МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ
ДЕРЖАВНИЙ ПОДАТКОВИЙ УНІВЕРСИТЕТ

ФАКУЛЬТЕТ ФІНАНСІВ ТА ЦИФРОВИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ І СИСТЕМ

КВАЛІФІКАЦІЙНА РОБОТА

на тему:
Створення модульної та інтерактивної системи моніторингу з адаптивним веб-інтерфейсом

Здобувача вищої освіти другого (магістерського) рівня,
II курсу, групи КМІ-24-2, спеціальності 122 «Комп'ютерні науки»,
освітня програма «Інформаційні управляючі системи і технології в економіці»

_____ Упірова Володимира Володимировича
(підпис) (прізвище, ім'я, по батькові)

Науковий керівник _____ Горбовий Артур Юліанович, д-р техн. наук, професор
(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Керівник освітньо-професійної програми _____ Омельчук Антон Анатолійович, канд. техн. наук
(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Допущено до захисту за рішенням кафедри протокол від «02» січня 2026 № 10

Завідувач кафедри _____ Філоненко Михайло Миколайович, канд. фіз.-мат. наук, доцент
(підпис) (прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Ірпінь – 2026

Реєстрація №3 16.01.2026 _____
(дата) (підпис відповідальної особи)

Результати перевірки до захисту 02.01.2026 _____
(до захисту, на доопрацювання) (дата) (підпис)

Результат захисту 5 (відмінно) 04.02.2026 _____
(оцінка) (дата) (підпис)

Голова комісії: _____ О.С. Дюбанов
(підпис)

Члени комісії: _____ А.Ю. Горбовий
(підпис)

_____ М.М. Філоненко
(підпис)

_____ І.Д. Погорєловська
(підпис)

БЛАНК ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ

Об'єкти контролю	Діапазон оцінки в балах	Фактична кількість балів	Примітки	Дата, П.І.Б.: керівника, голови комісії членів комісії,секретаря, (підписи)
ОЦІНЮВАННЯ КВАЛІФІКАЦІЙНОЇ РОБОТИ КОМІСІЄЮ ПО ЗАХИСТУ				
1. Зміст кваліфікаційної роботи (якість теоретичного матеріалу, його зв'язок з практичним, вміння представляти матеріал в логічній послідовності, наявність висновків та пропозицій)	**від 0 до 30**	29		04.02.2026 Дюбанов О.С. Горбовий А.Ю. Філоненко М.М. Погорєловська І.Д. Суховецька Л.П.
2. Дотримання вимог щодо оформлення	**від 0 до 10**	10		
3. Вчасне виконання роботи на всіх етапах у встановлені кафедрою терміни	**від 0 до 5**	5		
4. Рецензент	**від 0 до 5**	5		
РАЗОМ БАЛІВ	**від 0 до 50**	49		

Оцінювання кваліфікаційної роботи науковим керівником - 48 балів.

ПЕРЕВЕДЕННЯ ОЦІНКИ ЗА 100-БАЛЬНОЮ ШКАЛОЮ ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалоюЄКТС	Оцінка за національною шкалою	Дата	П.І.Б.: голови, членів комісії, секретаря, (підпис)
97	A	5	04.02.2026	Дюбанов О.С. Горбовий А.Ю. Філоненко М.М. Погорєловська І.Д. Суховецька Л.П.

ПРАВИЛА ПЕРЕВЕДЕННЯ ОЦІНОК ДО НАЦІОНАЛЬНОЇ ШКАЛИ ТА ШКАЛИ ЄКТС

Оцінка за 100-бальною шкалою	Оцінка за шкалою ЄКТС	Оцінка за національною шкалою
90-100	A	5 (відмінно)
80-89	B	4 (добре)
70-79	C	
60-69	D	3 (задовільно)
50-59	E	
35-49	FX	2 (незадовільно)

Факультет фінансів та цифрових технологій
Кафедра комп'ютерних та інформаційних технологій і систем
Галузь знань 12 «Інформаційні технології»
Спеціальність 122 «Комп'ютерні науки»
Освітня програма «Інформаційні управляючі системи і технології в економіці»

Затверджую
Завідувач кафедри _____ Філоненко М.М.
«11» грудня 2024

Завдання на кваліфікаційну роботу
здобувачу вищої освіти
Упірову Володимиру Володимировичу
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи: створення модульної та інтерактивної системи моніторингу з адаптивним веб-інтерфейсом.
Затверджена наказом ректора від «10» грудня 2024 № 1852
2. Строк здачі закінченої роботи «02» січня 2026
3. Мета роботи: розробка модульної системи моніторингу з адаптивним веб-інтерфейсом, яка забезпечує збір, агрегацію та візуалізацію даних з гетерогенних джерел.
Об'єкт дослідження: процеси збору, агрегації та візуалізації даних з гетерогенних джерел.
Предмет дослідження: архітектурні рішення та технології розробки платформи для збору, обробки та візуалізації метрик.
Орієнтовна база дослідження: система моніторингу Metrix, що розробляється як модульна платформа з адаптивним веб-інтерфейсом.
4. Вихідні дані до роботи: технічна документація технологій React, TypeScript, MongoDB, Express; наукові публікації з питань моніторингу та візуалізації даних; документація систем Prometheus, Grafana; специфікації OpenAPI та WebSocket.
5. Перелік графічного матеріалу: перелік графічного матеріалу: загальна архітектура системи Metrix; схема бази даних платформи; діаграма потоку даних Ingest Pipeline; діаграма компонентів системи віджетів; структура RESTful API платформи; структура клієнтської частини платформи.
6. Зміст кваліфікаційної роботи (перелік питань за кожним розділом)
Розділ 1. Теоретико-методологічні засади систем моніторингу (огляд існуючих рішень, моделі збору даних, технології веб-розробки, обґрунтування технологічного стеку).
Розділ 2. Проєктування архітектури платформи Metrix (загальна архітектура, мультитенантна база даних, модель збору даних, RESTful API, адаптивний інтерфейс).
Розділ 3. Реалізація системи моніторингу (серверна та клієнтська частини. тестування, демонстрація роботи).

7. Календарний план виконання роботи

№ п/п	Назва етапів кваліфікаційноїроботи	Строк виконання етапів роботи	
		За планом	Фактично
1	2	3	4
1	Вибір теми, підбір літературних джерел, формування структури кваліфікаційної роботи	До 15.12.2024	До 15.12.2024
2	**І розділ** (розкрити сутність і стан дослідження наукової проблематики роботи, після чого навести її загальну характеристику; огляд літератури за темою і вибір конкретних напрямів досліджень)	30.05.2025	30.05.2025
3	**ІІ розділ** (аналіз експериментальної частини про методики досліджень)	10.09.2025	10.09.2025
4	**ІІІ розділ** (проведені теоретичні і (або) експериментальні дослідження та проектні рішення)	20.11.2025	20.11.2025
5	**Висновки**	11.12.2025	11.12.2025
6	Доопрацювання роботи з урахуванням зауважень наукового керівника	25.12.2025	25.12.2025
7	Отримання відгуку керівника	28.01.2026	28.01.2026
8	Отримання зовнішньої рецензії	30.01.2026	30.01.2026

8. Дата видачі завдання «11» грудня 2024

9. Керівник кваліфікаційної роботи _____ (підпис) Горбовий А.Ю. (прізвище, ініціали)

10. Керівник освітньо-професійної програми _____ (підпис) Омельчук А.А. (прізвище, ініціали)

11. Завдання до виконання отримав здобувач вищої освіти _____ (підпис) Упіров В.В. (прізвище, ініціали)

АНОТАЦІЯ

Створення модульної та інтерактивної системи моніторингу з адаптивним веб-інтерфейсом.

(назва кваліфікаційної роботи)

Мета роботи полягає у розробці модульної системи моніторингу з адаптивним веб-інтерфейсом, яка забезпечує збір, агрегацію та візуалізацію даних з гетерогенних джерел. Завданнями роботи є: аналіз існуючих систем моніторингу та виявлення їх переваг і недоліків; дослідження сучасних технологій збору даних та протоколів обміну інформацією; обґрунтування вибору технологічного стеку; проєктування модульної архітектури системи з підтримкою мультитенантності; реалізація механізмів збору та обробки метрик; розробка адаптивного веб-інтерфейсу з інтерактивними дашбордами; тестування та оцінка ефективності рішення.

Об'єктом дослідження виступають процеси збору, агрегації та візуалізації даних з гетерогенних джерел.

Предметом дослідження є архітектурні рішення та технології розробки платформи для збору, обробки та візуалізації метрик.

Базою дослідження виступає система моніторингу Metrix, що розробляється як модульна платформа з адаптивним веб-інтерфейсом.

За результатами дослідження сформульовано принципи побудови модульної архітектури системи моніторингу з мультитенантною організацією даних, розроблено модель збору даних з підтримкою push-методів та автоматичним виявленням структури метрик, спроєктовано розширювану систему віджетів та адаптивний веб-інтерфейс із використанням технологій TypeScript, React, Redux Toolkit.

Одержані результати можуть бути використані для моніторингу IT-інфраструктури, аналізу бізнес-метрик, контролю IoT-пристроїв та інших сценаріїв, що потребують централізованого збору та візуалізації даних.

Кваліфікаційна робота містить _47_ сторінок, _1_ таблицю, _5_ рисунків, список використаних джерел з _46_ найменувань, _12_ додатків.

Ключові слова: моніторинг, метрики, веб-інтерфейс, модульна архітектура.

Рік виконання кваліфікаційної роботи 2025-2026.

Рік захисту роботи 2026.

ABSTRACT

Development of a Modular and Interactive Monitoring System with an Adaptive Web Interface.

(title of the qualification work)

The purpose of this work is to develop a modular monitoring system with an adaptive web interface that provides collection, aggregation, and visualization of data from heterogeneous sources. The objectives of the work include: analysis of existing monitoring systems and identification of their advantages and disadvantages; research of modern data collection technologies and communication protocols; justification of the technology stack selection; design of modular system architecture with multi-tenancy support; implementation of metrics collection and processing mechanisms; development of an adaptive web interface with interactive dashboards; testing and evaluation of the solution effectiveness.

The object of the research is the processes of collecting, aggregating, and visualizing data from heterogeneous sources.

The subject of the research is architectural solutions and technologies for developing a platform for collecting, processing, and visualizing metrics.

The research base is the Metrix monitoring system, developed as a modular platform with an adaptive web interface.

As a result of the research, principles for building a modular monitoring system architecture with multi-tenant data organization were formulated, a data collection model with push-method support and automatic metric structure detection was developed, an extensible widget system and adaptive web interface using TypeScript, React, and Redux Toolkit technologies were designed.

The obtained results can be used for IT infrastructure monitoring, business metrics analysis, IoT device control, and other scenarios requiring centralized data collection and visualization.

The qualification work contains _47_ pages, _1_ table, _5_ figures, a list of references with _46_ titles, _12_ appendices.

Keywords: monitoring, metrics, web interface, modular architecture.

Year of qualification work completion: 2025-2026.

Year of defense: 2026.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ 3

ВСТУП 4

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ЗАСАДИ СИСТЕМ МОНІТОРИНГУ 6

1.1. Огляд існуючих рішень для збору та візуалізації метрик 6
1.2. Моделі та протоколи збору даних з гетерогенних джерел 10
1.3. Сучасні технології веб-розробки для інтерактивних інтерфейсів 14
1.4. Обґрунтування технологічного стеку платформи Metrix 17

РОЗДІЛ 2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ПЛАТФОРМИ METRIX 22

2.1. Загальна архітектура системи 22
2.2. Мультитенантна архітектура бази даних 25
2.3. Модель збору та обробки даних 28
2.4. Проєктування RESTful API 32
2.5. Проєктування адаптивного веб-інтерфейсу 34

РОЗДІЛ 3. РЕАЛІЗАЦІЯ СИСТЕМИ МОНІТОРИНГУ 37

3.1. Реалізація серверної частини 37
3.2. Реалізація клієнтської частини 39
3.3. Тестування та забезпечення якості 41
3.4. Демонстрація роботи системи та перспективи розвитку 42

ВИСНОВКИ 45

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 48

ДОДАТКИ 53

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА СКОРОЧЕНЬ

API - Application Programming Interface (програмний інтерфейс застосунку)

CRUD - Create, Read, Update, Delete (базові операції з даними)

CSS - Cascading Style Sheets (каскадні таблиці стилів)

DOM - Document Object Model (об'єктна модель документа)

DTO - Data Transfer Object (об'єкт передачі даних)

HTML - HyperText Markup Language (мова розмітки гіпертексту)

HTTP - HyperText Transfer Protocol (протокол передачі гіпертексту)

IoT - Internet of Things (Інтернет речей)

JSON - JavaScript Object Notation (формат обміну даними)

MQTT - Message Queuing Telemetry Transport (протокол телеметрії)

NPM - Node Package Manager (менеджер пакетів Node.js)

OpenAPI - специфікація для опису REST API

REST - Representational State Transfer (архітектурний стиль веб-сервісів)

SaaS - Software as a Service (програмне забезпечення як послуга)

TICK - Telegraf, InfluxDB, Chronograf, Kapacitor (стек моніторингу)

TSDB - Time Series Database (база даних часових рядів)

WebSocket - протокол двонаправленого зв'язку в реальному часі

Workspace - робочий простір, ізольоване середовище для даних користувача або проєкту

backend - серверна частина застосунку, що обробляє бізнес-логіку та взаємодіє з базою даних

dashboard - інформаційна панель для візуалізації даних та метрик

dashlet - окремий елемент дашборду, контейнер для розміщення віджета

data bucket - сховище даних, контейнер для зберігання метрик певного типу

frontend - клієнтська частина застосунку, що відповідає за інтерфейс користувача

widget - віджет, компонент візуалізації даних на дашборді

ВСТУП

Сучасний світ інформаційних технологій характеризується стрімким зростанням кількості розподілених систем, хмарних сервісів та IoT-пристроїв. За даними дослідження JetBrains Developer Ecosystem Survey 2023, яке охопило понад 26 тисяч розробників з усього світу, мікросервісна архітектура стала домінуючим підходом до побудови корпоративних систем [1]. Це призводить до того, що організації стикаються з необхідністю збирати, обробляти та візуалізувати величезні обсяги метрик з різноманітних джерел даних.

Традиційні системи моніторингу, такі як Zabbix чи Nagios, проектувалися для роботи з монолітними застосунками та фіксованою інфраструктурою. Однак у контексті динамічного середовища Kubernetes, контейнеризованих мікросервісів та edge-обчислень ці інструменти виявляються недостатньо гнучкими [2]. Водночас комерційні рішення на кшталт DataDog чи New Relic, хоча й пропонують широкі можливості, часто є економічно недоступними для невеликих команд, стартапів чи освітніх закладів.

Актуальність теми дослідження зумовлена кількома факторами. По-перше, дослідження Observability Pulse 2024 показує, що 52% організацій намагаються отримати кращу видимість витрат на моніторинг, що на 16% більше порівняно з попереднім роком [3]. По-друге, зростає потреба в уніфікованих платформах, здатних працювати з гетерогенними джерелами даних - від серверних метрик до показників IoT-сенсорів та бізнес-аналітики. По-третє, сучасні веб-технології, зокрема React та TypeScript, дозволяють створювати інтерактивні інтерфейси, що раніше були доступні лише в desktop-застосунках.

Мета дослідження полягає у розробці модульної системи моніторингу з адаптивним веб-інтерфейсом, яка забезпечує збір, агрегацію та візуалізацію даних з різнорідних джерел.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- провести аналіз існуючих систем моніторингу та виявити їх переваги й недоліки;

- дослідити сучасні технології збору даних та протоколи обміну інформацією;
- обґрунтувати вибір технологічного стеку для розробки платформи;
- спроектувати модульну архітектуру системи з підтримкою мультитенантності;
- реалізувати механізми збору та обробки метрик з різних джерел;
- розробити адаптивний веб-інтерфейс з інтерактивними дашбордами;
- провести тестування та оцінку ефективності розробленого рішення.

Об'єктом дослідження є процеси збору, агрегації та візуалізації даних з гетерогенних джерел.

Предметом дослідження є архітектурні рішення та технології розробки платформи для збору, обробки та візуалізації метрик.

У роботі застосовано комплекс методів: аналіз та синтез для дослідження існуючих рішень; порівняльний аналіз для оцінки технологій; моделювання для проектування архітектури; експериментальний метод для тестування продуктивності системи.

Наукова новизна роботи полягає у:

- розробці модульної архітектури системи моніторингу з мультитенантною організацією даних на базі MongoDB;
- впровадженні Type-First підходу до розробки з використанням бібліотеки Zod для забезпечення типобезпеки на всіх рівнях системи;
- створенні універсальної системи типів метрик з підтримкою автоматичної нормалізації одиниць вимірювання.

РОЗДІЛ 1. ТЕОРЕТИКО-МЕТОДОЛОГІЧНІ ЗАСАДИ СИСТЕМ МОНІТОРИНГУ

1.1. Огляд існуючих рішень для збору та візуалізації метрик

Сучасний ринок інструментів моніторингу є досить різноманітним і включає як відкриті рішення, так і комерційні платформи. Для формування вимог до власної системи необхідно детально розглянути існуючі підходи та визначити їхні сильні та слабкі сторони.

