Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра інженерії програмного забезпечення

КВАЛІФІКАЦІЙНА РОБОТА

ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

РОЗРОБКА ТА ДОСЛІДЖЕННЯ СИСТЕМИ АВТОМАТИЧНОГО МОНІТОРИНГУ СПРАВНОСТІ МЕРЕЖЕВОГО ОБЛАДНАННЯ У ЛОКАЛЬНІЙ МЕРЕЖІ

DEVELOPMENT AND RESEARCH OF A SYSTEM FOR AUTOMATIC MONITORING OF THE SERVICEABILITY OF NETWORK EQUIPMENT IN A LOCAL NETWORK

спеціальність 121 «Інженерія програмного забезпечення»
освітня програма «Інженерія програмного забезпечення»

Виконав: здобувач вищої освіти
групи ІПЗм-21
Шмаровоз С. В.
Керівник:
д.т.н., професор
Андрущак І. Є.

Кваліфікаційну роботу
допущено до захисту
«___» _____ 20__ р.
Гарант освітньої програми:
к.т.н., доцент Суринович О. М.
_____

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *інженерії програмного забезпечення*
Ступінь вищої освіти *магістр*
Галузь знань: *12 «Інформаційні технології»*
Спеціальність: *121 «Інженерія програмного забезпечення»*
Освітня програма: «*Інженерія програмного забезпечення»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____
«___» _____ 202__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА

ДРУГОГО (МАГІСТЕРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Шмаровозу Станіславу Васильовичу

1. Тема кваліфікаційної роботи: Розробка та дослідження системи автоматичного моніторингу справності мережевого обладнання у локальній мережі
Керівник роботи: Андрущак Ігор Євгенович, д.т.н., професор

затверджені наказом закладу вищої освіти від «29» березня 2025року № 190/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи: 04 грудня 2025 р.

3. Вихідні дані до роботи *технічне та програмне забезпечення ЕОМ, вимоги до розробки програмного забезпечення, ергономічні вимоги до функціонування програмного засобу.*

4. Зміст розрахунково-пояснювальної записки: *аналіз проблематики автоматичного моніторингу мережевого обладнання та вибір методів дослідження, обґрунтування технологій та практичну реалізацію системи моніторингу на основі асинхронної архітектури, експериментальне дослідження результативності розробленого програмного забезпечення.*

5. Перелік графічного матеріалу: 13 рисунків, 5 таблиць, 8 лістингів коду

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Андрущак І. Є.*		
Теоретичне дослідження та практична реалізація	*Андрущак І. Є.*		
Експериментальне дослідження системи	*Андрущак І. Є.*		
Нормоконтроль	*Повстяна Ю. С.*		
Гарант ОП	*Андрущак І. Є.*		
Показник запозичень тексту	___%		
Академічна доброчесність	*Андрущак І.Є*		

7. Дата видачі завдання *«02 квітня 2025 р.»*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи магістра	Строк виконання етапів роботи	Примітка
1	Провести огляд літературних джерел по темі кваліфікаційної роботи	02.05.2025	
2	Провести аналіз загальної проблеми і вибір напрямків дослідження	24.09.2025	
3	Розробити функціональну модель та архітектуру системи	01.11.2025	
4	Описати засоби розробки об'єкта проектування	19.11.2025	
5	Практична реалізація об'єкта проектування	26.11.2025	
6	Розробити методику для проведення експерименту	05.11.2025	
7	Провести аналіз результатів експерименту	15.11.2025	
8	Здача чистового варіанту кваліфікаційної роботи на кафедру	04.12.2025	

Здобувач вищої освіти _____ Шмаровоз С. В.

Керівник кваліфікаційної роботи _____ Андрущак І. Є

АНОТАЦІЯ

Шмаровоз С. В. Розробка та дослідження системи автоматичного моніторингу справності мережевого обладнання у локальній мережі. Рукопис. Кваліфікаційна робота магістра ОП «Інженерія програмного забезпечення» спеціальності 121 Інженерія програмного забезпечення. Луцький національний технічний університет. Луцьк, 2025. Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел (згідно структури кваліфікаційної роботи, затвердженої кафедрою).

Кваліфікаційна робота магістра присвячена розробці та дослідженню системи автоматичного моніторингу справності мережевого обладнання у локальній мережі.

Основна частина містить детальний огляд проблематики традиційних методів мережевого опитування, аналіз світових тенденцій Observability та AIOps, а також обґрунтування вибору високопродуктивної асинхронної архітектури. Детально описано всі етапи розробки: проектування гнучкої моделі даних з використанням гібридного сховища часових рядів, реалізацію асинхронного ядра моніторингу на базі сучасних інструментів Python, а також створення інтерактивного веб-дашборду.

Результатом розробки є повністю контейнеризований програмний комплекс, здатний забезпечити багаторазове прискорення циклу моніторингу порівняно з традиційними блокуючими методами. Проведено експериментальне дослідження, що підтвердило високу масштабованість системи та надійність функціоналу придушення хибних тривог.

Результати розробки є загальними та можуть бути використані для впровадження економічно вигідних проактивних рішень моніторингу у локальних мережах середнього та малого бізнесу.

Ключові слова: мережевий моніторинг, асинхронність, Python, веб-додаток, SNMP, часові ряди, придушення тривог, контейнеризація.

ABSTRACT

Shmarovoz S. V. Development and Research of a System for Automatic Monitoring of the Serviceability of Network Equipment in a Local Network. Manuscript. Master's Qualification Thesis of the Educational Program "Software Engineering" specialty 121 Software Engineering. Lutsk National Technical University. Lutsk, 2025.

The Master's Qualification Thesis is dedicated to the development and research of an automated system for monitoring the health of network equipment in a local network.

The main part contains a detailed overview of the problems associated with traditional network polling methods, an analysis of global trends in Observability and AIOps, and the justification for choosing a high-performance asynchronous architecture. All development stages are described in detail: designing a flexible data model using a hybrid time-series storage, implementing the asynchronous monitoring core based on modern Python tools, and creating an interactive web dashboard.

The result of the development is a fully containerized software complex capable of providing a manifold acceleration of the monitoring cycle compared to traditional blocking methods. An experimental study was conducted, which confirmed the high scalability of the system and the reliability of the false alarm suppression functionality.

The results of the development are general and can be used for implementing cost-effective proactive monitoring solutions in local networks of small and medium-sized businesses.

Keywords: network monitoring, asynchronous, Python, web application, SNMP, time series, alarm suppression, containerization.

ЗМІСТ

ВСТУП

Сучасний розвиток інформаційних технологій характеризується тотальною цифровізацією бізнес-процесів та переходом до архітектури розподілених систем, що функціонують на базі локальних обчислювальних мереж (LAN). Критична залежність операційної діяльності будь-якої організації від доступності та швидкодії мережевої інфраструктури зробила поняття справності мережевого обладнання (комутаторів, маршрутизаторів, брандмауерів) не просто технічним, а стратегічним бізнес-показником.

Оцінка сучасного стану проблеми. На сьогодні практично розв'язані задачі з моніторингу мереж переважно базуються на традиційному реактивному підході (протокол SNMPv1/v2c), де система моніторингу періодично опитує пристрої. Однак, зростання обсягів мережевого трафіку, збільшення кількості пристроїв (IoT) та потреба у сервісах реального часу (VoIP, відеоконференції) виявили суттєві прогалини у знаннях та існуючих рішеннях. Основними проблемами є низька продуктивність синхронного опитування, що унеможливлює моніторинг у реальному часі. Також вагомою є відсутність єдиного простого інструменту, який поєднував би класичний SNMP та сучасну стрімінгову телеметрію. А передові AIOps платформи мають високу вартість.

Світові тенденції вирішення поставлених задач свідчать про перехід від реактивного моніторингу до проактивної спостережуваності (Observability) та впровадження концепції AIOps (AI for IT Operations) [1]. Тенденції включають використання асинхронних протоколів телеметрії (gNMI, RESTCONF) для високошвидкісного збору даних та гібридних time-series баз даних (TimescaleDB) для ефективного зберігання великих обсягів метрик. Активно розвиваються модульні та контейнеризовані рішення (Docker, Prometheus + Grafana, FastAPI) для забезпечення гнучкості та простоти розгортання [2].

Актуальність теми зумовлена необхідністю забезпечення операційної стійкості бізнесу в умовах, коли година простою мережі може коштувати великим підприємствам сотні тисяч доларів [3]. Існуючі системи є або надто складними

та дорогими для малого та середнього бізнесу, або застарілими та неефективними для сучасних високошвидкісних мереж. Таким чином, розробка гнучкого, високопродуктивного та економічно вигідного рішення для автоматичного моніторингу справності мережевого обладнання, яке інтегрує сучасні асинхронні підходи, є надзвичайно актуальною задачею для IT-індустрії.

Метою кваліфікаційної роботи є розробка та дослідження системи автоматичного моніторингу справності мережевого обладнання, яка поєднує високу продуктивність асинхронного збору даних з гнучкістю налаштувань та надійним механізмом сповіщень.

Завдання на роботу:

– проаналізувати та обґрунтувати архітектурний та технологічний стек;

– розробити модель даних з оптимізованим сховищем для часових рядів;

– спроектувати та реалізувати високопродуктивне асинхронне ядро опитування;

– реалізувати механізми інтелектуальної обробки сповіщень;

– створити RESTful API для керування системою з безпечною автентифікацією;

– розробити інтерактивний веб-дашборд для візуалізації метрик у реальному часі;

– забезпечити високу переносимість системи шляхом повної контейнеризації;

– провести експериментальне дослідження результативності розробленої системи.

Об’єкт дослідження – це процеси моніторингу справності та продуктивності мережевого обладнання у локальній обчислювальній мережі.

Предметом дослідження є методи та інструменти розробки програмного забезпечення для автоматизованого збору, аналізу та візуалізації метрик мережевого обладнання.

Наукова новизна одержаних результатів – запропоновано архітектуру програмного комплексу моніторингу, яка базується на асинхронному

«Python-воркері» («asyncio.gather») для паралельного SNMP-опитування та використанні гібридної бази даних TimescaleDB, що дозволяє досягти багаторазового скорочення часу циклу моніторингу порівняно з традиційними синхронними рішеннями. Також буде удосконалено механізм обробки сповіщень шляхом реалізації комбінованого алгоритму «Edge Triggering» та логіки «тихих годин» (DND).

Практична цінність роботи полягає у створенні повністю функціонального, контейнеризованого (Docker) програмного комплексу, який може бути негайно впроваджений у практику експлуатації локальних мереж. Система надає адміністраторам інструмент проактивного моніторингу, що значно знижує ризики фінансових та репутаційних втрат від мережевих збоїв. Отримані результати можуть бути використані як навчальний матеріал для підготовки фахівців у сфері мережевих технологій та AIOps.

Апробація результатів дослідження. Результати аналізу для кваліфікаційної роботи були сформовані в статті та опубліковані в науковому журналі «Комп'ютерно-інтегровані технології: освіта, наука, виробництво». Луцьк: Луцький НТУ, 2025. Вип. № 59 [4] та Вип. №61 [5].

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМАТИКИ ЗА ТЕМОЮ РОБОТИ ТА ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1. Огляд і аналіз предметної області проблеми, результатів існуючих теоретичних та експериментальних досліджень

В умовах тотальної цифровізації та переходу до гібридних моделей роботи, роль локальних обчислювальних мереж (LAN) зазнала фундаментальної трансформації. Сучасна LAN – це не просто засіб з'єднання комп'ютерів та принтерів, а стратегічна інфраструктура, що є основою для функціонування критичних бізнес-сервісів. Зростання кількості пристроїв Інтернету речей (IoT) у виробництві та логістиці, широке впровадження хмарних сервісів (SaaS, IaaS) та постійна потреба у сервісах реального часу, таких як IP-телефонія та відеоконференцзв'язок, висувають безпрецедентні вимоги до стабільності, пропускної здатності та безпеки мережі.

Основними активними компонентами сучасних LAN залишаються комутатори (switches), що формують внутрішню структуру мережі, маршрутизатори (routers), які забезпечують зв'язок із зовнішнім світом, та брандмауери (firewalls), що виконують роль цифрових охоронців. Проте функціонал цих пристроїв значно розширився. Сучасні комутатори не лише з'єднують пристрої, але й живлять їх за технологією «Power over Ethernet» (PoE), підтримують складну сегментацію трафіку за допомогою VLAN та забезпечують гігабітні швидкості передачі даних. Маршрутизатори еволюціонували до інтелектуальних пристроїв, що оптимізують доступ до хмарних платформ за технологіями SD-WAN. Брандмауери нового покоління (NGFW) аналізують трафік на рівні додатків, інтегруючи в себе системи запобігання вторгненням (IPS) та глибокий аналіз пакетів (DPI).

Від злагодженої та передбачуваної роботи кожного з цих елементів напряму залежить не лише продуктивність персоналу, але й операційна стійкість та конкурентоспроможність усієї організації. Численні галузеві дослідження

кількісно підтверджують цей зв'язок, демонструючи руйнівні наслідки мережевих збоїв у кількох ключових аспектах.

По-перше, це прямі фінансові втрати. Згідно з аналітичними звітами Information Technology Intelligence Consulting (ITIC), вартість однієї години простою для 91 % великих підприємств перевищує 300 000 доларів, а для третини з них сягає понад 1 мільйон доларів [3]. Навіть у сегменті малого та середнього бізнесу ця цифра може становити десятки тисяч доларів, що включає втрачений дохід, штрафи за порушення угод про рівень обслуговування (SLA) та витрати на аварійне відновлення.

По-друге, це параліч операційної діяльності та втрата продуктивності. Збій у мережі миттєво зупиняє ключові бізнес-процеси: співробітники втрачають доступ до CRM-систем, баз даних та хмарних сервісів, логістичні ланцюги розриваються, а на виробництвах можуть зупинятися цілі конвеєрні лінії. Важливо зазначити, що загальний час втрат продуктивності завжди перевищує тривалість самого збою через час, необхідний на відновлення робочих процесів та синхронізацію всіх систем.

По-третє, це довгострокова репутаційна шкода. У сучасному бізнес-середовищі, де очікується доступність сервісів у режимі 24/7, будь-який помітний збій миттєво стає відомим і призводить до втрати довіри клієнтів. Це, в свою чергу, стимулює їх перехід до більш надійних конкурентів та вимагає від компанії значних інвестицій у відновлення пошкодженого іміджу бренду.

