

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

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

**«Вебсистема моніторингу та сповіщення про стан
серверного обладнання»**

Виконав студент групи ІПЗ-22-2
ЯРОШИНСЬКИЙ Станіслав Валентинович

Керівник роботи:
ЧИЖМОТРЯ Олена Геннадіївна

Рецензент:
ЄФРЕМОВ Юрій Миколайович

Житомир – 2026

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

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна Вакалюк

«27» лютого 2026 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **ЯРОШИНСЬКИЙ Станіслав Валентинович**

Керівник роботи: **ЧИЖМОТРЯ Олена Геннадіївна**

Тема роботи: **«Вебсистема моніторингу та сповіщення про стан серверного обладнання»**, затверджена Наказом закладу вищої освіти від **«27» лютого 2026 р., № 92/с**

Вихідні дані для роботи: розробка вебсистеми моніторингу та сповіщення про стан серверного обладнання, що автоматизує збір апаратних метрик, дозволить проактивно виявляти критичні збої у реальному часі і повністю цифровізувати безперервний контроль корпоративної ІТ-інфраструктури.

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

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Чижмотря О.Г.	01.03.2026р.	01.03.2026р
2	Чижмотря О.Г.	01.03.2026р	01.03.2026р
3	Чижмотря О.Г.	01.03.2026р	01.03.2026р

РЕФЕРАТ

Випускна кваліфікаційна робота присвячена розробці вебсистеми моніторингу та сповіщення про стан серверного обладнання. Робота складається зі вступу, 3-х розділів, висновків, списку використаних джерел із 32 найменувань (3 стор.), 7 додатків (16 стор.), 6 таблиць та 32 ілюстрацій по тексту. Повний обсяг роботи становить 93 сторінки, з них 68 сторінок основного тексту.

Метою роботи є розробка програмного комплексу, що забезпечує автоматизований збір системних апаратних метрик (CPU, RAM, Disk, Network та опціонально GPU), виявлення інцидентів у реальному часі, локальну буферизацію телеметрії при втраті мережевого з'єднання, надсилання сповіщень у месенджер та інтерактивну візуалізацію стану контрольованої ІТ-інфраструктури.

У роботі проведено аналіз предметної області та існуючих корпоративних систем моніторингу, визначено функціональні й нефункціональні вимоги, спроектовано розподілену архітектуру за моделлю «Агент – Сервер», поліглотну базу даних і реактивні алгоритми роботи системи. Реалізовано клієнтську частину у вигляді SPA-застосунку на React і серверну частину на FastAPI з використанням реляційної СУБД PostgreSQL, спеціалізованої time-series бази InfluxDB та Docker Compose. Створено автономного програмного агента з механізмом відмовостійкості на Python (із вбудованою SQLite). Проведено наскрізне тестування інтерфейсу (E2E), навантажувальне тестування серверного ядра та автоматизоване CI/CD-розгортання.

КЛЮЧОВІ СЛОВА: АГЕНТ, ВЕБСИСТЕМА, МОНІТОРИНГ, СЕРВЕР, ТЕЛЕМЕТРІЯ.

					ІПЗ.КР.Б – 121 – 26 – ПЗ			
Змн.	Арк.	№ докум.	Підпис	Дата	Вебсистема моніторингу та сповіщення про стан серверного обладнання	Літ.	Арк.	Аркушів
Розроб.		Ярошинський С.В.						
Керівник		Чижмотря О.Г.					3	93
Рецензент		Єфремов Ю.М.				Житомирська політехніка, група ІПЗ-22-2		
Зав. каф.		Вакалюк Т.А.						

ABSTRACT

The bachelor's qualification work is devoted to the development of a web system for monitoring and alerting about the state of server equipment. The work consists of an introduction, 3 chapters, conclusions, a list of references of 32 items (3 pages), 7 appendices (16 pages), 6 tables, and 32 illustrations within the text. The total volume of the thesis is 93 pages, including 68 pages of the main text.

The aim of the work is to develop a software complex that provides automated collection of system hardware metrics (CPU, RAM, Disk, Network, and optionally GPU), real-time incident detection, local telemetry buffering in case of network connection loss, sending notifications to a messenger, and interactive visualization of the controlled IT infrastructure state.

The work includes an analysis of the subject area and existing corporate monitoring systems, defines functional and non-functional requirements, and designs a distributed "Agent-Server" architecture, a polyglot database, and reactive system algorithms. The client part was implemented as a React SPA application, and the server part was built on FastAPI utilizing the PostgreSQL relational database, the specialized InfluxDB time-series database, and Docker Compose. An autonomous software agent with a fault-tolerance mechanism was developed in Python (with built-in SQLite). End-to-end (E2E) interface testing, load testing of the server core, and automated CI/CD deployment were performed.

KEYWORDS: AGENT, MONITORING, SERVER, TELEMETRY, WEB SYSTEM.

					ИПЗ.КР.Б – 121 – 26 – ИПЗ	Арк.
						4

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	6
ВСТУП.....	7
РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО ВЕБСИСТЕМИ МОНІТОРИНГУ ТА СПОВІЩЕННЯ ПРО СТАН СЕРВЕРНОГО ОБЛАДНАННЯ	10
1.1 Постановка задачі.....	10
1.2 Аналіз існуючих систем моніторингу	12
1.3 Вибір архітектури вебсистеми	16
1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	21
Висновки до першого розділу	25
РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБСИСТЕМИ МОНІТОРИНГУ ТА СПОВІЩЕННЯ ПРО СТАН СЕРВЕРНОГО ОБЛАДНАННЯ.....	27
2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи	27
2.2 Розробка бази даних вебсистеми	30
2.3 Проєктування алгоритмів роботи вебсистеми	36
2.4. Реалізація вебсистеми моніторингу	40
Висновки до другого розділу	49
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ВЕБСИСТЕМОЮ МОНІТОРИНГУ ТА СПОВІЩЕННЯ ПРО СТАН СЕРВЕРНОГО ОБЛАДНАННЯ	51
3.1 Порядок встановлення та налаштування параметрів системи	51
3.2 Структура інтерфейсу вебсистеми	54
3.3 Тестування роботи системи	68
Висновки до третього розділу.....	71
ВИСНОВКИ.....	73
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	75
ДОДАТКИ.....	78

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

ACID – Atomicity, Consistency, Isolation, Durability, набір базових властивостей транзакцій баз даних (атомарність, узгодженість, ізолюваність, довговічність).

API – Application Programming Interface, прикладний програмний інтерфейс.

CI/CD – Continuous Integration / Continuous Deployment, безперервна інтеграція та безперервне розгортання.

CPU – Central Processing Unit, центральний процесор.

DTO – Data Transfer Object, об'єкт передавання даних.

GPU – Graphics Processing Unit, графічний процесор.

HTTP – HyperText Transfer Protocol, протокол передавання гіпертексту.

JSON – JavaScript Object Notation, текстовий формат обміну даними.

JWT – JSON Web Token, криптографічний токен для автентифікації та авторизації користувачів.

ORM – Object-Relational Mapping, об'єктно-реляційне відображення бази даних.

RAM – Random Access Memory, оперативна пам'ять.

RBAC – Role-Based Access Control, управління доступом на основі ролей.

REST – Representational State Transfer, архітектурний стиль взаємодії клієнта і сервера.

SPA – Single Page Application, односторінковий вебзастосунок.

SQL – Structured Query Language, мова структурованих запитів до баз даних.

TSM – Time-Structured Merge Tree, рушій збереження даних, оптимізований для запису часових рядів.

UI – User Interface, користувацький інтерфейс.

UML – Unified Modeling Language, уніфікована мова моделювання.

UUID – Universally Unique Identifier, універсальний унікальний ідентифікатор.

ВСТУП

Сучасний етап розвитку інформаційних технологій характеризується стрімким зростанням обчислювальних потужностей, ускладненням архітектури застосунків та масштабуванням серверних інфраструктур підприємств. Забезпечення безперебійної роботи таких систем вимагає безперервного контролю їхнього стану, оскільки навіть короточасні апаратні чи програмні збої можуть призвести до зупинки критичних бізнес-процесів, втрати даних та значних фінансових збитків. Водночас частина компаній стикається з труднощами при організації ефективного моніторингу. Використання традиційних локальних систем часто супроводжується високим споживанням системних ресурсів та вимагає послаблення корпоративної мережевої безпеки, зокрема через необхідність відкриття вхідних портів на контрольованих серверах. З іншого боку, перехід на комерційні хмарні платформи змушує бізнес передавати конфіденційну системну телеметрію на сервери сторонніх провайдерів, що створює ризики витоку інформації та зумовлює постійні ліцензійні витрати.

Актуальність теми зумовлена необхідністю впровадження ресурсоефективних засобів контролю ІТ-інфраструктури, що збирають телеметрію без втручання в конфігурації міжмережевих екранів. Вирішення проблем втрати даних при мережевих збоях та усунення залежності від комерційних хмарних платформ досягається шляхом розробки комплексної вебсистеми моніторингу. Запропоноване рішення забезпечує централізоване відстеження апаратних показників та генерацію сповіщень, гарантуючи повну ізоляцію корпоративної інформації.

Метою роботи є розробка вебсистеми моніторингу та сповіщення про стан серверного обладнання для автоматизації збору системних метрик, виявлення інцидентів у реальному часі та підвищення загальної надійності контрольованої ІТ-інфраструктури.

Для досягнення поставленої мети необхідно виконати комплекс інженерних завдань, що охоплює повний цикл розробки програмного

					ІПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						7

забезпечення. Насамперед потрібно провести аналіз предметної області та існуючих рішень для моніторингу, за результатами якого сформувані функціональні та нефункціональні вимоги до програмного комплексу. Наступним кроком є проєктування загальної архітектури за моделлю «Агент – Сервер» із застосуванням push-моделі передачі даних. Після цього необхідно розробити комбіновану підсистему баз даних, що включає проєктування реляційної схеми для збереження бізнес-логіки та оптимізацію структури часової бази даних для накопичення телеметрії. Далі слід реалізувати серверну частину на базі асинхронного фреймворку, розробити легковагий автономний програмний агент збору метрик із механізмом локального буферування та створити клієнтський односторінковий застосунок для візуалізації даних. На завершальному етапі потрібно налаштувати інфраструктуру контейнеризованого розгортання в хмарному середовищі та впровадити конвеєр безперервної інтеграції і доставки.

Об'єктом дослідження є процес збору, збереження та аналізу телеметричних даних стану серверного обладнання в розподілених інформаційних мережах.

Предметом дослідження виступають методи, алгоритми та програмні засоби розробки кросплатформної вебсистеми моніторингу серверної інфраструктури.

Інформаційну базу дослідження склали наукові праці та офіційна технічна документація у галузі побудови високонавантажених систем. Для розв'язання поставлених завдань було застосовано методи системного аналізу для формалізації вимог, підходи реляційного та нереляційного моделювання баз даних для організації поліглотного зберігання, а також методи об'єктно-орієнтованого та компонентного проєктування інформаційних систем.

Практичне значення одержаних результатів полягає у створенні готового до експлуатації програмного комплексу, який дозволяє оптимізувати процес контролю серверного парку. Впровадження системи забезпечує зниження навантаження на мережеву інфраструктуру завдяки використанню ефективного

агента моніторингу з механізмами плавної деградації (graceful degradation) та адаптивної агрегації даних. Використання двох спеціалізованих систем керування базами даних гарантує транзакційну цілісність конфігураційної інформації та високу швидкодію під час обробки мільйонів точок телеметрії. Крім того, автоматизований процес розгортання в середовищі Microsoft Azure з налаштованим локальним агентом виконання завдань забезпечує високий рівень безпеки та відтворюваність процесів оновлення програмного забезпечення.

					ІПЗ.КР.Б – 121 – 26 – ІЗ	Арк.
						9

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ВИМОГ ДО ВЕБСИСТЕМИ МОНІТОРИНГУ ТА СПОВІЩЕННЯ ПРО СТАН СЕРВЕРНОГО ОБЛАДНАННЯ

1.1 Постановка задачі

Розробка надійних засобів контролю ІТ-інфраструктури є критично важливим завданням для сучасних підприємств [30]. Програмний продукт, що розробляється, позиціонується як внутрішній корпоративний інструмент класу Self-Hosted Enterprise Tool, орієнтований на використання у сегменті B2B [28]. Цільовою аудиторією системи є співробітники ІТ-відділів, системні адміністратори та інженери з експлуатації (DevOps). Головною цінністю цієї вебсистеми є запобігання фінансовим та репутаційним втратам компанії, які зазвичай виникають внаслідок простою сервісів та недоступності систем. Ця мета досягається шляхом проактивного моніторингу телеметрії та автоматичної генерації інцидентів ще до того моменту, коли серверне обладнання вийде з ладу або досягне критичного рівня деградації продуктивності.

З точки зору моделі доступу та безпеки, вебсистема розгортається на публічній хмарній інфраструктурі, що забезпечує безперебійний віддалений доступ для адміністративного персоналу з будь-якої точки світу. Проте даний програмний комплекс принципово не є публічним вебсайтом, і в ньому повністю відсутній механізм відкритої реєстрації користувачів, характерний для масового сегмента B2C [28]. Доступ до графічного інтерфейсу, інформаційних панелей та аналітичних графіків жорстко обмежений і здійснюється лише через форму авторизації для попередньо створених адміністративних облікових записів. Такий підхід гарантує високий рівень конфіденційності даних телеметрії та надійний захист від несанкціонованого втручання в конфігурацію інфраструктури підприємства.

Основним технічним завданням роботи є проєктування та реалізація розподіленої вебсистеми за моделлю «Агент - Сервер» [30]. Процес функціонування системи передбачає збір базових апаратних метрик, серед яких рівень використання центрального процесора, оперативної пам'яті, вільного

					ІПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						10

дискового простору, обсяг вхідного і вихідного мережевого трафіку а також опціональний збір телеметрії графічних прискорювачів. Збір інформації має здійснюватися за допомогою легковагих автономних агентів, які встановлюються на цільових серверах. Агенти повинні самостійно генерувати та надсилати дані на центральний сервер управління телеметрією (push-модель), що дозволяє суттєво знизити навантаження на мережу та уникнути необхідності постійного опитування вузлів з боку центрального компонента [17]. Далі серверна частина повинна забезпечити валідацію, обробку та довгострокове збереження цих даних.

Враховуючи специфіку даних безперервного моніторингу, перед системою ставиться жорстка вимога щодо ефективного управління великими масивами інформації. Для вирішення цієї задачі необхідно імплементувати підхід поліглотного зберігання даних (polyglot persistence) [27]. Реляційна модель має використовуватися тільки для зберігання конфігураційної інформації, профілів користувачів, даних про зареєстровані сервери, налаштувань порогових значень та статусів згенерованих інцидентів. Водночас для збереження часових рядів телеметрії ставиться вимога щодо застосування спеціалізованої бази даних, архітектурно здатної обробляти мільйони точок вимірювання з надвисокою швидкістю запису та зчитування [20]. На серверному рівні також необхідно реалізувати алгоритми аналізу метрик у реальному часі для автоматичного відкриття та закриття інцидентів на основі заданих порогових значень із вбудованим механізмом уникнення дублювання записів.

Для забезпечення зручної взаємодії з платформою ставиться завдання розробити клієнтський вебзастосунок за принципом єдиної сторінки (SPA), який взаємодіятиме з серверною частиною через програмний інтерфейс за архітектурним стилем REST [9]. Клієнтський інтерфейс повинен надавати вичерпні інструменти для управління парком серверів та перегляду графіків завантаженості в реальному часі. Завершальним етапом постановки задачі є вимога щодо контейнеризації всіх розроблених компонентів та налаштування

конвеєра безперервної інтеграції і розгортання коду на цільовому сервері для мінімізації людського фактора при оновленні програмного забезпечення [12].

1.2 Аналіз існуючих систем моніторингу

Під час проєктування програмного комплексу було проведено розширене дослідження існуючих індустріальних рішень у сфері моніторингу серверної інфраструктури. Для глибокого порівняльного аналізу обрано системи Zabbix, Prometheus та Datadog, які є визнаними стандартами на ринку корпоративних послуг з управління телеметрією. Дослідження цих продуктів дозволило не лише виявити їхні сильні сторони, але й деталізувати архітектурні обмеження, що стало фундаментальним підґрунтям для формування як функціональних, так і нефункціональних вимог до розроблюваної системи. Аналіз проводився за критеріями ресурсомісткості, безпеки мережевої взаємодії, фінансових витрат на експлуатацію та рівня конфіденційності даних.

Першим програмним аналогом детально розглянуто систему Zabbix, яка є потужним інструментом корпоративного рівня для комплексного збору метрик мережевих сервісів, серверів та хмарних ресурсів [32]. Архітектура цього рішення відзначається високою універсальністю, дозволяючи використовувати як активні (Push), так і пасивні (Pull) перевірки стану обладнання. Проте класичною та найпоширенішою конфігурацією Zabbix залишається саме pull-модель. Суть цієї моделі полягає в тому, що центральний сервер повинен періодично самостійно ініціювати мережеві з'єднання з кожним клієнтським вузлом для збору даних. Це є суттєвим архітектурним недоліком з точки зору інформаційної безпеки та адміністрування, оскільки вимагає примусового відкриття входних мережевих портів (за замовчуванням 10050) та постійного внесення винятків до міжмережевих екранів (брандмауерів) на всіх контрольованих серверах [32]. Крім того, в процесі ретельного дослідження було встановлено, що для специфічної задачі ізольованого моніторингу базових апаратних метрик ця система є надмірно перевантаженою та складною в первинному розгортанні. Традиційне збереження гігантських масивів часових рядів у класичних реляційних базах даних, які Zabbix використовує за

замовчуванням, призводить до стрімкої деградації продуктивності дискової підсистеми при масштабуванні парку контрольованих серверів. Крім того, графічний інтерфейс системи містить значну кількість надлишкового функціоналу, що виходить за межі реальних потреб вузькоспеціалізованого ІТ-відділу, головною метою якого є виключно проактивне запобігання простоям обладнання (рис. 1.1).



Рис. 1.1. Інтерфейс головної панелі управління системи Zabbix

Другим програмним продуктом, що підлягав глибокому аналізу, є екосистема Prometheus. Ця платформа побудована навколо високоефективної спеціалізованої бази даних часових рядів і відзначається потужними можливостями агрегації метрик за допомогою спеціалізованої мови запитів PromQL [22]. Головним архітектурним патерном Prometheus також є застосування моделі збору даних на основі опитування (pull-модель). Як було зазначено раніше, такий підхід успадковує недоліки щодо необхідності відкриття вхідних мережевих портів та постійного внесення змін до правил міжмережевих екранів. Застосування цієї моделі в розроблюваній системі було визнано концептуально неприйнятним, оскільки це прямо суперечить сучасним принципам інформаційної безпеки та концепції мінімізації мережевих вразливостей. Додатковим архітектурним недоліком є відсутність повноцінного вбудованого інтерфейсу користувача, через що візуалізація зібраних метрик неминуче вимагає розгортання та інтеграції додаткового програмного забезпечення, зокрема платформи Grafana (рис. 1.2).

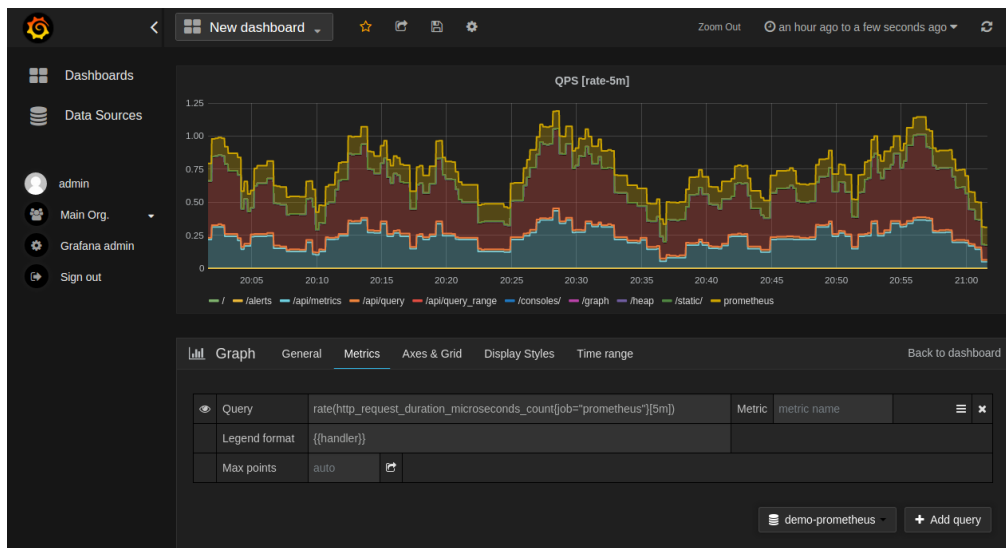


Рис. 1.2. Візуалізація системних метрик у середовищі Grafana

Третім індустріальним рішенням для аналізу виступає комплексна хмарна платформа Datadog. Даний комерційний продукт забезпечує глибокий та всебічний моніторинг інфраструктури завдяки використанню оптимізованих агентів збору даних, що працюють у фоновому режимі [4]. Беззаперечною перевагою платформи є високий рівень автоматизації виявлення аномалій, наявність готових інтеграцій із сотнями зовнішніх хмарних сервісів та інтуїтивно зрозуміла інформаційно-аналітична панель (рис. 1.3).

