

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

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО

В.о. завідувача кафедри
Олександр КОВАЛЬ

«__» _____ 2026 р.

Дипломна робота

**на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення**

**на тему: «Порівняльний аналіз Kubernetes та Docker Swarm для оркестрації
контейнеризованих застосунків у задачах цифровізації енергетичних систем»**

Виконав:

студент IV курсу, групи ТВ-23

Невмержицький Олександр Миколайович

(прізвище, ім'я, по батькові)

(підпис)

Керівник:

Асистент, к.т.н, Єрошкін Ю.М.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Рецензент:

Професор, д.т.н., проф., Волощук В.А.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Засвідчую, що у цій дипломній роботі
немає запозичень із праць інших авторів
без відповідних посилань.

Студент _____

(підпис)

Київ – 2026

**Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»**

Навчально-науковий інститут атомної та теплової енергетики

Кафедра інженерії програмного забезпечення в енергетиці

Рівень вищої освіти перший (бакалаврський)

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення
інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ

В.о. завідувача кафедри

_____ Олександр КОВАЛЬ

«___» _____ 2026р.

ЗАВДАННЯ

на дипломну роботу студенту

_____ Невмержицькому Олександру Миколайовичу

(прізвище, ім'я, по батькові)

1. Тема роботи: Порівняльний аналіз Kubernetes та Docker Swarm для оркестрації
контейнеризованих застосунків у задачах цифровізації енергетичних систем

керівник роботи Єрошкін Ю.М., к.т.н

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «28» травня 2026 р. № 1992-с

2. Строк подання студентом роботи «10» червня 2026 р.

3. Вихідні дані до роботи: Docker Swarm, Kubernetes, мова Python, мова Javascript,
бібліотека React, СУБД PostgreSQL, Prometheus, Grafana, локальний кластер kind

4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно
розробити): проаналізувати підходи до оркестрації, дослідити архітектурні
особливості та принципи функціонування Kubernetes і Docker Swarm, розгорнути
однакову систему в середовищах Kubernetes і Docker Swarm та провести
порівняльний аналіз, сформулювати рекомендації щодо доцільності застосування

5. Перелік ілюстративного матеріалу: діаграми системи, схема бази даних, схема та
результати експериментів,

6. Дата видачі завдання «31» жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	31.10.2025	Виконано
2	Дослідження предметної області	01.11.2025 – 31.11.2025	Виконано
3	Дослідження існуючих рішень	01.12.2025 – 31.12.2025	Виконано
4	Постановка вимог до проєктування системи	01.01.2026 – 31.01.2026	Виконано
5	Розробка програмного продукту	01.02.2026 – 03.05.2026	Виконано
6	Тестування	04.05.2026 – 10.05.2026	Виконано
7	Захист програмного продукту	11.05.2026 – 15.05.2026	Виконано
8	Оформлення дипломної роботи	18.05.2026 – 31.05.2026	Виконано
9	Передзахист	01.06.2026 – 05.06.2026	Виконано
10	Захист	15.06.2026 – 19.06.2026	Виконано

Студент

(підпис)

Олександр НЕВМЕРЖИЦЬКИЙ
(ім'я, ПРІЗВИЩЕ)

Керівник роботи

(підпис)

Юрій ЄРОШКІН
(ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 81 сторінок, 9 рисунків, 10 таблиць, 2 додатки та 21 посилань.

Метою роботи є здійснення практичного і теоретичного аналізу можливостей Kubernetes та Docker Swarm на основі прототипу мікросервісної системи, яка моделює задачу моніторингу або управління енергоспоживанням.

Для досягнення визначеної мети було виконано наступні кроки: опрацьовано теоретичні основи контейнеризації та оркестрації застосунків, проаналізовано архітектуру, функціональні механізми та особливості експлуатації платформ Kubernetes і Docker Swarm. Розроблено прототип мікросервісної платформи обліку електроенергії. Реалізовано локальний експериментальний стенд із використанням Multipass та kind для Kubernetes і Docker Swarm. Проведено експериментальне порівняння оркестраторів. Сформовано рекомендації щодо вибору платформи оркестрації.

Практичне значення одержаних результатів полягає в тому, що робота дозволить енергетичним підприємствам/ІТ-відділам оцінити оптимальний стек технологій для впровадження контейнеризованих рішень залежно від рівня складності, масштабування та доступної інфраструктури.

Ключові слова: контейнеризація, оркестрація контейнерів, Kubernetes, Docker Swarm, мікросервіси, FastAPI, платформа обліку електроенергії, моніторинг енергоспоживання, Prometheus, Grafana, PostgreSQL, NetworkPolicy, failover, rolling update, цифровізація енергетики, Docker.

ABSTRACT

Structure and scope of the thesis. The work contains 81 pages, 9 figures, 10 table, 2 appendices and 21 references.

The purpose of the work is to conduct a practical and theoretical analysis of the capabilities of Kubernetes and Docker Swarm based on a prototype microservices system that simulates the task of monitoring or managing energy consumption. To achieve this objective, the following steps were taken: the theoretical foundations of containerisation and application orchestration were examined; the architecture, functional mechanisms and operational characteristics of the Kubernetes and Docker Swarm platforms were analysed. A prototype microservice platform for electricity metering was developed. A local experimental testbed was implemented using Multipass and kind for Kubernetes and Docker Swarm. An experimental comparison of orchestrators was carried out. Recommendations were formulated regarding the choice of orchestration platform.

The practical value of the obtained results lies in the fact that the work will enable energy companies/IT departments to evaluate the optimal technology stack for implementing containerized solutions depending on the level of complexity, scalability, and available infrastructure.

Keywords: containerization, container orchestration, Kubernetes, Docker Swarm, microservices, FastAPI, electricity accounting platform, energy consumption monitoring, Prometheus, Grafana, PostgreSQL, NetworkPolicy, failover, rolling update, energy digitization, Docker.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.....	8
ВСТУП.....	9
1 ПОСТАНОВКА ЗАДАЧІ ПОРІВНЯЛЬНОГО АНАЛІЗУ KUBERNETES ТА DOCKER SWARM	11
1.1 Методи дослідження	11
1.2 Програмні засоби дослідження	12
1.3 Архітектурні рішення прототипу	14
Висновки до розділу 1.....	16
2 АНАЛІЗ, ДОСЛІДЖЕННЯ ІСНУЮЧИХ МЕТОДІВ	17
2.1 Загальний опис предметної області.....	17
2.2 Аналіз платформ оркестрації контейнерів	19
2.2.1 Kubernetes: архітектура та можливості	19
2.2.2 Docker Swarm: архітектура та можливості	20
2.2.3 Альтернативні платформи оркестрації	21
2.3 Порівняльний аналіз для енергетичного сектору	22
2.3.1 Масштабованість та продуктивність	22
2.3.2 Безпека та відмовостійкість.....	24
2.4 Практичні кейси впровадження	25
2.4.1 Energyworx: міграція на Kubernetes	25
2.4.2 Sunrock: чиста енергетична майбутність з Kubernetes	26
Висновки до розділу 2.....	26
3 ЗАСОБИ РОЗРОБКИ СИСТЕМИ.....	28
3.1 Середовище розробки PyCharm.....	29
3.2 Мова програмування Python та FastAPI.....	30
3.3 Програмні засоби для роботи з базою даних.....	31
3.4 Засоби контейнеризації та оркестрації системи.....	32
3.5 Інструменти побудови експериментального середовища.....	33
3.6 Засоби моніторингу та аналізу показників системи	34
Висновки до розділу 3.....	35
4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ	36

4.1 Архітектура програмного забезпечення	36
4.2 Опис програмних компонентів стенда	38
4.2.1 Логічна модель системи та потік даних	40
4.3 Модель бази даних і міграції схеми	42
4.4 Платформи розгортання та їх роль у дослідженні	44
4.4.1 Підсистема моніторингу та збір метрик	46
4.4.2 Експериментальний контур порівняння Kubernetes та Docker Swarm	47
Висновки до розділу 4	48
5 ПОРІВНЯННЯ ПЛАТФОРМ ОРКЕСТРАЦІЇ КОНТЕЙНЕРІВ	49
5.1 Опис експериментів	49
5.1.1 Експериментальний стенд і умови проведення	50
5.1.2 Протокол реєстрації результатів	52
5.1.3 Опис сценаріїв експериментів	54
5.2 Результати експериментального дослідження	58
5.2.1 Обробка даних	58
5.2.2 Результати симетричних сценаріїв	59
5.3.3 Результати масштабування під навантаженням	60
5.3 Порівняльна характеристика платформ оркестрації контейнерів	64
5.3.1 Функціональне порівняння	65
5.3.2 Порівняння у контексті цифровізації енергетики	67
5.4 Рекомендації щодо вибору платформи оркестрації для задач цифровізації ..	68
5.4.1 Методичні підстави рекомендацій	68
5.4.2 SCADA-системи	69
5.4.3 Облік енергоресурсів	71
5.4.4 Попереджувальний аналіз та системи на основі ML	72
5.4.5 Інші типові задачі	73
Висновки до розділу 5	77
ВИСНОВКИ	78
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	80
ДОДАТОК А Фрагмент лістингу програми	82
ДОДАТОК Б Презентація	86

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

HPA	Horizontal Pod Autoscaler (механізм автоматичного горизонтального масштабування).
API	Application Programming Interface (програмний інтерфейс застосунку).
AMR	Automated Meter Reading (автоматизоване зчитування показань лічильників).
B2B	Business-to-Business (взаємодія між компаніями).
B2C	Business-to-Consumer (взаємодія бізнесу зі споживачем).
GPU	Graphics Processing Unit (графічний процесор).
IEC	International Electrotechnical Commission (Міжнародна електротехнічна комісія).
IT	Information Technology (інформаційні технології).
MDM	Meter Data Management (система керування даними обліку).
MVP	Minimum Viable Product (мінімально життєздатний продукт).
OT	Operational Technology (операційні технології).
SCADA	Supervisory Control and Data Acquisition (диспетчерське керування та збір даних).

ВСТУП

Розвиток цифрових платформ в енергетичних системах є одним із ключових напрямків еволюції енергетичного сектору на сьогодні. Зі зростанням популярності інтелектуальних пристроїв та систем моніторингу й управління розгортання надійної IT-інфраструктури стає критично важливим, і контейнеризація пропонує значні переваги в цьому плані, гарантуючи переносимість, ізолюваність та швидке оновлення сервісів. Таким чином, оркестрування контейнеризованих додатків ставить перед енергетичними системами унікальні виклики через важливість їх стабільної та безпечної роботи.

Один з викликів, пов'язаних з оркеструванням контейнеризованих сервісів, полягає у підтримці стабільної роботи великої кількості додатків, що виконують збір, обробку та аналіз даних в енергетичних системах. Інструменти оркестрування контейнерів, такі як Kubernetes та Docker Swarm, стають вирішальними в цій ситуації, забезпечуючи автоматизацію та гарантуючи високу доступність додатків завдяки їхньому ефективному розгортанню та масштабуванню, а також автоматичному відновленню сервісів у разі збою.

Оркестрування контейнерів стає невід'ємним елементом цифрових енергетичних систем, на якому базуються такі рішення, як системи моніторингу енергоспоживання, диспетчерські платформи, служби прогнозування навантаження, обробка телеметричних даних та управління розподіленими енергетичними ресурсами. Завдяки оркеструванню мікросервісів стає можливою адаптованість до мінливих вимог щодо робочого навантаження та підвищення надійності послуг.

Тим не менш, з оркеструванням контейнерів пов'язано кілька викликів. Хоча Kubernetes вирізняється своєю гнучкістю, масштабованістю та досконалими екосистемами, його впровадження виявляється складнішим порівняно з іншими фреймворками оркестрування. Водночас Docker Swarm демонструє відносно

простий процес налаштування та злагоджену взаємодію з додатками Docker, пропонуючи при цьому менше можливостей щодо масштабування та управління порівняно з іншими платформами.

З огляду на всі ці фактори, вибір між Kubernetes та Docker Swarm у цифрових енергетичних системах стає актуальним. Для енергетичних систем ключовими питаннями є продуктивність, безпека, надійність та зручність обслуговування цифрових послуг серед інших елементів екосистеми. Тому, обговорюючи питання цифрових енергетичних систем, слід звертати увагу на їхні специфічні вимоги, і вибір повинен їх відображати.

У цифрових енергетичних системах продуктивність, здатність до обробки даних у реальному часі та кібербезпека відіграють вирішальну роль поряд із надійністю системи. Оркестратори контейнерів повинні керувати сервісами, забезпечувати відновлення після збою, ізолювати компоненти, пропонувати масштабоване управління та дозволяти оновлення сервісів без переривання роботи системи. Отже, доречно порівняти Kubernetes та Docker Swarm у цьому контексті, щоб обрати найкращий підхід.

Отже, вивчення особливостей Kubernetes та Docker Swarm в оркеструванні контейнеризованих додатків для цифровізації енергетичних систем дозволяє зробити висновки щодо їхніх сильних і слабких сторін, а також придатності для сучасних енергетичних інфраструктур. Таке порівняння може слугувати основою для вибору правильного інструменту оркестрування контейнерів залежно від певних факторів.

1 ПОСТАНОВКА ЗАДАЧІ ПОРІВНЯЛЬНОГО АНАЛІЗУ KUBERNETES ТА DOCKER SWARM

З метою досягнення мети та проведення практичного й теоретичного дослідження властивостей та особливостей Kubernetes і Docker Swarm у контексті реалізації прототипу мікросервісної архітектури для цифровізації енергетичних систем, рекомендується використовувати комплекс методологій, програмного забезпечення та інструментів проектування, що забезпечить об'єктивність порівняльного дослідження та практичну цінність отриманих результатів.

1.1 Методи дослідження

В якості основного методу дослідження було обрано порівняльний аналіз, оскільки він дає змогу оцінити сильні та слабкі сторони двох оркестраторів контейнерів за однакових умов експлуатації. Цей метод передбачає аналіз архітектурних особливостей Kubernetes та Docker Swarm, принципів управління контейнеризованими додатками, а також підходів до масштабування, розподілу навантаження, забезпечення відмовостійкості, мережевої взаємодії та безпеки. Порівняльний аналіз є доречним, оскільки тема даної роботи безпосередньо зосереджена на виборі оптимального технологічного рішення для енергетичних компаній, де стабільність, гнучкість розгортання та ефективність використання ресурсів мають вирішальне значення.

Додатково використовується метод експериментального дослідження, який полягає у розгортанні однакової мікросервісної системи в середовищах Kubernetes та Docker Swarm і подальшому проведенні серії контрольованих випробувань. Такий підхід дозволяє отримати не лише теоретичні, а й практичні результати щодо поведінки обох оркестраторів у типових сценаріях цифровізації енергетичних систем. До таких сценаріїв належать масштабування сервісів при зростанні

навантаження, обробка збоїв вузлів, повторне розгортання після оновлення, оцінка затримок запуску сервісів, а також вимірювання споживання процесорних ресурсів і оперативної пам'яті.

Для збору та інтерпретації результатів доцільно застосувати методи моніторингу, статистичного узагальнення та візуального аналізу. Їх використання дає змогу фіксувати ключові технічні показники, порівнювати їх між собою та формувати обґрунтовані висновки щодо придатності кожного рішення до використання в енергетичній галузі. Отримані експериментальні дані мають бути подані у вигляді таблиць, графіків і узагальнених порівняльних характеристик.

1.2 Програмні засоби дослідження

Основним засобом для контейнеризації програмних компонентів стане Docker. Вибір цієї платформи пояснюється тим, що Docker є де-факто стандартом для розробки та запуску контейнерних додатків. Крім того, Docker забезпечує середовище виконання для сервісу, дає змогу переносити додатки на різні платформи та є основою для Docker Swarm і Kubernetes. Нарешті, завдяки використанню Docker стає можливим формування стандартних образів для двох обраних платформ, що полегшує їх порівняння.

Під час проектування та тестування системи на початковому етапі в локальному середовищі рекомендується використовувати Docker Compose. Docker Compose може допомогти швидко встановити зв'язок між усіма сервісами та перевірити їх взаємодію до того, як система почне працювати в кластерному середовищі. Таким чином, Docker Compose стає допоміжною технологією, яка зменшує складність початкового прототипу.

Для реалізації першого об'єкта порівняння обрано Kubernetes, оскільки ця платформа є найбільш поширеним рішенням для оркестрації контейнерів у складних розподілених системах. Kubernetes надає широкі можливості автоматичного масштабування, самовідновлення сервісів, декларативного

керування конфігураціями, балансування навантаження, керування секретами та підтримки складних мережових політик. У контексті цифровізації енергетики це має особливе значення, адже інформаційні системи підприємств часто повинні працювати безперервно, підтримувати різнорівневі сервіси та швидко адаптуватися до змін навантаження. Для розгортання Kubernetes у дипломній роботі доцільно використати Minikube або k3s. Minikube є зручним для локального тестування та відтворення кластерного середовища на одному вузлі, тоді як k3s є полегшеною реалізацією Kubernetes, що краще імітує використання в умовах обмежених ресурсів, характерних для окремих прикладних рішень у промисловості та енергетиці.

Другим об'єктом порівняння є Docker Swarm, який обрано як більш простий у налаштуванні та експлуатації оркестратор контейнерів. Його перевагами є швидке розгортання кластера, нижчий поріг входження, зрозуміла інтеграція з Docker та менша складність адміністрування. У межах дипломної роботи Docker Swarm доцільно використовувати як альтернативний підхід для організації контейнеризованої інфраструктури, орієнтований на сценарії, де відсутня потреба в надто складних механізмах оркестрації. Це особливо актуально для локальних або невеликих корпоративних систем енергетичного моніторингу, де важливими є швидкість впровадження та простота підтримки.

Рекомендується забезпечити зовнішній доступ до мікросервісів та направляти трафік через Traefik або NGINX Ingress. Такі інструменти слід обирати, оскільки необхідно мати єдине рішення для управління зовнішнім доступом, балансуванням навантаження та маршрутизацією трафіку. Якщо брати до уваги екосистему Kubernetes, то слід використовувати контролер Ingress, оскільки він відповідає архітектурі системи. У випадку Docker Swarm доцільно розглянути Traefik, оскільки це рішення може взаємодіяти з сервісами та автоматично налаштовувати маршрути відповідно до специфікацій контейнерів. Такі рішення допомагають імітувати умови експериментів на практиці.

Для моніторингу поточного стану системи, збору метрик про її продуктивність та візуалізації експериментальних даних необхідний наступний набір інструментів. Це Prometheus та Grafana. Prometheus дозволить збирати всі метрики, а Grafana допоможе створювати інформаційні панелі, що полегшить аналіз навантажень, використання ресурсів, часу відгуку та стабільності сервісів. Вибір саме цих інструментів обумовлений їхньою сумісністю з сучасними контейнеризованими платформами та корисністю для технічного аналізу.

Для автоматизації експериментів, запуску сценаріїв навантаження, збирання журналів і обробки результатів доцільно застосувати Bash або Python. Bash є зручним для написання простих сценаріїв автоматизації у Linux-середовищі, зокрема для запуску команд оркестраторів, перезапуску сервісів або емуляції збоїв. Python, у свою чергу, є доцільним для створення більш гнучких сценаріїв тестування, обробки зібраних даних, формування звітів і побудови допоміжної аналітики. Використання засобів автоматизації є важливим для забезпечення повторюваності експериментів і мінімізації впливу людського фактора на результати дослідження.

1.3 Архітектурні рішення прототипу

У межах роботи доцільно обрати мікросервісну архітектуру прототипу системи, оскільки саме вона найбільш повно розкриває можливості сучасних оркестраторів контейнерів. Прототип може складатися щонайменше з трьох основних компонентів: клієнтського вебінтерфейсу, серверного API та бази даних. Така структура дозволяє змодельовати типовий приклад інформаційної системи для енергетики, наприклад систему моніторингу енергоспоживання, обліку ресурсів або збору телеметричних даних. Мікросервісний підхід дає змогу окремо масштабувати компоненти, аналізувати їхню поведінку при відмовах і оцінювати ефективність балансування навантаження.