По-четверте, це виникнення вразливостей у системі безпеки. Під час збою мережі часто виходять з ладу й системи безпеки, що моніторять трафік. Це створює «сліпі зони», якими можуть скористатися зловмисники для проведення атак. Крім того, поспішні дії під час відновлювальних робіт можуть призвести до помилок у конфігурації, що створюють нові вектори для атак.

Таким чином, дані про наслідки збоїв підкреслюють, що надійність мережевої інфраструктури є не просто технічним, а критично важливим бізнес-показником, який напряму впливає на фінансову стабільність, операційну ефективність та ринкові позиції компанії.

Поняття «справність» мережевого обладнання еволюціонувало і тепер розглядається через призму кінцевого користувацького досвіду (End-User Experience). Воно включає комплекс динамічних параметрів, що безпосередньо впливають на якість роботи сервісів:

– доступність (Availability) – гарантований час безперебійної роботи (uptime) пристрою та його інтерфейсів;

– продуктивність (Performance) – показники завантаженості CPU/RAM, відсоток відкинутих пакетів (packet drop), затримка (latency) та джиттер (jitter), які є критичними для VoIP та відео;

– надійність (Reliability) – відсутність апаратних помилок на портах (CRC errors), стабільність операційної системи пристрою, мінімізація часу на відновлення після збою (MTTR – Mean Time to Recovery);

– стан середовища (Environmental status) – температурні показники, стан вентиляторів та блоків живлення, аномалії в енергоспоживанні, що можуть свідчити про майбутній збій.

Відсутність превентивного контролю цих метрик веде до неминучих деградацій сервісу або повних відмов. Традиційний реактивний підхід («зламалося – ремонтуємо») є абсолютно неприйнятним в сучасних умовах [6]. Парадигма управління IT-інфраструктурою змістилася в бік проактивного моніторингу та концепції AIOps (AI for IT Operations) [1]. Традиційні системи моніторингу просто збирають дані та показують їх. AIOps йде набагато далі, проходячи три ключові етапи.

Платформа AIOps збирає величезні обсяги різноманітних даних з усіх можливих джерел:

– метрики з серверів та мережевого обладнання (завантаженість CPU, пам'ять);

– логи з додатків та операційних систем (текстові записи про події);

– трасування запитів (відстеження шляху користувацького запиту через різні сервіси).

Штучний інтелект починає виконувати свої дії:

– виявлення аномалій – алгоритм помічає нетипову поведінку, наприклад, різкий стрибок завантаженості диска о 3 годині ночі, хоча зазвичай у цей час навантаження нульове;

– кореляція подій – система аналізує тисячі подій, що відбуваються одночасно, і розуміє, що збій у базі даних, скарги користувачів у техпідтримку та помилки в логах веб-сервера – це все наслідки однієї першопричини: несправності мережевого комутатора. Це дозволяє уникнути «шуму» із сотень не пов'язаних сповіщень;

– прогнозування – аналізуючи історичні дані, система може передбачити майбутні проблеми. Наприклад: «Місце на цьому сервері закінчиться приблизно через 3 дні».

На основі аналізу AIOps-платформа діє автоматично:

– створення інциденту – автоматично створює завдання в системі (наприклад, Jira) з детальним описом проблеми та її ймовірної причини;

– запуск скриптів – може виконати заздалегідь підготовлений сценарій для вирішення проблеми – перезавантажити сервіс, очистити кеш або переключити трафік на резервний канал;

– надання рекомендацій – пропонує інженеру найкращий спосіб вирішення проблеми на основі попередніх успішних кейсів.

Для реалізації завдань моніторингу використовуються як традиційні, так і новітні протоколи та технології.

Базовим протоколом для перевірки доступності залишається ICMP, проте його діагностична цінність вкрай обмежена.

SNMP [7] (Simple Network Management Protocol) залишається фундаментальним галузевим стандартом для збору широкого спектра метрик з мережевого обладнання. Протокол еволюціонував через три основні версії, кожна з яких вирішувала недоліки попередньої, особливо в частині безпеки та ефективності.

SNMPv1. Перша версія протоколу, що заклала основи архітектури «менеджер-агент». Її головною перевагою була простота, що сприяло швидкому

поширенню. Однак вона мала критичний недолік у системі безпеки, яка базувалася на так званих «community strings» (рядках спільноти). Це, по суті, паролі, що передавалися у відкритому, незашифрованому вигляді разом із кожним запитом. Будь-хто, маючи змогу перехопити мережевий трафік, міг легко прочитати цей рядок і отримати повний доступ до управління пристроєм. Це робить SNMPv1 абсолютно неприйнятним для використання в сучасних мережах.

SNMPv2c. Ця версія стала значним кроком уперед з точки зору продуктивності. Ключовим нововведенням стала операція GETBULK, яка дозволила SNMP-менеджеру отримувати великий обсяг даних за один запит, суттєво зменшуючи навантаження на мережу порівняно з послідовними GETNEXT запитами у v1. Також були розширені типи даних. Однак літера «c» у назві означає «community», що вказує на те, що ця версія успадкувала ту саму вразливу модель безпеки на основі community strings, що й SNMPv1. Через свою простоту та підвищену ефективність SNMPv2c досі є найпоширенішою версією, але її використання також не рекомендується в середовищах, де безпека є пріоритетом.

SNMPv3. Третя версія стала революційною, оскільки була повністю перероблена з акцентом на безпеку. Вона ввела модульну архітектуру та User-based Security Model (USM) – модель безпеки на основі користувачів. Замість одного незахищеного рядка для всього пристрою, SNMPv3 оперує обліковими записами користувачів, для кожного з яких можна налаштувати свій рівень безпеки. Це забезпечує три ключові сервіси:

– автентифікація (Authentication) – перевірка того, що повідомлення надійшло від легітимного джерела. Для цього використовуються хеш-алгоритми, такі як MD5 або, що більш безпечно, SHA;

– конфіденційність (Privacy/Encryption) – шифрування даних для захисту від перехоплення. Підтримуються алгоритми шифрування, такі як DES, 3DES та сучасний стандарт AES (128, 192, 256 біт);

– цілісність (Integrity) – гарантія того, що дані не були змінені під час передачі, що забезпечується механізмами автентифікації.

На зміну SNMP, з його реактивною моделлю опитування, приходять більш сучасні, гнучкі та продуктивні підходи, орієнтовані на автоматизацію та аналіз даних у реальному часі. Їх можна розділити на дві основні категорії: API-орієнтований моніторинг та стрімінгову телеметрію.

API-орієнтований моніторинг (програмованість та гнучкість).

Цей підхід перетворює мережевий пристрій з «чорної скриньки», яку можна лише опитувати за допомогою специфічних протоколів, на повноцінний програмний сервіс. Основою є надання доступу до даних та конфігурації через стандартизовані програмні інтерфейси (API).

NETCONF (Network Configuration Protocol) [8] – це протокол, розроблений IETF як заміна застарілим методам конфігурування через CLI та SNMP. Він має чітку структуру та надає механізми для надійної та транзакційної зміни конфігурації:

- працює поверх SSH, що забезпечує надійне шифрування та автентифікацію;
- використовується XML (eXtensible Markup Language) для структурування запитів та відповідей;
- модель даних базується на моделях даних YANG, що дозволяє чітко та однозначно описати всю конфігурацію пристрою;
- надає механізми блокування конфігурації, транзакцій (можливість застосувати або відкотити групу змін), що робить його ідеальним для автоматизації налаштувань.

RESTCONF [8] є, «веб-оболонкою» для NETCONF. Він бере потужну модель даних YANG та робить її доступною через знайомий усім веб-розробникам інтерфейс RESTful API:

- працює поверх HTTPS, використовуючи стандартні HTTP-методи («GET» для читання, «POST/PUT/PATCH» для зміни, «DELETE» для видалення);

– зазвичай використовується JSON (JavaScript Object Notation), який є значно легшим та простішим для обробки, ніж XML;

– дуже низький поріг входу для розробників. Будь-яка система, що може робити HTTP-запити, може взаємодіяти з пристроєм. Це значно спрощує інтеграцію мережевої інфраструктури з сучасними програмними системами, скриптами автоматизації та платформами моніторингу.

Стрімінгова телеметрія (швидкість та гранулярність) – це нова парадигма, де мережевий пристрій сам, за моделлю «push», безперервним потоком надсилає дані про свій стан на систему моніторингу. Це кардинально відрізняється від SNMP, де система моніторингу повинна постійно «смикати» пристрій із запитаннями.

Цей підхід базується на сучасних технологіях.

gRPC (Google Remote Procedure Call) [9] – це високопродуктивний фреймворк для віддаленого виклику процедур, що працює поверх HTTP/2. Він є транспортом для передачі даних телеметрії та забезпечує:

– ефективність, бо використовує бінарний протокол Protobuf, що значно компактніший за текстові JSON/XML;

– потоковість – підтримує довготривалі з'єднання та двонаправлену потокову передачу даних, що є основою для телеметрії;

– безпеку – має вбудовану підтримку шифрування через TLS.

gNMI (gRPC Network Management Interface) [10] – це єдиний протокол для управління та отримання даних, що об'єднує в собі найкраще з обох світів:

– управління дозволяє змінювати конфігурацію пристрою (аналогічно до NETCONF);

– телеметрія надає механізм підписки (Subscribe), за допомогою якого колектор даних може «замовити» у пристрою певний набір метрик з потрібною частотою (наприклад, «надсилай мені дані про завантаження CPU кожні 5 секунд»);

– gNMI дозволяє мати єдиний інтерфейс і для конфігурації, і для моніторингу, що значно спрощує автоматизацію. Він дозволяє отримувати тисячі

оновлень на секунду, забезпечуючи безпрецедентну видимість процесів у мережі в реальному часі, що є недосяжним для SNMP.

Ринок систем моніторингу представлений різноманітними рішеннями, кожне з яких має свою нішу та архітектурні особливості. Їх можна умовно розділити на дві великі категорії: системи з відкритим вихідним кодом, що надають гнучкість та не вимагають ліцензійних відрахувань, та комерційні SaaS-платформи, що пропонують комплексний функціонал та простоту розгортання.

Системи з відкритим вихідним кодом («Open Source») – ці рішення є популярними завдяки своїй гнучкості, великій спільноті та відсутності прямої плати за ліцензію, хоча й вимагають більших затрат на налаштування та підтримку.

Zabbix є потужним та універсальним рішенням, що працює за принципом «все в одному» [11]. Його архітектура включає центральний Zabbix-сервер, який відповідає за збір, зберігання даних та логіку сповіщень; Zabbix-агентів, що встановлюються на керовані хости; та Zabbix-проксі, які можуть збирати дані від імені сервера, зменшуючи навантаження у великих та розподілених мережах.

Ключові переваги:

– універсальність адже він підтримує величезну кількість методів збору даних «з коробки»: SNMP, ICMP, IPMI, JMX, а також власні агенти для всіх популярних ОС;

– має потужні механізми автоматичного виявлення (network discovery) та низькорівневого виявлення (low-level discovery), що дозволяє автоматично знаходити пристрої, їх інтерфейси, файлові системи тощо;

– вбудований функціонал адже на відміну від інших рішень, Zabbix надає вбудований веб-інтерфейс, систему шаблонів, механізм сповіщень та базові можливості візуалізації без необхідності інтегрувати сторонні інструменти.

Prometheus + Grafana [2] – модульний стек для хмарних середовищ. На відміну від Zabbix, ця зв'язка є модульною і складається з кількох незалежних

компонентів. Prometheus виступає як ядро системи, що відповідає за збір та зберігання даних, а Grafana – як потужний інструмент для їх візуалізації.

Prometheus працює за моделлю «pull» (опитування), де він сам періодично звертається до спеціальних HTTP-ендпоінтів (exporters) на сервісах для збору метрик. Він пропонує надзвичайно ефективну модель зберігання часових рядів («time-series data») та потужну мову запитів PromQL.

Grafana є де-факто стандартом для візуалізації метрик. Вона дозволяє створювати інтерактивні дашборди з різноманітними типами графіків, таблиць та індикаторів, об'єднуючи дані з Prometheus та десятків інших джерел.

Ця зв'язка є особливо популярною в хмарних та DevOps-середовищах, зокрема для моніторингу Kubernetes, оскільки її архітектура ідеально підходить для динамічних, тимчасових сервісів. Однак її налаштування «з нуля», особливо системи сповіщень через окремий компонент Alertmanager, вимагає глибоких технічних знань [8].

Комерційні SaaS-платформи та концепція Observability.

Сегмент комерційних продуктів еволюціонував від класичних «on-premise» рішень, як SolarWinds, до гнучких хмарних SaaS-платформ. Лідерами тут є Datadog, New Relic та Dynatrace. Вони пропонують не просто моніторинг, а комплексні платформи спостережуваності (observability).

Концепція observability базується на «трьох стовпах», які разом дають повне розуміння стану системи:

– метрики (Metrics) – числові показники, що збираються через певні проміжки часу (напр., завантаження CPU);

– логи (Logs) – текстові записи про конкретні події, що відбулися в системі в певний час;

– трасування (Traces) – відстеження повного шляху одного запиту через усі мікросервіси та компоненти системи, що дозволяє знаходити «вузькі місця».

Ці платформи, як правило, використовують єдиного агента, який встановлюється на хост і автоматично збирає всі три типи даних. Їх переваги – швидке розгортання, інтуїтивно зрозумілі інтерфейси та потужні

аналітичні інструменти на базі ШІ (AIOps), що автоматично корелюють події та вказують на першопричину проблеми. Головними недоліками є висока вартість, що зазвичай залежить від обсягу даних або кількості хостів, та потенційна залежність від одного постачальника.

Проведений аналіз предметної області свідчить, що задача автоматичного моніторингу є багатогранною та критично важливою для сучасного бізнесу. Технології моніторингу стрімко розвиваються, переходячи від простого опитування по SNMP до інтелектуального аналізу потоків телеметрії в реальному часі.

Аналіз існуючих рішень виявив наступну проблематику:

– комерційні SaaS-платформи пропонують передовий функціонал, але їх вартість може бути невиправдано високою для моніторингу суто мережевої інфраструктури в малих та середніх компаніях;

– потужні відкриті системи (стек Prometheus/Grafana, Zabbix) є безкоштовними, але вимагають значних інвестицій часу та високої кваліфікації персоналу для їх розгортання, інтеграції та підтримки[12];

– існує розрив між класичними системами, орієнтованими на SNMP, та новітніми, що працюють з телеметрією. Часто бракує єдиного, простого інструменту, який би ефективно поєднував обидва підходи.

