointer' }}>
            <Avatar icon={<UserOutlined />} style={{ backgroundColor: '#1677ff' }} />
            <Text strong>{user?.username}</Text>
            <Text type="secondary" style={{ fontSize: 12 }}>{user?.role}</Text>
          </Space>
        </Dropdown>
      </Header>
      <Content style={{ margin: 24, padding: 24, background: colorBgContainer, borderRadius:
        borderRadiusLG, minHeight: 'calc(100vh - 112px)', overflow: 'auto' }}>
        <Outlet />
      </Content>
    </Layout>
  </Layout>
);
}

```

Лістинг Г.5 — Файл src/pages/LoginPage.jsx (сторінка авторизації)

```

import React, { useState } from 'react';
import { Form, Input, Button, Card, Typography, Alert, Space } from 'antd';
import { UserOutlined, LockOutlined } from '@ant-design/icons';
import { useDispatch } from 'react-redux';
import { useNavigate, Link } from 'react-router-dom';
import { setCredentials } from '../store/authSlice';
import api from '../api';

```

```

const { Title, Text } = Typography;

```

```

export default function LoginPage() {
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");
  const dispatch = useDispatch();
  const navigate = useNavigate();

  const onFinish = async (values) => {
    { setLoading(true); setError("");
    try {
      const { data } = await api.post('/auth/login', values);
      dispatch(setCredentials(data)); navigate('/dashboard');
    } catch (err) {
      setError(err.response?.data?.error || 'Помилка входу');
    } finally { setLoading(false); }
    };
  };

```

```

return (
  <div style={{ minHeight: '100vh', display: 'flex', alignItems: 'center', justifyContent: 'center',
    background: 'linear-gradient(135deg, #001529 0%, #003a8c 100%)' }}>
    <Card style={{ width: 400, borderRadius: 12, boxShadow: '0 8px 32px rgba(0,0,0,0.3)' }}>
      <Space direction="vertical" size={4} style={{ width: '100%', marginBottom: 24, textAlign: 'center' }}>
        <Title level={2} style={{ margin: 0 }}> HybridCloud </Title>
        <Text type="secondary">Оркестрація хмарних інфраструктур</Text>
      </Space>

      {error && <Alert type="error" message={error} showIcon style={{ marginBottom: 16 }} />}

      <Form layout="vertical" onFinish={onFinish}>
        <Form.Item name="email" rules={[{ required: true, type: 'email', message: 'Введіть email' }]}>
          <Input prefix={<UserOutlined />} placeholder="Email" size="large" />
        </Form.Item>
        <Form.Item name="password" rules={[{ required: true, message: 'Введіть пароль' }]}>
          <Input.Password prefix={<LockOutlined />} placeholder="Пароль" size="large" />
        </Form.Item>
        <Button type="primary" htmlType="submit" block size="large" loading={loading}> Увійти
        </Button>
      </Form>
      <div style={{ textAlign: 'center', marginTop: 16 }}>
        <Text type="secondary">Немає акаунту? </Text>
        <Link to="/register">Зареєструватись</Link>
      </div>
      <Alert type="info" style={{ marginTop: 16 }} showIcon
        message="Demo: admin@hybrid-cloud.local / admin123" />
    </Card>
  </div>
);
}

```

Лістинг Г.6 — Файл src/pages/DashboardPage.jsx (головний дашборд)

```

import React, { useEffect, useState } from 'react';
import { Row, Col, Card, Statistic, Typography, Table, Badge, Tag, Space, Spin } from 'antd';
import { CloudServerOutlined, NodeIndexOutlined, ContainerOutlined, BellOutlined, CheckCircleOutlined,
  WarningOutlined } from '@ant-design/icons';
import { LineChart, Line, XAxis, YAxis, Tooltip, ResponsiveContainer, CartesianGrid, Legend } from 'recharts'; import api
from '../api';
import { io } from 'socket.io-client'; const