Архітектурно доцільно реалізувати прототип за моделлю «frontend + backend API + database», де frontend відповідає за візуалізацію даних і взаємодію з користувачем, backend API реалізує бізнес-логіку та обмін даними, а база даних забезпечує збереження показників моніторингу. Такий підхід є достатньо простим для реалізації в межах дипломної роботи та водночас достатньо репрезентативним для проведення порівняльного аналізу. Для підвищення достовірності експерименту всі компоненти системи повинні бути однаковими в обох середовищах оркестрації, а відмінності мають стосуватися лише механізмів розгортання, масштабування, відновлення та керування.

З погляду архітектури інфраструктури доцільно використовувати кластерний підхід, у межах якого вузли виконують ролі керування та виконання контейнерів. Це дозволяє змодельовати умови, наближені до реальної експлуатації цифрових платформ енергетичних підприємств, для яких критичними є безперервність роботи, відмовостійкість і можливість горизонтального масштабування. Для забезпечення коректного порівняння архітектура кластерів Kubernetes і Docker Swarm має бути максимально наближеною за кількістю вузлів, доступними ресурсами та набором розгорнутих сервісів.

Особливу увагу в роботі доцільно приділити питанням надійності та безпеки. Для енергетичних систем це є принципово важливим, оскільки навіть часткові збої або некоректна конфігурація доступу можуть призвести до втрати актуальних даних, затримок у прийнятті рішень або порушення процесів контролю. Саме тому в межах експериментів потрібно оцінювати не лише продуктивність, а й поведінку системи у випадку відмов вузлів, механізми автоматичного відновлення контейнерів, а також засоби ізоляції мережевої взаємодії та захисту службових конфігурацій.

Висновки до розділу 1

У першому розділі визначено вибір методів і програмних засобів, які є обґрунтованими у дипломній роботі та відповідають поставленій меті дослідження. Поєднання порівняльного аналізу, практичного експерименту, моніторингу метрик і автоматизації випробувань дозволяє отримати достовірні результати щодо можливостей Kubernetes і Docker Swarm. Обрані інструментальні засоби – Docker, Docker Compose, Kubernetes, Docker Swarm, Traefik або NGINX Ingress, Prometheus, Grafana, Bash і Python – формують достатню технологічну базу для побудови прототипу, проведення тестів і формування рекомендацій щодо доцільності використання того чи іншого оркестратора в задачах цифровізації енергетичних систем.

2 АНАЛІЗ, ДОСЛІДЖЕННЯ ІСНУЮЧИХ МЕТОДІВ

2.1 Загальний опис предметної області

Сучасний енергетичний сектор переживає фундаментальну трансформацію, зумовлену необхідністю інтеграції відновлюваних джерел енергії, впровадження інтелектуальних мереж та підвищення енергоефективності. Дослідження підтверджують, що цифровізація є ключовим фактором досягнення цілей сталого розвитку та забезпечення енергетичної безпеки. Згідно з даними Світового економічного форуму, енергетична індустрія активно впроваджує промисловий Інтернет речей, що дозволяє операційним технологіям цифрово управляти фізичними енергетичними активами та інтегруватися з інформаційними технологіями, які використовують штучний інтелект, машинне навчання та хмарні сховища.

Одним із ключових напрямів є забезпечення високої доступності та масштабованості розподілених енергетичних ресурсів через інтеграцію Інтернету речей, систем SCADA та платформ оркестрації контейнерів. Kubernetes та Docker Swarm представляють два альтернативні підходи до управління контейнеризованими застосунками, кожен з яких має власні переваги та обмеження в контексті енергетичного сектору [1].

Енергетичний сектор активно впроваджує контейнерні технології для управління розподіленими енергетичними ресурсами, інтеграції відновлюваних джерел енергії та оптимізації роботи інтелектуальних мереж [2]. Ця трансформація стає критичною для досягнення цілей сталого розвитку та забезпечення енергетичної безпеки.

Енергетична інфраструктура відноситься до критично важливих систем, де навіть короточасні збої можуть призвести до значних економічних збитків та

загроз безпеці. Забезпечення високої доступності (High Availability) та відмовостійкості (Fault Tolerance) є першочерговими вимогами для енергетичних застосунків [3]. Дослідження демонструють, що платформи оркестрації контейнерів забезпечують необхідні механізми автоматичного відновлення, багаторегіональне розгортання та розподілене управління робочими навантаженнями.

Kubernetes пропонує розширені можливості для досягнення високої доступності через самовідновлення (self-healing), горизонтальне автомасштабування подів та реплікацію компонентів панелі управління. Docker Swarm забезпечує вбудовану відмовостійкість через Raft Consensus Protocol, який гарантує узгодженість стану кластера навіть при відмові окремих керівних вузлів [1].

Інтелектуальні енергетичні мережі (Smart Grids) представляють фундаментальну трансформацію традиційної енергетичної інфраструктури через інтеграцію цифрових технологій, двостороннього зв'язку та розподілених енергетичних ресурсів [4]. На відміну від традиційних мереж, які базувалися на односторонньому потоці енергії від централізованих електростанцій до споживачів, Smart Grids забезпечують динамічне управління генерацією, розподілом та споживанням енергії. Архітектура Smart Grid включає численні компоненти: розумні лічильники (smart meters), IoT-датчики, системи SCADA, розподілені енергетичні ресурси (сонячні панелі, вітрові турбіни, накопичувачі енергії), системи зарядки електромобілів та платформи управління даними [5]. Для ефективної роботи цих компонентів необхідна складна програмна інфраструктура, здатна обробляти великі обсяги даних в режимі реального часу.

2.2 Аналіз платформ оркестрації контейнерів

2.2.1 Kubernetes: архітектура та можливості

Kubernetes є платформою оркестрації контейнерів з відкритим вихідним кодом, розробленою Google та підтримуваною Cloud Native Computing Foundation (CNCF). Архітектура Kubernetes базується на моделі master-worker, де панель управління (control plane) виконується на головних вузлах, а робочі навантаження розгортаються на робочих вузлах. Основні компоненти панелі управління включають API-сервер, який є центральним пунктом взаємодії, scheduler для розподілу задач між вузлами, etcd для зберігання стану кластера та controller manager для підтримки бажаного стану системи [1].

Робочі вузли виконують контейнери в абстракціях, що називаються подами (pods), які є найменшими одиницями розгортання в Kubernetes. Kubelet є агентом на кожному робочому вузлі, який забезпечує зв'язок з панеллю управління та управляє життєвим циклом контейнерів. Kubernetes використовує Docker або інші сумісні середовища виконання контейнерів (CRI) для фактичного запуску контейнерів [1].

Kubernetes був розроблений для великомасштабних розгорнутих систем і може масштабуватися до тисяч вузлів у кластері. Платформа забезпечує автоматичне горизонтальне масштабування подів (Horizontal Pod Autoscaler) на основі метрик процесора, пам'яті або користувацьких метрик. Дослідження демонструють, що Kubernetes має перевагу у продуктивності для великих та складних робочих навантажень, хоча і характеризується вищими накладними витратами порівняно з Docker Swarm.

Система планування Kubernetes використовує складні алгоритми для оптимального розміщення подів на вузлах з урахуванням вимог до ресурсів, обмежень та афінності. Платформа підтримує rolling updates та canary deployments, що дозволяє оновлювати застосунки без простоїв та з мінімальним ризиком. Для

енергетичних систем ця здатність є критичною, оскільки забезпечує безперервну роботу моніторингу та управління.

Kubernetes має найбільшу екосистему серед платформ оркестрації, підтримувану CNCF та величезною спільнотою розробників. Платформа інтегрується з широким спектром інструментів для моніторингу (Prometheus, Grafana), мережних рішень (Calico, Flannel, Cilium), управління секретами (HashiCorp Vault), service mesh (Istio, Linkerd) та безпеки. Всі основні хмарні провайдери пропонують керовані сервіси Kubernetes (Amazon EKS, Google GKE, Azure AKS), що спрощує розгортання та управління.

Для енергетичного сектору особливо важливою є можливість інтеграції Kubernetes з платформами великих даних, аналітики та штучного інтелекту. Компанія Capital Energy використовує BigQuery, Looker, Pub/Sub, Cloud Composer та Vertex AI разом з Kubernetes для застосування AI до управління енергією та оптимізації роботи сотень тисяч IoT-датчиків на вітрових та фотоелектричних установках.

2.2.2 Docker Swarm: архітектура та можливості

Docker Swarm є власним рішенням Docker для оркестрації контейнерів, інтегрованим безпосередньо в Docker Engine. На відміну від Kubernetes, Docker Swarm використовує більш плоску архітектуру, де кожен вузол може виступати як менеджер (manager) або робочий вузол (worker). Менеджери відповідають за оркестрацію та управління кластером, тоді як робочі вузли виконують задачі [6].

Swarm використовує Raft Consensus Algorithm для забезпечення узгодженості стану між менеджерами. Якщо один з менеджерів виходить з ладу, інші можуть продовжувати управління кластером без переривання обслуговування. Сервіси в Docker Swarm визначаються декларативно, і платформа автоматично підтримує бажаний стан шляхом створення або видалення реплік контейнерів [6].

Головною перевагою Docker Swarm є простота налаштування та використання. На відміну від Kubernetes, який вимагає встановлення та налаштування численних компонентів, Docker Swarm активується однією командою на існуючих Docker-вузлах. Ця простота робить Docker Swarm привабливим вибором для команд з обмеженими ресурсами або для проєктів, що не потребують складних можливостей оркестрації.

Docker Swarm має вбудовану систему балансування навантаження через routing mesh, яка автоматично розподіляє мережевий трафік між контейнерами сервісу. Це спрощує налаштування порівняно з Kubernetes, де балансування навантаження вимагає додаткової конфігурації через Service та Ingress ресурси. Для невеликих енергетичних проєктів або дослідних установок ця простота може бути вирішальним фактором.

Хоча Docker Swarm і підтримує масштабування, його можливості обмежені порівняно з Kubernetes. Docker Swarm ефективно працює з кластерами до сотень вузлів, але може мати проблеми з продуктивністю у дуже великих середовищах, що вимагають складного управління ресурсами. Масштабування в Docker Swarm переважно ручне або через прості команди, тоді як Kubernetes пропонує складні механізми автоматичного масштабування.

Екосистема Docker Swarm є значно меншою порівняно з Kubernetes. Хоча існують інструменти для роботи зі Swarm, вибір інтеграцій та підтримуваних рішень обмежений. Відсутність активного розвитку нових функцій та зменшення промислової підтримки також є факторами, що обмежують привабливість Docker Swarm для довгострокових проєктів.

2.2.3 Альтернативні платформи оркестрації

Apache Mesos є розподіленою системою управління кластерами, яка абстрагує процесор, пам'ять, сховище та інші ресурси. Mesos підтримує як контейнеризовані, так і неконтейнеризовані робочі навантаження, що робить його

універсальним рішенням. Marathon є фреймворком оркестрації контейнерів, побудованим на основі Mesos.

Основні переваги Apache Mesos включають високу масштабованість (до десятків тисяч вузлів), детальний розподіл ресурсів та підтримку різноманітних типів робочих навантажень. Однак Mesos характеризується високою складністю налаштування та управління, а також меншою підтримкою спільноти порівняно з Kubernetes. Спільнота Mesos є невеликою, і Marathon більше не розвивається активно, що обмежує його застосування у нових проєктах.

HashiCorp Nomad є легковаговим оркестратором, здатним управляти контейнерами, віртуальними машинами та застарілими застосунками. Nomad інтегрується з іншими інструментами HashiCorp, такими як Consul для service discovery та Vault для управління секретами. Платформа пропонує простоту налаштування та низькі накладні витрати, що робить її привабливою для організацій з різноманітними типами робочих навантажень.

Однак Nomad має меншу екосистему порівняно з Kubernetes та обмежені вбудовані можливості для розширеного мережевого управління та моніторингу. Користувачі відзначають, що хоча Nomad є гнучким та легким, більшість компаній обирають Kubernetes через його комплексність та підтримку. Для енергетичного сектору, де критична підтримка спільноти та широка екосистема інтеграцій, Kubernetes залишається більш безпечним вибором.

2.3 Порівняльний аналіз для енергетичного сектору

2.3.1 Масштабованість та продуктивність

Енергетичні системи бувають найрізноманітнішими: від локальних мікромереж з об'єктами відновлюваної енергетики до регіональних енергомереж, що забезпечують мільйони людей і використовують тисячі джерел енергії. Така

диверсифікація вимагає систем оркестрування, здатних масштабуватися від експериментальних проєктів до промислових застосувань.

Інфраструктура великих енергетичних корпорацій використовує сотні тисяч пристроїв Інтернету речей (IoT) для збору даних, генеруючи таким чином величезний обсяг даних, які потрібно обробляти в режимі реального часу. У таких сценаріях система оркестрування має масштабуватися до тисяч вузлів з ефективним розподілом ресурсів між різними типами робочих навантажень. З іншого боку, у менших організаціях або в пілотних проєктах може бути потреба у більш легких системах оркестрування з меншими накладними витратами.

Kubernetes має переваги для великих систем. Він демонструє явну перевагу для великомасштабних енергетичних систем завдяки здатності масштабуватися до тисяч вузлів та розширеним можливостям автоматичного масштабування. Horizontal Pod Autoscaler дозволяє автоматично збільшувати або зменшувати кількість реплік подів на основі метрик використання процесора, пам'яті або користувацьких метрик, таких як довжина черги повідомлень або інтенсивність запитів.

Для енергетичних систем, що працюють з змінними навантаженнями (наприклад, періоди високої та низької генерації відновлюваної енергії), автоматичне масштабування є критичним для забезпечення оптимального використання ресурсів. Компанія Energyworx успішно мігрувала свої конвеєри обробки даних на Kubernetes, що дало їм повний контроль над автоматичним масштабуванням через Horizontal Pod Autoscalers. Це дозволило обробляти тисячі контейнерів одночасно та ефективно керувати обробкою даних від численних енергетичних активів.

Натомість кластер Docker Swarm видається кращим вибором для малих та середніх енергетичних проєктів, у яких на першому місці стоять простота розгортання та низькі експлуатаційні витрати. Це програмне забезпечення ефективно працює в кластерах, що складаються з кількох сотень вузлів, а також забезпечує базову масштабованість за допомогою простих команд або

налаштувань. Для локальних мікромереж, наукових установ, що досліджують відновлювані джерела енергії, або кооперативів в енергетичній галузі Docker Swarm має бути цілком достатнім.

Практичний досвід показує, що Docker Swarm працює досить ефективно, коли використовується у відносно простих сценаріях. Дійсно, були випадки, коли деякі компанії підтримували кластери Docker Swarm протягом тривалого часу без серйозних проблем, використовуючи рішення Serf для забезпечення високої доступності. Однак слід усвідомлювати, що кластер Docker Swarm не підтримує нові технології, і тому більшість компаній віддають перевагу Kubernetes.

2.3.2 Безпека та відмовостійкість

Питання забезпечення безпеки оркестрування контейнерів в енергетичному секторі є критично важливим фактором, який слід враховувати, оскільки системи управління можуть стати мішенями, що заважатимуть постачанню енергії та становитимуть загрозу національній безпеці. Обидві платформи забезпечують безпеку, проте вони відрізняються за можливостями та рівнем складності.

Kubernetes пропонує розширені функції безпеки, такі як контроль доступу на основі ролей (RBAC), мережеві політики, управління секретними даними, сканування образів на наявність вразливостей та інтеграція зі сторонніми рішеннями з безпеки. Згідно з емпіричними дослідженнями, проведеними щодо безпеки оркестрування контейнерів у хмарних сервісах, для ефективної безпеки необхідне впровадження політик безпеки на рівні мережі, заходів безпеки під час виконання, контролю доступу, безпеки ланцюга постачання та забезпечення безпеки самої платформи оркестрування. У свою чергу, це призводить до зменшення порушень безпеки на 35%, кращого дотримання нормативних вимог на 25% та підвищення операційної ефективності на 20%.

Що стосується Docker Swarm, то деякі з заходів безпеки, які він забезпечує, включають шифрування комунікацій між вузлами за допомогою TLS, ізоляцію

мережі та управління секретами. Проте, порівняно з Kubernetes, спектр додаткових інструментів безпеки, сумісних з Docker Swarm, є значно вужчим. Для тих, хто працює у сфері виробництва енергії та стикається з жорсткими вимогами безпеки, Kubernetes є кращим рішенням.

2.4 Практичні кейси впровадження

2.4.1 Energyworx: міграція на Kubernetes

Компанія Energyworx звернулася до Nordcloud для допомоги в міграції їх конвеєрів обробки даних на Kubernetes та досягнення економії витрат на обчислювальних навантаженнях. Великою частиною кодової бази не було тісно пов'язано з моделлю програмування Beam, тому команда змогла повністю видалити фреймворк Beam та контейнеризувати конвеєри, що зробило можливим їх запуск на Kubernetes без Beam runner [7].

Одним з ключових викликів була обробка відмов - питання, такі як повторні спроби при виконанні API-викликів або моніторинг статусу застосунків, раніше покривалися Dataflow, але тепер потребували спеціального рішення. При запуску тисяч контейнерів у Kubernetes вони часто завершуються, що може відбутися посеред обробки або відправки вихідних даних, спричиняючи проблеми з продуктивністю та дублювання даних. Команда розробила нові сервіси для обробки таких ситуацій [7].

Kubernetes надав повний контроль над автоматичним масштабуванням через Horizontal Pod Autoscalers. Хоча функція автоматичного масштабування в принципі була великою перевагою, відсутність можливості тонкого налаштування на детальному рівні є значним недоліком попередніх рішень. Kubernetes дозволяє повний контроль над автоматичним масштабуванням, що є критичним для обробки змінних навантажень у енергетичних застосунках [7].

2.4.2 Sunrock: чиста енергетична майбутність з Kubernetes

Ринок відновлюваної енергії швидко розвивається, і для того, щоб залишатися попереду, компанії Sunrock потрібно було впровадити передові системи управління енергією та інноваційні рішення для оптимального впровадження та налаштування їх енергетичних активів з використанням даних реального часу у різних локаціях. З зростанням портфоліо Sunrock важливо впровадити масштабоване рішення, здатне обробляти такий обсяг даних [8].

Для адаптації до вимог масштабованості Pionative впровадила та інтегрувала стек Kubernetes у середовище Azure компанії Sunrock, забезпечуючи надійну та безпечну платформу для застосунків Sunrock. Стек Kubernetes постійно моніториться Pionative з використанням сучасних технологій спостережливості. Інженери Sunrock використовують стек для підвищення якості розробки, що також допомагає їм виявляти проблеми раніше в циклі випуску, прискорюючи швидкість розробки [8].

З Kubernetes як масштабованою платформою, Sunrock ідеально позиціонована для майбутніх інновацій. Оскільки сектор відновлюваної енергії розвивається, Sunrock матиме гнучкість, безпеку та масштабованість, які потрібні в майбутньому. Старший інженер з даних Sunrock відзначив, що Kubernetes є критичним для управління великими обсягами даних та забезпечення надійності систем моніторингу та управління .

Висновки до розділу 2

У другому розділі було розглянуто предметну область дослідження, пов'язану з цифровізацією енергетичних систем, а також проаналізовано підходи до оркестрації контейнеризованих застосунків. Встановлено, що впровадження мікросервісних і контейнерних технологій є доцільним для побудови масштабованих, гнучких і відмовостійких інформаційних систем в енергетичній

галузі. У межах розділу здійснено аналіз платформ Kubernetes і Docker Swarm, їх архітектурних особливостей, функціональних можливостей, переваг і обмежень. Визначено, що Kubernetes є більш потужним і функціонально розвиненим рішенням, яке доцільно використовувати у складних розподілених системах з підвищеними вимогами до масштабованості, автоматизації, безпеки та відмовостійкості. Docker Swarm, у свою чергу, характеризується простотою розгортання та адміністрування, що робить його придатним для невеликих або середніх за масштабом систем із меншими вимогами до інфраструктури. Також було розглянуто альтернативні платформи оркестрації контейнерів і встановлено, що за рівнем поширеності, зрілості екосистеми та можливостями інтеграції вони поступаються Kubernetes і Docker Swarm у контексті поставленого завдання. Окрему увагу приділено критеріям, що є визначальними для енергетичного сектору, зокрема масштабованості, продуктивності, безпеці, відмовостійкості.