Prometheus у поєднанні з Grafana становить найпопулярніший відкритий стек для моніторингу. Prometheus розроблявся компанією SoundCloud починаючи з 2012 року та у 2016 році став другим проектом Cloud Native Computing Foundation після Kubernetes [4]. Основною особливістю Prometheus є pull-модель збору даних, за якої сервер періодично опитує endpoints застосунків для отримання метрик. Дані зберігаються у власній time-series базі даних з використанням мови запитів PromQL (Prometheus Query Language).

Дослідження, опубліковане у 2024 році, демонструє, що інтеграція Prometheus та Grafana дозволяє створити уніфіковану систему моніторингу для мікросервісів [5]. Автори відзначають, що ключовими перевагами цього стеку є гнучкість налаштування алертів, підтримка широкого спектру експортерів для різних технологій та активна спільнота розробників.

Водночас Prometheus має певні обмеження. По-перше, архітектура передбачає локальне зберігання даних, що ускладнює горизонтальне масштабування. По-друге, pull-модель вимагає мережевої доступності всіх цільових систем, що не завжди можливо в розподілених середовищах. По-третє, для візуалізації необхідно використовувати окремий інструмент - Grafana, що збільшує складність розгортання.

Grafana являє собою платформу для візуалізації та аналітики, яка підтримує понад 150 джерел даних [6]. Її головною перевагою є можливість створення складних дашбордів з різноманітними типами графіків, таблиць та індикаторів. Дослідження патернів дизайну дашбордів, проведене Bach та

співавторами, виявило 42 типові шаблони, які активно використовуються в Grafana [7].

DataDog та New Relic представляють сегмент комерційних SaaS-рішень для моніторингу. DataDog пропонує комплексну платформу, що включає інфраструктурний моніторинг, APM (Application Performance Monitoring), логування та синтетичний моніторинг [8]. За даними офіційної документації, система підтримує понад 700 інтеграцій з різними технологіями та сервісами.

New Relic позиціонується як платформа full-stack observability, що забезпечує наскрізну видимість від браузера кінцевого користувача до бази даних [9]. Ключовою особливістю є використання агентів, які автоматично інструментують застосунки та збирають детальну телеметрію.

Головним недоліком комерційних рішень є їхня вартість. За оцінками аналітиків, витрати на моніторинг можуть становити значну частину IT-бюджету організації. Дослідження Observability Pulse 2024 показує, що 52% компаній активно шукають способи оптимізації витрат на моніторинг [3]. Крім того, залежність від хмарного постачальника створює ризики vendor lock-in.

Zabbix та Nagios належать до категорії традиційних систем моніторингу, що існують на ринку понад два десятиліття. Zabbix пропонує комплексне рішення для моніторингу мережевої інфраструктури, серверів та застосунків. Система підтримує як агентний, так і безагентний збір даних через SNMP (Simple Network Management Protocol), IPMI (Intelligent Platform Management Interface) та інші протоколи.

Nagios вважається одним з піонерів у галузі моніторингу з відкритим кодом. Його архітектура побудована на концепції перевірок (checks), які виконуються для визначення стану сервісів. Однак, як зазначають дослідники, традиційні системи погано адаптуються до динамічних хмарних середовищ [10].

Систематичний огляд Giamattei та співавторів виявив 71 інструмент моніторингу, які було класифіковано на бібліотеки (для збору даних) та платформи (для аналізу даних з різних джерел) [11]. Більшість платформ фокусуються на продуктивності (89%) та надійності (75%), і лише 21% - на

безпеці. Дослідження показало, що тільки 42% інструментів підтримують розподілене трасування запитів між мікросервісами, що є критичним для сучасних архітектур. Автором проведено дослідження застосування систем моніторингу для підвищення ефективності електронної комерції [12].

Для роботи з часовими рядами існують спеціалізовані платформи. InfluxDB є однією з найпопулярніших time-series баз даних, яка разом з Telegraf (агент збору) та Chronograf (візуалізація) утворює стек TICK [13]. Дослідження Tripathi та Miraz порівнює InfluxDB з MongoDB для управління даними часових рядів в IoT-середовищах [14]. Результати показують, що під високим навантаженням MongoDB демонструє кращу масштабованість.

TimescaleDB є розширенням PostgreSQL для роботи з часовими рядами. Це рішення дозволяє використовувати звичний SQL (Structured Query Language) для запитів до time-series даних. Перевагою є можливість зберігати в одній базі як метрики, так і реляційні дані.

ThingsBoard спеціалізується на IoT-платформах і пропонує комплексне рішення для управління пристроями, збору телеметрії та візуалізації [15]. Система підтримує протоколи MQTT, CoAP (Constrained Application Protocol) та HTTP для взаємодії з пристроями. Однак її архітектура орієнтована саме на IoT-сценарії і може бути надмірною для простішого моніторингу.

De Nardis та співавтори провели всебічний огляд понад 30 IoT-платформ для академічних досліджень та розробки [16]. Дослідження виявило, що підтримка множинних протоколів комунікації є критичним фактором для універсальності платформи. Автори класифікують платформи за призначенням та виділяють ключові характеристики для різних сценаріїв використання.

Для систематизації проведеного аналізу сформулюємо основні критерії порівняння систем моніторингу:

1. Модульність та розширюваність - можливість додавання нових джерел даних та типів візуалізацій без зміни ядра системи.

2. Підтримка гетерогенних джерел - здатність працювати з різними типами метрик: інфраструктурними, прикладними, бізнесовими, IoT.

3. Гнучкість візуалізації - наявність різноманітних типів графіків, можливість створення кастомних дашбордів, адаптивний дизайн.

4. Простота розгортання - кількість компонентів, вимоги до інфраструктури, час на початкове налаштування.

5. Вартість володіння - ліцензійні платежі, витрати на інфраструктуру, потреба у спеціалістах для підтримки.

Результати порівняльного аналізу представлено в таблиці 1.1. Детальне порівняння систем наведено в Додатку А

Таблиця 1.1.

Порівняння існуючих систем моніторингу

Система	Переваги	Недоліки
Prometheus + Grafana	Безкоштовна; 200+ exporters; потужна візуалізація Grafana; активна спільнота; гнучке налаштування алертів	Складне розгортання (2+ компоненти); локальне сховище обмежує масштабування; pull-модель вимагає мережевої доступності
DataDog	Проста (SaaS); 700+ інтеграцій; повний стек моніторингу; автоматичне масштабування	Дорога підписка; vendor lock-in; залежність від хмари
New Relic	Проста (SaaS); full-stack observability; автоінструментація агентами; наскрізна видимість	Дорога підписка; vendor lock-in; залежність від хмари
Zabbix	Безкоштовна; один компонент; стабільна; підтримка SNMP/IPMI	Не адаптована до хмарних середовищ; базова візуалізація; погано працює з мікросервісами
InfluxDB (TICK)	Безкоштовна; оптимізована для time-series; мільйони записів/сек	Складне розгортання; тільки time-series дані; гірша масштабованість під навантаженням; власна мова Flux

Джерело: [розроблено автором]

Проведений аналіз дозволяє зробити висновок, що жодне з розглянутих рішень не є універсальним. Prometheus з Grafana пропонують потужний, але складний у налаштуванні стек. Комерційні рішення вимагають значних фінансових інвестицій. Традиційні системи не адаптовані до сучасних архітектур. Спеціалізовані платформи вирішують лише частину задач.

Це обумовлює доцільність розробки власної системи моніторингу, яка б поєднувала модульність відкритих рішень, простоту використання комерційних платформ та гнучкість для різних сценаріїв застосування. Луценко І.А. підкреслює важливість модульного підходу для створення гнучких та

масштабованих систем, зазначаючи, що модульна архітектура дозволяє розробляти платформи, які можуть еволюціонувати разом із потребами користувачів [17].

1.2. Моделі та протоколи збору даних з гетерогенних джерел

Збір даних є фундаментальним етапом у системах моніторингу, і від правильного вибору моделі збору залежить ефективність всієї платформи. У практиці моніторингу розрізняють дві основні моделі: push та pull (Рис. 1.1.), кожна з яких має свої переваги та обмеження.

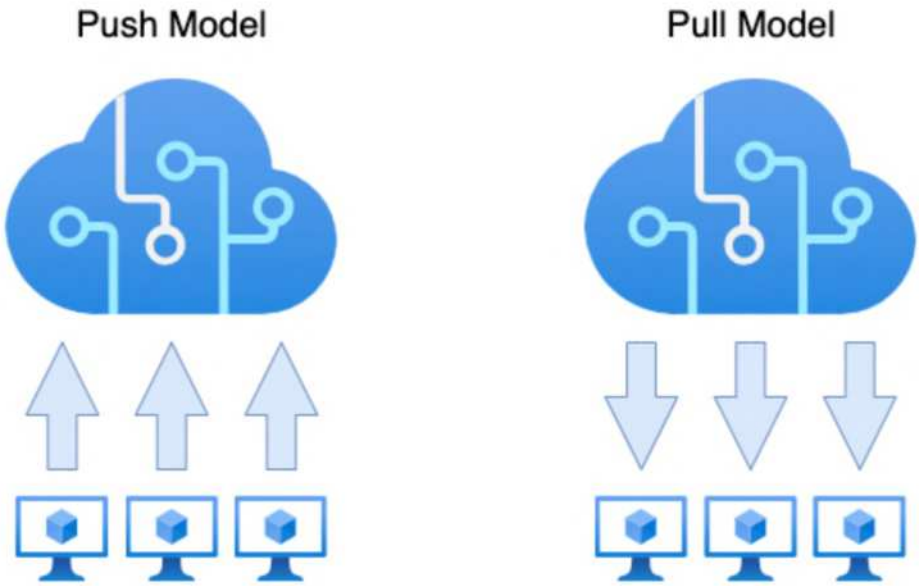


Рис. 1.1. Моделі збору даних Push та Pull

Pull-модель (або polling) передбачає, що сервер моніторингу періодично опитує цільові системи для отримання метрик. Цей підхід використовується у Prometheus, де центральний сервер виконує HTTP-запити до так званих експортерів - спеціальних компонентів, що надають метрики у стандартизованому форматі [4]. Інтервал опитування зазвичай становить від 15 до 60 секунд.

Перевагою pull-моделі є централізований контроль над процесом збору. Сервер моніторингу самостійно визначає, коли і які дані збирати, що спрощує управління навантаженням. Крім того, відсутність постійних з'єднань зменшує споживання ресурсів на стороні клієнтів. Як зазначено у дослідженні

моніторингу Kubernetes, pull-модель добре підходить для середовищ, де всі компоненти знаходяться в межах однієї мережі [2].

Водночас pull-модель має суттєві обмеження. По-перше, вона вимагає мережевої доступності всіх цільових систем з боку сервера моніторингу. Це може бути проблемою у випадку NAT, файрволів або географічно розподілених систем. По-друге, короткотривалі процеси (наприклад, batch jobs) можуть завершитися до того, як сервер встигне їх опитати. По-третє, збільшення кількості цілей лінійно збільшує навантаження на сервер.

Push-модель працює протилежним чином: клієнти самостійно надсилають метрики на центральний сервер. Цей підхід використовується у системах логування (наприклад, Elastic Stack), деяких APM-рішеннях та IoT-платформах. Клієнт формує пакет даних і відправляє його через HTTP POST, UDP або інші протоколи.

Головною перевагою push-моделі є можливість роботи через NAT та файрволи, оскільки з'єднання ініціюється з боку клієнта. Це критично для IoT-пристроїв, мобільних застосунків та розподілених систем [15]. Крім того, клієнт може буферизувати дані локально у випадку недоступності сервера та надіслати їх пізніше.

Дослідження cloud-native моніторингу демонструє, що push-модель краще підходить для динамічних середовищ, де компоненти часто створюються та знищуються [18]. Контейнери в Kubernetes можуть існувати лише кілька хвилин, і push-модель гарантує, що їхні метрики будуть зібрані.

Недоліком push-моделі є потенційне перевантаження сервера у випадку, коли велика кількість клієнтів одночасно надсилають дані. Також виникає питання автентифікації та авторизації клієнтів. На практиці часто використовуються гібридні підходи, що поєднують обидві моделі.

Протоколи передачі даних відіграють важливу роль у забезпеченні надійності та ефективності збору метрик. HTTP/REST є найпоширенішим протоколом для систем моніторингу завдяки своїй простоті та універсальності.

Prometheus використовує текстовий формат, що легко читається людиною та машиною [4].

WebSocket забезпечує двонаправлену комунікацію в реальному часі. Згідно з RFC 6455, протокол працює поверх TCP (Transmission Control Protocol) і дозволяє серверу надсилати дані клієнту без попереднього запиту [19]. Це корисно для дашбордів, що потребують оновлення в реальному часі без постійного polling.

MQTT (Message Queuing Telemetry Transport) широко застосовується в IoT через свою легковагість та підтримку ненадійних мереж. Протокол працює за моделлю publish/subscribe і оптимізований для пристроїв з обмеженими ресурсами [15].

Формати даних також суттєво впливають на ефективність системи. JSON є найпопулярнішим форматом завдяки читабельності та широкій підтримці у всіх мовах програмування. Однак текстове представлення збільшує обсяг даних порівняно з бінарними форматами.

Protocol Buffers (protobuf), розроблений Google, забезпечує компактне бінарне представлення даних зі схемою. Це зменшує обсяг трафіку та прискорює серіалізацію, але ускладнює налагодження.

Зберігання метрик вимагає спеціалізованих баз даних, оптимізованих для часових рядів. Time-series database (TSDB) проектуються з урахуванням специфіки даних: переважання операцій запису, послідовний доступ за часом, необхідність агрегації.

InfluxDB є однією з найпопулярніших спеціалізованих TSDB. Вона використовує власну мову запитів Flux та оптимізована для високої швидкості запису. За даними бенчмарків InfluxData, система здатна обробляти мільйони записів на секунду [20].

Дослідження Tripathi та Miraz порівнює InfluxDB з MongoDB для управління даними часових рядів [1]. Результати показують, що при 50 конкурентних з'єднаннях MongoDB демонструє латентність 0.00372 секунди

порівняно з 0.00337 у InfluxDB. Автори роблять висновок, що MongoDB забезпечує кращу масштабованість для високонавантажених IoT-застосунків.

MongoDB, починаючи з версії 5.0, підтримує нативні Time Series Collections [21]. Ця функція оптимізує зберігання та запити для даних з часовими мітками. За даними документації, Time Series Collections використовують колоночне зберігання та автоматично групують дані за часовими інтервалами, що зменшує обсяг диску до 16 разів порівняно зі звичайними колекціями.

Вибір TSDB залежить від конкретних вимог проекту. Спеціалізовані рішення, такі як InfluxDB або TimescaleDB, забезпечують максимальну продуктивність для time-series навантажень. MongoDB пропонує гнучкість та можливість зберігати різні типи даних в одній системі. Для невеликих проектів Prometheus зі вбудованою TSDB може бути достатнім.

Адаптери та конектори забезпечують інтеграцію з різними джерелами даних. У екосистемі Prometheus існує понад 200 офіційних та community exporters для різних технологій: баз даних, веб-серверів, черг повідомлень, хмарних сервісів тощо [4].

Концепція адаптерів дозволяє абстрагувати специфіку джерела даних від ядра системи моніторингу. Кожен адаптер відповідає за взаємодію з конкретною технологією та перетворення отриманих даних у стандартизований формат.

Для розроблюваної системи Metrix обрано підхід, що поєднує push та pull моделі. Push-based ingest API приймає дані від клієнтів через HTTP POST, що забезпечує сумісність з IoT-пристроями та розподіленими системами. У майбутніх версіях планується додати pull-based scraper для інтеграції з існуючими Prometheus exporters.

Як базу даних обрано MongoDB з Time Series Collections. Це рішення обумовлене кількома факторами. По-перше, MongoDB забезпечує гнучку схему, що важливо для роботи з гетерогенними джерелами даних. По-друге, мультитенантна архітектура потребує ізоляції даних на рівні бази, що легко реалізується в MongoDB. По-третє, єдина СУБД (система управління базами даних) для операційних та time-series даних спрощує розгортання та підтримку.

1.3. Сучасні технології веб-розробки для інтерактивних інтерфейсів

Розвиток веб-технологій за останнє десятиліття суттєво розширив можливості створення інтерактивних інтерфейсів. Якщо раніше складні дашборди реалізовувалися переважно як desktop-застосунки, то сьогодні веб-платформи здатні забезпечити аналогічний рівень інтерактивності та швидкодії.

JavaScript залишається домінуючою мовою веб-розробки. За даними дослідження JetBrains Developer Ecosystem Survey 2023, яке охопило понад 26 тисяч розробників, JavaScript утримує першість як найпопулярніша мова програмування вже сьомий рік поспіль [1]. Це пояснюється універсальністю мови: вона використовується як на клієнтській стороні (браузер), так і на серверній (Node.js).

TypeScript набуває все більшої популярності як надбудова над JavaScript, що додає статичну типізацію. Згідно з тим же дослідженням, TypeScript входить до п'ятірки найпопулярніших мов серед веб-розробників. Типізація дозволяє виявляти помилки на етапі компіляції, покращує автодоповнення в IDE (Integrated Development Environment) та полегшує рефакторинг коду.

React, розроблений компанією Meta (раніше Facebook), став де-факто стандартом для побудови інтерактивних веб-інтерфейсів [22]. За даними опитування Rising Stars 2023, React залишається найпопулярнішим frontend-фреймворком з найбільшою кількістю GitHub-зірок [23]. Основними концепціями React є компонентний підхід, віртуальний DOM та одностороння прив'язка даних.

