Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

# КВАЛІФІКАЦІЙНА РОБОТА

Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP
Назва теми

Рівень вищої освіти *перший (бакалаврський)*

Галузь знань *12 «Інформаційні технології»*
Шифр, назва

Спеціальність *123 «Комп'ютерна інженерія»*
Шифр, назва

Освітня програма *«Комп'ютерна інженерія та програмування»*
Назва

Шифр КвРКІ 2301116.23.01.01 ПЗ

Виконав здобувач III курсу, група КІ2с-23-1 — Підпис — Кірил МЕЛЬНИК (Ініціали, прізвище)

Керівник — Науковий ступінь, учене звання — Підпис — Дмитро МЕДЗАТИЙ (Ініціали, прізвище)

Нормоконтролер — Науковий ступінь, учене звання — Підпис — Сергій ЛИСЕНКО (Ініціали, прізвище)

До захисту допускаю:
завідувач кафедри КІІС
«01» червня 2026 р. — Підпис — Ольга ПАВЛОВА (Ініціали, прізвище)
дата

Хмельницький 2026

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ

Рівень вищої освіти ПЕРШИЙ (БАКАЛАВРСЬКИЙ)

Галузь знань 12 ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ

Спеціальність 123 КОМП'ЮТЕРНА ІНЖЕНЕРІЯ

Освітня програма «КОМП'ЮТЕРНА ІНЖЕНЕРІЯ ТА ПРОГРАМУВАННЯ»

ЗАТВЕРДЖУЮ
Завідувачка кафедри КІІС
Ольга ПАВЛОВА

" 10 " 01 2026 р.

**ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ**

Мельнику Кирилу Валерійовичу
Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP
Керівник проекту (роботи) Медзатий Дмитро Миколайовч , к.т.н., доцент
Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 20.01.2026 р. № 7

2. Термін подання здобувачем роботи на кафедру 01.06.2026 р.

3. Вихідні дані до роботи Завдання на кваліфікаційну роботу

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP

Проєктування системи обробки інформації у система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP

Програмно-апаратна реалізація системи моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

Архітектура ПЗ проєкту

Архітектура ПЗ для кіберфізичної системи

Апаратне забезпечення проєкту

6. Консультанти розділів кваліфікаційної роботи

| Розділ | Прізвище, ініціали та посада консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийн |
| | | | |
| | | | |

7. Дата видачі завдання «_10_»_____01_____2026 р.

## КАЛЕНДАРНИЙ ПЛАН

| №з/п | Назва етапів (розділів) дипломного проекту (роботи) | Термін виконання етапів проекту (роботи) | Приміт |
|---|---|---|---|
| 1 | Вибір напряму дослідження та узгодження тематики кваліфікаційної роботи з керівником | 10.01.2026 | виконан |
| 2 | Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження | 01.02.2026 | виконан |
| 3 | Робота над розділом 1 – дослідження предметної області та постановка задачі | 01.03.2026 | виконан |
| 4 | Робота над розділом 2 – вибір компонентів для проєктування вбудованої системи моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP | 01.04.2026 | виконан |
| 5 | Робота над розділом 3 – проєктування вбудованої системи моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP | 29.04.2026 | виконан |
| 6 | Оформлення пояснювальної записки згідно вимог | 25.05.2026 | виконан |
| 7 | Попередній захист ВКР | 26.05.2026 | виконан |
| 8 | Захист ВКР на засіданні ЕК | Червень 2026 року | |

Здобувач _____ Кірил МЕЛЬНИК
Підпис Ім'я, ПРІЗВИЩЕ

Керівник кваліфікаційної роботи _____ Дмитро МЕДЗАТИЙ
Підпис Ім'я, ПРІЗВИЩЕ

| № рядка | формат | Позначення | Найменування | Кіл. листів | № екз | Примітка |
|---|---|---|---|---|---|---|
| | | | Текстові документи | | | |
| 1 | | КвРКІ 2301116.23.01.01 ПЗ | Пояснювальна записка | 57 | | |
| | | | | | | |
| | | | Графічні матеріали | | | |
| 2 | | КвРКІ 2301116.23.01.01Е8 | Архітектура програмного | 1 | | |
| | | | забезпечення проєкту | | | |
| 3 | | | | | | |
| | | КвРКІ 2301116.23.01.01Е8 | Блок-схеми алгоритмів | 1 | | |
| | | | програмного забезпечення | | | |

| Зм | Арк | № докум | Підпис | Дата | КвРКІ 2301116.23.01.01 ПЗ | | |
|---|---|---|---|---|---|---|---|
| Розробив | | Мельник | | | Відомість проекту | Літера | Аркуш | Аркушів |
| Перевір. | | Медзатий | | 29.05 | | У | 1 | 1 |
| | | | | | | ХНУ, КІ2с-23-1 | | |
| Н. контр. | | Лисенко | | | | | | |
| Затв. | | Павлова | | 01.06 | | | | |

# АНОТАЦІЯ

Тема кваліфікаційної роботи: «Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP».

Автор роботи: Кірил МЕЛЬНИК.

Керівник роботи: Дмитро МЕДЗАТИЙ.

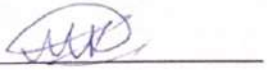
Пояснювальна записка: 57 с., 4 рис., 6 табл., 3 дод., 50 джерел.

Графічна частина: 2 креслення.

БАЗА ДАНИХ, ВБУДОВАНА СИСТЕМА, МОНІТОРИНГ, ПРОТОКОЛ SNMP, RASPBERRY PI, САМООРГАНІЗАЦІЯ, СЕРВЕРНЕ ОБЛАДНАННЯ.

Кваліфікаційна робота бакалавра присвячена розробці вбудованої системи моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP. Актуальність теми зумовлена зростанням вимог до надійності серверної інфраструктури підприємств: своєчасне виявлення перевантаження ресурсів та збоїв обладнання дозволяє попереджати аварійні ситуації і знижувати ризики простоїв. Протокол SNMPv2c забезпечує спостереження за гетерогенним парком обладнання різних виробників без встановлення стороннього агентського програмного забезпечення.

Метою роботи є проєктування та реалізація комплексу для збору, зберігання та візуалізації метрик серверного обладнання з автоматичним сповіщенням про інциденти. Для її досягнення виконано аналіз підходів до побудови систем моніторингу; обрано стек Python 3, Flask, SQLAlchemy та pysnmp; розроблено модульну N-tier архітектуру; спроєктовано алгоритми паралельного опитування через asyncio, порогової перевірки метрик з дедублікацією сповіщень та скінченного автомата стану пристрою; реалізовано веб-інтерфейс з часовими графіками та REST API.



Підпис здобувача

30.05.2026

Дата

## ЗМІСТ

# ВСТУП

У сучасних умовах цифровізації стабільна робота серверної інфраструктури є критично важливою для функціонування будь-якого підприємства. Зростання обсягів даних та складності мережевих архітектур вимагає безперервного контролю параметрів середовища та апаратних ресурсів. Вихід з ладу серверного обладнання через перегрів, збої в системі живлення або критичні навантаження може призвести до значних фінансових втрат та витоку конфіденційної інформації. Тому розробка надійних систем моніторингу, здатних оперативно реагувати на відхилення у роботі обладнання, залишається пріоритетним завданням для системних адміністраторів та IT-інженерів.

Традиційні промислові рішення для моніторингу часто є високовартісними або вимагають значних ресурсів для впровадження та підтримки. Використання мікрокомп'ютерів, зокрема платформи Raspberry Pi, дозволяє створити гнучку, енергоефективну та бюджетну вбудовану систему, яка за функціональністю не поступається комерційним аналогам. Завдяки наявності портів GPIO та підтримці широкого спектру датчиків, Raspberry Pi стає ідеальною базою для збору телеметрії безпосередньо в місці розміщення серверних стійок.

Ключовим аспектом ефективної інтеграції такої системи в існуючу IT-інфраструктуру є використання стандартизованих протоколів передачі даних. Протокол SNMP (Simple Network Management Protocol) є галузевим стандартом, який забезпечує сумісність моніторингового рішення з більшістю мережевих пристроїв та систем управління (NMS). Використання SNMP дозволяє уніфікувати процес збору даних про стан процесора, температуру, вологість та інші критичні показники, забезпечуючи зручну візуалізацію та сповіщення про аварійні ситуації в режимі реального часу. Крім того, такий підхід забезпечує високу масштабованість, дозволяючи легко додавати нові вузли моніторингу в міру розширення серверного парку.

# 1 ВБУДОВАНА СИСТЕМА МОНІТОРИНГУ СТАНУ СЕРВЕРНОГО ОБЛАДНАННЯ ТА ПОСТАНОВКА ЗАДАЧІ ЩОДО ЇЇ РОЗРОБКИ НА БАЗІ RASPBERRY PI

## 1.1 Аналіз предметної області і виявлення наявних проблем і завдань

Сучасна IT-інфраструктура характеризується постійним зростанням щіль ності розміщення обчислювальних потужностей, що висуває жорсткі вимоги до середовища експлуатації серверного обладнання. Стабільність роботи серверів безпосередньо залежить від підтримання оптимальних фізичних показників, так их як температура, вологість та якість електроживлення. Навіть незначне відхил ення від норми може призвести до деградації напівпровідникових компонентів, раптових відмов або скорочення терміну служби дороговартісного обладнання. У зв'язку з цим, моніторинг стану серверної кімнати перестає бути допоміжною функцією і стає критично важливою складовою забезпечення безперервності бі знес-процесів.

Аналіз предметної області свідчить про наявність значного розриву між в нутрішніми засобами самодіагностики серверів та зовнішніми системами контр олю середовища. Більшість сучасних серверів оснащені інтерфейсами IPMI або iDRAC, які дозволяють відстежувати внутрішню температуру процесора чи шв идкість обертання вентиляторів, проте вони «сліпі» до зовнішніх чинників, таки х як протікання води, задимлення або несанкціоноване відкриття серверної шаф и. Це створює потребу у впровадженні незалежних вбудованих систем, які здат ні агрегувати дані з різноманітних зовнішніх датчиків та інтегрувати їх у загаль ну систему управління мережею.

Вибір протоколу SNMP (Simple Network Management Protocol) як бази для передачі даних обумовлений його статусом галузевого стандарту. Незважаючи на появу новіших протоколів, таких як MQTT або HTTP REST API, SNMP зали шається найбільш сумісним рішенням для корпоративного сектору. Він дозволя є стандартизувати представлення даних через ієрархічну структуру MIB-об'єкті

в, що забезпечує легку інтеграцію вбудованої системи в існуючі програмні ком плекси моніторингу типу Zabbix або Nagios. Використання SNMP v3 додатково вирішує питання безпеки, забезпечуючи автентифікацію та шифрування трафік у, що є критичним для промислових та корпоративних мереж.

Застосування платформи Raspberry Pi у якості апаратної основи вбудован ої системи є обґрунтованим з точки зору балансу між обчислювальною потужні стю та гнучкістю інтерфейсів. На відміну від мікроконтролерів класу Arduino а бо ESP32, Raspberry Pi працює під управлінням повноцінної операційної систем и сімейства Linux, що дозволяє реалізувати складні алгоритми обробки даних, л окальне кешування при розриві зв'язку та підтримку повноцінного стеку проток олів TCP/IP. Наявність портів GPIO та підтримка протоколів I2C, SPI та 1-Wire дозволяє підключати широкий спектр цифрових сенсорів без використання дод аткових перетворювачів, що спрощує архітектуру пристрою та підвищує його н адійність.

Однією з ключових проблем існуючих комерційних рішень є їхня висока вартість та закритість екосистеми. Пропрієтарні системи часто вимагають прид бання дорогих ліцензій на програмне забезпечення або підключення лише спец ифічних датчиків від того ж виробника. Це обмежує можливості масштабуванн я та адаптації системи під конкретні умови серверного приміщення. Вбудована система на базі Raspberry Pi вирішує цю проблему завдяки використанню відкр итого коду та можливості програмної корекції логіки роботи «на льоту», що ро бить її економічно привабливою для малого та середнього бізнесу.

Процес моніторингу зазвичай включає збір даних про температуру та вол огість повітря в холодних і гарячих коридорах серверної. Висока температура п рискорює хімічні процеси в конденсаторах блоків живлення, а низька вологість сприяє накопиченню статичного заряду, що загрожує пробоєм мікросхем. Засто сування цифрових датчиків, таких як DHT22 або BME280, підключених до Raspberry Pi, забезпечує високу точність вимірювань та стабільність показань п ротягом тривалого часу. Програмна частина системи повинна не лише зчитуват

и ці значення, а й порівнювати їх із заданими порогами (thresholds) для негайної генерації SNMP-трапів у разі аварійних ситуацій.

Окрему увагу при аналізі слід приділити питанню енергоефективності та автономності самої системи моніторингу. Оскільки пристрій має контролювати стан серверів навіть у випадку збою основного живлення, актуальним є завданн я розробки підсистеми резервного живлення або використання технології PoE (Power over Ethernet). Це дозволяє системі надіслати останнє повідомлення про зникнення напруги в мережі, що є критично важливою інформацією для систем ного адміністратора. Raspberry Pi має відповідні модулі розширення, що робить таку реалізацію технічно можливою та компактною.

Важливим аспектом проектування є організація взаємодії між апаратною частиною та SNMP-агентом. Агент повинен динамічно оновлювати значення в базі MIB на основі даних, отриманих від фонових процесів опитування сенсорів. Це вимагає розробки багаторівневої програмної архітектури, де драйвери датчи ків ізольовані від логіки мережевого протоколу. Такий підхід забезпечує модул ьність системи: при необхідності додавання нового типу сенсора (наприклад, да тчика витоку води) достатньо додати відповідний програмний модуль без переб удови всієї структури SNMP-агента.

Виявлені завдання проєкту включають не лише технічну реалізацію збору даних, а й забезпечення інформаційної безпеки. Оскільки система моніторингу має доступ до мережевого сегмента управління, вона може стати вектором атак и. Тому аналіз предметної області вказує на необхідність суворого налаштуванн я рівнів доступу в протоколі SNMP та мінімізації відкритих портів на Raspberry Pi. Також необхідно передбачити механізми захисту від «брязкоту» датчиків та фільтрацію хибних спрацювань, які можуть перевантажити систему сповіщень.

Підсумовуючи аналіз, можна стверджувати, що створення бюджетної, ма сштабованої та відкритої системи моніторингу на базі Raspberry Pi є актуальни м науково-технічним завданням. Вона дозволяє нівелювати ризики, пов'язані з людським фактором та техногенними аваріями в серверних приміщеннях. Вико

ристання протоколу SNMP гарантує довговічність рішення та його сумісність з майбутніми оновленнями IT-інфраструктури підприємства. Основними етапами подальшої розробки стануть вибір оптимальної сенсорної бази, розробка демон а опитування на мові Python або C++ та конфігурація сервісу snmpd для коректн ої трансляції фізичних параметрів у мережеві змінні.

Окремої уваги заслуговує питання стандартизації форматів збережуваних даних. Оскільки система моніторингу збирає показники з різнорідного обладнання від різних виробників, виникає необхідність нормалізації вхідних потоків до єдиної внутрішньої схеми. У промислових умовах нерідко трапляються ситуації, коли одне і те ж фізичне значення, наприклад температура процесора, може бути представлено в різних одиницях або з різною розрядністю залежно від версії прошивки обладнання. Розроблювана система повинна передбачати рівень нормалізації даних, який перетворює різнорідні вхідні значення до єдиного формату перед їх збереженням у базі даних. Це забезпечує коректність порівняльного аналізу показників за тривалими часовими проміжками.

Важливим елементом аналізу предметної області є розгляд питань масштабованості рішення в умовах зростання кількості підконтрольних вузлів. При збільшенні парку серверного обладнання зростає і навантаження на систему моніторингу в частині частоти та обсягів опитування. Дослідження існуючих підходів свідчить, що вертикальне масштабування (нарощення ресурсів одного вузла збору даних) є менш ефективним, ніж горизонтальне, яке передбачає розподіл підконтрольних пристроїв між кількома агентами з подальшою агрегацією результатів у центральній точці. Raspberry Pi як платформа підходить для горизонтального масштабування саме завдяки своїй низькій вартості та компактним розмірам, що дозволяє розмістити окремий агент поруч із кожним серверним кластером або навіть у кожній стійці.

## 1.2 Порівняльний аналіз переваг та недоліків існуючих рішень

Аналіз ринку систем моніторингу серверної інфраструктури дозволяє вид ілити кілька ключових векторів розвитку, кожен з яких має свої специфічні пер еваги та обмеження. Першим і найбільш розповсюдженим класом рішень є інте гровані виробником системи віддаленого управління, такі як Intelligent Platform Management Interface (IPMI), Hewlett Packard Enterprise iLO (Integrated Lights-Out) або Dell iDRAC (Integrated Dell Remote Access Controller). Ці системи базу ються на спеціалізованих мікроконтролерах BMC (Baseboard Management Controller), що інтегровані безпосередньо в материнську плату сервера. Основн ою перевагою таких рішень є їхня незалежність від стану основної операційної системи сервера, оскільки BMC живиться від чергової лінії блока живлення. Це дозволяє адміністратору проводити діагностику навіть при вимкненому або зав ислому сервері. Проте суттєвим недоліком є замкненість цих систем всередині корпусу сервера. Вони ідеально відстежують внутрішні показники - напругу на шинах процесора, швидкість обертання штатних кулерів чи стан RAID-масивів, але абсолютно не здатні контролювати зовнішні параметри середовища, такі як витік води під фальшпідлогою, задимлення в приміщенні чи рівень вологості, щ о критично для запобігання корозії та електростатичних розрядів.