3 ЗАСОБИ РОЗРОБКИ СИСТЕМИ

Для якісного проведення дослідження та реалізації програмного прототипу необхідно заздалегідь обрати інструментальні засоби й технології, які забезпечать коректне розгортання, тестування та порівняння контейнеризованих застосунків. Обґрунтований вибір програмних засобів дає змогу не лише реалізувати мікросервісну систему, а й забезпечити об'єктивність експериментів, відтворюваність результатів і практичну цінність отриманих висновків. У межах дипломної роботи доцільно використати сучасний стек технологій, орієнтований на побудову контейнеризованих розподілених систем.

Розроблюваний прототип матиме мікросервісну архітектуру та складатиметься з таких основних компонентів: клієнтського веб-інтерфейсу, серверного API та бази даних. Для контейнеризації програмних компонентів буде використано Docker, що забезпечить уніфіковане середовище виконання сервісів. Для локального моделювання та початкового налагодження взаємодії між компонентами доцільно застосувати Docker Compose. Як платформи оркестрації контейнерів для подальшого порівняльного аналізу буде використано Kubernetes та Docker Swarm, що дозволить оцінити їхні можливості в однакових умовах функціонування.

Для організації зовнішнього доступу до сервісів і маршрутизації мережевого трафіку доцільно використати Traefik або NGINX Ingress, а для моніторингу технічних показників системи – Prometheus і Grafana. Автоматизація запуску експериментів, збору метрик і обробки результатів може бути реалізована за допомогою Bash або Python. Такий набір інструментів формує достатню технологічну основу для побудови прототипу мікросервісної системи та проведення порівняльного аналізу Kubernetes і Docker Swarm у задачах цифровізації енергетичних систем.

3.1 Середовище розробки PyCharm

Для розробки програмної частини даного дослідження було обрано інтегроване середовище розробки PyCharm. PyCharm можна визначити як програмне забезпечення, розроблене спеціально для роботи над проектами, що реалізуються на мові Python.

Крім того, у режимі Pro можна розробляти застосунки для Django, Flask та Pyramid. Також повністю підтримується HTML (включно з HTML5), CSS, JavaScript та XML: ці мови входять до складу IDE за допомогою плагінів та ввімкнені за замовчуванням. Підтримку інших мов та фреймворків також можна додати за допомогою плагінів [9]. Зокрема, PyCharm надає користувачеві такі можливості, як написання коду, запуск програм, налагодження програмного забезпечення, управління структурою проектів та додавання додаткових залежностей – і все це з одного місця. На рисунку 3.1 зображено інтерфейс середовища розробки PyCharm.

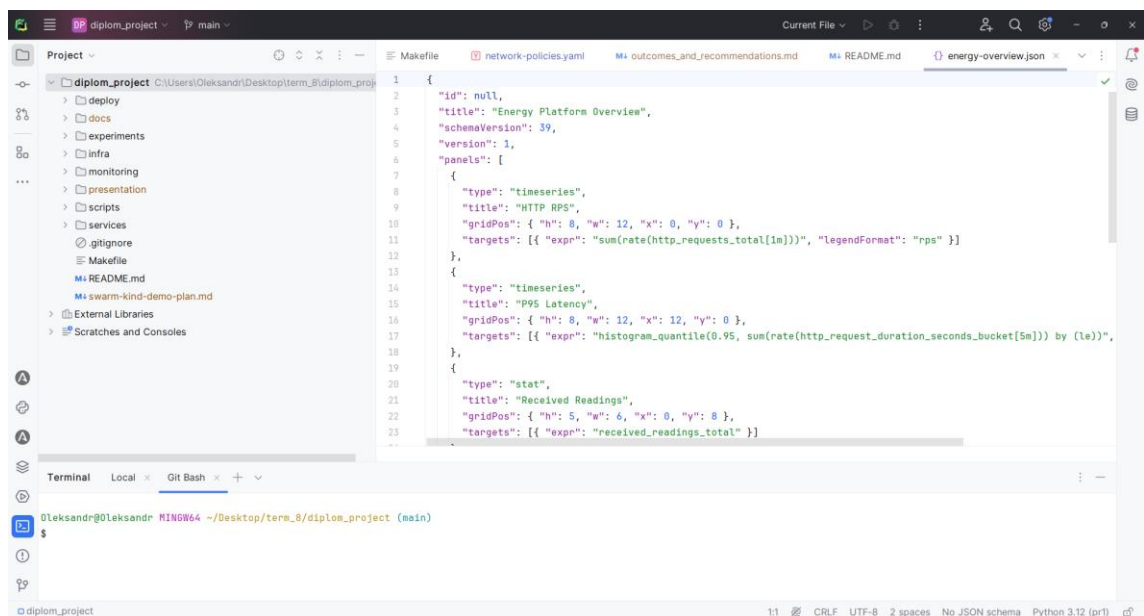


Рисунок 3.1 – Середовище розробки PyCharm

Використання такого програмного забезпечення є логічним, оскільки це дозволяє вести структуровану роботу з усіма файлами, пов'язаними з проектом, а

також отримувати доступ до необхідних інструментів програмування. PyCharm використовувався для розробки всіх компонентів досліджуваного програмного забезпечення.

3.2 Мова програмування Python та FastAPI

Для роботи було обрано Python як інструмент для розробки серверної частини системи, її симулятора, різних утиліт, засобів тестування та експериментальних модулів.

Python – це легка для вивчення та потужна мова програмування. Вона має ефективні високорівневі структури даних та простий, але дієвий підхід до об'єктно-орієнтованого програмування. Елегантний синтаксис Python та динамічна типізація, разом з його інтерпретованою природою, роблять її ідеальною мовою для написання сценаріїв та швидкої розробки додатків у багатьох областях на більшості платформ [10]. Таке рішення обумовлено кількома причинами. По-перше, завдяки своїй гнучкості та придатності для різних видів розробки програмного забезпечення ця мова програмування є досить універсальною та зручною для створення численних функціонально різноманітних програмних компонентів на основі єдиної технологічної платформи. По-друге, з огляду на наявність численних бібліотек та можливостей для організації вихідного коду програм, Python можна вважати оптимальним вибором для створення науково-дослідного програмного рішення для порівняння Kubernetes та Docker Swarm.

FastAPI – це сучасний, швидкий (високопродуктивний) веб-фреймворк для створення API на Python, що базується на стандартних підказках щодо типів Python [11]. FastAPI у межах даної дипломної роботи було використано як фреймворк для реалізації серверної частини мікросервісної системи. Це рішення є доцільним завдяки високій швидкодії, підтримці асинхронної обробки запитів, зручній побудові REST API та автоматичній генерації інтерактивної документації, що спрощує розробку, тестування й подальший супровід застосунку. Використання

FastAPI дало змогу побудувати backend-компонент дослідної системи в сучасному стилі, забезпечити чітку організацію програмного коду та зручну взаємодію між сервісами в межах експериментального середовища порівняння Kubernetes і Docker Swarm.

3.3 Програмні засоби для роботи з базою даних

У проєкті для зберігання даних на стороні сервера використовується база даних. До технологій зберігання та обробки даних належать: PostgreSQL; ORM SQLAlchemy; Alembic.

PostgreSQL – це потужна об'єктно-реляційна система баз даних з відкритим кодом, яка використовує та розширює мову SQL у поєднанні з багатьма функціями, що безпечно зберігають та масштабують найскладніші робочі навантаження даних. Витоки PostgreSQL сягають 1986 року як частина проєкту POSTGRES в Каліфорнійському університеті в Берклі та має майже 40 років активної розробки на основній платформі [12].

PostgreSQL заслужив міцну репутацію завдяки своїй перевірній архітектурі, надійності, цілісності даних, потужному набору функцій, розширюваності та відданості спільноти розробників відкритого коду, яка стоїть за цим програмним забезпеченням, постійному наданню продуктивних та інноваційних рішень. PostgreSQL працює на всіх основних операційних системах, сумісний з ACID з 2001 року та має потужні доповнення, такі як популярний розширювач геопросторової бази даних PostGIS. Не дивно, що PostgreSQL став реляційною базою даних з відкритим кодом для багатьох людей та організацій [12]. PostgreSQL виступає в ролі основної бази даних і гарантує надійне зберігання даних.

SQLAlchemy розглядає базу даних як механізм реляційної алгебри, а не просто як набір таблиць. Рядки можна вибирати не лише з таблиць, але й з об'єднань та інших операторів вибору; будь-яку з цих одиниць можна об'єднати в більшу структуру. Мова виразів SQLAlchemy базується на цій концепції з самого

початку [13]. ORM SQLAlchemy дозволяє взаємодіяти з базою даних із коду на стороні сервера, що спрощує операції з даними.

Alembic – це інструмент для міграції баз даних, написаний автором SQLAlchemy [14]. Водночас Alembic забезпечує можливість поступового внесення змін до схеми бази даних. Усі згадані інструменти підтримують роботу серверної частини та забезпечують узгодженість даних.

3.4 Засоби контейнеризації та оркестрації системи

З метою розгортання системи мікросервісів із використанням контейнеризації в рамках даного дослідження було застосовано кілька інструментів, а саме: Docker, Docker Compose, Docker Swarm та Kubernetes. Кожен інструмент відповідає за окреме завдання, що дає змогу розгорнути систему в різних сценаріях – від локального тестування до розгортання в кластері.

Перший використаний інструмент, Docker, відповідає за генерацію контейнерних образів елементів системи. Docker надає можливість пакувати та запускати застосунок у слабо ізольованому середовищі, яке називається контейнером. Ізоляція та безпека дозволяють запускати багато контейнерів одночасно на певному хості. Контейнери легкі та містять усе необхідне для запуску застосунку, тому вам не потрібно покладатися на те, що встановлено на хості. Ви можете ділитися контейнерами під час роботи та бути впевненим, що всі, з ким ви ділитесь ними, отримують той самий контейнер, який працює однаково [15].

За допомогою Docker можна побудувати контейнеризовані мікросервіси таким чином, щоб їх можна було розгорнути як локально, так і в кластерах Docker Swarm та Kubernetes.

Інший важливий інструмент, Docker Compose, використовується для налаштування локального тестового середовища експериментальної системи. Опис взаємозв'язків між мікросервісами та базою даних, а також іншими компонентами системи, дозволяє дуже швидко налаштувати все локальне тестове середовище на

одній машині. Docker Compose – це інструмент для визначення та запуску багатоконтейнерних застосунків. Він є ключем до оптимізації та ефективності розробки та розгортання [16].

Docker Swarm використовується як один із варіантів оркестрації контейнерів для кластерного деплою. У рамках дипломного проєкту Docker Swarm застосовується для порівняння з Kubernetes, оскільки він дозволяє розгорнути мікросервісну систему в середовищі з кількома вузлами та забезпечити базові можливості оркестрації, масштабування та відмовостійкості. Режим Swarm – це розширена функція для керування кластером демонів Docker [17].

Kubernetes, також відомий як K8s, – це система з відкритим кодом для автоматизації розгортання, масштабування та управління контейнерними додатками. Він групує контейнери, що складають додаток, у логічні одиниці для зручного керування та виявлення [18].

Kubernetes у проєкті використовується як основна платформа оркестрації контейнерів у кластерному середовищі. На базі Kubernetes реалізується розгортання мікросервісної системи з можливістю автоматичного масштабування, самовідновлення сервісів, балансування навантаження та керування конфігураціями. Kubernetes служить порівнюваним рішенням до Docker Swarm і дозволяє оцінити відмінності у підходах до оркестрації в умовах, наближених до реальних сценаріїв цифровізації енергетичних систем.

3.5 Інструменти побудови експериментального середовища

Kind – це інструмент для запуску локальних кластерів Kubernetes за допомогою вузлів контейнера Docker. Kind був розроблений в першу чергу для тестування самого Kubernetes, але може використовуватися для локальної розробки або неперервної інтеграції [19].

Це дає змогу відтворити кластерне середовище на одному комп'ютері, виконувати тестування конфігурацій, перевіряти розгортання сервісів і проводити локальні експерименти без потреби у виділеній фізичній інфраструктурі.

Multipass – це інструмент для швидкої генерації віртуальних машин Ubuntu у хмарному стилі на Linux, macOS та Windows. Він надає простий, але потужний інтерфейс командного рядка, який дозволяє швидко отримати доступ до командного рядка Ubuntu або створити власну локальну міні-хмару [20].

Multipass застосовується для створення трьох віртуальних машин з операційною системою Ubuntu, на базі яких розгортається кластер Docker Swarm. Такий підхід дозволяє змодельовати багатовузлове середовище, наближене до реальних умов експлуатації, і забезпечує можливість дослідження кластерного деплою, взаємодії вузлів та поведінки системи при оркестрації контейнерів.

3.6 Засоби моніторингу та аналізу показників системи

Prometheus – це набір інструментів для моніторингу та оповіщення систем з відкритим кодом, спочатку створений на SoundCloud. З моменту свого створення в 2012 році багато компаній та організацій перейняли Prometheus, і проект має дуже активну спільноту розробників та користувачів. Зараз це окремий проект з відкритим кодом, який підтримується незалежно від будь-якої компанії [21]. Prometheus використовується в проєкті як система збору та зберігання метрик продуктивності контейнеризованих сервісів. На основі Prometheus формуються часові ряди даних, які містять інформацію про використання ресурсів, навантаження на сервіси, відгуки систем та ключові показники стабільності. Це дозволяє проводити кількісне порівняння роботи Kubernetes і Docker Swarm у різних експериментальних сценаріях.

Grafana застосовується для візуалізації метрик, зібраних Prometheus. За допомогою Grafana створюються інформаційні панелі, які відображають споживання процесора, оперативної пам'яті, затримки відповідей, кількість запитів

та інші технічні показники системи у зручному графічному вигляді. Це забезпечує наочне порівняння ефективності обох оркестраторів і спрощує інтерпретацію результатів експериментів.

Висновки до розділу 3

У третьому розділі було розглянуто програмні засоби, використані для реалізації, розгортання та супроводу дослідної мікросервісної системи. Обґрунтовано вибір інструментів, що забезпечують роботу з базою даних, контейнеризацію програмних компонентів, оркестрацію сервісів, формування локального й кластерного середовища, а також моніторинг і візуалізацію технічних показників системи. Для збереження структурованих даних у проєкті використано PostgreSQL, для організації взаємодії між серверною частиною та базою даних – SQLAlchemy ORM, а для керування змінами структури бази даних – Alembic. Такий підхід забезпечує цілісність зберігання інформації, зручність роботи з даними та впорядкований супровід схеми бази даних у процесі розвитку системи. Для контейнеризації програмних компонентів використано Docker, що забезпечує створення єдиних образів сервісів для різних середовищ розгортання. Docker Compose застосовано для побудови локального стенда, Docker Swarm – для кластерного деплою як одного з об'єктів порівняння, а Kubernetes – як основну платформу оркестрації контейнерів у межах експериментального дослідження. Додатково kind було використано для створення локального Kubernetes-кластера в середовищі Docker, а Multipass – для розгортання трьох віртуальних машин Ubuntu, на яких моделюється кластер Docker Swarm. Для збору технічних показників функціонування системи використано Prometheus, а для їх візуалізації та подальшого аналізу – Grafana. Це дало змогу отримати кількісні характеристики роботи дослідної системи та сформуванати наочну основу для порівняння ефективності Kubernetes і Docker Swarm у різних сценаріях навантаження та розгортання.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

У цьому розділі наведено опис програмної реалізації навчально-демонстраційної платформи обліку електроенергії, побудованої як мікросервісний контейнеризований застосунок із локальним розгортанням через Docker Compose . Реалізація охоплює backend, frontend, симулятор надходження показників, базу даних і стек моніторингу, що дозволяє відтворити типовий сценарій цифровізації енергетичної системи в локальному середовищі. Окрім локального запуску, застосунок також адаптовано для розгортання в середовищах Docker Swarm та Kubernetes, що забезпечує можливість порівняння підходів до оркестрації контейнерів, масштабування сервісів і керування їхньою взаємодією в різних умовах експлуатації.

4.1 Архітектура програмного забезпечення

Архітектура реалізована за принципом розподілу відповідальності між контейнеризованими сервісами. Прикладний рівень містить backend (REST API та доступ до даних), frontend (вебінтерфейс), simulator (джерело тестового навантаження показань) та PostgreSQL як єдине сховище доменних даних. Окремо виділяється підсистема спостережуваності: Prometheus, Grafana, node-exporter та cAdvisor, які збирають прикладні, системні та контейнерні метрики без модифікації бізнес-таблиць.

Мікросервісне розбиття на рівні контейнерів є спільною передумовою для Docker Swarm і Kubernetes: обидва керують життєвим циклом незалежних одиниць розгортання (задачі Swarm / Pod у Kubernetes), маршрутизацією сервісного трафіку та масштабуванням реплік. Коли прикладні сервіси не прив'язані до конкретного оркестратора (одні й ті самі образи описуються різними маніфестами), порівняння зміщується з «якість коду» на «властивості платформи»: швидкість планування,

політики оновлення, наявність декларативних мережевих політик, автоматичне горизонтальне масштабування тощо. Моніторинговий шар, однаковий за складом метрик на стенді, забезпечує узгоджену базу спостережень для обох платформ і знижує ризик хибних висновків через різні інструменти збору даних. Архітектура експериментального стенду зображено на рисунку 4.1

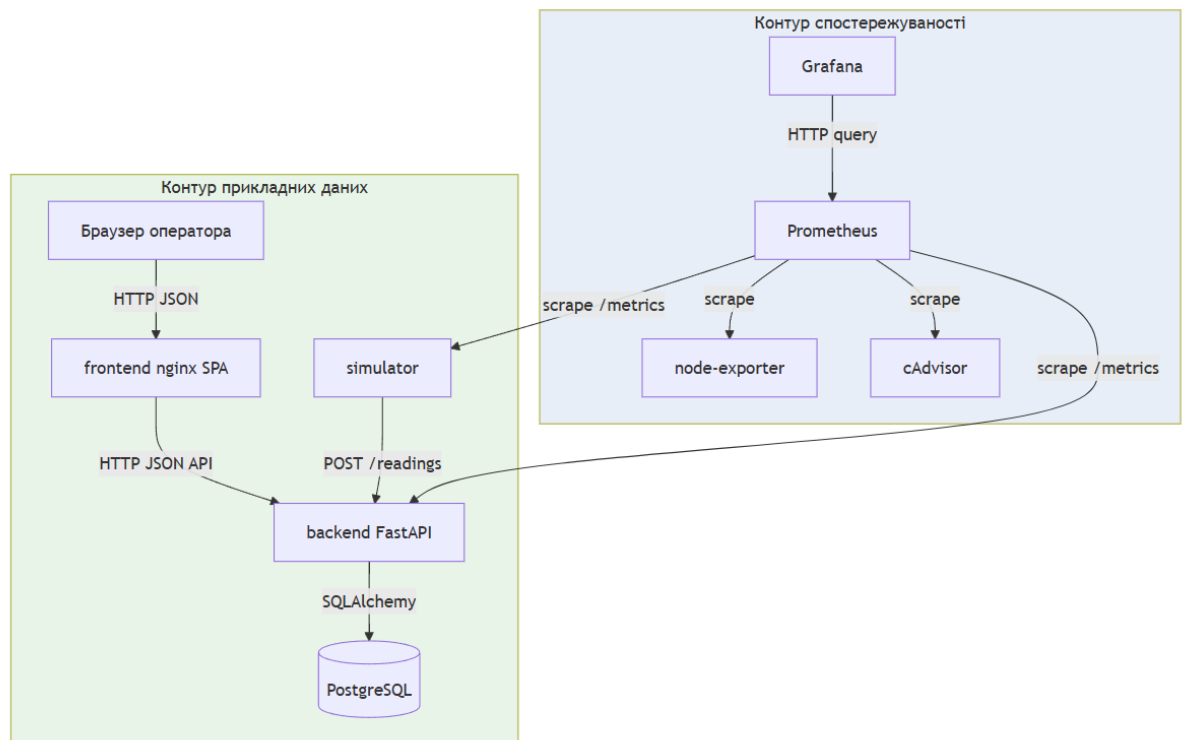


Рисунок 4.1 – Архітектура експериментального стенда

Ця діаграма деталізує взаємодію між контуром прикладних даних і контуром спостережуваності в реалізованому стенді платформи обліку електроенергії. У лівій частині показано прикладний контур, де браузер оператора працює через frontend nginx SPA, simulator генерує потік показань, backend FastAPI обробляє запити POST /readings, а PostgreSQL використовується для зберігання даних.