{ Title } = Typography;

export default function DashboardPage()
{ const [stats, setStats] = useState(null);
  const [metrics, setMetrics] = useState([]);
  const [loading, setLoading] = useState(true);

  useEffect(() => {
    const load = async () =>
    { try {
      const [statsRes, metricsRes] = await Promise.all([api.get('/stats'), api.get('/metrics/latest')]); setStats(statsRes.data);
      setMetrics(metricsRes.data);
    }

```

		Вакалюк Т.А.			ІІЗ.КР.Б – 121 – 26 – ІІЗ	90
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    } catch {} finally { setLoading(false); }
  };
  load();

  const wsUrl = import.meta.env.VITE_WS_URL || window.location.origin.replace(/:\d+/, ':5000');
  const socket = io(wsUrl);
  socket.on('metrics:update', ({ nodeId, metrics: m }) => {
    setMetrics(prev => prev.map(n => n.nodeId === nodeId ? { ...n, ...m } : n));
  });
  return () => socket.disconnect();
}, []);

if (loading) return <div style={{ textAlign: 'center', padding: 80 }}><Spin size="large" /></div>; const

kpis = [
  { title: 'Інфраструктура', value: stats?.totalInfra ?? 0, icon: <CloudServerOutlined style={{ color: '#1677ff' }} />, color:
'#e6f4ff' },
  { title: 'Активні вузли', value: `${stats?.onlineNodes ?? 0} / ${stats?.totalNodes ?? 0}`, icon: <NodeIndexOutlined
style={{ color: '#52c41a' }} />, color: '#f6ffed' },
  { title: 'Контейнери', value: stats?.runningContainers ?? 0, icon: <ContainerOutlined style={{ color: '#722ed1' }}
/>, color: '#f9f0ff' },
  { title: 'Активні сповіщення', value: stats?.activeAlerts ?? 0, icon: <BellOutlined style={{ color: '#fa8c16' }} />,
color: '#fff7e6' },
];

const cols = [
  { title: 'Вузол', dataIndex: 'nodeName', key: 'nodeName' },
  { title: 'CPU', key: 'cpu', render: (_, r) => {
    const v = r.cpu?.toFixed(1) ?? '--';
    const color = r.cpu > 80 ? 'red' : r.cpu > 60 ? 'orange' : 'green';
    return <Tag color={color}>{v}%</Tag>;
  }},
  { title: 'RAM', key: 'ram', render: (_, r) => {
    const v = r.ram?.toFixed(1) ?? '--';
    const color = r.ram > 85 ? 'red' : r.ram > 70 ? 'orange' : 'blue';
    return <Tag color={color}>{v}%</Tag>;
  }},
  { title: 'Диск', key: 'disk', render: (_, r) => {
    const v = r.disk?.toFixed(1) ?? '--';
    return <Tag color={r.disk > 85 ? 'red' : 'default'}>{v}%</Tag>;
  }},
  { title: 'Статус', key: 'status', render: (_, r) =>
    { const ok = (r.cpu ?? 0) < 80 && (r.ram ?? 0) <
      85;
      return ok ? <Badge status="success" text="OK" /> : <Badge status="warning" text="Увага" />;
    }},
];

return (
  <div>
    <Title level={3} style={{ marginBottom: 24 }}>Головний дашборд</Title>
    <Row gutter={[16, 16]} style={{ marginBottom: 24 }}>
      {kpis.map((k, i) => (
        <Col xs={24} sm={12} xl={6} key={i}>
          <Card style={{ background: k.color, border: 'none' }}>
            <Statistic title={<Space>{k.icon}{k.title}</Space>} value={k.value} />
          </Card>
        </Col>
      ))}
    </div>
  )

```

		АНТОНЮК А.Р.				
--	--	--------------	--	--	--	--

```

    ))}
  </Row>
  <Row gutter={[16, 16]}>
    <Col xs={24} lg={14}>
      <Card title="Поточні метрики вузлів" size="small">
        <Table dataSource={metrics} columns={cols} rowKey="nodeId" pagination={false} size="small" />
      </Card>
    </Col>
    <Col xs={24} lg={10}>
      <Card title="CPU (реальний час)" size="small">
        <ResponsiveContainer width="100%" height={220}>
          <LineChart data={metrics.map(m => ({ name: m.nodeName, CPU: parseFloat(m.cpu?.toFixed(1) ?? 0) })))>
            <CartesianGrid strokeDasharray="3 3" />
            <XAxis dataKey="name" tick={{ fontSize: 11 }} />
            <YAxis domain={[0, 100]} unit="%" tick={{ fontSize: 11 }} />
            <Tooltip formatter={(v) => [`${v}%`, 'CPU']} />
            <Line type="monotone" dataKey="CPU" stroke="#1677ff" strokeWidth={2} dot={{ r: 4 }} />
          </LineChart>
        </ResponsiveContainer>
      </Card>
    </Col>
  </Row>
</div>
);
}