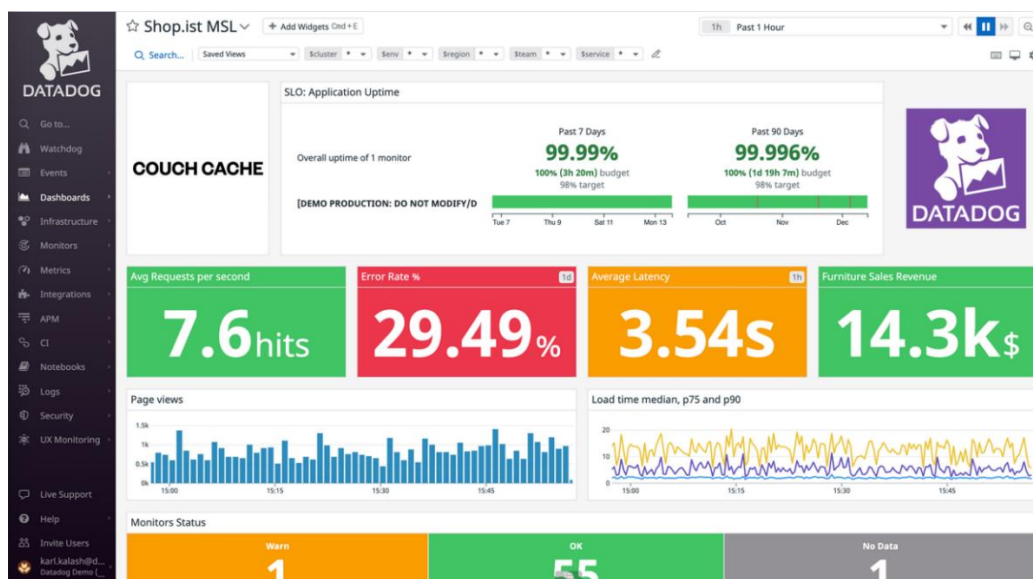


Рис. 1.3. Панель моніторингу серверної інфраструктури в системі Datadog

Незважаючи на потужний функціонал, головним критичним недоліком Datadog у контексті сформульованої задачі є пропрієтарна SaaS-модель розповсюдження [4]. Впровадження цієї платформи неминуче призводить до

того, що конфіденційні дані про стан та архітектуру внутрішньої корпоративної інфраструктури передаються на обробку та тривале зберігання сторонньому постачальнику послуг. Такий підхід грубо порушує суворі вимоги ізоляції та інформаційної безпеки, які безальтернативно висуваються до програмного забезпечення класу Self-Hosted Enterprise Tool. Окрім знакових безпекових ризиків, комерційне ліцензування продукту, яке розраховується на основі кількості підключених хостів, створює значне та постійне фінансове навантаження на бюджет підприємства.

На основі проведеного дослідження недоліків і переваг існуючих систем було обґрунтовано доцільність розробки власного програмного забезпечення. Для наочної оцінки та підсумку відповідності розглянутих аналогів та розроблюваної системи ключовим нефункціональним вимогам було сформовано порівняльну таблицю (табл. 1.1).

Таблиця 1.1

Порівняльний аналіз систем моніторингу серверної інфраструктури за
ключовими критеріями

№	Характеристика (Критерій оцінки)	Zabbix	Prometheus	Datadog	Розроблювана система
1	Push-модель збору даних (без відкриття портів)	+/-	-	+	+
2	Легковагість агента та відсутність перевантаженості ПЗ	-	+	+	+
3	Локальне розгортання та повна ізоляція даних (Self-Hosted)	+	+	-	+
4	Використання оптимізованої бази часових рядів (TSDB)	-	+	+	+
5	Відсутність ліцензійних платежів (Безкоштовність)	+	+	-	+

З наведеного порівняння чітко видно, що розроблювана вебсистема повністю задовольняє всім встановленим ключовим вимогам проєкту. На відміну від архітектури Prometheus та класичних інсталяцій Zabbix, які

покладаються на pull-модель, розроблена система забезпечує безкомпромісний рівень мережевої безпеки завдяки використанню суто push-моделі. Це повністю виключає необхідність втручання в налаштування брандмауерів клієнтських серверів. Крім того, власна розробка гарантує вищу продуктивність збереження часових рядів порівняно із Zabbix завдяки інтеграції InfluxDB та не перевантажена надлишковим функціоналом. У порівнянні з Datadog, власна розробка гарантує абсолютну ізоляцію корпоративних даних (модель Self-Hosted) та повністю виключає будь-які ліцензійні витрати, що робить її оптимальним та економічно вигідним інструментом для розв'язання поставленої інженерної задачі.

1.3 Вибір архітектури вебсистеми

Під час проектування програмного комплексу ключовим етапом став вибір оптимальної архітектурної моделі, яка б забезпечувала високу надійність, масштабованість та дотримання жорстких корпоративних стандартів безпеки інформації. Традиційні монолітні архітектури, у яких підсистеми збору даних, їх обробки та візуалізації об'єднані в єдиний тісно пов'язаний програмний блок, були відкинуті на початковому етапі аналізу. Основною причиною відмови від монолітного підходу стала неможливість гнучкого горизонтального масштабування окремих вузлів та високий ризик каскадної відмови всієї системи при пікових навантаженнях на модуль запису телеметрії [17]. На основі проведеного системного аналізу було прийнято рішення про побудову програмного комплексу за розподіленою багаторівневою моделлю «Агент – Сервер» із застосуванням мікросервісного підходу для розгортання центральної серверної частини [17].

Мережева взаємодія між віддаленими компонентами системи концептуально базується на застосуванні моделі ініціалізації з'єднання з боку клієнта (push-моделі). Відповідно до цього підходу, центральний сервер управління не здійснює періодичного синхронного опитування підключених вузлів. Натомість автономний програмний агент, що функціонує на кожному цільовому сервері, самостійно ініціює з'єднання та надсилає зібрані пакети

телеметрії на центральний сервер. Такий архітектурний вибір є критично важливим для дотримання принципів ізоляції корпоративного контуру безпеки. Він повністю усуває необхідність відкриття входних портів на контрольованих серверах, дозволяючи агентам працювати через стандартні вихідні з'єднання, які зазвичай не блокуються корпоративними міжмережевими екранами (брандмауерами) [30].

Архітектура підсистеми збору даних представлена легковагим автономним агентом, що функціонує як системний фоновий процес (демон). Його головним завданням є безперервний збір базових апаратних метрик. Для забезпечення стійкості до мережових збоїв та відмовостійкості, в архітектуру агента закладено патерн плавної деградації (graceful degradation) з використанням механізму локального буферування. У випадку тимчасової недоступності центрального сервера агент не відкидає зібрані точки даних, а серіалізує їх та надійно зберігає у локальній базі даних SQLite. Після відновлення мережевого зв'язку накопичений масив даних надсилається пакетним запитом (batch processing). Ідентифікація та авторизація агента на сервері здійснюється безстаново (stateless) за допомогою унікального криптографічного ключа (API Key), що виключає можливість підміни джерела телеметрії (рис. 1.4).

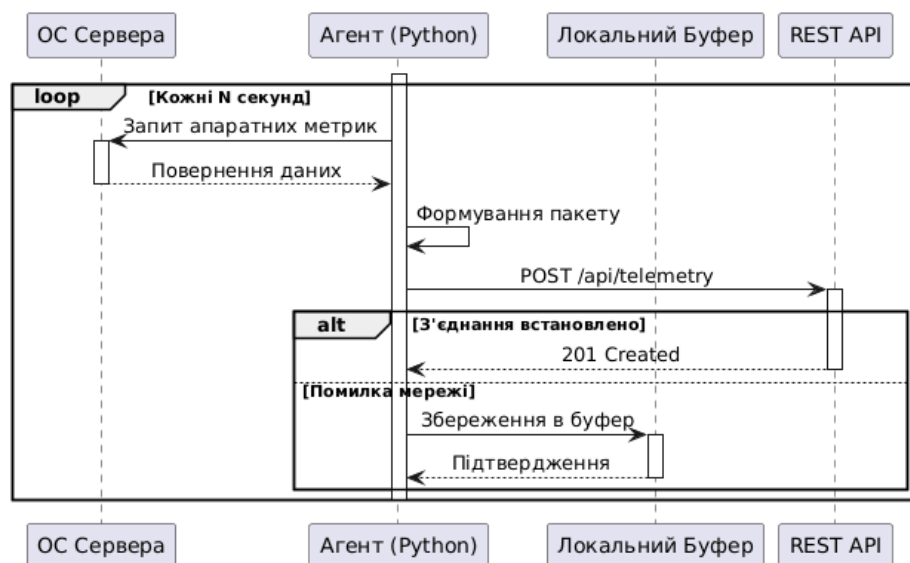


Рис. 1.4. Діаграма послідовності роботи автономного агента збору телеметрії

Центральна серверна частина системи (бекенд) спроектована з урахуванням жорстких вимог до обробки високого рівня конкурентних мережеских запитів. Ядром обробки даних виступає прикладний програмний інтерфейс (REST API), побудований на базі асинхронної моделі вводу-виводу. Замість класичного блокуючого (синхронного) підходу, серверна архітектура використовує неблокуючий цикл подій (event loop) та асинхронні пули з'єднань з базами даних. Це архітектурне рішення дозволяє одному процесорному потоку ефективно обробляти тисячі одночасних підключень від агентів, перемикаючи контекст виконання під час очікування відповіді від бази даних.

Внутрішня архітектура серверної частини побудована за фундаментальним принципом чіткого розділення відповідальності (Separation of Concerns). Систему логічно розділено на три незалежні шари: шар маршрутизації (Routers), який приймає HTTP-запити та проводить первинну структурну та типову валідацію корисного навантаження; шар бізнес-логіки (Services), де реалізовано алгоритми перевірки метрик на перевищення встановлених порогових значень; та шар доступу до даних (Repositories), що абстрагує взаємодію з конкретними сховищами інформації за допомогою патерну Data Access Object. Загальна компонентна схема внутрішньої організації серверної частини наведена на рисунку (рис. 1.5).

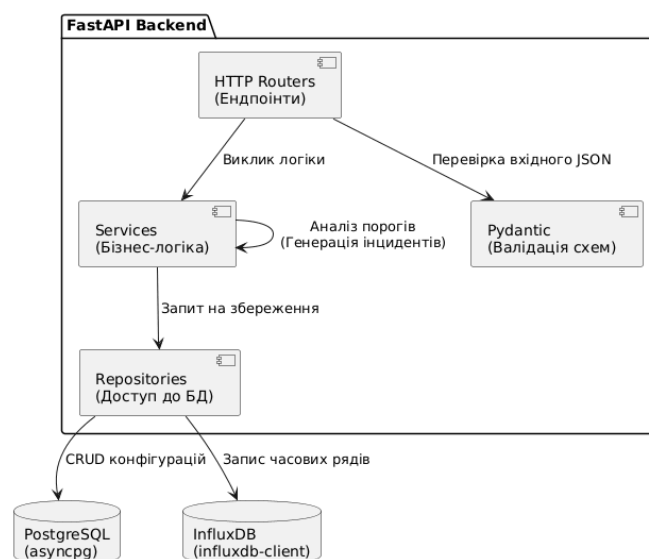


Рис. 1.5. Компонентна діаграма архітектури програмного інтерфейсу (REST API)

Особлива увага при проектуванні архітектури була приділена організації підсистеми зберігання даних. Враховуючи специфіку безперервного моніторингу телеметрії, використання єдиної реляційної бази даних для всіх типів інформації було визнано архітектурним антипатерном. Традиційні деревоподібні індекси (B-tree) та механізми багатоваріантного керування конкурентним доступом (MVCC), які лежать в основі класичних реляційних СУБД, зазнають критичної деградації продуктивності при необхідності безперервного запису мільйонів точок метрик (append-only навантаження). Для розв'язання цієї проблеми в архітектуру системи було імплементовано парадигму поліглотного зберігання даних (polyglot persistence) [27].

Відповідно до цієї парадигми, інформаційна модель поділяється на дві логічні частини. Перша частина – це бізнес-дані, які вимагають суворого дотримання принципів ACID (атомарність, узгодженість, ізолюваність, довговічність). До цієї категорії належать облікові записи користувачів, рольові моделі доступу, реєстр підключених серверів, конфігурації порогових значень та статус-модель інцидентів. Для цих даних спроектовано класичну реляційну схему. Друга частина – це безпосередньо часові ряди телеметрії, які характеризуються величезним обсягом, впорядкованістю у часі та відсутністю необхідності оновлення існуючих записів. Для них в архітектуру закладено використання спеціалізованої бази даних часових рядів (Time-Series Database), яка використовує оптимізовані структури для стиснення та швидкого агрегування потокових даних [13].

Алгоритм обробки телеметрії на сервері включає реактивну підсистему управління інцидентами. При надходженні нового пакету даних шар бізнес-логіки, після асинхронного завантаження конфігурацій, порівнює отримані значення з поточними конфігураціями порогових значень. У разі перевищення критичного порогу генерується подія відкриття інциденту. Для уникнення лавинного дублювання записів, яке часто виникає в системах моніторингу, реалізовано механізм перевірки стану (state tracking): система перевіряє наявність вже відкритого інциденту для конкретного вузла та конкретної

метрики. При поверненні показників у межі норми архітектура передбачає автоматичну зміну статусу інциденту на вирішений (auto-resolution) із фіксацією часової мітки завершення.

Архітектура клієнтської частини спроектована за патерном односторінкового застосунку (SPA). На відміну від класичної архітектури рендерингу на стороні сервера (SSR), де кожна дія користувача призводить до перезавантаження всієї HTML-сторінки, SPA-застосунок завантажується в браузер користувача тільки один раз [25]. Надалі інтерфейс взаємодіє з сервером за допомогою асинхронних HTTP-запитів, обмінюючись легкими JSON-структурами. Такий підхід радикально знижує обсяг мережевого трафіку та забезпечує високу плавність роботи інформаційних панелей при оновленні графіків у реальному часі. У площині UI-архітектури також було впроваджено концепцію делікатної валідації: замість традиційних блокуючих модальних вікон або агресивних червоних панелей з помилками, система використовує вбудовані підказки (inline hints), що з'являються безпосередньо поруч із невалідними полями, мінімізуючи когнітивне навантаження на оператора.

Завершальним елементом архітектури є підсистема маршрутизації та розгортання. Для ізоляції внутрішніх сервісів від прямого доступу з глобальної мережі застосовано архітектурний патерн «Зворотний проксі» (Reverse Proxy) [18]. Проксі-сервер виступає єдиною точкою входу, розпізнаючи типи запитів: звернення до прикладного інтерфейсу маршрутизуються до бекенду, а запити до візуального інтерфейсу – до статичних файлів SPA. Забезпечення кросплатформності реалізовано через контейнеризацію всіх компонентів. Контейнери об'єднуються у закриту віртуальну мережу, що гарантує відсутність доступу до баз даних ззовні. Логічна структурна схема розгортання та мережевої взаємодії компонентів наведена на рисунку (рис. 1.6).

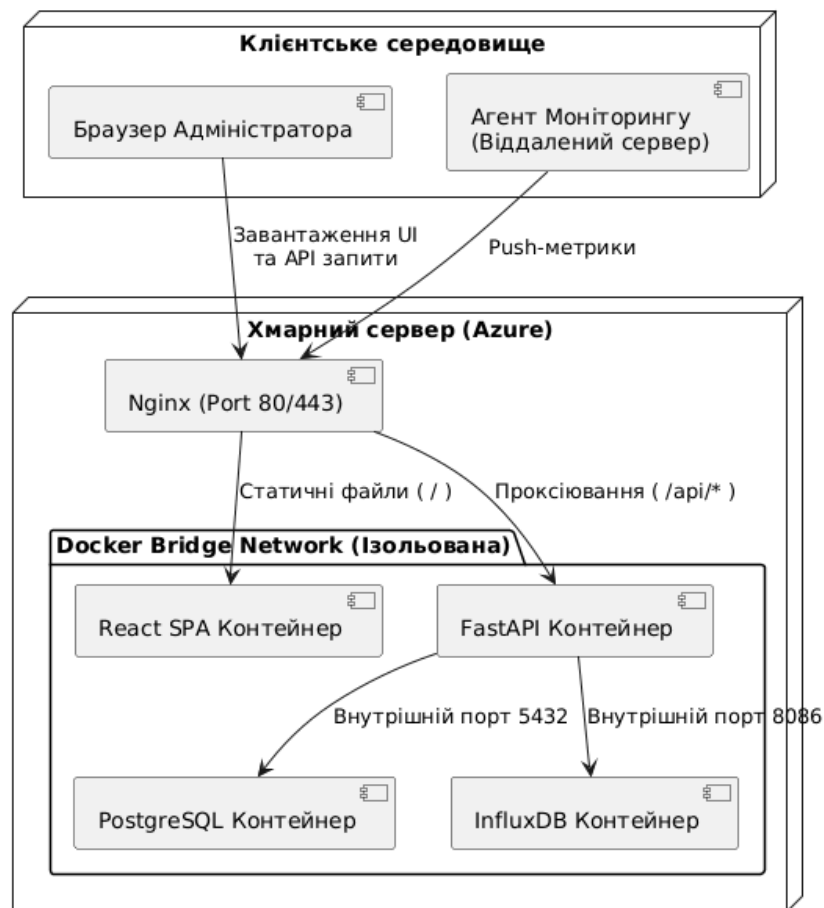


Рис. 1.6. Діаграма розгортання та мережевої взаємодії компонентів вебсистеми

1.4 Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Для практичної реалізації спроектованої архітектури програмного комплексу було проведено детальний аналіз сучасних технологічних стеків та інструментальних засобів. Вибір мов програмування, фреймворків та систем керування базами даних здійснювався з урахуванням вимог щодо високої продуктивності, здатності обробляти велику кількість конкурентних з'єднань, а також можливості забезпечення надійного контейнеризованого розгортання.

Базовою мовою програмування для розробки як автономного агента збору телеметрії, так і центрального прикладного програмного інтерфейсу (REST API) було обрано Python [24]. Цей вибір обґрунтовується наявністю широкого спектра вбудованих бібліотек для низькорівневої взаємодії з операційною системою (зокрема, кросплатформної бібліотеки psutil [23]), що дозволяє агенту ефективно

збирати апаратні метрики незалежно від типу цільової операційної системи. Для розробки серверної частини було прийнято рішення відмовитися від класичних синхронних фреймворків, оскільки вони використовують блокуючу модель обробки запитів, яка є неефективною для систем моніторингу реального часу. Натомість було інтегровано високопродуктивний асинхронний фреймворк FastAPI [8]. Його використання дозволяє нативно застосовувати можливості бібліотеки asuncio, забезпечуючи паралельну обробку тисяч вхідних з'єднань від агентів. Додатковою перевагою FastAPI є тісна інтеграція з інструментом Pydantic, що гарантує сувору типізацію та автоматичну валідацію вхідних JSON-пакетів із метриками безпосередньо на етапі їх прийняття маршрутизатором.

Для створення клієнтського вебзастосунку було обрано екосистему мови JavaScript, зокрема бібліотеку React версії 19 [25]. На відміну від фреймворків з жорсткою структурою, React надає необхідну гнучкість у побудові компонентної архітектури та використовує механізм віртуального об'єктного дерева документів (Virtual DOM) для мінімізації кількості операцій рендерингу інтерфейсу під час потокового оновлення графіків. Для збірки фронтенд-застосунку застосовано сучасний інструмент Vite [31], який забезпечує миттєвий холодний запуск середовища розробки завдяки використанню нативних модулів ECMAScript. У контексті проектування користувацького інтерфейсу та механізмів обробки форм на стороні клієнта було використано бібліотеку React Hook Form. Це дозволило реалізувати парадигму делікатної валідації вводу, за якої система уникає використання традиційних агресивних червоних панелей з попередженнями про помилки. Натомість розроблено механізм виведення тонких вбудованих підказок (inline hints) безпосередньо під відповідними полями, що значно знижує когнітивне навантаження на адміністратора під час налаштування критичних порогових значень.

Реалізація архітектурного підходу поліглотного зберігання даних вимагала вибору двох різних за своєю структурою систем керування базами даних. Для збереження бізнес-логіки, облікових записів та статусної моделі інцидентів було обрано об'єктно-реляційну систему PostgreSQL [21]. Ця СУБД є індустріальним

					ІПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						22

стандартом транзакційної надійності, що повністю підтримує принципи ACID та дозволяє гнучко налаштовувати зовнішні ключі для забезпечення цілісності зв'язків між зареєстрованими серверами та їхніми метриками. Взаємодія бекенду з базою даних здійснюється через асинхронний драйвер `asynsrg`, що виключає блокування основного потоку виконання. Водночас для збереження безперервного потоку телеметрії було обрано спеціалізовану базу даних часових рядів InfluxDB [13]. На відміну від реляційних сховищ, InfluxDB використовує унікальний рушій зберігання TSM (Time-Structured Merge Tree). Цей інструмент оптимізовано для екстремально швидкого додавання нових даних (append-only) та забезпечення високого рівня компресії числових масивів.

Для забезпечення відтворюваності середовища, спрощення процесу розгортання та ізоляції компонентів було обрано платформу контейнеризації Docker та інструмент оркестрації Docker Compose [6]. Завдяки цим інструментам усі компоненти системи упаковуються в незалежні контейнери з власною файловою системою та залежностями. Як зворотний проксі-сервер (Reverse Proxy) обрано вебсервер Nginx [18]. Його використання є критично важливим для забезпечення мережевої безпеки, оскільки він приймає весь зовнішній трафік на стандартних портах і маршрутизує його до відповідних контейнерів усередині закритої віртуальної мережі Docker.

З метою мінімізації людського фактора при оновленні програмного забезпечення було впроваджено автоматизацію процесів безперервної інтеграції та розгортання (CI/CD) за допомогою платформи GitLab [12]. Інструментарій GitLab CI дозволив налаштувати автоматизований конвеєр, за якого агент GitLab Runner ініціює процес оновлення безпосередньо на хмарному сервері при кожному злитті нового коду. Схема процесу автоматизованого розгортання із застосуванням обраних інструментів наведена на рисунку (рис. 1.7).

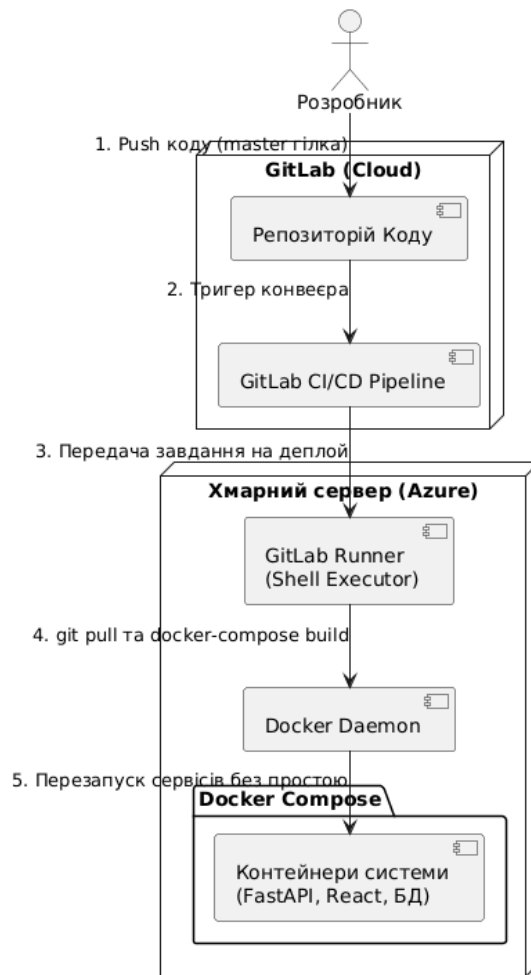


Рис. 1.7. Схема процесу автоматизованого розгортання компонентів за допомогою GitLab CI/CD

Ефективне та безперебійне функціонування розробленої системи вимагає суворого дотримання вимог до апаратного забезпечення. Оскільки архітектура розділена на серверну та клієнтську частини з різним рівнем обчислювального навантаження, вимоги формуються диференційовано для кожного типу вузла. Для серверів корпоративної інфраструктури, які підлягають моніторингу і на яких встановлюється програмний агент, апаратні вимоги є мінімальними завдяки малій ресурсоемності розробленого скрипта. Водночас вимоги до центрального хмарного вузла Microsoft Azure, на якому розгортається повний стек платформи (контейнери баз даних, бекенду та реверс-проксі), є значно вищими через необхідність безперервної агрегації великих масивів інформації рушієм бази InfluxDB. Зведені вимоги до апаратного забезпечення для обох підсистем наведено у таблиці (табл. 1.2).