Таким чином, на ринку існує потреба в гнучкому, простому в розгортанні та використанні рішенні, яке б дозволило організаціям ефективно моніторити свою мережеву інфраструктуру, не вимагаючи при цьому значних фінансових або кадрових ресурсів.

1.2 Огляд і аналіз методів та засобів розробки системи автоматичного моніторингу справності мережевого обладнання у локальній мережі для вирішення проблеми дослідження

Після визначення проблематики та постановки завдань дослідження на основі аналізу предметної області, наступним кроком є вибір та обґрунтування

технологічного стеку для розробки програмної системи. Цей підрозділ присвячений аналізу архітектурних підходів, мов програмування, фреймворків та інструментів, що дозволять найбільш ефективно реалізувати поставлені цілі.

Архітектура програмного забезпечення є фундаментальним елементом, що визначає структуру системи, взаємодію її компонентів, а також можливості для подальшого масштабування та підтримки. Для проєкту такого типу можна розглянути три основні архітектурні підходи:

- монолітна архітектура передбачає, що всі функціональні компоненти системи (збір даних, обробка логіки, користувацький інтерфейс) об'єднані в єдиний програмний модуль. Такий підхід спрощує розробку та розгортання на початкових етапах, однак із ростом складності проєкту стає важким для модифікації та масштабування окремих частин;
- мікросервісна архітектура є протилежністю моноліту. Система розбивається на низку невеликих, незалежних сервісів, кожен з яких відповідає за вузьку бізнес-логіку (наприклад, сервіс опитування по SNMP, сервіс сповіщень). Хоча цей підхід забезпечує максимальну гнучкість та відмовостійкість, його реалізація та підтримка є значно складнішими і вимагають додаткових інструментів для оркестрації сервісів, що є надлишковим для завдань даної роботи;
- клієнт-серверна архітектура є оптимальним компромісом. Вона передбачає чіткий поділ системи на дві основні частини: сервер (backend), що відповідає за всю бізнес-логіку, збір, обробку та зберігання даних, та клієнт (frontend), який є користувацьким інтерфейсом і відповідає лише за візуалізацію даних, отриманих від сервера. Така структура забезпечує модульність, дозволяє незалежно розробляти та оновлювати серверну і клієнтську частини та є галузевим стандартом для веб-додатків. Враховуючи це, для розробки системи моніторингу доцільним є вибір саме клієнт-серверної архітектури.

Серверна частина є ядром системи. Її головні завдання: періодичне опитування мережевих пристроїв, обробка отриманих даних, збереження їх у базі даних та надання програмного інтерфейсу (API) для клієнтської частини. Вибір

мови програмування та фреймворку для бекенду є критично важливим, оскільки він напряму впливає на продуктивність, масштабованість та швидкість розробки системи. Було проаналізовано декілька провідних технологій.

Python є надзвичайно релевантним для даного завдання завдяки ідеальному балансу між швидкістю розробки та потужними можливостями для роботи з мережею [13].

Найбагатша екосистема для мережевих завдань – це ключовий фактор. Python має величезну кількість готових, зрілих бібліотек, що дозволяє не «винаходити колесо», а зосередитись на бізнес-логіці. Серед них:

– «pysnmp» забезпечує повноцінну реалізацію протоколу SNMP, що дозволяє легко відправляти запити та обробляти відповіді;

– «paramiko» є стандартом індустрії для взаємодії з обладнанням через SSH. «Netmiko» надає зручні абстракції для роботи з CLI-інтерфейсами пристроїв різних виробників;

– «icmplib» ідеально підходить для ефективного виконання ICMP-запитів, а «scapy» є потужним інструментом для глибокого аналізу та маніпуляції мережевими пакетами.

Лаконічний та читабельний синтаксис Python дозволяє реалізовувати складну логіку меншою кількістю коду, що значно прискорює процес розробки та спрощує подальшу підтримку.

Сучасні асинхронні фреймворки такі як FastAPI та Aiohttp, побудовані на базі «asyncio», дозволяють ефективно обробляти тисячі одночасних мережевих з'єднань (наприклад, опитувати сотні пристроїв паралельно), не блокуючи основний потік виконання.

Все ж є і недоліки, адже Python є інтерпретованою мовою, тому в «чистих» обчисленнях він поступається компільованим мовам, як Go чи Rust. Однак для завдань моніторингу, де основний час витрачається на очікування відповіді від мережі (I/O-bound операції), цей недолік нівелюється завдяки асинхронності.

Go – це компільована мова, розроблена Google спеціально для створення високопродуктивних та надійних мережевих сервісів [13].

Перевагами цієї мови є:

– Go компілюється в нативний машинний код, що забезпечує продуктивність, близьку до C;

– модель паралелізму в Go, що базується на горутинах («goroutines») та каналах («channels»), є надзвичайно легкою та ефективною. Запустити тисячі паралельних завдань для опитування пристроїв у Go значно простіше та менш ресурсозатратно, ніж у багатьох інших мовах;

– має чудові вбудовані інструменти для роботи з мережею, HTTP, TCP/UDP;

– статична типізація, яка допомагає виявляти багато помилок ще на етапі компіляції.

Також є і недоліки:

– кількість бібліотек для Go стрімко зростає, вона все ще поступається Python у розмаїтті готових рішень саме для взаємодії з пропрієтарним мережевим обладнанням;

– реалізація тієї ж логіки може вимагати написання більшої кількості коду порівняно з Python.

Node.js – це середовище виконання JavaScript, побудоване на рушії V8 від Google [13]. Його ключова особливість – подієво-орієнтована, неблокуюча модель вводу-виводу. А його переваги це:

– чудова продуктивність для I/O-операцій адже Node.js ідеально підходить для завдань, де потрібно обробляти велику кількість одночасних з'єднань, що є типовим для систем моніторингу;

– єдина мова для backend та frontend – JavaScript (або TypeScript для кращої типізації) на сервері та в клієнті спрощує розробку та дозволяє команді бути більш гнучкою;

– менеджер пакунків NPM є найбільшим у світі, що надає доступ до мільйонів готових модулів.

Недоліками є:

– як і у випадку з Go, знайти готові бібліотеки для роботи зі специфічним обладнанням (напр., для аналізу MIB-баз SNMP) може бути складніше, ніж у Python;

– сучасні «async/await» значно спростили роботу, управління складними асинхронними ланцюжками («callback hell») може бути проблемою [9].

Для створення вебсервера та API на Python було проаналізовано три провідні фреймворки: Django, Flask та FastAPI. Кожен із них пропонує свій унікальний підхід та філософію, і вибір оптимального рішення залежить від конкретних вимог проєкту.

Django – це повнофункціональний, високорівневий фреймворк, розроблений для швидкого створення складних, базованих на даних вебсайтів [14]. Його основна філософія – надати розробнику всі необхідні інструменти «з коробки», що мінімізує потребу у сторонніх бібліотеках.

Django дотримується архітектурного шаблону «Model-View-Template». Він має чітку та строгу структуру проєкту, що з одного боку робить розробку передбачуваною, а з іншого – менш гнучкою.

Ключові компоненти:

– Django ORM (Object-Relational Mapper) – це найпотужніша частина фреймворку. Вона дозволяє розробникам працювати з базами даних (PostgreSQL, MySQL тощо), використовуючи Python-об'єкти замість написання SQL-запитів. Це значно прискорює розробку та підвищує безпеку;

– автоматично генерована адміністративна панель, яка дозволяє керувати даними в базі через зручний веб-інтерфейс. Це ідеальний інструмент для швидкого створення внутрішніх панелей управління;

– має вбудовані компоненти для автентифікації користувачів, маршрутизації, шаблонізації HTML-сторінок та безпеки (захист від CSRF, XSS-атак).

Недоліки даного проєкту:

– для системи, основна мета якої – надання API, більшість функціоналу Django (наприклад, система шаблонів, адмін-панель) є надлишковою. Це робить проєкт більш громіздким;

– Django є синхронним фреймворком (базується на WSGI), що робить його менш ефективним для завдань з великою кількістю I/O-операцій, таких як одночасне опитування сотень мережевих пристроїв.

Flask є повною протилежністю Django – це мікрофреймворк, що означає, що він надає лише абсолютно необхідний мінімум для створення веб-додатку: маршрутизацію та обробку запитів [15].

Flask не нав'язує розробнику жодної структури проєкту чи інструментів. Ви самі вирішуєте, яку ORM використовувати (наприклад, SQLAlchemy), як структурувати свій код і які бібліотеки підключати.

Перевагами цього проєкту є:

– повну свободу у виборі технологій та архітектури;

– його ядро дуже легко вивчити, що дозволяє швидко почати розробку.

Недоліки даного проєкту:

– для реалізації API потрібно вручну підключати та налаштовувати безліч розширень: SQLAlchemy для роботи з базою даних, Marshmallow для валідації та серіалізації даних, Flask-RESTful або аналоги для побудови REST API – це збільшує складність та час розробки;

– як і Django, Flask є синхронним (WSGI), що є недоліком для високопродуктивних мережевих завдань.

FastAPI займає ідеальну нішу між цими двома підходами [16]. Він пропонує сучасний підхід, що поєднує високу продуктивність з винятковою зручністю для розробника, запозичуючи найкраще з обох світів.

Ключова відмінність FastAPI полягає в його асинхронній архітектурі, що базується на ASGI (Asynchronous Server Gateway Interface). У поєднанні з ASGI-сервером, таким як Uvicorn, це забезпечує продуктивність на рівні компільованих мов.

FastAPI значно покращує досвід розробки завдяки вбудованим інструментам:

– використання бібліотеки «Pydantic» та підказок типів Python дозволяє автоматично валідувати дані без підключення сторонніх бібліотек;

– FastAPI автоматично створює документацію «Swagger UI» та «ReDoc» безпосередньо з коду, що завжди гарантує її актуальність і спрощує тестування;

– вбудована система впровадження залежностей, що дозволяє створювати чистий та легко тестований код.

Таким чином, у порівнянні з громіздкістю Django та необхідністю ручного налаштування Flask, зв'язка Python + FastAPI є обґрунтованим і сучасним вибором для розробки серверної частини системи моніторингу. Вона забезпечує необхідну продуктивність, значно прискорює розробку та підвищує надійність коду [17].

Клієнтська частина має забезпечувати зручний та інформативний веб-інтерфейс для користувачів, дозволяючи переглядати стан пристроїв, аналізувати історичні дані у вигляді графіків та налаштовувати параметри моніторингу. Основою будь-якого веб-інтерфейсу є HTML, CSS та JavaScript.

Для створення сучасного динамічного інтерфейсу доцільно використовувати один із популярних JavaScript-фреймворків. Кожен із них пропонує власну філософію та набір інструментів, тому вибір залежить від вимог проєкту, досвіду команди та бажаного рівня гнучкості. Найбільш поширеними на сьогодні є Angular, Vue.js та React.

Angular, розроблений та підтримуваний компанією Google, є повноцінним, комплексним фреймворком. Його головна ідея – надати розробникам єдину, стандартизовану платформу для створення великих та складних корпоративних додатків.

– це фреймворк, а не бібліотека. Він диктує чітку структуру проєкту та архітектуру, зазвичай «Model-View-Controller» (MVC);

– за замовчуванням використовує TypeScript, що додає статичну типізацію до JavaScript. Це підвищує надійність коду та спрощує його підтримку у великих проєктах;

– надає величезну кількість вбудованих інструментів «з коробки»: потужний CLI (Command Line Interface), власні рішення для маршрутизації, управління формами, HTTP-запитів та впровадження залежностей. Це забезпечує стабільність та передбачуваність розробки;

– має значно вищий поріг входу та вважається більш складним для вивчення. Його строга структура може бути надлишковою та обмежуючою для проєктів середнього розміру, де потрібна більша гнучкість.

Vue.js, створений Еваном Ю, позиціонується як прогресивний фреймворк. Це означає, що його можна легко інтегрувати в існуючі проєкти для модернізації окремих частин або використовувати для побудови повноцінних односторінкових додатків (SPA) з нуля. Він особливий тим, що:

– Vue прагне знайти баланс між структурованістю Angular та гнучкістю React;

– відомий своєю документацією та низьким порогом входу, що робить його дуже привабливим для новачків. Його однофайлові компоненти розширення «.vue», які об'єднують HTML-шаблон, CSS-стилі та JavaScript-логіку в одному файлі, часто хвалять за зручність та читабельність;

– хоча його екосистема стрімко зростає, вона все ще поступається за розміром екосистемі React, що може означати менший вибір готових бібліотек для вирішення вузькоспеціалізованих завдань.

Для даного проєкту було обрано бібліотеку React, розроблену та підтримувану компанією Meta (Facebook) [18]. На відміну від Angular, React є бібліотекою, що фокусується виключно на створенні користувацьких інтерфейсів. Це надає розробникам максимальну гнучкість у виборі інших інструментів та архітектури. Вибір обґрунтований наступними факторами:

– React дозволяє будувати інтерфейс з невеликих, незалежних та ізольованих компонентів (графік, таблиця, індикатор стану). Для системи

моніторингу це ідеальний підхід, оскільки кожен елемент дашборду можна розробляти, тестувати та оновлювати окремо, а потім комбінувати їх у складніші структури. Це значно спрощує розробку та повторне використання коду;

- React має найбільшу екосистему серед усіх frontend-технологій. Існує велика кількість готових бібліотек для будь-яких завдань, зокрема для візуалізації даних (наприклад, «Chart.js», «Recharts», «D3.js»), що дозволяє швидко реалізувати побудову інтерактивних графіків завантаженості CPU, пам'яті тощо. Крім того, є потужні рішення для управління станом («Redux», «MobX») та великі бібліотеки готових UI-компонентів («Material-UI», «Ant Design»);
- завдяки величезній спільноті та підтримці з боку Meta, для React існує безліч навчальних матеріалів, форумів та готових рішень. Це значно спрощує вирішення будь-яких проблем, що можуть виникнути у процесі розробки.