Другим значним сегментом є професійні промислові системи моніторинг у навколишнього середовища, прикладом яких є лінійка APC NetBotz від Schneider Electric або рішення від компанії Raritan. Ці пристрої проектуються як спеціалізовані мережеві хаби, до яких можна підключати десятки різноманітних сенсорів. Їхньою головною перевагою є надзвичайна надійність, наявність серт ифікатів відповідності промисловим стандартам та повна підтримка протоколу SNMP «з коробки». Проте в контексті дипломного проектування та впроваджен ня в реальні умови малого або середнього бізнесу такі системи мають критичні й недолік - надзвичайно високу вартість. Крім того, ці рішення часто мають зак риту архітектуру, що унеможливлює використання дешевих сторонніх датчиків або написання власних програмних скриптів для специфічної обробки даних. Б удь-яка модернізація чи розширення такої системи вимагає значних капіталовкл

адень у фірмові аксесуари, що робить їх економічно недоцільними для невелик их серверних вузлів або розподілених edge-обчислювальних точок.

Порівнюючи промислові системи з аматорськими або DIY (Do It Yourself) рішеннями на базі мікроконтролерів сімейства ESP8266 або ESP32, можна помі тити протилежну крайність. Рішення на базі ESP32 є надзвичайно дешевими та мають вбудовану підтримку Wi-Fi, що на перший погляд спрощує розгортання. Однак у серверних приміщеннях з високим рівнем електромагнітних завад вико ристання бездротового зв'язку є ненадійним та часто небезпечним з точки зору кібербезпеки. Мікроконтролери мають обмежений обсяг оперативної пам'яті, щ о робить реалізацію повноцінного стеку SNMP версії 3 (з шифруванням та авте нтифікацією) складним завданням. Більшість бібліотек SNMP для мікроконтрол ерів підтримують лише версію 1 або 2c, які передають дані у відкритому вигляд і, що є неприпустимим для корпоративної мережі. Також відсутність повноцінн ої операційної системи на таких пристроях ускладнює реалізацію локального ло гування даних у разі втрати мережевого з'єднання, що призводить до втрати ва жливої телеметрії в моменти аварій.

Використання Raspberry Pi як обчислювального вузла вбудованої системи моніторингу є компромісним та найбільш ефективним варіантом, що поєднує п ереваги промислових контролерів та гнучкість DIY-платформ. На відміну від п ростих мікроконтролерів, Raspberry Pi працює під управлінням ОС Linux (Debian/Raspberry Pi OS), що відкриває доступ до використання стандартних си стемних утиліт та пакетів, таких як Net-SNMP. Це дозволяє не просто зчитувати дані з датчиків, а й виконувати їх первинну обробку, фільтрацію шумів та агрег ацію безпосередньо на «краю» мережі. Перевагою архітектури Raspberry Pi є на явність великої кількості GPIO-контактів, що підтримують цифрові протоколи передачі даних I2C, SPI та 1-Wire. Це дозволяє використовувати професійні ци фрові сенсори, які мають значно вищу точність та стабільність порівняно з анал оговими компонентами, що часто використовуються в бюджетних системах.

У розрізі порівняльного аналізу програмних засобів передачі даних важли во розглянути альтернативи SNMP, зокрема протокол MQTT (Message Queuing Telemetry Transport). MQTT є дуже легким та зручним для IoT-пристроїв, оскіль ки працює за моделлю «видавець-підписник». Його перевагою є здатність прац ювати на нестабільних каналах зв'язку. Проте в корпоративному середовищі сер верного обладнання SNMP має незаперечну перевагу - універсальність. Всі існу ючі системи управління мережею (NMS), такі як Zabbix, PRTG або SolarWinds, орієнтовані на SNMP як на основний метод збору даних. Впровадження систем и на базі MQTT вимагало б розгортання додаткового брокера повідомлень та на писання складних шлюзів для інтеграції з існуючими панелями моніторингу. SNMP-агент на базі Raspberry Pi, навпаки, сприймається мережею як стандартн ий мережевий пристрій, що значно спрощує адміністрування та знижує витрати на налаштування програмного забезпечення.

Ще одним аспектом порівняння є надійність зберігання даних та стійкість до збоїв. Пропрієтарні системи часто мають вбудовану пам'ять обмеженого обс ягу, і при заповненні старі дані просто перезаписуються. У системі на базі Raspberry Pi можна використовувати SD-карти великої ємності або навіть підкл ючати зовнішні SSD-накопичувачі, що дозволяє зберігати архіви показань датч иків за кілька років. Це критично важливо для проведення ретроспективного ан алізу причин виникнення аварій. Крім того, наявність повноцінного мережевого стеку Linux дозволяє реалізувати паралельну відправку даних: через SNMP для моніторингу в реальному часі та, наприклад, через HTTP POST запити в зовніш ню базу даних InfluxDB для побудови детальних графіків у Grafana. Така гібрид на архітектура недоступна більшості закритих комерційних рішень.

Проблема електроживлення пристроїв моніторингу також є предметом п орівняльного аналізу. Комерційні пристрої високого класу часто підтримують Power over Ethernet (PoE), що дозволяє живити їх від того ж кабелю, яким перед аються дані. Це спрощує монтаж, оскільки не потребує встановлення додаткови х розеток у кожній серверній шафі. Для Raspberry Pi реалізація PoE можлива за

допомогою спеціальних модулів розширення (PoE HAT), що вирівнює можливо сті саморобної системи та дорогого промислового зразка. При цьому загальна в артість комплекту Raspberry Pi з PoE модулем та набором датчиків залишається в 5–10 разів нижчою, ніж вартість аналогічного за функціоналом пристрою від відомих брендів. Це відкриває шлях до масштабованості: за ціну одного NetBotz можна оснастити індивідуальними модулями моніторингу кожну стійк у в дата-центрі, що забезпечить набагато вищу гранулярність даних.

Важливим недоліком використання мікрокомп'ютерів типу Raspberry Pi п орівняно зі спеціалізованими контролерами є вразливість файлової системи до раптових вимкнень живлення. Якщо промисловий контролер використовує read-only пам'ять, то ОС Linux на Raspberry Pi може пошкодити дані на SD-карт і при аварійному знеструмленні. Ця проблема потребує програмно-апаратного в ирішення в рамках дипломного проекту, наприклад, шляхом налаштування опе раційної системи в режимі «тільки читання» або використання джерела безпере бійного живлення (UPS) для Raspberry Pi. Саме ці технічні виклики і складають наукову та інженерну новизну розробки, перетворюючи звичайний мікрокомп' ютер на надійний пристрій промислового класу.

Підсумовуючи порівняльний аналіз, можна визначити, що запропонована система на базі Raspberry Pi та протоколу SNMP займає оптимальну нішу. Вона перевершує внутрішні сервісні процесори (IPMI) за рахунок можливості підклю чення зовнішніх сенсорів, є значно функціональнішою за прості мікроконтроле ри (ESP32) завдяки повноцінній ОС та підтримці сучасних стандартів безпеки SNMP v3, і при цьому є в рази доступнішою за закриті промислові рішення. Ос новні завдання проекту, що випливають з цього аналізу, полягають у створенні стабільного програмного агента, забезпеченні надійності зберігання даних та ро зробці зручної схеми підключення різнорідних датчиків для повного охоплення всіх критичних параметрів серверного обладнання та приміщення.

## 1.3 Підходи до вирішення задачі за темою дослідження

Для комплексної реалізації вбудованої системи моніторингу стану та пара метрів мікроклімату серверного обладнання найбільш доцільним і технічно обґ рунтованим підходом є використання одноплатного комп'ютера Raspberry Pi. Н а відміну від класичних апаратних мікроконтролерів, які мають обмежені обчис лювальні ресурси та мережеві можливості, ця платформа функціонує під управ лінням повноцінної операційної системи на базі ядра Linux. Це дозволяє розгор нути стандартизований SNMP-агент безпосередньо на пристрої, забезпечуючи с товідсоткову сумісність із промисловими системами управління мережею (NMS), такими як Zabbix, Nagios або PRTG. Завдяки наявності повноцінної ОС, розробник отримує можливість використовувати розширені системні інструмен ти для діагностики, локального логування процесів та автоматичного відновлен ня роботи агентів після збоїв за допомогою системного менеджера systemd. Крі м того, такий підхід спрощує реалізацію багаторівневої програмної архітектури, де процеси збору фізичних показників, їх математичної обробки та мережевої п ередачі ізольовані один від одного.

Програмна логіка для збору первинних даних із цифрових датчиків темпе ратури, відносної вологості та інтенсивності повітряного потоку реалізується м овою програмування Python. Вибір цього інструменту зумовлений наявністю ро звиненої екосистеми спеціалізованих бібліотек для безпосередньої взаємодії з а паратними інтерфейсами GPIO, а також низькорівневими протоколами передачі даних I2C та 1-Wire. Використання об'єктно-орієнтованого підходу в Python дозволяє створити модульну структуру коду, де кожен тип датчика представлений окремим програмним класом. Це не лише полегшує первинне налагодження сис теми, але й дозволяє в майбутньому безперешкодно масштабувати апаратну час тину, додаючи нові сенсори (наприклад, датчики наявності диму, витоку води а бо струму) без необхідності переписування основного ядра програми. Додатков о впроваджуються алгоритми програмної фільтрації (наприклад, медіанний філ

ьтр) для відсічення хибних показників, що можуть виникати через електромагні тні наведення в серверній стійці.

Для забезпечення стабільної та надійної передачі зібраної телеметрії, а та кож для безшовної інтеграції в існуючу корпоративну інфраструктуру, оптимал ьним рішенням є розгортання та конфігурація програмного пакету Net-SNMP. Д аний інструментарій дозволяє гнучко розширювати та налаштовувати бази керу ючої інформації (MIB) під конкретні потреби підприємства. Крім того, він забез печує реалізацію механізмів асинхронних миттєвих сповіщень (SNMP Traps і Inform), що гарантує негайне інформування адміністраторів або автоматизовани х систем реагування у разі виходу контрольованих параметрів за межі встановл ених критичних значень. Важливим елементом цього підходу є використання п ротоколу SNMP версії 3, який підтримує сучасні криптографічні стандарти шиф рування трафіку та строгу автентифікацію користувачів. Налаштування Net-SNMP передбачає створення спеціального скрипта-посередника (pass-through script), який динамічно пов'язує локальні змінні середовища з унікальними іден тифікаторами об'єктів (OID) у кастомному MIB-дереві, завдяки чому Raspberry Pi сприймається мережею як професійний промисловий контролер моніторингу.

Принципово важливим технічним рішенням є вибір стратегії зберігання і обробки даних безпосередньо на агенті. Існує два принципових підходи: режим «тонкого клієнта», при якому агент лише збирає сирі дані та миттєво передає їх на центральний сервер, та режим «розумного агента», де первинна обробка й агрегація виконуються на місці. Другий підхід є кращим для Raspberry Pi, оскільки дозволяє суттєво знизити мережевий трафік в сторону центральної системи управління. Замість передачі кожного окремого вимірювання агент накопичує показники протягом визначеного проміжку часу, обчислює середні, мінімальні та максимальні значення і передає вже агреговані результати. Це особливо актуально для мереж із обмеженою пропускною здатністю або при моніторингу географічно розподілених об'єктів.

Реалізація надійної системи сповіщень вимагає особливої уваги до проблеми дублювання тривог. При традиційному підході кожне перевищення порогового значення негайно генерує сповіщення, що при нестабільних показниках призводить до так званого «шторму тривог». Адміністратор отримує сотні однотипних повідомлень упродовж декількох хвилин, більшість із яких є повторними і не несуть нової інформації. Для вирішення цієї проблеми в розроблюваній системі застосовується механізм дедублікації: нова тривога по конкретному параметру може бути згенерована лише після того, як відповідний показник повернувся до нормального діапазону і знову перетнув порогове значення. Такий підхід суттєво знижує інформаційний шум і дозволяє адміністратору зосередитись на справді нових та актуальних подіях.

## 1.4 Постановка задачі та визначення вимог до розроблюваної системи

Метою дипломного проєкту є обґрунтування, проектування та розробка а втономного програмно-апаратного комплексу, призначеного для безперервного цілодобового контролю параметрів навколишнього середовища, мікроклімату т а загального технічного стану серверних вузлів. Для досягнення поставленої ме ти необхідно вирішити низку взаємопов'язаних завдань: провести глибокий ана ліз вимог до сучасних систем моніторингу дата-центрів, підібрати оптимальну к омпонентну базу з урахуванням співвідношення ціна-якість, а також розробити архітектуру апаратно-програмного рішення. Окремим важливим завданням є пр оектування схеми резервного живлення пристрою, що забезпечить його автоно мність та здатність відправити передсмертне сповіщення (last gasp) під час крит ичних аварій в електромережі.

На основі послідовного виконання визначених науково-технічних та інже нерних завдань необхідно спроектувати, зібрати та програмно реалізувати повн офункціональну працездатну вбудовану систему моніторингу стану серверного обладнання на базі архітектури Raspberry Pi. Створюваний комплекс має забезп

ечити високу точність вимірювань, стабільність функціонування в умовах пром ислової експлуатації, а також гарантувати надійне й оперативне сповіщення сис темних адміністраторів про будь-які критичні чи позаштатні зміни робочих умо в у серверній кімнаті. Крім того, розроблена система повинна мати інтуїтивно з розумілу конфігурацію і гнучко масштабуватися шляхом підключення додатков их сенсорних модулів через стандартні шини обміну даними.

Серед функціональних вимог до розроблюваної системи особливе місце з аймає вимога до масштабованості архітектури. Система повинна бути здатна бе з суттєвої переробки програмного коду забезпечити одночасний моніторинг від одного до щонайменше п'ятдесяти мережевих вузлів. Досягнення цієї характери стики можливе лише за умови використання асинхронної моделі опитування пр истроїв, яка дозволяє паралельно очікувати відповіді від кількох хостів, не блок уючи основний потік виконання програми. У цьому контексті бібліотека asyncio мови Python є технічно обґрунтованим вибором, оскільки реалізує подіє во-орієнтовану модель вводу-виводу без накладних витрат, властивих багатопо токовому підходу.

Окремою нефункціональною вимогою є надійність збереження даних при нестабільному мережевому з'єднанні між агентом та сервером баз даних. Систе ма повинна підтримувати режим локального кешування зібраних метрик у разі тимчасової недоступності сховища, з подальшою синхронізацією після відновл ення з'єднання. Це унеможливлює втрату телеметричних даних у критичні мом енти, наприклад під час планового або позапланового технічного обслуговуван ня мережевої інфраструктури. Реалізація такого механізму вимагає впроваджен ня проміжного шару черги повідомлень або локального буфера на рівні файлов ої системи Raspberry Pi.

Вимоги до безпеки системи базуються на принципі мінімальних привілеїв та захисту мережевого трафіку. Усі комунікації між компонентами системи, що виходять за межі локальної підмережі управління, мають бути захищені засоба ми автентифікації та шифрування, передбаченими стандартом SNMPv3. Зокрем

а, конфіденційність трафіку забезпечується алгоритмом AES-128, а цілісність та автентичність повідомлень – за допомогою HMAC-SHA. Для локальних взаємо дій між мікросервісами системи, що функціонують на одному вузлі Raspberry Pi, допускається використання спрощеного протоколу SNMPv2c у межах мережево го інтерфейсу зворотного зв'язку (loopback), що виключає несанкціонований до ступ ззовні.

Важливою якісною вимогою є зрозумілість та зручність інтерфейсу керув ання системою для системного адміністратора. Веб-інтерфейс повинен відобра жати актуальний стан усіх підконтрольних пристроїв на єдиній панелі із застос уванням семантичного кольорового кодування: зелений – нормальний стан, жов тий – попереджувальний рівень, червоний – критичний стан або недоступність вузла. Відповідно до вимог зручності використання, час реакції веб-інтерфейсу на запит користувача не повинен перевищувати двох секунд при стандартному навантаженні на систему. Для досягнення цього показника необхідна оптимізац ія запитів до бази даних, зокрема індексування полів часових міток та ідентифік аторів пристроїв у таблицях збережених метрик.

Визначені вимоги також включають можливість гнучкого налаштування порогових значень спрацювання сповіщень без перезапуску системи. Адміністр атор повинен мати змогу через веб-інтерфейс або конфігураційний файл у форм аті YAML задати індивідуальні пороги для кожного пристрою окремо або визна чити глобальні значення за замовчуванням. Архітектурно це вимагає впровадже ння сервісу конфігурації, який зберігає поточні налаштування у базі даних та за безпечує їх гарячу перезавантаження (hot reload) для модулів перевірки порогів. Такий підхід відповідає сучасним практикам DevOps та дозволяє адаптувати си стему до мінливих умов навантаження на серверне обладнання без технологічн их вікон обслуговування.

Підсумовуючи, сформульована постановка задачі визначає розробку прог рамно-апаратного комплексу, що поєднує вбудовані обчислювальні засоби Raspberry Pi, стандартизований протокол SNMP та сучасні фреймворки веброзр

обки для створення комплексного, надійного та економічно ефективного інстру менту моніторингу серверної інфраструктури. Виконання всіх сформульованих вимог є необхідною умовою для досягнення поставленої мети дипломного прое кту.

Окремим критерієм порівняльного аналізу є питання документування та підтримки системи протягом усього життєвого циклу. Комерційні рішення зазвичай супроводжуються вичерпною технічною документацією та гарантійними зобов'язаннями виробника, однак оновлення прошивок нерідко виходять із запізненням або взагалі припиняються після завершення офіційного терміну підтримки пристрою. Системи на базі Raspberry Pi, навпаки, спираються на відкриту спільноту розробників і постійно оновлюване програмне забезпечення з відкритим кодом. Це означає, що виявлені вразливості безпеки усуваються оперативно, а нові можливості можуть бути інтегровані адміністратором самостійно без очікування нового релізу від вендора. Така модель підтримки є особливо цінною в умовах постійно змінюваного ландшафту кіберзагроз, коли швидкість реакції на нові вразливості безпосередньо впливає на захищеність всієї інфраструктури.