```

Лістинг Г.7 — Файл src/pages/ContainersPage.jsx (управління контейнерами)

```

import React, { useEffect, useState } from 'react';
import {
  Table, Button, Modal, Form, Input, Select, Card, Space, Popconfirm,
  Typography, Badge, message, Tooltip, Drawer, Tag
} from 'antd'; import
{
  PlusOutlined, PlayCircleOutlined, PauseCircleOutlined,
  ReloadOutlined, DeleteOutlined, FileTextOutlined
} from '@ant-design/icons';
import api from '../api';
import { useSelector } from 'react-redux';

const { Title, Text } = Typography;
const STATUS_BADGE = { running: 'success', stopped: 'default', error: 'error', removed: 'default' };

export default function ContainersPage()
{
  const [data, setData] = useState([]);
  const [nodes, setNodes] = useState([]);
  const [loading, setLoading] = useState(false);
  const [deployOpen, setDeployOpen] = useState(false);
  const [deploying, setDeploying] = useState(false);
  const [logsDrawer, setLogsDrawer] = useState({ open: false, name: '', text: '' });
  const [form] = Form.useForm();
  const { user } = useSelector(s => s.auth);
  const canEdit = ['admin', 'operator'].includes(user?.role);

  const load = async () =>
  {
    setLoading(true);
    try { const r = await api.get('/containers'); setData(r.data); }
  }

```

```

    finally { setLoading(false); }
  };

useEffect(() =>
  { load();
    api.get('/nodes').then(r => setNodes(r.data));
  }, []);

const doAction = async (id, action, name) => { try {
  await api.post(`/containers/${id}/action`, { action });
  message.success(`${name}: ${action} виконано`); load();
} catch (e) { message.error(e.response?.data?.error || 'Помилка'); }
};

const showLogs = async (container) => {
  setLogsDrawer({ open: true, name: container.name, text: 'Завантаження...' }); try {
    const r = await api.get(`/containers/${container.id}/logs`);
    setLogsDrawer(prev => ({ ...prev, text: r.data.logs || '(порожньо)' }));
  } catch { setLogsDrawer(prev => ({ ...prev, text: 'Помилка отримання логів' })); }
};

const onDeploy = async (values) =>
  { setDeploying(true);
    try {
      const ports = values.ports
        ? values.ports.split("\n").map(p => p.trim()).filter(Boolean)
        : [];
      const env = values.env
        ? values.env.split("\n").map(e =>
            { const idx = e.indexOf('=');
              if (idx < 0) return null;
              return { key: e.slice(0, idx).trim(), value: e.slice(idx + 1).trim() };
            }).filter(Boolean)
        : [];
      await api.post('/containers/deploy', { ...values, ports, env });
      message.success('Контейнер розгорнуто'); setDeployOpen(false);
      form.resetFields(); load();
    } catch (e) {
      message.error(e.response?.data?.error || 'Помилка розгортання');
    } finally { setDeploying(false); }
  };

const cols = [
  { title: 'ID', dataIndex: 'containerId', key: 'cid', width: 110, render: v => <code style={{ fontSize: 11 }}>{v}</code> },
  { title: 'Назва', dataIndex: 'name', key: 'name' },
  { title: 'Образ', dataIndex: 'image', key: 'image', render: v => <code style={{ fontSize: 11 }}>{v}</code> },
  { title: 'Вузол', key: 'node', render: (_, r) => r.node?.name || '—' },
  {
    title: 'Статус', dataIndex: 'status', key: 'status', width: 110,
    render: v => <Badge status={STATUS_BADGE[v] || 'default'} text={v} />,
  },
];