У підсумку, хоча Angular є потужним вибором для великих стандартизованих проєктів, а Vue – чудовим збалансованим рішенням, для розробки гнучкого дашборду моніторингу React є оптимальним варіантом завдяки своїй компонентній архітектурі та найбагатшій екосистемі, що дозволяє швидко інтегрувати складні елементи візуалізації даних [19].

Система моніторингу генерує великі обсяги даних, які за своєю природою є часовими рядами («time-series data»), тобто набором показників, прив'язаних до часових міток. Ефективність зберігання та обробки таких даних є ключовим фактором продуктивності системи.

Було розглянуто два підходи. Перший – використання спеціалізованих NoSQL баз даних, таких як InfluxDB, які «з коробки» оптимізовані для роботи з часовими рядами. Другий – використання традиційних реляційних СКБД. Враховуючи необхідність зберігати не тільки метрики, а й реляційну інформацію (список пристроїв, їхні налаштування, користувачів системи), було прийнято рішення зупинитися на реляційній моделі.

Оптимальним вибором у цьому сегменті є PostgreSQL – потужна та надійна об'єктно-реляційна СКБД з відкритим кодом. Ключовою перевагою PostgreSQL для нашого завдання є наявність розширення TimescaleDB. Це розширення

перетворює стандартні таблиці PostgreSQL на гіпертаблиці, оптимізовані для швидкого запису та складних аналітичних запитів до часових рядів. Такий підхід дозволяє поєднати надійність і гнучкість SQL для зберігання конфігураційних даних із високою продуктивністю спеціалізованої «time-series» бази даних для метрик [20].

Для забезпечення якості та ефективності процесу розробки будуть використані стандартні для галузі інструменти:

- Docker – платформа контейнеризації, яка дозволить «запакувати» всі частини програми (backend, frontend, базу даних) в ізольовані контейнери [21];
- Git – розподілена система контролю версій для відстеження змін у вихідному коді, організації командної роботи та створення резервних копій [22].

Це гарантує, що система буде однаково працювати в будь-якому середовищі та значно спрощує процес її розгортання на сервері.

На основі проведеного аналізу методів та засобів розробки для реалізації програмної системи моніторингу було сформовано наступний технологічний стек. Система буде побудована за клієнт-серверною архітектурою. Серверна частина (backend) буде реалізована мовою Python з використанням асинхронного веб-фреймворку FastAPI. Для зберігання даних буде використана СКБД PostgreSQL з розширенням TimescaleDB. Інтерактивний користувацький інтерфейс (frontend) буде створено за допомогою бібліотеки React. Увесь процес розробки буде вестися в системі контролю версій Git, а для розгортання системи буде застосовано технологію контейнеризації Docker.

1.3 Постановка завдання на кваліфікаційну роботу магістра

В ході досліджень для виконання кваліфікаційної роботи було проаналізовано сучасний стан проблеми, визначено актуальність завдання та проаналізовано предметну область для розробки веб-сайтів.

Спираючись на проведений аналіз було сформовано мету роботи. Розробити систему автоматичного моніторингу справності мережевого

обладнання, яка поєднує високу продуктивність асинхронного збору даних з гнучкістю налаштувань та надійним механізмом сповіщень.

Також відповідно до визначеної мети роботи, сформульовано наступні конкретні завдання, які необхідно досягти для успішного виконання кваліфікаційної роботи:

– проаналізувати та обґрунтувати архітектурний та технологічний стек;

– розробити модель даних з оптимізованим сховищем для часових рядів;

– спроектувати та реалізувати високопродуктивне асинхронне ядро опитування;

– реалізувати механізми інтелектуальної обробки сповіщень;

– створити RESTful API для керування системою з безпечною автентифікацією;

– розробити інтерактивний веб-дашборд для візуалізації метрик у реальному часі;

– забезпечити високу переносимість системи шляхом повної контейнеризації;

– провести експериментальне дослідження результативності розробленої системи.

РОЗДІЛ 2
ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ СИСТЕМИ АВТОМАТИЧНОГО МОНІТОРИНГУ СПРАВНОСТІ МЕРЕЖЕВОГО ОБЛАДНАННЯ У ЛОКАЛЬНІЙ МЕРЕЖІ

2.1 Обґрунтування вибору шляхів, технологій, алгоритмів і засобів вирішення поставленого завдання

Для розробки системи автоматичного моніторингу мережевого обладнання, яка вимагає високої швидкості обробки запитів, роботи в реальному часі та зручної візуалізації даних, обрано технологічний стек, що складається з Python (FastAPI) для бекенду, React для фронтенду та PostgreSQL (з розширенням TimescaleDB) для зберігання даних. Такий вибір забезпечує ефективну обробку великої кількості мережевих метрик, масштабованість системи та сучасний користувацький досвід.

Обраний стек є збалансованим поєднанням інструментів, що повністю відповідає специфіці предметної області. Система моніторингу передбачає постійне опитування сотень пристроїв, що вимагає асинхронності, потребу у зберіганні історичних даних (часових рядів) та наявність інтерактивних дашбордів для інженерів. Саме тому обрані технології мають не лише відповідати сучасним стандартам, а й гарантувати стабільність при високих навантаженнях.

Ключовим критерієм для бекенду системи моніторингу є здатність виконувати тисячі мережевих запитів (SNMP, ICMP) одночасно, не блокуючи роботу системи. Для цього було обрано мову Python та фреймворк FastAPI. На відміну від традиційних синхронних фреймворків, FastAPI базується на стандарті ASGI та дозволяє використовувати асинхронний код, що критично важливо для швидкого опитування мережі. Це добре ілюструє порівняння, наведене у таблиці 2.1.

Таблиця 2.1 – Порівняння ключових Python-фреймворків для бекенду

Характеристика	FastAPI	Django	Flask
Архітектура	Асинхронна (ASGI), ідеальна для I/O операцій	Синхронна (WSGI), блокуюча модель	Синхронна (WSGI), блокуюча модель
Швидкодія	Дуже висока (на рівні Go/Node.js)	Середня, має накладні витрати.	Середня
Робота з даними	Вбудована валідація через Pydantic	Потужна вбудована ORM	Потребує підключення сторонніх розширень
Документація API	Генерується автоматично (Swagger UI)	Потребує додаткових інструментів	Потребує ручного налаштування
Фокус застосування	Високонавантажені API, мікросервіси, Real-time системи	Класичні веб-сайти, CMS, E-commerce	Прості сервіси, прототипи

Зважаючи на вимоги до системи щодо відображення стану мережевого обладнання в режимі реального часу та динамічної побудови графіків завантаженості каналів, клієнтська частина програмного комплексу реалізована за принципом односторінкового застосунку (SPA). Для цього було обрано бібліотеку React.

Ключовою перевагою React у контексті даної роботи є використання технології «Virtual DOM», яка дозволяє мінімізувати кількість звернень до реального «DOM-дерева» браузера. Це забезпечує високу продуктивність при частому оновленні даних, що є критично важливим для моніторингу, де параметри змінюються щосекунди.

Компонентний підхід React дозволяє побудувати модульний інтерфейс, де кожен елемент системи моніторингу є ізольованим компонентом із власною логікою та стилями. Це не лише спрощує підтримку та тестування коду, але й

дозволяє повторно використовувати компоненти в різних частинах панелі керування. Крім того, React має розвинену екосистему, що полегшує інтеграцію спеціалізованих бібліотек для візуалізації даних (наприклад, «Recharts» або «Chart.js»), необхідних для побудови аналітичних графіків.

У порівнянні з фреймворком Angular, React є більш гнучким та менш громіздким, що дозволяє розробнику самостійно обирати інструменти для маршрутизації та управління станом, оптимізуючи кінцевий розмір застосунку. Порівняно з Vue.js, React має ширшу підтримку у корпоративному сегменті та більшу кількість готових архітектурних патернів для масштабованих систем. Детальне порівняння характеристик розглянутих технологій наведено у таблиці 2.2.

Таблиця 2.2 – Порівняння інструментів розробки інтерфейсу

Характеристика	React	Angular	Vue.js
Тип	Бібліотека, надає гнучкість архітектури	Фреймворк, диктує жорстку структуру	Прогресивний фреймворк
Екосистема	Найбільша, величезний вибір бібліотек для графіків	Велика, але часто «замкнена» на собі	Активно розвивається, але менша за React
Крива навчання	Середня, вимагає розуміння JSX	Висока багато специфічних концепцій	Низька, простий старт
Підходить для	Складних дашбордів з динамічними даними	Великих корпоративних ERP-систем	Проєктів, де важлива швидкість розробки

Система моніторингу за своєю природою генерує два принципово різні типи даних, які вимагають різних підходів до зберігання. Використання звичайної SQL-бази для зберігання часових рядів є неефективним через необхідність частої індексації та партиціонування вручну. По мірі накопичення даних, операції вставки (INSERT) в стандартній реляційній базі уповільнюються,

а запити на агрегацію даних за великий період часу стають надмірно ресурсомісткими. Водночас, використання суто NoSQL-баз, таких як InfluxDB, ускладнює роботу зі зв'язками між метриками та конфігураційними даними, що робить складним виконання запитів, що вимагають об'єднання інформації про пристрій з його часовими метриками.

Тому для реалізації системи обрано гібридний підхід: PostgreSQL як основна реляційна база даних, доповнена розширенням TimescaleDB. Це рішення дозволяє поєднати надійність, цілісність та потужні можливості SQL для зберігання конфігурації зі швидкістю, оптимізованою для запису метрик. TimescaleDB автоматично перетворює таблиці часових рядів на гіпертаблиці, які оптимізовані для масової вставки та запитів по часовому діапазону за рахунок автоматичного партиціонування. Це забезпечує необхідну швидкість запису метрик, підтримуючи при цьому можливість використання стандартних SQL-запитів та зв'язків з реляційними даними. Це демонструє порівняння, наведене у таблиці 2.3.

Таблиця 2.3 – Порівняння підходів до зберігання даних моніторингу

Характеристика	PostgreSQL + TimescaleDB	MySQL (Standard)	InfluxDB (NoSQL)
Тип даних	Гібридний: Реляційні дані + часові ряди	Тільки реляційні дані	Тільки часові ряди
Мова запитів	Стандартний SQL	Стандартний SQL	Специфічна мова запитів (Flux/InfluxQL)
Швидкість запису	Оптимізована для масового запису метрик	Знижується при великих обсягах даних	Висока
Цілісність даних	Підтримка зовнішніх ключів (JOIN) між пристроями та їх метриками	Підтримується	Не підтримує зв'язки (JOIN)

Таким чином, комбінація FastAPI, React та PostgreSQL (TimescaleDB) дозволяє створити архітектуру, яка поєднує продуктивний асинхронний бекенд, інтерактивний фронтенд та спеціалізоване сховище даних. Архітектура проекту зображена нижче на рисунку 2.1.

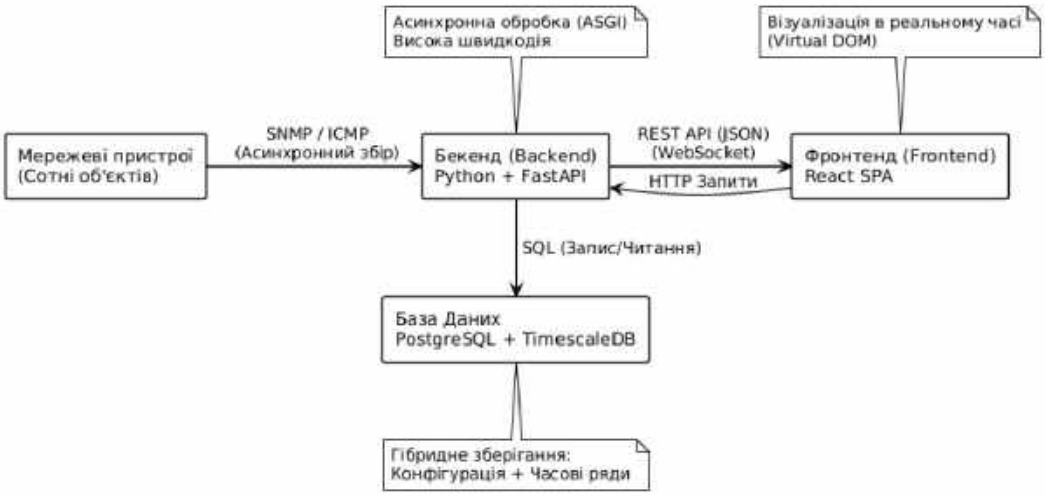

Рисунок 2.1 – UML-діаграма архітектури проєкту

Такий стек технологій забезпечує необхідну швидкодію опитування, зручність аналізу даних та можливість масштабування, що робить його оптимальним для реалізації системи моніторингу мережевого обладнання.

2.2 Практична реалізація об'єкта проектування

Практична реалізація програмної системи моніторингу мережевого обладнання виконувалася ітеративним методом. Розробка була розділена на кілька логічних етапів: проектування моделі даних, реалізація асинхронного ядра опитування (Backend), створення клієнтського інтерфейсу (Frontend) та налаштування інфраструктури розгортання.

Фундаментом інформаційної системи є шар збереження даних, який спроектовано з урахуванням вимог до високого навантаження при записі метрик та гнучкості конфігурації. У якості системи керування базами даних обрано

PostgreSQL. Для взаємодії з БД на рівні Python-додатку використано сучасний асинхронний драйвер «asyncpg», який забезпечує неблокуючий ввід-вивід та високу швидкість серіалізації даних.

Архітектура бази даних реалізована за підходом «Code-First» з використанням ORM SQLAlchemy. Це означає, що структура таблиць описується у вигляді Python-класів (моделей), що дозволяє використовувати об'єктно-орієнтовані патерни проектування.

Структура бази даних, описана у модулі «models.py», складається з нормалізованих таблиць для конфігураційних даних та оптимізованої таблиці для часових рядів.

Сутність «Device» – це центральна таблиця системи. Для забезпечення унікальності пристроїв у мережі, на поле «ip_address» встановлено обмеження унікальності («unique=True»). Особливу увагу приділено реалізації бізнес-логіки «тихих годин». Використання спеціалізованого типу даних «Time» (SQL TIME) для полів «dnd_start_time» та «dnd_end_time» дозволяє виконувати точні темпоральні порівняння на рівні бази даних, незалежно від дати.

Сутності «DeviceType» та «OidConfig» реалізують гнучку систему конфігурації за принципом «One-to-Many». Таблиця «oid_configs» містить зовнішній ключ (ForeignKey) на «device_types». Також налаштовано каскадне видалення (cascade="all, delete-orphan"), що гарантує автоматичне очищення всіх OID при видаленні типу пристрою, підтримуючи референційну цілісність даних.