Компонентний підхід дозволяє розбивати інтерфейс на незалежні, повторно використовувані частини. Кожен компонент інкапсулює свою логіку, стан та представлення. Дослідження архітектури React-застосунків демонструє, що правильна декомпозиція на компоненти критично впливає на підтримку коду [24].

Віртуальний DOM є ключовою оптимізацією React. Замість прямої маніпуляції реальним DOM браузера, React працює з його легковаговою копією в пам'яті. При зміні стану React порівнює віртуальний DOM з попередньою

версією та оновлює лише змінені частини реального DOM. Це забезпечує високу продуктивність навіть при частих оновленнях інтерфейсу.

Бібліотека Redux Toolkit стала стандартним рішенням для управління станом у середніх та великих React-застосунках. Дослідження порівняння Redux та Zustand показує, що Redux домінує в enterprise-застосунках з часткою 59.6% і використовується у 72% великих проектів [25]. Redux Toolkit спрощує роботу з Redux, зменшуючи кількість шаблонного коду та вбудовуючи кращі практики.

Ключовими концепціями Redux є store (єдине сховище стану), actions (описи подій) та reducers (функції оновлення стану). Redux Toolkit додає createSlice для спрощеного створення reducers, createAsyncThunk для асинхронних операцій та configureStore для налаштування store [26].

Порівняльне дослідження демонструє, що Redux Toolkit у 34.5 рази швидший за нативні підходи до управління станом при роботі зі складними даними [25]. Час виконання операцій становить 5.50 мс для Redux Toolkit проти 189.80 мс для ручного управління.

Бібліотеки візуалізації даних є критично важливими для систем моніторингу. Recharts, побудований на основі React та D3.js, забезпечує декларативний підхід до створення графіків [27]. Кожен тип графіка представлений окремим компонентом: LineChart, AreaChart, BarChart, PieChart тощо.

Перевагою Recharts є його інтеграція з React-екосистемою. Графіки можуть реагувати на зміну props та state так само, як і звичайні компоненти. Бібліотека підтримує анімації, tooltips, легенди та адаптивні контейнери.

Бібліотека D3.js (Data-Driven Documents) є низькорівневою бібліотекою для візуалізації даних, що надає повний контроль над SVG-елементами. Bostock та співавтори у фундаментальній роботі описують D3 як бібліотеку для маніпуляції DOM-елементів, що надає низькорівневий контроль над кожним аспектом візуалізації [28]. Такий підхід дозволяє створювати унікальні та високо кастомізовані графіки, проте вимагає значних зусиль розробника та глибокого

розуміння SVG, CSS та JavaScript. D3 використовується як основа для багатьох інших бібліотек, включаючи Recharts.

У власному дослідженні автора розглянуто компонентний підхід з використанням React та бібліотек D3.js, Recharts для створення адаптивного веб-інтерфейсу з оновленням даних в реальному часі [29].

Tailwind CSS версії 4, випущений у 2024 році, приніс покращення продуктивності та нові можливості. За даними Rising Stars 2023, Tailwind CSS є найпопулярнішим CSS-фреймворком серед JavaScript-розробників [23].

Бібліотека Radix UI є бібліотекою так званих headless компонентів [30]. Ці компоненти забезпечують поведінку, але не накладають стилів. Розробник має повну свободу у візуальному оформленні. Radix UI включає компоненти для діалогів, меню, табів, форм та багатьох інших елементів інтерфейсу. Важливою особливістю Radix UI є вбудована підтримка доступності (a11y). Компоненти автоматично керують фокусом, підтримують навігацію клавіатурою та надають правильні ARIA-атрибути (Accessible Rich Internet Applications - специфікація доступності веб-застосунків). Це критично для забезпечення доступності інтерфейсу для користувачів з обмеженими можливостями.

Бібліотека Vite стала провідним інструментом для збірки frontend-застосунків [31]. За даними Rising Stars 2023, Vite є найпопулярнішим build tool з часткою 38% серед JavaScript-розробників [23]. Vite використовує ES Modules для миттєвого запуску dev-сервера без попередньої збірки.

Маршрутизація в Single Page Applications (SPA) забезпечує навігацію між сторінками без повного перезавантаження. React Router є найпопулярнішим рішенням, але його розмір може бути надмірним для простих застосунків.

Бібліотека wouter є мінімалістичною альтернативою з розміром близько 4 КБ [32]. Бібліотека надає компоненти Route, Switch, Link та хуки useLocation, useRoute. Простота wouter робить його привабливим вибором для проектів, де не потрібні складні можливості React Router.

Адаптивний дизайн забезпечує коректне відображення інтерфейсу на пристроях з різними розмірами екрану. Tailwind CSS надає систему breakpoints

(sm, md, lg, xl, 2xl), що дозволяє визначати різні стилі для різних розмірів екрану. Дослідження патернів дизайну дашбордів виявило, що 94% проаналізованих дашбордів використовують сітковий макет [7]. Бібліотека react-grid-layout дозволяє створювати draggable та resizable сітки, де користувач може самостійно розміщувати віджети [33].

Toasa та співавтори у своєму дослідженні описують техніки візуалізації real-time даних з використанням Node.js, Socket.io та Redis [34]. Автори демонструють можливість побудови системи дашбордів для аналізу даних із затримкою менше 100 мілісекунд від надходження даних на сервер до їх відображення на інтерфейсі Використання WebSocket з двонаправленим зв'язком забезпечує ефективне оновлення візуалізацій без постійного polling.

Для розроблюваної системи Metrix обрано стек технологій, що включає React 19, TypeScript, Redux Toolkit, Tailwind CSS 4, Radix UI, Recharts та Vite. Цей вибір обумовлений балансом між продуктивністю, розміром бібліотек та можливостями для кастомізації.

1.4. Обґрунтування технологічного стеку платформи Metrix

Вибір технологій для розробки програмного забезпечення є критичним рішенням, що визначає швидкість розробки, масштабованість, підтримуваність та довгострокову життєздатність продукту. У цьому підрозділі обґрунтовано вибір кожного компонента технологічного стеку платформи Metrix на основі проведеного аналізу та визначених вимог.

Вимоги до платформи сформульовано на основі аналізу існуючих рішень та потреб потенційних користувачів:

1. Типобезпека - система повинна мінімізувати помилки часу виконання через статичну типізацію на всіх рівнях.
2. Модульність - архітектура має дозволяти додавання нових типів метрик, візуалізацій та джерел даних без переписування ядра.
3. Мультитенантність - підтримка ізольованих робочих просторів для різних користувачів або проектів.

4. Продуктивність - здатність обробляти тисячі метрик на секунду та відображати дані в реальному часі.

5. Простота розгортання - мінімальна кількість залежностей, можливість запуску на одному сервері.

6. Адаптивність інтерфейсу - коректна робота на різних пристроях від мобільних телефонів до великих моніторів.

TypeScript обрано як основну мову програмування для всіх частин системи: клієнта, сервера та спільних модулів. Cherny у своїй книзі описує TypeScript як ключовий інструмент для зменшення runtime-помилок через static type checking [35]. Це рішення обумовлено кількома факторами [36].

По-перше, статична типізація дозволяє виявляти значну частину помилок ще до запуску коду. Це особливо важливо для системи моніторингу, де помилка може призвести до втрати критичних даних.

По-друге, TypeScript забезпечує чудову підтримку в IDE: автодоповнення, перехід до визначень, рефакторинг. Це прискорює розробку та зменшує когнітивне навантаження на розробника.

По-третє, використання однієї мови для frontend та backend спрощує повторне використання коду. Схеми валідації, типи даних та утиліти можуть бути визначені один раз і використовуватися на обох сторонах.

Zod обрано як бібліотеку для валідації схем даних [37]. На відміну від традиційних підходів, де типи TypeScript та валідація визначаються окремо, Zod дозволяє виводити типи безпосередньо зі схем валідації. Це забезпечує синхронізацію між типами та валідацією і реалізує Type-First підхід. Додатковою перевагою Zod є можливість автоматичної генерації OpenAPI-документації. Бібліотека zod-to-openapi перетворює Zod-схеми на OpenAPI-специфікації, що автоматизує документування API [38].

React 19 обрано як frontend-фреймворк з огляду на його домінуючу позицію на ринку та розвинену екосистему [22]. Компонентний підхід React ідеально підходить для побудови модульної системи віджетів, де кожен тип візуалізації є окремим компонентом.

Redux Toolkit обрано для управління глобальним станом застосунку [26]. Результати дослідження показують, що Redux Toolkit демонструє значно вищу продуктивність порівняно з ручним управлінням станом [25]. Крім того, передбачуваність потоку даних у Redux спрощує налагодження та тестування.

Tailwind CSS 4 обрано як систему стилізації завдяки її продуктивності та гнучкості [39]. Utility-first підхід усуває необхідність перемикатися між файлами стилів та розміткою. CSS Variables забезпечують підтримку тем оформлення.

Бібліотека Radix UI надає базові компоненти з вбудованою accessibility [30]. Це звільняє від необхідності самостійно реалізовувати складну логіку діалогів, меню, табів тощо. Відсутність нав'язаних стилів дозволяє повністю кастомізувати вигляд компонентів.

Бібліотеку Recharts обрано як бібліотеку візуалізації завдяки її React-орієнтованому підходу [27]. Графіки визначаються через компоненти, що природно вписується в архітектуру застосунку. Бібліотека підтримує всі необхідні типи графіків та має хорошу продуктивність.

Бібліотека Vite використовується як інструмент збірки завдяки його швидкості [31]. Миттєвий запуск dev-сервера та швидкий Hot Module Replacement суттєво прискорюють цикл розробки.

Бібліотека wouter обрано як маршрутизатор через його мінімальний розмір та достатню функціональність [32]. Для системи моніторингу не потрібні складні можливості React Router, тому легший варіант є оптимальним.

Бібліотека Express 5 залишається стандартом для Node.js веб-серверів завдяки простоті та гнучкості [40]. Версія 5 принесла покращену підтримку async/await та TypeScript. Луценко І.А. підкреслює, що Node.js завдяки event-driven архітектурі є оптимальним вибором для масштабованих веб-систем [41]. Використання JavaScript/TypeScript на всьому стеку дозволяє працювати з єдиною екосистемою, що підвищує продуктивність розробки.

Протокол WebSocket забезпечує bidirectional real-time комунікацію через persistent connection. Lombardi у своїй книзі описує переваги WebSocket порівняно з polling: з'єднання встановлюється один раз через HTTP upgrade, після

чого обидві сторони можуть надсилати повідомлення без overhead повторних запитів [42]. Для платформи моніторингу це забезпечує низьку латентність оновлення дашбордів.

Ефективність full-stack TypeScript підходу з використанням shared Zod schemas для забезпечення типобезпеки на всіх рівнях системи підтверджено в дослідженні архітектури веб-платформ з real-time оновленням [43].

MongoDB обрано як основну базу даних з кількох причин [44]. По-перше, документна модель забезпечує гнучкість при роботі з різними структурами метрик. По-друге, Time Series Collections оптимізують зберігання часових рядів [21]. По-третє, мультитенантна архітектура легко реалізується через окремі бази для кожного workspace.

Дослідження порівняння MongoDB та InfluxDB показує, що MongoDB забезпечує кращу масштабованість під високим навантаженням [1]. Крім того, використання однієї СУБД для операційних та time-series даних спрощує інфраструктуру.

Бібліотека Pino використовується для логування завдяки найвищій продуктивності серед Node.js логерів [45]. Структуровані JSON-логи спрощують аналіз та інтеграцію з системами агрегації логів.

OpenAPI зі Scalar UI забезпечує інтерактивну документацію API [38]. Автоматична генерація з Zod-схем гарантує актуальність документації та зменшує ручну роботу.

Архітектура монорепозиторію з NPM Workspaces обрана для організації коду [46]. Це дозволяє спільно використовувати код між клієнтом та сервером, забезпечуючи єдине джерело істини для типів та схем.

Структура монорепозиторію включає три workspace: client (React-застосунок), server (Express API) та shared (спільні модулі). Shared містить Zod-схеми, типи, константи та утиліти, що використовуються обома частинами.

Підсумовуючи, технологічний стек платформи Metrix включає:

Frontend: React 19, TypeScript, Redux Toolkit, Tailwind CSS 4, Radix UI, Recharts, wouter, Vite.

Backend: Node.js, Express 5, TypeScript, MongoDB 6, Zod, Pino, OpenAPI/Scalar.

Спільне: TypeScript, Zod, ESLint.

Цей стек забезпечує баланс між продуктивністю, типобезпекою, швидкістю розробки та підтримуваністю. Використання сучасних технологій гарантує актуальність рішення у середньостроковій перспективі.

РОЗДІЛ 2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ПЛАТФОРМИ METRIX

2.1. Загальна архітектура системи

Проєктування архітектури є ключовим етапом розробки програмного забезпечення, що визначає структуру системи, взаємодію компонентів та можливості масштабування. У цьому підрозділі представлено загальну архітектуру платформи Metrix, організацію кодової бази та архітектурні шари системи.

Для організації кодової бази обрано підхід монорепозиторію (monorepo), за якого весь код проекту зберігається в одному репозиторії з логічним розділенням на модулі. Цей підхід набув популярності серед великих технологічних компаній та рекомендується для проектів, де клієнтська та серверна частини тісно пов'язані [46].

Структура монорепозиторію Metrix включає три основні шари:

- client - веб-інтерфейс користувача;
- server - веб-сервер, що надає REST API та обробляє бізнес-логіку;
- shared - спільні модулі, що використовуються обома частинами.

NPM Workspaces забезпечують технічну реалізацію монорепозиторію. Ця функціональність npm дозволяє керувати залежностями на рівні кореневого package.json файлу та виконувати команди для окремих workspace. Спільні залежності, такі як TypeScript та ESLint, встановлюються на кореневому рівні, що зменшує дублювання та спрощує оновлення.

Модуль shared містить код, що повторно використовується на клієнті та сервері. Сюди входять Zod-схеми для валідації даних, типи TypeScript, константи, утиліти та класи помилок. Така організація забезпечує єдине джерело істини для структур даних і гарантує узгодженість між API та клієнтом.

Архітектура системи побудована за принципом багатошаровості, де кожен шар має чітко визначену відповідальність та взаємодіє лише з сусідніми шарами.

Це забезпечує низьку зв'язаність компонентів та високу згуртованість всередині шарів.

1) Presentation Layer (шар представлення) відповідає за відображення даних користувачу та обробку його дій. На клієнті цей шар реалізовано через React-компоненти: екрани (screens), макети (layouts) та базові елементи інтерфейсу (bricks). На сервері шар представлення відсутній, оскільки API повертає дані у форматі JSON.

2) Application Layer (прикладний шар) містить логіку координації між шаром представлення та доменним шаром. На клієнті це Redux store (централізоване сховище стану застосунку) зі slices, selectors та async thunks. На сервері - контролери (controllers), що приймають HTTP-запити, викликають відповідні сервіси та формують відповіді.

3) Domain Layer (доменний шар) інкапсулює бізнес-логіку системи. На сервері це workflows для обробки даних, правила валідації та трансформації. Доменний шар не залежить від технічних деталей зберігання чи представлення даних.

4) Data Layer (шар даних) відповідає за взаємодію з базою даних та зовнішніми сервісами. На сервері це MongoDB collections з методами для CRUD-операцій. На клієнті - API-клієнт для взаємодії з сервером через HTTP.

Діаграма загальної архітектури системи представлена на рисунку 2.1.

Взаємодія між клієнтом та сервером здійснюється через REST API. Клієнт надсилає HTTP-запити до сервера, який обробляє їх та повертає відповіді у форматі JSON. Всі точки доступу (endpoints) згруповані під префіксом /api для чіткого розділення API від статичних ресурсів.

Потік даних у системі є односпрямованим, що спрощує відстеження змін та налагодження. На клієнті користувач ініціює дію, яка диспатчиться в Redux store. Async thunk виконує API-запит, отримує відповідь та оновлює стан. React-компоненти автоматично перерендерюється при зміні стану.

На сервері запит проходить через ланцюжок middleware: CORS (Cross-Origin Resource Sharing), парсинг JSON, логування, аутентифікація. Контролер

валідує вхідні дані через Zod-схеми, викликає відповідний workflow та формує відповідь. Глобальний обробник помилок перехоплює помилки та повертає стандартизовану структуру.

Система підтримує два режими роботи: розробка (development) та продакшен (production). У режимі розробки Vite надає dev-сервер з Hot Module Replacement для клієнта, а сервер запускається через tsx з автоматичним перезавантаженням при змінах коду. API-запити проксуються з Vite на Express.

У продакшен-режимі клієнтський код збирається Vite у статичні файли (HTML, CSS, JavaScript), які розміщуються в папці dist/public. Express-сервер обслуговує ці файли та обробляє API-запити. Така архітектура дозволяє розгортати систему на одному сервері без необхідності окремого веб-сервера для статичних файлів.

Модульність системи забезпечується через чітке розділення відповідальності між компонентами. Кожен тип метрики, візуалізації чи джерела даних може бути доданий як окремий модуль без зміни ядра системи. Registry Pattern використовується для реєстрації та отримання модулів за типом.

Масштабування системи може здійснюватися горизонтально через запуск кількох інстансів сервера для балансування навантаження. Stateless-архітектура API (сесії зберігаються в базі даних) забезпечує можливість обробки запитів будь-яким інстансом.

Для забезпечення надійності система використовує принцип fail-fast: помилки виявляються та обробляються якомога раніше. Валідація вхідних даних відбувається на рівні контролера, невалідні запити відхиляються до виконання бізнес-логіки. Типізація TypeScript запобігає багатьом помилкам на етапі компіляції.