Таблиця 1.2.

Мінімальні та рекомендовані вимоги до апаратного забезпечення системи

Компонент системи	Центральний процесор (CPU)	Оперативна пам'ять (RAM)	Дисковий простір (SSD)	Мережеве з'єднання
Агент моніторингу (цільовий клієнтський сервер)	Мінімум 1 ядро (від 1 ГГц)	Від 128 МБ	Від 50 МБ	Вихідний доступ до інтернету
Центральний сервер (хмарне середовище Azure)	Рекомендовано 4 vCPU	Рекомендовано від 8 ГБ	Від 100 ГБ (для БД)	Статична IP-адреса, порти 80/443

Наведені апаратні характеристики центрального сервера розраховані з урахуванням необхідності кешування даних у пам'яті для бази InfluxDB та обробки пікових навантажень вебсервером Nginx під час одночасного доступу адміністраторів до графічного інтерфейсу.

Висновки до першого розділу

У першому розділі було проведено комплексний аналіз предметної області та формалізовано постановку задачі на розробку вебсистеми моніторингу стану серверного обладнання. У результаті дослідження існуючих індустріальних аналогів було доведено об'єктивну необхідність створення власного програмного забезпечення. Запропоноване рішення концептуально позбавлене надлишкового функціоналу, усуває ризики передачі конфіденційних даних стороннім постачальникам послуг та не створює додаткового фінансового навантаження на підприємство у вигляді витрат на комерційне ліцензування. Для досягнення поставлених цілей було спроектовано багаторівневу розподілену архітектуру за моделлю «Агент – Сервер», що використовує безпечну push-модель передачі телеметрії, яка гарантує збереження цілісності мережевого периметра корпоративної інфраструктури та мінімізує ризики зовнішнього втручання.

Фундаментальним архітектурним рішенням стала імплементація підходу поліглотного зберігання даних. Це дозволило ефективно розподілити навантаження на підсистему зберігання шляхом використання транзакційної

реляційної системи PostgreSQL для ізоляції конфігураційної інформації та спеціалізованої бази InfluxDB для високошвидкісного запису часових рядів телеметрії. На основі розробленої архітектури було здійснено обґрунтований вибір інструментальних засобів, що включає мови програмування Python та JavaScript із застосуванням сучасних фреймворків FastAPI та React. Такий технологічний стек забезпечує високу пропускну здатність при обробці асинхронних з'єднань та оптимізовану взаємодію з користувачем через односторінковий застосунок. Додатково було визначено вимоги до апаратного забезпечення, а також обрано технології контейнеризації Docker та автоматизації конвеєра GitLab CI/CD для забезпечення надійності розгортання. Отримані результати формують вичерпне теоретичне та технічне підґрунтя для переходу до етапу безпосереднього проектування структур баз даних та програмних компонентів системи.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ВЕБСИСТЕМИ МОНІТОРИНГУ ТА СПОВІЩЕННЯ ПРО СТАН СЕРВЕРНОГО ОБЛАДНАННЯ

2.1 Визначення варіантів використання та об'єктно-орієнтованої структури системи

Процес проєктування програмного комплексу моніторингу розпочато зі специфікації вимог до кінцевого продукту. Визначення функціональних та нефункціональних вимог є критично важливим етапом, що дозволяє формалізувати очікування від системи та сформувати базис для подальшої архітектурної декомпозиції [28]. Функціональні вимоги описують безпосередню поведінку системи, набір операцій, які вона повинна виконувати, та визначають можливості користувачів різних типів.

Серед ключових функціональних вимог виділено необхідність забезпечення автентифікації користувачів за допомогою електронної пошти та пароля з генерацією криптографічного JWT-токена. Система повинна надавати адміністраторам вичерпний інструментарій для управління парком серверів, що включає додавання, редагування та видалення записів із відповідною автоматичною генерацією унікальних API-ключів для кожного вузла. Також вимагається наявність функціоналу для встановлення критичних та попереджувальних порогових значень для кожної апаратної метрики. У підсистемі збору даних програмний агент повинен забезпечувати безперервний збір показників використання центрального процесора, оперативної пам'яті, вільного дискового простору, мережевого трафіку а також навантаження на графічний прискорювач з подальшим їх надсиланням на центральний сервер через REST API. Для забезпечення надійності вимагається реалізація механізму плавної деградації (graceful degradation), за якої агент локально буферизує зібрані дані у разі тимчасової втрати мережевого з'єднання. Клієнтська частина системи повинна відображати аналітичні дашборди у режимі реального часу, візуалізуючи часові ряди завантаженості. Крім того, на серверному рівні має бути реалізована жорстка логіка автоматичної генерації інцидентів при

перевищенні заданих порогів та їх автоматичного закриття після нормалізації стану апаратного забезпечення.

Окрім функціональних можливостей, до системи висувається ряд нефункціональних вимог, які регламентують атрибути якості програмного продукту, такі як продуктивність, масштабованість, безпека та надійність. Особлива увага приділяється часовим характеристикам реакції системи, оскільки програмний комплекс призначений для роботи з телеметрією. Визначено, що тривалість обробки запиту на збереження пакету даних від агента не повинна перевищувати 200 мілісекунд для уникнення блокування мережових з'єднань. Час повного завантаження клієнтського застосунку та побудови графіків обмежується 1.5 секундами за нормальних умов мережі. Вимоги до пропускної здатності передбачають здатність серверної архітектури витримувати одночасне підключення щонайменше 1000 системних агентів без відчутної деградації продуктивності. У контексті безпеки передачі даних закладено вимогу щодо використання лише зашифрованого протоколу та повної ізоляції внутрішніх контейнерів на рівні віртуальної мережі. Вимоги до відмовостійкості передбачають, що при відмові клієнтського веб-інтерфейсу, агент та серверна підсистема збору метрик повинні продовжувати своє безперебійне функціонування.

Специфікація варіантів використання здійснюється за допомогою виділення акторів – сутностей, що взаємодіють із програмним продуктом. В інформаційній системі моніторингу виділено трьох головних акторів, які принципово відрізняються за природою своєї взаємодії з програмним комплексом. Першим актором є «Системний адміністратор» – оператор, що має повні привілеї доступу в системі, ініціює створення облікових записів, налаштовує параметри моніторингу та управляє життєвим циклом підключених серверів. Другим актором визначено роль «Спостерігач» (Viewer), якій надано права тільки на перегляд аналітичних панелей та статусів інцидентів без можливості внесення конфігураційних змін. Третім специфічним актором виступає «Програмний агент» – автономний фоновий процес, що виконується на

					ІПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						28

контрольованому обладнанні. Агент не має доступу до візуального інтерфейсу і взаємодіє з ядром системи через програмний інтерфейс REST API, проходячи автентифікацію за допомогою унікального статичного ключа. Графічне представлення визначених варіантів використання наведено на діаграмі (рис. 2.1).

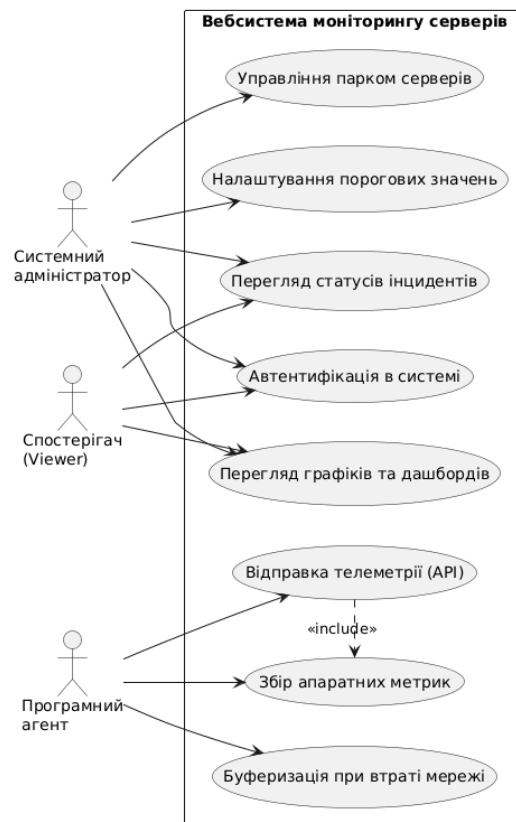


Рис. 2.1. Діаграма варіантів використання (Use Case) вебсистеми моніторингу

Для забезпечення гнучкості розробки та подальшого супроводу було проведено об'єктно-орієнтовану декомпозицію предметної області [11]. Структура серверної частини системи (бекенду) базується на використанні патерну Object-Relational Mapping (ORM) за допомогою бібліотеки SQLAlchemy. Основними класами-сутностями реляційної моделі виступають User, Server, Threshold та Incident. Клас Server інкапсулює атрибути контрольованого обладнання (ідентифікатор, IP-адресу, тип операційної системи) та реалізує методи для генерації криптографічних API-ключів і оновлення часової мітки останньої активності (update_last_seen). Клас Threshold містить параметри критичних і попереджувальних меж для конкретних метрик та пов'язаний із

класом Server композиційним зв'язком. Згідно з правилами каскадного видалення, ліквідація об'єкта Server автоматично ініціює видалення всіх пов'язаних об'єктів порогів та інцидентів, що запобігає виникненню аномалій бази даних.

Специфічною особливістю розробленої об'єктної структури є імплементація об'єктів передачі даних (Data Transfer Objects, DTO) для роботи з часовими рядами телеметрії [10]. Оскільки показники завантаженості зберігаються в нереляційній СУБД InfluxDB, телеметрія не має прямого відображення у вигляді ORM-класу бази даних PostgreSQL. Натомість вона репрезентується набором класів валідації на базі бібліотеки Pydantic: TelemetryCreate (для вхідних пакетів від агента з суворою типізацією меж значень, таких як відсоток завантаження процесора у діапазоні від 0 до 100) та TelemetryHistoryPoint (для повернення агрегованих точок на графіки фронтенду). Узагальнену структуру основних об'єктів системи, їх атрибутів та базових методів наведено на діаграмі класів (рис. 2.2).

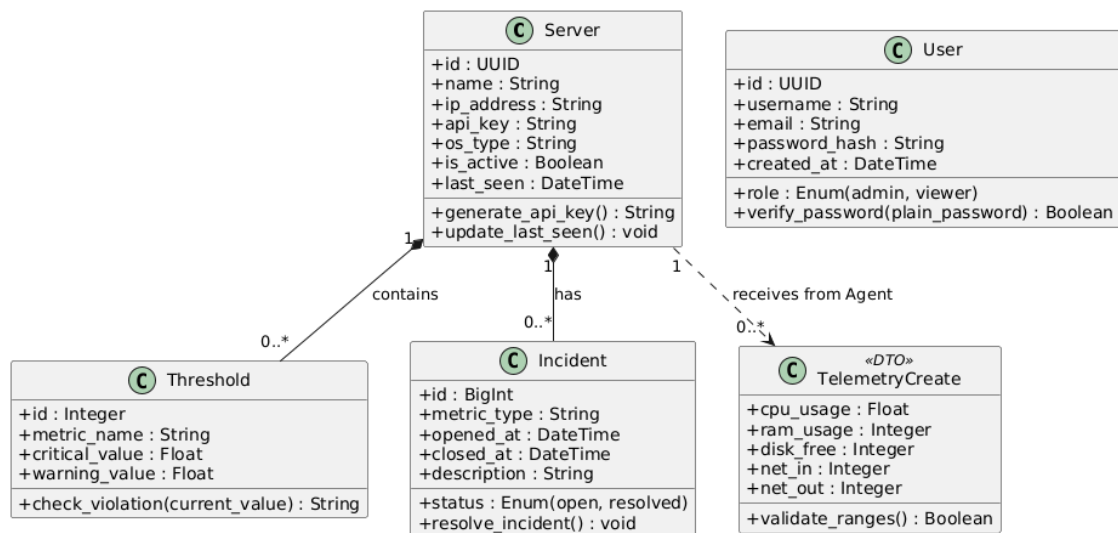


Рис. 2.2. Діаграма класів основних сутностей вебсистеми моніторингу

2.2 Розробка бази даних вебсистеми

Проектування підсистеми зберігання інформації є одним із найвідповідальніших етапів розробки програмного комплексу, оскільки від обраної фізичної моделі бази даних [7] безпосередньо залежить продуктивність, масштабованість та надійність усієї системи. Враховуючи специфіку завдання

безперервного моніторингу, що вимагає одночасної підтримки транзакційної цілісності для конфігураційних даних та екстремально високої швидкості запису для телеметрії, було застосовано архітектурний підхід поліглотного зберігання (polyglot persistence) [27]. Відповідно до цього підходу, фізична модель бази даних розділена на дві ізольовані підсистеми: реляційну схему під управлінням об'єктно-реляційної СУБД PostgreSQL та нереляційне сховище часових рядів на базі InfluxDB [14].

Реляційна частина бази даних спроектована у третій нормальній формі (3NF) для уникнення надмірності та усунення аномалій оновлення. Використання класичних деревоподібних індексів (B-tree) [3], що лежать в основі PostgreSQL, забезпечує високу швидкість виконання запитів із жорсткими умовами об'єднання (JOIN) та підтримує вимоги стандарту ACID [5]. Реляційна модель складається з чотирьох основних фізичних таблиць. Фундаментальними сутностями виступають реєстр облікових записів адміністраторів та реєстр підключених серверів. Для забезпечення високого рівня безпеки та унеможливлення підбору ідентифікаторів методом прямого перебору, як первинні ключі (Primary Keys) для цих таблиць було обрано використання універсальних унікальних ідентифікаторів (UUID) замість класичних цілочисельних лічильників з автоінкрементом.

Логіку зв'язків між сутностями реалізовано за допомогою механізму зовнішніх ключів (Foreign Keys). Таблиці порогових значень та інцидентів підпорядковуються таблиці серверів із відношенням «один до багатьох» (1:N). Для підтримки цілісності бази даних на рівні фізичної схеми застосовано обмеження каскадного видалення (ON DELETE CASCADE): при отриманні запису про сервер із реєстру, фізичний рушій бази даних автоматично знищує всі пов'язані з ним налаштування порогів та історичні записи про інциденти. Візуалізацію фізичної схеми реляційної частини бази даних наведено на ER-діаграмі (рис. 2.3).

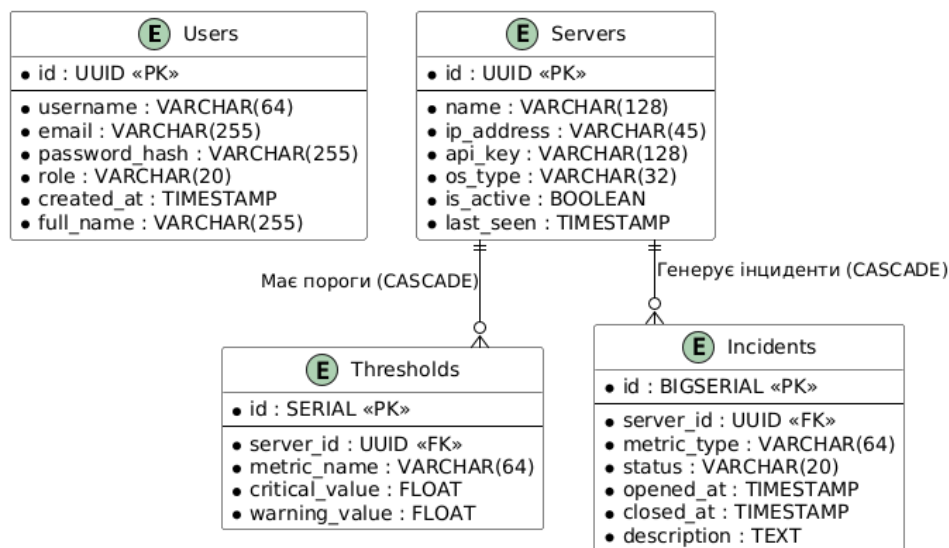


Рис. 2.3. Фізична ER-модель реляційної бази даних PostgreSQL

Для формалізації структури зберігання бізнес-даних було сформовано детальний опис кожної спроектованої таблиці. Фізична таблиця користувачів призначена для збереження облікових даних адміністративного персоналу та їхніх рольових моделей доступу. Структура таблиці наведена у відповідній таблиці (табл. 2.1).

Таблиця 2.1

Структура фізичної таблиці Users

Назва атрибута	Тип даних (PostgreSQL)	Обмеження та ключі	Опис призначення поля
id	UUID	Primary Key, Not Null	Унікальний ідентифікатор користувача
username	VARCHAR(64)	Unique, Not Null	Логін (ім'я) користувача в системі
email	VARCHAR(255)	Unique, Not Null	Електронна пошта для автентифікації
password_hash	VARCHAR(255)	Not Null	Хеш-сума пароля (алгоритм bcrypt)
role	VARCHAR(20)	Not Null	Роль користувача (admin або viewer)
created_at	TIMESTAMP	Default NOW()	Часова мітка створення облікового запису
full_name	VARCHAR(255)	Nullable	Додаткове поле з інформацією про користувача (необов'язкове)

Реєстрація контрольованого апаратного забезпечення здійснюється у таблиці серверів, яка також зберігає згенеровані криптографічні ключі для авторизації автономних агентів. Структура цієї таблиці відображена нижче (табл. 2.2).

Таблиця 2.2

Структура фізичної таблиці Servers

Назва атрибута	Тип даних (PostgreSQL)	Обмеження та ключі	Опис призначення поля
id	UUID	Primary Key, Not Null	Унікальний ідентифікатор сервера
name	VARCHAR(128)	Not Null	Зрозуміле для адміністратора ім'я вузла
ip_address	VARCHAR(45)	Not Null	Мережева адреса (IPv4 або IPv6)
api_key	VARCHAR(128)	Unique, Not Null	Згенерований ключ для доступу до REST API
os_type	VARCHAR(32)	Not Null	Тип операційної системи (Linux, Windows)
is_active	BOOLEAN	Default TRUE	Прапорець логічного стану моніторингу
last_seen	TIMESTAMP	Nullable	Час останнього успішного зв'язку з агентом

Для забезпечення проактивного моніторингу спроектовано таблицю порогових значень, яка дозволяє встановлювати індивідуальні межі попереджень для кожної конкретної апаратної метрики окремого сервера. Її фізичну структуру описано у таблиці (табл. 2.3).

Таблиця 2.3

Структура фізичної таблиці Thresholds

Назва атрибута	Тип даних (PostgreSQL)	Обмеження та ключі	Опис призначення поля
id	SERIAL	Primary Key, Not Null	Автоінкрементний ідентифікатор порогу
server_id	UUID	Foreign Key, Not Null	Посилання на сервер (каскадне видалення)
metric_name	VARCHAR(64)	Not Null	Назва метрики (наприклад, cpu_usage)
critical_value	FLOAT	Not Null	Значення для відкриття критичного інциденту

1.	2.	3.	4.
warning_value	FLOAT	Not Null	Значення для попереджувального повідомлення

Система автоматичного реагування реєструє всі виявлені аномалії у таблиці інцидентів. Враховуючи потенційно велику кількість історичних записів про збої, для первинного ключа цієї таблиці обрано розширений цілочисельний тип великого обсягу. Фізична структура таблиці наведена нижче (табл. 2.4).

Таблиця 2.4

Структура фізичної таблиці Incidents

Назва атрибута	Тип даних (PostgreSQL)	Обмеження та ключі	Опис призначення поля
id	BIGSERIAL	Primary Key, Not Null	Великий автоінкрементний ідентифікатор
server_id	UUID	Foreign Key, Not Null	Посилання на проблемний сервер
metric_type	VARCHAR(64)	Not Null	Тип метрики, що спричинила інцидент
status	VARCHAR(20)	Not Null	Поточний статус (open, resolved)
opened_at	TIMESTAMP	Default NOW()	Часова мітка фіксації перевищення порогу
closed_at	TIMESTAMP	Nullable	Часова мітка нормалізації метрики
description	TEXT	Nullable	Детальний опис зафіксованої аномалії

Другим компонентом поліготної моделі виступає база даних часових рядів InfluxDB, архітектура якої кардинально відрізняється від реляційних підходів [13]. Ця СУБД використовує власну ієрархію зберігання. На найвищому рівні інформація організовується у глобальні логічні контейнери баз даних (Buckets). У розробленій системі було створено єдиний такий контейнер під назвою server_telemetry. Базовою одиницею інформації всередині контейнера є точка даних (Data Point), яка фізично записується на диск за допомогою

спеціалізованого рушія TSM (Time-Structured Merge Tree). Цей рушій розроблений спеціально для стиснення потокових даних та обробки навантажень типу append-only, що усуває проблему фрагментації дискового простору [20].

Кожна точка телеметрії в InfluxDB має сувору структуру, що складається з чотирьох елементів: назви вимірювання (Measurement, що логічно виступає аналогом таблиці), мітки часу (Timestamp), набору тегів (Tags) та набору полів (Fields). Щоб чітко розмежувати ієрархію збереження, в межах глобального контейнера server_telemetry єдиною назвою вимірювання (таблиці) визначено константу server_metrics. Ключовим етапом проєктування схеми InfluxDB став правильний розподіл атрибутів між тегами та полями. Теги є індексованими метаданими у форматі рядків, які фізичний рушій використовує для швидкого пошуку та фільтрації. До категорії тегів було віднесено ідентифікатор сервера (server_id) та тип операційної системи (os_type). На противагу цьому, поля (Fields) не індексуються базою даних, що дозволяє значно економити оперативну пам'ять та ресурси процесора під час масового запису. Як поля було спроектовано безпосередні числові значення метрик: використання процесора (cpu_usage, тип float), обсяг зайнятої пам'яті (ram_usage, тип int), залишок диска (disk_free, int), мережеві показники (net_in, net_out, int), а також телеметрія графічного прискорювача: завантаження (gpu_load, тип float), температура (gpu_temperature, тип float) та використана відеопам'ять (gpu_memory_used, тип int). Логічну структуру запису однієї точки телеметрії наведено на концептуальній діаграмі (рис. 2.4).