Сутність «StatusHistory» – це таблиця спроектована для зберігання великих обсягів історичних даних. Композитний первинний ключ – для однозначної ідентифікації запису та оптимізації індексів використано пару полів «timestamp» та «device_id». А використання бінарного JSON (JSONB) дозволяє зберігати неструктуровані дані. Це критично важливо для гетерогенних мереж, де різні пристрої віддають різний набір параметрів (наприклад, комутатор передає статус портів, а сервер – завантаження RAM). JSONB підтримує індексацію, що дозволяє в майбутньому виконувати швидкий пошук по ключах всередині JSON-об'єкта. Нижче в лістингу 2.1 наведено код сутності «StatusHistory».

Лістинг 2.1 – Демонстрація сутності «StatusHistory»

```
    class StatusHistory(Base):
        __tablename__ = "status_history"
        timestamp: Mapped[DateTime] =
mapped_column(DateTime(timezone=True), server_default=func.now(),
primary_key=True)
        device_id: Mapped[int] =
mapped_column(ForeignKey("devices.id"), primary_key=True)
        is_online: Mapped[bool] = mapped_column(Boolean, default=False)
        metrics: Mapped[dict | None] = mapped_column(JSONB,
nullable=True)
        device: Mapped["Device"] =
relationship(back_populates="status_history")
        def __repr__(self): return
f"<Status(device_id={self.device_id}, ts={self.timestamp},
online={self.is_online})>"
```

Кінець лістингу 2.1

Сутність «User» зберігає облікові дані адміністраторів. Паролі не зберігаються у відкритому вигляді, замість цього у поле «hashed_password» записується хеш, згенерований алгоритмом «PBKDF2».

Налаштування взаємодії з БД реалізовано у модулі «database.py». Використовується асинхронний рушій «create_async_engine». Важливим технічним рішенням є налаштування фабрики сесій «async_sessionmaker» з значенням «False» параметру, назвою «expire_on_commit». Це запобігає автоматичному «старінню» об'єктів після коміту транзакції, що є необхідною умовою для коректної роботи асинхронного коду при зверненні до атрибутів об'єктів поза межами активної сесії.

Для контролю версій схеми бази даних впроваджено інструмент «Alembic». Це дозволяє безпечно вносити зміни в структуру БД (наприклад, додавати нові поля) без втрати даних, генеруючи файли ревізій на основі змін у моделях SQLAlchemy.

Серверна частина системи реалізована на базі високопродуктивного мікрофреймворку FastAPI. Вибір цього інструменту обумовлений його базовою підтримкою асинхронного програмування (стандарт ASGI), що є критично

важливим для «I/O-bound» задач, таких як множинні запити до бази даних та мережевого обладнання.

Архітектура додатку (файл «main.py») побудована на патерні «Dependency Injection» (DI). Це дозволяє створювати слабкозв'язаний код та ефективно керувати ресурсами. Зокрема, реалізовано залежність «get_db», яка відповідає за життєвий цикл підключення до бази даних: вона створює асинхронну сесію перед обробкою запиту та гарантовано закриває її після відправки відповіді клієнту. Це запобігає витоку ресурсів (connection leaks) при високому навантаженні.

Окрім того, для забезпечення взаємодії з клієнтською частиною (React), яка розміщена на іншому домені/порті, налаштовано «CORS Middleware» (Cross-Origin Resource Sharing). Це дозволяє браузеру безпечно виконувати AJAX-запити до API, обмежуючи доступ лише довіреним джерелам (наприклад, «localhost:5173»).

Для забезпечення цілісності даних на етапі входу в систему реалізовано шар «Data Transfer Objects» (DTO) за допомогою бібліотеки «Pydantic». У модулі «schemas.py» описано суворі схеми даних для запитів та відповідей. Цей підхід вирішує дві задачі:

– валідація – автоматична перевірка типів даних (наприклад, чи є переданий рядок валідною IP-адресою, чи вказано обов'язкові поля);

– безпека – відокремлення внутрішніх моделей БД від публічних схем API.

Наприклад, схема «UserRead» повертає лише ім'я та роль користувача, автоматично виключаючи поле з хешем пароля, що унеможливлює випадковий витік чутливих даних.

Відповідно до методології «Twelve-Factor App», конфігурація відокремлена від коду. Модуль «config.py» використовує бібліотеку «pydantic-settings» для зчитування параметрів зі змінних середовища («.env»). Код модуля «config.py» наведено нижче в лістингу 2.2. Валідатор гарантує, що додаток не запуститься, якщо відсутні критичні змінні, такі як «DATABASE_URL» або «SECRET_KEY».

Лістинг 2.2 – Код модуля config.py

```
from pydantic_settings import BaseSettings, SettingsConfigDict
class Settings(BaseSettings):
        model_config = SettingsConfigDict(env_file=".env",
env_file_encoding='utf-8')
    DATABASE_URL: str
    SECRET_KEY: str
    ALGORITHM: str
    ACCESS_TOKEN_EXPIRE_MINUTES: int
    BOT_TOKEN: str
    CHAT_ID: str
settings = Settings()
```

Кінець лістингу 2.2

Система автентифікації (auth.py) реалізована на основі стандарту «OAuth2» (схема «Password Flow») з використанням JWT-токенів (JSON Web Tokens).

Для захисту паролів застосовано алгоритм хешування «PBKDF2-HMAC-SHA256» (бібліотека «passlib»), що відповідає сучасним стандартам криптографічної стійкості.

Реалізовано «Role-Based Access Control» (RBAC). У системі виділено дві ролі: «viewer» (лише перегляд) та «admin» (повний доступ). Залежність «get_admin_user» у «main.py» блокує доступ до модифікуючих операцій (створення, видалення пристроїв) для звичайних користувачів, повертаючи помилку «403 Forbidden». Код цієї залежності наведено нижче в лістингу 2.3.

Лістинг 2.3 – Код залежності «get_admin_user»

```
async def get_admin_user(
current_user: Annotated[schemas.UserRead, Depends(get_current_user)]
) -> schemas.UserRead:

if current_user.role != "admin":
raise HTTPException(
status_code=status.HTTP_403_FORBIDDEN,
detail="Недостатньо прав. Потрібна роль адміністратора."
)
return current_user
```

Кінець лістингу 2.3

API надає «RESTful» інтерфейс, логічно розділений на групи маршрутів:

– «/api/v1/auth» – маршрути реєстрації та отримання JWT-токенів;

– «/api/v1/devices» – забезпечує повний цикл CRUD-операцій. Для оптимізації продуктивності при читанні списків пристроїв використано стратегію «жадібного завантаження» (selectinload) в SQLAlchemy. Це дозволяє одним SQL-запитом отримати дані про пристрій, його тип та пов'язані OID, уникаючи проблеми «N+1 запитів»;

– «/api/v1/device-types» – ендпоінти для керування шаблонами обладнання та динамічного додавання OID;

– «/api/v1/dashboard» – спеціалізований агрегуючий ендпоінт, який формує комплексний зріз стану системи (пристрій + його останній статус) для швидкого відображення на головній сторінці.

Важливою особливістю реалізації є автоматична генерація інтерактивної документації Swagger UI (/docs), що значно спрощує тестування API та взаємодію з фронтенд-розробниками. Знімок екрана сторінки «\docs» зорбражено нижче на рисунку 2.2.

Рисунок 2.2 – Сторінка «\docs»

Ядром системи моніторингу є автономний модуль «monitoring_worker.py», спроектований як фоновий демон-процес. Його архітектура повністю відокремлена від веб-сервера API, що забезпечує стабільність збору даних навіть у випадку високого навантаження на користувацький інтерфейс. Головна задача модуля – забезпечення високопродуктивного, конкурентного опитування гетерогенного парку обладнання з мінімальними затримками.

Для вирішення проблеми блокування вводу-виводу (I/O Blocking), характерної для мережевих операцій, використано бібліотеку «asyncio». На відміну від традиційного багатопотокового підходу (Threading), який має накладні витрати на перемикання контексту операційною системою, асинхронний підхід використовує «Event Loop» (цикл подій) в одному потоці.

Функція «run_monitoring_cycle» реалізує паттерн «Fan-Out»:

– система завантажує з БД список усіх цільових пристроїв;
– для кожного пристрою формується об'єкт асинхронної задачі («Task»).

Задачі запускаються на виконання методом «asyncio.gather», що дозволяє відправляти запити до сотень пристроїв фактично одночасно, очікуючи відповіді паралельно. Код функції «run_monitoring_cycle» та методу «asyncio.gather» наведено нижче в лістингу 2.4.

Лістинг 2.4 – Код функції run_monitoring_cycle та методу «asyncio.gather»

```
async def check_single_device(device) -> models.StatusHistory:

# 1. PING
     is_online = await ping_device(device.ip_address)
     metrics = {}

# 2. SNMP
     if is_online and device.device_type and device.snmp_community:
     oids = device.device_type.oids
     if oids:
     for oid_obj in oids:
     metric_name = oid_obj.metric_name
     oid_str = oid_obj.oid_value.strip()
     try:
     errorIndication, errorStatus, errorIndex, varBinds = await
getCmd(
     SnmpEngine(),
```

```
        CommunityData(device.snmp_community, mpModel=1),
        UdpTransportTarget((device.ip_address,     161),     timeout=1,
retries=0),
ContextData(),
        ObjectType(ObjectIdentity(oid_str))
        if not errorIndication and not errorStatus:
        for varBind in varBinds:
        metrics[metric_name] = str(varBind[1])
        except Exception:
        pass

return models.StatusHistory(
        device_id=device.id,
        is_online=is_online,
        metrics=metrics,
        timestamp=datetime.now())

tasks = [check_single_device(d) for d in devices]
        history_entries = await asyncio.gather(*tasks)
```

Кінець лістингу 2.4

Перевірка доступності вузлів реалізована через асинхронний запуск підпроцесу системи за допомогою «asyncio.create_subprocess_exec». Це дозволяє використовувати нативну, високоефективну утиліту «ping» операційної системи. Важливою особливістю реалізації є кросплатформеність: алгоритм автоматично визначає тип ОС (Windows або Linux/Unix) за допомогою модуля «platform» та адаптує аргументи команди (-n для Windows проти -c для Linux). Це дозволяє розгортати систему як на Linux-серверах (Docker), так і на робочих станціях розробників під Windows без зміни коду.

Для збору детальних метрик використано бібліотеку pysnmp (версія 4.4.12). Функція «check_single_device» реалізує логіку динамічного опитування:

– для кожного пристрою визначається його тип (через відношення ORM «device.device_type»);

– з БД завантажується список OID («oid_configs»), налаштованих саме для цього типу;

– виконується асинхронний запит «getCmd» по протоколу UDP. Такий підхід дозволяє системі бути гнучкою: додавання нової метрики (наприклад,

температури для нового типу комутаторів) не вимагає перекомпіляції або перезапуску воркера, а лише додавання запису в базу даних.

Для запобігання дублюванню сповіщень (alert spamming) реалізовано механізм збереження стану в оперативній пам'яті («last_known_status»). Сповіщення генерується лише за умови зміни статусу (Edge Triggering), тобто коли «current_status != previous_status». Це дозволяє уникнути ситуації, коли адміністратор отримує повідомлення «Device Offline» кожні 30 секунд, поки пристрій недоступний.

Унікальною функцією системи є інтелектуальна обробка часу тиші. Логіка перевірки, реалізована у воркері, враховує складні часові інтервали, включаючи перехід через північ. Алгоритм перевірки наступний:

– якщо «Start < End» (наприклад, 09:00-18:00): перевіряється входження поточного часу в діапазон «Start <= Now <= End»;

– якщо «Start > End» (наприклад, 22:00-07:00): використовується логіка «АБО» – час тиші активний, якщо «Now >= Start» АБО «Now <= End».

Це критично важливо для моніторингу офісного обладнання, яке планово вимикається на ніч, запобігаючи хибним викликам чергових адміністраторів. Код функції не турбувати наведено нижче в лістингу 2.5.

Лістинг 2.5 – Код функції «Не турбувати»

```
dnd_start = device_obj.dnd_start_time
      dnd_end = device_obj.dnd_end_time
      is_in_dnd = False
      if dnd_start and dnd_end:
      if dnd_start <= dnd_end:
      is_in_dnd = dnd_start <= now_time_kyiv <= dnd_end
      else:
      is_in_dnd = now_time_kyiv >= dnd_start or now_time_kyiv <= dnd_end
```

Кінець лістингу 2.5

Система сповіщень Інтеграція з месенджером реалізована через модуль «telegram_bot.py» з використанням асинхронної бібліотеки «aiogram 3.x». Бот ініціалізується в режимі «DefaultBotProperties»(parse_mode=ParseMode.HTML),

що дозволяє форматувати повідомлення (жирний шрифт для назв пристроїв, емодзі для статусу). Відправка повідомлень також відбувається асинхронно, щоб затримки API Telegram не впливали на загальний цикл моніторингу. Код циклу відправки сповіщень телеграм ботом наведено нижче в лістингу 2.6. Передбачено обробку помилок мережі та випадків, коли чат недоступний, що забезпечує відмовостійкість «воркера».

Лістинг 2.6 – Код циклу телеграм бота

```
async def send_telegram_message(message_text: str):
    if not bot:
         logging.warning("Telegram бот не ініціалізовано. Повідомлення не
надіслано.")
        return
    try:
        # Використовуємо CHAT_ID з settings
                    await    bot.send_message(chat_id=settings.CHAT_ID,
text=message_text)
       logging.info(f"Надіслано повідомлення в Telegram: {message_text}")
    except TelegramBadRequest as e:
        if "chat not found" in str(e).lower():
            logging.error(f"Помилка Telegram: Не вдалося знайти чат з ID
{settings.CHAT_ID}. Перевір ID.")
        else:
            logging.error(f"Помилка Telegram API: {e}")
    except Exception as e:
       logging.error(f"Не вдалося надіслати повідомлення в Telegram: {e}")
async def shutdown_telegram_bot():
    """ Закриває сесію бота при зупинці """
    if bot:
        # Перевіряємо наявність сесії перед закриттям
        if bot.session and not bot.session.closed:
            await bot.session.close()
            logging.info("Telegram бот сесію закрито.")
        else:
            logging.info("Telegram бот сесія вже закрита або не існує.")
```

Кінець лістингу 2.6

Клієнтська частина програмного комплексу реалізована як односторінковий веб-додаток (SPA – Single Page Application). У якості технологічного стеку обрано бібліотеку React у середовищі виконання Vite.