```

```

title: 'Оновлено', dataIndex: 'updatedAt', key: 'upd', width: 150, render:
v => new Date(v).toLocaleString('uk'),
},
{
title: 'Дії', key: 'actions', width: 200,
render: (_, r) => (
<Space size={4}>
<Tooltip title="Логи"><Button size="small" icon={<FileTextOutlined />} onClick={() => showLogs(r)}
/></Tooltip>
{r.status !== 'running' && (
<Tooltip title="Запустити">
<Button size="small" type="primary" ghost icon={<PlayCircleOutlined />}
onClick={() => doAction(r.id, 'start', r.name)} />
</Tooltip>
)}
{r.status === 'running' && (
<Tooltip title="Зупинити">
<Button size="small" icon={<PauseCircleOutlined />}
onClick={() => doAction(r.id, 'stop', r.name)} />
</Tooltip>
)}
<Tooltip title="Перезапустити">
<Button size="small" icon={<ReloadOutlined />}
onClick={() => doAction(r.id, 'restart', r.name)} />
</Tooltip>
<Popconfirm title="Видалити контейнер?" onConfirm={() => doAction(r.id, 'remove', r.name)}>
<Tooltip title="Видалити"><Button size="small" danger icon={<DeleteOutlined />} /></Tooltip>
</Popconfirm>
</Space>
),
},
];

const onlineNodes = nodes.filter(n => n.status === 'online');

return (
<div>
<div style={{ display: 'flex', justifyContent: 'space-between', alignItems: 'center', marginBottom: 16 }}>
<Title level={3} style={{ margin: 0 }}>Контейнери</Title>
{canEdit && (
<Button type="primary" icon={<PlusOutlined />} onClick={() => setDeployOpen(true)}>
Розгорнути контейнер
</Button>
)}
</div>

<Card>
<Table
dataSource={data} columns={cols} rowKey="id"
loading={loading} size="small"
rowClassName={r => r.status === 'error' ? 'ant-table-row-error' : ''}
/>
</Card>

{/* Deploy Modal */}
<Modal
title="Розгорнути новий контейнер"
open={deployOpen}

```

```

onCancel={() => { setDeployOpen(false); form.resetFields(); }}
onOk={() => form.submit()}
okText="Розгорнути"
confirmLoading={deploying}
width={540}
>
<Form form={form} layout="vertical" onFinish={onDeploy}>
  <Form.Item name="nodeId" label="Вузол розгортання" rules={[{ required: true, message: 'Виберіть вузол' }]}>
    <Select
      placeholder={onlineNodes.length ? 'Виберіть вузол' : 'Немає активних вузлів'} options={onlineNodes.map(n =>
        ({ value: n.id, label: `${n.name} (${n.host})` }) )}>
    </Form.Item>
    <Form.Item name="image" label="Docker образ" rules={[{ required: true, message: 'Введіть образ' }]}>
      <Input placeholder="nginx:latest або node:18-alpine" />
    </Form.Item>
    <Form.Item name="name" label="Назва контейнера (необов'язково)">
      <Input placeholder="my-web-server" />
    </Form.Item>
    <Form.Item name="ports" label="Прив'язка портів (один на рядок: хост:контейнер)">
      <Input.TextArea rows={2} placeholder={"8080:80\n3000:3000"} style={{ fontFamily: 'monospace' }} />
    </Form.Item>
    <Form.Item name="env" label="Змінні середовища (один на рядок: KEY=value)">
      <Input.TextArea rows={3} placeholder={"NODE_ENV=production\nPORT=3000"} style={{ fontFamily: 'monospace' }} />
    </Form.Item>
    <Form.Item name="cmd" label="Команда запуску (необов'язково)">
      <Input placeholder="node server.js" style={{ fontFamily: 'monospace' }} />
    </Form.Item>
  </Form>
</Modal>

{/* Logs Drawer */}
<Drawer
  title={`Лог: ${logsDrawer.name}`}
  open={logsDrawer.open}
  onClose={() => setLogsDrawer({ open: false, name: "", text: "" })}
  width={680}
  styles={{ body: { padding: 0 } }}
>
  <pre style={{
    margin: 0, padding: 16,
    background: '#1e1e1e', color: '#d4d4d4',
    minHeight: '100%', fontSize: 12,

    fontFamily: 'monospace', whiteSpace: 'pre-wrap',
    wordBreak: 'break-all',
  }}>
    {logsDrawer.text}
  </pre>
</Drawer>
</div>

```

Лістинг Г.8 — Файл src/pages/MonitoringPage.jsx (моніторинг з графіками)

```

import React, { useEffect, useState } from 'react';
import { Card, Select, Row, Col, Typography, Spin, Empty, Radio, Space, Tag } from 'antd';