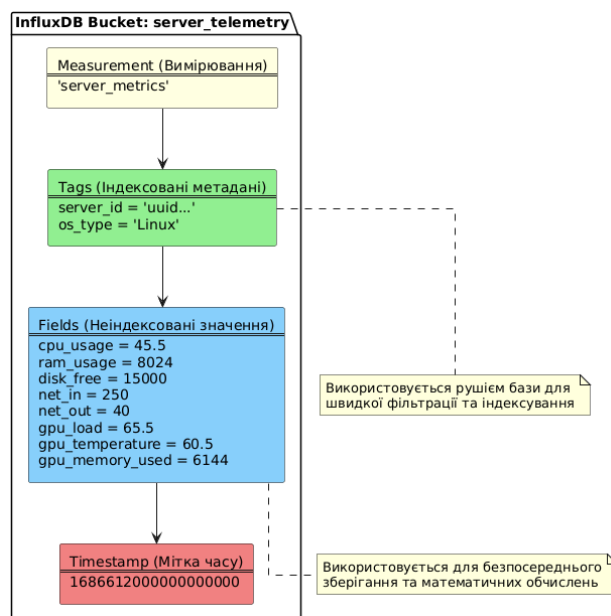


Рис. 2.4. Логічна структура точки даних часового ряду в базі InfluxDB

2.3 Проєктування алгоритмів роботи вебсистеми

Проєктування алгоритмічного забезпечення є ключовим етапом створення надійної розподіленої системи, оскільки воно визначає правила поведінки компонентів у штатних та критичних ситуаціях [30]. Враховуючи мікросервісну архітектуру розроблюваного програмного комплексу, загальний алгоритм роботи було декомпоновано на три ізольовані, але взаємопов'язані процеси: алгоритм безперервного збору апаратних метрик на стороні автономного агента, серверний алгоритм управління телеметрією та життєвим циклом інцидентів, а також алгоритм мережевої взаємодії для агрегації та візуалізації даних на стороні клієнтського застосунку.

Підсистема збору даних функціонує на основі циклічного алгоритму (daemon loop), який ініціюється автоматично під час запуску операційної системи цільового сервера. На початку кожної ітерації програмний агент звертається до ядра операційної системи через системні виклики (із застосуванням інструментарію бібліотеки psutil) для отримання сирих значень утилізації процесора, заповненості оперативної пам'яті, вільного дискового простору, обсягу переданого мережевого трафіку а також опціонально опитує системні утиліти (nvidia-smi) для зчитування метрик GPU [23]. Зібрані показники проходять етап первинної нормалізації, після чого об'єднуються у

стандартизовану структуру формату JSON. До сформованого пакету даних обов'язково додається унікальний криптографічний API-ключ, який слугує єдиним ідентифікатором вузла для його автентифікації.

Особливістю спроектованого алгоритму агента є вбудований механізм плавної деградації (graceful degradation) для забезпечення високої відмовостійкості [17]. Перед спробою надсилання пакету через HTTP POST запит алгоритм ініціює блок обробки виняткових ситуацій. У разі успішного встановлення з'єднання із центральним сервером (отримання коду відповіді 201 Created), алгоритм завершує ітерацію та переходить у режим очікування згідно із заданим інтервалом. Однак, якщо виникає помилка мережевого тайм-ауту (Timeout) або сервер повертає статус внутрішньої помилки (серія 5XX), алгоритм перехоплює виняток і перенаправляє сформований JSON-пакет до локальної бази даних SQLite. На наступній ітерації алгоритм перевіряє наявність даних у буфері і, у разі відновлення зв'язку, здійснює пакетне надсилання (batch processing) накопиченої історії разом із новими показниками. Графічне представлення описаної логіки наведено на блок-схемі алгоритму (рис. 2.5).

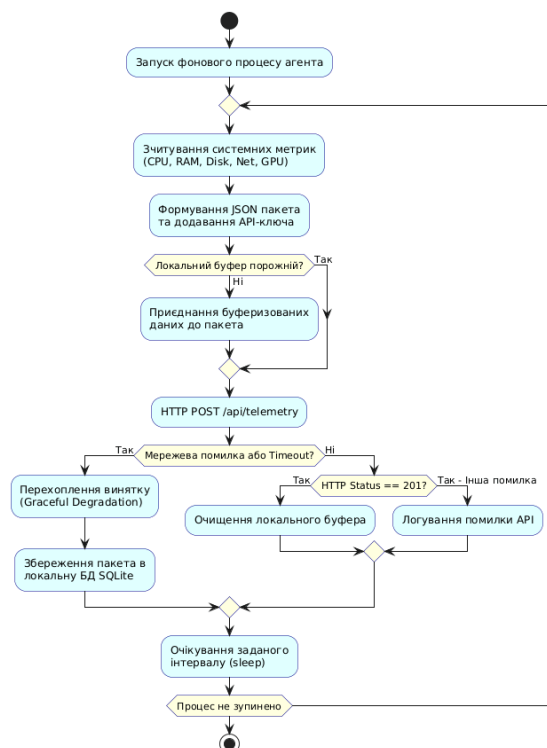


Рис. 2.5. UML діаграма діяльності (Activity Diagram) алгоритму роботи програмного агента

Центральним компонентом бізнес-логіки бекенду є серверний алгоритм обробки телеметрії та управління інцидентами. Процес ініціюється при отриманні вхідного HTTP-запиту від агента. На першому етапі маршрутизатор здійснює асинхронну валідацію наданого API-ключа шляхом запиту до реляційної бази даних PostgreSQL. У разі підтвердження автентичності, корисне навантаження запиту десеріалізується та перевіряється на відповідність структурі об'єкта передачі даних TelemetryCreate. Після успішної типової валідації алгоритм розгалужується на два паралельні потоки виконання. Перший потік відповідає за фізичне збереження незмінених часових рядів телеметрії у базу даних InfluxDB із відповідним маркуванням тегами `server_id` та `os_type` [13].

Другий потік виконання реалізує реактивну підсистему генерації інцидентів. Алгоритм послідовно ітерує кожну отриману метрику та порівнює її поточне значення із заздалегідь встановленими лімітами у таблиці `Thresholds`. Якщо показник (наприклад, завантаженість процесора) перевищує встановлений критичний поріг, система ініціює процедуру генерації інциденту. Для уникнення лавинного дублювання записів, яке є поширеною проблемою систем моніторингу [15], реалізовано механізм перевірки стану (*state tracking*). Перед створенням нового запису в таблиці `Incidents`, алгоритм здійснює пошук вже існуючого відкритого інциденту (де `status = 'open'`) для комбінації конкретного сервера та конкретної метрики. Якщо такий запис існує, створення дубліката блокується.

Водночас реалізовано алгоритм автоматичного розв'язання інцидентів (*auto-resolve*). Якщо поточне значення метрики знаходиться в межах норми, алгоритм перевіряє наявність відкритого інциденту в базі даних. У разі його виявлення, статус інциденту автоматично змінюється на `resolved`, а в поле `closed_at` записується поточна часова мітка. Логіку цього серверного алгоритму візуалізовано на відповідній діаграмі (рис. 2.6).

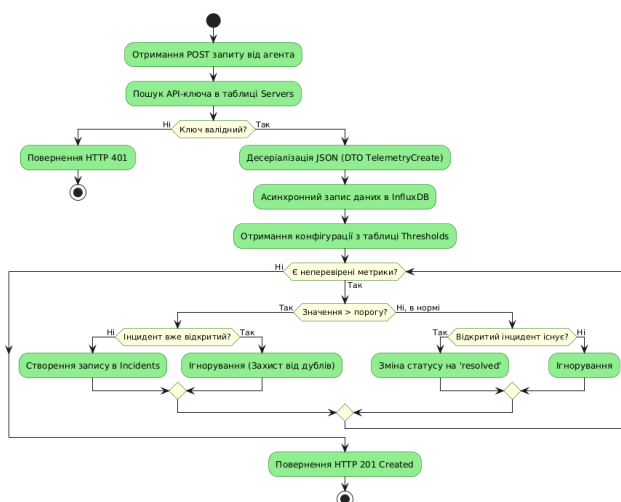


Рис. 2.6. UML діаграма діяльності серверного алгоритму управління інцидентами

Завершальним етапом алгоритмічної взаємодії є процес формування агрегованих даних для клієнтського застосунку. Враховуючи, що база даних часових рядів може накопичувати мільйони точок телеметрії для кожного сервера, пряма передача всіх сирих значень до браузера користувача призвела б до критичного перевантаження мережевого каналу та блокування інтерфейсу. Для вирішення цієї проблеми було розроблено алгоритм проміжної агрегації.

Процес розпочинається, коли клієнтський односторінковий застосунок (SPA), побудований на базі бібліотеки React, відправляє авторизований запит із JWT-токеном на отримання історичних даних за певний період (наприклад, за останні 24 години). Зворотний проксі-сервер Nginx перенаправляє цей запит до мікросервісу FastAPI. Бекенд генерує оптимізований запит до InfluxDB, використовуючи вбудовані функції агрегації рушія (такі як `mean()` для обчислення середнього значення та `max()` для пікових навантажень) із заданим вікном групування [13]. Отриманий стислий набір даних серіалізується у масив об'єктів формату `TelemetryHistoryPoint` та повертається клієнту у вигляді JSON. На фінальному етапі алгоритму клієнтський застосунок використовує механізм віртуального дерева документів (Virtual DOM) для ефективного оновлення графіків без повного перезавантаження сторінки [25]. Послідовність взаємодії компонентів системи під час виконання цього алгоритму відображено на діаграмі послідовності (рис. 2.7).

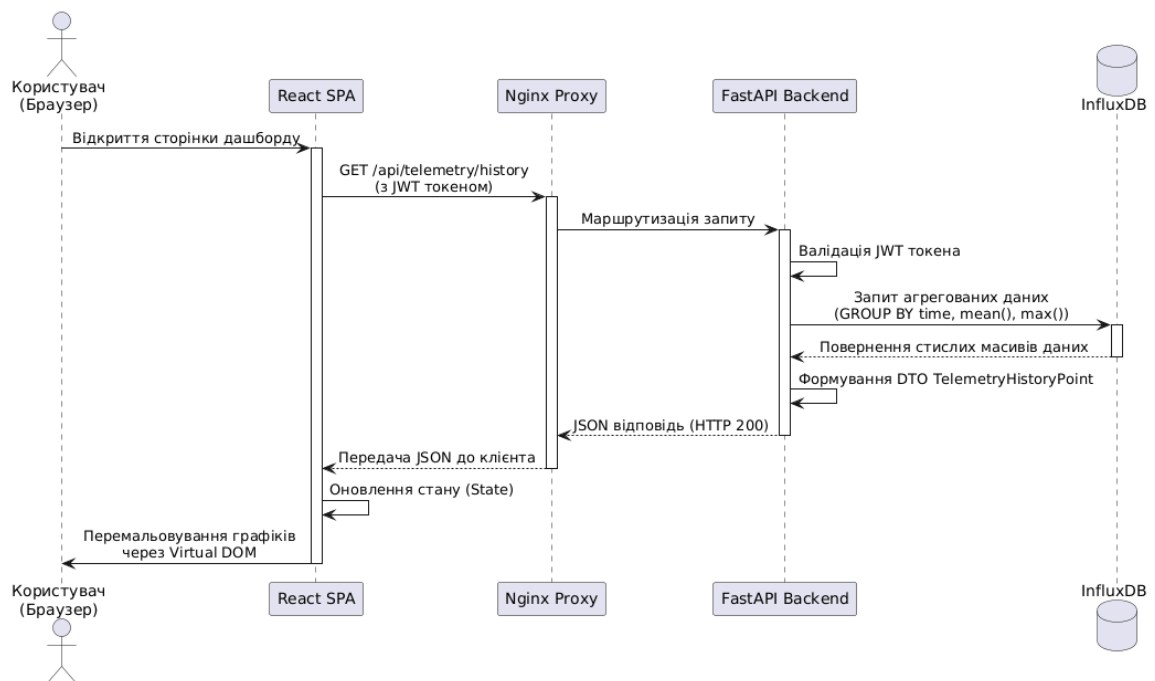


Рис. 2.7. UML діаграма послідовності алгоритму агрегації та візуалізації телеметрії

2.4. Реалізація вебсистеми моніторингу

Практична реалізація серверного ядра системи виконана мовою програмування Python із застосуванням високопродуктивного асинхронного фреймворку FastAPI [8, 24]. Фундаментальним елементом бекенду є підсистема прийому та первинної обробки телеметрії, яка функціонує за моделлю програмного інтерфейсу REST API [26]. Обробка вхідного потоку даних від програмних агентів здійснюється через спеціалізований маршрутизатор (контролер), який приймає мережеві запити за методом HTTP POST. Важливою архітектурною особливістю реалізованого контролера є інтенсивне використання механізму впровадження залежностей (Dependency Injection), що дозволяє ізолювати логіку автентифікації від безпосередньої бізнес-логіки обробки пакетів телеметрії.

Для забезпечення інформаційної безпеки та захисту від несанкціонованого доступу реалізовано залежність автентифікації агента. Під час кожного звернення до ендпоінту маршрутизатор автоматично отримує криптографічний ключ із заголовка авторизації (Bearer token) та виконує асинхронний запит до реляційної бази даних PostgreSQL за допомогою інструментарію SQLAlchemy

[21]. Функція автентифікації перевіряє наявність ключа в реєстрі та активний статус відповідного сервера. Якщо ключ недійсний або сервер деактивовано адміністратором, система миттєво відхиляє запит із кодом помилки доступу 401 Unauthorized. У разі успішної перевірки, об'єкт ідентифікованого сервера передається до основного тіла контролера для подальшої обробки.

З метою уникнення підміни джерела даних (spoofing) реалізовано додатковий механізм перехресної перевірки. Система порівнює ідентифікатор сервера, який міститься у тілі JSON-запиту, з ідентифікатором сервера, якому фактично належить наданий API-ключ. Заборона обробки запиту при виявленні невідповідності (генерація помилки 403 Forbidden) гарантує, що скомпрометований агент не зможе надсилати хибну телеметрію від імені інших вузлів інфраструктури.

Корисне навантаження запиту проходить сувору типову та семантичну валідацію за допомогою об'єкта передачі даних, реалізованого на базі бібліотеки Pydantic. Після успішного проходження всіх етапів перевірки контролер ініціює паралельне виконання ключових асинхронних операцій: виклик методу фізичного запису сирих часових рядів до спеціалізованої бази InfluxDB та запуск сервісної функції перевірки порогових значень (_check_thresholds). На цьому ж етапі здійснюється оновлення часової мітки останньої активності сервера (last_seen) та фіксація змін у реляційній базі даних. Особливістю реалізації є інтеграція механізму сповіщень: для кожного згенерованого інциденту контролер створює окреме фонове завдання (asyncio.create_task) для надсилання повідомлень через месенджер Telegram, що дозволяє не блокувати основний цикл обробки HTTP-відповіді. Лістинг програмної реалізації маршрутизатора прийому телеметрії та функції автентифікації агента наведено у додатку А.

Реалізація бізнес-логіки управління інцидентами становить основу реактивної підсистеми моніторингу програмного комплексу. Програмна імплементація цього механізму інкапсульована в асинхронній сервісній функції, яка отримує на вхід екземпляр ідентифікованого сервера, об'єкт передачі даних із показниками телеметрії та поточну сесію бази даних. На початковому етапі

алгоритму з отриманого об'єкта отримуються актуальні значення апаратних метрик, після чого ініціюється асинхронний запит до реляційної СУБД PostgreSQL за допомогою методів бібліотеки SQLAlchemy. Цей запит дозволяє отримати масив усіх налаштованих порогових значень, які адміністратор встановив для цього конкретного сервера.

Процес аналізу стану інфраструктури виконується шляхом ітеративної перевірки кожної метрики за допомогою спеціалізованого логічного предиката. Особливістю реалізації цього предиката є диференційований підхід до оцінки апаратних характеристик: для показників використання процесора, оперативної пам'яті та графічного прискорювача критичним станом вважається перевищення заданого порогу, тоді як для показника вільного дискового простору інцидент ініціюється лише при падінні значення нижче встановленої адміністратором межі. З метою запобігання надмірному навантаженню на базу даних та уникнення дублювання записів впроваджено механізм відстеження поточного стану. Перед генерацією нового інциденту програмний код виконує додатковий SQL-запит для перевірки наявності вже відкритого збою для цього сервера та конкретної метрики. Якщо інцидент із відповідним статусом вже існує в системі, процедура створення нового запису блокується, що повністю усуває проблему лавинного дублювання повідомлень. У разі відсутності такого запису створюється новий об'єкт класу Incident, якому присвоюється статус відкритого та додається форматований текстовий опис із зазначенням поточного та критичного значень.

На рівні програмного коду також імплементовано алгоритм автоматичного розв'язання (auto-resolve), який відповідає за закриття інцидентів без втручання людини. У випадку, коли значення метрики не перевищує поріг (апаратне забезпечення функціонує в штатному режимі), алгоритм перевіряє наявність незакритого інциденту в базі даних. Якщо такий запис виявляється, його стан модифікується: статус змінюється на вирішений, у поле часу закриття записується поточна часова мітка у стандарті UTC, а до первинного опису інциденту додається системний запис про повернення показників до норми. Усі

згенеровані під час ітерації попередження та повідомлення про відновлення стабільної роботи накопичуються у вигляді текстового масиву, який повертається до головного маршрутизатора для створення незалежних фонових завдань із надсилання сповіщень у TelegramBot. Лістинг алгоритму управління інцидентами наведено у додатку Б.

Практична реалізація підсистеми збереження часових рядів відображає концепцію поліглотного зберігання на рівні програмного коду бекенду. Взаємодія з нереляційною базою даних InfluxDB інкапсульована у спеціалізованому класі-менеджері, який абстрагує низькорівневі операції роботи з мережевим клієнтом. Для конфігурації параметрів підключення (мережевої адреси, криптографічного токена, назви організації та ідентифікатора логічного контейнера bucket) застосовано підхід, заснований на читанні змінних оточення операційної системи. Це архітектурне рішення відповідає сучасній методології побудови застосунків (Twelve-Factor App) та дозволяє безпечно розгортати систему у різних середовищах без втручання у вихідний код. Під час ініціалізації екземпляра клієнта також встановлюється жорсткий мережевий тайм-аут у 10000 мілісекунд, що запобігає зупинці серверного потоку у випадку недоступності бази даних.

Центральним елементом програмної реалізації є механізм формування точки телеметрії (об'єкта класу Point). Формування запису здійснюється з використанням каскадного виклику методів (method chaining), де базовому вимірюванню присвоюється константна назва server_metrics. Згідно з попереднім проєктуванням, програмний код чітко розмежовує індексовані метадані та безпосередні значення: ідентифікатор сервера та тип операційної системи додаються до запису за допомогою методу tag(), забезпечуючи їх фізичне індексування рушієм бази даних [13]. Водночас усі апаратні метрики додаються через метод field() із суворим приведенням типів (до цілочисельного формату int або числа з плаваючою крапкою float). Варто відзначити високу гнучкість реалізованого алгоритму: окрім статичного набору метрик (оперативна пам'ять, дисковий простір, мережа), код динамічно ітерує масив завантаженості

окремих ядер процесора та опціонально додає телеметрію графічного прискорювача за умови її наявності у вхідному пакеті.

Кожній сформованій точці телеметрії автоматично присвоюється мітка часу у стандарті UTC із наносекундною точністю. Запис об'єкта до фізичного сховища виконується через програмний інтерфейс клієнта `write_api` у жорсткому синхронному режимі (SYNCHRONOUS). Використання синхронного клієнта всередині асинхронного фреймворку FastAPI є свідомим архітектурним компромісом: хоча це потенційно блокує цикл подій (event loop) на час очікування відповіді від бази, такий підхід гарантує успішне підтвердження збереження запису (durability) на рівні застосунку. Серверний потік примусово чекає відповіді від рушія InfluxDB перед тим, як повернути успішний статус програмному агенту [14]. Водночас у коді передбачено архітектурне зауваження для подальшого масштабування: при екстремально високому навантаженні (High RPS) синхронний клієнт може бути безболісно замінений на використання єдиного екземпляра `write_api` у пакетному (batch) асинхронному режимі роботи для уникнення блокування циклу подій. Лістинг програмної реалізації ініціалізації клієнта бази даних та підсистеми запису часових рядів наведено у додатку В.

Фізична реалізація підсистеми збору даних виконана у вигляді автономного програмного агента на базі мови Python. Архітектура скрипта спроектована за принципом фонового демона, який функціонує в нескінченному асинхронному циклі. За безпосередню взаємодію з ядром операційної системи та зчитування апаратних метрик відповідає бібліотека `psutil` [23]. Програмний код агента реалізує збір розширеного спектра системних показників, зокрема: використання обчислювальних ресурсів центрального процесора (як загального, так і по кожному ядру окремо), обсяг зайнятої та загальної оперативної пам'яті, залишок вільного дискового простору та динаміку мережевого трафіку. Для коректного обчислення швидкості операцій вводу-виводу (I/O) на диск та швидкості передачі даних мережею, скрипт використовує механізм збереження попередніх значень у глобальних змінних (`_prev_disk_read`, `_prev_net_in` тощо) із

наступним обчисленням різниці за встановлений інтервал опитування. Додатково імплементовано інтеграцію з утилітою `nvidia-smi` для отримання телеметрії графічних прискорювачів (у разі їх наявності), що робить агента універсальним інструментом для моніторингу різних типів серверного обладнання.

Визначною архітектурною перевагою реалізованого агента є механізм відмовостійкості, що базується на використанні локальної вбудованої СУБД `SQLite` для потреб буферизації. На відміну від зберігання даних в оперативній пам'яті (яка очищується при перезапуску процесу), використання `SQLite` гарантує збереження історії телеметрії навіть у випадку критичного збою живлення цільового сервера. Процес надсилення метрик реалізовано за допомогою асинхронної `HTTP`-бібліотеки `aiohttp` [1]. Алгоритм перевіряє стан локальної бази даних перед кожним запитом: якщо з'єднання із центральним сервером втрачено (наприклад, через виникнення винятку `aiohttp.ClientError`), агент автоматично перенаправляє сформований `JSON`-пакет до таблиці `buffer` у файлі `buffer.db` [29].