4.2 Опис програмних компонентів стенда

Бекенд написано на Python і використовує FastAPI разом із сервером Uvicorn ASGI. Зчитування даних здійснюється за допомогою SQLAlchemy 2, конфігурації та перевірка – за допомогою Pydantic, міграція схем баз даних – за допомогою Alembic, драйвером PostgreSQL є psycopg2, а експорт метрик за допомогою prometheus-client відбувається за шляхом /metrics. З функціональної точки зору, бекенд веде бази даних посилань на об'єкти вимірювання (мережі, підприємства, об'єкти, лічильники), приймає показання за адресою /readings, перевіряє та агрегує показання, генерує звіти та сигнали тривоги, а також документацію API (зазвичай OpenAPI, на іншому порту локально).

Бекенд є основним носієм навантаження експерименту: ця служба підлягає таким вимірюванням, як масштабування реплік, перезапуск після примусового завершення контейнера/пода, вимірювання тривалості послідовного оновлення, реакція НРА на метрику використання ЦП під час роботи на Kubernetes. Відтворюваність збірки контейнерів бекенду як у кластерах Swarm, так і Kubernetes гарантує, що будь-які відмінності у часі відгуку та стабільності походять виключно від оркестрування.

Фронтенд – це SPA на базі React 18 та Vite, що використовує React Router як механізм маршрутизації, Axios для HTTP-операцій та Recharts для візуалізації; React Flow + dagre дозволяють створювати графіки, що відображають топологію мережі. Статичний контент обслуговується nginx у виробничому середовищі всередині контейнера. Базова URL-адреса API вводиться у фронтенд під час побудови за допомогою змінної VITE_API_BASE_URL, що дозволяє браузерним запитам до бекенду залишатися однаковими у Compose, кластері nodeport та зворотньому проксі.

Як приклад звичайної зовнішньої служби з конкретним кінцевим пунктом буде використано інтерфейс користувача. Для оркестраторів наведена вище конфігурація дозволяє оцінити не лише взаємодію між контейнерами всередині

інструменту оркестрації, а й методи маршрутизації трафіку (Kubernetes Ingress проти Swarm Overlay, Nginx), а також правильність налаштування клієнтів щодо URL-адреси API після розгортання.

Симулятор поводить себе як система, яка отримує показання від активних лічильників через API бекенду, генерує інтервальні значення на основі файлу `scenario.yaml`, надсилає їх за допомогою методу `POST /readings`, виставляє метрики на `/metrics` та приймає команди керування за допомогою власного HTTP API (запуск, зупинка, завантаження нового сценарію).

Таким чином, симулятор стає настроюваним генератором навантаження на додатки: кількість запитів та їхні сценарії можна змінювати експериментально (що особливо потрібно для тестування горизонтального автомасштабування подів у Kubernetes та симетричного HTTP-навантаження у Swarm). Це дозволяє імітувати навантаження, спричинене вимірюванням, у разі відсутності доступу до розгортання в польових умовах, та забезпечує відтворюваність тестових вимірювань на обох платформах.

PostgreSQL – це єдине місце, де зберігаються всі дані, пов'язані з доменом (архітектура OLTP, спільна база даних для бекенду в MVP). Така конструкція спрощує підтримку транзакційної узгодженості та єдиної семантики даних незалежно від конфігурації розгортання.

Використання єдиної бази даних виключає відмінності між системами з точки зору «різних репозиторіїв у різних сервісах». Таким чином, тепер можна порівнювати підходи до персистентності (томи, претензії, монтування) залежних сервісів, їх готовності, а також методи оновлень, коли компоненти зі станом співіснують з API без стану. Склад сервісів стенду представлено в таблиці 4.1.

Таблиця 4.1 – Склад сервісів стенда

Сервіс	Технології	Інтерфейси (типово, Compose)	Основні залежності
backend	Python, FastAPI, Uvicorn, SQLAlchemy, Alembic, prometheus-client	TCP 8000 HTTP (/docs, /metrics)	PostgreSQL
frontend	React, Vite, nginx	TCP 8081→80 HTTP (SPA)	backend (VITE_API_BASE_URL)
simulator	Python, HTTP-клієнт	TCP 8010 HTTP (керування, /metrics)	Backend API_URL
postgres	PostgreSQL 16	TCP 5432 (внутрішня мережа)	немає
backend-migrate	той самий образ, що backend	одноразовий запуск Alembic	PostgreSQL healthy
prometheus	Prometheus	TCP 9090 HTTP	таргети з prometheus.yml
grafana	Grafana	TCP 3000 HTTP	Prometheus як datasource
node-exporter	prom/node-exporter	TCP 9100 /metrics	Хост /proc, /sys
cadvisor	cAdvisor	TCP 8080 /metrics	privileged, Docker socket / cgroup

4.2.1 Логічна модель системи та потік даних

Логічна модель відображає типові процеси цифровізації енергетики: збір і облік показань, підтримка топологічної моделі мережі, аналітика та надання оператору інтерактивного доступу через вебінтерфейс.

Користувач взаємодіє з frontend у браузері; клієнт надсилає HTTP JSON-запити до backend. Simulator автономно отримує перелік лічильників і публікує показання через POST /readings. Backend виконує читання й запис у PostgreSQL через SQLAlchemy. Жоден інший описаний сервіс не оновлює доменні таблиці напряму.

Prometheus періодично отримує метрики з /metrics backend і simulator, а також з node-exporter та cAdvisor. Grafana підключається до Prometheus як джерела даних і відображає підготовлені дашборди.

Розділення прикладного контуру та телеметричного дозволяє під час експерименту відокремити деградацію бізнес-операцій (латентність API, помилки запису) від поведінки збору метрик (затримки scrape, навантаження на експортери). Для Kubernetes і Swarm це важливо: обидві платформи по-різному планують Pod/задачі на вузлах і по-різному організовують сервісне виявлення; однаковий набір scrape-цілей зменшує кількість неконтрольованих змінних при інтерпретації результатів порівняння. контекстна DFD-подібна діаграма двох підсистем («Облік енергії» та «Моніторинг») зображена на рисунку 4.2

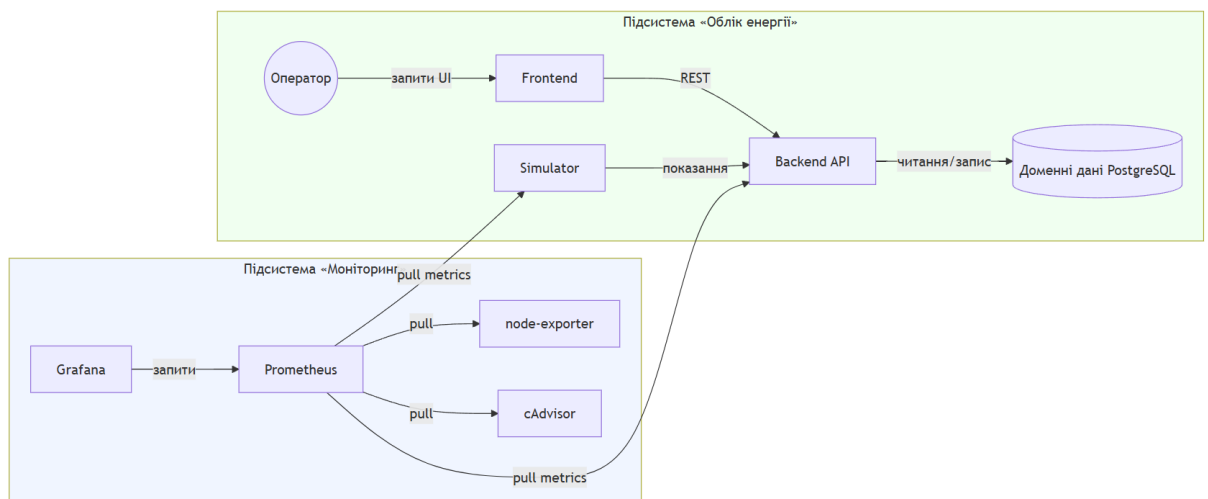


Рисунок 4.2 – контекстна DFD-подібна діаграма двох підсистем («Облік енергії» та «Моніторинг»)

Ця діаграма відображає загальну архітектуру розробленого стенда платформи обліку електроенергії та взаємодію її основних компонентів. На ній показано, що підсистема обліку енергії включає Frontend, Simulator, Backend API та PostgreSQL, а підсистема моніторингу побудована на базі Prometheus, Grafana, node-exporter і cAdvisor, які забезпечують збір та візуалізацію метрик стану сервісів.

4.3 Модель бази даних і міграції схеми

Дані таких об'єктів, як топологія, лічильники, необроблені та перевірені показання, агрегати, сповіщення та події, зберігаються в базі даних PostgreSQL. Поточний стан схеми бази даних відстежується за допомогою міграцій Alembic, а нові версії схеми бази даних розгортаються з коду бекенду шляхом запуску команди ``alembic upgrade head`` як ізольованого процесу з обмеженим часом виконання після налаштування сервера бази даних. Для Docker Compose процес міграції виконується контейнером backend-migrate; для Kubernetes – за допомогою ресурсу Job; для Docker Swarm – на окремому етапі перед розгортанням та масштабуванням основного контейнера служби API.

Області для порівняння. Відокремлення процесу міграції бази даних надає можливість порівняти платформи з точки зору їхніх можливостей щодо попередніх умов розгортання додатків, таких як обробка залежностей між репліками бази даних та додатків, повторне виконання завдань (Jobs), визначення політик перезапуску та вказівка порядку додатків у маніфесті. У Kubernetes декларативні визначення завдань у поєднанні з умовними залежностями (що досягається, наприклад, за допомогою скрипту kubectl або хуків Helm) дозволяють задавати більш широкий набір операцій. І навпаки, у Swarm ті самі операції реалізуються шляхом виконання серії команд із менеджера Swarm або шляхом створення додаткових тимчасових служб. Діаграма ER основних сутностей бази даних показана на рисунку 4.3.

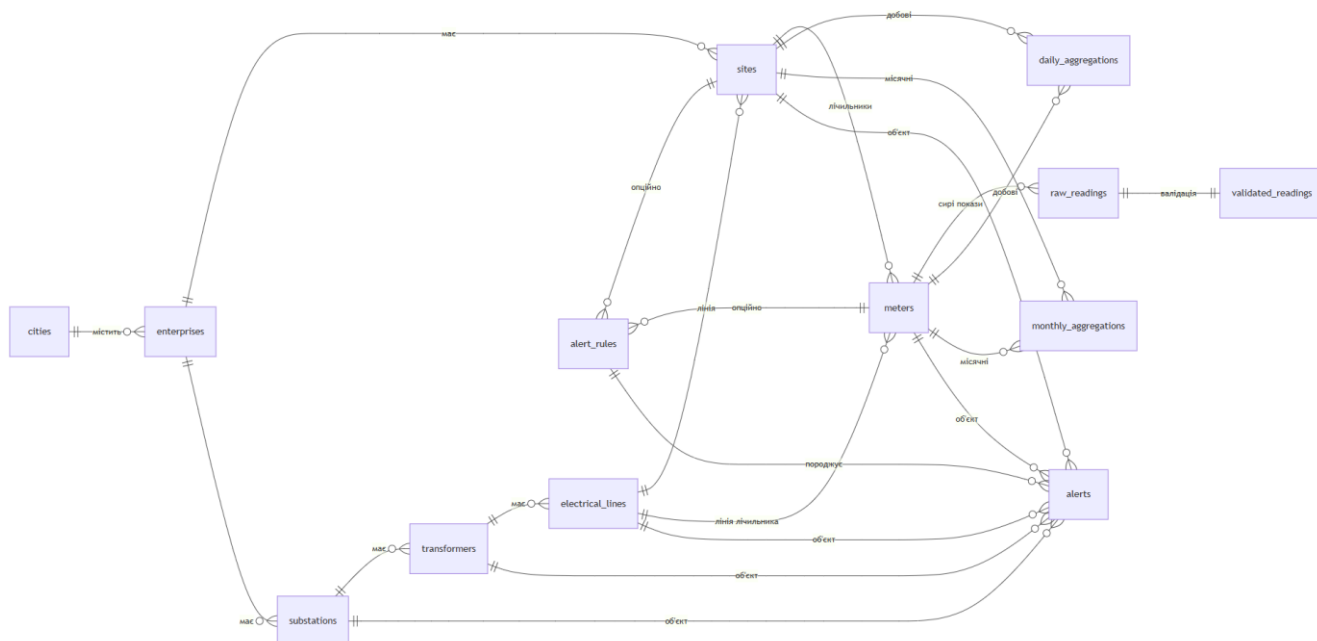


Рисунок 4.3 – ER-діаграма основних сутностей БД

Схема включає в себе декілька основних сутностей, які мають відображати відповідну галузеву інформацію. Так, сутності, такі як міста, підприємства та об'єкти, будуть представляти географічні довідкові дані, а також інформацію про підприємства та об'єкти обліку. Підстанції, трансформатори та лінії електропередачі будуть представляти топологічні дані, пов'язані з електромережею, включаючи підстанції, трансформатори та лінії електропередачі.

Об'єкт «лічильники» відповідатиме за ведення обліку лічильників, їх підключення до об'єктів та ліній, а також їхній статус (працює або неактивний). Сирі показання включатимуть неперевірені інтервальні показання, отримані або з API, або з симулятора. Перевірені показання міститимуть результати перевірки, а також інформацію про якість показань та будь-які коментарі щодо них.

Для звітів `daily_aggregations` та `monthly_aggregations` зберігатимуть минулі агрегації. Правила сповіщень визначаються шляхом переліку різних порогових значень та інших критеріїв, які призведуть до генерації сповіщень, тоді як таблиця сповіщень відстежує раніше згенеровані події/попередження, які були виявлені оператором. Знімки навантаження використовуватимуться для збереження знімка

результатів розрахунку навантаження для кожного вузла графа мережі. Інформація про користувачів зберігається в таблиці `users`, включаючи роль оператора.

Агреговані та перевірені дані, такі як `validated_readings`, `daily_aggregations` та `monthly_aggregations`, разом із посиланнями на лічильники, об'єкти та підприємства, використовуються для створення звітів на основі довідкових даних. Запити до API звітності представлені в метриках Prometheus через `report_requests_total`.

Сповіщення генеруються відповідно до критеріїв сповіщення, встановлених у `alert_rules` (наприклад, `threshold_kwh` для визначення енергетичного порогу, `window_days` для вказівки вікна спостереження та прив'язки до `site_id` або `meter_id`). Створені записи сповіщень з'являться у таблиці `alerts`, яка містить тип, рівень серйозності, опис сповіщення та, за бажанням, може містити посилання на пов'язані підстанції, трансформатори та/або лінії. Сповіщення відстежуються за допомогою метрики `alerts_generated_total`.

4.4 Платформи розгортання та їх роль у дослідженні

Реалізація підтримує три моделі виконання одного й того самого набору образів.

Docker Compose призначений для локального відтворення стенда на одному хості: оптимальний для розробки, швидкої перевірки API та UI, відладки конфігурації Prometheus. Типовий логічний ланцюг після готовності PostgreSQL: міграції - запуск `backend` - `frontend` і `simulator` - підйом моніторингу. Compose у контексті дипломної теми не є об'єктом основного порівняння з Kubernetes; він виконує роль еталонного середовища для перевірки коректності образів і конфігурацій перед винесенням їх у кластери.

Docker Swarm є першим об'єктом кластерного порівняння. Розгортання описується `stack`-файлом з `overlay`-мережею для багатовузлової взаємодії, параметрами `deploy.resources`, `update_config`, `restart_policy` для керування ресурсами

та оновленнями, reverse-проху на базі nginx для вхідного HTTP і docker secret для конфіденційних даних. Для експортерів моніторингу практично застосовується режим global (аналог DaemonSet), що забезпечує симетрію збору метрик по вузлах.

Kubernetes – другий об’єкт порівняння. Маніфести описують Namespace, Secret, PersistentVolumeClaim, Deployment і Service для прикладних компонентів і бази даних, Job для міграцій, Ingress (декларативно), HorizontalPodAutoscaler для автоматичного масштабування backend за метриками ресурсів, NetworkPolicy для моделювання сегментації трафіку, ConfigMap для конфігурацій (зокрема симулятора). Стек спостережуваності доповнюється kube-state-metrics (стан кластерних об’єктів) і metrics-server (джерело метрик для HPA).

Поруч описані платформи створюють пару Swarm – Kubernetes із відомим набором контрольованих відмінностей: наявність HPA і NetworkPolicy як перших класів API у Kubernetes проти простішої мережевої та масштабової моделі Swarm; DaemonSet-подібні експортери в обох вимагають узгодженого розгортання для справедливого порівняння метрик. Compose фіксує «нижню» межу складності для перевірки інваріантів застосунку. Порівняння моделей розгортання у дослідженні представлено в таблиці 4.2

Таблиця 4.2 – Порівняння моделей розгортання у дослідженні.

Критерій	Docker Compose	Docker Swarm	Kubernetes
1	2	3	4
Мережа	Default bridge проєкту, DNS імена сервісів	Overlay-мережа, DNS tasks.<service>	CNI кластера, Service ClusterIP, NodePort з хоста
Масштабування API	scale / профілі (ручне)	docker service scale/ update	kubectl scale, HPA(backend)

Продовження таблиці 4.2

1	2	3	4
Секрети	env / plaintext (демо)	docker secret	Secret, envFrom
Rolling-update	Recreate залежно від policy образу	update_config (parallelism, delay, rollback)	Deployment RollingUpdate, rollout restart
Ресурси CPU/RAM	Обмеження за потреби	deploy.resources limits/reservations	resources.requests/limits (важливо для HPA)
Моніторинг	Сервіси в одному compose-файлі	Сервіси в stack, той самий prometheus.yml	Namespace monitoring, ConfigMap, RBAC
Вхідний трафік	Порти напряду (:8081, :8000)	Глобальний nginx reverse-proxy	NodePort + порти kind; Ingress декларативно
Підготовчі job (міграції)	Контейнер backend-migrate	Крок деплою на manager	Job backend-migrate

4.4.1 Підсистема моніторингу та збір метрик

Підсистема моніторингу є обов'язковою частиною експериментального стенда, оскільки саме на вимірюваних характеристиках будується порівняльний аналіз оркестраторів, а не лише на описі їхніх можливостей. Prometheus виконує збір метрик з /metrics прикладних сервісів, з node-exporter (завантаження вузлів) та cAdvisor (метрики контейнерів). Grafana візуалізує часові ряди й використовує попередньо підготовлені дашборди. У Kubernetes додатково підключаються kube-state-metrics для відстеження бажаного та фактичного стану Deployment, Pod і HPA, а metrics-server забезпечує подання метрик споживання ресурсів для автоматичного масштабування.

Узгоджений стек Prometheus/Grafana/експортерів дозволяє порівнювати інфраструктурну й прикладну поведінку на одних і тих самих запитах і панелях. Різниця між платформами проявляється у кривих використання CPU та пам'яті, у часі досягнення цільової кількості реплік, у стабільності scrape під час перевантаження control plane або мережі overlay. Утім, інтерпретація NetworkPolicy у Kubernetes на навчальних кластерах з обмеженим CNI може вимагати окремого зауваження щодо повноти застосування політик – без цього методично коректніше трактувати політики як декларативну модель майбутнього production-середовища.

4.4.2 Експериментальний контур порівняння Kubernetes та Docker Swarm

Експериментальна установка реалізована в папці experiments/ у вигляді набору скриптів Bash, які використовують загальний параметр --platform=swarm|kind. Перелік реалізованих випадків включає масштабування (scale), відмовостійкість (failover), послідовне оновлення (rolling_update), навантаження НРА для Kubernetes (hpa_load) та симетричне навантаження HTTP для Swarm (swarm_load). Результати вимірювання часу зберігаються у структурованих логах зі стовпцями t*_ms.

Ініціалізація кластера здійснюється за допомогою Makefile та скриптів у папці infra/local/, які готують віртуальні машини та Swarm, створюють кластер kind, завантажують образи та розгортають маніфести. Ці дії автоматизують процес, зменшуючи участь людини у виконанні послідовності команд та роблячи можливим відтворення всієї серії експериментів. Наявність програмно керованих скриптів дає змогу перейти від якісного аналізу («на платформі було легше») до кількісної оцінки з використанням показників продуктивності часу реакції оркестратора. Блок-схема послідовності експериментів зображена на рисунку 4.4.