Логування через бібліотеку Pino забезпечує структуровані JSON-логи з мітками часу, рівнями важливості та контекстною інформацією [45]. Логи включають деталі HTTP-запитів, помилки валідації та інформацію про операції з базою даних. Це полегшує діагностику проблем та моніторинг роботи системи.

Важливим аспектом архітектури є організація зберігання даних для підтримки кількох незалежних користувачів, що розглядається у наступному підрозділі.

2.2. Мультитенантна архітектура бази даних

Мультитенантність (multi-tenancy) є архітектурним підходом, за якого одна інстанція програмного забезпечення обслуговує кількох незалежних клієнтів (tenants). Кожен клієнт працює в ізольованому середовищі, не маючи доступу до даних інших клієнтів. Цей підхід широко використовується в SaaS-продуктах і є ключовим для платформи Metrix.

У контексті Metrix tenant реалізується через концепцію Workspace - ізольованого робочого простору для збору та візуалізації метрик. Кожен workspace може належати окремому користувачу, команді чи проекту. Дані одного workspace повністю ізольовані від інших, що забезпечує безпеку та конфіденційність.

Існує кілька підходів до реалізації мультитенантності на рівні бази даних:

1. Shared Database, Shared Schema - всі клієнти використовують одну базу та одні таблиці, розділення через tenant_id. Простий у реалізації, але складний для ізоляції та масштабування.
2. Shared Database, Separate Schema - одна база, але окремі схеми для кожного клієнта. Середній рівень ізоляції.
3. Separate Database - кожен клієнт має власну базу даних. Найвищий рівень ізоляції, простіше масштабування та резервне копіювання.

Для платформи Metrix обрано гібридний підхід: спільна база для глобальних даних та окремі бази для кожного workspace. Це забезпечує високий рівень ізоляції для чутливих даних метрик при збереженні простоти управління глобальними сутностями.

Головна база даних main-db містить дані, що є спільними для всієї системи:

- accounts - облікові записи користувачів з email, хешами паролів та профільною інформацією;

- sessions - активні сесії користувачів з auth-токенами та термінами дії;

- workspaces - метадані робочих просторів: назва, власник, учасники;

Workspace-специфічні бази даних мають формат імені w-{workspaceId}, де workspaceId є унікальним ідентифікатором у форматі MongoDB ObjectId. Кожна така база містить:

- dashboards - конфігурації дашбордів з розміщенням віджетів;
- widgets - налаштування віджетів: тип, джерело даних, параметри візуалізації;
- collectors - конфігурації колекторів для збору метрик;
- data_buckets - визначення сховищ даних зі схемами полів;
- bucket_{bucketId} - Time Series колекції для зберігання метрик або Tabular колекції для структурованих даних;
- training_data - тимчасові дані для навчання колекторів.
- access_tokens - токени для програмного доступу до API.

Такий розподіл має кілька переваг. По-перше, повна ізоляція даних метрик гарантує, що навіть при помилці в коді дані одного workspace не потраплять до іншого. По-друге, резервне копіювання та відновлення можна виконувати для окремих workspace. По-третє, видалення workspace зводиться до простого видалення всієї бази данних.

MongoDB обрано як СУБД з кількох причин [44]. Документна модель забезпечує гнучкість при роботі з різними структурами метрик - кожен bucket може мати унікальний набір полів. Нативна підтримка Time Series Collections оптимізує зберігання часових рядів [21]. Горизонтальне масштабування через шардінг доступне при потребі.

Time Series Collections, введені в MongoDB 5.0, спеціально оптимізовані для даних з часовими мітками [21]. Вони використовують внутрішнє групування документів за часовими інтервалами (bucketing), що зменшує обсяг зберігання та прискорює запити. За даними документації, Time Series Collections можуть займати до 16 разів менше місця порівняно зі звичайними колекціями. Структура документа у Time Series колекції Metrix показана Рис. 2.1.

```
{
  _id: ObjectId,
  timestamp: Date,
  key: "cpu_usage",
  value: 45.5,
  tags: { host: "server-01", region: "eu-west" }
}
```

Рис. 2.1. Приклад Time Series документа в базі даних

Поле key містить назву метрики, value - числове значення, tags - додаткові мітки для фільтрації та групування. Такий формат дозволяє зберігати різні метрики в одній колекції та ефективно виконувати агрегаційні запити.

Tabular колекції призначені для даних, де важлива кореляція між полями. На відміну від Time Series, де кожне значення є окремим документом, у Tabular всі поля одного запису зберігаються разом. Структура документа у Tabular колекції Metrix показана Рис. 2.2.

```
{
  _id: ObjectId,
  timestamp: Date,
  tags: { ... },
  data: { cpu_usage: 45.5, memory_mb: 2048,  disk_iops: 150 }
}
```

Рис. 2.2. Приклад Tabular документа в базі даних

Цей формат оптимальний для аналітичних запитів, що потребують доступу до кількох полів одночасно.

Для абстракції роботи з колекціями використовується патерн Repository, реалізований через клас BaseCollection. Цей клас делегує виклики до нативного MongoDB Driver та забезпечує трансформацію документів.

Індексація є критичною для продуктивності запитів. MongoDB автоматично створює індекси на полях timeField та metaField для Time Series колекцій. Додатково створюються індекси:

- sessions: {authToken: 1} для швидкого пошуку сесій за токеном;
- workspaces: {ownerId: 1} для пошуку workspace користувача;
- collectors: {status: 1, type: 1} для фільтрації за статусом;
- training_data: {timestamp: 1} з TTL (Time To Live) 7 днів для автоматичного видалення.

TTL (Time To Live) індекси забезпечують автоматичне видалення застарілих документів. Для training_data це важливо, оскільки дані потрібні лише на етапі налаштування колектора.

Схема бази даних представлена в рисунках в Додатку В.

2.3. Модель збору та обробки даних

Збір та обробка даних є центральною функцією системи моніторингу. У цьому підрозділі описано архітектуру компонентів, що забезпечують надходження метрик до системи, їх валідацію, нормалізацію та збереження.

Система колекторів (Collectors) забезпечує надходження метрик до платформи. Колектор є абстракцією джерела даних, що визначає спосіб отримання метрик, їх структуру та цільове сховище. Кожен колектор має унікальний ідентифікатор, назву, тип та статус.

Типи колекторів визначають механізм отримання даних:

- ingest - push-based колектор, що приймає дані через HTTP POST. Зовнішні системи надсилають метрики на endpoint /api/ingest/{collectorId}. Цей тип є основним для інтеграції з довільними джерелами.
- scraper - pull-based колектор (планується), що періодично опитує зовнішні endpoints для отримання метрик.

- Статуси колекторів відображають їх життєвий цикл:
- training - режим навчання, коли система збирає зразки даних для визначення структури полів. Дані зберігаються у тимчасову колекцію training_data з TTL 7 днів;
- production - активний режим, коли дані проходять повний pipeline обробки та зберігаються у цільові bucket-колекції;
- disabled - колектор викнений, вхідні дані відхиляються.

Життєвий цикл колектора передбачає послідовний перехід між статусами. Новий колектор створюється у статусі training з порожньою конфігурацією полів. Користувач надсилає тестові дані, система автоматично виявляє поля та їх типи. Після налаштування маппінгів на bucket колектор переводиться у production режим.

Сховища даних (Buckets) є контейнерами для зберігання метрик. Bucket визначає схему даних - набір полів з їх типами, одиницями вимірювання та описами. Два типи bucket відповідають різним методам використання.

Time Series Bucket оптимізований для потокових метрик з високою частотою надходження. Кожне значення зберігається як окремий документ з полями timestamp, key (назва метрики), value та опціональними tags. MongoDB Time Series Collections забезпечують автоматичну оптимізацію зберігання.

Tabular Bucket призначений для структурованих даних, де важлива кореляція між полями. Всі поля одного запису зберігаються в єдиному документі. Це оптимально для аналітичних запитів, що потребують одночасного доступу до кількох полів.

На даний момент система підтримує 19 типів метрик, організованих у чотири категорії:

1. Фізичні вимірювання включають time (часові інтервали), temperature, distance, weight, pressure, frequency, speed та ph. Кожен тип має набір одиниць вимірювання з коефіцієнтами конвертації.

2. Обчислювальні метрики охоплюють cpu (використання процесора), memory, storage, iops, throughput та bitrate. Базовими одиницями є cores, bytes, bytes/s тощо.

3. Електричні метрики включають voltage, amperage та power для моніторингу енергоспоживання.

4. Бізнес-метрики представлені типами gauge (безодиничні значення), ratio (пропорції 0-1), percentage (відсотки 0-100) та count (лічильники).

Нормалізація одиниць вимірювання є важливим етапом обробки даних. Дані завжди зберігаються у базових одиницях (bytes, seconds, celsius тощо), незалежно від одиниць джерела. Це забезпечує узгодженість при запитах та агрегації.

Функції convertToBase та convertFromBase перетворює значення з одиниці джерела у базову одиницю та навпаки відповідно.

Ingest Pipeline реалізує послідовну обробку вхідних даних. При надходженні HTTP POST запиту на /api/ingest/{collectorId} виконуються такі етапи:

1. Валідація колектора - перевірка існування та статусу колектора. Якщо колектор не знайдено або має статус disabled, запит відхиляється з відповідною помилкою.

2. Валідація структури даних - перевірка формату вхідних даних через Zod-схему. Запит повинен містити timestamp, data (об'єкт з метриками) та опціонально tags.

3. Збереження у training - якщо колектор у статусі training, дані зберігаються у колекцію training_data для подальшого аналізу структури. Pipeline завершується.

4. Отримання конфігурації bucket - для production-колектора визначається цільовий bucket через bucketMappings.

5. Нормалізація даних - значення конвертуються у базові одиниці відповідно до налаштувань полів колектора.

6. Фільтрація полів - залишаються лише поля, що визначені у схемі bucket. Невідомі поля ігноруються.

7. Збереження записів - дані записуються у відповідну колекцію (Time Series або Tabular).

8. Оновлення lastUsedAt - фіксується час останнього використання колектора.

Діаграма потоку даних представлена в Додатку Г.

Методи агрегації забезпечують отримання узагальнених показників з часових рядів. Система підтримує понад 30 методів:

- Базові методи: sum, avg, min, max, count, first, last - стандартні статистичні операції.
- Статистичні методи: median, mode, stddev, variance - для глибшого аналізу розподілу.
- Персентилі: p50, p75, p90, p95, p99, p99_9, p99_99 - для аналізу латентності та SLA (Service Level Agreement).
- Темпові методи: rate (швидкість зміни), increase (приріст за період), irate (миттєва швидкість).

Віконні методи: moving_avg, exponential_moving_avg - для згладжування шуму.

Агрегація виконується на рівні MongoDB через Aggregation Pipeline, що забезпечує високу продуктивність для великих обсягів даних.

Часові діапазони та інтервали визначають параметри візуалізації. Система надає набір пресетів: last5m, last15m, last30m, last1h, last3h, last6h, last12h, last24h, last7d, last30d. Кожному діапазону відповідає оптимальний інтервал агрегації для отримання 12-30 точок на графіку:

- last5m → 10s
- last1h → 5m
- last24h → 1h
- last7d → 6h

Таке налаштування забезпечує баланс між деталізацією та продуктивністю запитів.

2.4. Проєктування RESTful API

Програмний інтерфейс (API) є ключовим компонентом, що забезпечує взаємодію між клієнтською частиною та сервером, а також інтеграцію із зовнішніми системами. У цьому підрозділі описано архітектуру REST API платформи Metrix, систему автентифікації та механізми забезпечення якості.

REST (Representational State Transfer) є архітектурним стилем, що визначає набір обмежень для побудови веб-сервісів. Платформа Metrix реалізує RESTful API з дотриманням основних принципів: клієнт-серверна архітектура, відсутність стану (stateless), кешування, уніфікований інтерфейс та багатошарова система.

Для організації роутерів розроблено власний патерн ApiRouter, що розширює можливості стандартного Express Router. Клас ApiRouter інкапсулює логіку маршрутизації, валідації та документування в єдиному інтерфейсі.

Конструктор ApiRouter приймає базовий шлях та реєструє роутер у глобальному реєстрі. Методи get, post, patch, delete приймають конфігурацію маршруту, що включає: summary та description для документації, tags для групування, requireAuth для вимоги автентифікації, request з Zod-схемами для body, params та query, responses зі схемами відповідей, errors з можливими типами помилок та handler з асинхронною функцією-обробником.

Приклад застосування розробленого ApiRouter вказано в Додатку Е.

Такий підхід забезпечує co-location документації та коду. Кожен маршрут містить всю інформацію для генерації OpenAPI специфікації без додаткових файлів чи анотацій.

Структура API організована за ресурсним принципом. Базовий шлях /api розділяється на функціональні групи:

- /api/account - управління обліковим записом. Включає endpoints для автентифікації (sign-in, sign-up, sign-out) та оновлення профілю. Ці маршрути доступні без автентифікації (окрім profile).
- /api/workspaces - управління робочими просторами. Отримання списку доступних просторів та створення нових.
- /api/workspace/:workspaceId - ресурси конкретного простору. Параметр workspaceId присутній у всіх дочірніх маршрутах та визначає контекст операції.

Вкладені ресурси простору включають:

- /buckets - сховища даних з операціями CRUD та управлінням полями;
- /collectors - колектори метрик з налаштуванням статусу та маппінгів;
- /dashboards - дашборди з управлінням dashlets;
- /widgets - віджети візуалізації;
- /access-tokens - токени доступу для API.
- - /api/ingest/:collectorId - спеціальний endpoint для прийому метрик від зовнішніх систем. Використовує Bearer-автентифікацію замість session-based.

Система автентифікації реалізує два механізми залежно від типу клієнта.

Session-based автентифікація використовується для веб-інтерфейсу. При успішному вході сервер створює сесію та повертає auth-token. Токен зберігається на клієнті та передається у заголовку auth-token з кожним запитом. Сесія зберігається в колекції sessions з полями userId, authToken, createdAt та expiresAt.

Access Token автентифікація призначена для зовнішніх систем та колекторів. Токени створюються через інтерфейс управління та можуть мати термін дії або бути безстроковими. Передача відбувається через заголовок Authorization з префіксом Bearer. Функція validateAccessToken перевіряє наявність токена у базі та термін його дії.

Розмежування типів автентифікації забезпечує безпеку: session tokens мають короткий термін дії та прив'язані до браузера, тоді як access tokens призначені для автоматизованих систем та можуть бути відкликані окремо.

Валідація даних є критичним етапом обробки запитів. Zod-схеми забезпечують типобезпечну валідацію на в ході роботи програми. Функція parseRequestPayload перевіряє body, params та query параметри, за допомогою бібліотеки Zod перевіряє схему отриманих даних і у випадку неузгодженості створюється BadRequestError з детальним описом проблем.

Обробка помилок реалізована через ієрархію класів, що розширюють базовий ApiError. Кожен клас відповідає HTTP-статусу: UnauthorizedError (401), ForbiddenError (403), NotFoundError (404), BadRequestError (400), ValidationError (400), InternalServerError (500). Приклад коду помилок вказано в Додатку Д на Рис. Д.2.5.

Глобальний error handler шар роутингу перехоплює всі помилки та формує уніфіковану відповідь. ApiError повертаються з відповідним статус-кодом, невідомі помилки логуються та повертаються як Internal Server Error.

Документація API генерується автоматично з конфігурацій роутерів. При реєстрації кожного маршруту ApiRouter передає метадані до OpenAPI registry. OpenAPI специфікація доступна за шляхом /api/openapi.json. Інтерактивна документація на базі Scalar UI доступна за /api/docs та дозволяє тестувати endpoints безпосередньо з браузера. Приклад використання Scalar UI зазначенно в Додатку Ж.

2.5. Проєктування адаптивного веб-інтерфейсу

Користувацький інтерфейс є точкою взаємодії між системою та користувачем. Адаптивність інтерфейсу забезпечує однаково зручну роботу на пристроях з різними розмірами екранів - від мобільних телефонів до широкоформатних моніторів. У цьому підрозділі описано архітектуру клієнтської частини платформи Metrix.

Структура компонентів організована за принципом атомарного дизайну з модифікаціями під потреби проекту. Ієрархія включає чотири рівні.

- Базові компоненти (bricks) - найменші будівельні блоки інтерфейсу: кнопки, поля введення, чекбокси, селекти. Кожен компонент інкапсулює

стилізацію та поведінку. Директорія містить понад 40 компонентів, кожен у власній папці.

- Макети (layouts) визначають структуру сторінок. AppLayout формує базову структуру застосунку з бічною панеллю та основним контентом. SettingsLayout забезпечує двоколонкову структуру для сторінок налаштувань. Макети адаптуються залежно від розміру екрану.

- Екрани (screens) представляють повні сторінки застосунку: дашборди, налаштування колекторів, редактор віджетів, автентифікація. Кожен екран комбінує компоненти нижчих рівнів.

- Віджети (widgets) - спеціалізовані компоненти для візуалізації даних. Кожен тип віджета має view-компонент для відображення та config-компонент для налаштування.

Radix UI є основою для інтерактивних компонентів. Бібліотека надає unstyled примітиви з вбудованою доступністю (accessibility) та управлінням станом. Проект використовує понад 20 компонентів Radix: Dialog, Dropdown Menu, Accordion, Tabs, Popover, Select, Toast та інші. Обгортка над Radix забезпечує уніфіковану стилізацію через функцію cn, що об'єднує класи Tailwind.