# 2 ПРОЄКТУВАННЯ СИСТЕМИ ЗБОРУ ТА ОБРОБКИ ДАНИХ ВБУДОВАНОЇ СИСТЕМИ МОНІТОРИНГУ СТАНУ СЕРВЕРНОГО ОБЛАДНАННЯ НА БАЗІ RASPBERRY PI ТА ПРОТОКОЛУ SNMP

## 2.1 Концептуальні основи та структурна організація системи

Сучасна інформаційна інфраструктура підприємств складається з гетерог енних компонентів: серверів під управлінням різних операційних систем, актив ного мережевого обладнання та систем зберігання даних. Ефективне управлінн я такою інфраструктурою потребує безперервного збору діагностичної інформа ції, своєчасного виявлення аномалій та автоматизованого реагування на нештат ні ситуації.

У класичній архітектурі систем моніторингу виокремлюють три основні рівні. Першим є рівень збору даних, де спеціальні агенти взаємодіють із підконтрольними вузлами через мережеві протоколи. Другим є рівень обробки та зберігання, який працює як центральний вузол для збору метрик і підтримки бази знань про стан мережі. Третім є рівень представлення, що забезпечує зручний інтерфейс для взаємодії з операційним персоналом.

Спроектована система реалізує повний стек усіх трьох рівнів на єдиній ап аратній платформі - Raspberry Pi 5. Такий підхід характерний для вбудованих і «легких» систем моніторингу, що не потребують окремої серверної інфраструкт ури.

За принципом ініціювання збору даних розрізняють два підходи: pull-мод ель, де менеджер активно запитує метрики від агентів із заданим інтервалом, та push-модель, де агенти самостійно надсилають дані на сервер за власним розкла дом або при настанні певних подій. SNMP реалізує pull-модель: підконтрольні а генти лише відповідають на запити, не ведучи власного таймера збору. Це суттє во спрощує програмне забезпечення агента, яке де-факто вбудоване у прошивку мережевого обладнання. Push-модель, характерна для систем Prometheus із Pushgateway або InfluxDB з агентом Telegraf, надає меншу затримку доставки п

одій, однак потребує складнішої конфігурації на боці кожного агента та надійні шої мережевої зв'язності. Для корпоративних середовищ із обладнанням різних виробників pull-модель через SNMP залишається домінуючим підходом завдяк и стандартизованості та десятиліттям промислового використання.

Технологічною основою системи є протокол SNMP (Simple Network Management Protocol) - де-факто стандарт для моніторингу та управління мереж евими пристроями, визначений у серії документів RFC. Протокол реалізує моде ль «менеджер - агент»: менеджер надсилає запити агентам, що виконуються на кожному підконтрольному пристрої, та отримує значення змінних, організован их у вигляді ієрархічного дерева об'єктів - Бази Управляючої Інформації (Management Information Base, MIB).

У спроектованій системі використовується версія SNMPv2c - найпоширен іша у корпоративних середовищах завдяки балансу між простотою реалізації та функціональністю. Порівняно з SNMPv1 ця версія додає підтримку 64-бітних лі чильників (тип `Counter64`, необхідний для мережевих інтерфейсів зі швидкіст ю 1 Гбіт/с і вище), операцію `GETBULK` для масового зчитування табличних д аних та покращену обробку помилок. Автентифікація здійснюється через рядок спільноти (community string), що для ізольованих внутрішніх мереж є прийнятн им рішенням.

Транспортним протоколом слугує UDP: відсутність встановлення з'єднан ня мінімізує накладні витрати і дозволяє виявляти недоступні хости через тайма ут без зависання активного з'єднання. Стандартний порт SNMP-агента - 161/UDP.

Варто зазначити, що SNMPv3 вводить криптографічну аутентифікацію (м одель USM) та шифрування трафіку, що усуває основний недолік SNMPv2c - пе редачу community string у відкритому вигляді. Однак SNMPv3 значно складніш ий в налаштуванні та вимагає узгодження ключів для кожного пристрою. У кон тексті ізольованого внутрішнього моніторингового полігону, де мережевий тра фік не виходить за межі організації, використання SNMPv2c з community string

"public" є практично загальноприйнятим рішенням, що значно знижує операційну складність розгортання.

Ієрархічна структура об'єктів OID формується від загального кореня і послідовно розгалужується. Наприклад, стандартний шлях до базової гілки управління MIB-2 записується як 1.3.6.1.2.1, де кожна цифра позначає певний рівень класифікації від глобальної організації ISO до стандартів мережі Інтернет. Комерційні виробники обладнання реєструють власні унікальні розширення бази даних у спеціальній гілці enterprises, яка має базовий ідентифікатор 1.3.6.1.4.1. Для прикладу, мережеве обладнання Cisco використовує кінцевий індекс 9, а програмний пакет Net-SNMP для систем Linux ідентифікується числом 8072.

Використання стандартизованих гілок MIB є критично важливим для забезпечення універсальної сумісності пристроїв різних виробників. Якщо системне програмне забезпечення підтримує базові галузеві стандарти RFC 1213 та RFC 2790, воно гарантовано відповідатиме на ідентичні запити OID незалежно від своєї апаратної платформи.

Програмна реалізація цього процесу в розробленій системі базується на сучасній бібліотеці pysnmp актуальної шостої версії від розробників lextudio. Вона забезпечує повноцінну підтримку протоколу SNMPv2c та пропонує асинхронний програмний інтерфейс на базі технології asyncio. Цей архітектурний вибір є значно ефективнішим порівняно зі старими версіями бібліотеки, де застосовувався застарілий механізм asyncore.

Для збору телеметрії в системі залучено дві стандартні групи MIB. Перша з них, відома під назвою MIB-II згідно зі стандартом RFC 1213, відповідає за отримання загальної системної інформації та характеристик мережевих інтерфейсів. На таблиці 2.1 наведено основні ідентифікатори об'єктів OID стандартної групи MIB-II.

Таблиця 2.1 – Основні об'єкти групи MIB-II

| OID | Символічна назва | Опис |
|---|---|---|

| OID | Символічна назва | Опис |
|---|---|---|
| 1.3.6.1.2.1.1.1.0 | sysDescr | Текстовий опис обладнання та операційної системи |
| 1.3.6.1.2.1.1.3.0 | sysUpTime | Час безперервної роботи системи у сотих частках секунди |
| 1.3.6.1.2.1.1.5.0 | sysName | Мережеве ім'я пристрою (хоста) |
| 1.3.6.1.2.1.2.1.0 | ifNumber | Загальна кількість мережевих інтерфейсів пристрою |
| 1.3.6.1.2.1.2.2.1.2.N | ifDescr | Текстовий опис N-го мережевого інтерфейсу |
| 1.3.6.1.2.1.2.2.1.8.N | ifOperStatus | Поточний операційний стан інтерфейсу (1 = активний, 2 = неактивний) |
| 1.3.6.1.2.1.2.2.1.10.N | ifInOctets | Кількість вхідних байтів на N-му інтерфейсі |
| 1.3.6.1.2.1.2.2.1.16.N | ifOutOctets | Кількість вихідних байтів на N-му інтерфейсі |

На таблиці 2.2 наведено перелік ключових ідентифікаторів цієї групи, за допомогою яких система збирає розширену телеметрію про обсяги оперативної пам'яті, тип і стан дискового простору, а також рівень поточного завантаження кожного ядра центрального процесора.

Таблиця 2.2 – Основні об'єкти групи HOST-RESOURCES-MIB (RFC 2790)

| OID | Символічна назва | Опис |
|---|---|---|
| 1.3.6.1.2.1.25.2.2.0 | hrMemorySize | Загальний обсяг оперативної пам'яті у кілобайтах |
| 1.3.6.1.2.1.25.2.3.1.2.N | hrStorageType | Тип носія інформації (оперативна пам'ять, жорсткий диск, флеш-накопичувач) |
| 1.3.6.1.2.1.25.2.3.1.4.N | hrStorageAllocation Units | Розмір блоку розподілу даних або кластера у байтах |
| 1.3.6.1.2.1.25.2.3.1.5.N | hrStorageSize | Загальний об'єм сховища, виражений у кількості блоків |
| 1.3.6.1.2.1.25.2.3.1.6.N | hrStorageUsed | Кількість фактично зайнятих блоків у сховищі |
| 1.3.6.1.2.1.25.3.3.1.2.N | hrProcessorLoad | Рівень завантаженості N-го ядра центрального процесора (%) |

Операція SNMP WALK - послідовність операцій `GETNEXT`, що обходи ть усі OID у межах вказаного піддерева. Вона дозволяє читати табличні структу ри змінної довжини (наприклад, таблиці з різною кількістю ядер ЦП або мереж евих інтерфейсів) без попереднього знання кількості рядків. Для ефективного а синхронного обходу в бібліотеці pysnmp 6.x реалізований генератор `walkCmd`, що видає результати по одному і не завантажує всю таблицю в пам'ять.

Обчислення відсотка використання ОЗП потребує узгодження двох джере л даних. Скалярний OID `hrMemorySize` надає загальний об'єм у кілобайтах, то ді як таблиця `hrStorageTable` містить значення в одиницях блоків розподілу (`hrStorageAllocationUnits`, байти). Для ідентифікації рядка оперативної пам'яті необхідно виконати WALK таблиці та знайти рядок, де `hrStorageType` дорівню

є `1.3.6.1.2.1.25.2.1.2` (hrStorageRam). Формула обчислення: `ram% = (hrStorageUsed × hrStorageAllocationUnits) / (hrMemorySize × 1024) × 100`. Аналогічна двоетапна процедура ідентифікації за типом застосовується для визначення відсотка завантаженості диска через тип `hrStorageFixedDisk`. Такий підхід є необхідним, оскільки таблиця може містити довільну кількість записів різних типів (RAM, диск, flash, cd-rom) з непередбачуваними індексами.

Система побудована за принципом модульності та слабкого зв'язку між компонентами. Архітектурне рішення ґрунтується на розподілі функціональних обов'язків між чотирма незалежними шарами, що відповідає класичній N-tier архітектурі:

Шар збору даних відповідає за взаємодію з підконтрольними пристроями через SNMP. Він ізольований від деталей зберігання та представлення і повертає уніфіковані словники метрик. Завдяки цьому заміна SNMP на інший протокол (наприклад, IPMI або Redfish) потребуватиме змін виключно в цьому шарі.

Шар зберігання відповідає за персистентність даних через об'єктно-реляційний маппер (ORM). Використання SQLAlchemy як абстракції над базою даних дозволяє при необхідності замінити SQLite на PostgreSQL без зміни коду прикладного рівня.

Шар бізнес-логіки реалізує порогову перевірку та управління сповіщеннями. Цей шар не залежить ні від протоколу збору, ні від технології зберігання - він оперує виключно числовими значеннями метрик і правилами з конфігураційного файлу.

Шар представлення реалізує два підінтерфейси: REST API для програматичного доступу та веб-інтерфейс для оперативного персоналу, що можуть розвиватися незалежно.

Взаємодія між шарами визначена через мінімальні програмні інтерфейси з чіткими контрактами. Шар збору повертає словник Python з фіксованою схемою ключів: `cpu_percent`, `ram_percent`, `disk_percent`, `interfaces`, `sys_name`, `sys_descr`, `uptime`. Шар бізнес-логіки оперує виключно числовими значенням

и та порогами з конфігурації, не залежачи від протоколу чи джерела даних. Ша р представлення споживає дані лише через REST API, що дозволяє замінити ве б-клієнт (наприклад, на мобільний застосунок або Grafana-дашборд) без жодних змін у бекенді. Цей підхід відповідає принципу відкритості/закритості зі станда рту SOLID: кожен шар відкритий для розширення власної функціональності, ал е закритий для модифікацій, що диктуються змінами у суміжних шарах.

Симулятор мережевих пристроїв призначений для розробки та тестування без залучення реального обладнання. Запускає три незалежних SNMP-агенти на портах 1161–1163, що емулюють: сервер Ubuntu Linux 22.04 (4 ГБ ОЗП, 4 ядра CPU), сервер Windows Server 2022 (16 ГБ ОЗП, 4 ядра CPU) та маршрутизатор Cisco IOS 15.x (512 МБ ОЗП, 1 ядро CPU, 4 інтерфейси GigabitEthernet). Кожен агент читає статичний файл формату `.snmprec` зі значеннями OID у лексикогра фічному порядку і відповідає на SNMPv2c-запити. В умовах реального розгорта ння симулятор замінюється реальним мережевим обладнанням.

Колектор метрик являється ядром системи збору даних. Складається з трь ох модулів: `snmp_client.py` реалізує асинхронні функції SNMP GET та WALK; `oid_map.py` централізовано зберігає всі OID-константи, розподілені на скалярн і (системна інформація) та табличні (процесор, сховище, інтерфейси); `poller.py` формує структуровані результати опитування: відсоток завантаження CPU (сер еднє по ядрах з `hrProcessorLoad`), відсоток використання RAM (рядок типу `hrStorageRam` у таблиці сховищ), відсоток використання диска (рядок типу `hrStorageFixedDisk`) та список інтерфейсів з їх статусами та лічильниками тра фіку. Централізація OID-констант у окремому модулі виключає дублювання ма гічних рядків у багатьох файлах і забезпечує єдине місце для оновлення при змі ні версії MIB або переході на нове обладнання.

Сервіс опитування є фоновим компонентом на базі бібліотеки APScheduler. Запускається одночасно з веб-застосунком у вигляді `BackgroundScheduler` з інтервалом 60 секунд. Перший цикл виконується негай

но після запуску. Всі пристрої опитуються паралельно в межах одного циклу за допомогою механізму конкурентного виконання `asyncio.gather()`.

Підсистема зберігання побудована на SQLAlchemy 2.x з бекендом SQLite. Визначає три моделі даних: `Device` - реєстр пристроїв з полями конфігурації т а поточного статусу (`status`, `last_seen`, `unreachable_count`); `Metric` - часовий ряд метрик, де кожен рядок містить ідентифікатор пристрою, назву метрики, чи слове значення та мітку часу; `Alert` - журнал сповіщень з рівнем серйозності (`WARNING`/`CRITICAL`) та ознакою підтвердження. При 3 пристроях та 7 ме триках на пристрій з інтервалом 60 секунд система генерує близько 30 240 рядк ів на добу. Автоматичне очищення метрик старших за 30 днів підтримує розмір бази у передбачуваних межах.

Підсистема сповіщень аналізує метрики після кожного циклу опитування. Модуль `checker.py` порівнює значення CPU, RAM та диска з конфігурованими порогами (WARNING/CRITICAL) та перевіряє операційний статус мережевих і нтерфейсів. Передбачений механізм дедублікації: повторне сповіщення про ту с аму проблему для того самого пристрою не надсилається впродовж 5 хвилин. М одуль `telegram_notifier.py` відправляє сповіщення через Telegram Bot API у окр емому потоці, не блокуючи основний цикл опитування.

Веб-застосунок реалізований за патерном Application Factory на базі Flask 3.x. REST API надає сім ендпоінтів для читання метрик, переліку пристроїв та у правління сповіщеннями. Веб-інтерфейс побудовано на Bootstrap 5.3 та Chart.js 4.4 з темною кольоровою схемою. Включає три сторінки: зведений Dashboard, д еталі пристрою з інтерактивними часовими графіками та сторінку управління с повіщеннями з можливістю підтвердження.

Патерн Application Factory є невід'ємною частиною архітектури Flask-заст осунків з фоновими планувальниками. Функція `create_app()` викликається одн оразово при старті і повертає налаштований екземпляр Flask-застосунку. Парам етр `use_reloader=False` запобігає дублювання APScheduler при розробці (автоп ерезапуск Flask запускає процес двічі). Flask Blueprint в модулі `api/routes.py` до

зволяє логічно розділити маршрути REST API від основного об'єкта застосунку, що спрощує подальше масштабування веб-шару без змін у прикладному шарі.

Патерн Application Factory (фабрика застосунку) є невід'ємною частиною архітектури Flask-застосунків з фоновими планувальниками. Функція `create_app()` викликається одноразово при старті і повертає налаштований екзе мпляр Flask-застосунку. Це забезпечує коректну роботу APScheduler: фоновий планувальник запускається ровно один раз, а не двічі (що сталося б при автомат ичному перезапуску Flask у режимі розробки). Параметр `use_reloader=False` пр и запуску через `app.run()` ль запобіжним заходом проти дублювання планувальника. Патерн Flask Blueprint (модуль `api/routes.py`) дозволяє логічно розділити маршрути REST API від основного об'єкта застосунку, що спрощує подальше м асштабування веб-шару без змін в прикладному шарі.

Взаємодія між компонентами системи побудована за принципом однонаправленого потоку даних: від мережевих пристроїв через колектор до сховища та, зрештою, до користувача через веб-інтерфейс.

Взаємодія між компонентами системи побудована за принципом суворо однонаправленого, нисхідного потоку даних – від мережевих пристроїв через колектор безпосередньо до сховища, а потім до користувацького інтерфейсу. Головне архітектурне правило полягає в тому, що жоден компонент нижнього рівня не імпортує модулі з верхніх шарів. Наочно логіку інформаційного обміну та загальну структуру взаємозв'язків між компонентами зображено на рисунку 2.1.