```

		Вакалюк Т.А.			ІІЗ.КР.Б – 121 – 26 – ІІЗ	95
Змін.	Арх.	№ докум.	Підпис	Дата		

```

import { LineChart, Line, XAxis, YAxis, Tooltip, CartesianGrid, ResponsiveContainer, Legend } from 'recharts'; import
{ io } from 'socket.io-client';
import api from './api';

const { Title } = Typography;
const COLORS = ['#1677ff', '#52c41a', '#fa8c16', '#722ed1', '#f5222d']; const
RANGES = ['1h', '24h', '7d', '30d'];

export default function MonitoringPage() { const
[nodes, setNodes] = useState([]);
const [selectedNodes, setSelectedNodes] = useState([]);
const [metricsMap, setMetricsMap] = useState({});
const [range, setRange] = useState('24h');
const [loading, setLoading] = useState(false);

useEffect(() => { api.get('/nodes').then(r => { setNodes(r.data); if (r.data.length) setSelectedNodes([r.data.id]); }); }, []);

useEffect(() => {
if (!selectedNodes.length) return;
setLoading(true);
Promise.all(selectedNodes.map(id => api.get(`/nodes/${id}/metrics`, { params: { range } })).then(r => ({ id, data: r.data }))))
.then(results => { const
map = {};
results.forEach(({ id, data }) => { map[id] = data; }); setMetricsMap(map);
})
.finally(() => setLoading(false));
}, [selectedNodes, range]);

useEffect(() => {
const wsUrl = import.meta.env.VITE_WS_URL || window.location.origin.replace(/:\d+/, ':5000');
const socket = io(wsUrl);
socket.on('metrics:update', ({ nodeId, metrics, timestamp }) => { if
(selectedNodes.includes(nodeId)) {
setMetricsMap(prev => ({ ...prev, [nodeId]: [...(prev[nodeId] || []).slice(-99), { ...metrics, recordedAt:
timestamp } ] }));
}
});
return () => socket.disconnect();
}, [selectedNodes]);

const buildChartData = (metric) => {
const allTimestamps = [...new Set(selectedNodes.flatMap(id => (metricsMap[id] || []).map(m =>
m.recordedAt)))]);
return allTimestamps.slice(-50).map(ts => {
const point = { time: new Date(ts).toLocaleTimeString('uk') };
selectedNodes.forEach(id => {
const m = metricsMap[id]?.find(r => r.recordedAt === ts); const
node = nodes.find(n => n.id === id);
if (m && node) point[node.name] = parseFloat(m[metric]?.toFixed(1) ?? 0);
});
return point;
});
};

```

		АНТОНЮК А.Р.				
--	--	--------------	--	--	--	--

```

return (
  <div>
    <div style={{ display: 'flex', justifyContent: 'space-between', alignItems: 'center', marginBottom: 16 }}>
      <Title level={3} style={{ margin: 0 }}>Моніторинг ресурсів</Title>
      <Space>
        <Radio.Group value={range} onChange={e => setRange(e.target.value)} buttonStyle="solid">
          {RANGES.map(r => <Radio.Button key={r} value={r}>{r}</Radio.Button>)}
        </Radio.Group>
      </Space>
    </div>
    <Card style={{ marginBottom: 16 }}>
      <Select mode="multiple" style={{ width: '100%' }} placeholder="Виберіть вузли для порівняння (до 5)"
        value={selectedNodes} onChange={v => setSelectedNodes(v.slice(0, 5))}
        options={nodes.map(n => ({ value: n.id, label: n.name }))) />
    </Card>
    {loading ? <Spin size="large" style={{ display: 'block', margin: '40px auto' }} /> : (
      <Row gutter={[16, 16]}>
        {[ 'cpu', 'ram', 'disk' ].map((metric, mi) =>
          { const chartData = buildChartData(metric);
            const labels = { cpu: 'CPU (%)', ram: 'RAM (%)', disk: 'Диск (%)' };
            return (
              <Col xs={24} key={metric}>
                <Card title={labels[metric]} size="small">
                  {chartData.length ? (
                    <ResponsiveContainer width="100%" height={220}>
                      <LineChart data={chartData}>
                        <CartesianGrid strokeDasharray="3 3" />
                        <XAxis dataKey="time" tick={{ fontSize: 10 }} interval="preserveStartEnd" />
                        <YAxis domain={[0, 100]} unit="%" tick={{ fontSize: 11 }} />
                        <Tooltip formatter={(v) => [`${v}%`] />
                        <Legend />
                        {selectedNodes.map((id, i) => {
                          const node = nodes.find(n => n.id === id);
                          return node ? <Line key={id} type="monotone" dataKey={node.name} stroke={COLORS[i %
COLORS.length]} strokeWidth={2} dot={false} /> : null;
                        })}
                      </LineChart>
                    </ResponsiveContainer>
                  ) : <Empty description="Немає даних" />
                </Card>
              </Col>
            );
          })}
      </Row>
    );
  </div>
);
}