Бібліотека Tailwind CSS версії 4 використовується для стилізації. Конфігурація визначає дизайн-токени через CSS-змінні у директиві @theme: радіуси заокруглень, кольори фону та тексту, кольори акцентів. Система тем підтримує світлий та темний режими через CSS-змінні у :root та .dark селекторах.

Адаптивність реалізована через breakpoints Tailwind та кастомний hook useIsMobile. Точка розриву 768px розділяє мобільний та десктопний режими. Hook використовує window.matchMedia для відстеження змін розміру вікна.

Бічна панель (Sidebar) демонструє адаптивну поведінку. На десктопі вона фіксована з можливістю згортання до 60px. На мобільних - відкривається як модальне вікно з повною шириною екрану.

Бібліотека React Grid Layout використовується для дашбордів з можливістю зміни розташування віджетів [1]. Бібліотека забезпечує адаптивну стітку з різними конфігураціями для різних розмірів екрану: lg (1200px), md

(996px), sm (768px), xs (480px). На мобільних віджети вибудовуються в одну колонку.

Роутинг реалізовано бібліотекою wouter - легковісною альтернативою React Router. AppRouter визначає структуру навігації з вкладеними маршрутами та розділяє публічні маршрути (sign-in, sign-up) від захищених. AuthGuard є компонентом-обгорткою, що перевіряє стан автентифікації та перенаправляє неавторизованих користувачів.

Бібліотека Redux Toolkit забезпечує централізоване управління станом [23]. Store містить 8 слайсів для різних доменів: account, workspaces, dashboards, buckets, collectors, widgets, accessTokens, bucketData. Кожен слайс включає initial state, reducers для синхронних змін та extraReducers для асинхронних операцій через createAsyncThunk.

Бібліотека Recharts використовується для візуалізації графіків. Бібліотека надає декларативні React-компоненти для різних типів діаграм: LineChart, AreaChart, BarChart, PieChart. ResponsiveContainer забезпечує автоматичне масштабування при зміні розміру контейнера.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ СИСТЕМИ МОНІТОРИНГУ

3.1. Реалізація серверної частини

Серверна частина платформи Metrix реалізована на Node.js з використанням Express 5 та TypeScript. У цьому підрозділі описано ключові аспекти програмної реалізації серверних компонентів.

Структура серверного коду організована за принципом функціонального розділення. Директорія workflows містить бізнес-логіку, controllers - HTTP endpoints, collections - операції з базою даних, middleware - проміжні обробники.

Базовий клас BaseCollection інкапсулює типові операції з MongoDB колекціями. Клас приймає типізовану колекцію та надає методи для CRUD-операцій: insertOneAndReturn для вставки з поверненням документа, findById для пошуку за ідентифікатором, updateById для оновлення та existsById для перевірки існування.

Конкретні колекції розширюють базовий клас та додають специфічну логіку. Наприклад, CollectorCollection реалізує метод safeDTO для перетворення внутрішніх документів MongoDB у DTO-об'єкти з конвертацією ObjectId у рядкові ідентифікатори.

Фабричні функції створюють екземпляри колекцій з правильним контекстом бази даних. Функція getCollectorCollection отримує посилання на базу workspace та повертає типізовану колекцію.

Workflow для обробки вхідних даних (ingest) демонструє складну бізнес-логіку. Функція ingestData координує валідацію колектора, перевірку структури даних, збереження у тренувальну колекцію або production bucket залежно від статусу. Алгоритм розгалужується на два шляхи: для timeseries bucket дані трансформуються у формат key-value, для tabular - зберігаються як вкладені документи. Детальний лістинг наведено у Додатку А.

Нормалізація даних перетворює вхідні значення з одиниць джерела у базові одиниці. Функція normalizeData ітерує по точках даних, для кожного поля

знаходить метадані у конфігурації колектора та застосовує функцію convertToBase для конвертації одиниць.

Фільтрація полів забезпечує відповідність даних схемі bucket. Функція filterFields порівнює ключі записів з дозволеними полями схеми, повертаючи відфільтрований масив та список проігнорованих полів для логування.

Система типів метрик реалізує конвертування одиниць вимірювання. Словник MetricTypes містить визначення для 19 типів з відповідними одиницями та коефіцієнтами конвертації (Рис. 3.1.).

```
        id          : MetricTypeIds.memory,
        category    : MetricCategoryTypes.computing,
        label       : 'Memory',
        description : 'Tracks RAM (Random Access Memory) usage and allocation. Ideal for monitoring application memory consumption, container memory limits, heap sizes, and system RAM utilization.',
        baseUnit    : 'bytes',
        previewRange: [536870912, 8589934592], // 512MB to 8GB
        units       : {
         bytes    : { name: 'Bytes', symbol: 'B', toBase: 1 },
         kilobytes: { name: 'Kilobytes', symbol: 'KB', toBase: 1024 },
         megabytes: { name: 'Megabytes', symbol: 'MB', toBase: 1024 * 1024 },
         gigabytes: { name: 'Gigabytes', symbol: 'GB', toBase: 1024 * 1024 * 1024 },
         terabytes: { name: 'Terabytes', symbol: 'TB', toBase: 1024 * 1024 * 1024 * 1024 },
        },
        aggregations: [
         AggregationMethods.sum,
         AggregationMethods.avg,
         ...
         AggregationMethods.p95,
         AggregationMethods.p99,
        ],
```

Рис. 3.1. Приклад опису метрики типу “Memory”

Кожен тип має базову одиницю та набір похідних одиниць з множниками toBase.

Функції convertToBase та convertFromBase забезпечують двосторонню трансформацію значень. Для температури додатково враховується зміщення (offset) при конвертації між Цельсієм та Фаренгейтом.

Безпека реалізується через bcrypt для хешування паролів. При реєстрації пароль хешується з 10 раундами salt, при автентифікації порівнюється з збереженим хешем через функцію порівняння.

Обробка помилок реалізована через глобальний middleware. Обробник розрізняє типи помилок: ApiError повертається з відповідним статус-кодом, ZodError трансформується у ValidationError, невідомі помилки логуються та повертаються як Internal Server Error.

Логування реалізовано через бібліотеку Pino з підтримкою структурованих логів. Кожен модуль має власний логер з налаштованим рівнем. HTTP-запити логуються автоматично через pino-http middleware з часом виконання та статусом відповіді.

Структура кодової бази включно з модулем серверної частини продемонстрована в Додатку З.

3.2. Реалізація клієнтської частини

Клієнтська частина платформи побудована на React 19 з TypeScript та використовує Vite як збірник. У цьому підрозділі описано ключові аспекти реалізації інтерфейсу користувача.

Архітектура компонентів базується на модульній системі з чотирма рівнями. Директорія bricks містить базові UI-елементи, layouts - структурні макети, screens - повні сторінки, widgets - компоненти візуалізації даних.

Компоненти реалізовані за принципом композиції. Базовий компонент Card складається з кількох частин: Card, CardHeader, CardContent. Патерн asChild дозволяє замінити кореневий елемент через Radix Slot, що забезпечує гнучкість при композиції. Функція cn об'єднує класи Tailwind з уникненням конфліктів через tailwind-merge.

Кастомні hooks інкапсулюють повторювану логіку. Hook useSchemaValidation інтегрує Zod-валідацію з React: приймає схему та дані, повертає об'єкт помилок при невалідних даних. Hook useDialogManager спрощує

управління станом модальних вікон, надаючи методи show та hide з підтримкою передачі props.

Hook usePersistContextValue оптимізує перерендеринг контекстів. Функція порівнює попереднє та нове значення, повертаючи кешовану версію при незмінних даних. Це запобігає зайвим перерендерингам компонентів-споживачів контексту.

Контекст автентифікації управляє станом сесії та інтегрується з Redux. AuthContextProvider при монтуванні виконує запит на відновлення сесії через fetchCurrentSession, до завершення відображає індикатор завантаження.

Redux store конфігурується через Redux Toolkit з типізованими hooks. Store містить 8 слайсів для різних доменів: account, workspaces, dashboards, buckets, collectors, widgets, accessTokens, bucketData. Типізовані useAppDispatch та useAppSelector забезпечують автодоповнення та перевірку типів.

Слайси використовують createSlice та createAsyncThunk для асинхронних операцій. Утиліта addAsyncCases зменшує дублювання коду для обробки pending, fulfilled та rejected станів асинхронних thunks.

Система тем реалізована через CSS-змінні та Context API. ThemeContextProvider зберігає вибір у localStorage та підтримує три режими: light, dark та system. При виборі system тема визначається через prefers-color-scheme media query.

Роутинг реалізовано через wouter з вкладеними маршрутами та guards. AppRouter розділяє публічні маршрути (sign-in, sign-up) та захищені (workspace, settings). AuthGuard перенаправляє неавтентифікованих користувачів на сторінку входу.

API-клієнт інкапсулює HTTP-запити з автоматичним додаванням токена автентифікації. Базова функція apiRequest отримує токен з localStorage та додає до заголовків. Модулі API групують операції за ресурсами: dashboardsApi, bucketsApi, collectorsApi тощо.

Структура кодової бази включно з модулем клієнтської частини продемонстрована в Додатку З.

3.3. Тестування та забезпечення якості

Забезпечення якості програмного забезпечення є невід'ємною частиною процесу розробки. У цьому підрозділі описано підходи до тестування, валідації та верифікації платформи Metrix.

Типізація через TypeScript є першим рівнем захисту від помилок. Статичний аналіз на етапі компіляції виявляє некоректне використання типів, відсутні властивості та невідповідність сигнатур. Строгий режим (strict mode) активовано для всіх модулів із налаштуваннями: strict, noImplicitAny, strictNullChecks, noUnusedLocals та noUnusedParameters.

Zod-валідація забезпечує перевірку даних на рівні runtime. Схеми визначаються один раз та використовуються для валідації API-запитів, відповідей та внутрішніх операцій. Інференція типів з Zod-схем гарантує синхронізацію між валідацією та TypeScript-типами.

Інтеграційне тестування API реалізовано через Node.js скрипти. Тестовий модуль виконує послідовність операцій та перевіряє результати. Функції тестування надсилають HTTP-запити до API та валідують відповіді. Генерація тестових даних забезпечує реалістичні сценарії з випадковими значеннями метрик у визначених діапазонах.

Валідація оцінки сили пароля включає перевірку різних критеріїв: мінімальна довжина, наявність великих та малих літер, цифр та спеціальних символів. Функція повертає рахунок та детальну інформацію про пройдені перевірки.

Обробка граничних випадків інтегрована в бізнес-логіку. Перевірка існування ресурсів виконується перед операціями. Система генерує відповідні помилки NotFoundError або BadRequestError залежно від контексту.

Логування критичних операцій забезпечує аудит та діагностику. Pino структурує логи у JSON-формат з контекстною інформацією про операції, ресурси та помилки. Рівні логування (debug, info, warn, error, fatal) налаштовуються для різних модулів окремо: server, controller, socket, db.

Моніторинг помилок через глобальний обробник забезпечує централізоване відстеження. Внутрішні помилки (статус >= 500) логуються з рівнем fatal, клієнтські помилки - з рівнем error.

Валідація конфігурації при запуску перевіряє наявність необхідних змінних середовища: MONGODB_URI та SESSION_SECRET. Відсутність критичних параметрів призводить до негайного завершення роботи з інформативним повідомленням.

Перевірка з'єднання з базою даних при ініціалізації виконує команду ping та логує результат. У разі невдачі застосунок завершується з кодом помилки.

Graceful shutdown забезпечує коректне завершення роботи при отриманні сигналу SIGTERM. Послідовно закриваються HTTP-сервер та з'єднання з базою даних з логуванням кожного етапу.

Документація API через OpenAPI надає можливість інтерактивного тестування. Scalar UI дозволяє виконувати запити безпосередньо з браузера з автоматичним додаванням токенів автентифікації.

Перевірка типів при збірці запобігає потраплянню помилок у production. Скрипт виконує tsc --noEmit для перевірки типів перед збіркою Vite.

Таким чином, забезпечення якості інтегровано на всіх етапах розробки - від статичного аналізу до runtime-валідації та моніторингу в production.

3.4. Демонстрація роботи системи та перспективи розвитку

У цьому підрозділі представлено результати функціонування платформи Metrix та визначено напрямки подальшого розвитку.

Запуск системи здійснюється через npm-скрипти. Команда npm run dev запускає паралельно сервер та клієнт у режимі розробки з hot-reload. Сервер стартує на порту 3080 та виводить інформацію про доступні endpoints: основний API Dashboard та OpenAPI специфікацію.

Процес налаштування моніторингу включає кілька етапів. Після автентифікації користувач потрапляє до робочого простору, де може створити колектор для збору метрик.

Створення колектора виконується через інтерфейс або API з вказанням назви та типу (ingest). Колектор створюється у статусі training. У цьому режимі система приймає тестові дані та автоматично виявляє структуру полів.

Тренувальні данні зберігаються в окрему табличку.

Надсилання тестових метрик відбувається через endpoint /api/ingest/{collectorId} з Bearer-автентифікацією. Кожен запит містить timestamp, об'єкт data з метриками та опціональні tags для групування.

Оновлення отриманих тренувальних даних відбувається в режимі реально часу для зручності користувача.

Після отримання достатньої кількості зразків користувач налаштовує типи полів та одиниці вимірювання.

Створення сховища даних (bucket), зображене в Додатку Л, включає визначення схеми з типами полів та одиницями вимірювання. Для часових рядів використовується тип timeseries, для табличних даних - tabular.

Інтерфейс відображає виявлені поля з прикладами значень (Рис. 3.2.) для зручності користувача.

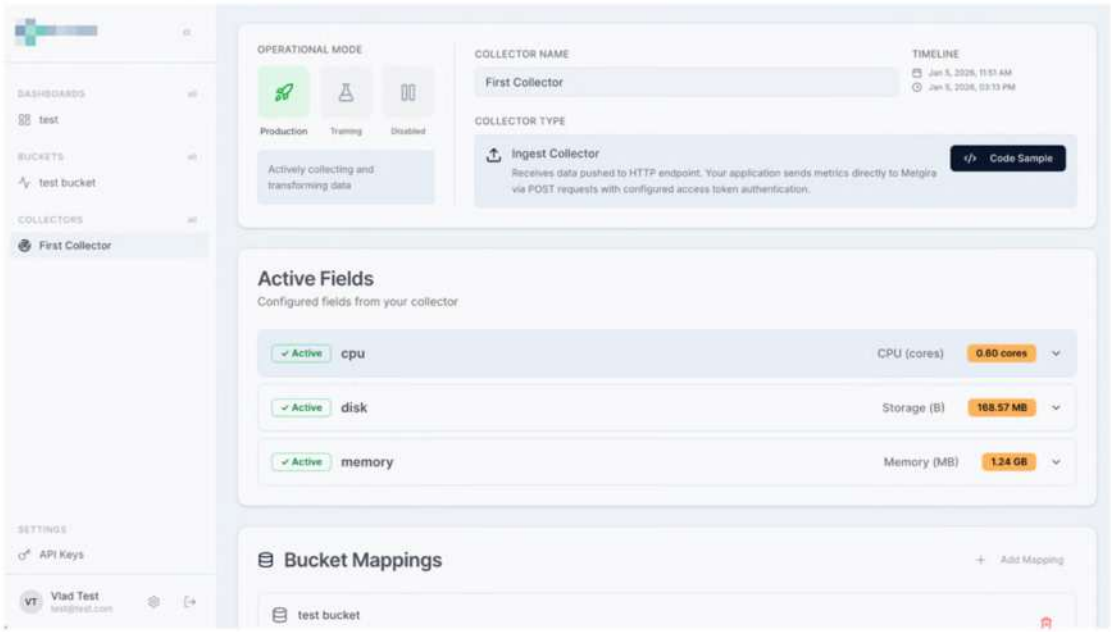


Рис. 3.2. Приклад налаштованого колектора

Налаштування маппінгу пов'язує колектор зі сховищем. Після цього колектор переводиться у статус production через API-запит і данні починають зберігатися вже безпосередньо на рівні пов’язаного data-bucket.

Візуалізація даних здійснюється через систему дашбордів та віджетів. Створення дашборду виконується з вказанням назви та опису. Додавання віджетів включає вибір типу (trends, current-state тощо), виду відображення (line, gauge, stat), конфігурацію серій даних з агрегацією та часовим діапазоном.

Приклад конфігурації віджета продемонстрована в Додатку И.

Інтерфейс дашборду, продемонстровий в Додатку К, дозволяє переміщувати та змінювати розмір віджетів через drag-and-drop. Зміни автоматично зберігаються.

Автооновлення віджетів забезпечує актуальність даних. Кожен віджет виконує API-запит з заданим інтервалом та отримує агреговані дані у форматі, готовому для візуалізації. Recharts рендерить графіки на основі отриманих даних з використанням ResponsiveContainer для адаптивності.

Адаптивність інтерфейсу забезпечує зручну роботу на різних пристроях. На мобільних пристроях бічна панель трансформується у Sheet, віджети вибудовуються у одну колонку.

ВИСНОВКИ

У даній кваліфікаційній роботі виконано дослідження та розробку модульної інтерактивної системи моніторингу з адаптивним веб-інтерфейсом. За результатами роботи можна зробити такі висновки.