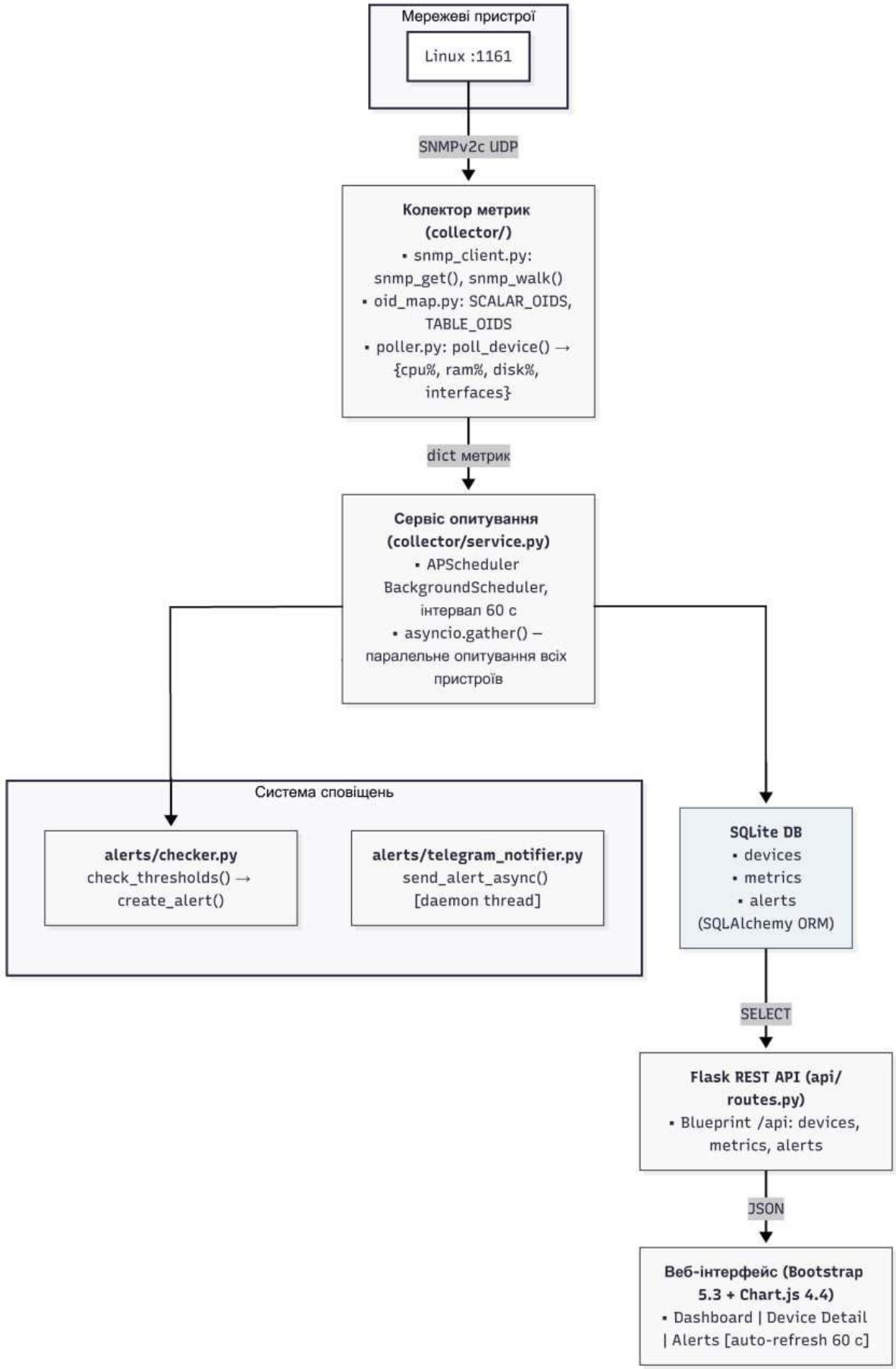



Рисунок 2.1 – Схема взаємодії між компонентами

Як детально показано на схемі, єдиним архітектурним винятком із загального правила є імпорт моделі Alert з шару зберігання до модуля перевірки alerts/checker.py. Це коректний патерн проектування, оскільки підсистема зберігання даних повинна залишатися повністю ізольованою і не містити логіки формування сповіщень. Головним результатом такої структури є глибока модульність системи. Наприклад, у разі необхідності впровадження нового протоколу збору даних (такого як IPMI), розробнику достатньо замінити або розширити лише відповідний шар збору, абсолютно не порушуючи роботу шарів аналізу, зберігання та веб-представлення.

Апаратною платформою для системи обрано Raspberry Pi 5 з 1 ГБ операти вної пам'яті. Це одноплатний комп'ютер з 64-бітним чотириядерним процесоро м Arm Cortex-A76 (2.4 ГГц), вбудованим гігабітним Ethernet-контролером та по ртами USB 3.0. Попри скромні характеристики, платформа є достатньою для мо ніторингу до 50 пристроїв при інтервалі опитування 60 секунд, а її переваги - н изьке енергоспоживання (5–8 Вт), малі габарити та доступна ціна - є визначальн ими для вбудованих систем.

Організація зберігання часових рядів метрик є окремим архітектурним завданням, що суттєво впливає на продуктивність системи в довгостроковій перспективі. Реляційні бази даних загального призначення, такі як PostgreSQL або MySQL, не є оптимальним вибором для телеметричних даних через неефективне стиснення та повільну обробку запитів типу «покажи всі значення за минулий місяць». Для збереження показників обраний підхід на базі SQLite з партиціонуванням таблиць за часовим принципом. Старі записи автоматично видаляються або переносяться в архівне сховище після закінчення встановленого терміну зберігання, що запобігає нескінченному зростанню бази даних і деградації продуктивності запитів. Такий підхід не вимагає запуску окремого серверного процесу бази даних, що особливо важливо для ресурсно-обмежених вбудованих систем.

З точки зору мережевої топології, система може функціонувати у двох режимах розгортання. Перший передбачає розміщення агента в тому ж мережевому сегменті, що й підконтрольні пристрої, що забезпечує мінімальну затримку при опитуванні. Другий режим дозволяє агенту знаходитись у виділеному сегменті управління і спілкуватись із серверами через маршрутизовану мережу. В обох випадках необхідно враховувати налаштування міжмережевих екранів: SNMP використовує UDP-порт 161 для запитів і порт 162 для пасток (traps). Коректна конфігурація правил фільтрації трафіку є передумовою надійної роботи системи і має бути задокументована разом із схемою розгортання для подальшого використання адміністраторами.

Вибір формату зберігання конфігурації агента є важливим аспектом зручності адміністрування. Двійкові формати конфігурації, хоча й забезпечують швидке завантаження, є нечитабельними без спеціалізованих інструментів і ускладнюють відлагодження. Натомість текстовий формат YAML поєднує зручну читабельність для людини з достатньою структурованістю для машинної обробки. Конфігураційний файл системи описує перелік підконтрольних пристроїв, параметри SNMP-підключення для кожного з них, порогові значення для генерації тривог та налаштування інтерфейсу сповіщень. Розподіл конфігурації між кількома ієрархічними файлами дозволяє системному адміністратору вносити зміни в налаштування окремих пристроїв без ризику порушення глобальних параметрів системи.

Вибір Raspberry Pi 5 над попередніми моделями обумовлений суттєвим п риростом продуктивності: процесор Cortex-A76 приблизно втричі швидший за Cortex-A72 у Raspberry Pi 4 у задачах обробки даних, а збільшена пропускна зда тність пам'яті LPDDR4X (64-бітна шина) забезпечує кращу продуктивність при одночасних операціях читання та запису до SQLite. Новий контролер PCIe 2.0 д озволяє підключати NVMe-накопичувачі через M.2 HAT-плату для максимальн о надійного сховища метрик. Для порівняння: Raspberry Pi Zero 2 W мала б дост

атньо ресурсів лише для моніторингу 3–5 пристроїв через однопотокові обмеження.

Програмне середовище: Raspberry Pi OS (Debian 12 Bookworm) з Python 3.11 у ізольованому віртуальному середовищі (`venv`) за адресою `/opt/snmp-monitor/venv`. Ізоляція через `venv` гарантує відтворюваність залежностей та уникнення конфліктів із системними пакетами.

Для автоматичного запуску та перезапуску при збоях використовуються два systemd-юніти: `snmp-simulator.service` (запускається першим, оскільки позначений як `Before=snmp-monitor.service`) та `snmp-monitor.service` (запускає основний застосунок від непривілейованого користувача з директивою `Restart=on-failure`). Для продуктивного середовища рекомендується запуск через Gunicorn з одним робочим процесом (`-w 1`) - кілька процесів призводили б до дублювання APScheduler та конкурентного запису до бази даних.

Для тривалої безперебійної роботи базу даних рекомендується розміщувати на USB 3.0 SSD-накопичувачі, а не на SD-картці: SD-карти мають обмежений ресурс циклів запису й не оптимізовані для інтенсивного випадкового I/O, характерного для часових рядів метрик.

При стандартній конфігурації системи (3 пристрої, 7 метрик, інтервал 60 с) генерується близько 30 000 рядків на добу, що відповідає приблизно 6–8 МБ даних - навіть 32 ГБ SSD забезпечить зберігання понад п'яти років метрик.

Навантаженість процесора Raspberry Pi 5 при моніторингу 3 пристроїв не перевищує 5–8%, що залишає достатній резерв для масштабування до 30–50 під контрольних вузлів без зміни апаратної платформи. При досягненні цієї межі горизонтальне масштабування можливе через перехід до клієнт-серверної моделі зі збереженням колектора на Raspberry Pi та перенесенням зберігання і веб-сервера на більш потужний хост.

## 2.2 Функціональна організація та алгоритми роботи компонентів системи

Перш ніж розглядати конкретні задачі, необхідно провести розмежування між паралелізмом та конкурентністю. Паралелізм передбачає одночасне фізичн е виконання кількох обчислень на різних ядрах процесора. Конкурентність – це структурування програми таким чином, щоб кілька задач могли виконуватись у перекривних часових інтервалах, навіть якщо у кожен момент виконується лиш е одна з них.

Python реалізує обидві концепції різними засобами. Для задач, обмежених вводом-виводом (мережеві запити, звернення до бази даних), конкурентність че рез asyncio є найефективнішим підходом: процесор простоює в очікуванні мере жевих пакетів, і asyncio дозволяє використовувати цей час для інших задач. Для задач, обмежених обчисленнями, необхідний справжній паралелізм через multiprocessing, оскільки глобальне блокування інтерпретатора (GIL) не дозволя є кільком потокам одночасно виконувати байт-код. Оскільки розроблена систем а моніторингу є типовим застосунком, обмеженим вводом-виводом (час циклу опитування майже повністю визначається мережевими затримками SNMP), сам е asyncio є природним вибором для шару збору даних.

Важливою властивістю asyncio є детермінована відсутність стану гонки: о скільки усі корутини виконуються в одному потоці, між двома операціями очік ування виконання гарантовано не переривається. Це дозволяє безпечно читати т а модифікувати спільний стан. Для поєднання результатів паралельних корутин використовується функція asyncio.gather із параметром return_exceptions=True. Цей параметр є архітектурно критичним: без нього перше ж виключення (напри клад, таймаут від недоступного пристрою) скасувало б усі інші задачі. З ним ко жен недоступний пристрій повертає об'єкт виключення, дозволяючи основному циклу продовжуватись.

У системі реалізовано декілька ключових завдань, що виконуються конку рентно або паралельно:

Паралельне опитування мережевих пристроїв. Найбільш явним прикладом конкурентності є одночасне опитування всіх підконтрольних пристроїв у межах одного циклу. При конфігурованому таймауті SNMP у 5 секунд та 2 повторних спробах максимальний час очікування від одного пристрою становить 15 секунд. Порівняння ефективності підходів наведено в таблиці 2.3.

Як видно з таблиці, при масштабуванні до 50 пристроїв послідовне опитування фізично неможливо виконати в межах 60-секундного інтервалу, тому паралельний підхід є обов'язковою умовою масштабованості.

Таблиця 2.3 – Порівняння часу опитування підконтрольних пристроїв

| Кількість пристроїв | Послідовне опитування, с | Паралельне опитування, с | Виграш |
|---|---|---|---|
| 3 | до 45 | до 15 | 3× |
| 10 | до 150 | до 15 | 10× |
| 50 | до 750 | до 15 | 50× |

Збереження метрик і перевірка порогів. Після завершення опитування виникають дві незалежні задачі: збереження метрик у базу даних (I/O-операція) та аналіз значень на предмет перевищення порогів (обчислювальна операція). Архітектурно вони ізольовані, що дозволяє виконувати їх конкурентно.

Надсилання сповіщень. Відправлення повідомлень через Telegram є операцією з непередбачуваною затримкою. Вона виконується у окремому daemon-потоці за патерном «fire-and-forget», що ізолює мережеві затримки месенджера від основного циклу. При недоступності API Telegram повідомлення може бути втрачено, але це не впливає на збереження метрик.

Спільна робота HTTP-сервера та планувальника. Flask обробляє запити в основному потоці, тоді як APScheduler виконує опитування у фоновому потоці.

Обидва компоненти спільно використовують підключення до бази SQLite. Завд яки використанню режиму WAL (Write-Ahead Logging) у базі даних, тривалі за пити на читання (наприклад, побудова графіків) виконуються без блокування за пису нових метрик.

Сумарна характеристика цих завдань демонструє, що в системі реалізован о ортогональні рівні конкурентності, що забезпечує мінімальне навантаження н а процесор Raspberry Pi при збереженні високої швидкодії.

## 2.3 Реалізація самоорганізації в архітектурі системи

Самоорганізація у програмних системах – це здатність системи адаптуват и свою поведінку у відповідь на зміни стану середовища без прямого втручання оператора. Концепція автономних обчислень (Autonomic Computing), запропоно вана IBM, визначає чотири ключові властивості: самоконфігурація, самолікуван ня, самооптимізація та самозахист. Спроектована система моніторингу повніст ю реалізує перші три властивості, використовуючи декларативні конфігураційні файли та динамічні правила замість жорстко прописаних алгоритмів.

Самоконфігурація та автономна ініціалізація. При першому запуску систе ма виконує повну ініціалізацію без ручного втручання: створює схему бази дан их, реєструє пристрої з конфігураційного файлу та генерує демонстраційні дані для миттєвої візуалізації. Конфігурація пристроїв описана у YAML-файлі за пр инципом Infrastructure-as-Code. Додавання нового пристрою або зміна порогови х значень набуває чинності автоматично під час наступного циклу опитування.

Система автономно відстежує стан кожного пристрою через скінченний а втомат із чотирма станами: unknown (початковий), up (відповідає), degraded (від повідає з проблемами) та down (не відповідає). Переходи між станами відбуваю ться автоматично. Перехід у стан down фіксується лише після трьох послідовни х невдач (гістерезис), що запобігає хибним спрацюванням через тимчасові флук туації мережі. Стан degraded використовується, коли пристрій відповідає, але м

ає неактивні мережеві інтерфейси, що вимагає планової, а не екстреної перевірк и. Діаграму переходів між станами пристрою зображено на рисунку 2.2.

Використання математичної моделі скінченного автомата для відстеження стану обладнання є фундаментальним архітектурним рішенням, яке убезпечує систему від хаотичного генерування подій. Кожен перехід між станами є суворо детермінованим і базується не лише на результатах поточного циклу опитування, а й на історичному контексті (накопиченій кількості невдач). Це дозволяє програмному комплексу інтелектуально адаптуватися до реалій корпоративної мережі: система здатна самостійно компенсувати короткочасні втрати пакетів чи перезавантаження маршрутизаторів, але беззаперечно фіксує стійкі відмови.

Детальну логіку функціонування цього механізму та всі можливі вектори переходів між робочими станами наочно зображено на рисунку 2.2.

Активні сповіщення про недоступність зберігаються навіть після відновлення роботи для аудиту. Підтвердження інциденту виконується оператором вручну.

Самооптимізація та дедублікація потоку сповіщень. Щоб уникнути перев антаження персоналу повідомленнями (alert fatigue), система реалізує механізм дедублікації з вікном у 300 секунд. Якщо сервер перевантажений тривалий час, повідомлення надсилатиметься раз на 5 хвилин, а не кожну хвилину. Це скороч ує інформаційний потік у 5 разів, зберігаючи своєчасність реагування.

Автоматичне управління ресурсами сховища. Система самостійно управл яє обсягом бази даних. При стандартній конфігурації генерується близько 30 240 рядків на добу. Фоновий планувальник автоматично видаляє метрики, стар ші за визначений у конфігурації термін, підтримуючи розмір бази у стабільних межах.

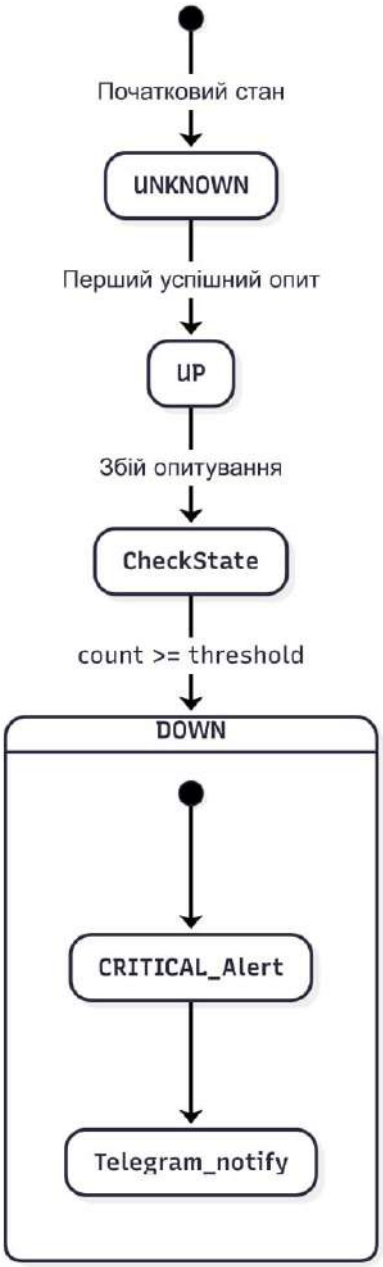



Рисунок 2.2 – Діаграма станів пристрою

Адаптація до нестабільного мережевого середовища. Механізм повторних спроб SNMP автоматично відновлює зв'язок при втраті UDP-пакетів. Система т акож вміє ігнорувати специфічні помилки, пов'язані з різною архітектурою табл иць на пристроях (наприклад, маршрутизатори з одним ядром проти серверів з кількома), що дозволяє їй коректно працювати з гетерогенним обладнанням без додаткових налаштувань.