При відновленні мережевого зв'язку система переходить у режим масового надсилення (`bulk upload`). Агент ініціює зчитування всіх накопичених історичних точок із локальної бази даних, об'єднує їх із поточними показниками в єдиний масив і надсилає на спеціалізований ендпоінт бекенду (`POST /api/telemetry/bulk`). У разі отримання підтвердження від сервера про успішне збереження (`HTTP 201 Created`), агент атомарно очищує локальну таблицю буфера, після чого повертається до штатного режиму надсилення одиничних пакетів. Життєвим циклом програми керує головна асинхронна функція `main()`, яка підтримує єдиний пул мережевих з'єднань (`ClientSession`) для оптимізації витрат ресурсів на встановлення `TCP`-з'єднань. Лістинг реалізації основного циклу програмного агента, функцій збору метрик та логіки буферизації наведено у додатку Г.

Реалізація клієнтської частини програмного комплексу виконана за патерном односторінкового застосунку (`Single Page Application`) з використанням бібліотеки `React` [25]. Для забезпечення швидкого холодного

старту середовища розробки та глибокої оптимізації фінальної статичної збірки застосовано сучасний інструментарій Vite [31], який використовує нативні модулі ECMAScript. Навігація між логічними екранами системи (наприклад, перехід від загального списку серверів до детальної інформаційної панелі конкретного вузла) здійснюється лише на стороні браузера за допомогою бібліотеки клієнтського маршрутизатора, що виключає необхідність повного перезавантаження вебсторінки при кожній дії адміністратора. Особливу увагу під час розробки інтерфейсу було приділено ергономіці та зниженню когнітивного навантаження: для обробки конфігураційних форм (додавання серверів, налаштування порогових значень) впроваджено парадигму делікатної валідації на базі React Hook Form. Замість блокуючих модальних вікон або агресивних червоних панелей, система використовує вбудовані текстові підказки (inline hints), які з'являються безпосередньо під невалідними полями вводу.

Головним аналітичним вузлом користувацького інтерфейсу виступає комплексний компонент візуалізації телеметрії. Програмна імплементація цього компонента базується на використанні функціональних підходів та хуків управління станом (зокрема, useState та useEffect). Після монтування компонента у віртуальне дерево документів (Virtual DOM), система ініціює асинхронний HTTP-запит до програмного інтерфейсу бекенду для отримання історичних даних за обраний період. Отриманий масив сирих даних проходить етап обов'язкової нормалізації безпосередньо на стороні клієнта: показники обсягу оперативної пам'яті та дискового простору конвертуються з байтів у гігабайти, швидкості – у мегабайти за секунду, а вкладені масиви навантаження на окремі ядра процесора розгортаються у пласку структуру (flattening) для коректної побудови лінійних графіків.

Для забезпечення безперервності відображення даних у режимі реального часу реалізовано алгоритм тихого фонового опитування (silent polling). Під час першого завантаження або при зміні часового проміжку користувачеві демонструється індикатор завантаження. Надалі компонент встановлює інтервальний таймер, який періодично здійснює запити за новими даними.

Ключовою особливістю цього алгоритму є блокування зміни стану завантаження під час фонових оновлень, завдяки чому графіки оновлюються плавно, без візуального мерехтіння або перекриття інтерфейсу індикаторами завантаження. Безпосередня побудова графіків реалізована за допомогою декларативної бібліотеки Recharts, яка створює оптимізовані векторні (SVG) елементи з підтримкою градієнтних заливок, динамічних осей та інтерактивних підказок при наведенні курсора. Лістинг програмної реалізації клієнтського компонента візуалізації з алгоритмом фонових опитувань наведено у додатку Г.

Розгортання програмного комплексу та забезпечення надійної ізоляції його компонентів реалізовано за допомогою технології контейнеризації Docker та інструменту декларативного управління інфраструктурою Docker Compose [6]. Інфраструктура системи спроектована у вигляді набору незалежних мікросервісів, кожен з яких інкапсульовано у власному ізолюваному середовищі з чітко визначеними системними залежностями [17]. Конфігурація оркестратора описує чотири основні сервіси: транзакційну базу даних PostgreSQL, спеціалізоване сховище часових рядів InfluxDB, прикладний серверний застосунок (бекенд) та клієнтський вебсервер (фронтенд). Для гарантування збереження накопиченої телеметрії та конфігураційної інформації при перезапуску або плановому оновленні контейнерів, бази даних фізично прив'язані до іменованих постійних томів (named volumes) на рівні файлової системи хост-машини. Критично важливою особливістю процесу ініціалізації системи є використання вбудованого механізму перевірки стану працездатності (healthchecks). Оркестратор періодично виконує низькорівневі системні виклики (зокрема, утиліту pg_isready для PostgreSQL та influx ping для InfluxDB) і жорстко блокує запуск контейнера бекенду до того моменту, поки обидві підсистеми баз даних не підтвердять свою повну готовність до обробки вхідних мережевих з'єднань.

З точки зору мережевої безпеки та дотримання принципів глибокоєшелонованого захисту, розроблена архітектура розгортання реалізує сувору ізоляцію внутрішніх портів [30]. Контейнери бази даних PostgreSQL та

прикладного інтерфейсу FastAPI принципово не мають відкритих портів, що «дивляться» у глобальну мережу. Їхня взаємодія здійснюється суто в межах закритої внутрішньої віртуальної мережі (bridge network) за допомогою вбудованого механізму вирішення доменних імен Docker DNS. Передача конфіденційної конфігураційної інформації, такої як криптографічні ключі, паролі доступу до баз даних, JWT-секрети та токени інтеграції з API Telegram, здійснюється динамічно під час запуску через змінні оточення (environment variables). Це унеможливорює витік секретів через систему контролю версій.

Єдиною відкритою точкою входу для кінцевих користувачів виступає контейнер фронтенду, який прослуховує стандартний HTTP-порт 80. Усередині цього контейнера розгорнуто високопродуктивний вебсервер Nginx, який виконує функцію зворотного проксі-сервера (Reverse Proxy) [18]. Nginx приймає весь вхідний трафік, самостійно надає статичні скомпільовані файли React-застосунку, а всі запити, що містять префікс прикладного інтерфейсу (наприклад, /api/telemetry), прозора маршрутизує до ізольованого контейнера бекенду. Таким чином забезпечується єдина точка доступу, що додатково захищає внутрішні компоненти від прямих мережових атак. Лістинг конфігураційного файлу оркестратора контейнерів, який описує зазначену інфраструктуру та правила мережової маршрутизації, наведено у додатку Д.

Автоматизація процесів безперервного розгортання (CI/CD) програмного комплексу реалізована за допомогою інструментарію платформи GitLab [12]. Для мінімізації людського фактора під час оновлення системи на цільовому хмарному сервері Microsoft Azure було розгорнуто локальний агент GitLab Runner, який налаштовано для функціонування у режимі системної оболонки (shell executor). Процес автоматизованого розгортання ініціюється жорстко за правилом гілкування: тригер конвеєра спрацьовує тільки при злитті нового вихідного коду до головної гілки репозиторію (master). При активації конвеєр виконує єдину стадію розгортання (deploy), що складається з послідовності команд системного рівня.

На першому етапі скрипт розгортання виконує синхронізацію файлів проєкту з репозиторію у цільову робочу директорію сервера за допомогою системної утиліти rsync. Критично важливою інженерною деталлю цієї операції є явне виключення з процесу копіювання службових каталогів системи контролю версій (.git) та, що найважливіше, локальної бази даних SQLite (buffer.db). Це архітектурне рішення гарантує збереження накопиченого буфера телеметрії та запобігає випадковому перезапису робочого стану системи під час розгортання нової версії. Після успішної синхронізації файлів оркестратор Docker Compose здійснює м'яку зупинку поточних контейнерів без обриву активних з'єднань баз даних, ініціює перекомпіляцію змінених образів та запускає оновлені сервіси у фоновому режимі (detached mode). Завершальним кроком конвеєра є автоматичне очищення дискового простору від залишкових (dangling) образів Docker за допомогою команди примусового очищення. Такий підхід дозволяє підтримувати стабільну роботу хмарного сервера в умовах обмежених ресурсів та унеможливорює переповнення накопичувача після багатьох ітерацій розгортання. Лістинг конфігураційного файлу конвеєра автоматизованого розгортання наведено у додатку Е.

Висновки до другого розділу

У другому розділі було здійснено комплексне проєктування та програмну реалізацію інформаційного забезпечення і компонентів вебсистеми моніторингу серверного обладнання. На початковому етапі формалізовано функціональні та нефункціональні вимоги до програмного продукту, визначено ключові часові характеристики реакції системи та побудовано діаграму варіантів використання, що розмежовує рівні доступу для системних адміністраторів, спостерігачів та автономних програмних агентів. Для забезпечення гнучкості архітектури спроектовано об'єктно-орієнтовану структуру системи із застосуванням патерну об'єктно-реляційного відображення для бізнес-логіки та об'єктів передачі даних (DTO) для валідації потоків телеметрії.

Фундаментальним етапом роботи стала розробка фізичної схеми бази даних за гібридною (поліглотною) моделлю. Проєктування реляційної частини

на базі PostgreSQL забезпечило дотримання транзакційної цілісності при зберіганні облікових записів, реєстру серверів та конфігурацій порогових значень із використанням механізмів каскадного видалення. Водночас проєктування структури сховища InfluxDB дозволило оптимізувати збереження великих масивів часових рядів завдяки чіткому розділенню індексованих тегів (ідентифікатори серверів) та неіндексованих полів (апаратні метрики).

На основі розроблених структур даних було спроектовано та візуалізовано алгоритмічне забезпечення програмного комплексу. Для автономного агента розроблено алгоритм безперервного збору метрик із вбудованим механізмом відмовостійкості, що забезпечує буферизацію телеметрії у локальній базі даних при втраті мережевого з'єднання. На серверному рівні реалізовано алгоритм реактивного управління інцидентами, який включає механізми захисту від лавинного дублювання записів та автоматичного розв'язання інцидентів після стабілізації показників апаратного забезпечення. Додатково спроектовано алгоритм агрегації та нормалізації даних для їх оптимальної передачі клієнтському застосунку.

Завершальним етапом стала безпосередня практична реалізація всіх спроектованих алгоритмів та підсистем мовами програмування Python та JavaScript. Розроблено серверне ядро на базі асинхронного фреймворку FastAPI, створено автономного програмного агента з використанням бібліотек psutil та aiohttp, а також імплементовано клієнтський односторінковий застосунок на базі React із застосуванням парадигми делікатної валідації інтерфейсу. Для забезпечення відтворюваності середовища, ізоляції внутрішніх компонентів та виконання вимог мережевої безпеки налаштовано інфраструктуру контейнеризації Docker Compose із використанням зворотного проксі-сервера Nginx. Успішне налаштування конвеєра безперервної інтеграції та розгортання GitLab CI підтверджує готовність програмного комплексу до етапу комплексного тестування та дослідження результатів роботи, що відкриває шлях до фінального етапу виконання кваліфікаційної роботи.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ВЕБСИСТЕМОЮ МОНІТОРИНГУ ТА СПОВІЩЕННЯ ПРО СТАН СЕРВЕРНОГО ОБЛАДНАННЯ

3.1 Порядок встановлення та налаштування параметрів системи

Розгортання програмного комплексу базується на централізованій моделі управління з децентралізованими вузлами збору телеметрії. Завдяки використанню інструментарію контейнеризації Docker та оркестратора Docker Compose [6], ядро системи є кросплатформним: його можна розгорнути як на серверах під управлінням операційних систем сімейства Linux (рекомендуються дистрибутиви на базі Debian або Ubuntu), так і на базі ОС Windows. Аналогічно, вибір середовища хостингу є гнучким: у межах цієї роботи як цільову машину використано хмарне середовище Microsoft Azure [17], однак система здатна функціонувати на будь-якому віртуальному або фізичному сервері відповідної потужності. Процес ініціалізації інфраструктури розпочинається з безпечного копіювання файлів проєкту на обраний сервер та детального налаштування глобального конфігураційного файлу оточення (файлу `.env`).

Цей файл відіграє роль єдиного центру управління безпекою та параметрами підключення. У ньому адміністратор визначає критичні змінні середовища, які динамічно передаватимуться до ізольованих контейнерів під час їх запуску. До обов'язкових параметрів належать конфігурації доступу до реляційної бази даних PostgreSQL (змінні `POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_DB`) та налаштування первинної ініціалізації сховища часових рядів InfluxDB, включаючи назву організації, назву логічного контейнера (`bucket`) та криптографічний токен адміністратора (`INFLUXDB_ADMIN_TOKEN`). Окремо генерується та фіксується криптографічний секрет (`JWT_SECRET`), який використовується бекендом на базі FastAPI для підпису та верифікації токенів сесій користувачів. Такий підхід гарантує, що жодні критичні дані та паролі не зберігаються безпосередньо у вихідному коді застосунку, що повністю відповідає принципам інформаційної безпеки.

Окремим важливим етапом базового налаштування є конфігурація підсистеми сповіщень. Системний адміністратор має створити віртуального бота через сервіс BotFather для отримання маркера доступу (TELEGRAM_BOT_TOKEN) та ініціювати з ним діалог, щоб зафіксувати ідентифікатор цільового чату (TELEGRAM_CHAT_ID). Ці два параметри, необхідні для подальшої маршрутизації критичних сповіщень про інциденти, також інтегруються до загального файлу оточення.

Після збереження конфігурації запуск інфраструктури здійснюється за допомогою стандартної команди оркестратора docker-compose up -d для старту в ізольованому фоновому режимі. Оркестратор автоматично створює захищену віртуальну мережу (bridge network), монтує іменовані постійні томи (named volumes) для забезпечення стійкості баз даних до перезавантажень та ініціює послідовний запуск контейнерів. Критичною ланкою цього процесу є виконання вбудованих механізмів перевірки працездатності (healthchecks) [6]. Контейнери баз даних періодично опитуються системними утилітами (pg_isready та influx ring), і лише після підтвердження їхньої повної готовності до обробки вхідних мережевих запитів оркестратор дозволяє запуск контейнера серверного бекенду. Це унеможливорює виникнення стану «гонитви» (race condition) та помилок підключення на етапі старту.

Наступним кроком є підготовка до розгортання автономних програмних агентів на цільовому серверному обладнанні підприємства. Відповідно до принципів корпоративної безпеки [30], підключення будь-якого нового вузла вимагає його обов'язкової попередньої реєстрації в системі. Адміністратор здійснює авторизований вхід до клієнтського односторінкового вебзастосунку, ініціює процедуру додавання нового сервера через відповідну екранну форму та визначає його назву й тип операційної системи. У результаті серверне ядро алгоритмічно генерує універсальний унікальний ідентифікатор вузла (UUID) та криптографічно стійкий API-ключ. Ці згенеровані облікові дані переносяться до локального конфігураційного файлу .env самого програмного агента,

забезпечуючи його авторизований та ідентифікований доступ до центрального маршрутизатора телеметрії.

Для цільових серверів під управлінням операційної системи Windows реалізовано парадигму розгортання з нульовими залежностями (zero-dependency). Вихідний код агента компілюється в автономний виконуваний файл [24] із застосуванням спеціальних інструкцій, що приховують вікно командного рядка та перетворюють скрипт на фоновий процес. Це позбавляє адміністратора необхідності встановлювати інтерпретатор Python чи додаткові бібліотеки, зводячи процес інсталяції до простого копіювання виконуваного та конфігураційного файлів у робочу директорію.

Розгортання програмного агента на серверах сімейства Linux (зокрема, для цілей самомоніторингу центрального хмарного сервера) здійснюється з використанням механізмів віртуалізації середовища та інтеграції з підсистемою ініціалізації операційної системи. Після копіювання вихідного коду скрипта на цільовий вузол створюється ізольоване віртуальне середовище (virtual environment) за допомогою вбудованого модуля інтерпретатора. Встановлення необхідних програмних залежностей виконується шляхом прямого виклику менеджера пакетів із зазначенням конкретних бібліотек (зокрема aiohttp, psutil та python-dotenv). Для забезпечення індустріального стандарту безперебійної фонові роботи агента в Linux-середовищі, розгортання не обмежується простим запуском скрипта. Натомість створюється повноцінний конфігураційний файл системного сервісу (systemd). Цей файл вказує операційній системі робочу директорію агента, шлях до ізольованого інтерпретатора та визначає політику автоматичного перезапуску у випадку непередбаченого завершення процесу (Restart=always). Стандартний потік виводу та потік помилок перенаправляються у спеціалізований лог-файл (наприклад, agent_azure.log) для можливості подальшого аудиту та діагностики роботи алгоритму буферизації. Під час першого успішного запуску агент автоматично формує локальну базу даних SQLite для забезпечення відмовостійкості, після чого переходить у штатний

режим збору апаратної телеметрії. Загальну логічну схему розгортання ресурсів та мережевої взаємодії компонентів системи наведено на рисунку (рис. 3.1).

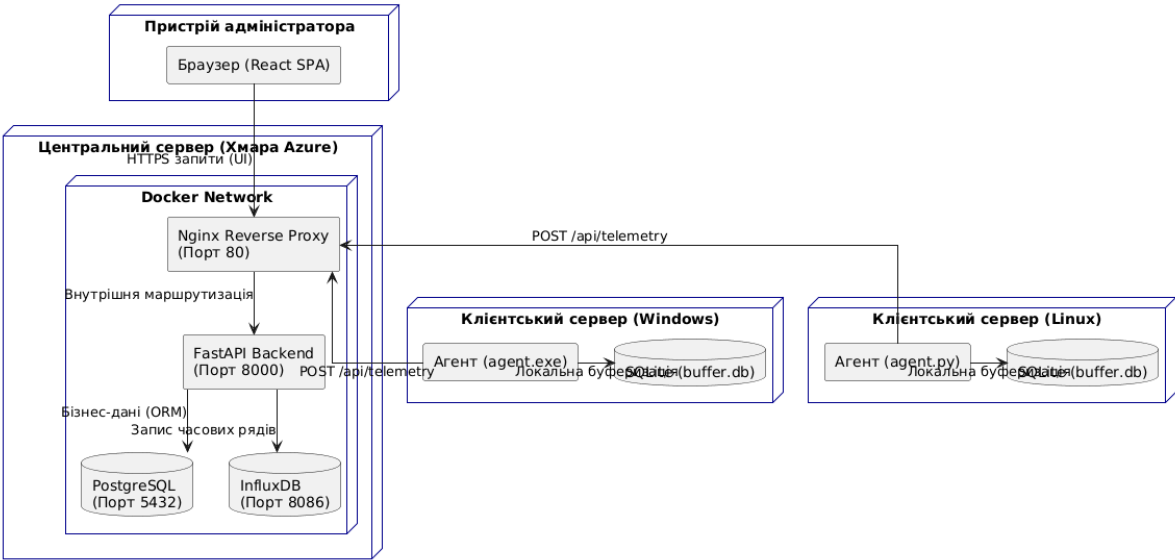


Рис. 3.1. UML Діаграма розгортання ресурсів та компонентів системи моніторингу.

3.2 Структура інтерфейсу вебсистеми

Проектування графічного інтерфейсу користувача та визначення потоків інформації в системі реалізовано згідно з концепцією односторінкового застосунку (Single Page Application). Клієнтська частина базується на екосистемі бібліотеки React та використовує інструментарій Vite для забезпечення швидкої збірки та оптимізації модулів [25, 31]. Фундаментальний процес управління операціями та потік даних в інтерфейсі будується на механізмі реактивного оновлення стану віртуального дерева документів (Virtual DOM). Будь-яка взаємодія адміністратора з елементами керування, така як перехід між сторінками або зміна параметрів фільтрації, не викликає повного перезавантаження вебсторінки. Натомість система ініціює зміну локального стану компонентів, що формує відповідні асинхронні HTTP-запити до програмного інтерфейсу. Запити проходять через зворотний проксі-сервер Nginx, який маршрутизує їх до бекенду на базі FastAPI. Серверна частина виконує необхідну бізнес-логіку, витягує інформацію з баз даних PostgreSQL або InfluxDB та повертає результат у форматі структурованого JSON-документа.

Отримані дані десеріалізуються на стороні клієнта та використовуються для миттєвої перебудови лише тих частин інтерфейсу, що потребують оновлення.

Важливою складовою загальної архітектури клієнтського застосунку є забезпечення високого рівня ергономіки та адаптивності під індивідуальні потреби оператора. Для зниження зорового навантаження під час тривалого моніторингу на глобальному рівні інтерфейсу імплементовано механізм перемикання візуальних тем (світла та темна). Цей функціонал реалізовано за допомогою глобального контексту React (Context API), який динамічно змінює каскадні таблиці стилів для всіх дочірніх компонентів системи та зберігає обрані налаштування у локальному сховищі браузера (LocalStorage) для їх відновлення при наступних сеансах авторизації. Узагальнену архітектуру клієнтського застосунку, процес управління операціями та потік даних між ключовими програмними компонентами системи візуалізовано на відповідній діаграмі (рис. 3.2).

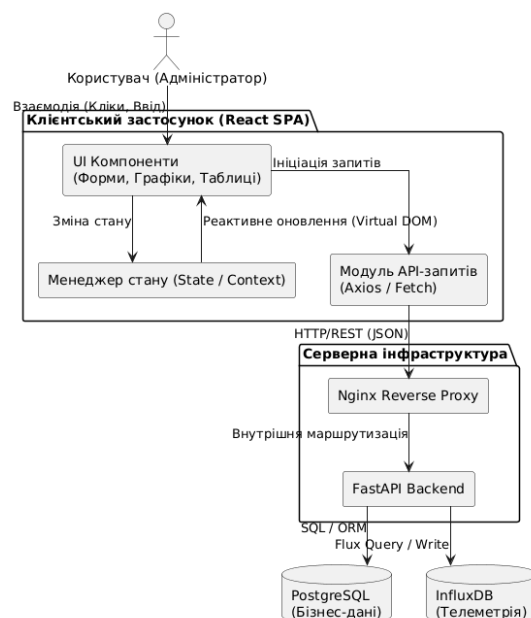


Рис. 3.2. UML Діаграма компонентів та потоку даних вебсистеми моніторингу

Взаємодія із системою розпочинається з екрана автентифікації, представленого центрованою формою з полями введення та глобальним перемикачем візуальної теми. Первинна валідація вводу використовує нативні механізми браузера для блокування порожніх запитів та очищення логіна від

початкових та кінцевих пробілів. Після успішного асинхронного запиту сервер повертає JWT-токен доступу (access_token), який зберігається у локальному сховищі браузера. Хоча такий підхід потенційно створює ризики XSS-атак порівняно з використанням захищених HTTP-only cookies, у межах цієї роботи це є свідомим архітектурним компромісом. Безпека гарантується тим, що вебсистема працює у закритому корпоративному контурі без можливості реєстрації сторонніх користувачів, а клієнтський застосунок на базі React використовує вбудовані механізми автоматичного екранування (escaping) вхідних даних, що суттєво ускладнює ін'єкцію шкідливих скриптів. Далі паралельно запитується профіль користувача та його рольова модель для запису у глобальний стан застосунку. При неправильних даних інтерфейс тимчасово блокує кнопку входу і виводить повідомлення про помилку. Реалізовано також механізм відновлення сеансу: при завантаженні сторінки система валідує збережений токен і автоматично очищує сесію у разі його закінчення (експірації). Інтерфейс форми авторизації наведено на рисунку (рис. 3.3).