Проведено комплексний аналіз існуючих систем моніторингу, зокрема Prometheus, Grafana, Datadog, Zabbix та New Relic. Виявлено їх сильні сторони та обмеження. Встановлено, що сучасні рішення часто характеризуються надмірною складністю налаштування, вузькою спеціалізацією або високою вартістю. Обґрунтовано доцільність створення платформи, що поєднує гнучкість open-source рішень із простотою комерційних продуктів.

Досліджено сучасні технології збору даних та протоколи обміну інформацією. Проаналізовано push та pull моделі збору метрик, визначено їх переваги та сфери застосування. Розглянуто протоколи HTTP, WebSocket та MQTT для передачі даних. Обґрунтовано вибір push-based підходу як більш універсального для інтеграції з гетерогенними джерелами даних, включаючи IoT-пристрої та розподілені системи.

Виконано порівняльний аналіз та обґрунтовано вибір технологічного стеку для розробки платформи. Обрано TypeScript як основну мову програмування для забезпечення типобезпеки на всіх рівнях системи. Для побудови інтерфейсу використано React 19 з Redux Toolkit для управління станом. Серверну частину реалізовано на Express 5 з інтеграцією Zod для валідації даних. Для зберігання метрик обрано MongoDB з Time Series Collections, що забезпечує оптимальну продуктивність для часових рядів.

Спроєктовано модульну архітектуру системи з підтримкою мультитенантності. Розроблено структуру монорепозиторію з трьох модулів: client, server та shared, що забезпечує повторне використання коду та узгодженість типів. Реалізовано мультитенантну архітектуру бази даних з ізоляцією даних на рівні окремих баз для кожного робочого простору. Спроєктовано систему віджетів на основі Registry Pattern з розділенням типу

даних та способу відображення. Розроблено RESTful API з власним ApiRouter pattern та автоматичною генерацією OpenAPI документації.

Реалізовано механізми збору та обробки метрик з різних джерел. Розроблено систему колекторів з підтримкою режиму training для автоматичного виявлення структури полів. Імплементовано pipeline обробки даних, що включає валідацію, нормалізацію одиниць вимірювання та фільтрацію полів. Система підтримує 19 типів метрик з автоматичною конвертацією між одиницями вимірювання. Реалізовано два типи сховищ даних: Time Series для потокових метрик та Tabular для структурованих даних.

Розроблено адаптивний веб-інтерфейс з інтерактивними дашбордами. Використано Radix UI для створення доступних компонентів із вбудованою підтримкою ARIA-атрибутів. Стилізацію реалізовано через Tailwind CSS з підтримкою світлої та темної тем. Для компонування дашбордів застосовано React Grid Layout з можливістю drag-and-drop переміщення та зміни розміру віджетів. Інтерфейс коректно відображається на пристроях різних розмірів від мобільних телефонів до широкоформатних моніторів.

Проведено тестування та оцінку ефективності розробленого рішення. Впроваджено статичний аналіз коду через TypeScript у строгому режимі та runtime-валідацію через Zod. Реалізовано структуроване логування через Pino для діагностики та моніторингу роботи системи. Продемонстровано повний цикл роботи платформи від налаштування колекторів та збору метрик до їх візуалізації на дашбордах.

Практична значущість роботи полягає у створенні функціональної платформи моніторингу, що може використовуватися для моніторингу серверної інфраструктури та застосунків, збору та візуалізації IoT-метрик, бізнес-аналітики на основі операційних даних, а також як навчальний приклад сучасної веб-архітектури.

Наукова новизна роботи полягає у розробці методу виявлення структури метрик на етапі налаштування колектора, запропонованому підході до розділення

типу даних та способу відображення у системі віджетів, а також реалізації ApiRouter pattern з інтегрованою генерацією OpenAPI специфікації.

Визначено напрямки подальшого розвитку платформи, що включають реалізацію pull-based колекторів для інтеграції з Prometheus exporters, підключення до хмарних провайдерів, впровадження системи сповіщень та аналітичних можливостей на основі машинного навчання.

Всі поставлені завдання виконано. Мета роботи - створення модульної та інтерактивної системи моніторингу з адаптивним веб-інтерфейсом - досягнута.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. JavaScript and TypeScript Trends 2024. JetBrains Developer Ecosystem Survey 2023. https://www.jetbrains.com/lp/devecosystem-2023/javascript/

2. Verma S. K., Tripathi A. K. Monitoring and Alerting for Horizontal Auto-Scaling Pods in Kubernetes Using Prometheus. International Journal of Advanced Computer Science and Applications. 2023. Vol. 14, No. 10. https://www.researchgate.net/publication/376675119

3. Observability Pulse 2024. Logz.io. 2024.https://logz.io/observability-pulse-2024/

4. Prometheus Documentation. Prometheus Authors. https://prometheus.io/docs/ (дата звернення: 01.12.2025).

5. Unified Monitoring for Microservices: Implementing Prometheus and Grafana for Scalable Solutions. Journal of Artificial Intelligence, Machine Learning & Data Science. 2024. Vol. 2, No. 3. P. 45-58. https://www.researchgate.net/publication/383585960

6. Grafana Documentation. Grafana Labs. https://grafana.com/docs/ (дата звернення: 01.12.2025).

7. Bach B., Freeman E., Abdul-Rahman A., et al. Dashboard Design Patterns. IEEE Transactions on Visualization and Computer Graphics. 2023. Vol. 29, No. 1. P. 342-352. https://www.researchgate.net/publication/363869631

8. DataDog Documentation. Datadog, Inc. https://docs.datadoghq.com/ (дата звернення: 01.12.2025).

9. New Relic Documentation. New Relic, Inc. https://docs.newrelic.com/ (дата звернення: 01.12.2025).

10. Observability in Microservices: An In-Depth Exploration of Frameworks, Challenges, and Deployment Paradigms. IEEE Access. 2025. Vol. 13. P. 1-15. https://www.researchgate.net/publication/390903567

11. Giamattei L., Guerriero A., Pietrantuono R., et al. Monitoring tools for DevOps and microservices: A systematic grey literature review. Journal of Systems and Software. 2024. Vol. 208. P. 111906. https://doi.org/10.1016/j.jss.2023.111906

12. Упіров В. В. Застосування систем моніторингу в інформаційних технологіях для підвищення ефективності електронної комерції. Економіка і бізнес: нові реалії, технологічні тренди, цифрові інновації: збірник матеріалів Міжнародної науково-практичної конференції здобувачів вищої освіти та молодих вчених (Львів-Ірпінь, 15 травня 2025 р.). Львів: СПДФО Омельченко В. Г., 2025. С. 242–244.

13. InfluxDB Documentation. InfluxData. https://docs.influxdata.com/ (дата звернення: 01.12.2025).

14. Tripathi P., Miraz M. H. Comparative Analysis of MongoDB and InfluxDB forTime Series Data Management in IoT Environments: A Study on Performance, Scalability, and Concurrency. IEEE International Conference on Computing, Electronics & Communications Engineering (iCCECE). 2023. P. 1-6. https://ieeexplore.ieee.org/document/10384962/

15. ThingsBoard Documentation. ThingsBoard, Inc. https://thingsboard.io/docs/ (дата звернення: 01.12.2025).

16. De Nardis L., Mohammadpour A., Caso G., Ali-Vehmas T., Di Benedetto M.-G. Internet of Things platforms for academic research and development: A critical review. Applied Sciences. 2022. Vol. 12. No. 4. DOI: 10.3390/app12042172 https://doi.org/10.3390/app12042172

17. Луценко І. А. Дослідження Node.js мікросервісів для побудови масштабованих веб-додатків. Вісник Національного технічного університету "ХПІ". 2023. № 2. С. 45-52.

18. Research on cloud-native monitoring system based on Prometheus. International Conference on Computer Science and Communication Technology. 2024. P. 1-8. https://www.researchgate.net/publication/380383452

19. RFC 6455. The WebSocket Protocol. IETF. 2011. https://tools.ietf.org/html/rfc6455

20. InfluxDB vs MongoDB for Time Series Workloads. InfluxData Technical Report. 2023. https://www.influxdata.com/blog/influxdb-is-27x-faster-vs-mongodb-for-time-series-workloads/

21. MongoDB Time Series Collections Documentation. MongoDB, Inc. https://www.mongodb.com/docs/manual/core/timeseries-collections/ (дата звернення: 01.12.2025).

22. React Documentation. Meta Platforms, Inc. https://react.dev/ (дата звернення: 01.12.2025).

23. 2023 JavaScript Rising Stars. Best of JS. 2024. https://risingstars.js.org/2023/en

24. Bulletproof React. Alan Alickovic. GitHub Repository. 2023. https://github.com/alan2207/bulletproof-react

25. Veeri V. State Management in React: Redux vs Zustand - A Comprehensive Guide. International Journal of Scientific Research in Computer Science, Engineering and Information Technology. 2024. Vol. 10, No. 6. P. 211-218. https://www.researchgate.net/publication/385694701

26. Redux Toolkit Documentation. Redux Team. https://redux-toolkit.js.org/ (дата звернення: 01.12.2025).

27. Recharts Documentation. Recharts Group. https://recharts.org/ (дата звернення: 01.12.2025).

28. Bostock M., Ogievetsky V., Heer J. D3: Data-Driven Documents. IEEE Transactions on Visualization and Computer Graphics. 2011. Vol. 17. No. 12. P. 2301-2309. DOI: 10.1109/TVCG.2011.185 https://doi.org/10.1109/TVCG.2011.185

29. Упіров В. В. Застосування веб-технологій для візуалізації даних. Електронні інформаційні ресурси: створення, використання, доступ та управління: збірник матеріалів Міжнародної науково-практичної Інтернет-конференції (20-21 листопада 2025 р.). Суми/Вінниця: НІКО/КЗВО «Вінницька академія безперервної освіти», 2025. С. 365–366.

30. Radix UI Documentation. WorkOS. https://www.radix-ui.com/ (дата звернення: 01.12.2025).

31. Vite Documentation. Evan You and Vite Contributors. https://vitejs.dev/ (дата звернення: 01.12.2025).

32. wouter Documentation. Alexey Raspopov. https://github.com/molefrog/wouter (дата звернення: 01.12.2025).

33. react-grid-layout Documentation. React Grid Layout Contributors. https://github.com/react-grid-layout/react-grid-layout (дата звернення: 01.12.2025).

34. Toasa R. M., González M., Arnau D., Guamán D. Data Visualization Techniques for Real-Time Information - A Custom and Dynamic Dashboard for Analyzing Surveys' Results. IEEE World Forum on Internet of Things. 2019. P. 1-6. https://doi.org/10.1109/WF-IoT.2019.8767348

35. Cherny B. Programming TypeScript: Making Your JavaScript Applications Scale. Sebastopol: O'Reilly Media, 2019. 322 p. ISBN: 978-1492037651 https://www.oreilly.com/library/view/programming-typescript/9781492037644/

36. TypeScript Documentation. Microsoft Corporation. https://www.typescriptlang.org/docs/ (дата звернення: 01.12.2025).

37. Zod Documentation. Colin McDonnell. https://zod.dev/ (дата звернення: 01.12.2025).

38. OpenAPI Specification. OpenAPI Initiative. https://spec.openapis.org/oas/v3.1.0 (дата звернення: 01.12.2025).

39. Tailwind CSS Documentation. Tailwind Labs Inc. https://tailwindcss.com/docs (дата звернення: 01.12.2025).

40. Express.js Documentation. OpenJS Foundation. https://expressjs.com/ (дата звернення: 01.12.2025).

41. Луценко І. А. Мікросервісна архітектура в хмарних програмно-апаратних комплексах з використовуванням технології Node.JS: кваліфікаційна робота бакалавра. Київ: КПІ ім. Ігоря Сікорського, 2021. https://ela.kpi.ua/items/13ff1ddf-f130-47b4-aae2-fd1313b65ab9

42. Lombardi A. WebSocket: Lightweight Client-Server Communications. Sebastopol: O'Reilly Media, 2015. 88 p. ISBN: 978-1449369279 https://www.oreilly.com/library/view/websocket/9781449369262/

43. Упіров В. В. Архітектура веб-орієнтованої платформи моніторингу з real-time оновленням даних. Інновації та перспективні шляхи розвитку інформаційних технологій (ІПШРІТ-2025): збірник тез доповідей IV Міжнародної науково-практичної інтернет-конференції (25 листопада 2025 р., м. Черкаси). Черкаси: ЧДТУ, 2025. С. 249–251.

44. Banker K., Bakkum P., Verch S., Garrett D., Hawkins T. MongoDB in Action. 2nd ed. Shelter Island: Manning Publications, 2016. 480 p. https://www.manning.com/books/mongodb-in-action

45. Pino Logger Documentation. Pino Contributors. https://getpino.io/ (дата звернення: 01.12.2025).

46. Duldulao D. N., Cabagnot R. J. Practical Enterprise React: Become an Effective React Developer in Your Team. Berkeley: Apress, 2021. 330 p. https://link.springer.com/book/10.1007/978-1-4842-6975-6

ДОДАТКИ

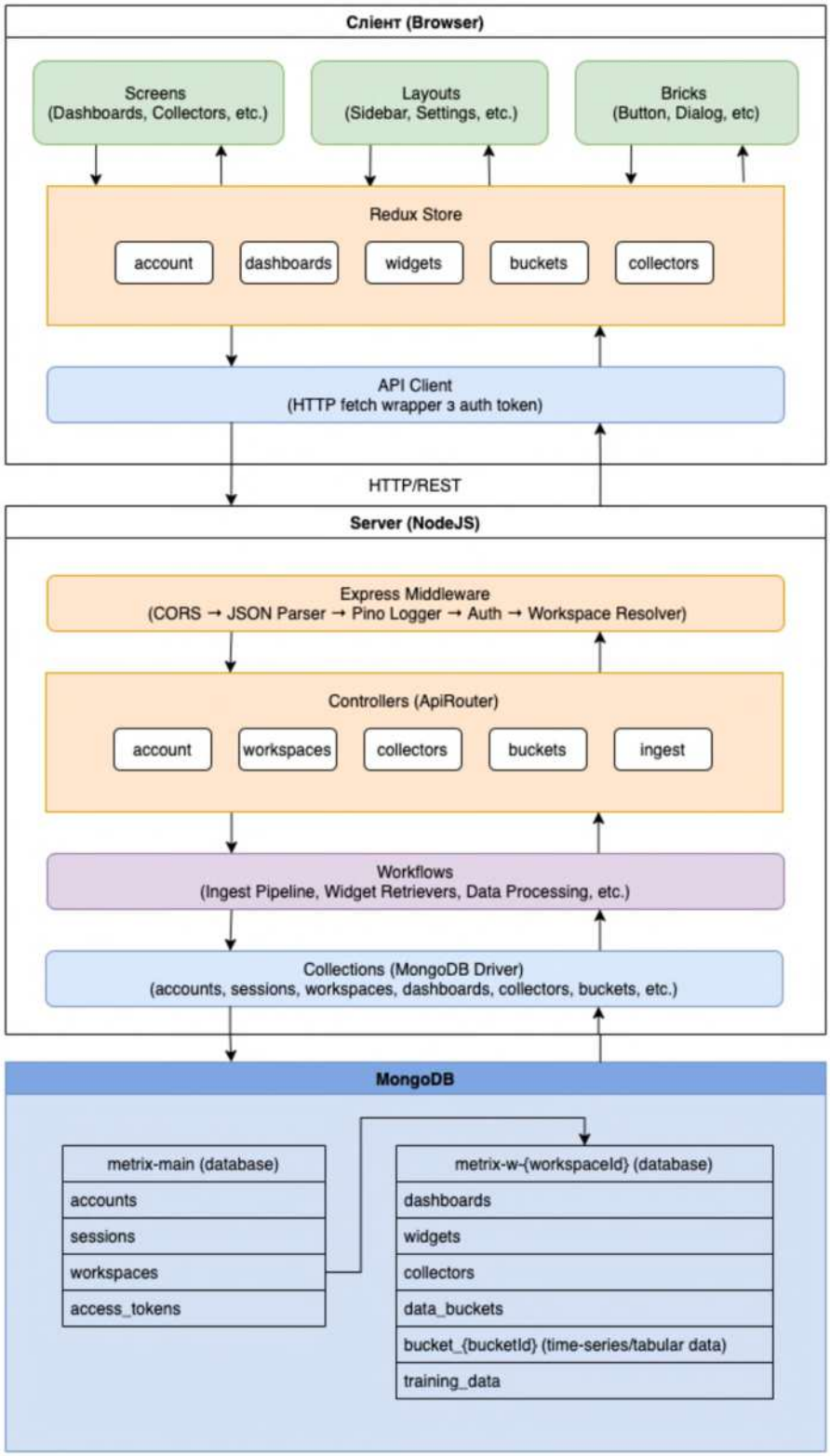
Додаток А

Порівняльний аналіз систем моніторингу

Критерій	Prometheus + Grafana	DataDog	New Relic	Zabbix	InfluxDB (TICK)
Модульність та розширюваність	200+ exporters, плагіни Grafana	700+ інтеграцій	Агенти	Шаблони, обмежені модулі	Telegraf плагіни, тільки time-series
Підтримка гетерогенних джерел	Інфраструктура, застосунки	Повний стек	Повний стек	Переважно інфраструктура	Time-series, IoT
Гнучкість візуалізації	Висока, 150+ джерел	Вбудовані дашборди	Вбудовані дашборди	Базові графіки	Обмежена
Простота розгортання	Складна (2+ компоненти)	Проста (SaaS)	Проста (SaaS)	Помірна (1 компонент)	Складна (4 компоненти)
Вартість	Безкоштовна (open-source)	Дорога (SaaS-підписка)	Дорога (SaaS-підписка)	Безкоштовна (open-source)	Безкоштовна (open-source)
Масштабованість	Обмежена (локальне сховище)	Автоматична (хмара)	Автоматична (хмара)	Обмежена	Гірша під навантаженням