Такий вибір забезпечує високу швидкість збірки проекту (HMR – Hot Module Replacement) та оптимізацію кінцевого бандлу.

Архітектура проекту є модульною: логіка розділена на презентаційні компоненти («components»), сторінки-маршрути («pages»), контексти управління станом («contexts») та утиліти взаємодії з API. Структура фронтенд частини зображена на рисунку 2.3.

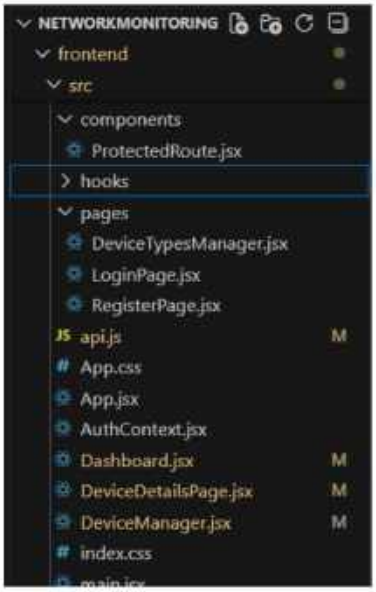

Рисунок 2.3 – Структура фронтенду

Дизайн-система та UX/UI. Для побудови сучасного та адаптивного інтерфейсу використано бібліотеку компонентів «Material UI» (MUI). У файлі «App.jsx» реалізовано глобальну дизайн-систему через компонент «ThemeProvider». Налаштовано темну тему («darkTheme»), яка є стандартом для професійних систем моніторингу, оскільки зменшує навантаження на зір операторів при роботі в затемнених серверних приміщеннях. Для нормалізації стилів браузера та забезпечення кросбраузерної сумісності використано компонент «CssBaseline».

Замість передачі даних про користувача через дерево компонентів реалізовано глобальний провайдер «AuthProvider». Він ініціалізується при старті додатку, перевіряє наявність токена в «localStorage» та автоматично відновлює

сесію користувача. Кастомний хук «useAuth» надає доступ до методів «login/logout» та об'єкта user у будь-якій точці додатку.

Взаємодія з API інкапсульована у модулі «api.js». Реалізовано патерн «Intercepting Filter»:

– створено екземпляр axios з базовою URL-адресою;

– налаштовано перехоплювач запитів (interceptors.request), який автоматично додає HTTP-заголовок «Authorization: Bearer <token>» до кожного вихідного запиту, якщо користувач авторизований. Це гарантує безпеку передачі даних та спрощує код компонентів;

– для розмежування доступу створено компонент-обгортку «ProtectedRoute.jsx». Він перевіряє стан автентифікації перед рендерингом захищених маршрутів («/../devices») і автоматично перенаправляє неавторизованих користувачів на сторінку входу.

Інтерактивний Дашборд («Dashboard.jsx»). Головна панель реалізує візуалізацію стану інфраструктури в реальному часі.

«Drag-and-Drop» – за допомогою бібліотеки «@hello-pangea/dnd» реалізовано можливість групування пристроїв. Використано компоненти «DragDropContext», «Droppable» та «Draggable». Логіка обробки перетягування («onDragEnd») реалізує «оптимістичне оновлення інтерфейсу»: спочатку оновлюється локальний стан React (для миттєвої реакції UI), а потім відправляється асинхронний запит PUT на сервер для збереження змін.

«Polling» (Періодичне опитування) – для актуалізації статусів («Online»/«Offline») використано хук «useEffect» з таймером, який кожні 15 секунд запитує агреговані дані з ендпоінту «/dashboard/status». Вигляд дашборду зображено на рисунку 2.4.

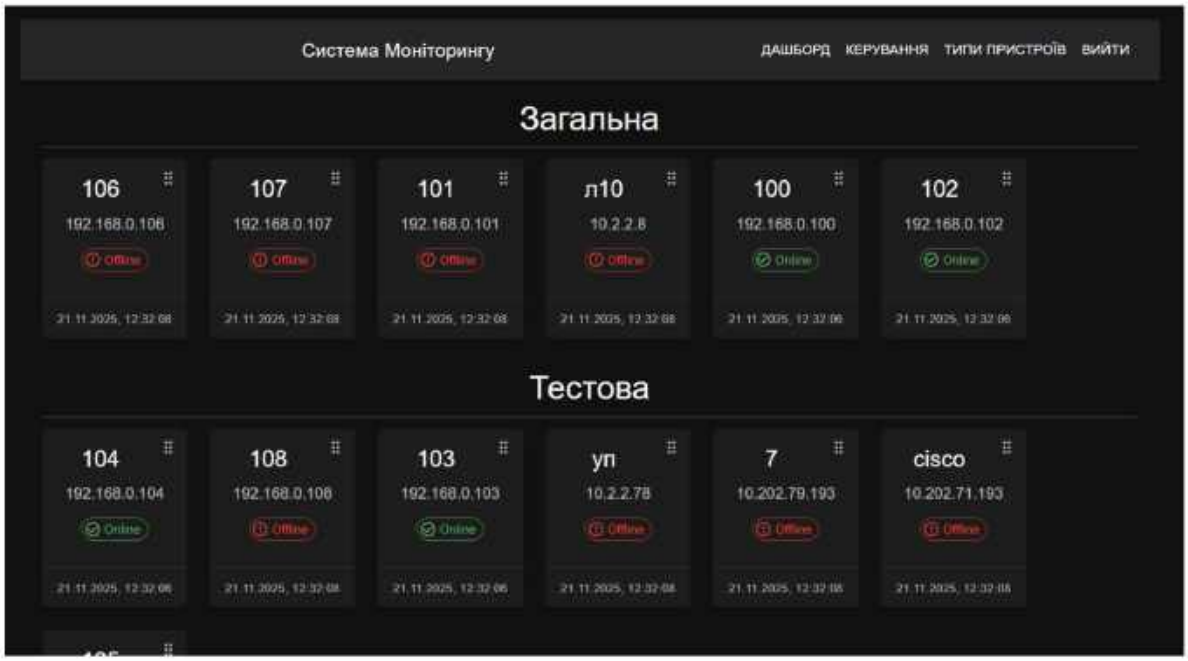

Рисунок 2.4 – Вигляд дашборду

Візуалізація телеметрії («DeviceDetailsPage.jsx») Для відображення історичних даних інтегровано бібліотеку «Recharts». Оскільки сервер повертає «сирі» дані, на клієнті реалізовано шар трансформації даних:

– часові мітки конвертуються у локальний формат;

– булевий статус («is_online») перетворюється у числовий показник (0/1) для відображення на графіку;

– графік доступності реалізовано як «AreaChart», де періоди активності зафарбовуються зеленим градієнтом;

– компонент аналізує ключі в JSON-об'єкті «metrics» і автоматично генерує лінії («LineChart») для кожного параметра (CPU, RAM тощо), призначаючи їм унікальні кольори.

На рисунку 2.5 зображено вигляд графіку сформованого для окремого девайсу.

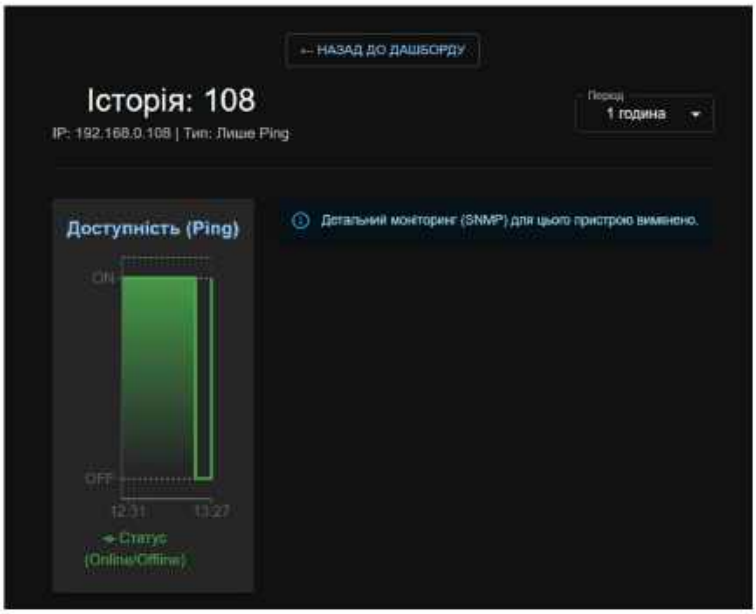

Рисунок 2.5 – Графік доступності для девайсу «108»

Динамічне керування типами налаштовано в файлі «DeviceTypesManager.jsx». Інтерфейс дозволяє адміністратору розширювати можливості моніторингу без втручання в код. Реалізовано динамічні форми для додавання нових OID. На рисунку 2.6 зображено вигляд сторінки «Типи пристроїв».

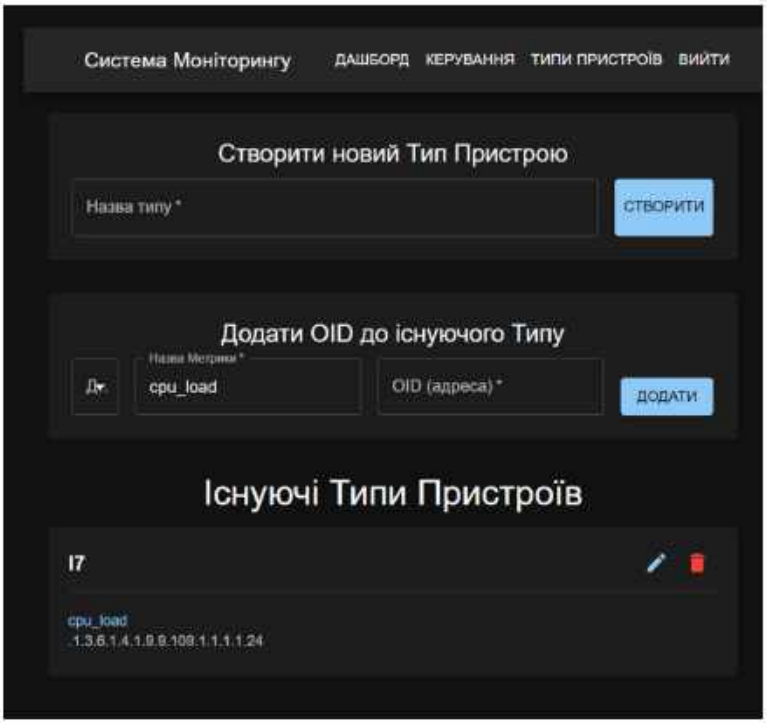

Рисунок 2.6 – Екран налаштування типів пристроїв

Для покращення UX (User Experience), інтегровано бібліотеку «react-toastify», яка відображає спливаючі повідомлення про успішне збереження налаштувань або помилки валідації. На рисунку 2.7 зображено сторінку керування пристроями з активним сповіщенням про успішне видалення пристрою.

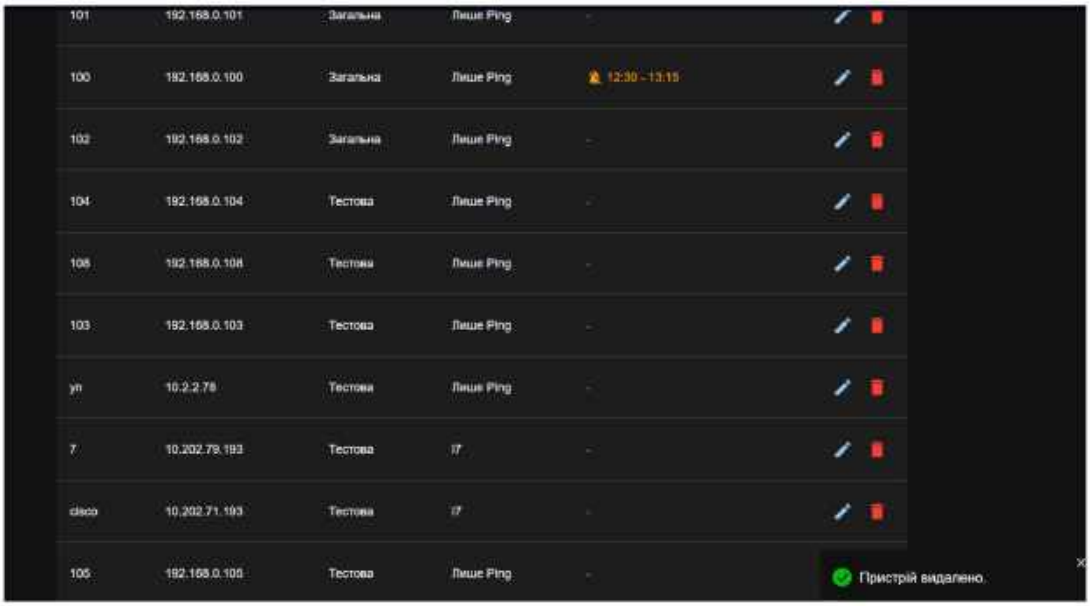

Рисунок 2.7 – Сторінка «Керування» з сповіщенням на екрані

Для забезпечення ізольованості процесів, передбачуваності середовища виконання та спрощення масштабування, програмний комплекс було контейнеризовано за допомогою технології Docker. Цей підхід дозволяє упакувати застосунок разом з усіма його залежностями в єдиний артефакт (образ), гарантуючи принцип «Build once, run anywhere».

Стратегія побудови Docker-образу Процес збірки описано у файлі «Dockerfile». Зміст цих файлів наведено нижче в лістингу 2.7.