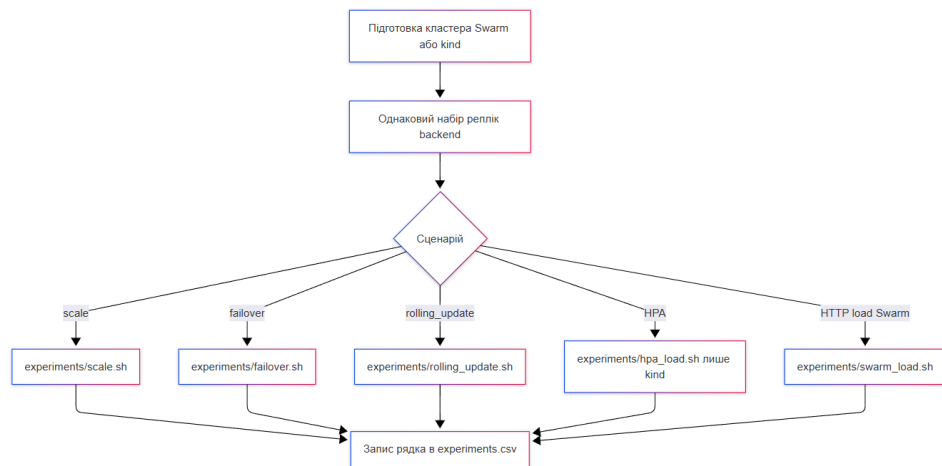


Рисунок 4.4 – Блок-схема послідовності експериментів

Висновки до розділу 4

У четвертому розділі розглядається реалізація програмного проекту розробленої системи, а також ключові аспекти її проектування, що визначають функціонування окремих компонентів. Детально пояснюються структура програмного проекту, функції різних модулів, правила їх взаємодії, а також підходи, що застосовуються при організації архітектури серверів, баз даних та контейнерів.

На етапі реалізації було побудовано мікросервісну архітектуру додатка з використанням сервера, технологій обробки даних, контейнерів та інших елементів, які згодом допоможуть розгорнути систему в різних середовищах. Це дозволило створити робочу модель програмної системи, яку можна запускати, тестувати та досліджувати емпірично.

Забезпечуючи необхідну функціональність та узгодженість між елементами, представлені рішення дозволяють у разі потреби масштабувати або модифікувати систему. Поточне програмне рішення закладає основу для наступного кроку – розгортання, тестування та оцінки системи в середовищах Kubernetes та Docker Swarm.

5 ПОРІВНЯННЯ ПЛАТФОРМ ОРКЕСТРАЦІЇ КОНТЕЙНЕРІВ

У цьому розділі виконується експериментальне порівняння платформ оркестрації контейнерів Kubernetes і Docker Swarm з використанням розробленого програмного прототипу. Docker Swarm забезпечує кластерне керування сервісами на основі бажаного стану, реплік і ролей manager/worker, тоді як Prometheus і Grafana застосовуються для збору, зберігання та візуалізації метрик, що дає змогу кількісно оцінити поведінку системи в різних умовах навантаження.

5.1 Опис експериментів

Експериментальний цикл у межах дипломного дослідження було сформовано з метою отримання відтворюваних кількісних оцінок поведінки платформ оркестрації контейнерів Docker Swarm і Kubernetes за умови збереження незмінними контейнерних образів, складу прикладних сервісів і засобів моніторингу. Такий підхід відповідає принципу «все інше рівне», за якого порівняння оркестраторів виконується не в абстрактному середовищі, а на одному й тому самому прикладному навантаженні, із максимально однаковими вхідними умовами для обох платформ. У результаті фіксовані відмінності в часових показниках та характері реакції системи можуть бути інтерпретовані як наслідок особливостей механізмів оркестрації, а не зміни логіки застосунку чи відмінностей у контейнерних збірках.

Об'єктом вимірювання в експериментах обрано сервіс backend, що реалізує REST API платформи обліку електроенергії. Такий вибір є методично виправданим, оскільки саме backend-сервіс виступає центральною точкою прикладної взаємодії, опрацьовує HTTP-запити, взаємодіє з базою даних і є найбільш придатним компонентом для фіксації часових характеристик, пов'язаних

із запуском реплік, відновленням після відмови, перезапуском під час оновлення та реакцією на навантаження. Frontend, simulator, PostgreSQL, Prometheus і Grafana в цьому підрозділі розглядаються як елементи оточення, необхідні для забезпечення типового робочого контуру платформи, але не як основні цілі кількісного вимірювання.

Методологічно експерименти побудовано як серію сценаріїв, у яких змінюється лише керований фактор, пов'язаний із діями оркестратора: масштабування, відновлення після аварійного завершення, rolling update або реакція на навантаження. В усіх сценаріях реєструються однакові за змістом часові мітки, що уможливорює подальшу агрегацію результатів, обчислення узагальнених характеристик та побудову коректних порівнянь між платформами. Окрему увагу приділено повторюваності: кожен блок сценаріїв у планах `plan_kind.sh` та `plan_swarm.sh` виконується по 5 разів, що дозволяє знизити вплив випадкових коливань у локальному стенді та підготувати матеріал для наступного підрозділу з результатами.

Структура експериментального циклу включає як симетричні сценарії, у яких поведінка Docker Swarm і Kubernetes порівнюється безпосередньо, так і асиметричні сценарії, у яких оцінюються платформно-специфічні властивості. До першої групи належать масштабування, failover і rolling update. До другої групи належать сценарії навантаження `hpa_load` для Kubernetes та `swarm_load` для Docker Swarm, які не утворюють прямої числової пари, оскільки Kubernetes має нативний механізм Horizontal Pod Autoscaler, тоді як у Docker Swarm відсутній аналогічний вбудований механізм автоматичного горизонтального масштабування.

5.1.1 Експериментальний стенд і умови проведення

Експериментальна частина дослідження виконувалася у двох середовищах, кожне з яких відтворювало окрему модель оркестрації. Перше середовище побудовано на базі Docker Swarm і розгорнуто на трьох віртуальних машинах

Multipass. Для прикладного розгортання використовується стек energy, а керування Docker-командами виконується через `multipass exec` на вузлі `node-1`, який виконує функції керівного вузла кластера. Друге середовище побудовано на базі Kubernetes у локальному кластері `kind`, де використовується кластер `energy`, а ізоляція ресурсів прикладної системи організована в `namespace energy`. Для виконання команд застосовується контекст `kubectl --context kind-energy -n energy`, тоді як контекст у Kubernetes загалом виступає способом фіксації цільового кластера та `namespace` у конфігурації `kubectl`.

Обидва середовища обслуговують одну й ту саму контейнеризовану платформу обліку електроенергії. Прикладна частина включає backend на FastAPI, frontend на React, `simulator` для генерації подій або модельних значень, PostgreSQL для зберігання даних, а також Prometheus і Grafana для спостереження за технічними параметрами системи. Використання однакових Docker-образів у двох оркестраторах забезпечує незмінність програмного навантаження, а отже, робить коректним подальше зіставлення часових характеристик. Моніторингова підсистема залишається спільною за призначенням: Prometheus збирає часові ряди метрик, а Grafana надає візуальні дашборди для оперативного спостереження.

Окремою умовою Kubernetes-середовища є наявність `metrics-server`, який необхідний для роботи Horizontal Pod Autoscaler. Його підготовку було інтегровано в процес розгортання через `infra/local/kind/up.sh`, а сам НРА описано у файлі `deploy/k8s/hpa.yaml`. Це відповідає загальному принципу НРА у Kubernetes, де автоматичне масштабування робочих навантажень виконується на основі спостережуваних метрик, зокрема завантаження CPU. У Docker Swarm порівнюваний за ідеєю сценарій навантаження виконується без `autoscale`, що відображає фактичну функціональну асиметрію між платформами. Для зручності опис експериментальних середовищ доцільно подати в компактному вигляді в таблиці 5.1.

Таблиця 5.1 – Опис експериментальних середовищ

Середовище	Оркестратор	Інфраструктура	Ідентифікатор розгортання	Спосіб керування
Середовище 1	Docker Swarm	3 VM Multipass	stack energy	multipass exec на node-1 для Docker-команд
Середовище 2	Kubernetes (kind)	локальний кластер kind	cluster energy, namespace energy	kubectl --context kind-energy -n energy

Додатково для оперативного спостереження за перебігом сценаріїв використовуються Grafana-дашборди `replicas-k8s.json`, `replicas-swarm.json`, `infra-metrics.json` та `energy-overview.json`. Вони не є первинним джерелом кількісної фіксації результатів, однак виконують важливу допоміжну функцію: дають змогу візуально контролювати зміну кількості реплік, стан інфраструктурних ресурсів та загальну поведінку системи під час виконання прогонів. Водночас джерелом офіційно зафіксованих результатів для подальшого аналізу виступає не дашборд, а файловий протокол, описаний у наступному підпункті.

5.1.2 Протокол реєстрації результатів

З метою забезпечення відтворюваності та приведення вихідних даних у стан, придатний для подальшої автоматизованої обробки, було застосовано стандартизований підхід до ведення журналів із використанням файлів CSV. Головний файл журналу знаходиться в папці `infra/local/out/experiments.csv`, де реєструються всі запуски всіх сценаріїв. Крім того, щоб мати можливість відстежувати конкретний запуск і відрізнити аномалії від звичайних подій, для кожного запуску створюється окремий файл походження, який зберігається в папці `experiment_runs/` і дозволяє при необхідності перевірити окремі моменти часу.

Структура кожного запису журналу містить стандартизовану інформацію щодо запуску сценарію та складається з таких полів, як `ts_iso`, `platform`, `scenario`, `service`, `from_n`, `to_n`, `t_request_ms`, `t_first_ready_ms`, `t_all_ready_ms`, `t_done_ms`, `notes`. Поле `ts_iso` фіксує мітку часу виникнення події у стандартизованому форматі; `platform` позначає цільовий оркестратор, що використовується; `scenario` – це поточний сценарій; `service` – це назва вимірюваної служби; `from_n` та `to_n` описують процес переходу між станами (наприклад, зміна кількості реплік тощо). Параметри часу дозволяють нам по-різному інтерпретувати кожен крок у рамках сценарію: надсилання запиту на управління для ініціювання дії, час до появи першої готової репліки після дії, час до повної готовності цільового стану, відповідно, та загальний час, необхідний для виконання сценарію.

Кожна точка часу відбирається з інтервалом опитування 0,2 секунди з максимальним обмеженням у 300 секунд для завершення виконання сценарію. Така параметризація дозволяє отримати необхідний обсяг інформації про зміни стану, не приводячи до надмірного очікування у разі незавершення виконання сценарію або проблем зі стабільністю в середовищі. Відтворюваність досягається завдяки тому, що кожен блок у файлах `plan_kind.sh` та `plan_swarm.sh` повторюється п'ять разів. Подальше агрегування результатів та обчислення їх середнього значення можна виконати за допомогою скрипта `python experiments/aggregate_runs.py`. Відповідність між основними сценаріями та полями журналу представлено в таблиці 5.2.

Таблиця 5.2 – Відповідність сценарію полям CSV для аналізу

Сценарій	Основний файл виконання	Ключові поля CSV	Призначення метрики
E1/E2 scale	scale.sh	t_first_ready_ms, t_all_ready_ms	Оцінка швидкості початку масштабування та досягнення цільового числа готових реплік
E3 failover	failover.sh	t_first_ready_ms, t_all_ready_ms	Оцінка часу появи заміни після збою та повного відновлення бажаного стану
E4 rolling_update	rolling_update.sh	t_done_ms	Оцінка тривалості контрольованого перезапуску або rolling update
E5 hpa_load	hpa_load.sh	t_request_ms, t_first_ready_ms, t_all_ready_ms, t_done_ms, notes	Фіксація реакції НРА на прикладне навантаження та часу повернення до базового стану
E6 swarm_load	swarm_load.sh	t_request_ms, t_done_ms, notes	Фіксація поведінки сервісу під тим самим навантаженням без autoscale

За рахунок такої уніфікації протоколу стає можливим не лише порівнювати окремі запуски між собою, а й виконувати подальшу агрегацію без ручного втручання. Саме це забезпечує перехід від окремих прогонів до систематизованих результатів.

5.1.3 Опис сценаріїв експериментів

Ці сценарії розроблені з метою охоплення всіх основних типів операцій оркестратора, що мають практичне значення для контейнеризованої платформи обліку електроенергії: масштабування сервісів, відновлення після збою, послідовне оновлення, балансування навантаження та динаміка змін кількості реплік. Усі згадані сценарії будуть виконуватися автоматично за допомогою скриптів Bash scale.sh, failover.sh, rolling_update.sh, hpa_load.sh, swarm_load.sh та run_all.sh,

доповнених додатковими допоміжними функціями, доступними у скрипті `lib/common.sh`.

Сценарії E1 та E2 розглядають поведінку оркестратора у відповідь на зміни кількості реплік сервісу бекенду. Хоча в Docker Swarm операція масштабування виконується за допомогою команди `docker service update --replicas`, а в Kubernetes – за допомогою команд `kubectl scale`, ці незначні синтаксичні відмінності не впливають на суть сценарію – отримання оркестратором команди управління для зміни кількості екземплярів сервісу бекенду від початкового до цільового значення.

У цьому сценарії основними часовими метриками є `t_first_ready_ms` та `t_all_ready_ms`. Перша з них відображає час між відправленням команди масштабування та отриманням інформації про першу новостворену готову репліку як час реакції оркестратора на команду масштабування. Друга метрика вимірює час, необхідний для того, щоб усі репліки, зазначені в цільовому стані, стали повністю готовими. Це означає, що перехід до нового масштабу завершено. Як для Docker Swarm, так і для kind ці метрики утворюють симетричну пару, оскільки в обох випадках відбувається одна й та сама операція.

Варто зазначити, що конкретні зміни реплік відрізняються для Swarm і kind. А саме, тоді як Swarm використовує перехід від 2 до 5, kind застосовує перехід масштабу реплік від 4 до 8. Однак ця відмінність нічого не змінює для самого сценарію, але впливає на фактичні вимірювання часу через різний масштаб.

Сценарій E3 перевіряє здатність оркестраторів відновлюватися після відмови окремого екземпляра служби. Тому як попередня умова має бути наявність щонайменше двох реплік. У Docker Swarm сценарій відмови передбачає виконання команди `docker kill` на одному з екземплярів завдання служби, тоді як у KIND виконується команда `kubectl delete pod --force` для примусового видалення под. На рівні сценарію ці налаштування передбачають один і той самий сценарій – виведення з ладу одного з екземплярів сервісу бекенду з подальшим моніторингом реакції системи на цей інцидент.

Для цього сценарію показники, що нас цікавлять, такі самі, як і вище, і також позначаються в журналі як `t_first_ready_ms` та `t_all_ready_ms`, але їх потрібно інтерпретувати з точки зору виходу з ладу та відновлення. А саме, перший показник відображає час отримання інформації про перший випадок відновлення служби, а другий – час відновлення цільової кількості реплік. Така інтерпретація допомагає розрізнити реакцію оркестратора на збій та час відновлення.

Цей сценарій є методологічно дуже важливим, оскільки розглянуті часові показники дозволяють оцінити не тільки час реакції, але й можливість підтримувати роботу сервісів додатка навіть у разі відмови контейнера. Ця властивість є критично важливою для платформи обліку електроенергії, оскільки навіть відмова локального екземпляра сервісу бекенду не повинна спричиняти відмову API.

Сценарій E4 розглядає послідовне оновлення служби. Як і попередній, він вимагає щонайменше двох реплік для реалізації. У Docker Swarm послідовне оновлення служби можна ініціювати за допомогою команди `docker service update -force`, тоді як у Kubernetes це передбачає перезапуск Deployment шляхом ініціювання нового розгортання за допомогою команди `kubectl rollout restart`.

На відміну від сценаріїв масштабування та відновлення після збою, де корисними були часові мітки для отримання інформації про готові екземпляри, сценарій послідовного оновлення можна охопити одним основним показником – `t_done_ms`. Він відповідає загальному часу послідовного перезапуску служби та має безпосереднє значення для додатка обліку електроенергії, оскільки в ході рутинних операцій конфігурації можна розгортати заново, а служби оновлювати без шкоди для стабільності REST API.

Сценарій E5 застосовується лише для Kubernetes і призначений для перевірки того, як Horizontal Pod Autoscaler реагує на HTTP-навантаження додатка. Зокрема, параметри НРА встановлені на `min=2`, `max=8`, а цільове навантаження дорівнює 75%. Таким чином, сценарій базується на стандартному підході до горизонтального масштабування в Kubernetes. Щоб зосередити навантаження на сервісі бекенду,

симулятор робочого навантаження призупиняється, щоб виключити фонові процеси, і генерує навантаження за допомогою 20 одночасних викликів `curl` до кінцевої точки сайту `/sites` протягом 120 секунд. Після завершення генерації навантаження настає фаза зменшення масштабу. Вона триває до охолодження та повернення системи до низької кількості реплік. Загальний час очікування може становити до 600 секунд.

Під час виконання сценарію відбувається кілька етапів: підготовка, навантаження, масштабування вгору, зупинка, масштабування вниз. На етапі підготовки забезпечуються початкові стани сервісу та НРА. Далі йде етап навантаження, під час якого до додатка подається контрольоване навантаження. Наступна фаза фіксує реакцію НРА на зростаюче навантаження та створення додаткових реплік. Після завершення штучного генерування навантаження останнім етапом є моніторинг швидкості зменшення масштабу. Хоча деякі інтервали цього сценарію перетинаються з часовими діапазонами інших експериментів, цей сценарій слід розглядати як самостійний експеримент на платформі, а не як прямий чисельний аналог відповідного сценарію `Swarm`.

Нарешті, сценарій `E6` розглядається як версія `Docker Swarm`, аналогічна попередньому сценарію. Експеримент передбачає використання того самого `HTTP`-навантаження для тестування продуктивності бекенду в умовах, коли можливості горизонтального масштабування для нього не передбачені на рівні ядра. Саме тому аналіз охоплює не те, як реагує автоскалер, а те, як система втримується в робочому стані в умовах статичного масштабування.

Варто зазначити, що, на відміну від попереднього сценарію `E5`, цей не є прямим числовим аналогом попереднього сценарію. Насправді експеримент з `Kubernetes` спрямований на дослідження поведінки просунутої системи з працюючим НРА та автоматичним регулюванням кількості реплік відповідно до метрик ЦП. У випадку з `Docker Swarm` немає еквівалентного вбудованого інструменту автомасштабування, тому цей тест може надати інформацію лише щодо того, як система реагує на навантаження в межах фіксованої конфігурації

реплік. Тому подальший аналіз результатів для сценаріїв E5 та E6 слід проводити окремо, зосередившись на функціональних відмінностях, а не на прямій числовій аналогії. Класифікація сценаріїв представлена в таблиці 5.3.

Таблиця 5.3 – Класифікація сценаріїв

Група сценаріїв	Позначення	Платформи	Тип зіставлення
Масштабування	E1–E2 scale	Swarm, kind	Пряме симетричне порівняння
Відмовостійкість	E3 failover	Swarm, kind	Пряме симетричне порівняння
Контрольований перезапуск	E4 rolling_update	Swarm, kind	Пряме симетричне порівняння
Автоматичне масштабування під навантаженням	E5 hpa_load	лише kind	Асиметричний платформний сценарій
Навантаження без autoscale	E6 swarm_load	лише Swarm	Асиметричний платформний сценарій

5.2 Результати експериментального дослідження

5.2.1 Обробка даних

Результати були отримані на основі лог-файлів, що містяться у файлі `infra/local/out/experiments.csv`, записаних під час збору даних. Аналіз даних проводився за допомогою скрипта `experiments/aggregate_runs.py`, де медіана виступала основним статистичним показником для кожної ситуації. Для кожного виміру в наведеній таблиці вказано кількість прогонів (n), що використовувалася для їхнього узагальнення. Важливо враховувати ці результати як такі, що

стосуються локального середовища, а не як загальні характеристики виробничих середовищ.

5.2.2 Результати симетричних сценаріїв

У таблиці 5.4 наведено медіанні результати симетричних сценаріїв, у яких для Docker Swarm і kind виконувалась однакова за змістом дія над сервісом backend.