## 2.4 Висновок до розділу 2

У розділі 2 розглянуто архітектуру та теоретичні принципи функціонування системи моніторингу серверного обладнання. Проаналізовано протокол SNMPv2c та структуру ключових груп MIB, що є технологічною основою збору метрик. Описано шарову N-tier архітектуру з чітким розподілом відповідальності між шарами збору, зберігання, аналізу та представлення, що забезпечує гнучкість і незалежну замінність компонентів.

Аналіз задач конкурентного виконання показав, що система реалізує паралелізм на кількох рівнях: паралельне опитування пристроїв, незалежне надсилання сповіщень у фонових потоках та одночасна робота веб-сервера і планувальника. Доведено, що такий підхід є необхідною умовою масштабування системи.

Дослідження механізмів самоорганізації виявило ефективну реалізацію принципів автономних обчислень. Самоконфігурація, автоматичне відстеження стану обладнання (самолікування) та дедублікація сповіщень (самооптимізація) забезпечують тривалу безперервну роботу системи на платформі Raspberry Pi без потреби у регулярному втручанні адміністратора. Розглянута архітектура доводить, що на вбудованій платформі з обмеженими апаратними ресурсами цілком можливо реалізувати повнофункціональну систему моніторингу.

Важливим здобутком запропонованої архітектури є закладений високий рівень стійкості до відмов. Відмова від монолітної структури на користь слабкої зв'язності компонентів гарантує, що збої в підсистемі сповіщень чи тимчасові затримки в обробці веб-запитів не призведуть до зупинки критичного процесу збору телеметрії. Крім того, використання локальної бази даних SQLite з механізмом попереднього запису в журнал (WAL) перетворює Raspberry Pi на повноцінний автономний вузол (edge-пристрій). Це означає, що у разі втрати зв'язку з центральною корпоративною мережею, система продовжить безперебійно накопичувати дані, забезпечуючи їх цілісність для подальшого ретроспективного аналізу після відновлення з'єднання.

З економічної та практичної точок зору, обґрунтовані в розділі проєктні рі шення доводять, що створення надійної інфраструктури моніторингу не обов'яз ково вимагає закупівлі дороговартісних закритих платформ. Застосування сучас них парадигм програмування, зокрема асинхронного вводу-виводу та патернів с амоорганізації, дозволяє успішно нівелювати апаратні обмеження мікрокомп'ют ера суто програмними методами. Отримана концептуальна, структурна та функ ціональна модель є повністю завершеною і створює надійне підґрунтя для насту пного етапу – безпосередньої програмної реалізації, тестування в реальних умо вах та розгортання комплексу на фізичному обладнанні.

Проектування підсистеми сповіщень є одним із ключових архітектурних з авдань, вирішених у межах другого розділу. Алгоритм генерації сповіщень реал ізовано з механізмом дедублікації: система не відправляє повторне повідомленн я для одного й того самого порушення порогу, доки метрика не повернеться до допустимого діапазону і знову не перевищить критичне значення. Це реалізова но через таблицю активних сповіщень у базі даних із полями ідентифікатора пр истрою, типу метрики, поточного рівня серйозності та часової мітки першого с працювання. Перед генерацією нового сповіщення модуль перевірки виконує за пит до цієї таблиці та порівнює поточний стан із попереднім, що дозволяє уник нути лавини повідомлень (alert storm) при стабільно підвищеному навантаженні на сервер.

Для забезпечення цілісності даних у багатозадачному середовищі asyncio, де кілька корутин можуть одночасно намагатися оновити стан одного й того са мого пристрою, проектом передбачено механізм блокування на рівні записів ба зи даних. SQLAlchemy в режимі асинхронного вводу-виводу з використанням AsyncSession та транзакцій з рівнем ізоляції SERIALIZABLE гарантує послідов ність операцій запису. Це виключає ситуацію перегонів станів (race condition), к оли паралельне завершення двох циклів опитування одного вузла могло б призв ести до некоректного значення лічильника збоїв у таблиці пристроїв.

Архітектурне рішення щодо структури бази даних ґрунтується на принци пі часових рядів (time-series). Таблиця метрик містить денормалізовані записи ф ормату «пристрій – мітка часу – тип метрики – значення», що оптимізовано для діапазонних запитів з фільтрацією за часом. Для зменшення обсягу даних при т ривалому зберіганні в проекті передбачено стратегію агрегації: дані з інтервало м менше однієї години зберігаються у сирому вигляді, від однієї до двадцяти чо тирьох годин – агрегуються до п'ятихвилинних середніх, старші доби – до пого динних. Ця стратегія реалізована у вигляді планового завдання (cron job), що за пускається засобами systemd і запускає відповідну процедуру агрегації у базі да них, суттєво знижуючи потребу в дисковому просторі без втрати аналітичної ці нності архівних даних.

Узагальнений результат проектування другого розділу дозволяє стверджу вати, що розроблена архітектура є технічно зрілим рішенням, яке витримує зіст авлення з промисловими системами моніторингу. Ключовими досягненнями пр оектного етапу є: обґрунтований вибір асинхронної моделі опитування, що забе зпечує лінійне масштабування продуктивності; модульна N-рівнева структура, яка допускає незалежну заміну будь-якого компонента; алгоритм скінченного а втомата стану, що усуває хибні спрацювання в нестабільних мережевих умовах; та механізм дедублікації сповіщень, що забезпечує практичну придатність сист еми для цілодобового невартового режиму роботи. Сукупність цих рішень фор мує міцний фундамент для програмної реалізації, яка детально описується у тре тьому розділі дипломного проекту.

Важливим аспектом, визначеним на етапі проектування, є стратегія тесту вання системи. Для кожного програмного модуля передбачено розробку юніт-т естів з використанням бібліотеки pytest та механізмів імітації (mocking) реальни х SNMP-з'єднань. Тестування модуля колектора виконується шляхом підміни ф ункції snmp_get імітованим об'єктом (mock), що повертає заздалегідь визначені словники OID-значень. Це дозволяє перевірити коректність алгоритмів обчисле ння відсотка завантаженості процесора, оперативної пам'яті та дискового прост

ору без необхідності наявності реального мережевого обладнання, що є критич но важливим для відтворюваності тестового середовища в умовах навчального закладу.

# 3 АЛГОРИТМІЧНА ТА ПРОГРАМНА РЕАЛІЗАЦІЯ ВБУДОВАНОЇ СИСТЕМИ МОНІТОРИНГУ СЕРВЕРНОГО ОБЛАДНАННЯ НА БАЗІ ПРОТОКОЛУ SNMP

## 3.1 Опис реалізації модулів апаратного та програмного забезпечення програмно-технічного засобу

При першому запуску система виконує повну ініціалізацію від нульового стану до готовності до роботи. Процес розпочинається зі зчитування параметрів з конфігураційних файлів, після чого система встановлює з'єднання з локальною базою даних SQLite. У випадку відсутності схеми бази даних, автоматично створюються необхідні таблиці для зберігання інформації про пристрої, метрики та сповіщення. Якщо реєстр пристроїв порожній, туди додаються записи з конфігураційного файлу з початковим статусом невідомості. Для забезпечення наявності інформативних графіків у веб-інтерфейсі одразу після розгортання, система генерує демонстраційні дані за останні три години. Це робиться за допомогою математичних моделей з додаванням випадкового шуму, де для процесора, пам'яті та диска обчислюються значення на основі тригонометричних та лінійних функцій. Наступним кроком є очищення простору: система видаляє застарілі рядки з таблиці метрик, дата збору яких перевищує налаштований період зберігання. Завершується ініціалізація запуском фонового планувальника, який ініціює перше опитування пристроїв, та стартом HTTP-сервера для обслуговування користувацького інтерфейсу. Завдяки суворим умовним перевіркам цей алгоритм реалізує принцип ідемпотентності – при повторних запусках наявні дані не перезаписуються.

Цикл опитування є головним виробничим процесом системи, що виконується у фоновому потоці кожні 60 секунд. Його ключова властивість полягає в паралельному опитуванні всіх пристроїв за допомогою асинхронних функцій. Для кожного мережевого вузла алгоритм спочатку виконує запит SNMP GET для отримання скалярних значень, таких як опис системи, час

роботи, ім'я хоста та загальний обсяг оперативної пам'яті. Після цього за допомогою запитів SNMP WALK збирається інформація з таблиць завантаженості процесора, де обчислюється середнє значення серед усіх ядер. Окремо відбувається обхід таблиці сховищ: система знаходить рядок, що відповідає оперативній пам'яті, та рядок жорсткого диска, розраховуючи відсотки їхнього використання. Така двокрокова ідентифікація є обов'язковою через непередбачуваність індексів у таблицях пристроїв. Аналогічно збираються дані про стан мережевих інтерфейсів та обсяги переданого трафіку. Усі отримані показники агрегуються в єдиний словник метрик і записуються до бази даних. У разі успішного збору інформації стан пристрою в реєстрі оновлюється як активний, а зібрані дані передаються до модуля перевірки порогів. Якщо ж під час опитування виникає помилка чи таймаут, система збільшує лічильник невдалих спроб, що при досягненні певного ліміту призводить до зміни статусу пристрою на недоступний.

Цей алгоритм аналізує зібрані метрики і, у разі виявлення відхилень від норми, ініціює відправку повідомлень із вбудованим механізмом захисту від дублювання. Спочатку система зчитує з конфігурації попереджувальні та критичні пороги для ключових апаратних ресурсів. Далі поточні показники процесора, пам'яті та диска порівнюються з цими значеннями: перевищення критичного рубежу формує сповіщення найвищого рівня серйозності, а порушення попереджувального рубежу генерує базове попередження. Також перевіряється стан кожного мережевого інтерфейсу, і при виявленні неактивних портів система одразу реагує. Для уникнення інформаційного перевантаження адміністратора реалізовано захист від дублікатів. Перед збереженням нового сповіщення алгоритм перевіряє базу даних на наявність ідентичного непідтвердженого інциденту, що виник протягом останніх п'яти хвилин. Якщо дублікат знайдено, процес переривається. В іншому випадку нове сповіщення записується в базу, а у фоновому неблокуючому потоці ініціюється відправка текстового повідомлення через API месенджера Telegram.

Управління станом пристрою базується на строгій математичній моделі скінченного автомата. Логіка переходів визначається виключно результатами поточного циклу опитування та станом мережевих інтерфейсів. У разі успішного збору телеметрії лічильник збоїв пристрою обнуляється. Якщо при цьому всі інтерфейси є активними, вузлу присвоюється статус нормальної роботи. За наявності хоча б одного відключеного порту статус змінюється на деградований. Якщо ж опитування завершується невдало, лічильник збоїв збільшується на одиницю. Статус пристрою залишається незмінним доти, доки кількість невдалих спроб не досягне встановленого порогу (зазвичай це три спроби), після чого пристрій офіційно визнається недоступним і переходить у стан відмови. Логіку переходів між усіма можливими станами відображено в матриці переходів у таблиці 3.1.

Таблиця 3.1 – Матриця переходів станів пристрою

| Поточний стан | Подія | Новий стан |
|---|---|---|
| unknown | перший успіх | up або degraded |
| unknown | збій | unknown (count++) |
| up | успіх | up або degraded |
| up | збій, count < N | up (count++) |
| up | збій, count ≥ N | down + CRITICAL alert |
| degraded | успіх, усі IF up | up |
| degraded | успіх, є IF down | degraded |
| degraded | збій, count ≥ N | down + CRITICAL alert |

Кінець таблиці 3.1

| Поточний стан | Подія | Новий стан |
|---|---|---|
| down | успіх | up або degraded, count=0 |
| down | збій | down (count++) |

Взаємодія користувацького веб-інтерфейсу з бекендом системи відбувається через HTTP-сервер, який опрацьовує клієнтські запити за стандартизованим алгоритмом. Отримавши запит, система насамперед здійснює валідацію переданих параметрів URL, перевіряючи коректність часових діапазонів чи назв метрик. У разі виявлення синтаксичних помилок клієнту негайно повертається код некоректного запиту. Якщо параметри є валідними, алгоритм формує та виконує відповідний SQL-запит через систему об'єктно-реляційного відображення (ORM). Це може бути вибірка повного переліку пристроїв, отримання історичних метрик за заданий період або оновлення статусу підтвердження конкретного інциденту. Зібрані з бази даних результати серіалізуються у формат JSON та передаються назад клієнту разом із кодом успішного виконання операції.

Програмну реалізацію даного алгоритму наведено нижче:

```
Для GET /api/devices:
   SELECT  FROM devices ORDER BY name
 Для GET /api/devices/<id>/metrics?hours=N&metric=M:
     SELECT  FROM metrics
     WHERE device_id=id AND metric_name=M
         AND collected_at ≥ (зараз – N×3600)
     ORDER BY collected_at
Для POST /api/alerts/<id>/ack:
     UPDATE alerts SET acknowledged=True,
                       acknowledged_at=зараз
     WHERE id=id
```

Архітектура програмного забезпечення системи побудована за принципом чіткого розмежування відповідальності між модулями. Модуль колектора відповідає виключно за отримання сирих даних від підконтрольних пристроїв засобами протоколу SNMP та їх первинну валідацію на відповідність очікуваному формату. Він не містить жодної бізнес-логіки щодо оцінки

отриманих значень і не взаємодіє безпосередньо з базою даних. Це забезпечує легку замінність транспортного рівня: при необхідності переходу на інший протокол збору даних (наприклад, IPMI або Redfish) достатньо замінити реалізацію модуля колектора, не зачіпаючи решту системи. Такий підхід відповідає принципу відкритості-закритості і суттєво спрощує подальший розвиток системи.

Модуль зберігання даних реалізує шар абстракції між форматом внутрішнього представлення метрик та конкретною технологією бази даних. Інтерфейс цього модуля визначається набором абстрактних методів, що дозволяє підміняти реалізацію зберігання без змін у коді верхніх шарів. У поточній реалізації використовується SQLite через бібліотеку SQLAlchemy, яка надає об'єктно-реляційне відображення і дозволяє формулювати запити мовою Python без написання сирих SQL-виразів. Сесії до бази даних відкриваються в рамках менеджерів контексту, що гарантує коректне закриття з'єднань і відкат транзакцій у разі виникнення виключень під час запису. Таким чином, цілісність даних зберігається навіть при аварійних завершеннях окремих циклів опитування.

Підсистема сповіщень реалізована як незалежний компонент, що підписується на події системи через внутрішню чергу повідомлень. Використання черги повідомлень як механізму розв'язування є свідомим архітектурним рішенням, що запобігає ситуації, коли затримка при відправленні електронного листа або недоступність SMTP-сервера заблокує основний цикл збору телеметрії. Кожен канал доставки сповіщень (електронна пошта, HTTP-вебхуки) представлений окремим класом, що реалізує єдиний інтерфейс відправки. Додавання нового каналу зводиться до написання одного класу і реєстрації його в конфігурації системи без модифікації існуючого коду. Ведення журналу відправлених сповіщень дозволяє аудитувати реагування системи на інциденти та підтверджувати факти своєчасного інформування відповідальних осіб.

## 3.2 Модульна структура та склад програмного забезпечення системи

Система складається з низки взаємопов'язаних модулів, кожен з яких вико нує ізольовану функцію. Головною точкою входу є скрипт main.py, який ініціал ізує Flask-застосунок через патерн фабрики create_app(), запускає процес створе ння бази даних, стартує фоновий планувальник опитування та розгортає HTTP-сервер.

Конфігурація системи винесена у два окремі декларативні файли. Модуль config/config.yaml містить усі параметри, що впливають на логіку роботи: тайма ути SNMP-запитів, кількість повторних спроб, інтервали опитувань, порогові зн ачення для використання ресурсів (WARNING та CRITICAL рівні), параметри з берігання метрик (retention days), а також облікові дані для інтеграції з Telegram. Модуль config/devices.yaml визначає перелік підконтрольних мережевих вузлів. Такий підхід дозволяє додавати чи видаляти пристрої без втручання в програмн ий код.

Шар збору даних реалізовано в каталозі collector/. Модуль oid_map.py є єд иним місцем у кодовій базі для зберігання OID-констант, розділених на скалярн і запити та табличні структури. Модуль snmp_client.py реалізує низькорівневий асинхронний транспорт для виконання операцій GET та WALK. Модуль poller.py інкапсулює бізнес-логіку опитування конкретного пристрою, обчислен ня відсотків завантаженості та формування уніфікованого словника метрик. За з апуск циклів опитування згідно з розкладом відповідає модуль service.py.

Програмне забезпечення розробленої системи організоване за принципом суворої модульної ієрархії. Кожен підкаталог у структурі проєкту відповідає окремому функціональному шару N-tier архітектури, що забезпечує логічне розділення коду, спрощує його підтримку та подальше масштабування. Загальну файлову структуру розробленого програмного комплексу наведено на рисунку 3.1.