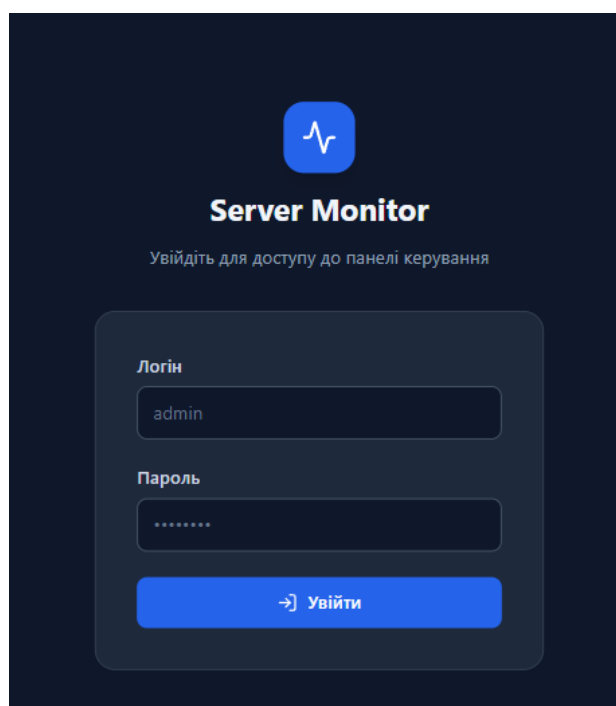


Рис. 3.3. Екран авторизації користувача (логін та пароль)

Після автентифікації користувач перенаправляється на головну інформаційну панель (Dashboard), поділену на декілька логічних зон. У верхній частині розміщено навігаційні елементи, ідентифікатор користувача та

					ІПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						56

посилання на інтеграцію з Telegram-ботом. Нижче розташовано блок з чотирьох карток-лічильників, що відображають загальну кількість зареєстрованих серверів, вузли у статусі «Online» та «Offline», а також поточну кількість активних інцидентів. Головним елементом дашборду виступає зведена таблиця серверного парку. Загальний вигляд головної інформаційної панелі наведено на рисунку (рис. 3.4).

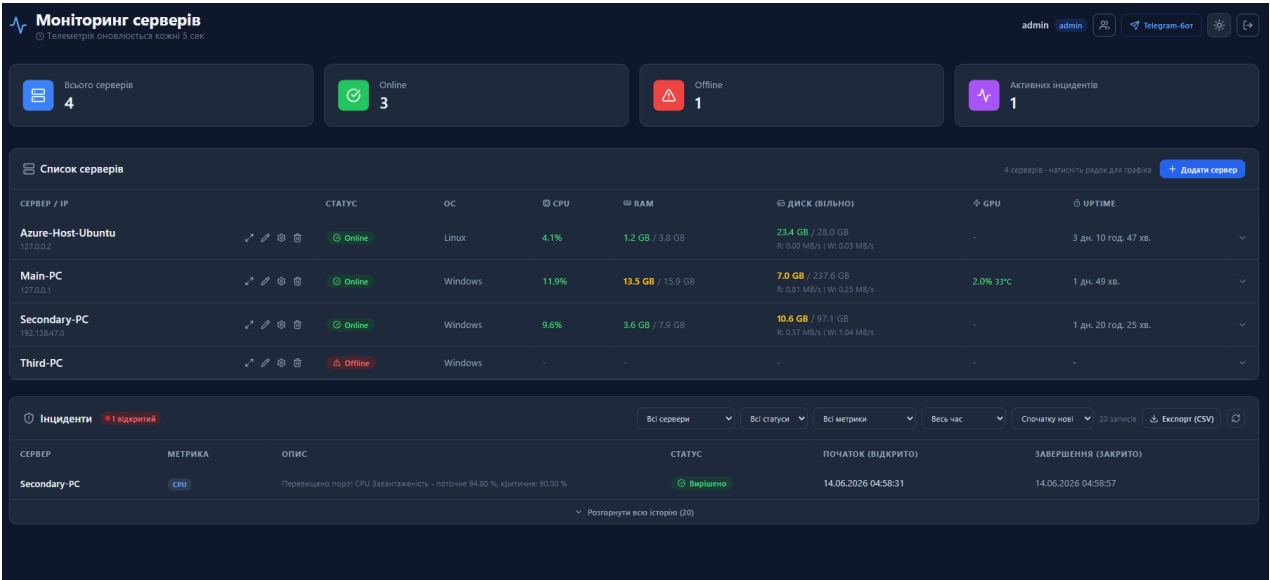


Рис. 3.4. Головна інформаційна панель (Dashboard) – загальний вигляд

Зведена таблиця надає оперативну інформацію про кожен підключений сервер: ідентифікаційні дані, статус з'єднання, час безперервної роботи (Uptime) та ключові апаратні метрики (CPU, обсяг RAM у гігабайтах, вільний дисковий простір зі швидкістю I/O, показники GPU). Важливою ергономічною особливістю є система кольорового кодування: значення підсвічуються зеленим, жовтим або червоним залежно від налаштованих індивідуальних порогів сервера. Для вільного дискового простору реалізовано інверсну логіку, де червоний колір активується при падінні показника нижче встановленої межі. Для підтримки актуальності даних клієнтський застосунок виконує диференційовані фонові запити: з метою оптимізації навантаження статус списку серверів оновлюється кожні 15 секунд, а динамічна телеметрія – кожні 5 секунд паралельним потоком.

Для глибшого аналізу безпосередньо у рядку сервера розгортається панель телеметрії на базі SVG-графіків Recharts із вкладками загального огляду, CPU (із

деталізацією навантаження на окремі обчислювальні ядра), RAM, Disk та GPU. Водночас показники мережевого трафіку на даному етапі розвитку системи акумулюються у базі даних для фонові аналітики та доступні через програмний інтерфейс (API), що створює основу для подальшого розширення візуальних панелей. Ключовою функцією є перемикач часового діапазону для перегляду агрегованої історії у межах від 5 хвилин до 7 днів. Алгоритм застосунку динамічно адаптує частоту фонового оновлення графіків (silent polling) залежно від обраного часового діапазону, що забезпечує ефект моніторингу в реальному часі для коротких проміжків. Вигляд розгорнутої метрики сервера з візуалізацією графіків наведено на рисунку (рис. 3.5).



Рис. 3.5. Розгорнута метрика конкретного сервера на головній панелі з візуалізацією графіків

Підсистема управління інцидентами інтегрована безпосередньо у головну інформаційну панель і представлена у вигляді інтерактивного журналу. Структура таблиці інцидентів містить вичерпну інформацію про збої: назву сервера, тип метрики з кольоровим індикатором, розширений текстовий опис проблеми, поточний статус (відкритий або вирішений), а також точні часові мітки виникнення та завершення. Для облікових записів з правами адміністратора реалізовано можливість інтерактивної взаємодії з відкритими

інцидентами: функція взяття в роботу відкриває спеціальне модальне вікно для додавання коментарів без зміни загального статусу збою. Журнал підтримує два режими відображення: згорнутий, де демонструється лише найновіший запис, та розгорнутий, у якому активується посторінкова навігація з відображенням десяти записів на сторінку. Для миттєвого інформування оператора застосунок запитує дозвіл та використовує нативний програмний інтерфейс браузера для виведення системних сповіщень при появі нових записів. Дані в журналі оновлюються асинхронно кожні 5 секунд.

Для ефективної роботи з великими масивами історичних даних у шапці журналу реалізовано комплексну систему фільтрації. Оператор має змогу сортувати та обмежувати записи за конкретним сервером, статусом інциденту, типом апаратної метрики, часовим періодом та напрямком сортування. Усі параметри фільтрації обробляються на стороні бекенду, а їх зміна автоматично скидає пагінацію на початкову сторінку та оновлює загальний лічильник знайдених збоїв. Розгорнуту таблицю інцидентів із демонстрацією пагінації та панелі фільтрів наведено на рисунку (рис. 3.6).

СЕРВЕР	МЕТРИКА	ОПИС	СТАТУС	ПОЧАТОК (ВІДКРИТО)	ЗАВЕРШЕННЯ (ЗАКРИТО)
Secondary-PC	CPU	Перевищено поріг CPU Завантаженість - поточне 94.80 %, критичн...	✓ Вирішено	14.06.2026 04:58:31	14.06.2026 04:58:57
Secondary-PC	CPU	Перевищено поріг CPU Завантаженість - поточне 99.30 %, критичн...	✓ Вирішено	13.06.2026 04:56:03	13.06.2026 04:57:01
Secondary-PC	CPU	Перевищено поріг CPU Завантаженість - поточне 99.80 %, критичн...	✓ Вирішено	12.06.2026 19:58:10	12.06.2026 19:58:15
Secondary-PC	CPU	Перевищено поріг CPU Завантаженість - поточне 92.50 %, критичн...	✓ Вирішено	12.06.2026 19:38:09	12.06.2026 19:38:40
Main-PC	CPU	Перевищено поріг CPU Завантаженість - поточне 97.90 %, критичн...	✓ Вирішено	11.06.2026 22:45:52	11.06.2026 22:45:57
Main-PC	CPU	Перевищено поріг CPU Завантаженість - поточне 16.60 %, критичн...	✓ Вирішено	11.06.2026 20:23:58	11.06.2026 20:24:23

Рис. 3.6. Розгорнута таблиця інцидентів із демонстрацією пагінації та фільтрів

Окремим функціональним модулем є підсистема генерації та експорту аналітичних звітів. Ініціація цієї функції відкриває спеціалізоване модальне вікно, де користувач задає необхідні критерії вибірки даних за періодом, метрикою та сервером. Після виконання відповідного HTTP-запиту клієнтський застосунок отримує масив даних та алгоритмічно перетворює його у формат CSV

безпосередньо у пам'яті браузеру. Для забезпечення повної сумісності з табличним процесором Microsoft Excel при відображенні кирилических символів, алгоритм експорту використовує крапку з комою як роздільник та автоматично додає маркер послідовності байтів (BOM). Сформований файл завантажується на пристрій користувача з динамічно згенерованою назвою, що логічно відображає застосовані параметри вибірки. Вікно завантаження звіту інцидентів із доступними інструментами сортування наведено на рисунку (рис. 3.7).

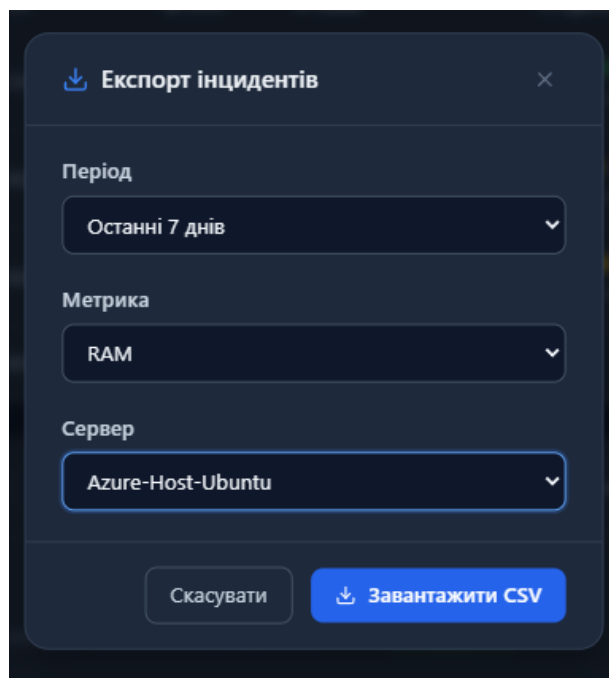


Рис. 3.7. Вікно завантаження звіту інцидентів із доступними способами сортування

Для детального моніторингу конкретного вузла інфраструктури розроблено індивідуальну сторінку сервера. Інтерфейс сторінки містить інформаційну шапку з метаданими, динамічним бейджем статусу та ідентифікатором системи. Для зручності адміністратора реалізовано функцію безпечного копіювання ідентифікатора в буфер обміну з візуальним підтвердженням дії. Центральну частину сторінки займає компонент візуалізації телеметрії, який синхронно оновлює графіки в режимі реального часу. Важливою архітектурною особливістю клієнтського застосунку є часткова імплементація рольової моделі доступу (RBAC) на рівні інтерфейсу. Елементи керування сервером, такі як кнопки редагування, налаштування порогів та

видалення, формуються у віртуальному дереві документів тільки для облікових записів із роллю адміністратора і повністю приховуються для звичайних спостерігачів. Це рішення є ергономічним заходом для розвантаження інтерфейсу, тоді як фундаментальне блокування несанкціонованих дій гарантується на рівні серверного ядра. Окрему сторінку сервера з панеллю керування наведено на рисунку (рис. 3.8).

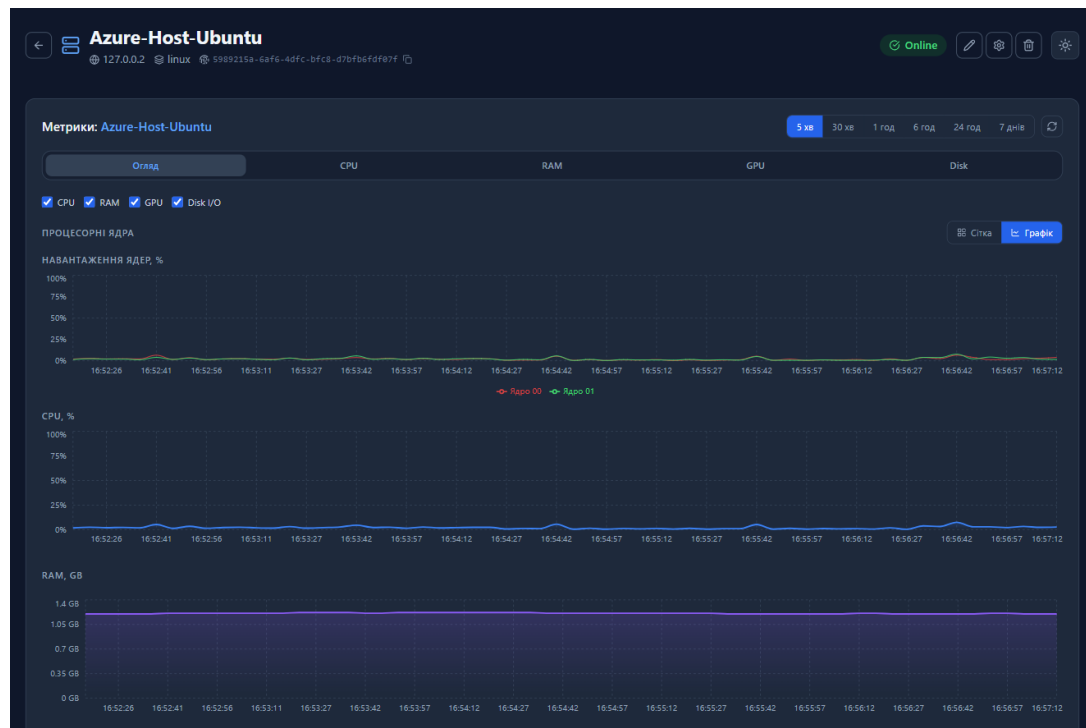


Рис. 3.8. Окрема сторінка детальної інформації про сервер

Управління життєвим циклом підключеного сервера здійснюється через систему спеціалізованих модальних вікон. Інтерфейс редагування дозволяє модифікувати назву вузла, його мережеву адресу та тип операційної системи, апаратно блокуючи надсилання запиту до моменту заповнення всіх обов'язкових полів. Вікно зміни базової інформації про сервер наведено на рисунку (рис. 3.9).

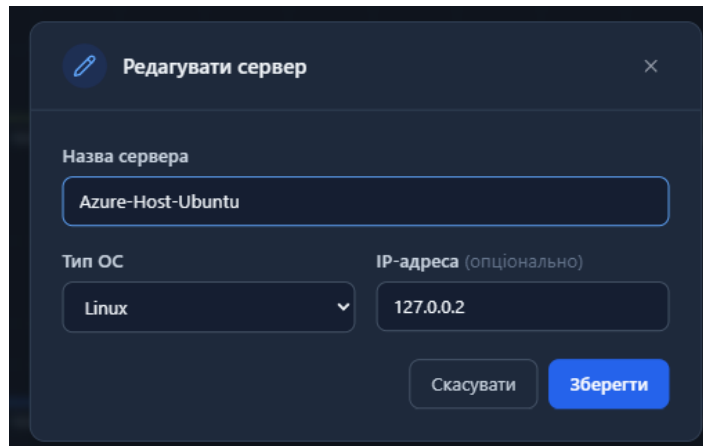


Рис. 3.9. Вікно зміни базової інформації про сервер

Особливу увагу приділено інтерфейсу налаштування порогових значень, який дозволяє гнучко конфігурувати параметри генерації інцидентів для кожної апаратної метрики. Для метрик оперативної та дискової пам'яті клієнтський застосунок автоматично виконує двосторонню конвертацію значень із байтів у гігабайти для зручного сприйняття людиною. Задля уникнення логічних помилок при конфігурації впроваджено алгоритм суворої делікатної валідації. Для більшості метрик (процесор, оперативна пам'ять, графічний прискорювач) система програмно гарантує, що значення попереджувального порогу є строго меншим за критичне. Водночас для метрики вільного дискового простору застосовується інверсна математична валідація: попереджувальний поріг має бути більшим за критичний, оскільки загрозу становить саме зменшення доступного обсягу. У разі порушення цих правил інтерфейс блокує збереження та виводить відповідні вбудовані текстові підказки. Вікно налаштування порогових значень із демонстрацією правил валідації наведено на рисунку (рис. 3.10).

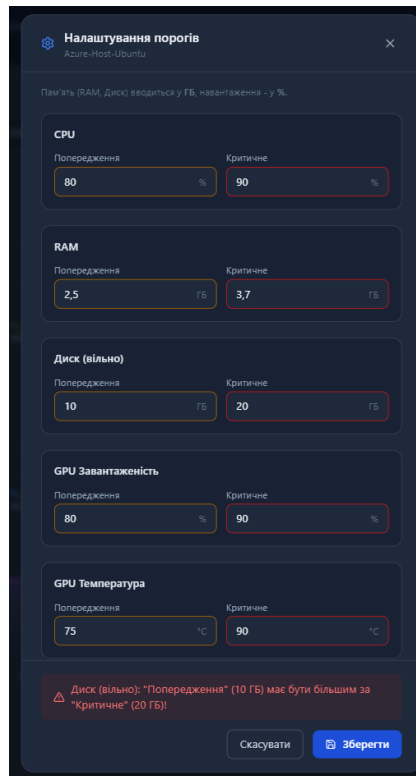


Рис. 3.10. Вікно налаштування порогових значень із відображенням валідації

Процес видалення сервера з інфраструктури захищено від випадкових дій за допомогою універсального модального вікна підтвердження. Інтерфейс чітко інформує адміністратора про безповоротну втрату всієї накопиченої історичної телеметрії та відповідного журналу інцидентів, вимагаючи явного натискання попереджувальної кнопки для ініціації операції безповоротного видалення. Вікно підтвердження видалення сервера наведено на рисунку (рис. 3.11).

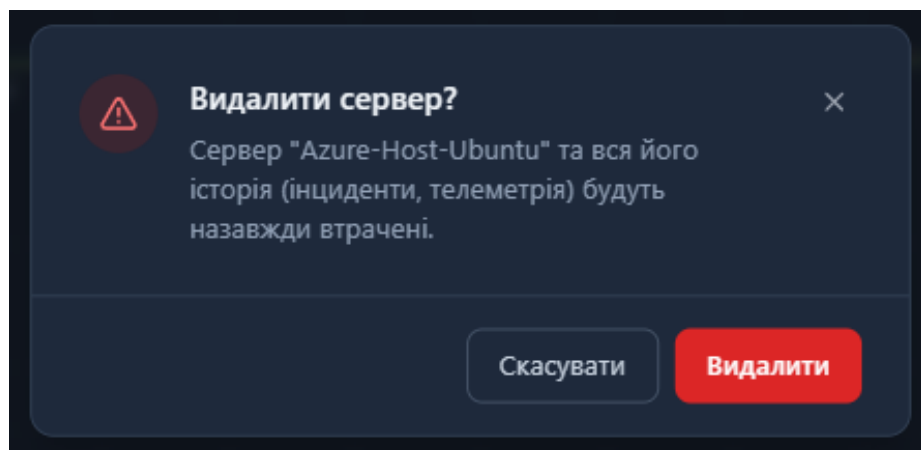


Рис. 3.11. Вікно підтвердження видалення сервера

Процес розширення інфраструктури моніторингу ініціюється через спеціалізоване модальне вікно реєстрації нового сервера, яке стає доступним на головній панелі лише для користувачів із правами адміністратора. Інтерфейс реєстрації спроектовано для роботи у двоетапному режимі. На першому кроці оператор заповнює базову ідентифікаційну форму, яка містить текстове поле назви вузла з функцією автоматичного фокусування, обов'язковий селектор вибору типу операційної системи та опціональне поле мережевої адреси. Додатково інтерфейс превентивно інформує адміністратора про те, що після підтвердження форми серверне ядро автоматично згенерує унікальні облікові дані для підключення агента та встановить базовий набір порогових значень для чотирьох основних метрик. Захисний механізм клієнтської форми апаратно блокує кнопку надсилання запиту до моменту коректного заповнення всіх необхідних параметрів. Вікно створення нового сервера наведено на рисунку (рис. 3.12).