Таблиця 5.4 – Класифікація сценаріїв

Сценарій	Метрика	Docker Swarm	Kubernetes
Scale up (2→5/4→8)	t_first_ready	17,70 с	13,70 с
Scale up (2→5/4→8)	t_all_ready	18,60 с, n = 9	14,20 с, n = 9
Scale down (5→2/8→4)	завершення переходу	0,55 с	0,33–0,36 с
Failover R=2	drop	705 мс, n = 5	285 мс, n = 3
Failover R=2	recovery	21,80 с	13,60 с
Rolling update R=2	t_done	85,70 с, n = 5	13,40 с, n = 3

За наведеними даними Kubernetes у симетричних сценаріях загалом демонструє менші часові затримки. Найбільш помітна різниця спостерігається у сценарії rolling update, тоді як у масштабуванні вона є меншою. Для failover також фіксується швидша реакція та відновлення сервісу в середовищі Kubernetes.

Як доповнення слід зазначити, що для Kubernetes спостерігалися також результати failover і rolling update при R=3, однак основне порівняння в цьому підрозділі збережено для конфігурації R=2, оскільки саме вона утворює базову симетричну пару для двох платформ. В таблиці 5.5 показано порівняння за симетричними сценаріями.

Таблиця 5.5 – Узагальнене порівняння за симетричними сценаріями

Сценарій	Ключовий показник	Відносний результат
Scale up	t_all_ready	Swarm приблизно у 1,3 раза повільніше
Failover	drop	Swarm приблизно у 2,5 раза повільніше
Failover	recovery	Swarm приблизно у 1,6 раза повільніше
Rolling update	t_done	Swarm приблизно у 6,4 раза повільніше

Зведені дані підтверджують стабільний напрям різниці на користь kind. У межах локального експериментального стенда це дає підстави вважати, що Kubernetes швидше виконує критичні операції життєвого циклу сервісу backend.

5.3.3 Результати масштабування під навантаженням

Сценарії hpa_load і swarm_load подано окремо, оскільки вони не утворюють прямої числової пари. У Kubernetes досліджується реакція системи з Horizontal Pod Autoscaler, який автоматично змінює кількість реплік відповідно до навантаження, тоді як у Docker Swarm у межах цього стенда навантаження обробляється без нативного autoscale. В таблиці 5.6 показані результати цього тестування.

Таблиця 5.6 – Результати сценаріїв навантаження

Сценарій	Платформа	t_first_scale	t_max_scale	t_done	Peak реплік	n
hpa_load	kind	≈ 30,8 с	≈ 61,2 с	≈ 348,2 с	8	5
swarm_load	Docker Swarm	—	—	≈ 121,3 с	2	5

Наведені результати відображають різні режими роботи платформ під навантаженням. Для kind зафіксовано повний цикл автоматичного масштабування, тоді як для Swarm – роботу сервісу у фіксованій конфігурації реплік. Тому ці

значення слід інтерпретувати як функціональне зіставлення, а не як пряме парне порівняння.

За сукупністю сценаріїв E1–E4 можна констатувати, що kind стабільно демонструє швидшу реакцію в операціях масштабування, відновлення після відмови та rolling update. Найбільш чутливим до відмінностей платформ виявився саме сценарій rolling update.

Сценарії E5–E6 показують вже не стільки різницю в часі, скільки різницю в функціональних можливостях платформ. Kubernetes у конфігурації kind підтримує нативне автоматичне горизонтальне масштабування через НРА, тоді як Docker Swarm у межах досліджуваного стенда виконує обробку навантаження без автоматичної зміни кількості реплік.

Для візуального супроводу результатів використовувалися дашборди Grafana replicas-k8s, replicas-swarm, infra-metrics та energy-overview. Grafana призначена для побудови інформаційних панелей і відображення пов'язаних метрик у зручному для аналізу вигляді, а Prometheus забезпечує збір і зберігання метрик у форматі часових рядів.

Дашборди використовувалися як ілюстративний інструмент для спостереження за зміною кількості реплік, реакцією системи на навантаження та поведінкою сервісу під час failover і rolling update. Основним джерелом числового аналізу при цьому залишався CSV-журнал експериментів. Кількість доступних сервісів і ресурсів при проведенні тесту на масштабування для Docker Swarm зображена на рисунку 5.1. та 5.2.

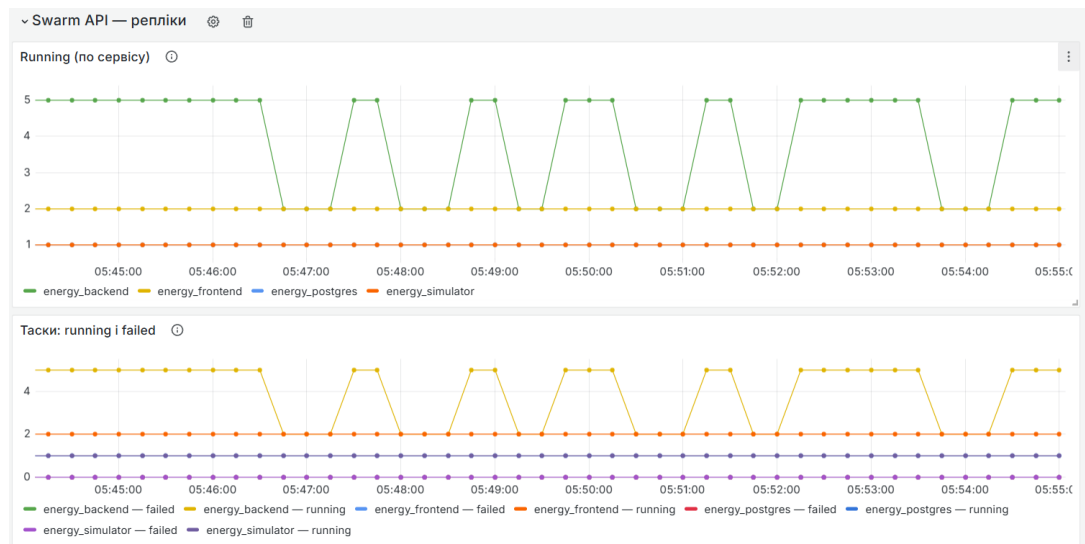


Рисунок 5.1 – Кількість реплік системи при масштабуванні Swarm

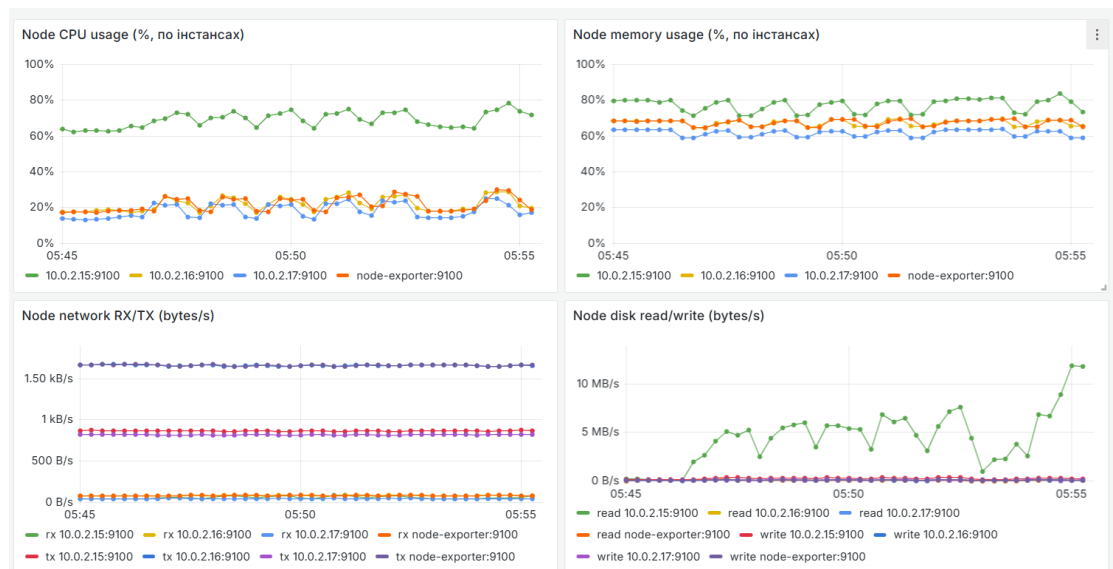


Рисунок 5.2 – Кількість ресурсів системи при масштабуванні Swarm

Кількість доступних сервісів і ресурсів при проведенні тесту на масштабування для Kubernetes зображена на рисунку 5.3. та 5.4.

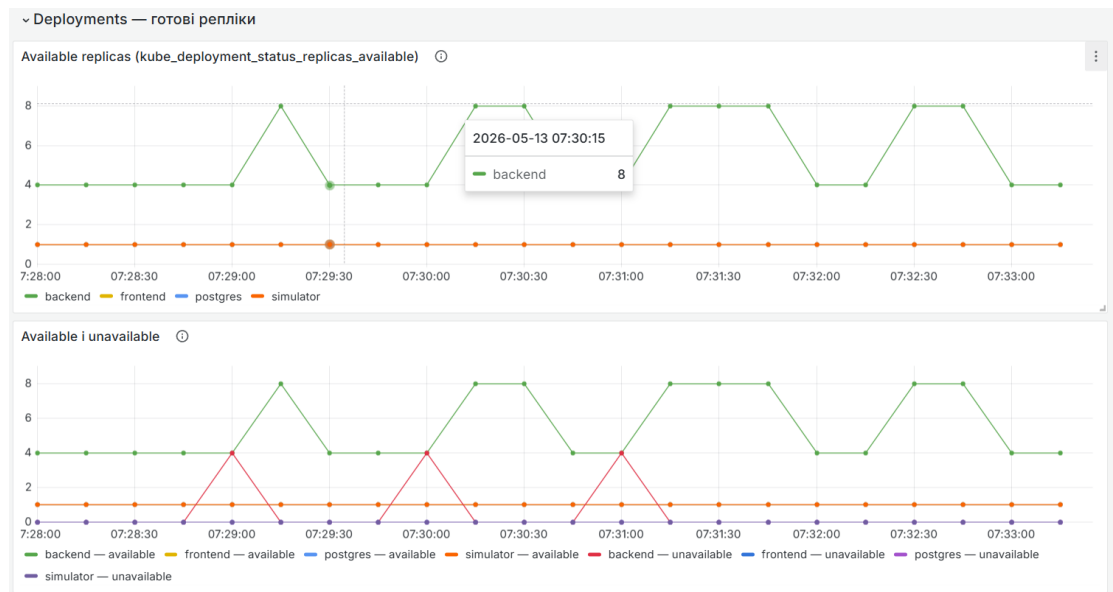


Рисунок 5.3 – Кількість реплік системи при масштабуванні Kubernetes

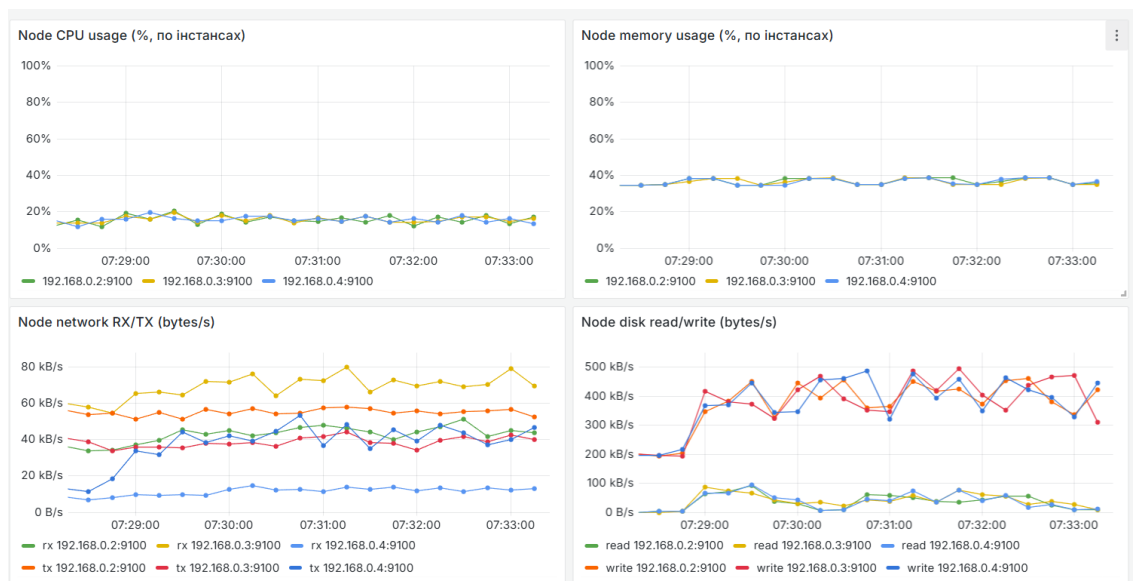


Рисунок 5.4 – Кількість реплік системи при масштабуванні Kubernetes

На основі отриманих медіанних результатів можна попередньо стверджувати, що в межах локального стенда kind демонструє кращі часові характеристики у симетричних сценаріях. Найбільш виражена перевага спостерігається під час rolling update, тоді як у масштабуванні відмінність є помірною. Окремо встановлено, що сценарії `hpa_load` і `swarm_load` відображають

різні функціональні режими платформ. Найбільша відносна перевага kind – у rolling update $\approx 6,4$ разів; найменша – у scale up $\approx 1,3$ разів.

5.3 Порівняльна характеристика платформ оркестрації контейнерів

Порівняльний аналіз Kubernetes та Docker Swarm у межах дипломного дослідження виконується у двох взаємодоповнювальних площинах: функціонально-архітектурній та експериментальній. У першому випадку увага зосереджується на механізмах, що визначають придатність платформи до розгортання контейнеризованих застосунків у задачах цифровізації енергетичних систем. У другому випадку враховуються кількісні результати локального стенда, отримані під час виконання сценаріїв масштабування, failover, rolling update та навантаження для сервісу backend.

Практичний контекст дослідження представлено стендом платформи обліку електроенергії, до складу якого входять backend на FastAPI, frontend, simulator потоку показань лічильників, PostgreSQL, Prometheus і Grafana. Використання симулятора та endpoint POST /readings дозволяє наблизити прикладне навантаження до metering- та AMR-подібних сценаріїв імітації окремих властивостей реальної системи обліку електроенергії. Для обох платформ використовувалися однакові Docker-образи, а самі експерименти виконувалися Bash-скриптами з каталогу experiments/ із параметром --platform=swarm|kind, що забезпечило однакову логіку постановки сценаріїв.

У контексті цифровізації енергетики вибір оркестратора пов'язаний із кількома практично значущими вимогами: здатністю витримувати піки навантаження під час формування звітності або масового надходження показань, підтримкою безперервного приймання даних, можливістю оновлення сервісів без простою, розмежуванням трафіку між OT- та IT-рівнями, а також наявністю засобів спостережуваності на базі Prometheus і Grafana. Саме тому подальше порівняння

будується не як загальний огляд платформ, а як аналіз їх релевантності для прикладної інфраструктури енергетичних інформаційних систем.

5.3.1 Функціональне порівняння

Функціональне порівняння дає можливість співвіднести властивості Kubernetes і Docker Swarm із типовими вимогами до експлуатації цифрових сервісів у енергетиці. До ключових критеріїв належать засоби керування кластером, рівень декларативності, наявність механізмів автоматичного масштабування, мережевої сегментації, оновлення сервісів, роботи з секретами, публікації застосунків і моніторингу. Таблиця 5.7 показує функціональне порівняння оркестраторів.

Таблиця 5.7 – Функціональне порівняння Kubernetes та Docker Swarm

Критерій	Kubernetes	Docker Swarm	Значення для energy/metering
1	2	3	4
Control plane	Повноцінний control plane з окремими керівними компонентами	Спрощена модель керування через Swarm manager	Kubernetes краще придатний для складних багатокомпонентних платформ
Декларативність	Декларативне керування ресурсами через YAML-об'єкти	Простіший підхід на рівні сервісів і stack	Kubernetes зручніший для відтворюваних конфігурацій
Автомасштабування	Підтримується НРА для автоматичної зміни кількості реплік	Нативний НРА відсутній	Для пікових навантажень Kubernetes має функціональну перевагу
Мережева сегментація	Підтримується NetworkPolicy для керування мережевим доступом	Мережева сегментація	Підтримується NetworkPolicy для керування мережевим доступом

Продовження таблиці 5.7

1	2	3	4
Rolling update / rollback	Розвинуті механізми rollout і rollback	Базова підтримка rolling update	Kubernetes дає вищу керованість релізами
Secrets / configs	Secret і ConfigMap як окремі об'єкти	Підтримуються Docker secrets	Secrets / configs
Публікація сервісів	Service, Ingress та інші моделі маршрутизації	Публікація портів і reverse проху; у стенді використано nginx	Kubernetes має ширші можливості експонування сервісів
Managed cloud	Широка підтримка з боку cloud-провайдерів	Обмежене поширення в managed cloud	Kubernetes краще масштабується за межі локального стенда
Складність деплою	Вища, потребує більшої підготовки	Нижча, швидший старт	Docker Swarm простіший для швидкого впровадження
Моніторинг стану	Доступний kube-state-metrics для метрик стану об'єктів	Моніторинг стану	Доступний kube-state-metrics для метрик стану об'єктів
Модель мережі	Гнучка, придатна до політик доступу	Простіша overlay-модель	Kubernetes краще підходить до ізольованих середовищ
Екосистема	Розвинена cloud-native екосистема	Легша, але значно вужча екосистема	Kubernetes має вищий потенціал розвитку інфраструктури

Функціональне зіставлення показує, що Docker Swarm орієнтований на простоту початкового впровадження й нижчий адміністративний поріг, тоді як Kubernetes надає ширший набір механізмів для побудови масштабованих, сегментованих і керованих цифрових платформ. Отже, відмінність між платформами полягає не лише в наборі команд або способі розгортання, а в самій моделі експлуатації сервісів у середовищі зі складнішими вимогами.

5.3.2 Порівняння у контексті цифровізації енергетики

Для цифрових сервісів енергетичної галузі ключовими є не лише формальні характеристики оркестратора, а і його здатність підтримувати типовий прикладний цикл: стабільне приймання показань, масштабування backend-сервісів у періоди пікового доступу, безпечну публікацію API, контроль мережевої взаємодії та спостережуваність системи. У реалізованому стенді ці аспекти проявляються через роботу backend на FastAPI, simulator потоку показань, PostgreSQL та спільний стек Prometheus/Grafana.

Зіставлення функціональних властивостей із результатами локального експерименту показує, що Kubernetes поєднує дві групи переваг: ширші архітектурні засоби керування кластером і кращі часові характеристики в досліджених сценаріях життєвого циклу сервісу. Зокрема, підтримка НРА дозволяє автоматично масштабувати сервіси під навантаженням, а NetworkPolicy забезпечує формальний механізм керування мережевим доступом між компонентами. Для енергетичних платформ це означає вищу придатність до середовищ, де одночасно важливі еластичність, сегментація та контрольоване оновлення сервісів.

Водночас Docker Swarm зберігає значення як легший інструмент оркестрації, придатний для менших інфраструктур, локальних стендів і середовищ, де пріоритетом є простота впровадження. У контексті цього дослідження це особливо помітно на рівні архітектурної моделі: Swarm має нижчий поріг стартового налаштування, але поступається Kubernetes у гнучкості автоматичного масштабування, мережевої ізоляції та глибині інтеграції з cloud-native інструментами.

Таким чином, у межах практичного стенда платформи обліку електроенергії було зафіксовано не лише кількісну різницю в швидкості виконання окремих операцій, а й відмінність у самій експлуатаційній моделі платформ. Kubernetes орієнтований на складніший, але функціонально повніший життєвий цикл

контейнеризованого застосунку, тоді як Docker Swarm реалізує спрощений варіант оркестрації, достатній для частини базових сценаріїв.

Порівняння цих двох рішень показує, що між Docker Swarm і Kubernetes існує значна різниця у функціональних можливостях. Зокрема, Docker Swarm відрізняється простішою стратегією розгортання та легшим доступом до системи, тоді як Kubernetes надає потужні інструменти, що забезпечують автоматичне масштабування, поділ мережі та інтеграцію з системами моніторингу.