```

Лістинг Г.9 — Файл src/pages/TopologyPage.jsx (візуалізація топології)

```

import React, { useEffect, useState, useCallback } from 'react';
import { Card, Typography, Select, Empty, Badge, Tooltip } from 'antd';
import ReactFlow, { Background, Controls, MiniMap, useNodesState, useEdgesState } from 'reactflow'; import
'reactflow/dist/style.css';
import api from '../api';

```

```

const { Title } = Typography;
const STATUS_COLORS = { online: '#52c41a', offline: '#ff4d4f', warning: '#faad14' };

```

```

const NodeCard = ({ data }) => (
  <div style={{ padding: '8px 12px', background: '#fff', border: '2px solid ${data.color}', borderRadius: 8, minWidth: 120,
    boxShadow: '0 2px 8px rgba(0,0,0,0.1)' }}>
    <div style={{ fontWeight: 700, fontSize: 12 }}>{data.label}</div>
    <div style={{ fontSize: 11, color: '#666' }}>{data.host}</div>
    <div style={{ marginTop: 4 }}><Badge color={data.color} text={<span style={{ fontSize:
    10 }}>{data.status}</span>} /></div>
  </div>
);

const InfraCard = ({ data }) => (
  <div style={{ padding: '8px 16px', background: '#001529', borderRadius: 8, color: '#fff', fontWeight: 700 }}>
    ▲ {data.label}
  </div>
);

const nodeTypes = { nodeCard: NodeCard, infraCard: InfraCard };

export default function TopologyPage() {
  const [infras, setInfras] = useState([]);
  const [selectedInfra, setSelectedInfra] = useState(null);
  const [nodes, setNodes, onNodesChange] = useNodesState([]);
  const [edges, setEdges, onEdgesChange] = useEdgesState([]);

  useEffect(() => { api.get('/infrastructures').then(r => { setInfras(r.data); if (r.data.length)
    setSelectedInfra(r.data.id); }); }, []);

  useEffect(() => {
    if (!selectedInfra) return;
    api.get(`/infrastructures/${selectedInfra}`).then(r => { const
    infra = r.data;
    const newNodes = [
      { id: `infra-${infra.id}`, type: 'infraCard', position: { x: 300, y: 20 }, data: { label: infra.name } },
    ];
    const newEdges = [];
    (infra.nodes || []).forEach((node, i) =>
      { const col = i % 4;
        const row = Math.floor(i / 4);
        newNodes.push({
          id: `node-${node.id}`, type:
          'nodeCard',
          position: { x: col * 180 + 50, y: row * 140 + 120 },
          data: { label: node.name, host: node.host, status: node.status, color: STATUS_COLORS[node.status] || '#ccc' },
        });
        newEdges.push({ id: `e-${infra.id}-${node.id}`, source: `infra-${infra.id}`, target: `node-${node.id}`, type: 'smoothstep',
          animated: node.status === 'online' });

        // Container connections
        (node.containers || []).filter(c => c.status === 'running').forEach((c, ci) =>
          { newNodes.push({
            id: `c-${c.id}`,
            position: { x: col * 180 + 50 + ci * 20, y: row * 140 + 260 },
            data: { label: c.name },
            style: { background: '#f0f0ff', border: '1px solid #9254de', borderRadius: 4, padding: '2px 8px', fontSize: 11 },
          });