Лістинг 2.7 – Вміст файлів «Dockerfile»

```
    #Dockerfile backend
    FROM python:3.10-slim
    RUN apt-get update && apt-get install -y iputils-ping && rm -rf
/var/lib/apt/lists/*
    WORKDIR /app
```

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]

#Dockerfile frontend
FROM node:20-alpine AS builder
WORKDIR /app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build
FROM nginx:stable-alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
CMD ["nginx", "-g", "daemon off;"]
```

Кінець лістингу 2.7

В якості базового образу обрано «python:3.10-slim». Це офіційна, полегшена версія на базі «Debian», з якої видалено більшість стандартних пакетів Linux, не потрібних для виконання Python-коду. Вибір «слім-версії» дозволив:

– зменшити розмір фінального образу, що прискорює його завантаження на сервер;

– зменшити «поверхню атаки» (attack surface) шляхом мінімізації кількості встановленого ПЗ.

У «Dockerfile» реалізовано встановлення системної утиліти «iputils-ping». Це критично важливо для роботи модуля «monitoring_worker.py», оскільки перевірка доступності вузлів (ICMP) виконується через виклик системного процесу, а базовий образ не містить цієї утиліти за замовчуванням. Встановлення Python-бібліотек виконується з файлу «requirements.txt» з використанням прапора «--no-cache-dir». Це запобігає збереженню кешу «pip» всередині шару образу, що додатково зменшує його розмір. Також застосовано техніку кешування шарів Docker: копіювання файлу залежностей і їх встановлення відбувається до копіювання основного коду проекту. Це дозволяє Docker не перезбирати шар із бібліотеками при кожній зміні у вихідному коді програми, значно прискорюючи процес CI/CD.

Налаштування параметрів підключення до бази даних та зовнішніх сервісів здійснюється через змінні середовища, що відповідає стандарту «Twelve-Factor App». У конфігураційному файлі «.env» для підключення до бази даних вказано хост «db». Це свідчить про використання внутрішньої мережі Docker (Docker Network), де контейнери звертаються один до одного за іменами сервісів, без необхідності відкривати порти бази даних у зовнішню мережу.

В якості точки входу (CMD) налаштовано запуск ASGI-сервера «uvicorn» на хості «0.0.0.0», що робить API доступним ззовні контейнера.

У результаті виконання даного етапу було отримано повністю працездатний, переносимий програмний комплекс. Використання контейнеризації дозволяє розгортати систему як на локальних серверах підприємства, так і в хмарних середовищах (AWS, Azure) без необхідності складного налаштування оточення.

РОЗДІЛ 3
ЕКСПЕРЕМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕЗУЛЬТАТИВНОСТІ СИСТЕМИ АВТОМАТИЧНОГО МОНІТОРИНГУ СПРАВНОСТІ МЕРЕЖЕВОГО ОБЛАДНАННЯ У ЛОКАЛЬНІЙ МЕРЕЖІ

3.1 Методика проведення дослідження

Метою експериментального дослідження є кількісне та функціональне підтвердження переваг розробленого програмного комплексу перед традиційними синхронними методами моніторингу. Основна гіпотеза дослідження полягає в тому, що використання асинхронної архітектури «Python asyncio» для мережевого опитування значно підвищує швидкість і масштабованість системи в умовах високого навантаження.

Усі компоненти системи (Backend, Frontend, PostgreSQL + TimescaleDB) розгорнуті в ізольованому контейнеризованому середовищі Docker, що забезпечує стабільність та відтворюваність результатів. Для проведення тестів використано наявне обладнання.

Таблиця 3.1 – Опис етапів тестування проекту

№	Назва метрики	Опис та цільове призначення
1	Час повного циклу опитування («Full Polling Cycle Time»)	Загальний час, необхідний для виконання ICMP-перевірки та SNMP-запитів до пристрою відповідну кількість разів. Використовується для порівняння синхронного та асинхронного підходів
2	Надійність функціоналу DND («Don't Disturb»)	Перевірка коректності придушення сповіщень у Telegram при виявленні збою в період «тихих годин»
3	Коректність «Edge Triggering»	Оцінка надійності механізму придушення хибних повторних сповіщень (Alert Spamming), коли статус пристрою залишається незмінним («Offline»).

Дослідження проводилося у три основні етапи.

Перший етап передбачає кількісне порівняння продуктивності (Асинхронна та Синхронна модель).

Методика полягає у проведенні «бенчмарку» шляхом багаторазового вимірювання часу виконання двох ідентичних за логікою, але різних за архітектурою (блокуюча та неблокуюча) функцій опитування.

Створюється функція, яка послідовно виконує мережевий запит до пристрою певну кількість разів. Це моделює роботу традиційних систем. Наступним кроком використовується функція «run_monitoring_cycle» з ядром «asyncio.gather», що виконує 100 запитів паралельно.

Для забезпечення достовірності кожен цикл виконується 10 разів. Для вимірювання часу використовується високоточний системний інструмент «time.perf_counter()». Код основної функції модуля «benchmark.py» наведено нижче в лістингу 2.8.

Лістинг 2.8 – Код основної функції модуля «benchmark.py»

```
    def run_sync_test():
         print(f"\n[SYNC] Починаємо опитування {REQUESTS_COUNT} разів
послідовно...")
        start_time = time.perf_counter()

        for i in range(REQUESTS_COUNT):
            iterator = syncGetCmd(
                SyncSnmpEngine(),
                CommunityData(COMMUNITY, mpModel=1),
                  SyncUdpTransportTarget((TARGET_IP, 161), timeout=1.0,
retries=0),
                ContextData(),
                ObjectType(ObjectIdentity(OID))
            )
                errorIndication, errorStatus, errorIndex, varBinds =
next(iterator)
            if errorIndication:
                pass
        end_time = time.perf_counter()
        return end_time - start_time

    # --- 2. АСИНХРОННИЙ ТЕСТ ---
    async def single_async_request():
        try:
```

```
            errorIndication, errorStatus, errorIndex, varBinds = await
asyncGetCmd(
                AsyncSnmpEngine(),
                CommunityData(COMMUNITY, mpModel=1),
                    UdpTransportTarget((TARGET_IP, 161), timeout=1.0,
retries=0),
                ContextData(),
                ObjectType(ObjectIdentity(OID))
            )
        except Exception:
            pass