Джерело: [розроблено автором]

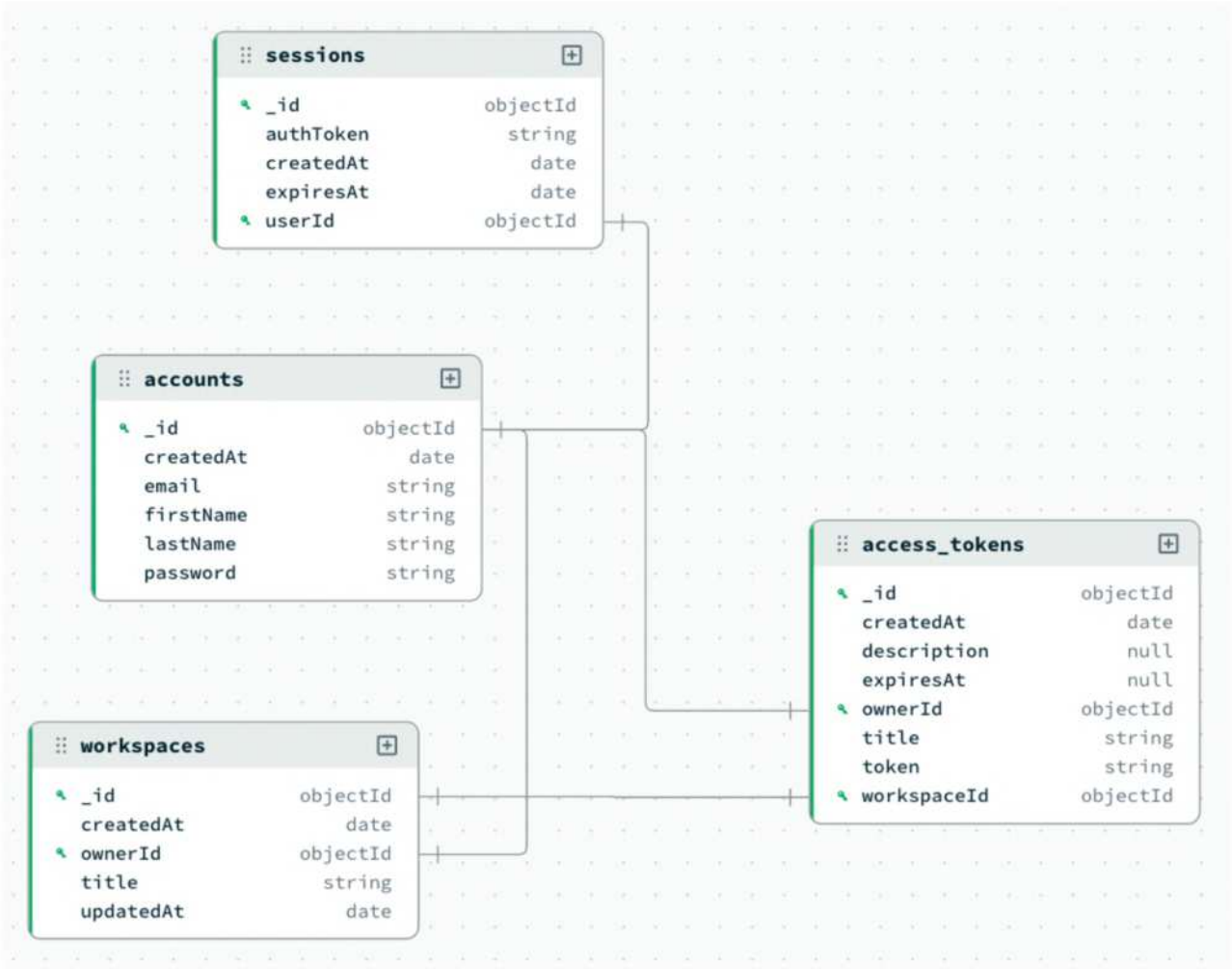
Додаток Б



Джерело: [розроблено автором]

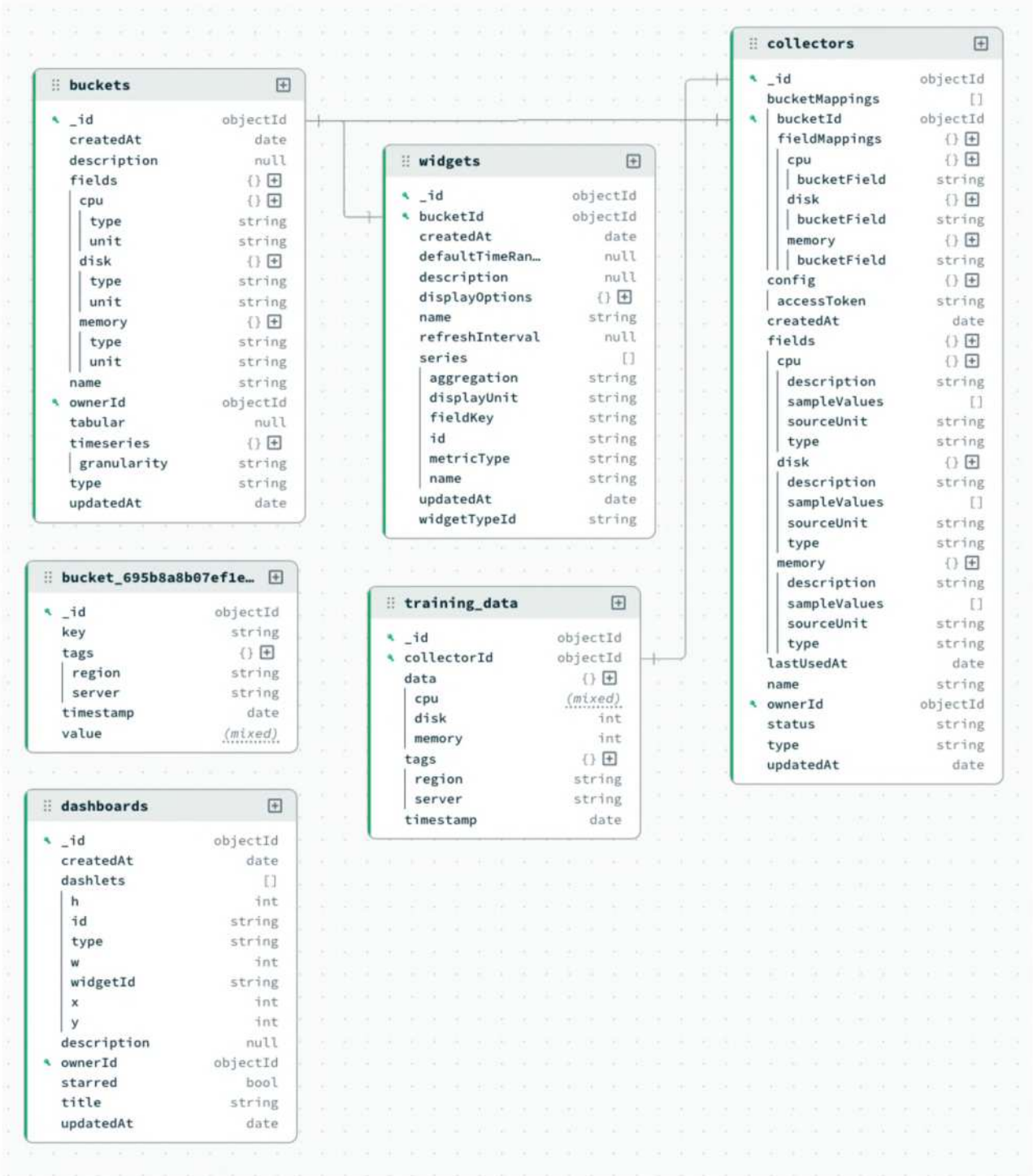
Рис. Б.2.1. Загальна архітектура системи Metrix

Додаток В



Джерело: [розроблено автором]

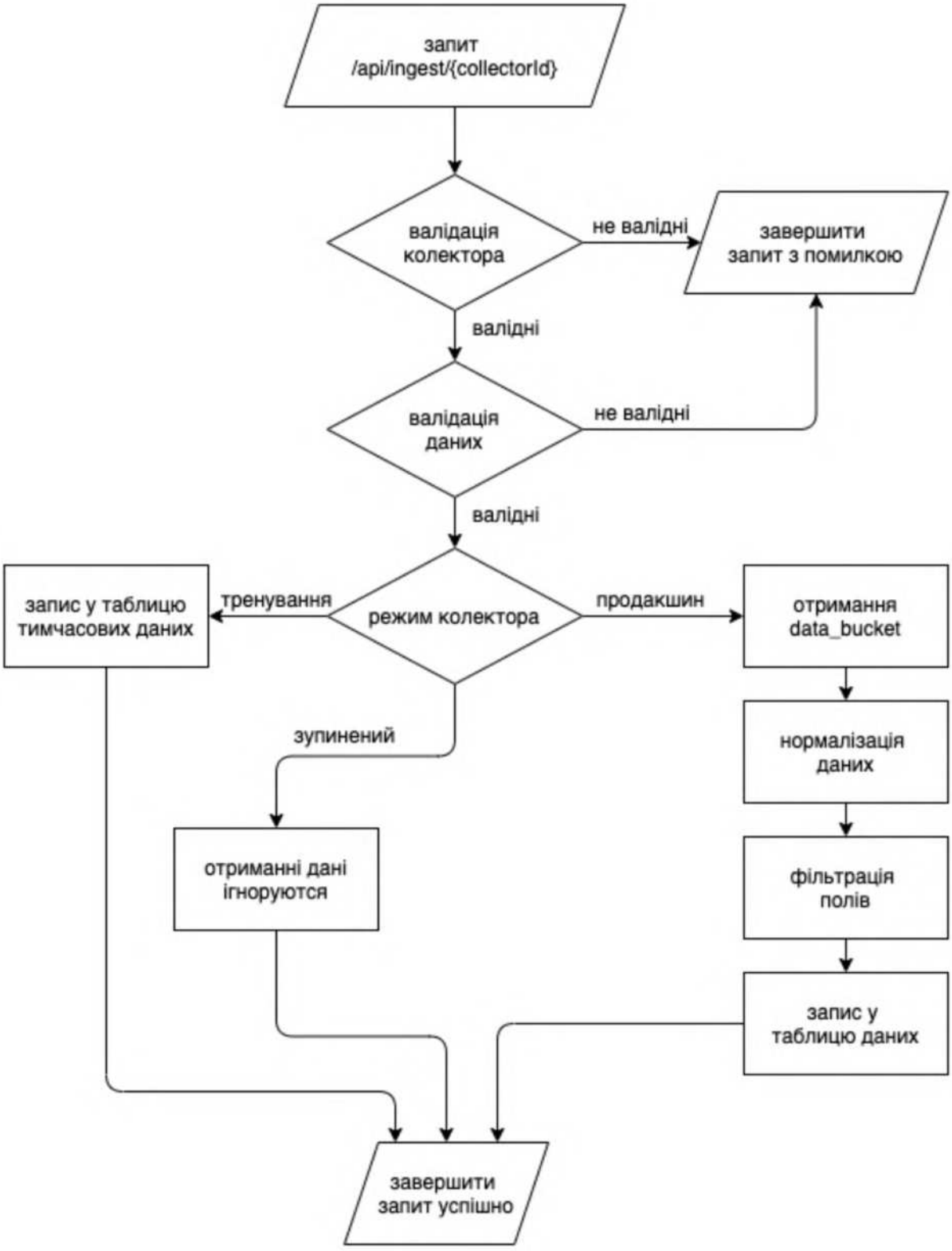
Продовження додатка В



Джерело: [розроблено автором]

Рис. В.2.2. Схема бази даних платформи

Додаток Г



Джерело: [розроблено автором]

Рис. Г.2.3. Діаграма потоку даних Ingest Pipeline

```
export class BadRequestError extends ApiError {
  constructor(message = 'Bad Request', details?: unknown) {
    super(400, CommonErrorCode.BAD_REQUEST, message, details)
  }
}

export class ValidationError extends ApiError {
  constructor(message = 'Validation Error', details?: unknown) {
    super(400, CommonErrorCode.VALIDATION_ERROR, message, details)
  }
}

export class NotFoundError extends ApiError {
  constructor(message = 'Resource Not Found', details?: unknown) {
    super(404, CommonErrorCode.NOT_FOUND, message, details)
  }
}

export class ForbiddenError extends ApiError {
  constructor(message = 'Forbidden', details?: unknown) {
    super(403, CommonErrorCode.FORBIDDEN, message, details)
  }
}

export class InternalServerError extends ApiError<object> {
  public readonly id: string

  constructor(message = 'Something went wrong!', details?: object) {
    const id = uuid()

    super(500, CommonErrorCode.INTERNAL_SERVER_ERROR, `ISE [${ id }]: ${ message
}`, details)

    this.id = id
  }

  toJSON() {
    return {
      ...super.toJSON(),
      id: this.id,
    }
  }
}
```

Джерело: [розроблено автором]

Рис. Д.2.4. Приклад класів API помилок

Додаток Е

```
// ============================================================================
// PATCH /api/workspace/:workspaceId/collectors/:collectorId
// ============================================================================
export const collectorsRouter = new ApiRouter('/workspace/:workspaceId/collectors')

const BasePathParamsSchema = z
  .object({
    workspaceId: mongoID,
  })

const CollectorPathParamsSchema = z
  .object({
    workspaceId: mongoID,
    collectorId: mongoID,
  })

type BasePathParams = z.infer<typeof BasePathParamsSchema>
type CollectorPathParams = z.infer<typeof CollectorPathParamsSchema>

collectorsRouter.patch<UpdateCollectorDTO, CollectorPathParams>('/:collectorId', {
  summary    : 'Update a collector',
  description: 'Update an existing collector',
  tags       : ['Collectors'],
  requireAuth: true as const,
  request    : {
    body  : UpdateCollectorDTOSchema,
    params: CollectorPathParamsSchema,
  },
  responses: {
    200: {
      description: 'Collector updated successfully',
      schema     : CollectorDTOSchema,
    },
  },
  errors: [
    ForbiddenError,
    NotFoundError,
  ],
  handler: async ({ params, body }): Promise<CollectorDTO> => {
    const workspaceId = toMongoId(params.workspaceId)
    const collectorId = toMongoId(params.collectorId)

    const collector = await updateCollector(workspaceId, collectorId, body)

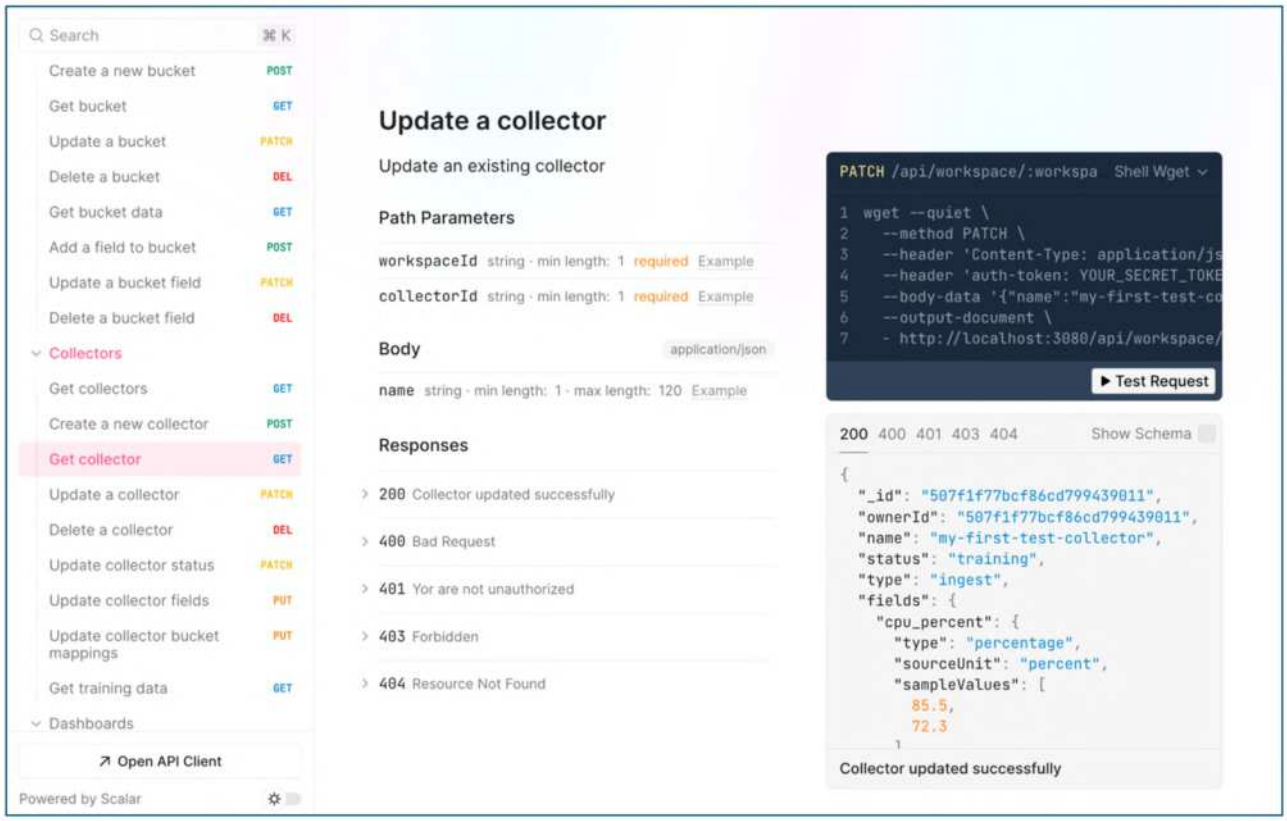
    if (!collector) {
      throw new NotFoundError('Collector not found')
    }

    return collector
  },
})
```