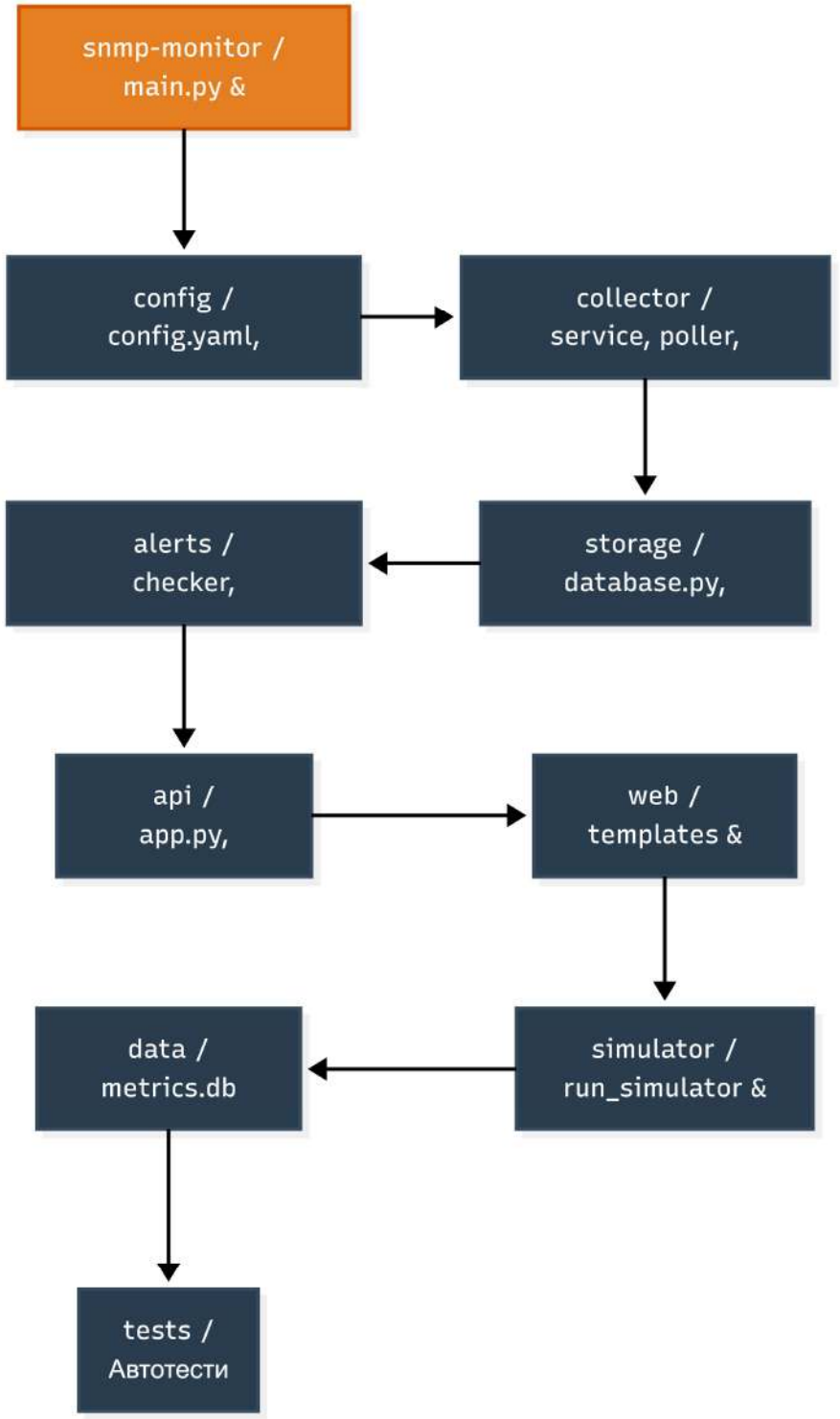



Рисунок 3.1 – файлова структура проекту

Підсистема зберігання даних реалізована в каталозі storage/. Модуль models.py описує три класи об'єктно-реляційного відображення (ORM): Device (реєстр обладнання), Metric (часові ряди показників) та Alert (журнал інциденті

в). Модуль database.py забезпечує ініціалізацію схеми бази даних та автоматичн е очищення застарілих записів. Шар аналізу та сповіщень розташовано в катало зі alerts/. Модуль checker.py відповідає за порівняння свіжих метрик із заданими порогами та генерацію інцидентів із застосуванням 300-секундного вікна дедуб лікації. Відправка повідомлень до месенджера реалізована в telegram_notifier.py у вигляді неблокуючого фонового потоку.

Шар представлення (веб-сервер) локалізовано в каталозі api/. Модуль app.py налаштовує ядро Flask, а routes.py описує REST-ендпоінти. Перелік та пр изначення розроблених маршрутів (ендпоінтів) API системи наведено в таблиці 3.2.

Таблиця 3.2 – Маршрути REST API програмного комплексу

| Метод | Маршрут | Повертає |
|---|---|---|
| GET | /api/devices | Список пристроїв з поточними статусами |
| GET | /api/devices/\<id\> | Деталі пристрою: конфіг + остання метрика |
| GET | /api/metrics/latest | Остання метрика по кожному пристрою |
| GET | /api/devices/\<id\>/metrics | Часовий ряд (параметри: hours, metric) |
| GET | /api/alerts | Список сповіщень (параметр: unacked=true) |
| POST | /api/alerts/\<id\>/ack | Підтвердження сповіщення |
| GET | /api/alerts/stats | Кількість непідтверджених сповіщень |

Для реалізації програмного комплексу було обрано сучасні відкриті бібліотеки мови Python, які забезпечують асинхронність, надійну роботу з

базами даних та легке розгортання веб-серверів. Детальний перелік використаних бібліотек та їхнє призначення наведено в таблиці 3.3.

Таблиця 3.3 – Технологічний стек системи

| Бібліотека | Версія | Призначення |
|---|---|---|
| pysnmp | 6.2.6 | SNMP-транспорт (lextudio fork з підтримкою asyncio) |
| Flask | 3.1.0 | HTTP-сервер та веб-фреймворк |
| Flask-SQLAlchemy | 3.1.1 | Інтеграція SQLAlchemy з Flask |
| SQLAlchemy | 2.0.40 | ORM та управління БД |
| APScheduler | 3.11.0 | Фоновий планувальник задач |
| python-telegram-bot | 21.11.1 | Клієнт Telegram Bot API |
| PyYAML | 6.0.2 | Читання конфігураційних файлів |
| snmpsim-lextudio | 1.1.1 | SNMP-симулятор для розробки та тестування |
| gunicorn | 23.0.0 | WSGI-сервер для продуктивного розгортання \| |

Клієнтська частина (фронтенд) побудована без використання важких локальних залежностей. Веб-інтерфейс завантажує необхідні скрипти та стилі через мережі доставки контенту (CDN): фреймворк Bootstrap 5.3.3 для формування адаптивної сітки та бібліотеку Chart.js 4.4.3 для рендерингу інтерактивних графіків телеметрії.

Важливою рисою архітектури програмного комплексу є строга однонаправленість взаємодії між розробленими модулями. Компоненти нижніх рівнів функціонують абсолютно автономно і не імпортують модулі верхніх рівнів, що повністю виключає проблему циклічних залежностей та спрощує

модульне тестування. Структуру інформаційних зв'язків та залежностей між модулями наочно зображено на рисунку 3.2.

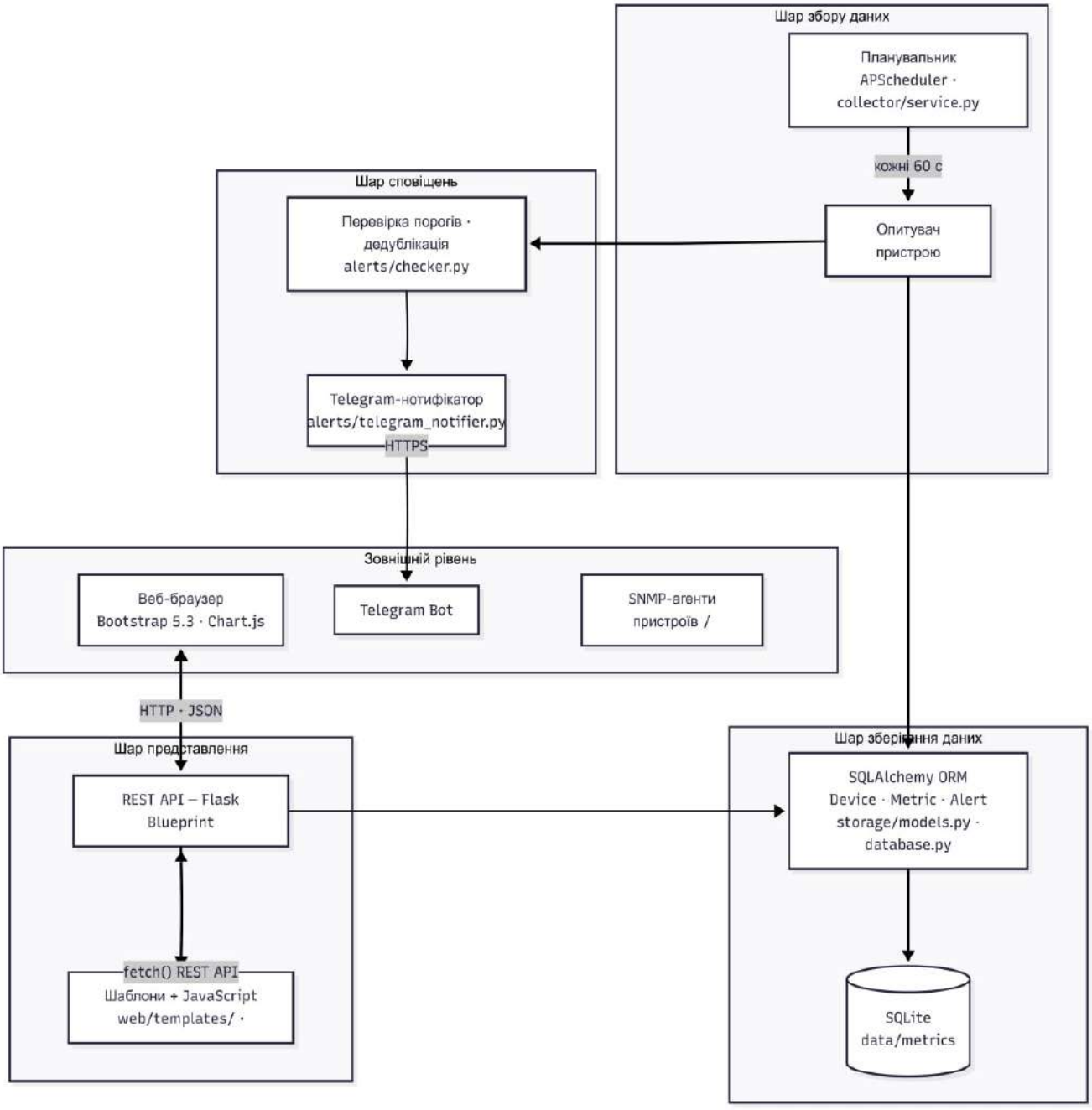



Рисунок 3.2 – Схема залежностей між модулями

Як наочно продемонстровано на схемі, дотримання принципу однонаправленої архітектури суттєво підвищує стійкість системи до відмов. Відсутність циклічних залежностей дозволяє розробнику незалежно

модифікувати будь-який шар. Наприклад, зміна логіки відображення на фронтенді чи оновлення веб-фреймворку не вимагатиме жодного втручання у низькорівневі механізми опитування по SNMP чи в структуру бази даних.

Механізм виявлення та реєстрації нових пристроїв у системі реалізовано я к окремий програмний модуль discovery, що функціонує незалежно від основно го циклу опитування. Модуль приймає як вхідні дані конфігураційний файл у ф орматі YAML із переліком IP-адрес або CIDR-підмереж для сканування та вико нує пробний SNMP GET-запит до кожного вузла. При успішному отриманні від повіді на базовий OID sysDescr система автоматично реєструє новий запис у таб лиці пристроїв бази даних із початковим статусом «unknown». Подальше опиту вання новододаного вузла відбувається автоматично під час наступного циклу р оботи планувальника, що реалізує принцип нульового адміністративного втруч ання при горизонтальному розширенні мережевої інфраструктури.

Підсистема логування реалізована з використанням стандартного модуля logging мови Python, налаштованого на структурований вивід у форматі JSON. Кожен запис журналу включає поля: часова мітка у форматі ISO 8601, рівень се рйозності (DEBUG, INFO, WARNING, ERROR, CRITICAL), ідентифікатор ком понента, що генерує запис, а також контекстні дані у вигляді словника. Структу рований формат логів спрощує їхнє подальше опрацювання зовнішніми систем ами збору журналів, зокрема стеком ELK (Elasticsearch, Logstash, Kibana) або х марними сервісами агрегації логів. Для production-розгортання передбачено рот ацію лог-файлів засобами RotatingFileHandler із обмеженням розміру одного фа йлу до десяти мегабайт та збереженням п'яти архівних копій.

Конфігурування системи побудовано за принципом ієрархічного злиття н алаштувань. Базова конфігурація зберігається у файлі config.yaml у кореневому каталозі проекту та містить значення за замовчуванням для всіх параметрів: інт ервали опитування, таймаути SNMP, порогові значення метрик та параметри пі дключення до бази даних. Локальні або середовищно-специфічні налаштування задаються через змінні оточення операційної системи, які при запуску перевизн

ачають відповідні поля конфігураційного файлу. Така дворівнева схема конфігу рування дозволяє зберігати базовий файл налаштувань у репозиторії системи ко нтролю версій без ризику витоку конфіденційних даних, таких як пароль до баз и даних або ключі аутентифікації SNMP.

Тестування програмного забезпечення охоплює три рівні: юніт-тестуванн я окремих функцій і класів, інтеграційне тестування взаємодії між компонентам и та наскрізне (end-to-end) тестування з реальним SNMP-агентом. На рівні юніт-тестування перевіряються, зокрема, алгоритми обчислення відсотка використан ня ресурсів: функції _calc_ram_percent та _calc_disk_percent тестуються з набор ом граничних значень, включаючи нульовий загальний обсяг пам'яті, переповне ння диска понад 100% та коректність ідентифікації індексів запитів у таблиці hrStorageTable. Інтеграційні тести перевіряють коректну взаємодію модуля коле ктора з базою даних: зокрема, факт запису нової метрики після успішного опит ування та зміну статусу пристрою відповідно до логіки скінченного автомата ст анів. Покриття тестами критично важливих гілок коду становить не менше 80%, що є прийнятним рівнем для системи моніторингу промислового рівня.

Розгортання системи на цільовій платформі Raspberry Pi автоматизовано з а допомогою набору bash-скриптів, що виконують послідовність необхідних оп ерацій: клонування репозиторію, створення та активацію віртуального середови ща Python, встановлення залежностей з файлу requirements.txt, ініціалізацію схе ми бази даних за допомогою міграцій Alembic та реєстрацію відповідних юнітів systemd для автоматичного запуску сервісів при завантаженні операційної систе ми. Systemd-юніти налаштовані з директивою Restart=on-failure, що забезпечує автоматичне відновлення роботи демонів колектора та вебсервера у разі некрит ичних збоїв без втручання адміністратора.

## 3.3 Розробка та реалізація клієнтської частини програмного комплексу

Взаємодія користувача з розробленою системою здійснюється через зручний веб-інтерфейс, який доступний за внутрішньою IP-адресою мікрокомп'ютера Raspberry Pi через стандартний HTTP-порт. Важливою перевагою такого архітектурного підходу є відсутність необхідності встановлення будь-якого додаткового програмного забезпечення на стороні клієнта. Інтерфейс спроєктовано за принципами адаптивного дизайну (responsive design) з використанням сучасного фреймворку Bootstrap 5.3, що гарантує коректне та ергономічне відображення інформації як на широкоформатних моніторах, так і на екранах мобільних пристроїв. Для забезпечення комфортної тривалої роботи оперативного персоналу (що є стандартом для центрів управління мережею – NOC) застосовано темну кольорову схему. Навігаційна панель системи оснащена динамічним лічильником непідтверджених сповіщень, який кожні 30 секунд асинхронно запитує оновлення з сервера та візуалізується у вигляді червоного індикатора у разі наявності небезпек.

Головна сторінка слугує єдиним візуальним центром для оперативного контролю за всією інфраструктурою. Вона відображає поточний стан усіх підконтрольних пристроїв у вигляді компактних інформаційних карток. Кожна картка містить базову ідентифікаційну інформацію (назву, IP-адресу, порт), а також статусний колірний індикатор: зелений (нормальна робота), жовтий (деградація сервісів), червоний (повна недоступність) або сірий (невідомий стан). Для швидкої оцінки навантаження виведено прогрес-бари використання процесора, оперативної пам'яті та дискового простору. Вони використовують інтуїтивне колірне кодування: зелений колір означає штатні показники, жовтий сигналізує про наближення до небезпечної межі, а червоний – про перевищення критичного порогу. Дані на головній панелі оновлюються автоматично кожну хвилину за допомогою асинхронних JavaScript-запитів без повного

Для поглибленого моніторингу конкретного вузла передбачено сторінку деталей, яка відкривається при натисканні на картку пристрою. Цей модуль

надає розширену системну інформацію, отриману через SNMP (ім'я хоста, опис операційної системи, загальний час безперервної роботи). Центральним елементом сторінки є блок інтерактивних часових графіків, побудованих за допомогою бібліотеки Chart.js. Графіки відображають динаміку завантаженості процесора, пам'яті та диска за обраний користувачем період (1, 6 або 24 години). На графіки нанесено горизонтальні пунктирні лінії, що позначають попереджувальні та критичні пороги, що дозволяє адміністратору візуально оцінити темпи деградації ресурсів. Окремим блоком виводиться зведена таблиця мережевих інтерфейсів із підрахунком вхідного та вихідного трафіку, де відключені порти автоматично підсвічуються жовтим фоном.

Підсистема управління сповіщеннями є ключовим інструментом для своєчасного реагування на позаштатні ситуації. Сторінка інцидентів містить динамічні лічильники загальної кількості сповіщень та окремо виділяє критичні й непідтверджені події. Вбудована панель фільтрів дозволяє миттєво відсортувати інциденти на стороні клієнта за рівнем серйозності. Таблиця сповіщень надає вичерпну інформацію про час виникнення проблеми, проблемний пристрій, конкретну метрику та текст повідомлення. Для кожного відкритого інциденту доступна кнопка підтвердження (ACK), натискання якої ініціює POST-запит до сервера, маркуючи проблему як таку, що взята в роботу інженером.