Рис. 3.12. Вікно створення нового сервера

Після успішної обробки запиту серверною підсистемою модальне вікно миттєво перемикається на етап відображення результату. На екрані генеруються поля з унікальним ідентифікатором вузла (UUID) та криптографічним ключем доступу. З огляду на дотримання суворих стандартів інформаційної безпеки,

інтерфейс виводить критичне попередження про те, що згенерований API-ключ демонструється виключно один раз, і його повторний перегляд у системі буде неможливим. Для надійного перенесення цих даних до локальної конфігурації програмного агента впроваджено універсальний інструмент копіювання в буфер обміну. Алгоритм копіювання використовує сучасний програмний інтерфейс браузера (Clipboard API) під час роботи в захищеному контексті, а також має вбудований резервний механізм (fallback) для безперебійного функціонування в умовах локального незахищеного HTTP-з'єднання. Про успішне збереження даних у буфері користувача інформує динамічна зміна відповідної іконки на індикатор підтвердження. Вікно успішного створення з відображенням облікових даних наведено на рисунку (рис. 3.13).

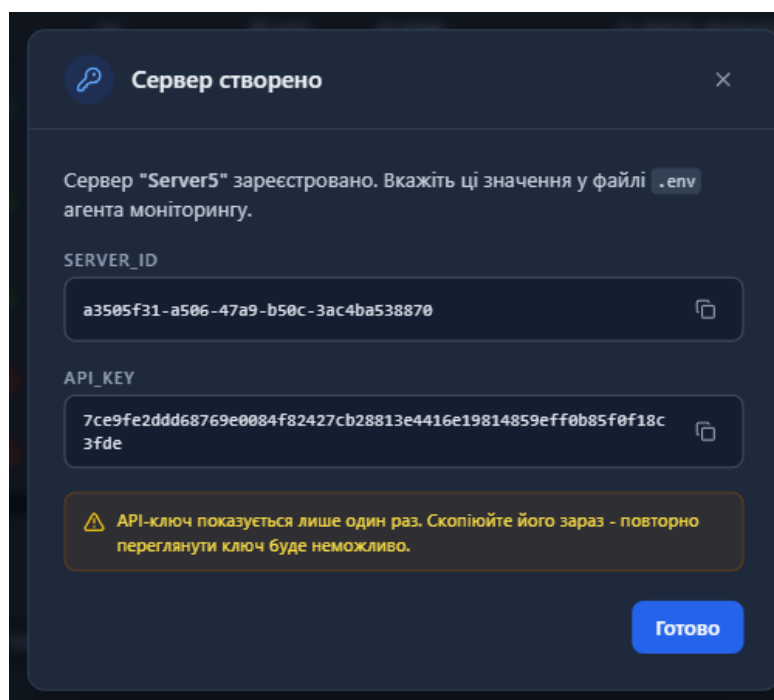


Рис. 3.13. Вікно успішного створення з відображенням згенерованого ідентифікатора та API-ключа

Адміністрування облікових записів та контроль доступу до системи моніторингу здійснюється через окрему підсистему управління користувачами, доступ до якої суворо обмежений рольовою моделлю і надається тільки адміністраторам. Інтерфейс цього розділу представлений інтерактивною таблицею, що відображає перелік зареєстрованих операторів, їхні логіни, повні імена та рівні доступу (адміністратор або спостерігач) із відповідними

кольоровими ідентифікаторами. Для кожного профілю передбачено панель дій, що дозволяє оперативно змінювати пароль або ініціювати видалення запису. З метою захисту від критичних помилок керування алгоритм системи динамічно розпізнає поточну сесію та приховує кнопку видалення для власного облікового запису адміністратора, що додатково дублюється захисним механізмом на рівні серверного бекенду. Сторінку управління обліковими записами користувачів наведено на рисунку (рис. 3.14).

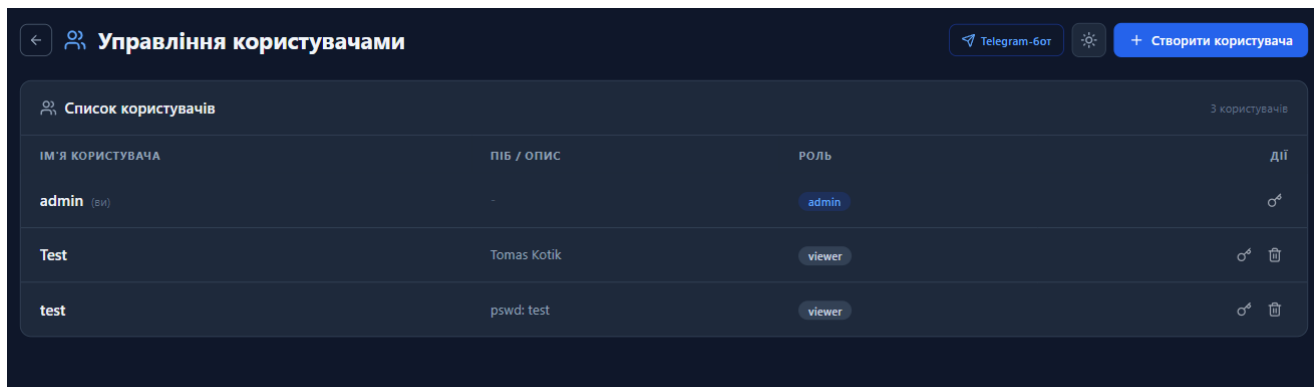


Рис. 3.14. Сторінка управління обліковими записами користувачів

Процес реєстрації нового оператора системи виконується через спеціалізоване модальне вікно. Форма створення вимагає обов'язкового заповнення текстових полів логіна та пароля, а також містить випадний список для призначення відповідної ролі доступу. Клієнтська частина застосунку автоматично обробляє серверні винятки, виводячи вбудовані попередження у разі спроби реєстрації вже зайнятого імені користувача. Вікно створення нового облікового запису наведено на рисунку (рис. 3.15).

Рис. 3.15. Вікно створення нового облікового запису користувача

Для забезпечення гнучкості політики безпеки розроблено окреме вікно зміни облікових даних. Цей інтерфейс дозволяє адміністратору примусово оновлювати криптографічний пароль будь-якого підлеглого профілю без необхідності повного перестворення запису. Після успішної обробки запиту сервером система автоматично інформує оператора повідомленням про успіх та закриває форму. Вікно зміни пароля користувача наведено на рисунку (рис. 3.16).

Рис. 3.16. Вікно зміни пароля користувача

3.3 Тестування роботи системи

Перевірка графічного інтерфейсу та коректності роботи механізмів делікатної валідації здійснювалася за допомогою сучасного фреймворку наскрізного тестування (End-to-End) Playwright [19]. Розроблені тестові сценарії імітують поведінку кінцевого користувача в браузері, забезпечуючи автоматизовану перевірку рендерингу векторних графіків телеметрії, коректності взаємодії з елементами управління (зокрема, вибору параметрів із випадних списків при реєстрації сервера) та перевірки логіки блокування збереження невалідних порогових значень. Використання автоматизованого підходу дозволило підтвердити, що клієнтський застосунок коректно обробляє відповіді від серверного маршрутизатора та динамічно оновлює стан віртуального дерева документів без критичних збоїв у відображенні.

Функціональне тестування алгоритму управління інцидентами проводилося шляхом симуляції аварійної ситуації на цільовому сервері (контрольний приклад). Під час штатної роботи автономного агента було штучно ініційовано стрес-тест, що призвело до максимального завантаження обчислювальних потужностей центрального процесора (100%). Відповідна телеметрія була миттєво зафіксована агентом і передана до центрального сервера. Бекенд-система успішно ідентифікувала перевищення встановленого критичного порогу, заблокувала створення дублікатів записів у базі даних (згідно з алгоритмом відстеження стану) та згенерувала єдиний активний інцидент. Паралельно спрацювала підсистема сповіщень, яка у фоновому режимі надіслала деталізоване повідомлення про аварію до месенджера Telegram. Після примусового завершення стрес-тесту на цільовому вузлі та падіння навантаження до нормальних значень, алгоритм автоматичного розв'язання (auto-resolve) успішно змінив статус інциденту на закритий, зафіксував час відновлення та надіслав підтвердження про стабілізацію роботи. Результати роботи підсистеми сповіщень під час контрольного прикладу наведено на рисунку (рис. 3.17).

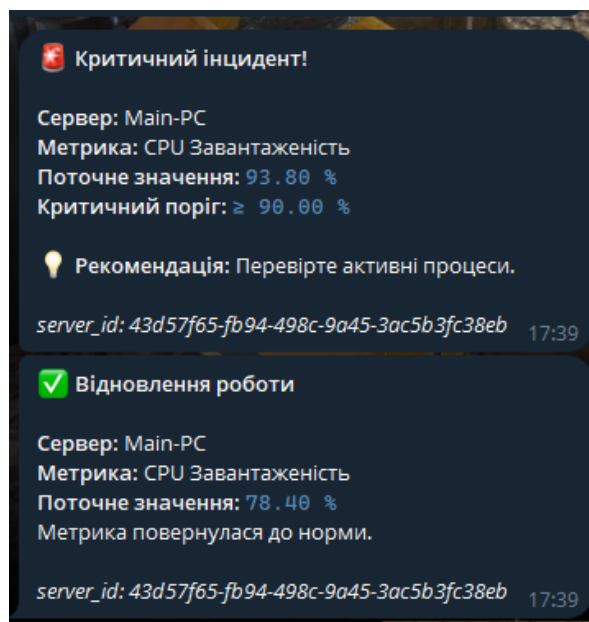


Рис. 3.17. Сповіщення у месенджері Telegram про виникнення та автоматичне закриття інциденту

Тестування механізмів відмовостійкості програмного агента проводилося в умовах штучно згенерованої недоступності центрального сервера. Для імітації обриву зв'язку було тимчасово зупинено контейнер серверного бекенду. Під час виконання штатного алгоритму збору метрик агент своєчасно зафіксував мережеву помилку на рівні SSL-з'єднання, пов'язану з відмовою віддаленого вузла у підключенні, після чого автоматично перенаправив потік телеметрії до локальної бази даних SQLite [29]. У системному журналі автономного скрипта було успішно зафіксовано серію попереджень про відсутність зв'язку із сервером та збереження даних у локальний буфер. Після відновлення роботи серверної частини (запуску контейнера бекенду) агент під час наступної ітерації опитування ідентифікував наявність збережених записів та ініціював їх масове надсилання на спеціалізований ендпоінт. Системні логи підтвердили успішне вивантаження масиву накопичених точок та атомарне очищення локального буфера, що на практиці доводить надійність спроектованої архітектури збереження часових рядів в умовах тривалої нестабільності мережі.

Перевірка нефункціональних вимог та визначення меж продуктивності серверного ядра виконувалися за допомогою спеціалізованого інструменту навантажувального тестування Apache JMeter [2]. З метою імітації реальних умов

експлуатації високонавантаженої інфраструктури було розроблено тестовий сценарій, який відтворював паралельну роботу 100 автономних агентів (віртуальних користувачів). У межах сесії тестування ці потоки виконали інтенсивне надсилання 1000 мережових POST-запитів із корисним навантаженням телеметрії у форматі JSON. Головною метою цього етапу була перевірка стабільності та пропускну здатності зв'язки асинхронного фреймворку FastAPI та синхронного клієнта часових рядів InfluxDB під час масового конкурентного запису даних [14]. Оскільки синхронний клієнт потенційно блокує цикл подій, навантажувальне тестування мало на меті довести, що на поточних обсягах трафіку цей архітектурний компроміс не призводить до критичної деградації продуктивності.

Результати тестування продемонстрували високий рівень стабільності та відмовостійкості серверного ядра: загальний показник успішних оброблених запитів (Success Rate) склав 99,7%. Під час впливу пікового навантаження не було зафіксовано жодної критичної помилки на рівні бізнес-логіки застосунку (помилки класу HTTP 500 Internal Server Error), а також не виявлено ознак деградації або відмов з боку підсистем баз даних PostgreSQL та InfluxDB. Три зафіксовані помилки, що склали 0,3% від загального обсягу тестового трафіку, мали суто мережеву природу (TCP Connection Timeout). Ці тайм-аути виникли на рівні вхідного вебсервера Nginx у момент пікового встановлення великої кількості паралельних TCP-з'єднань [18]. Такий показник втрат є абсолютно прийнятним і знаходиться в межах припустимої статистичної похибки для обраної конфігурації апаратного забезпечення у хмарному середовищі Microsoft Azure [16].

Критично важливо зазначити, що у реальних умовах розгортання та експлуатації програмного комплексу подібні мережеві мікро-відмови повністю нівелюються закладеним на етапі проєктування архітектурним патерном. Вбудований механізм відмовостійкості автономного агента розпізнає тайм-аут з'єднання як тимчасову недоступність сервера і миттєво перенаправляє пакет телеметрії до локальної бази даних SQLite. Після стабілізації мережі агент

виконує повторне масове надсилання (bulk upload) накопичених даних, що алгоритмічно гарантує 100% доставку телеметричної інформації [30]. Таким чином, результати навантажувального тестування підтверджують здатність системи ефективно масштабуватися та функціонувати в умовах високих навантажень без втрати цілісності даних. Зведений звіт результатів навантажувального тестування наведено на рисунку (рис. 3.18).

Summary Report

Name:

Summary Report

Comments:

Write results to file / Read from file

Filename

Label	# Sampl...	Average	Min	Max	Std. Dev.	Error % ↑	Throughput	Sent KB/s...	Received KB...	
HTTP Request	1000	99	25	21057	1149.46	0.30%	34.7/sec	27.68	11.43	
TOTAL	1000	99	25	21057	1149.46	0.30%	34.7/sec	27.68	11.43	

Рис. 3.18. Звіт результатів навантажувального тестування в Apache JMeter (Summary Report)

Висновки до третього розділу

У третьому розділі було виконано комплексний опис практичної реалізації, інтерфейсу та процесу тестування розробленої вебсистеми моніторингу серверного обладнання. На етапі впровадження розроблено алгоритм розгортання серверного ядра у хмарному середовищі Microsoft Azure із використанням інструментарію контейнеризації Docker Compose. Особливу увагу приділено клієнтській частині: для серверів під управлінням операційної системи Windows реалізовано парадигму розгортання з нульовими залежностями (zero-dependency) шляхом компіляції агента в автономний виконуваний файл, що значно спрощує процес підключення нових вузлів інфраструктури та підвищує загальний рівень безпеки.

Спроектовано та реалізовано графічний інтерфейс користувача за моделлю односторінкового застосунку (SPA), який забезпечує зручний моніторинг у режимі реального часу. Головна інформаційна панель об'єднує ключові показники стану серверів, журнал інцидентів та інтерактивні векторні графіки телеметрії. Ергономічність системи підвищено завдяки імплементації алгоритму

кольорового кодування метрик, підтримці динамічного перемикання часових діапазонів та механізму делікатної валідації вводу без блокуючих вікон. На рівні інтерфейсу також успішно інтегровано рольову модель керування доступом (RBAC), яка гарантує, що операції зміни конфігурації, редагування порогових значень та генерації API-ключів доступні тільки обліковим записам із правами адміністратора.

Надійність розроблених рішень підтверджено серією комплексних випробувань. Наскрізне тестування інтерфейсу (E2E) довело коректність роботи клієнтської частини та обробки серверних відповідей. Контрольний приклад із симуляцією критичного навантаження процесора підтвердив правильність функціонування алгоритму автоматичного управління інцидентами, включаючи генерацію подій, захист від дублікатів та миттєве надсилання сповіщень через месенджер Telegram. Навантажувальне тестування підтвердило високу продуктивність серверного ядра, обробивши 1000 запитів від 100 одночасних потоків із показником успішності 99,7% без жодної відмови бази даних. Зафіксовані одиничні мережеві тайм-аути ефективно нівелюються закладеним архітектурним патерном програмного агента, який автоматично буферизує невідправлену телеметрію в локальній базі даних SQLite та виконує масове повторне надсилання після відновлення зв'язку, що гарантує абсолютну цілісність історичних даних.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальну науково-прикладну задачу розробки вебсистеми моніторингу серверного обладнання. Основним результатом дослідження, у повній відповідності до поставленої мети, є створення та успішне впровадження відмовостійкого програмного комплексу, здатного забезпечити безперервний автоматизований контроль за станом апаратного забезпечення. Проведені дослідження дозволили сформувати низку пропозицій, які безпосередньо сприяють підвищенню ефективності функціонування об'єкта дослідження – ІТ-інфраструктури підприємств. Зокрема, запропоновано децентралізовану модель збору телеметрії з використанням автономних програмних агентів та гібридну архітектуру збереження даних, що дозволило усунути «вузькі місця» при пікових мережових навантаженнях та значно оптимізувати процеси обробки масивів часових рядів.

Перелік реалізованих функцій охоплює повний цикл обробки моніторингової інформації. Розроблено автономну підсистему збору телеметрії із вбудованим механізмом локальної буферизації у базі даних SQLite, що гарантує збереження метрик за умов нестабільного з'єднання. Спроектовано поліглотне серверне сховище, де транзакційна база PostgreSQL відповідає за цілісність конфігурацій, а InfluxDB забезпечує високошвидкісний запис показників. Програмно імплементовано алгоритм управління інцидентами з інтеграцією месенджера Telegram, захистом від лавинного дублювання сповіщень та модулем автоматичного розв'язання аварійних ситуацій (Auto-resolve). Створено ергономічний клієнтський вебзастосунок за моделлю Single Page Application, який забезпечує візуалізацію даних у реальному часі через графіки Recharts, підтримує сувору рольову модель доступу (RBAC) та механізми делікатної валідації вводу. Комплексна оцінка ступеня відповідності реалізованих функцій поставленому завданню підтверджує стовідсоткове виконання функціональних та нефункціональних вимог. Програмний продукт успішно пройшов етапи наскрізного тестування інтерфейсу (E2E) та

					ІПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						73

навантажувального тестування, довівши здатність стабільно обробляти великі потоки даних без відмов на рівні бізнес-логіки.

Практична цінність одержаних результатів полягає у створенні повністю працездатного, готового до промислової експлуатації програмного інструменту, впровадження якого суттєво оптимізує роботу системних адміністраторів та DevOps-інженерів. Використання розробленого комплексу дозволяє автоматизувати рутинні процеси виявлення апаратних збоїв, мінімізувати середній час реакції на критичні інциденти (Mean Time To Detect) та гарантувати цілісність накопиченої телеметричної аналітики для подальшого аудиту. Завдяки імплементації архітектури з нульовими залежностями (zero-dependency) для Windows-клієнтів та використанню технологій контейнеризації, розроблена система значно знижує поріг входу та витрати часу на розгортання моніторингу в існуючих інфраструктурах.

Напрямки подальшого наукового дослідження та вдосконалення програмного комплексу лежать у площині впровадження методів штучного інтелекту та розширення інтеграційних можливостей. Перспективним є інтеграція алгоритмів машинного навчання для предиктивного аналізу телеметрії, що дозволить системі завчасно попереджати про можливі апаратні деградації на основі виявлення прихованих патернів у часових рядах (Predictive Maintenance). Також доцільним є вдосконалення архітектури серверного ядра шляхом реалізації інструментів автоматичного горизонтального масштабування (Auto-scaling) при досягненні пікових навантажень. Подальший розвиток системи передбачає розширення спектра підтримуваних протоколів для глибокої інтеграції з корпоративними екосистемами та розробку відкритого програмного інтерфейсу (Public API) для взаємодії зі сторонніми платформами оркестрації.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aiohttp Documentation. URL: <https://docs.aiohttp.org/> (дата звернення: 20.04.2026).
2. Apache JMeter Documentation. URL: <https://jmeter.apache.org/> (дата звернення: 30.05.2026).
3. Bayer R., McCreight E. Organization and Maintenance of Large Ordered Indexes. Acta Informatica. 1972. Vol. 1, No. 3. P. 173-189.
4. Datadog Documentation. Datadog Inc. URL: <https://docs.datadoghq.com/> (дата звернення: 10.04.2026).
5. Date C. J. An Introduction to Database Systems. 8th ed. Boston: Addison-Wesley, 2003. 1008 p.
6. Docker Documentation. Docker Inc. URL: <https://docs.docker.com/> (дата звернення: 11.04.2026).
7. Elmasri R., Navathe S. B. Fundamentals of Database Systems. 7th ed. Boston: Pearson, 2015. 1248 p.
8. FastAPI Documentation. URL: <https://fastapi.tiangolo.com/> (дата звернення: 21.04.2026).
9. Fielding R. T. Architectural Styles and the Design of Network-based Software Architectures : Ph.D. dissertation. Irvine: University of California, 2000. 162 p
10. Fowler M. Patterns of Enterprise Application Architecture. Boston: Addison-Wesley, 2002. 560 p.
11. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Boston: Addison-Wesley, 1994. 416 p.
12. GitLab Documentation. GitLab B.V. URL: <https://docs.gitlab.com/> (дата звернення: 19.04.2026).
13. InfluxDB Documentation. InfluxData. URL: <https://docs.influxdata.com/influxdb/> (дата звернення: 04.04.2026).
14. Kleppmann M. Designing Data-Intensive Applications. Sebastopol: O'Reilly Media, 2017. 616 p.

15. Massie M. L., Chun B. N., Culler D. E. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Parallel Computing. 2004. Vol. 30, No 7. P. 817-840. DOI: 10.1016/j.parco.2004.04.001.
16. Microsoft Azure Documentation. URL: <https://learn.microsoft.com/en-us/azure/> (дата звернення: 02.06.2026).
17. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol: O'Reilly Media, 2021. 616 p.
18. NGINX Documentation. F5, Inc. URL: <https://nginx.org/en/docs/> (дата звернення: 02.06.2026).
19. Playwright Documentation. URL: <https://playwright.dev/> (дата звернення: 30.05.2026).
20. Pelkonen T., Franklin S., Teller J. Gorilla: A Fast, Scalable, In-Memory Time Series Database. Proceedings of the VLDB Endowment. 2015. Vol. 8, No. 12. P. 1816-1827.
21. PostgreSQL Documentation. The PostgreSQL Global Development Group. URL: <https://www.postgresql.org/docs/> (дата звернення: 13.06.2026).
22. Prometheus Documentation. Cloud Native Computing Foundation. URL: <https://prometheus.io/docs/introduction/overview/> (дата звернення: 10.04.2026).
23. Psutil Documentation. URL: <https://psutil.readthedocs.io/> (дата звернення: 14.06.2026).
24. Python Software Foundation. Python 3 Documentation. URL: <https://docs.python.org/3/> (дата звернення: 20.04.2026).
25. React Documentation. Meta Platforms, Inc. URL: <https://react.dev/> (дата звернення: 22.04.2026).
26. Richardson L., Ruby S. RESTful Web Services. Sebastopol: O'Reilly Media, 2007. 448 p.
27. Sadalage P. J., Fowler M. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Boston: Addison-Wesley, 2012. 192 p.
28. Sommerville I. Software Engineering. 10th ed. Boston: Pearson, 2015. 816 p.