Кількісні дані, зібрані за допомогою локального тестового середовища, підтверджують перевагу Kubernetes у плані можливостей масштабування, відмовостійкості та послідовних оновлень. Слід підкреслити, що цей розділ не містить жодних рекомендацій щодо вибору інструменту оркестрування, оскільки така інформація наводиться лише в наступному розділі.

5.4 Рекомендації щодо вибору платформи оркестрації для задач цифровізації

5.4.1 Методичні підстави рекомендацій

Рекомендації щодо вибору платформи оркестрації в межах цієї роботи сформульовано на основі двох груп підстав: функціонального порівняння Kubernetes і Docker Swarm та кількісних результатів локального експериментального стенда. У попередніх підрозділах було встановлено, що Kubernetes має ширші можливості автоматичного масштабування, мережевої сегментації та інтеграції з моніторингом, тоді як Docker Swarm характеризується нижчим порогом входження та простішою моделлю розгортання. На локальному стенді для сервісу backend також зафіксовано швидше відновлення після відмови в kind, коротший rolling update та наявність повного циклу автоматичного горизонтального масштабування через HPA.

Водночас рекомендації не повинні тлумачитися як універсальне правило, відірване від конкретної архітектури системи. Вибір оркестратора є частиною загального проектування цифрової платформи, а не самоціллю. Для енергетичних інформаційних систем значення мають не лише технологічні уподобання команди, а й профіль навантаження, вимоги до затримок, критичність безперервної роботи, характер оброблюваних даних, потреба в stateful-компонентах, рівень сегментації ОТ/ІТ та перспективи подальшого масштабування. Саме тому одна й та сама платформа може бути доцільною для одного класу систем і недоцільною для іншого.

Крім того, під час формулювання рекомендацій необхідно враховувати обмеження емпіричної бази. Отримані в роботі часові характеристики зафіксовано на локальному стенді kind/Multipass, а отже вони корисні для порівняльної інтерпретації, але не повинні переноситися в production як гарантовані SLA-показники. Особливо це стосується систем із жорсткими вимогами до затримок або регламентованою промисловою безпекою, де архітектурний вибір визначається не лише механізмами оркестрації, а й галузевими нормативами, політиками кіберзахисту та вимогами до сертифікованого обладнання.

5.4.2 SCADA-системи

Важливо зазначити, що системи SCADA в енергетичній галузі є досить різноманітними. Тому доцільно сформулювати рекомендації щодо SCADA принаймні на двох рівнях: ядро SCADA та периферія SCADA. Перш за все, поговоримо про ядро SCADA. Компоненти ядра SCADA входять до критичного технологічного ланцюга; вони працюють в умовах жорсткого або майже жорсткого режиму реального часу з дуже суворими обмеженнями щодо детермінованих затримок. Крім того, системи такого типу працюють у спеціалізованому ОТ-середовищі. Тому для ядра SCADA надзвичайно важливо забезпечити надзвичайно низьку затримку, контрольовану роботу під навантаженням, стабільність

середовища виконання та дотримання певних правил кібербезпеки, визначених у стандарті IEC 62443. У цьому випадку було б помилкою розглядати Kubernetes або Docker Swarm як найкращий інструмент для ядра SCADA.

Для цього завдання набагато більше підходять «голі» сервери, віртуальні машини або стандартні платформи промислової автоматизації (від таких виробників, як Schneider, ABB, Siemens тощо), ніж оркестрування контейнерів. Такий вибір пояснюється тим, що Kubernetes/Docker додає ще один рівень абстракції до технологічного стеку, але не пропонує жодних істотних переваг у випадку сфери ОТ та ускладнює процес розробки та валідації. Отже, ядро SCADA вимагає уникнення Kubernetes або Docker та надання переваги використанню передбачуваної інфраструктури з відповідною моделлю управління.

Периферійні компоненти SCADA, такі як шлюзи OPC UA, архіватори, агрегатори даних, служби інтеграції або проміжні буферні вузли для обміну даними між ОТ- та ІТ-рівнями, демонструють більш стабільне навантаження та меншу потребу в механізмах автомасштабування. Крім того, такі компоненти SCADA частіше представляють типові інфраструктурні служби, залежності яких визначені заздалегідь. Отже, для цієї категорії достатньо застосувати Docker Swarm як платформу з прийнятним рівнем кластеризації без додаткових ускладнень з точки зору розгортання та обслуговування. Крім того, можливість автомасштабування може бути непотрібною для цих компонентів SCADA і навіть може спричинити певні проблеми через непередбачуваний обмін даними або конфігурацію.

Kubernetes можна використовувати для периферійних компонентів SCADA лише за умови наявності команди щонайменше з трьох експертів DevOps або SRE, які мають навички та компетенції, необхідні для впровадження необхідного рівня політик безпеки, спостережності, ізоляції мережі та життєвого циклу кластера.

Таким чином, для SCADA життєво важливо встановити базове розділення: ядро SCADA – не контейнеризувати (використовуючи універсальний оркестратор); периферія SCADA – переважно Docker Swarm, але з Kubernetes, якщо є достатньо зріла команда платформи.

5.4.3 Облік енергоресурсів

Основним предметом дипломної роботи є системи управління енергоресурсами, такі як лічильники, системи білінгу, системи автоматичного зчитування показань (AMR) та системи типу MDM. Функціональні можливості таких систем включають збір даних, перевірку даних, зберігання вимірювань, формування звітів, створення сповіщень та API-інтерфейси для інтеграції з іншими підсистемами. На відміну від основної функціональності системи SCADA, головна вимога полягає не у високопродуктивній роботі в режимі реального часу, а в надійному фоновому обробленні даних із піковими запитами на секунду, піковим навантаженням на доступ до звітів, пакетною обробкою даних та відносно низькою – а не надзвичайно низькою – затримкою. Типові вимоги до таких систем включають SLA на рівні 99,9%+, можливість контролювати цикли оновлення послуг та особливу увагу до захисту приватної та комерційно чутливої інформації.

Для такого типу систем рекомендується застосовувати Kubernetes як цільову платформу. По-перше, автоматичне масштабування у разі нерівномірного навантаження запитів стає дуже важливим для вимірювання, а Kubernetes має вбудовані механізми горизонтального автомасштабування подів (HPA). Наприклад, у сценарії `hpa_load` на тестовому стенді було чітко видно, що HPA масштабував систему до восьми реплік, причому масштабування починалося приблизно через 31 секунду і закінчувалося приблизно через 348 секунд. Docker Swarm не підтримує вбудований HPA, і для сценарію `swarm_load` автоматичне масштабування не відбувалося, при цьому завжди підтримувалася конфігурація з двома репліками.

По-друге, для вимірювання ключовими функціями стають контрольоване відновлення після збою та цикли оновлення сервісу, оскільки потрібно мати можливість швидко реагувати на зміну статусу доступності сервісу. Результати показали, що Kubernetes працює швидше приблизно на 8 секунд при частоті відмов $R=2$ (приблизно 13,6 с проти 21,8 с для swarm) і значно швидше на етапі оновлення

сервісу (розгортання за 13,4 с проти 85,7 секунд для swarm), хоча точні цифри не можна узагальнити через різні налаштування. Така швидкість стає актуальною щодо кінцевих точок API, що отримують показання лічильників або запускають служби звітності.

По-третє, облік/виставлення рахунків пов'язане з приватними даними, для яких необхідні сегментація мережі, RBAC та мережеві політики. Kubernetes надає функції RBAC та NetworkPolicy і інтегрований з інструментами спостережності, такими як Prometheus, а також з додатковими агентами стану, такими як kube-state-metrics. Як результат, така комбінація стає набагато кращою з точки зору масштабованості з урахуванням політики безпеки та спостережності.

Такі рекомендації були підтверджені під час практичного випадку реалізації демонстраційного додатка платформи обліку електроенергії на базі бекенду та симулятора FastAPI, PostgreSQL та стеку Prometheus/Grafana, де кінцева точка POST /readings може відтворювати навантаження обліку/AMR. У цьому контексті Kubernetes стане платформою вибору для обліку та масштабованого обліку, тоді як Docker Swarm можна вважати прийнятним лише для пілотного проєкту без потреб в автомасштабуванні та розширених політик мережі та безпеки.

5.4.4 Попереджувальний аналіз та системи на основі ML

По-перше, архітектура систем прогнозової аналітики, профілактичного обслуговування та виявлення аномалій суттєво відрізняється від типових API або конвеєрів обробки даних. Зокрема, ці системи вимагають пакетного навчання моделей, онлайн-інференції, потокової обробки подій, інтеграції проміжного програмного забезпечення для обміну повідомленнями, такого як Kafka, а в деяких випадках – обчислень на базі графічних процесорів (GPU). Це вимагає гетерогенних розгортань, де будуть різні типи робочих навантажень, включаючи довготривалі служби для інференції, завдання пакетного перенавчання, заплановані завдання та виділення спеціалізованого обладнання.

У цьому відношенні Kubernetes є платформою вибору, оскільки це єдиний життєздатний варіант. Він має вбудовану підтримку одноразових та запланованих завдань через ресурси Kubernetes Jobs та CronJob, тим самим забезпечуючи природний спосіб реалізації періодичного навчання, пакетного оцінювання та планування для алгоритмів машинного навчання. Більше того, Kubernetes вбудовано підтримує концепцію плагінів пристроїв та навіть планування GPU, що робить його надзвичайно придатним для сучасних випадків використання ML. Разом це утворює ідеальну основу для побудови служб інференції ML, процесів пакетного навчання та подібних компонентів на кластері Kubernetes.

Крім того, практичні міркування вказують на те саме рішення. Дійсно, такі системи, як Kubeflow, KServe, KEDA та інші, які були розроблені спеціально з урахуванням машинного навчання, працюють на припущенні розгортання Kubernetes і не надають жодних абстракцій для виконання завдань/CronJob або розпізнавання GPU. Використання чогось на зразок Docker Swarm для систем прогнозного технічного обслуговування або виявлення аномалій неминуче призведе до великої кількості індивідуальних рішень та різко збільшить витрати на обслуговування. Отже, для машинного навчання та прогнозової аналітики рекомендується використовувати Kubernetes, тоді як Swarm слід повністю виключити.

5.4.5 Інші типові задачі

Доцільно об'єднати в одну категорію диспетчерські рішення, рішення, подібні до ринкових або до ROS, а також клієнтські портали B2C/B2B. Незважаючи на різні цілі та функції цих типів рішень, у них може бути одна спільна риса інфраструктури, яка вирізняє ці класи, – системна архітектура, що складається з рівня, що зберігає стан, фонових пакетних операцій та API/веб-інтерфейсу. Для диспетчерських і ринкових рішень слід враховувати управління розкладом, відмовостійкість, планування обчислень та можливість поєднання пакетних і

інтерактивних робочих процесів. У таких випадках Kubernetes буде кращим вибором, оскільки він надає більше можливостей з огляду на життєвий цикл і періодичність процесів у додатках.

З іншого боку, є також аспект, пов'язаний із клієнтським порталом. А саме, йдеться про портал B2C/B2B для приватних або бізнес-клієнтів. Такі рішення, хоча й не мають стану на рівні веб/HTTP, демонструють значні піки навантаження, наприклад, у вечірній час, після завершення розрахункового періоду або коли багато користувачів одночасно отримують доступ до рахунків-фактур та звітів. Щодо цього типу рішень, краще застосовувати більш гнучкий підхід. Починаючи з MVP або з ранніх версій продуктів, коли мова йде про невелике навантаження та необхідність швидкого виходу на ринок, Docker Swarm може бути хорошим стартом на трьох віртуальних машинах. Але, як тільки настає момент, коли кількість користувачів та навантаження стають неоднорідними і виникає потреба у масштабованості, можна порадити перехід на Kubernetes та HPA.

Таким чином, маючи справу з диспетчерськими/ринковими рішеннями, ми, безумовно, повинні розглянути можливість використання Kubernetes, тоді як для клієнтського порталу можна запропонувати еволюційний шлях, тобто від Swarm до Kubernetes. На практиці вибір оркестратора часто здійснюється не лише за класом системи, а й за поточним станом команди та проєкту. Тому доцільно додатково виділити типові сценарії прийняття рішень. Сформульовані рекомендації мають прикладний, але не універсальний характер. Вони спираються на результати локального експериментального стенда kind/Multipass, а тому абсолютні секундні або мілісекундні значення не слід переносити на production як готові SLA-оцінки. Крім того, SCADA core свідомо винесено за межі типових сценаріїв контейнерної оркестрації, оскільки для таких систем визначальними є вимоги промислової безпеки, детермінованості та відповідності ОТ-політикам, а не лише зручність розгортання. Остаточне рішення щодо вибору платформи в реальному проєкті повинно супроводжуватися аудитом замовника, оцінкою регуляторних вимог, кадрової зрілості команди та існуючої інфраструктури.

Для систематизації рекомендацій доцільно узагальнити їх у вигляді матриці, де поєднано тип системи, профіль навантаження та рекомендовану платформу. Матрицю представлено в таблиці 5.8.

Таблиця 5.8 – Зведена матриця рекомендацій щодо вибору платформи оркестрації

Клас системи	Профіль (real-time, SLA, stateful, autoscale, GPU)	Рекомендована платформа	Коротке обґрунтування
1	2	3	4
SCADA core	Hard real-time, ОТ-мережа, жорсткі вимоги до затримок, IEC 62443	Без контейнерної оркестрації загального призначення; bare metal / VM / vendor SCADA	Пріоритет детермінованості, верифікованості та промислової безпеки
SCADA periphery	Stable load, integration gateways, archivers, aggregators	Docker Swarm	Простота супроводу; autoscale зазвичай не потрібен
Облік енергоресурсів	Write spikes, SLA близько 99,9%, stateful backend + DB, autoscale бажаний	Kubernetes	HPA, кращі failover/rolling на стенді, NetworkPolicy, observability
Попереджувальний аналіз / ML	Batch + streaming, inference, Jobs, GPU	Kubernetes	Job/CronJob, GPU scheduling, ML-екосистема
Dispatch / market / ROS	Semi-real-time, stateful, batch jobs, інтеграції	Kubernetes	Краще підтримує складний життєвий цикл сервісів і планові задачі

Продовження таблиці 5.8

1	2	3	4
Customer portal	Stateless, піки трафіку, етапне зростання	Swarm для MVP; Kubernetes при зростанні	Швидкий старт у Swarm, подальша міграція на HPA-орієнтовану платформу

Якщо проєктується нова система metering або billing, доцільно одразу орієнтуватися на Kubernetes. Така рекомендація випливає з прикладного кейсу диплома, де саме для платформи обліку електроенергії підтверджено практичну цінність HPA, а також швидшого failover і rolling update на локальному стенді. Для систем, що з самого початку мають інтеграційний API, звітність, потенційні піки навантаження та вимоги до контролю доступу, Kubernetes є більш перспективною цільовою платформою.

Якщо команда є невеликою, строки запуску вимірюються тижнями, а основною метою є перевірка гіпотези або запуск MVP, доцільно використовувати Docker Swarm або траєкторію Compose – Swarm. Це дозволяє швидко перейти від локальних контейнерів до простого кластерного розгортання без повноцінної платформи Kubernetes. Водночас архітектура сервісів повинна від початку проєктуватися так, щоб у разі зростання навантаження забезпечити подальшу міграцію на Kubernetes.

Якщо система вже працює в Docker Swarm, але виникає потреба в NetworkPolicy, HPA, GPU-плануванні або складніших batch/job-сценаріях, рекомендується планова міграція на Kubernetes. Такий перехід доцільно трактувати не як технічну «моду», а як відповідь на зміни у функціональному профілі системи. Коли початкові переваги Swarm у простоті перестають компенсувати брак необхідних механізмів, Kubernetes стає більш раціональним інфраструктурним рішенням.

Загалом, доцільним можна вважати такий підхід. Використання Docker Swarm доцільне для систем із простими операціями, де очікуються стабільні робочі

навантаження, проводяться пілотні проекти та де вирішальне значення має швидкий запуск. З іншого боку, Kubernetes рекомендується для масштабованих архітектур, систем, що вимагають суворих політик безпеки, можливостей автоматичного масштабування, обчислень у сфері машинного навчання, складних завдань регулювання та забезпечення спостережуваності. Що стосується завдань, пов'язаних з цифровізацією обліку енергоресурсів, що є однією з головних тем, які розглядаються в цій роботі, Kubernetes можна розглядати як кращу цільову платформу. Щодо рішень SCADA, критично важливо чітко розмежовувати основну систему та її периферію.

Висновки до розділу 5

У п'ятому розділі було проведено емпіричну оцінку технологій Kubernetes та Docker Swarm проведеною на прикладі платформи обліку електроенергії. Таким чином, така тестова конфігурація дозволила зробити порівнянні висновки щодо масштабованості та відмовостійкості, а також щодо послідовних оновлень та умов навантаження. Результати свідчать, що Kubernetes зазвичай демонструє кращі показники в локальних умовах, особливо у випадках відмовостійкості та завдань послідовного оновлення. Крім того, ще одним аргументом на користь Kubernetes є те, що він дозволяє використовувати Horizontal Pod Autoscaler для автоматичного масштабування під навантаженням, а також NetworkPolicy для контролю поведінки мережі між сервісами. Водночас Docker Swarm має певні переваги з точки зору гнучкості розгортання та простоти експлуатації. Підсумовуючи, для вирішення проблеми цифровізації обліку електроенергії можна рекомендувати платформу Kubernetes, тоді як Docker Swarm можна використовувати як альтернативу лише для простих випадків.

ВИСНОВКИ

У даній дипломній роботі було проведено порівняльний аналіз платформ Kubernetes та Docker Swarm для оркестрації контейнеризованих застосунків у задачах цифровізації енергетичних систем. На основі теоретичного дослідження встановлено, що Kubernetes і Docker Swarm є самостійними платформами оркестрації з різними архітектурними підходами, моделями експлуатації та рівнями функціональної зрілості. Kubernetes орієнтований на складні масштабовані середовища з розвиненими механізмами автоматизації, тоді як Docker Swarm забезпечує простішу модель оркестрації з нижчим порогом входження.

Дослідження підтвердило, що вибір оркестратора не є самоціллю, а є частиною загального проєктування архітектури цифрової платформи. Для енергетичних інформаційних систем значення мають профіль навантаження, вимоги до затримок, критичність безперервної роботи, потреба в stateful-компонентах, рівень сегментації ОТ/ІТ та перспективи подальшого масштабування.

У практичній частині роботи реалізовано експериментальний стенд платформи обліку електроенергії, до складу якого входять backend на FastAPI, frontend, simulator потоку показань лічильників, PostgreSQL, Prometheus і Grafana. Використання симулятора та endpoint POST /readings дозволило наблизити прикладне навантаження до сценаріїв, характерних для систем комерційного обліку електроенергії (АСКОЕ), без відтворення повноцінної промислової інфраструктури.

Експериментальне порівняння Docker Swarm і Kubernetes проведено на однакових Docker-образах із використанням автоматизованих Bash-скриптів. За результатами дослідження встановлено, що Kubernetes у локальному стенді демонструє кращі часові характеристики в сценаріях масштабування, failover і rolling update. Додатковою перевагою Kubernetes є підтримка Horizontal Pod

Autoscaler для автоматичного масштабування, а також мережесхем політик для контролю взаємодії між сервісами.

Docker Swarm зберігає перевагу в простоті розгортання та нижчому порозі адміністрування, що робить його доцільним для невеликих пілотних середовищ, MVP-рішень та операційно простих систем зі стабільним навантаженням. Для масштабованих і політично керованих цифрових платформ енергетики більш доцільним є Kubernetes.

На основі отриманих результатів сформульовано рекомендації щодо вибору платформи оркестрації для типових класів ІТ-систем у енергетиці. Для систем SCADA core з жорсткими вимогами до реального часу контейнерна оркестрація загального призначення не рекомендується. Для SCADA periphery доцільним є Docker Swarm. Для систем обліку енергоресурсів, Систем попереджувального аналізу та ML-орієнтованих рішень рекомендовано Kubernetes.