```

		Вакалюк Т.А.			ІІЗ.КР.Б – 121 – 26 – ІІЗ	98
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    newEdges.push({ id: `ec-${node.id}-${c.id}`, source: `node-${node.id}`, target: `c-${c.id}`, type:
'smoothstep', style: { stroke: '#9254de', strokeDasharray: '4' } });
  });
});
setNodes(newNodes);
setEdges(newEdges);
});
}, [selectedInfra]);

return (
  <div>
    <div style={{ display: 'flex', justifyContent: 'space-between', alignItems: 'center', marginBottom: 16 }}>
      <Title level={3} style={{ margin: 0 }}>Топологія інфраструктури</Title>
      <Select style={{ width: 220 }} value={selectedInfra} onChange={setSelectedInfra}
        options={infras.map(i => ({ value: i.id, label: i.name }))) />
    </div>
    <Card bodyStyle={{ padding: 0 }}>
      <div style={{ height: 600 }}>
        {nodes.length ? (
          <ReactFlow nodes={nodes} edges={edges} onNodesChange={onNodesChange}
onEdgesChange={onEdgesChange}
          nodeTypes={nodeTypes} fitView>
            <Background />
            <Controls />
            <MiniMap nodeColor={n => n.data?.color || '#ccc'} />
          </ReactFlow>
        ) : <Empty description="Виберіть інфраструктуру" style={{ padding: 100 }} />}
      </div>
    </Card>
  </div>
);
}

```

		Антонюк А.Р.				
--	--	--------------	--	--	--	--

ДОДАТОК Д

Додаткові лістинги серверної частини (Node.js/Express)

Лістинг Д.1 — Файл src/sockets/socketHandlers.js (обробники подій

Socket.IO)

```
const jwt = require('jsonwebtoken');
const logger = require('../utils/logger');

/**
 * Ініціалізує обробники подій Socket.IO. Кожне з'єднання проходить
 * автентифікацію за тим самим JWT, що й REST API; неавтентифіковані
 * сокети негайно роз'єднуються. Клієнти приєднуються до «кімнат» за
 * вузлами, що дозволяє адресно транслявати метрики лише підписникам.
 */
const initSocketHandlers = (io) => {
  // Middleware автентифікації рукоштовування
  io.use((socket, next) => {
    const token = socket.handshake.auth && socket.handshake.auth.token;
    if (!token) return next(new Error('UNAUTHORIZED'));
    try {
      socket.user = jwt.verify(token, process.env.JWT_SECRET);
      return next();
    } catch (err) {
      return next(new Error('UNAUTHORIZED'));
    }
  });

  io.on('connection', (socket) => {
    logger.info(`Socket connected: ${socket.id} (user ${socket.user.id})`);

    // Підписка на оновлення конкретного вузла
    socket.on('subscribe:node', (nodeId) => {
      socket.join(`node:${nodeId}`);
    });

    socket.on('unsubscribe:node', (nodeId) => {
      socket.leave(`node:${nodeId}`);
    });

    socket.on('disconnect', (reason) => {
      logger.info(`Socket disconnected: ${socket.id} (${reason})`);
    });
  });

  logger.info('Обробники Socket.IO ініціалізовано');
};

module.exports = { initSocketHandlers };
```

Лістинг Д.2 — Файл src/config/swagger.js (специфікація OpenAPI)

```
const swaggerJSDoc = require('swagger-jsdoc');

/**
 * Конфігурація специфікації OpenAPI 3.0. Описи окремих ендпоінтів
 * винесено у JSDoc-анотації @openapi безпосередньо у файлах маршрутів,
```

		Вакалюк Т.А.			ІПЗ.КР.Б – 121 – 26 – ПЗ	100
Змін.	Арх.	№ докум.	Підпис	Дата		

```

* звідки swagger-jsdoc збирає їх під час старту сервера.
*/
const swaggerSpec =
  swaggerJSDoc({ definition: {
    openapi: '3.0.3',
    info: {
      title: 'Hybrid Cloud Orchestration API',
      version: '1.0.0',
      description:
        'REST API вебзастосунку для централізованого моніторингу та оркестрації
        + 'контейнеризованих навантажень у гібридній інфраструктурі.',
    },
    servers: [{ url: '/api', description: 'Базовий шлях API' }],
    components: {
      securitySchemes: {
        bearerAuth: { type: 'http', scheme: 'bearer', bearerFormat: 'JWT' },
      },
    },
    security: [{ bearerAuth: [] }],
  },
  apis: ['./src/routes/*.js'],
});

```

```
module.exports = { swaggerSpec };
```

Лістинг Д.3 — Файл `src/controllers/metricsController.js`

```

const { Metric, Node } = require('../models');
const { Op } = require('sequelize');

const RANGES = { '1h': 3600, '24h': 86400, '7d': 604800, '30d': 2592000 };