29. SQLite Documentation. URL: <https://www.sqlite.org/docs.html> (дата
звернення: 15.05.2026).
30. Tanenbaum A. S., Van Steen M. Distributed Systems: Principles and Paradigms.
2nd ed. Upper Saddle River: Prentice Hall, 2006. 704 p.
31. Vite Documentation. URL: <https://vitejs.dev/guide/> (дата звернення:
22.04.2026).
32. Zabbix Documentation. Zabbix LLC. URL:
<https://www.zabbix.com/documentation/current/en/manual> (дата звернення:
10.04.2026).

ДОДАТКИ

					ПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						78

Контролер прийому телеметрії (FastAPI)

Ендпоінт POST /api/telemetry:

```

@router.post(
    "/telemetry",
    status_code=status.HTTP_201_CREATED,
    summary="Прийом метрик від агента",
    response_description="Підтвердження успішного запису телеметрії",
)
async def receive_telemetry(
    payload: TelemetryCreate,
    db: AsyncSession = Depends(get_db),
    server: Server = Depends(verify_agent),
) -> dict:
    if str(server.id) != str(payload.server_id):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail=(
                f"server_id у запиті ({payload.server_id}) не збігається "
                f"з сервером, якому належить API-ключ ({server.id})."
            ),
        )

    await _write_to_influx(payload)

    pending_alerts = await _check_thresholds(server, payload, db)

    server.last_seen = datetime.now(tz=timezone.utc)
    await db.commit()

    logger.info(
        "Телеметрія отримана: server='%s' cpu=%.1f%% ram=%d bytes",
        server.name,
        payload.cpu_usage,
        payload.ram_usage,
    )

    for alert_text in pending_alerts:
        asyncio.create_task(send_telegram_alert(alert_text))

    recorded_at = payload.timestamp or datetime.now(tz=timezone.utc)
    return {
        "status": "ok",
        "message": f"Телеметрія для сервера '{server.name}' записана успішно.",
        "recorded_at": recorded_at.isoformat(),
        "new_incidents": len(pending_alerts),
    }

Залежність автентифікації агента (verify_agent):
async def verify_agent(
    credentials: HTTPAuthorizationCredentials = Depends(security),
    db: AsyncSession = Depends(get_db),
) -> Server:

```

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	Арк.
						79

```

result = await db.execute(
    select(Server).where(Server.api_key == credentials.credentials)
)
server: Server | None = result.scalar_one_or_none()
if server is None or not server.is_active:
    raise HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Недійсний або неактивний API-ключ.",
        headers={"WWW-Authenticate": "Bearer"},
    )
return server

```

Алгоритм управління інцидентами

Функція перевірки порогів, відкриття та автоматичного закриття інцидентів:

```

async def _check_thresholds(
    server: Server,
    payload: TelemetryCreate,
    db: AsyncSession,
) -> list[str]:
    current_values: dict[str, float] = {
        "cpu_usage": payload.cpu_usage,
        "ram_usage": float(payload.ram_usage),
        "disk_free": float(payload.disk_free),
    }
    if payload.gpu_load is not None:
        current_values["gpu_load"] = payload.gpu_load
    if payload.gpu_temperature is not None:
        current_values["gpu_temperature"] = payload.gpu_temperature
    totals: dict[str, float] = {
        "ram_usage": float(payload.ram_total),
        "disk_free": float(payload.disk_total),
    }
    threshold_rows = await db.execute(
        select(Threshold).where(Threshold.server_id == server.id)
    )
    thresholds: list[Threshold] = list(threshold_rows.scalars().all())
    pending_alerts: list[str] = []
    for threshold in thresholds:
        metric = threshold.metric_name
        current = current_values.get(metric)
        if current is None:
            continue
        open_incident_result = await db.execute(
            select(Incident).where(
                Incident.server_id == server.id,
                Incident.metric_type == metric,
                Incident.status == IncidentStatus.open,
            )
        )
        open_incident: Incident | None = open_incident_result.scalar_one_or_none()
        if _is_critical(metric, current, threshold.critical_value):
            if open_incident is not None:
                logger.debug(
                    "Інцидент для server='%s' metric='%s' вже відкрито - пропускаємо.",
                    server.name,
                    metric,
                )
                continue
            total = totals.get(metric)
            description = (
                f"Перевищено поріг: {translate_metric(metric)} - "
                f"поточне {format_metric_value(metric, current, total)}, "
                f"критичне: {format_metric_value(metric, threshold.critical_value)}"
            )

```

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	Арк.
						81

```

)
new_incident = Incident(
    server_id=server.id,
    metric_type=metric,
    status=IncidentStatus.open,
    description=description,
)
db.add(new_incident)
logger.warning(
    "Новий інцидент: server='%s' metric='%s' value=%.2f threshold=%.2f",
    server.name,
    metric,
    current,
    threshold.critical_value,
)
pending_alerts.append(
    _build_alert_message(
        server_name=server.name,
        server_id=server.id,
        metric_name=metric,
        current_value=current,
        critical_value=threshold.critical_value,
        total_value=total,
    )
)
else:
    if open_incident is None:
        continue
    total = totals.get(metric)
    auto_resolve_line = (
        f"\n[Auto-Resolve]: Метрика {translate_metric(metric)} повернулася до норми: "
        f"{format_metric_value(metric, current, total)}."
    )
    open_incident.status = IncidentStatus.resolved
    open_incident.closed_at = datetime.now(tz=timezone.utc)
    open_incident.description = (open_incident.description or "") + auto_resolve_line
    logger.info(
        "Інцидент закрито (відновлення): server='%s' metric='%s' value=%.2f",
        server.name,
        metric,
        current,
    )
    pending_alerts.append(
        f"✅ <b>Відновлення роботи</b>\n\n"
        f"<b>Сервер:</b> {server.name}\n"
        f"<b>Метрика:</b> {translate_metric(metric)}\n"
        f"<b>Поточне значення:</b> <code>{format_metric_value(metric, current, "
total)}</code>\n"
        f"Метрика повернулася до норми.\n\n"
        f"<i>server_id: {server.id}</i>"
    )
return pending_alerts

```

					ІПЗ.КР.Б – 121 – 26 – ПЗ	Арк.
						82

Допоміжний предикат визначення критичного стану:

```
def _is_critical(metric_name: str, value: float, critical_value: float) -> bool:
```

```
    if metric_name in _BELOW_THRESHOLD_METRICS:
```

```
        return value <= critical_value
```

```
    return value >= critical_value
```

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	Арк.
						83

Гібридна підсистема баз даних (InfluxDB)

Ініціалізація клієнта InfluxDB:

```
def __init__(self) -> None:
    self.url: str = os.getenv("INFLUXDB_URL", "http://localhost:8086")
    self.token: str = os.getenv("INFLUXDB_ADMIN_TOKEN", "")
    self.org: str = os.getenv("INFLUXDB_ORG", "my-org")
    self.bucket: str = os.getenv("INFLUXDB_BUCKET", "server_telemetry")

    self._client: InfluxDBClient = InfluxDBClient(
        url=self.url,
        token=self.token,
        org=self.org,
        timeout=10_000,
    )
```

Запис точки телеметрії (Point) у time-series базу:

```
ts = timestamp or datetime.now(tz=timezone.utc)
```

```
point = (
    Point("server_metrics")
    .tag("server_id", server_id)
    .tag("os_type", os_type)
    .field("cpu_usage", float(cpu_usage))
    .field("ram_usage", int(ram_usage))
    .field("ram_total", int(ram_total))
    .field("disk_free", int(disk_free))
    .field("disk_total", int(disk_total))
    .field("disk_read_speed", float(disk_read_speed))
    .field("disk_write_speed", float(disk_write_speed))
    .field("net_in", int(net_in))
    .field("net_out", int(net_out))
    .time(ts)
)
```

```
for idx, core_val in enumerate(cpu_cores):
    point = point.field(f"cpu_core_{idx}", float(core_val))
```

if uptime is not None:

```
point = point.field("uptime", float(uptime))
```

if gpu_load is not None:

```
point = point.field("gpu_load", float(gpu_load))
```

if gpu_memory_used is not None:

```
point = point.field("gpu_memory_used", int(gpu_memory_used))
```

if gpu_temperature is not None:

```
point = point.field("gpu_temperature", float(gpu_temperature))
```

```
write_api = self._client.write_api(write_options=SYNCHRONOUS)
```

```
write_api.write(bucket=self.bucket, org=self.org, record=point)
```

```
logger.debug(
```

```
    "InfluxDB: записано точки server_metrics для server_id=%s", server_id
```

```
)
```

Основний цикл програмного агента

Збір системних метрик через psutil:

def collect_metrics() -> dict:

global _prev_net_in, _prev_net_out, _prev_disk_read, _prev_disk_write

cpu_usage: float = psutil.cpu_percent(interval=None)

cpu_cores: list[float] = psutil.cpu_percent(percpu=True)

ram = psutil.virtual_memory()

ram_usage: int = ram.used

ram_total: int = ram.total

disk_root = "C:\\\\" if platform.system() == "Windows" else "/"

disk_stat = psutil.disk_usage(disk_root)

disk_free: int = disk_stat.free

disk_total: int = disk_stat.total

disk_io = psutil.disk_io_counters()

if disk_io is not None:

disk_read_delta = max(0, disk_io.read_bytes - _prev_disk_read)

disk_write_delta = max(0, disk_io.write_bytes - _prev_disk_write)

_prev_disk_read = disk_io.read_bytes

_prev_disk_write = disk_io.write_bytes

else:

disk_read_delta = disk_write_delta = 0

disk_read_speed: float = disk_read_delta / POLL_INTERVAL

disk_write_speed: float = disk_write_delta / POLL_INTERVAL

net = psutil.net_io_counters()

net_in: int = max(0, net.bytes_recv - _prev_net_in)

net_out: int = max(0, net.bytes_sent - _prev_net_out)

_prev_net_in = net.bytes_recv

_prev_net_out = net.bytes_sent

uptime: float = time.time() - psutil.boot_time()

gpu = get_gpu_metrics()

return {

"server_id": SERVER_ID,

"os_type": OS_TYPE,

"cpu_usage": cpu_usage,

"cpu_cores": cpu_cores,

"ram_usage": ram_usage,

"ram_total": ram_total,

"disk_free": disk_free,

"disk_total": disk_total,

"disk_read_speed": disk_read_speed,

"disk_write_speed": disk_write_speed,

"net_in": net_in,

"net_out": net_out,

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	Арк.
						85

```

"uptime":      uptime,
"gpu_load":    gpu["gpu_load"]      if gpu else None,
"gpu_memory_used": gpu["gpu_memory_used"] if gpu else None,
"gpu_temperature": gpu["gpu_temperature"] if gpu else None,
"timestamp":   datetime.now(tz=timezone.utc).isoformat(),
}

```

Асинхронна відправка телеметрії з логікою буферизації в SQLite:

```

async def send_telemetry(session: aiohttp.ClientSession, metrics: dict) -> None:
    headers = {"Authorization": f"Bearer {API_KEY}"}
    try:
        buffered_data = get_buffer_without_clearing()

        if not buffered_data:
            async with session.post(TELEMETRY_ENDPOINT, json=metrics, headers=headers) as
response:
                if response.status == 201:
                    _log_telemetry_success(metrics)
                else:
                    body = await response.text()
                    logger.warning("Бекенд повернув %d: %s", response.status, body[:200])
            else:
                all_data = buffered_data + [metrics]
                async with session.post(BULK_TELEMETRY_ENDPOINT, json=all_data,
headers=headers) as response:
                    if response.status == 201:
                        clear_buffer()
                        logger.info(
                            len(all_data),
                        )
                        _log_telemetry_success(metrics)
                    else:
                        body = await response.text()
                        logger.warning(
                            "Масова відправка повернула %d: %s", response.status, body[:200]
                        )

    except aiohttp.ClientError as exc:
        logger.warning(
            "Немає зв'язку з сервером, зберігаю в локальний буфер. Помилка: %s", exc
        )
        save_to_buffer(metrics)
    except Exception as exc:
        logger.warning("Непередбачена помилка при відправці телеметрії: %s", exc)

```

Функції локального SQLite-буфера та головний цикл:

```

def init_db() -> None:
    with sqlite3.connect(DB_PATH) as conn:
        conn.execute(
            "CREATE TABLE IF NOT EXISTS buffer "
            "(id INTEGER PRIMARY KEY AUTOINCREMENT, payload TEXT)"
        )
        conn.commit()

```

```

def save_to_buffer(payload: dict) -> None:
    """Зберігає один словник метрик у локальний буфер як JSON-рядок."""
    with sqlite3.connect(DB_PATH) as conn:
        conn.execute("INSERT INTO buffer (payload) VALUES (?)", (json.dumps(payload),))
        conn.commit()

def get_buffer_without_clearing() -> list[dict]:
    """Повертає всі збережені записи з буфера без їх видалення."""
    with sqlite3.connect(DB_PATH) as conn:
        rows = conn.execute("SELECT payload FROM buffer").fetchall()
    return [json.loads(row[0]) for row in rows]

def clear_buffer() -> None:
    """Видаляє всі записи з локального буфера після успішної масової відправки."""
    with sqlite3.connect(DB_PATH) as conn:
        conn.execute("DELETE FROM buffer")
        conn.commit()

def get_and_clear_buffer() -> list[dict]:
    """Атомарно читає всі записи з буфера та видаляє їх. Повертає список словників."""
    with sqlite3.connect(DB_PATH) as conn:
        rows = conn.execute("SELECT payload FROM buffer").fetchall()
        conn.execute("DELETE FROM buffer")
        conn.commit()
    return [json.loads(row[0]) for row in rows]

async def main() -> None:
    if not SERVER_ID:
        logger.error(
            "Змінна SERVER_ID не задана. "
            "Додайте SERVER_ID=<uuid> у файл agent/.env та перезапустіть агента."
        )
        raise ValueError("SERVER_ID не задано.")

    if not API_KEY:
        logger.error(
            "Змінна API_KEY не задана. "
            "Додайте API_KEY=<ключ з таблиці servers.api_key> у файл agent/.env та
перезапустіть агента."
        )
        raise ValueError("API_KEY не задано.")

    logger.info("=" * 55)
    logger.info(" Агента моніторингу запущено")
    logger.info(" Бекенд : %s", BACKEND_URL)
    logger.info(" Server ID: %s", SERVER_ID)
    logger.info(" ОС : %s", OS_TYPE)
    logger.info(" Інтервал : %d сек.", POLL_INTERVAL)
    logger.info("=" * 55)

```

```

connector = aiohttp.TCPConnector(force_close=True)
async with aiohttp.ClientSession(connector=connector) as session:
    while True:
        metrics = collect_metrics()
        await send_telemetry(session, metrics)
        await asyncio.sleep(POLL_INTERVAL)

```

Клієнтський компонент візуалізації (React)

Головний компонент MetricChart – завантаження, тихий polling та нормалізація часових рядів:

```
export default function MetricChart({ serverId, serverName, latestTelemetry, thresholds = [] }) {
  const [timeframe, setTimeframe] = useState('5m')
  const [activeTab, setActiveTab] = useState('overview')
  const [points, setPoints] = useState([])
  const [loading, setLoading] = useState(false)
  const [error, setError] = useState(null)
  const [overviewVis, setOverviewVis] = useState({ cpu: true, ram: true, gpu: true, disk: true })

  const { isDark } = useDarkMode()

  const fetchHistory = (silent = false) => {
    if (!silent) {
      setLoading(true)
      setError(null)
    }
    api
      .get(`/telemetry/${serverId}/history`, { params: { timeframe } })
      .then((res) => {
        const normalized = res.data.map((p) => {
          const coreFields = {}
          if (Array.isArray(p.cpu_cores)) {
            p.cpu_cores.forEach((val, i) => {
              coreFields[`core_${i}`] = val != null ? parseFloat(val.toFixed(1)) : null
            })
          }
          return {
            rawTime: p.time,
            cpu_usage: p.cpu_usage != null ? parseFloat(p.cpu_usage.toFixed(1)) : null,
            ram_gb: bytesToGb(p.ram_usage),
            gpu_load: p.gpu_load != null ? parseFloat(p.gpu_load.toFixed(1)) : null,
            gpu_temperature: p.gpu_temperature != null ? parseFloat(p.gpu_temperature.toFixed(1)) :
            null,
            disk_read_mb: bytesToMBps(p.disk_read_speed),
            disk_write_mb: bytesToMBps(p.disk_write_speed),
            ...coreFields,
          }
        })
        setPoints(normalized)
      })
      .catch((err) => {
        console.error('Помилка завантаження історії:', err)
        if (!silent) setError('Не вдалося завантажити дані графіка.')
      })
      .finally(() => { if (!silent) setLoading(false) })
  }

  useEffect(() => {
    if (!serverId) return

```

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	Арк.
						89

```
fetchHistory(false)
```

```
const intervalId = setInterval(() => fetchHistory(true), getPollInterval(timeframe))
```

```
return () => clearInterval(intervalId)  
}, [serverId, timeframe])
```

Приклад побудови графіка засобами Recharts:

```
function ChartRam({ points, height = 200, sharedAxisProps, sharedXAxisProps,  
sharedTooltipProps, gridColor }) {  
  return (  
    <  
      <SectionLabel>RAM, GB</SectionLabel>  
      <ResponsiveContainer width="100%" height={height}>  
        <AreaChart data={points} margin={{ top: 4, right: 8, bottom: 0, left: 0 }}>  
          <defs>  
            <linearGradient id="ramGrad" x1="0" y1="0" x2="0" y2="1">  
              <stop offset="5%" stopColor="#8b5cf6" stopOpacity={0.3} />  
              <stop offset="95%" stopColor="#8b5cf6" stopOpacity={0} />  
            </linearGradient>  
          </defs>  
          <CartesianGrid strokeDasharray="3 3" stroke={gridColor} />  
          <XAxis {...sharedXAxisProps} />  
          <YAxis tickFormatter={(v) => `${v} GB`} {...sharedAxisProps} width={56} />  
          <Tooltip  
            formatter={(v) => (v !== null ? [`${v.toFixed(2)} GB`, 'RAM'] : [null, 'RAM'])}  
            {...sharedTooltipProps}  
          />  
          <Area type="monotone" dataKey="ram_gb" name="RAM"  
            stroke="#8b5cf6" strokeWidth={2} fill="url(#ramGrad)" dot={false} connectNulls />  
        </AreaChart>  
      </ResponsiveContainer>  
    </>  
  )  
}
```

Сторінка сервера ServerPage, що інтегрує компонент графіків:

```
{!loading && !error && server && (  
  <div className="grid grid-cols-1 gap-6 lg:grid-cols-2">  
    <div className="lg:col-span-2">  
      <MetricChart  
        serverId={server.id}  
        serverName={server.name}  
        latestTelemetry={latestTelemetry}  
        thresholds={server.thresholds ?? []}  
      />  
    </div>  
  </div>  
)}  
))
```

Конфігурація контейнеризації

services:

PostgreSQL

db:

image: postgres:15

restart: unless-stopped

environment:

POSTGRES_USER: \${POSTGRES_USER}

POSTGRES_PASSWORD: \${POSTGRES_PASSWORD}

POSTGRES_DB: \${POSTGRES_DB}

volumes:

- postgres_data:/var/lib/postgresql/data

healthcheck:

test: ["CMD-SHELL", "pg_isready -U \${POSTGRES_USER} -d \${POSTGRES_DB}"]

interval: 10s

timeout: 5s

retries: 5

InfluxDB

influxdb:

image: influxdb:2.7

restart: unless-stopped

ports:

- "8086:8086"

environment:

DOCKER_INFLUXDB_INIT_MODE: setup

DOCKER_INFLUXDB_INIT_USERNAME: \${INFLUXDB_ADMIN_USERNAME}

DOCKER_INFLUXDB_INIT_PASSWORD: \${INFLUXDB_ADMIN_PASSWORD}

DOCKER_INFLUXDB_INIT_ORG: \${INFLUXDB_ORG}

DOCKER_INFLUXDB_INIT_BUCKET: \${INFLUXDB_BUCKET}

DOCKER_INFLUXDB_INIT_ADMIN_TOKEN: \${INFLUXDB_ADMIN_TOKEN}

volumes:

- influxdb_data:/var/lib/influxdb2

healthcheck:

test: ["CMD", "influx", "ping"]

interval: 15s

timeout: 5s

retries: 5

FastAPI

backend:

build: ./backend

restart: unless-stopped

depends_on:

db:

condition: service_healthy

influxdb:

condition: service_healthy

environment:

POSTGRES_HOST: db

					ІІЗ.КР.Б – 121 – 26 – ІІЗ	Арк.
						91

```

POSTGRES_PORT: 5432
POSTGRES_USER: ${POSTGRES_USER}
POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
POSTGRES_DB: ${POSTGRES_DB}
INFLUXDB_URL: http://influxdb:8086
INFLUXDB_ADMIN_TOKEN: ${INFLUXDB_ADMIN_TOKEN}
INFLUXDB_ORG: ${INFLUXDB_ORG}
INFLUXDB_BUCKET: ${INFLUXDB_BUCKET}
# JWT
JWT_SECRET: ${JWT_SECRET}
TELEGRAM_BOT_TOKEN: ${TELEGRAM_BOT_TOKEN}
TELEGRAM_CHAT_ID: ${TELEGRAM_CHAT_ID}

```

```

frontend:
  build: ./frontend
  restart: unless-stopped
  ports:
    - "80:80"
  depends_on:
    - backend

```

```

volumes:
  postgres_data:
  influxdb_data:

```

Конвеєр автоматизованого розгортання CI/CD

stages:

- deploy

deploy_local_runner:

stage: deploy

script:

- echo "Розгортання проєкту через локальний Azure Runner..."
- mkdir -p ~/monitoring-app
- rsync -av --exclude='.git' --exclude='buffer.db' ./ ~/monitoring-app/
- cd ~/monitoring-app
- docker-compose down || true
- docker-compose up --build -d
- docker image prune -f

only:

- master