    async def run_async_test():
         print(f"\n[ASYNC] Починаємо опитування {REQUESTS_COUNT} разів
паралельно...")
        start_time = time.perf_counter()
        tasks = [single_async_request() for _ in range(REQUESTS_COUNT)]
        await asyncio.gather(*tasks)
        end_time = time.perf_counter()
        return end_time - start_time
```

Кінець лістингу 2.8

Другий етап передбачає функціональну валідацію збору даних та виведення.

Методика спрямована на підтвердження коректності збору гетерогенних даних (SNMP) та їхнього відображення в інтерфейсі.

Для підтвердження можливості збору детальних метрик було використано реальний мережевий пристрій «Cisco 881» із налаштованим доступом по SNMPv2c. Пристрій «Cisco 881» додається до системи з відповідними OID для моніторингу завантаження CPU та пам'яті.

Дані аналізуються у веб-інтерфейсі (фронтенді) на сторінці деталей пристрою.

Третій етап – це валідація інтелектуальних механізмів сповіщень.

Методика перевіряє надійність механізмів, які запобігають надлишковій кількості сповіщень.

Тому окремо проводиться тест DND де для тестового пристрою встановлюється інтервал «тихих годин» (DND), що перетинає північ. Моніторинг проводиться у час, що потрапляє в DND-інтервал. Пристрій вимикається

(імітація збою). Система повинна зареєструвати статус «Offline» у БД, але не надіслати сповіщення в Telegram.

А також тест на «Edge Triggering») коли пристрій вимикається і система фіксує статус «Offline» (перший цикл). Далі система повинна надіслати одне сповіщення про збій. Протягом наступних 5 циклів моніторингу, поки пристрій залишається вимкненим, жодних додаткових сповіщень про цей пристрій не повинно надходити.

3.2 Обробка та аналіз отриманих результатів

Обробка та аналіз отриманих результатів здійснюється шляхом порівняння ключових метрик, отриманих на першому етапі, з теоретичними показниками синхронного підходу.

Проведено порівняльний аналіз часу циклу опитування. Для підтвердження висунутої гіпотези про переваги асинхронного підходу використаємо метрику «Час повного циклу опитування». Базою для порівняння є показники синхронної моделі, яка представляє традиційні системи моніторингу, де мережеві запити є блокуючими.

Таблиця 3.2 – Результати порівняння продуктивності опитування

№ п/п	Кількість запитів (N)	Синхронний режим, с	Асинхронний режим, с	Коефіцієнт прискорення
1	20	22,31	3,23	6,9
2	100	110,38	11,02	10,0
3	1000	1107,26	106,16	10,4

Для первинної оцінки ефективності розробленого підходу було проведено тестування з невеликою кількістю паралельних запитів. На рисунку 3.1 наведено результати «бенчмарку», який виконував 20 послідовних (синхронний режим) та 20 паралельних (асинхронний режим) SNMP-запитів до цільового пристрою.

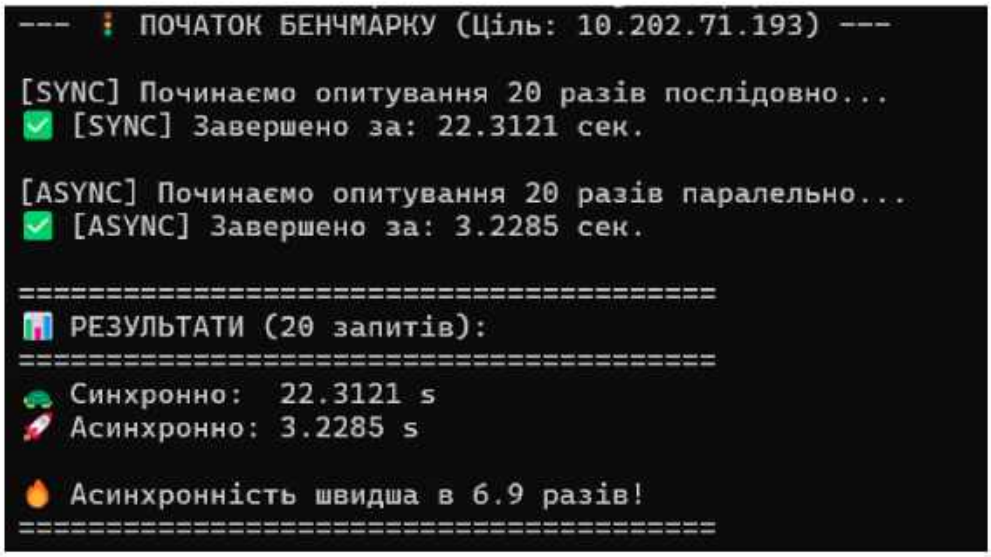

Рисунок 3.1 – Результати тестування для 20 ітерацій

Для моделювання умов середнього навантаження (типового для локальної мережі СМБ) кількість ітерацій була збільшена до 100. На рисунку 3.2 відображено, як співвідносяться часи виконання 100 послідовних та 100 паралельних запитів. Саме цей тест є найбільш репрезентативним для оцінки реальної продуктивності системи.

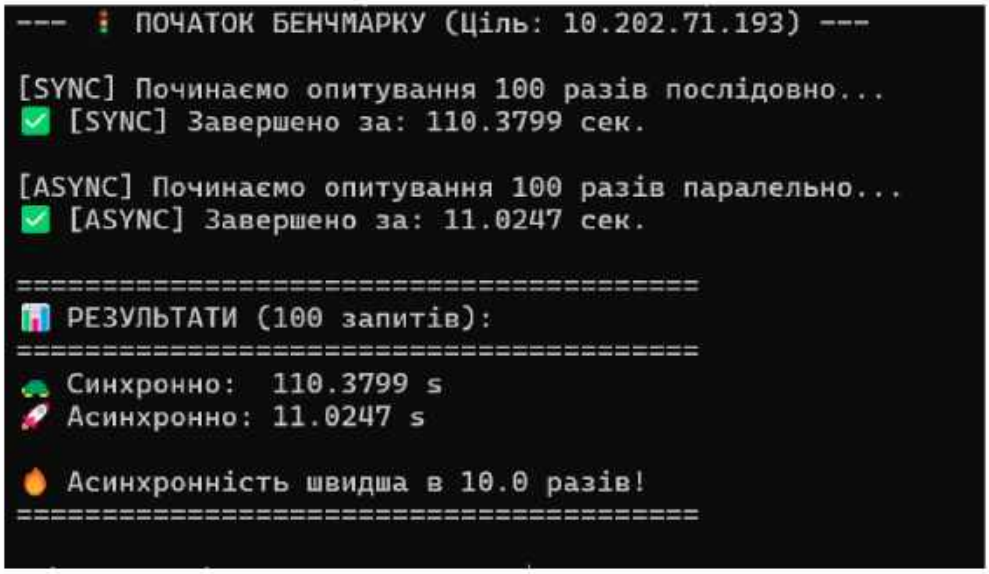

Рисунок 3.2 – Результати тестування для 100 ітерацій

З метою перевірки масштабованості системи в умовах високого навантаження (понад 1000 запитів) було проведено третій тест. На рисунку 3.3 продемонстровано результати, отримані для 1000 ітерацій, що дозволяє оцінити лінійність прискорення та ефективність асинхронного механізму «asyncio.gather» при значному зростанні кількості одночасних операцій вводу/виводу.

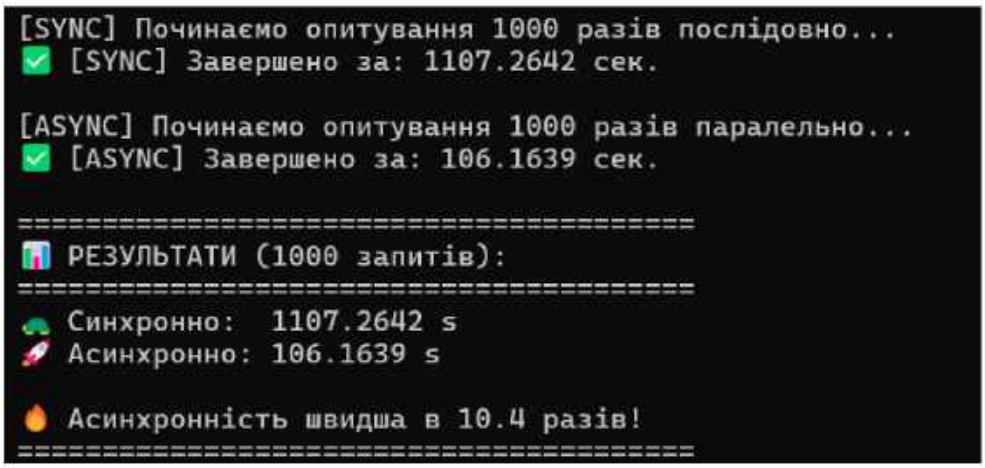

Рисунок 3.3 – Результати тестування для 1000 ітерацій

Тестування показує, що застосування неблокуючого вводу-виводу (I/O) на базі «asyncio» дозволило скоротити час повного циклу опитування. Цей результат підтверджує, що розроблена система може забезпечити моніторинг із високою гранулярністю (кожні кілька секунд) навіть для великих інфраструктур, що є недосяжним для блокуючих синхронних рішень. Це забезпечує можливість переходу від реактивного до проактивного моніторингу.

Оцінка інтеграції з реальним обладнанням (Cisco 881).

Факт успішного збору та графічного відображення метрик CPU та RAM з маршрутизатора «Cisco 881» підтверджує коректність реалізації:

– модуля SNMP-опитування (pysnmp) та його здатності обробляти відповіді з реального мережевого обладнання;

– гнучкості моделі даних, яка дозволяє динамічно додавати нові OID для різних типів пристроїв;

– коректної роботи фронтенд-модуля візуалізації (Recharts) для побудови графіків часових рядів.

Результати тестування продемонстровано на рисунку 3.4.

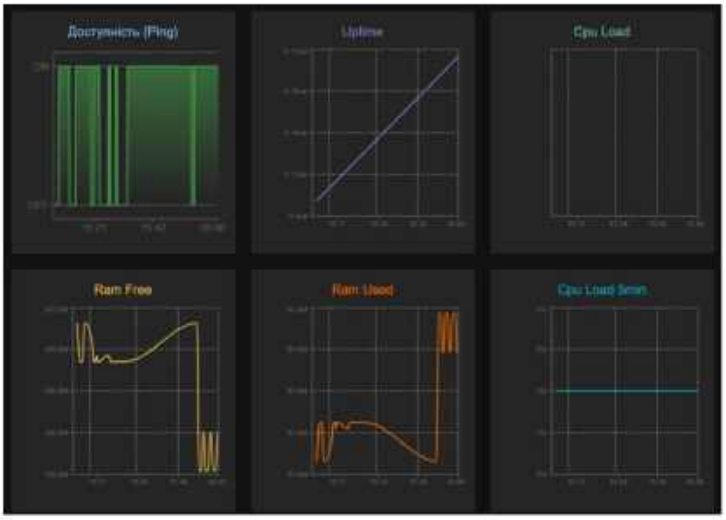

Рисунок 3.4 – Графіки OID метрик девайсу «Cisco 881»

Валідація інтелектуальних механізмів сповіщень.

Успішне вимикання сповіщення у часовому інтервалі 22:00–07:00 підтверджує коректність удосконаленого алгоритму, який враховує перехід доби. Це є ключовим показником надійності системи в режимі 24/7 та критично важливо для зменшення навантаження на черговий персонал. Результат продемонстровано на рисунку 3.5.

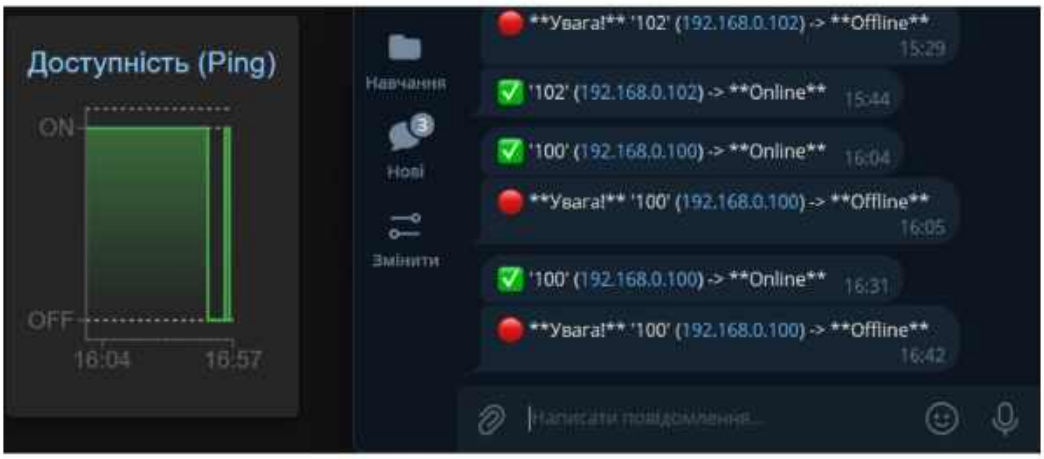

Рисунок 3.5 – Результат тесту на придушення спаму

Факт надходження лише одного сповіщення про збій, незважаючи на багаторазове повторне опитування вимкненого пристрою, підтверджує, що механізм придушення спаму працює коректно. Це підвищує інформативну цінність сповіщень і мінімізує «тривожну втому» адміністраторів. Результат продемонстровано на рисунку 3.6.

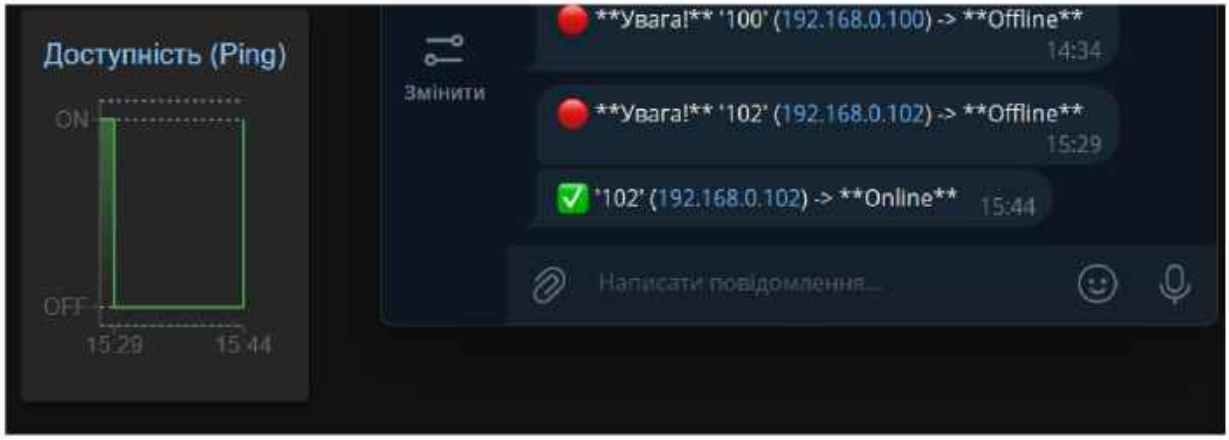

Рисунок 3.6 – Результат тесту на придушення спаму

Отримані результати та розроблений програмний комплекс мають високу практичну цінність та готові до впровадження.

Особливості впровадження:

– застосування технології Docker забезпечує швидке та надійне розгортання системи в будь-якому середовищі (фізичний сервер, приватна чи публічна хмара) без необхідності складного налаштування залежностей;

– гнучка модель даних з використанням JSONB дозволяє легко розширювати функціонал системи для збору нових, нестандартних метрик (наприклад, показники температури, стану джерела живлення, лічильники помилок CRC) без внесення змін до схеми бази даних;

– завдяки підтвердженому коефіцієнту прискорення, система може бути впроваджена в компаніях малого та середнього бізнесу (СМБ) як економічна альтернатива дорогим комерційним рішенням, забезпечуючи при цьому рівень моніторингу, близький до AIOps-платформ.

Розроблена система може бути впроваджена для:

– проактивного виявлення несправностей мережевих пристроїв (комутатори, маршрутизатори, точки доступу) у режимі реального часу;

– візуалізації історичних даних для планування потужностей та виявлення прихованих проблем продуктивності;

– автоматизованого сповіщення про критичні події через Telegram, інтегроване в робочий процес ІТ-відділу.

ВИСНОВКИ

В рамках кваліфікаційної роботи магістра була успішно розроблена та досліджена система автоматичного моніторингу справності мережевого обладнання, що базується на високопродуктивній асинхронній архітектурі.

Проведена робота дозволила досягти поставленої мети – створення ефективної системи моніторингу, яка поєднує високу швидкість збору даних із надійними механізмами сповіщень.

У процесі виконання роботи були повністю вирішені всі поставлені завдання:

– проаналізовано та обґрунтовано архітектурний та технологічний стек, який включає Python (FastAPI) для високопродуктивного асинхронного бекенду, React для динамічного фронтенду та PostgreSQL з розширенням TimescaleDB для оптимізованого зберігання часових рядів;

– розроблено модель даних з оптимізованим сховищем для часових рядів. Було впроваджено використання JSONB для гнучкого зберігання метрик та композитних ключів для оптимізації запису в таблицю «StatusHistory»;

– спроектовано та реалізовано високопродуктивне асинхронне ядро опитування («monitoring_worker») на базі «asyncio.gather». Експериментально підтверджено, що цей підхід забезпечує прискорення циклу опитування у 10 і більше разів порівняно із синхронною моделлю;

– реалізовано механізми інтелектуальної обробки сповіщень. Удосконалено алгоритм логіки «тихих годин» (DND), який коректно обробляє часові інтервали, що перетинають північ. Впроваджено механізм «Edge Triggering» для придушення дублюючих (хибних) сповіщень, що значно підвищує інформативну цінність системи;

– створено RESTful API (FastAPI) для керування системою з безпечною автентифікацією. Реалізовано Role-Based Access Control (RBAC) та впроваджено автоматичну генерацію документації Swagger UI;

– розроблено інтерактивний веб-дашборд (React), який забезпечує візуалізацію метрик у реальному часі, «Drag-and-Drop» групування пристроїв та графічне відображення історичних даних (зокрема, з реального пристрою «Cisco 881»);

– забезпечено високу переносимість системи шляхом повної контейнеризації усіх компонентів (backend, frontend, БД) за допомогою технології Docker, що гарантує швидке та надійне розгортання;

– проведено експериментальне дослідження результативності розробленої системи. Кількісні тести підтвердили високу масштабованість асинхронного ядра, а функціональні тести – надійність роботи інтелектуальних механізмів сповіщень.

Результатом виконаної кваліфікаційної роботи магістра є повністю функціональний програмний комплекс, який репрезентує сучасний підхід до моніторингу мережевої інфраструктури. Система є готовим продуктом, що поєднує переваги відкритого програмного забезпечення з високою продуктивністю та інтелектуальними функціями, необхідними для проактивного управління мережею. Проєкт має значні можливості для подальшого масштабування та майбутнього впровадження механізмів AIOps.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Gartner Inc. Market Guide for AIOps Platforms. Gartner Research. URL: https://www.gartner.com/en/research/magic-quadrant (дата звернення: 29.07.2025).

2. Elradi M. D. Prometheus & Grafana: A Metrics-focused Monitoring Stack. Journal of Computer Allied Intelligence. 2025. URL: https://surl.li/fbaacd (дата звернення: 05.08.2025).

3. DiDio L. ITIC 2021 Hourly Cost of Downtime Survey Results. Information Technology Intelligence Consulting. URL: https://surl.li/jrgbyq (дата звернення: 05.08.2025).

4. Андрущак І. Є., Шмаровоз С. В. Забезпечення надійності моніторингових систем у корпоративних мережах: типові проблеми і рішення. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. 2025. № 59. С. 37-42.

5. Андрущак І. Є., Шмаровоз С. В. Застосування SNMP протоколу для віддаленого контролю працездатності мережевої інфраструктури. Комп'ютерно-інтегровані технології: освіта, наука, виробництво. 2025. № 61. С. 236-241.

6. Song H. et al. Network Telemetry Framework. IETF Request for Comments 9232. URL: https://www.rfc-editor.org/rfc/rfc9232.html (дата звернення: 24.07.2025).

7. IETF. RFC 9010-9016: Simple Network Management Protocol (SNMP) Version 3. Internet Engineering Task Force. 2021. URL: https://datatracker.ietf.org/doc/rfc9016 (дата звернення: 07.09.2025).

8. Introduction to NETCONF and RESTCONF Cisco DevNet. Cisco Systems. URL: https://developer.cisco.com/docs/ios-xe/#!introduction-to-netconf-restconf (дата звернення: 07.09.2025).

9. gRPC Authors. Introduction to gRPC: Core concepts, architecture and lifecycle. gRPC.io. URL: https://grpc.io/docs/what-is-grpc/introduction/ (дата звернення: 08.09.2025).

10. gNMI gRPC Network Management Interface. OpenConfig. URL: https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md (дата звернення: 08.09.2025).

11. Zabbix SIA. What's New in Zabbix 7.0 LTS. Zabbix Official Blog. URL: https://www.zabbix.com/whats_new_7_0 (дата звернення: 09.10.2025).

12. Prometheus vs Zabbix: відмінне та подібне цих систем моніторингу. IT Education Center. https://itedu.center/ua/blog/comparisons/prometheus_vs_zabbix/ (дата звернення: 09.09.2025).

13. Stucka T. Comparative Analysis of Python Web Frameworks URL: https://is.muni.cz/th/ea6cj/Diploma_Thesis_Stucka.pdf (дата звернення: 13.09.2025).

14. Django Documentation. Django Software Foundation. URL: https://docs.djangoproject.com/en/stable/ (дата звернення: 13.09.2025).

15. Flask Documentation. Pallets Projects. URL: https://flask.palletsprojects.com/ (дата звернення: 13.09.2025).

16. FastAPI Official Documentation. URL: https://fastapi.tiangolo.com/ (дата звернення: 13.09.2025).

17. Bednarz B., Miłosz M. Benchmarking the performance of Python web frameworks. URL: https://surl.li/ftnsbq (дата звернення: 16.09.2025).

18. React Official Documentation. URL: https://react.dev/ (дата звернення: 13.09.2025).

19. State of JS 2022: The Annual JavaScript Survey. URL: https://2022.stateofjs.com/en-US/libraries/front-end-frameworks (дата звернення: 13.09.2025).

20. TimescaleDB vs. InfluxDB: Purpose Built Differently for Time-Series Data Timescale Blog. URL: https://surl.li/cocgpw/(дата звернення: 19.09.2025).

21. What is a Container? Docker Resources. Docker Inc. URL: https://www.docker.com/resources/what-container/ (дата звернення: 06.10.2025).

22. What is Git? Atlassian Git Tutorial. Atlassian. URL: https://www.atlassian.com/git/tutorials/what-is-git (дата звернення: 06.10.2025).