/** GET /api/nodes/:id/metrics?range=24h — часовий ряд метрик вузла. */
exports.getNodeMetrics = async (req, res) => {
  try {
    const { id } = req.params;
    const seconds = RANGES[req.query.range] || RANGES['24h'];
    const since = new Date(Date.now() - seconds * 1000);
    const metrics = await Metric.findAll({
      where: { nodeId: id, recordedAt: { [Op.gte]: since } },
      order: [['recordedAt', 'ASC']],
    });
    res.json(metrics);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

/** GET /api/metrics/latest — останній зріз метрик по всіх онлайн-вузлах. */
exports.getLatest = async (req, res) => {
  try {
    const nodes = await Node.findAll({ where: { status: 'online' } });
    const latest = [];
    for (const node of nodes) {
      const m = await Metric.findOne({
        where: { nodeId: node.id }, order: [['recordedAt', 'DESC']],
      });
      if (m) latest.push({ nodeId: node.id, name: node.name, ...m.toJSON() });
    }
    res.json(latest);
  } catch (err) {
  }
}

```

		Антонюк А.Р.				
--	--	--------------	--	--	--	--

		Вакалюк Т.А.			ІПЗ.КР.Б – 121 – 26 – ПЗ	101
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    res.status(500).json({ error: err.message });
  }
};

```

Лістинг Д.4 — Файл `src/controllers/logsController.js`

```

const { Log, Node } = require('../models');

/** GET /api/logs?level=&source= – журнал подій із фільтрацією. */
exports.list = async (req, res) => {
  try {
    const { level, source } = req.query;
    const where = {};
    if (level) where.level = level;
    if (source) where.source = source;
    const logs = await Log.findAll({
      where, order: [['createdAt', 'DESC']], limit: 500,
    });
    res.json(logs);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

/** GET /api/nodes/:id/logs – журнал подій конкретного вузла. */
exports.getNodeLogs = async (req, res) => {
  try {
    const { id } = req.params;
    const logs = await Log.findAll({
      where: { nodeId: id },
      include: [{ model: Node, as: 'node', attributes: ['name'] }],
      order: [['createdAt', 'DESC']], limit: 500,
    });
    res.json(logs);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

```

Лістинг Д.5 — Файл `src/controllers/userController.js`

```

const { User, Role, Log } = require('../models');

/** GET /api/users – перелік облікових записів (лише admin). */
exports.list = async (req, res) => {
  try {
    const users = await User.findAll({
      attributes: ['id', 'username', 'email', 'isBlocked', 'createdAt'],
      include: [{ model: Role, as: 'role', attributes: ['name'] }],
      order: [['createdAt', 'DESC']],
    });
    res.json(users);
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

/** PATCH /api/users/:id/role – зміна ролі користувача. */
exports.updateRole = async (req, res) => {
  try {
    const { id } = req.params;
    const { roleId } = req.body;
    const user = await User.findByPk(id);

```

		Антонюк А.Р.				
--	--	--------------	--	--	--	--

		Вакалюк Т.А.			ІПЗ.КР.Б – 121 – 26 – ПЗ	102
Змін.	Арх.	№ докум.	Підпис	Дата		

```

    if (!user) return res.status(404).json({ error: 'User not found' });
    await user.update({ roleId });
    await
      Log.create({ leve
        l: 'INFO',
        message: `Role of user ${user.username} changed by ${req.user.username}`,
        source: 'admin', userId: req.user.id,
      });
    res.json({ success: true });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

/** PATCH /api/users/:id/block – блокування/розблокування облікового запису. */
exports.toggleBlock = async (req, res) => {
  try {
    const user = await User.findByPk(req.params.id);
    if (!user) return res.status(404).json({ error: 'User not found' });
    await user.update({ isBlocked: !user.isBlocked });
    res.json({ success: true, isBlocked: user.isBlocked });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

/** DELETE /api/users/:id – видалення облікового запису. */
exports.remove = async (req, res) => {
  try {
    const user = await User.findByPk(req.params.id);
    if (!user) return res.status(404).json({ error: 'User not found' });
    await user.destroy();
    res.json({ success: true });
  } catch (err) {
    res.status(500).json({ error: err.message });
  }
};

```

		Вакалюк Т.А.			ІПЗ.КР.Б – 121 – 26 – ПЗ	103
Змін.	Арк.	№ докум.	Підпис	Дата		