Джерело: [розроблено автором]

Рис. Е.2.5. Приклад використання ApiRouter

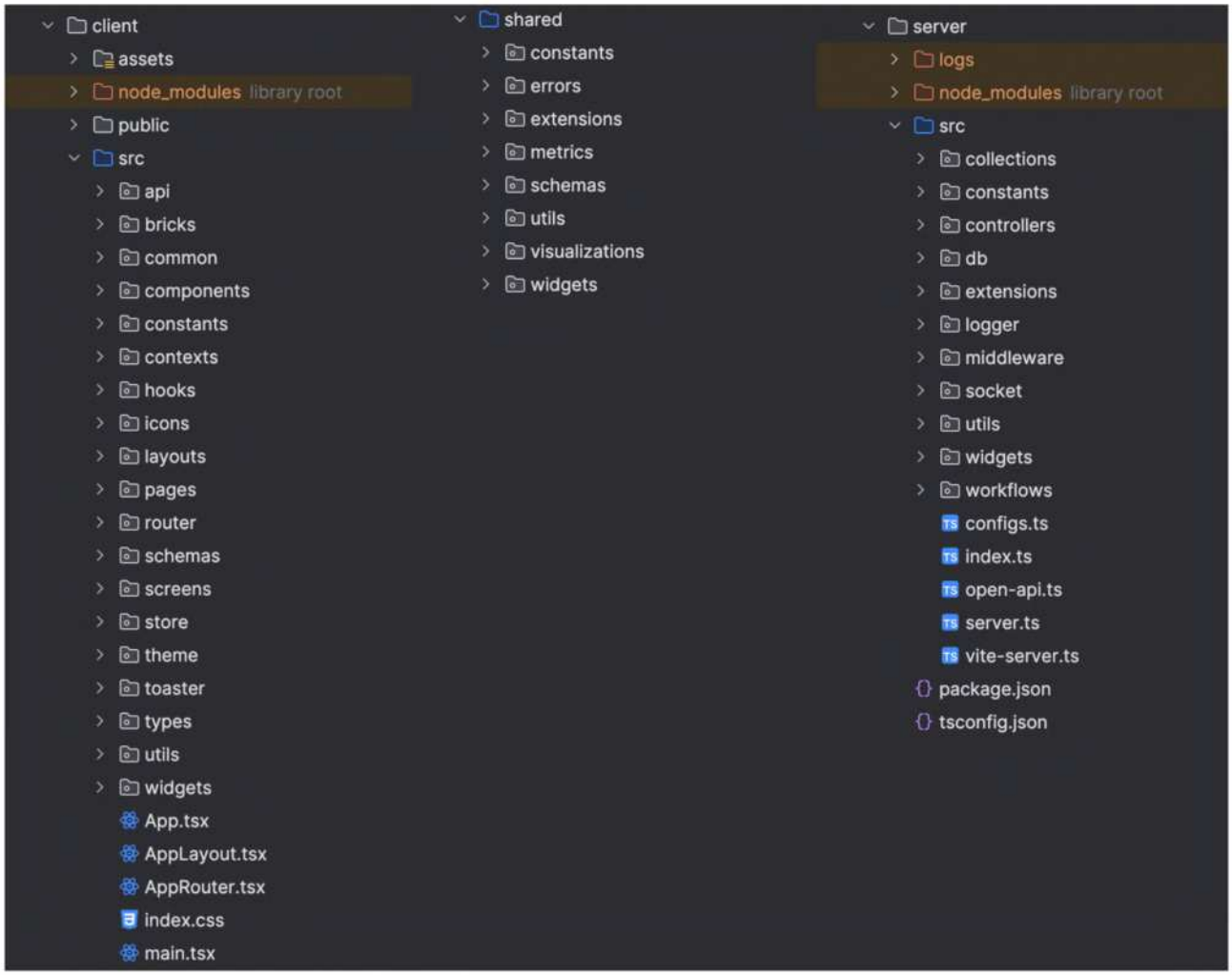
Додаток Ж



Джерело: [розроблено автором]

Рис. Ж.2.6. Приклад перегляду автоматично згенерованої API документації за допомогою Scalar UI

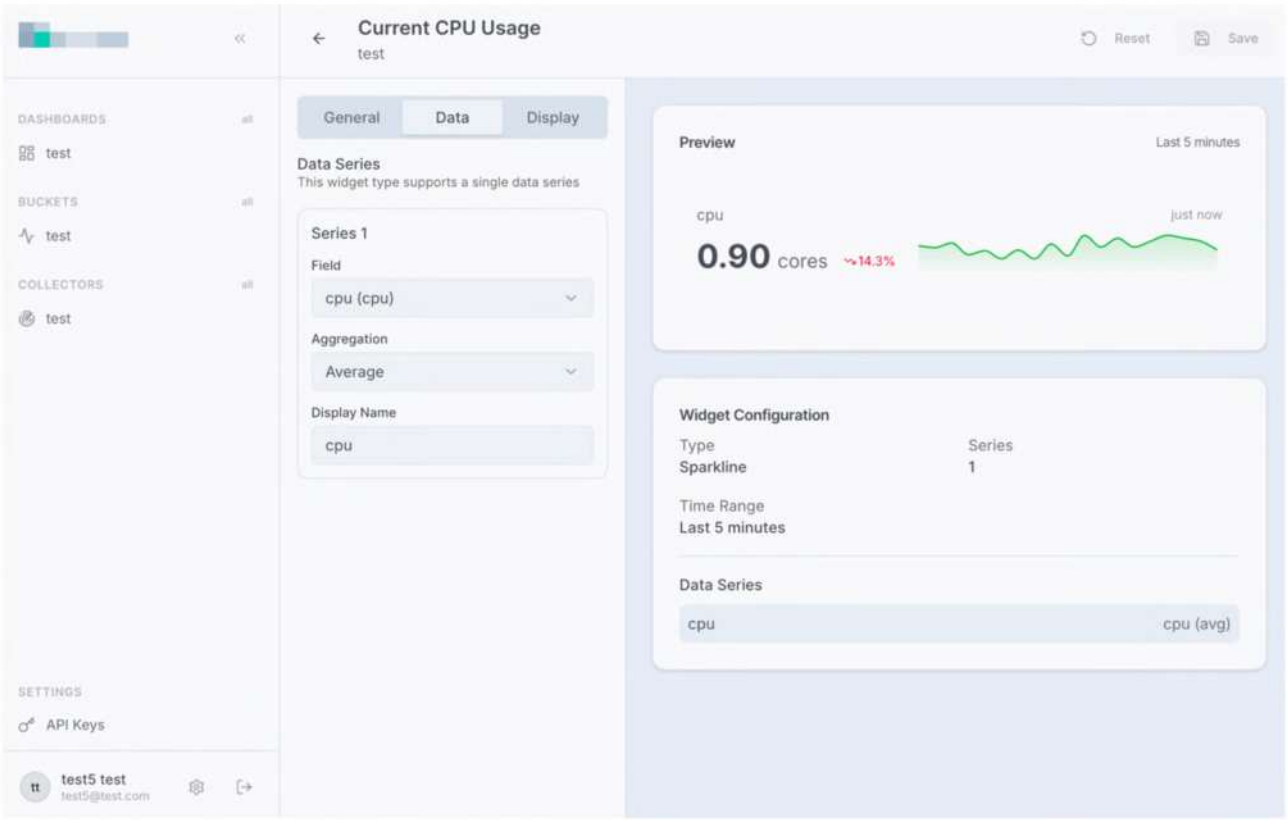
Додаток З



Джерело: [розроблено автором]

Рис. 3.3.3. Структура кодової бази платформи що розробляється

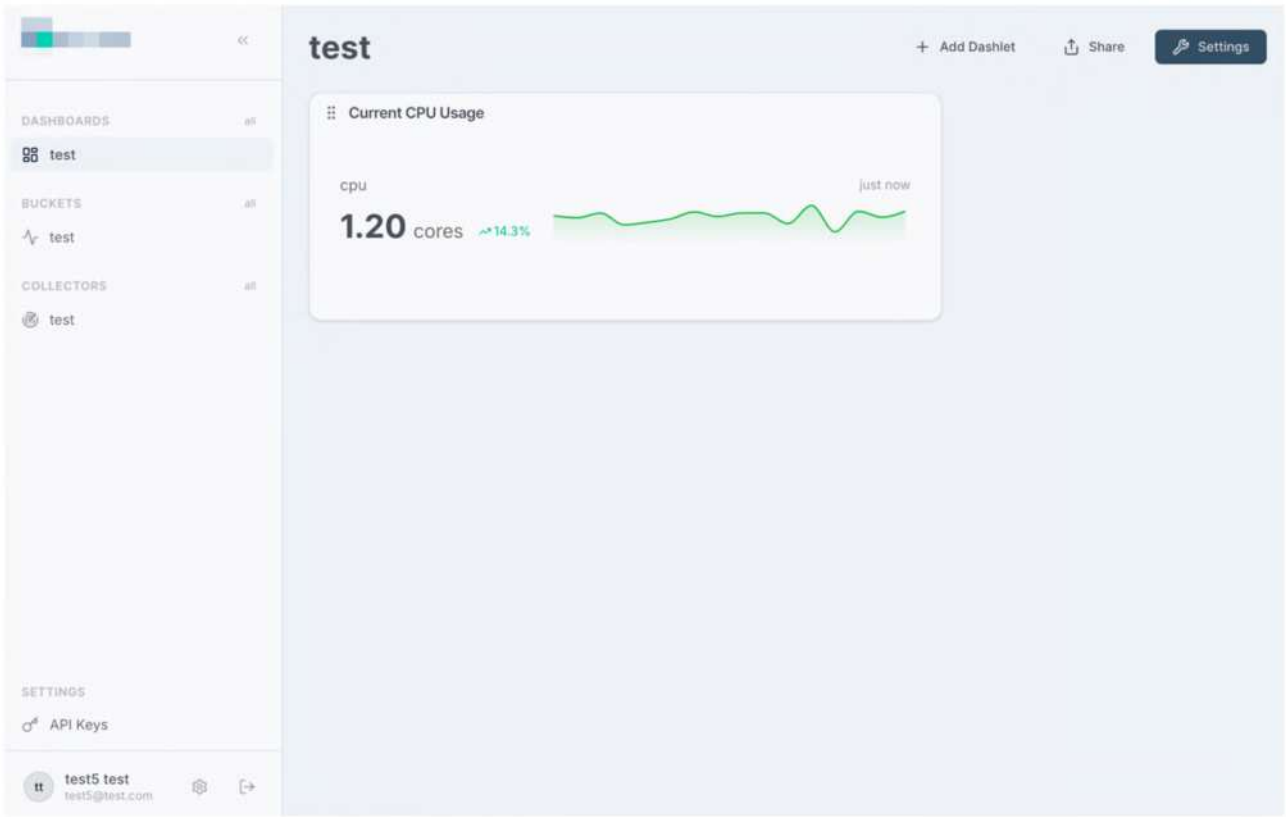
Додаток И



Джерело: [розроблено автором]

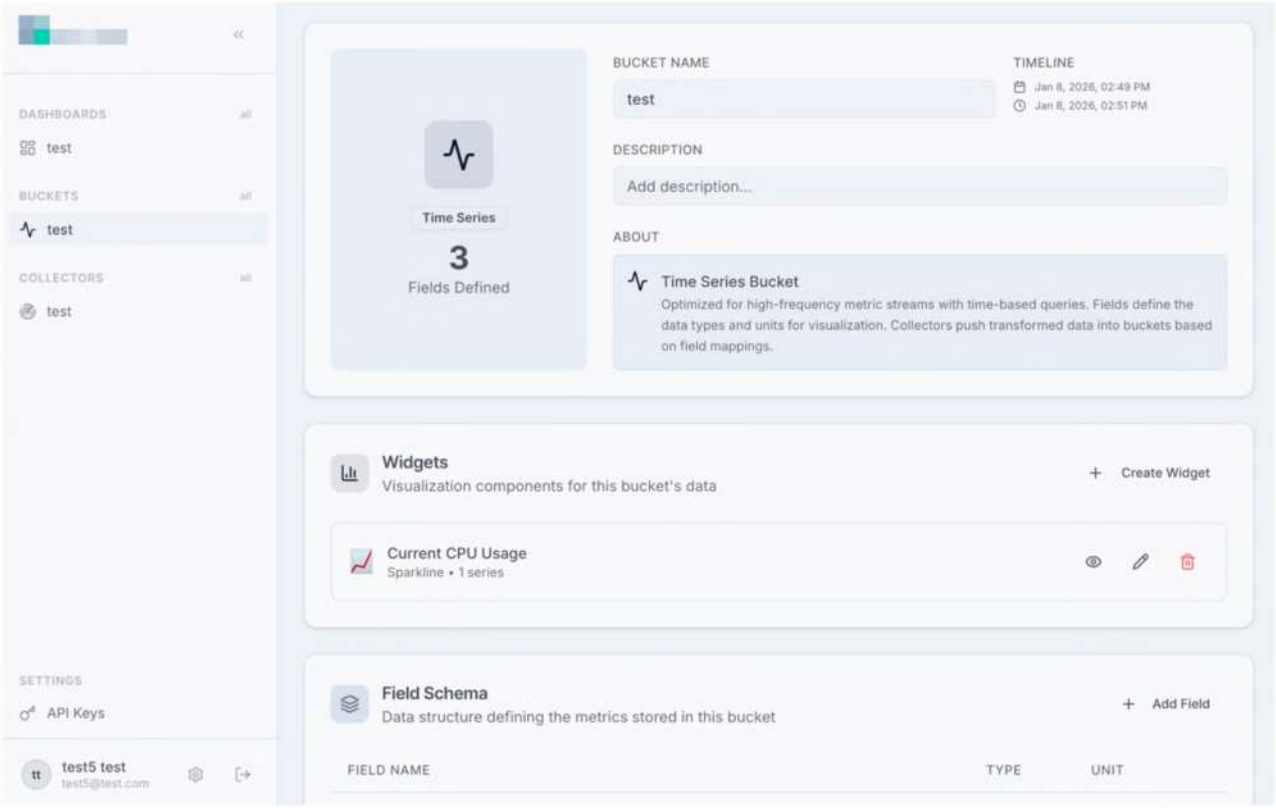
Рис. И.3.4. Екран налаштування віджету

Додаток К



Джерело: [розроблено автором]

Рис. К.3.5. Екран дешборду



Додаток Л

Джерело: [розроблено автором]

Рис. Л.3.6 Екран налаштування data-bucket

Додаток М

Сертифікати участі в наукових конференціях



МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
МІНІСТЕРСТВО ФІНАНСІВ УКРАЇНИ

Сертифікат

Володимир УПІРОВ

взяв(ла) участь у роботі

І Міжнародної науково-практичної конференції здобувачів вищої освіти та молодих вчених
«ЕКОНОМІКА І БІЗНЕС: НОВІ РЕАЛІЇ, ТЕХНОЛОГІЧНІ ТРЕНДИ, ЦИФРОВІ ІННОВАЦІЇ»
ECONOMY AND BUSINESS: NEW REALITIES, TECHNOLOGICAL TRENDS, DIGITAL INNOVATIONS

Андрій ГУКАЛЮК
Перший проректор

15 травня 2025 р.

Олександра СМІРНОВА
Проректор з наукової роботи

Продовження додатка М



Міністерство освіти і науки України
Вінницький національний технічний університет
Національна академія Державної прикордонної служби України ім. Богдана Хмельницького
Вінницький національний медичний університет ім. М.І. Пирогова
Державний заклад вищої освіти «Університет менеджменту освіти»
Державний податковий університет
Комунальний заклад вищої освіти "Вінницька академія безперервної освіти"
Сумський обласний інститут післядипломної педагогічної освіти
Інститут комп'ютерних систем і технологій "Індустрія 4.0" ім. П. Н. Платонова
Університет Бельсько-Бяльський (Польща)
Люблінська політехніка (Польща)

СЕРТИФІКАТ

засвідчує, що

Упіров Володимир Володимирович

взяв (взяла) участь у

Міжнародній науково-практичній Інтернет-конференції

«Електронні інформаційні ресурси: створення, використання, доступ та управління»

0.3 кредита ECTS

Степан ДРОВОЗЮК
Ректор КЗВО «Вінницька академія безперервної освіти»

Вінниця, 20-21 листопада 2025

Наказ №121 від 25.11.2025

Номер сертифіката 497.178

Продовження додатка М



ЧДТУ

NOOSPHERE

СЕРТИФІКАТ

учасника IV Міжнародної науково-практичної Інтернет-конференції

ІННОВАЦІЇ ТА ПЕРСПЕКТИВНІ ШЛЯХИ РОЗВИТКУ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Володимир Упіров

Голова оргкомітету д.т.н., проф. Прокопенко Т.О.

25 листопада 2025 року