У підсумку для задач цифровізації обліку енергоресурсів, які становлять предметний фокус цієї дипломної роботи, рекомендованою платформою є Kubernetes, а Docker Swarm доцільно розглядати переважно для простіших середовищ або початкових етапів впровадження. Отримані результати дозволяють обґрунтовано обирати платформу оркестрації в подальших проєктах цифровізації енергетичних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Docker Swarm vs Kubernetes: Container Orchestration Showdown. Wallarm | Advanced API Security. URL: <https://www.wallarm.com/cloud-native-products-101/docker-swarm-vs-kubernetes-container-orchestration> (дата звернення: 18.11.2025).
2. Revolutionizing the Energy Sector with API-Driven Solutions. KubeArk. URL: <https://kubeark.com/solutions-energy-industry> (дата звернення: 18.11.2025).
3. How to Achieve High Availability and Fault Tolerance in Kubernetes. Jeevi Academy. URL: <https://www.jeeviacademy.com/how-to-achieve-high-availability-and-fault-tolerance-in-devops> (дата звернення: 18.11.2025).
4. Impacts of digitalization on smart grids. ScienceDirect. URL: <https://www.sciencedirect.com/science/article/pii/S259017452400268X> (дата звернення: 18.11.2025).
5. Smart Grid Architect - An Overview. Energy Central. URL: <https://www.energycentral.com/energy-management/post/smart-grid-architect---an-overview-CLVhtQTunEPDSLt> (дата звернення: 18.11.2025).
6. Walker J. Docker Swarm vs. Kubernetes - Key Differences Explained. Spacelift. URL: <https://spacelift.io/blog/docker-swarm-vs-kubernetes> (дата звернення: 18.11.2025).
7. Energyworx - Case study - Nordcloud. Nordcloud. URL: <https://nordcloud.com/case-studies/energyworx/> (дата звернення: 18.11.2025).
8. Sunrock: Creating a clean energy future using Kubernetes – Pionative. Pionative. URL: <https://pionative.com/success-story-sunrock/> (дата звернення: 18.11.2025).

9. Quick start guide | PyCharm. PyCharm Help.
URL: <http://www.jetbrains.com/help/pycharm/quick-start-guide.html#hardware-requirements> (дата звернення: 01.06.2026).
10. The Python Tutorial. Python documentation.
URL: <https://docs.python.org/3.12/tutorial/index.html> (дата звернення: 28.05.2026).
11. FastAPI - FastAPI. *FastAPI* - *FastAPI*.
URL: <https://fastapi.tiangolo.com/> (дата звернення: 28.05.2026).
12. PostgreSQL: About. *PostgreSQL: The world's most advanced open source database*. URL: <https://www.postgresql.org/about/> (дата звернення: 29.05.2026).
13. SQLAlchemy. *SQLAlchemy - The Database Toolkit for Python*.
URL: <https://www.sqlalchemy.org/philosophy.html> (дата звернення: 29.05.2026).
14. alembic. *PyPI*. URL: <https://pypi.org/project/alembic/> (дата звернення: 29.05.2026).
15. Docker Inc. What is Docker?. *Docker Documentation*.
URL: <https://docs.docker.com/get-started/docker-overview/> (дата звернення: 01.06.2026).
16. Docker Inc. Docker Compose. *Docker Documentation*.
URL: <https://docs.docker.com/compose/> (дата звернення: 01.06.2026).
17. Docker Inc. Swarm mode. *Docker Documentation*.
URL: <https://docs.docker.com/engine/swarm/> (дата звернення: 01.06.2026).
18. Production-Grade Container Orchestration. *Kubernetes*.
URL: <https://kubernetes.io/> (дата звернення: 01.06.2026).
19. kind. *kind*. URL: <https://kind.sigs.k8s.io/> (дата звернення: 01.06.2026).
20. Multipass. *Multipass*.
URL: <https://documentation.ubuntu.com/multipass/stable/> (дата звернення: 01.06.2026).
21. Overview | Prometheus. *Prometheus - Monitoring system & time series database*. URL: <https://prometheus.io/docs/introduction/overview/> (дата звернення: 01.06.2026).

ДОДАТОК А ФРАГМЕНТ ЛІСТИНГУ ПРОГРАМИ

energy-platform.yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: energy
  labels:
    app.kubernetes.io/part-of: energy-platform
---
apiVersion: v1
kind: Secret
metadata:
  name: energy-secrets
  namespace: energy
type: Opaque
stringData:
  POSTGRES_DB: energy
  POSTGRES_USER: energy
  POSTGRES_PASSWORD: energy
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-data
  namespace: energy
spec:
  accessModes: [ReadWriteOnce]
  resources:
    requests:
      storage: 2Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  namespace: energy
  labels: { app: postgres, app.kubernetes.io/component: database }
spec:
  replicas: 1
  strategy:
    type: Recreate
  selector:
    matchLabels: { app: postgres }
  template:
    metadata:
      labels: { app: postgres }
    spec:
      automountServiceAccountToken: false
      securityContext:
        # postgres:16-alpine створює користувача `postgres` з UID 70.
        runAsNonRoot: true
        runAsUser: 70
        runAsGroup: 70
        fsGroup: 70
```

```

    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: postgres
    image: postgres:16-alpine
    imagePullPolicy: IfNotPresent
    securityContext:
      allowPrivilegeEscalation: false
      capabilities:
        drop: ["ALL"]
      readOnlyRootFilesystem: false
    envFrom:
    - secretRef: { name: energy-secrets }
    env:
    - { name: PGDATA, value: /var/lib/postgresql/data/pgdata }
    - { name: TZ, value: Europe/Kyiv }
    ports:
    - containerPort: 5432
    volumeMounts:
    - name: data
      mountPath: /var/lib/postgresql/data
    readinessProbe:
      exec:
        command: ["pg_isready", "-U", "energy", "-d", "energy"]
      initialDelaySeconds: 5
      periodSeconds: 5
    resources:
      requests: { cpu: "100m", memory: "128Mi" }
      limits: { cpu: "500m", memory: "512Mi" }
  volumes:
  - name: data
    persistentVolumeClaim:
      claimName: postgres-data
---
apiVersion: v1
kind: Service
metadata:
  name: postgres
  namespace: energy
spec:
  selector: { app: postgres }
  ports:
  - port: 5432
    targetPort: 5432
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  namespace: energy
  labels: { app: backend, app.kubernetes.io/component: api }
spec:
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels: { app: backend }
  template:

```

```

metadata:
  labels: { app: backend }
spec:
  automountServiceAccountToken: false
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
    seccompProfile:
      type: RuntimeDefault
  containers:
    - name: backend
      image: energy-backend:latest
      imagePullPolicy: IfNotPresent
      securityContext:
        allowPrivilegeEscalation: false
        readOnlyRootFilesystem: true
      capabilities:
        drop: ["ALL"]
      env:
        - name: DATABASE_URL
          value: postgresql+psycopg2://energy:energy@postgres:5432/energy
        - name: CORS_ORIGINS
          value: http://localhost,http://localhost:8081,http://127.0.0.1:8081
        - name: TZ
          value: Europe/Kyiv
      ports:
        - containerPort: 8000
      readinessProbe:
        httpGet: { path: /ready, port: 8000 }
        initialDelaySeconds: 10
      livenessProbe:
        httpGet: { path: /health, port: 8000 }
        initialDelaySeconds: 15
      resources:
        requests: { cpu: "100m", memory: "128Mi" }
        limits: { cpu: "400m", memory: "512Mi" }
      volumeMounts:
        - { name: tmp, mountPath: /tmp }
  volumes:
    - name: tmp
      emptyDir: { medium: Memory, sizeLimit: 64Mi }
---
apiVersion: v1
kind: Service
metadata:
  name: backend
  namespace: energy
spec:
  selector: { app: backend }
  ports:
    - port: 8000
      targetPort: 8000
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: energy
  labels: { app: frontend, app.kubernetes.io/component: ui }

```

```

spec:
  replicas: 1
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  selector:
    matchLabels: { app: frontend }
  template:
    metadata:
      labels: { app: frontend }
    spec:
      automountServiceAccountToken: false
      containers:
        - name: frontend
          image: energy-frontend:latest
          imagePullPolicy: IfNotPresent
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: false
            capabilities:
              drop: ["ALL"]
              add: ["NET_BIND_SERVICE", "CHOWN", "SETUID", "SETGID"]
          seccompProfile:
            type: RuntimeDefault
          env:
            - name: TZ
              value: Europe/Kyiv
          ports:
            - containerPort: 80
          readinessProbe:
            httpGet: { path: /, port: 80 }
            initialDelaySeconds: 5
          resources:
            requests: { cpu: "20m", memory: "32Mi" }
            limits: { cpu: "200m", memory: "128Mi" }
---
apiVersion: v1
kind: Service
metadata:
  name: frontend
  namespace: energy
spec:
  selector: { app: frontend }
  ports:
    - port: 80
      targetPort: 80

```

ДОДАТОК Б ПРЕЗЕНТАЦІЯ



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ПОРІВНЯЛЬНИЙ АНАЛІЗ KUBERNETES ТА DOCKER SWARM ДЛЯ ОРКЕСТРАЦІЇ КОНТЕЙНЕРИЗОВАНИХ ЗАСТОСУНКІВ У ЗАДАЧАХ ЦИФРОВІЗАЦІЇ ЕНЕРГЕТИЧНИХ СИСТЕМ

виконав: студент групи ТВ-23 Невмержицький Олександр Миколайович

керівник: асистент Єрошкін Юрій Миколайович, кандидат технічних наук

Київ 2026

Актуальність теми

Актуальність теми

Цифрова трансформація енергетики вимагає сучасних ІТ-рішень.

Потрібні масштабовані та стійкі до збоїв інфраструктури.

Контейнерні оркестратори спрощують керування та розгортання сервісів.

Kubernetes і Docker Swarm є ключовими інструментами для таких задач.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

2/13

Мета і завдання

Мета роботи
На прототипі системи обліку електроенергії провести теоретичний і практичний аналіз Kubernetes та Docker Swarm.

Формулювання завдання

- Проаналізувати існуючі підходи до оркестрації контейнеризованих застосунків у задачах цифровізації енергетичних систем;
- Дослідити архітектурні особливості та принципи функціонування Kubernetes і Docker Swarm;
- Спроектувати мікросервісну систему, що моделює типову задачу енергетичної сфери.
- Реалізувати програмну частину системи, виконати її контейнеризацію та підготувати конфігурації для розгортання.
- Розгорнути однакову мікросервісну систему в середовищах Kubernetes і Docker Swarm та провести порівняльний аналіз їх можливостей за критеріями масштабованості, відмовостійкості, швидкості розгортання, споживання ресурсів;
- Зібрати й проаналізувати метрики продуктивності та надійності;
- Сформулювати рекомендації щодо доцільності застосування Kubernetes і Docker Swarm для типових задач енергетики



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

3/13

Аналіз предметної області

Цифровізація енергетичних систем



Автоматизований збір та обробка даних

Автоматизований збір, передавання та обробка даних від лічильників, сенсорів і сервісів керування.



Режим реального часу

Робота сервісів у режимі, близькому до реального часу: облік, моніторинг, візуалізація, аналітика.



Критичні вимоги

Високі вимоги до масштабованості, відмовостійкості, безперервності роботи та спостережуваності.



Складність архітектури

Використання мікросервісної архітектури ускладнює розгортання та керування сервісами.

Чому потрібна оркестрація

- Автоматичне розгортання і перезапуск сервісів
- Масштабування під навантаження
- Керування оновленнями без зупинки системи
- Контроль мережевої взаємодії та моніторинг



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

4/13

Огляд існуючих рішень

Docker

Призначення: Контейнеризація застосунків

Переваги: Просте пакування і запуск контейнерів

Недоліки: Не вирішує оркестрацію на рівні кластера

Docker Swarm

Призначення: Базова оркестрація контейнерів

Переваги: Простий старт, нижчий поріг входження

Недоліки: Менше можливостей автоматизації та керування великими середовищами

Kubernetes

Призначення:

Повнофункціональна оркестрація

Переваги: Автомасштабування, self-healing, мережеві політики, широка розширюваність

Недоліки: Вища складність впровадження та адміністрування

Постановка проблеми:

- Вибір платформи оркестрації для цифрових енергетичних систем
- Необхідність забезпечення масштабованості та відмовостійкості
- Потреба в керованості та безпеці сервісів
- Обмеження складності впровадження й супроводу
- Порівняння Kubernetes і Docker Swarm на однаковому мікросервісному стенді.



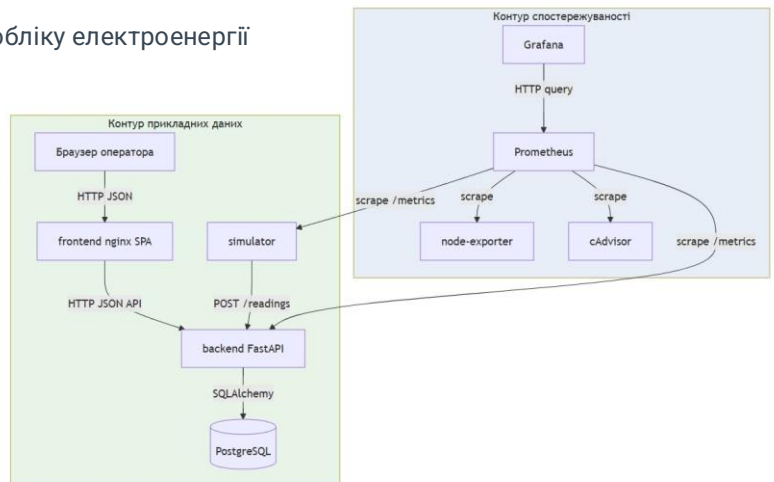
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

5/13

Архітектура системи обліку

Мікросервісний прототип платформи обліку електроенергії

- Backend (API) – бізнес-логіка
- Frontend – UI для перегляду показань
- Simulator – генерація потоку показань лічильників
- БД (часові ряди / реляційна) та Prometheus

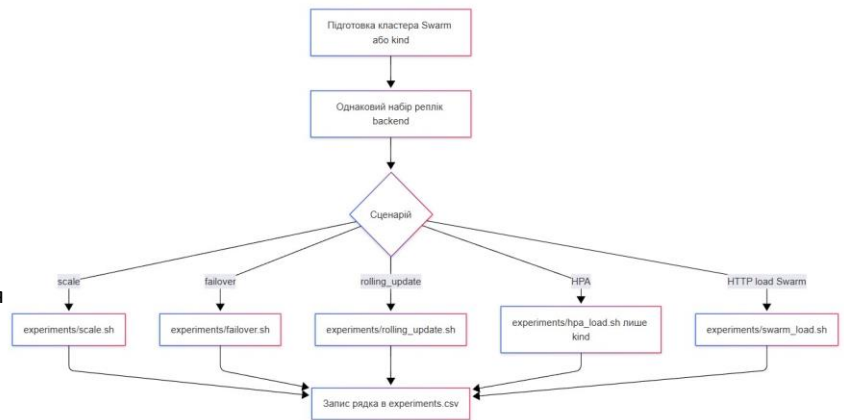


Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

6/13

Організація експериментів

- Однакові Docker-образи - однакова програмна база для обох платформ.
- `--platform=swarm | kind` - окреме збирання образів під цільове середовище.
- `Scale up / scale down` - перевірка масштабування сервісів.
- `Failover` - перевірка відновлення після збою.
- `Rolling update / load` - перевірка оновлення версії та роботи під навантаженням



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

7/13

Функціональне порівняння

Порівняння Kubernetes vs Docker Swarm

Kubernetes



- Автоматичне горизонтальне масштабування через HPA
- Підтримка NetworkPolicy для контролю мережевої взаємодії між Pod'ами
- Вища гнучкість для складних мікросервісних архітектур
- Краще підходить для масштабованих і політично керованих середовищ

Docker Swarm



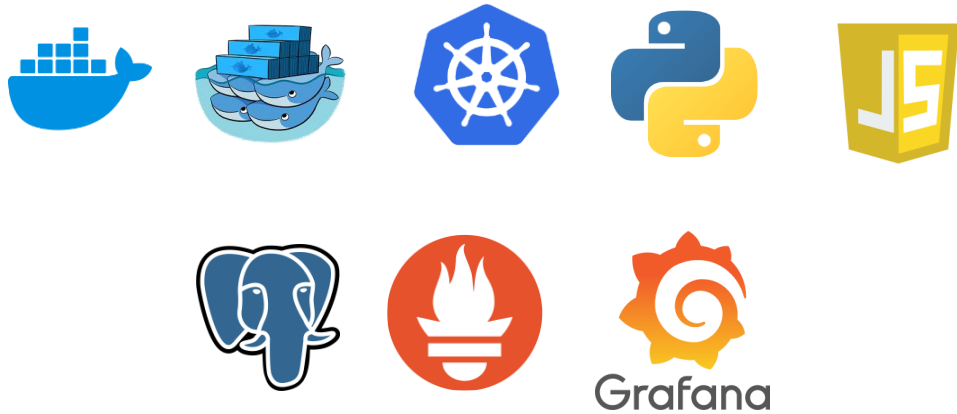
- Просте початкове розгортання через `docker swarm init` і `docker stack deploy`
- Нижчий поріг входження для адміністрування та швидкого старту
- Зручний для невеликих або менш складних інфраструктур
- Краще підходить для швидкого впровадження з меншими операційними витратами



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

8/13

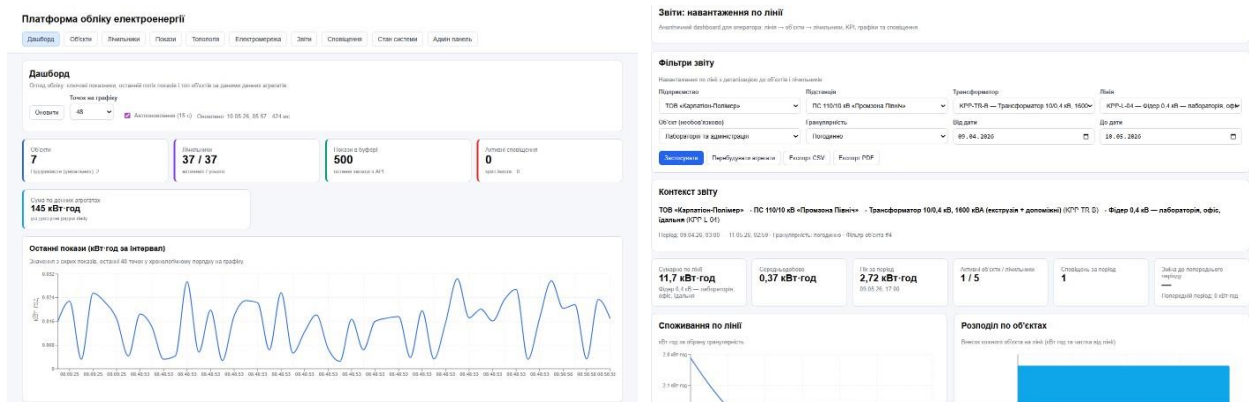
Використані технології



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

9/13

Інтерфейс системи обліку



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

10/13

Результати експериментів

Сценарій	Метрика	Docker Swarm	Kubernetes
Scale up (2 → 5/4 → 8)	t-first-ready	17,70 с	13,70 с
Scale up (2 → 5/4 → 8)	t-all-ready	18,60 с, n = 9	14,20 с, n = 9
Scale down (5 → 2/8 → 4)	завершення переходу	0,55 с	0,33–0,36 с
Failover R=2	drop	705 мс, n = 5	285 мс, n = 3
Failover R=2	recovery	21,80 с	13,60 с
Rolling update R=2	t-done	85,70 с, n = 5	13,40 с, n = 3



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

11/13

Висновки

- Проаналізовано існуючі підходи до оркестрації контейнеризованих застосунків у задачах цифровізації енергетичних систем.
- Досліджено архітектурні особливості та принципи функціонування Kubernetes і Docker Swarm.
- Спроектовано мікросервісну систему, що моделює типову задачу енергетичної сфери.
- Реалізовано програмну частину системи, виконати її контейнеризацію та підготувати конфігурації для розгортання.
- Розгорнуто однакову мікросервісну систему в середовищах Kubernetes і Docker Swarm та проведено порівняльний аналіз їхніх можливостей за критеріями масштабованості, відмовостійкості, швидкості розгортання, споживання ресурсів, мережевої взаємодії та безпеки.
- Зібрано й проаналізовано метрики продуктивності та надійності.
- Сформовано рекомендації щодо доцільності застосування Kubernetes і Docker Swarm для типових задач енергетики.



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

12/13



Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

National Technical University of Ukraine
"Igor Sikorsky Kyiv Polytechnic Institute"