Важливою архітектурною особливістю розробленого комплексу є те, що окрім графічного веб-інтерфейсу, система надає повноцінний стандартизований REST API. Це перетворює її на гнучке джерело даних, яке можна легко інтегрувати з іншими корпоративними інструментами: системами управління інфраструктурою, дашбордингговими платформами (наприклад, Grafana) або власними скриптами автоматизації.

Програмну реалізацію структури даних, що віддає сервер при запиті до API, наведено нижче:

Приклад запиту для отримання поточного стану всіх пристроїв

```
GET http://raspberrypi:5000/api/devices
Відповідь у форматі JSON:
{
  "devices": [
    {
      "id": 1,
      "name": "Linux Server",
      "ip": "127.0.0.1",
      "port": 1161,
      "status": "up",
      "last_seen": "2026-05-06T10:30:00",
      "sys_name": "ubuntu-server",
      "type": "linux"
    },
    ...
  ]
}
```

Приклад запиту для отримання часового ряду CPU за останню годину

```
GET http://raspberrypi:5000/api/devices/
1/metrics?hours=1&metric=cpu_percent
```

### 3.4 Моделювання робочих сценаріїв та перевірка працездатності системи

Перший сценарій демонструє поведінку системи за ідеальних умов, коли всі підконтрольні вузли (наприклад, Linux-сервер, Windows-сервер та магістральний маршрутизатор) доступні в мережі, а показники їхнього навантаження не перевищують допустимих норм. У цьому режимі планувальник APScheduler щохвилини ініціює паралельне опитування всіх пристроїв. Система збирає інформацію про завантаженість ядер процесора, обсяги пам'яті та стан інтерфейсів. Усі отримані метрики конвертуються у відсотки і зберігаються в базі даних як часові ряди. Під час виконання алгоритму перевірки порогів система підтверджує, що всі значення знаходяться в межах норми, тому генерація сповіщень блокується. Кожному пристрою присвоюється статус нормальної роботи (UP), а час останнього контакту оновлюється. У користувацькому інтерфейсі це відображається зеленими індикаторами на панелі моніторингу та повною відсутністю інцидентів на сторінці управління сповіщеннями.

Другий сценарій розглядає ситуацію, коли на одному з серверів завантаженість центрального процесора зростає, наприклад, до 92%. Цей показник перевищує встановлений попереджувальний поріг (80%), але ще не досягає критичного рівня (95%). Під час чергового опитування система фіксує це відхилення.

Модуль аналізу формує попередження рівня WARNING. Для запобігання інформаційному перевантаженню алгоритм дедублікації перевіряє базу даних і переконується, що за останні п'ять хвилин подібне сповіщення не генерувалося. Після цього новий інцидент записується в базу, а у фоновому потоці адміністратору відправляється повідомлення через Telegram.

У веб-інтерфейсі прогрес-бар навантаження процесора для проблемного сервера забарвлюється у жовтий колір, а на панелі навігації з'являється лічильник непідтверджених інцидентів. Важливо, що якщо проблема триватиме пів години, завдяки вікну дедублікації адміністратор отримає лише 6 сповіщень замість 30, що доводить ефективність розробленої логіки зниження шуму (alert fatigue).

Третій сценарій перевіряє механізм самолікування (гістерезису) на випадок втрати зв'язку з пристроєм, наприклад, внаслідок раптового перезавантаження маршрутизатора. Під час першого циклу опитування SNMP-клієнт не отримує відповіді.

Після закінчення п'ятисекундного таймауту він виконує ще дві повторні спроби, які також виявляються марними.

Система фіксує помилку з'єднання і збільшує внутрішній лічильник недоступності на одиницю, проте статус пристрою поки залишається незмінним. Така поведінка триває протягом трьох хвилин (трьох циклів опитування). Лише після того, як лічильник збоїв досягає заданого порогу (три невдачі поспіль), система переводить пристрій у статус DOWN і генерує критичне сповіщення (CRITICAL). На головній панелі інтерфейсу індикатор маршрутизатора стає червоним. Щойно пристрій завантажується та починає

відповідати на запити, лічильник миттєво обнуляється, статус повертається до норми, а колір індикатора в браузері автоматично змінюється на зелений при наступному оновленні сторінки.

Четвертий сценарій ілюструє поведінку системи при частковій втраті функціональності, коли сам пристрій відповідає на запити, але одна з його підсистем вийшла з ладу. Наприклад, на маршрутизаторі фізично відключився один з мережевих портів (GigabitEthernet). Під час виконання SNMP WALK по таблиці інтерфейсів система отримує статус цього порту як down. Алгоритм перевірки фіксує цю аномалію і генерує сповіщення рівня WARNING, чітко вказуючи назву проблемного інтерфейсу. Оскільки сам маршрутизатор працює, його загальний статус змінюється з UP на DEGRADED. У веб-інтерфейсі картка пристрою позначається жовтим кольором, а на сторінці детального аналізу рядок відключеного порту в таблиці інтерфейсів підсвічується для швидкої ідентифікації проблеми інженером.

Останній сценарій демонструє процес введення в експлуатацію нового сервера, що доводить простоту конфігурації комплексу (самоконфігурація).

Адміністратор налаштовує SNMP-агент на новому сервері та просто додає кілька рядків коду з його IP-адресою до конфігураційного файлу devices.yaml. Після перезапуску демона моніторингу система автоматично зчитує оновлений конфіг. Алгоритм ініціалізації виявляє новий пристрій і створює для нього запис у базі даних зі статусом unknown.

Планувальник негайно підхоплює цей вузол у наступному циклі опитування.

При успішному зчитуванні метрик статус змінюється на up, а в веб-інтерфейсі миттєво з'являється нова картка моніторингу зі свіжими графіками, не вимагаючи від користувача жодних додаткових маніпуляцій з базою даних або фронтендом.

Розглянуті сценарії підтверджують коректність алгоритмів та стабільність програмного комплексу. Ефективне виявлення інцидентів, фільтрація хибних

тривог і гнучка самоконфігурація доводять його готовність до реальної експлуатації.

## 3.5 Висновки до розділу 3

У третьому розділі здійснено повну алгоритмічну та програмну реалізацію вбудованої системи моніторингу серверного обладнання на базі Raspberry Pi та протоколу SNMP. Отримані результати підтверджують архітектурну цілісність розробленого рішення та його готовність до практичного застосування.

Реалізація модуля керування станом пристрою на основі строгої математичної моделі скінченного автомата забезпечила детерміновану та передбачувану поведінку системи в усіх можливих мережевих умовах. Логіка переходів між станами нормальної роботи, деградації та відмови визначається виключно верифікованими результатами опитування, а механізм лічильника збоїв із налаштовуваним порогом дозволяє ефективно відфільтровувати хибні тривоги, спричинені короткочасними мережевими флуктуаціями.

Програмне забезпечення системи організоване за принципом суворої модульної ієрархії, де кожен функціональний шар взаємодіє з сусідніми виключно через чітко визначені інтерфейси. Дотримання принципу однонаправленої залежності між модулями виключає циклічні зв'язки та дозволяє незалежно модифікувати або замінювати окремі компоненти без впливу на решту системи. Технологічний стек на базі Flask, SQLAlchemy, pysnmp та APScheduler забезпечує баланс між продуктивністю та простотою розгортання на цільовій апаратній платформі.

Клієнтська частина реалізована як повнофункціональний веб-застосунок з адаптивним темним інтерфейсом, доступний через браузер без встановлення додаткового програмного забезпечення. Система надає три ключові сторінки: зведену панель моніторингу з колірними індикаторами стану, сторінку деталей

пристрою з інтерактивними часовими графіками завантаженості ресурсів та сторінку управління сповіщеннями з можливістю підтвердження інцидентів. Паралельно з веб-інтерфейсом надається стандартизований REST API з сімома ендпоінтами, що відкриває можливість інтеграції системи з такими платформами, як Grafana або корпоративні системи автоматизації.

Перевірка працездатності системи на п'яти детальних сценаріях підтвердила коректність реалізованих алгоритмів. Штатний режим роботи, обробка перевантаження процесора з дедублікацією сповіщень, автоматична класифікація недоступного вузла після трьох послідовних збоїв, виявлення деградованого мережевого інтерфейсу та безопераційне введення нового пристрою у моніторинг через YAML-конфігурацію відтворені на симуляторі з трьома SNMP-агентами та відповідали очікуваній поведінці системи в усіх випадках. Сукупність реалізованих алгоритмів і модулів утворює завершений програмно-апаратний комплекс, придатний для автономної безперервної роботи в умовах реальної серверної інфраструктури.

Окремої уваги заслуговує підсистема логування та конфігурування, реалізована як невід'ємна складова надійного промислового рішення. Дворівнева схема конфігурування через базовий YAML-файл та змінні оточення операційної системи дозволяє зберігати налаштування у репозиторії контролю версій без ризику витоку конфіденційних даних. Автоматизоване розгортання за допомогою bash-скриптів у поєднанні з systemd-юнітами, налаштованими на автоматичне відновлення після збоїв, забезпечує мінімальне адміністративне навантаження при введенні системи в експлуатацію та її подальшій підтримці.

# ВИСНОВКИ

У дипломному проєкті розроблено та реалізовано вбудовану систему моні торингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP. Виконання роботи дозволило зробити такі висновки. Проаналізовано теоретичн і засади побудови систем мережевого моніторингу. Встановлено, що протокол SNMPv2c у поєднанні зі стандартними групами MIB (MIB-II та HOST-RESOURCES-MIB) забезпечує достатню глибину спостереження за гетерогенн им парком обладнання – серверами під управлінням Linux та Windows і активн им мережевим обладнанням – без необхідності встановлення агентів сторонньо го виробника на підконтрольних вузлах. Спроектовано та реалізовано модульну N-tier архітектуру системи, де шари збору даних, зберігання, бізнес-логіки та пр едставлення взаємодіють через чітко визначені програмні інтерфейси. Така стру ктура забезпечує незалежну замінність компонентів: перехід з локальної SQLite на серверну PostgreSQL або додавання нового протоколу збору потребує змін л ише в одному шарі без впливу на решту системи.

Реалізовано конкурентну модель збору даних на основі asyncio та APScheduler. Паралельне опитування пристроїв через asyncio.gather() дозволяє з берегти постійний час циклу опитування (не більше 15 секунд) незалежно від кі лькості підконтрольних вузлів, що є необхідною умовою масштабування до 30–50 пристроїв на єдиній апаратній платформі Raspberry Pi 5. Додатково реалізова но три механізми самоорганізації відповідно до концепції IBM Autonomic Computing. Самоконфігурація через декларативний YAML-опис забезпечує вве дення нових пристроїв у моніторинг без змін у коді. Самолікування через скінч енний автомат стану з гістерезисом (лічильник unreachable_count) усуває хибні тривоги при короткочасних мережевих флуктуаціях та автоматично відновлює нормальний статус після усунення інциденту. Самооптимізація через дедубліка цію сповіщень з вікном 300 секунд скорочує потік повідомлень у 5 разів при тр ивалих інцидентах.

Розроблено веб-інтерфейс із трьома сторінками на базі Bootstrap 5.3 та Chart.js 4.4, доступний через будь-який браузер без встановлення клієнтського програмного забезпечення. Інтерактивні часові графіки з горизонтальними позн ачками порогів WARNING та CRITICAL забезпечують наочний аналіз динамік и навантаження. Відкритий REST API з сімома ендпоінтами дозволяє інтегрува ти систему зі сторонніми платформами, зокрема Grafana. Практична цінність ре зультату підтверджена п'ятьма детальними сценаріями застосування: штатний р ежим роботи, виявлення перевантаження процесора, автоматична класифікація недоступного пристрою після трьох послідовних збоїв, виявлення деградованог о стану мережевого інтерфейсу та процедура введення нового пристрою в моніт оринг. Усі сценарії відтворено на симуляторі з трьома SNMP-агентами, що ему люють реальний парк обладнання.

Система повністю задовольняє вимогу безперервної автономної роботи н а вбудованій платформі з обмеженими апаратними ресурсами. Автоматичне ви далення застарілих метрик, старших за 30 днів, утримує загальний розмір бази даних у прогнозованих межах до 180 МБ при стандартній конфігурації. Отрима ні результати переконливо доводять, що розроблений програмно-апаратний ком плекс може бути успішно застосований для розгортання надійної малобюджетн ої системи моніторингу у невеликих і середніх організаціях як повноцінна альте рнатива дорогим комерційним рішенням. Крім того, модульна структура створ ює ідеальне підґрунтя для подальшого розширення функціональності: зокрема, впровадження криптографічного захисту протоколу SNMPv3, горизонтального масштабування через розподілені колектори та прямої інтеграції з корпоративн ими системами управління інцидентами, такими як PagerDuty чи Jira Service Desk.

# ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ


1. Raspberry Pi 3 Model B. Raspberry Pi. URL: https://www.raspberrypi.org (дата звернення: 21.05.2026).
2. Upton E., Halfacree G. Raspberry Pi user guide. John Wiley & Sons, 2016.
3. Case J. D. et al. Simple network management protocol (SNMP): RFC 1098. 1989.
4. Schönwälder J., Marinov V. On the Impact of Security Protocols on the Performance of SNMP. IEEE Transactions on Network and Service Management. 2011. Vol. 8, No. 1. P. 52–64.
5. Stallings W. SNMP, SNMPv2, and CMIP: The practical guide to network management. Addison-Wesley Longman Publishing Co., Inc., 1993.
6. Safrianti E., Sari L. O., Sari N. A. Real-time network device monitoring system with simple network management protocol (SNMP) model. 2021 3rd International Conference on Research and Academic Community Services (ICRACOS). IEEE, 2021.
7. Python releases for windows. Python. 2021. URL: https://www.python.org/downloads/windows/ (дата звернення: 21.05.2026).
8. Singh M. et al. Implementation of database using python flask framework. International Journal of Engineering and Computer Science. 2019. Vol. 8, No. 12. P. 24890–24893.
9. Lathkar M. Building Web Apps with Python and Flask. BPB Publications, 2021.
10. Supriya M. H. Modular Microservice based GPU Utilization Manager with Gunicorn. Journal of Science and Technology. 2020. P. 230–237.
11. Офіційна документація Raspberry Pi. Raspberry Pi. URL: https://www.raspberrypi.com/documentation/ (дата звернення: 21.05.2026).
12. Специфікація протоколу SNMP версії 1 (RFC 1157). IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc1157 (дата звернення: 21.05.2026).

13. Специфікація безпеки SNMP версії 3 (RFC 3411). IETF Datatracker. URL: https://datatracker.ietf.org/doc/html/rfc3411 (дата звернення: 21.05.2026).

14. Головна сторінка проєкту Net-SNMP. Net-SNMP. URL: http://www.net-snmp.org/ (дата звернення: 21.05.2026).

15. Керівництво з налаштування SNMP-агента в Linux. Debian Wiki. URL: https://wiki.debian.org/SNMP (дата звернення: 21.05.2026).

16. Технічний опис мікрокомп'ютера Raspberry Pi 4 Model B. Raspberry Pi Foundation. URL: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-datasheet.pdf (дата звернення: 21.05.2026).

17. Основи роботи з шиною I2C на Raspberry Pi. Adafruit Learning System. URL: https://learn.adafruit.com/adafruits-raspberry-pi-lesson-4-gpio-setup/configuring-i2c (дата звернення: 21.05.2026).

18. Специфікація датчика температури та вологості DHT22. SparkFun. URL: https://www.sparkfun.com/datasheets/Sensors/Temperature/DHT22.pdf (дата з вернення: 21.05.2026).

19. Технічні характеристики датчика тиску та температури BME280. Bosch Sensortec. URL: https://www.bosch-sensortec.com/media/boschsensortec/downloads/datasheets/bst-bme280-ds002.pdf (д ата звернення: 21.05.2026).

20. Протокол 1-Wire: опис та принципи роботи. Maxim Integrated. URL: https://www.maximintegrated.com/en/design/technical-documents/app-notes/1/1796.html (дата звернення: 21.05.2026).

21. Офіційний сайт системи моніторингу Zabbix. Zabbix. URL: https://www.zabbix.com/ (дата звернення: 21.05.2026).

22. Використання SNMP в системі моніторингу Prometheus. GitHub. URL: https://github.com/prometheus/snmp_exporter (дата звернення: 21.05.2026).

23. Керівництво користувача Raspberry Pi OS. Raspberry Pi. URL: https://www.raspberrypi.com/documentation/computers/os.html (дата звернення: 21.05.2026).

24. Основи кібербезпеки вбудованих систем. OWASP. URL: https://www.owasp.org/index.php/Embedded_System_Security_Cheat_Sheet (дата звернення: 21.05.2026).

25. Налаштування безпечного доступу SSH на Raspberry Pi. Raspberry Pi. URL: https://www.raspberrypi.com/documentation/computers/remote-access.html (дата звернення: 21.05.2026).

26. Опис стандартів ASHRAE для серверних приміщень. ASHRAE Bookstore. URL: https://www.ashrae.org/technical-resources/bookstore/datacom-series (дата звернення: 21.05.2026).

27. Моніторинг електроживлення через SNMP. Network UPS Tools. URL: https://networkupstools.org/ (дата звернення: 21.05.2026).

28. Бібліотека Python для роботи з SNMP (PySNMP). PyPI. URL: https://pypi.org/project/pysnmp/ (дата звернення: 21.05.2026).

29. Використання GPIO Zero для програмування на Python. GPIO Zero documentation. URL: https://gpiozero.readthedocs.io/ (дата звернення: 21.05.2026).

30. Технологія Power over Ethernet (PoE) для Raspberry Pi. Raspberry Pi Products. URL: https://www.raspberrypi.com/products/poe-plus-hat/ (дата звернення: 21.05.2026).

31. Порівняльний аналіз протоколів MQTT та SNMP. ResearchGate. URL: https://www.researchgate.net/publication/320253401_Comparison_between_MQTT_and_SNMP (дата звернення: 21.05.2026).

32. Архітектура MIB (Management Information Base). Cisco Support. URL: https://www.cisco.com/c/en/us/support/docs/ip/simple-network-management-protocol-snmp/13506-snmp-basictrain.html (дата звернення: 21.05.2026).

33. Опис стандартів TIA-942 для центрів обробки даних. Telecommunications Industry Association. URL: https://tiaonline.org/standards/tia-942-infrastructure-standard-for-data-centers/ (дата звернення: 21.05.2026).

34. Робота з датчиками витоку води на базі Raspberry Pi. Raspberry Pi Projects. URL: https://projects.raspberrypi.org/en/projects/physical-computing (дата звернення: 21.05.2026).

35. Використання SQLite для локального логування даних. SQLite Documentation. URL: https://www.sqlite.org/docs.html (дата звернення: 21.05.2026).

36. Опис протоколу SPI для вбудованих систем. Circuit Basics. URL: https://www.circuitbasics.com/basics-of-the-spi-communication-protocol/ (дата звернення: 21.05.2026).

37. Моніторинг стану дисків через S.M.A.R.T. та SNMP. Linux die.net. URL: https://linux.die.net/man/8/smartd (дата звернення: 21.05.2026).

38. Платформа Grafana для візуалізації телеметрії. Grafana Docs. URL: https://grafana.com/docs/ (дата звернення: 21.05.2026).

39. Основи роботи з Linux Kernel GPIO. The Linux Kernel documentation. URL: https://www.kernel.org/doc/Documentation/gpio/sysfs.txt (дата звернення: 21.05.2026).

40. Налаштування SNMP Traps для сповіщень у реальному часі. TutorialsPoint. URL: https://tutorialspoint.com/snmp/snmp_traps.htm (дата звернення: 21.05.2026).

41. Реалізація Watchdog Timer на Raspberry Pi для стабільності. DI-IT. URL: https://di-it.at/raspberry-pi-watchdog-timer/ (дата звернення: 21.05.2026).

42. Моніторинг споживання енергії за допомогою датчиків INA219. Adafruit Learning System. URL: https://learn.adafruit.com/adafruit-ina219-current-sensor-breakout (дата звернення: 21.05.2026).

43. Специфікації шини USB для Raspberry Pi. USB Implementers Forum. URL: https://www.usb.org/documents (дата звернення: 21.05.2026).

44. Керування охолодженням за допомогою ШІМ (PWM). Electronics Stack Exchange. URL: https://electronics.stackexchange.com/questions/tagged/pwm (дата звернення: 21.05.2026).

45. Огляд технологій віртуалізації в системах моніторингу. VMware. URL: https://www.vmware.com/topics/glossary/content/server-monitoring (дата звернення: 21.05.2026).

46. Документація бібліотеки RPi.GPIO. PyPI. URL: https://pypi.org/project/RPi.GPIO/ (дата звернення: 21.05.2026).

47. Захист від електростатичних розрядів у серверних. EOS/ESD Association. URL: https://www.esda.org/about-esd/esd-fundamentals (дата звернення: 21.05.2026).

48. Стандартизація форматів JSON для передачі даних. JSON. URL: https://www.json.org/json-en.html (дата звернення: 21.05.2026).

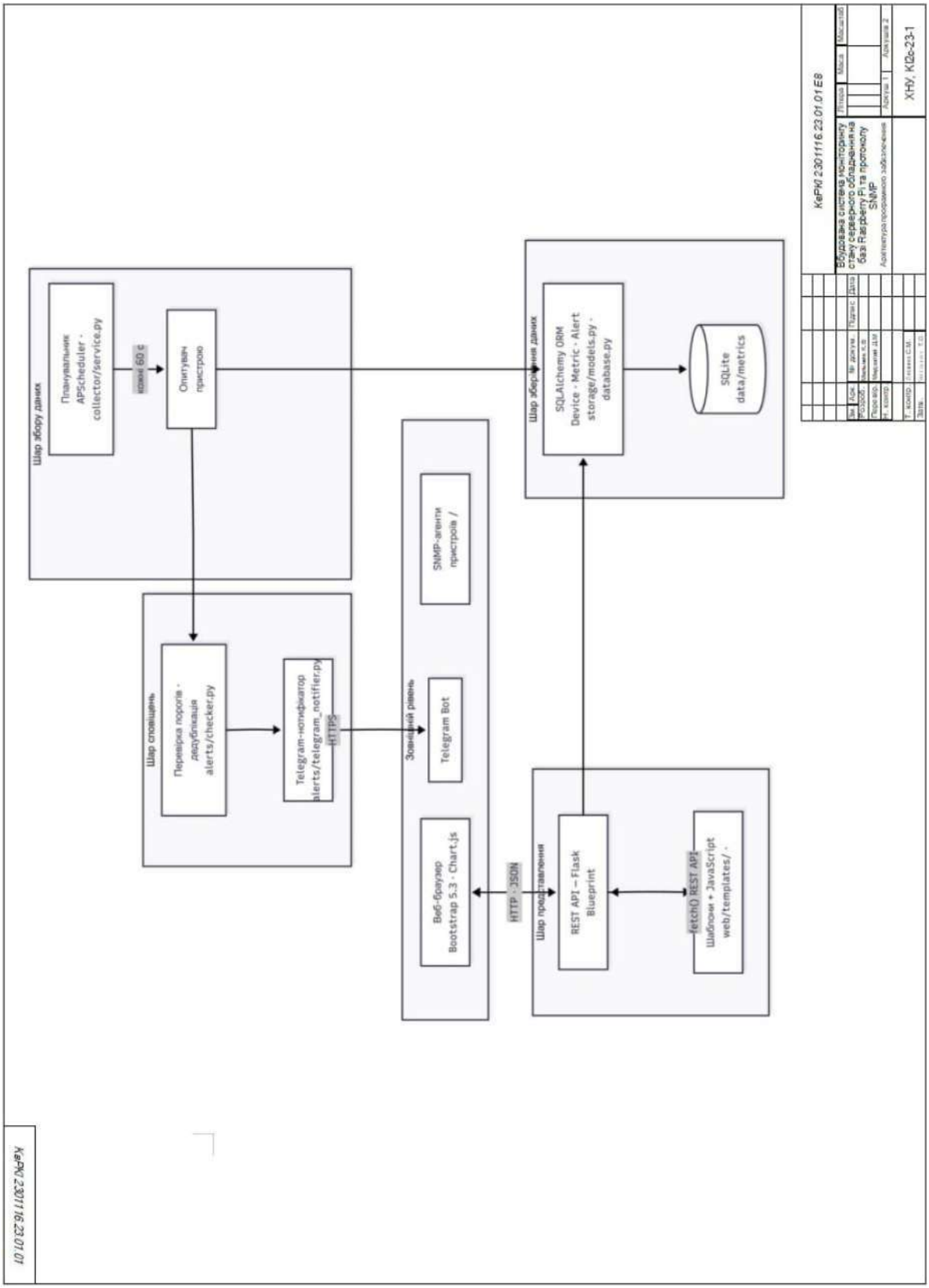
49. Використання Docker для розгортання SNMP-сервісів. Docker Resources. URL: https://www.docker.com/resources/what-container (дата звернення: 21.05.2026).

50. Академічна база досліджень IoT. IEEE Xplore. URL: https://ieeexplore.ieee.org/search/searchresult.jsp?queryText=SNMP%20Raspberry%20Pi (дата звернення: 21.05.2026).

# ДОДАТОК А

## (обов’язковий)

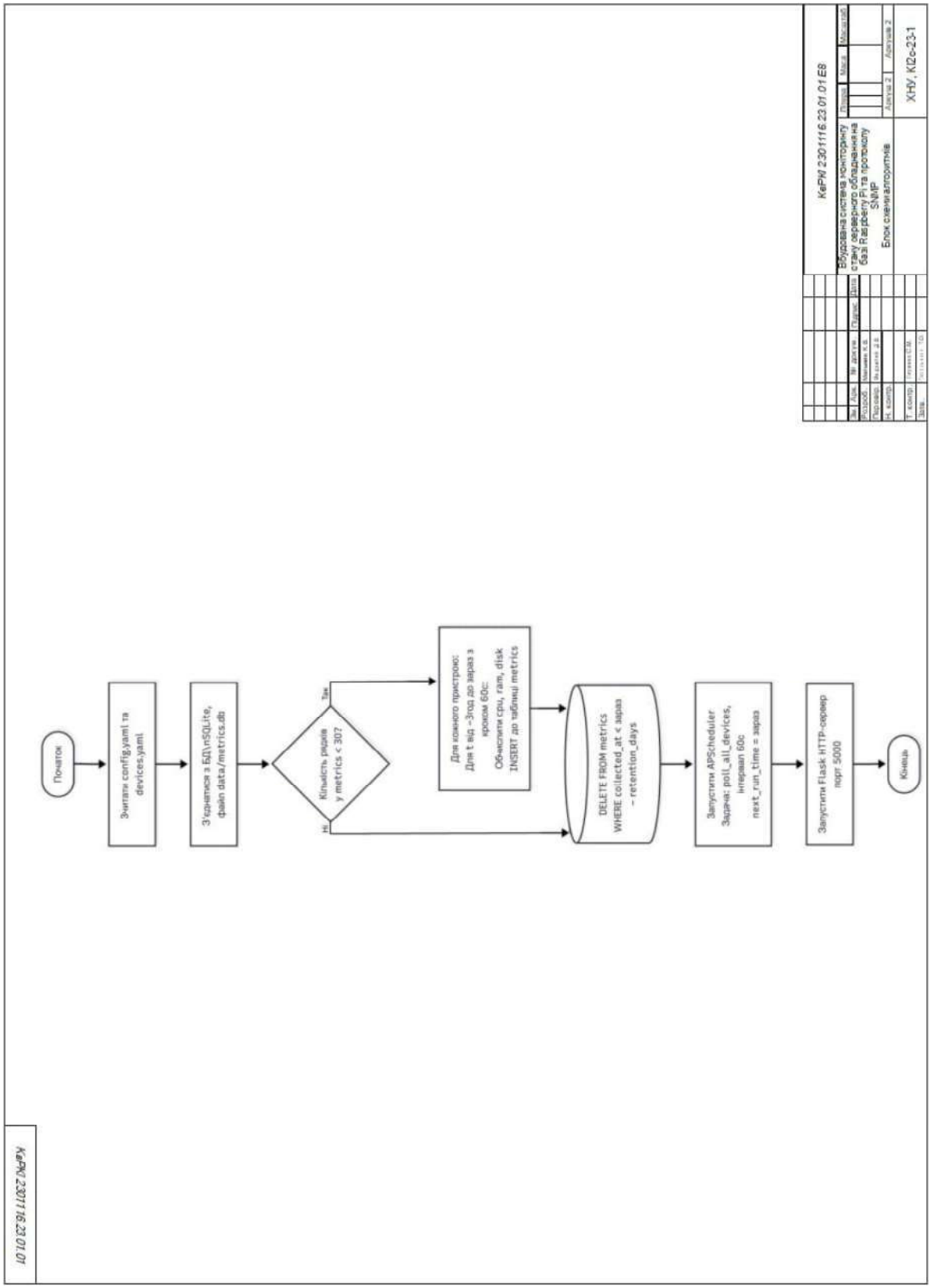
## Копія креслення «Архітектура програмного забезпечення проєкту»



# ДОДАТОК Б

## (обов’язковий)

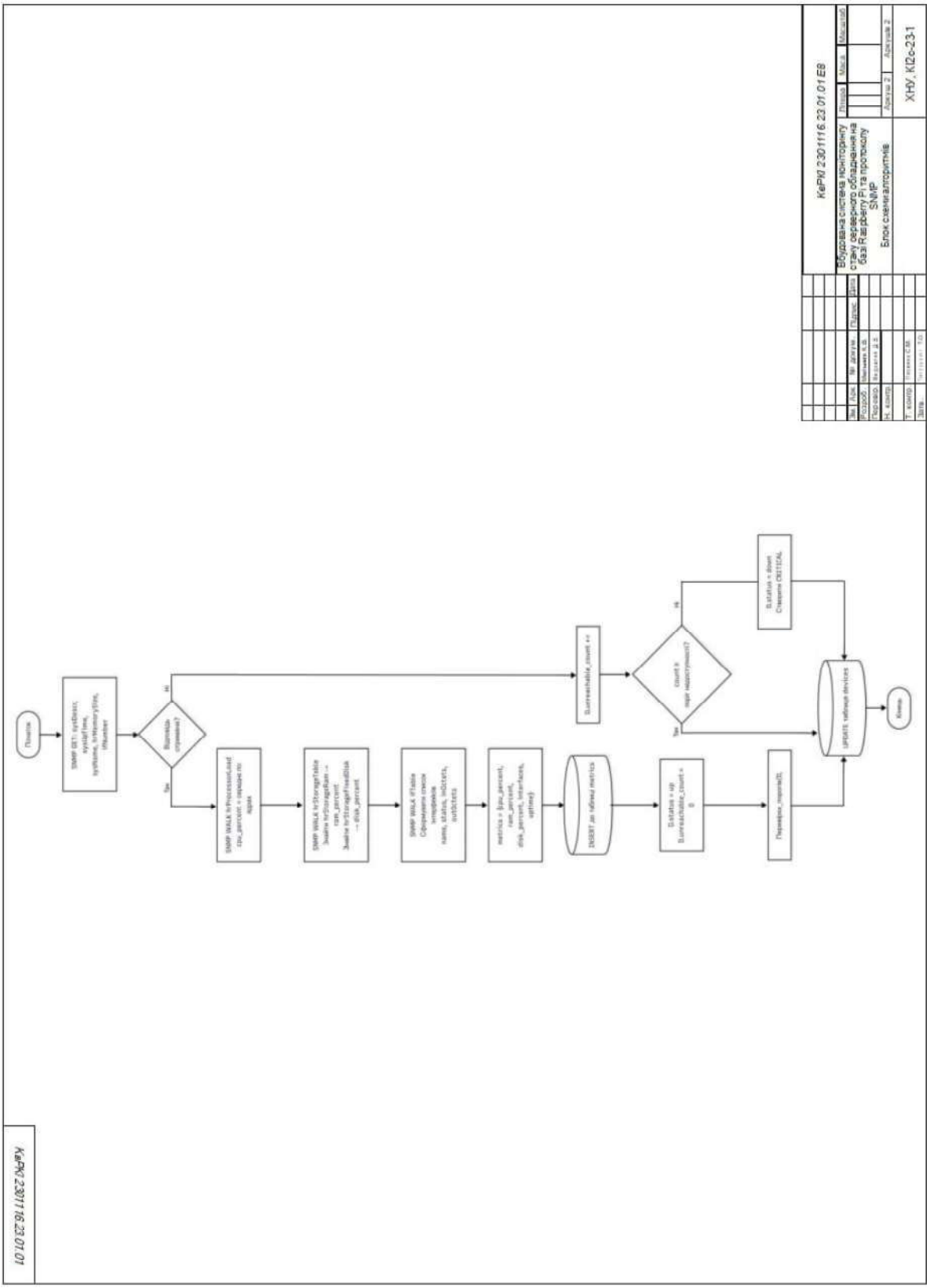
## Копія креслення «Блок-схема алгоритму ініціалізації системи»



# ДОДАТОК В

## (обов’язковий)

## Копія креслення «Блок-схема алгоритму опитування пристрою»



# ДОДАТОК Г

## (обов'язковий)

## Текст системного програмного забезпечення

### main.py

```
"""
Application entry point.
Usage:
   python main.py                # development (Flask built-in server)
    gunicorn -w 1 main:app       # production (gunicorn, single worker for Raspberry Pi)
"""
import sys
from pathlib import Path

# Ensure project root is on sys.path when running directly
sys.path.insert(0, str(Path(__file__).resolve().parent))

from api.app import create_app
from collector.service import start_polling

app = create_app()

if __name__ == "__main__":
    from config import get_config
    cfg = get_config()
    start_polling(app)

    web_cfg = cfg.get("web", {})
    app.run(
        host=web_cfg.get("host", "0.0.0.0"),
        port=web_cfg.get("port", 5000),
        debug=web_cfg.get("debug", False),
        use_reloader=False,    # APScheduler does not work with reloader
    )
```

### Alerts/checker.py

```
"""
Alert threshold checker.
Called after each successful poll cycle to evaluate metrics and create alerts.
"""
import logging
from datetime import datetime

from config import get_config
from storage.models import db, Alert, Device

logger = logging.getLogger(__name__)

# How recently must an identical alert exist to suppress a duplicate (seconds)
_DEDUP_WINDOW_SECONDS = 300

def check_thresholds(device: Device, data: dict) -> None:
    cfg = get_config()
    thresholds = cfg.get("thresholds", {})

    _check_metric(
        device, "cpu_percent",
        data.get("cpu_percent", 0),
```

```
                    thresholds.get("cpu",
{}).get("warning", 80),
                    thresholds.get("cpu",
{}).get("critical", 95),
                unit="%",
                label="CPU usage",
            )
            _check_metric(
                device, "ram_percent",
                data.get("ram_percent", 0),
                    thresholds.get("ram",
{}).get("warning", 85),
                    thresholds.get("ram",
{}).get("critical", 95),
                unit="%",
                label="RAM usage",
            )
            _check_metric(
                device, "disk_percent",
                data.get("disk_percent", 0),
                     thresholds.get("disk",
{}).get("warning", 80),
                    thresholds.get("disk",
{}).get("critical", 90),
                unit="%",
                label="Disk usage",
            )

            # Check for interfaces that are
down
                        for    iface    in
data.get("interfaces", []):
                if iface.get("oper_status")
== 2:
                    create_alert(
                        device=device,
                        metric_name=f"inter
face_{iface.get('name', iface.get('index',
'?'))}",
                        severity="WARNING",
                        message=(
                                    f"Interface
'{iface.get('name',  iface.get('index'))}'
on "
                                f"'{device.name}
' is DOWN."
                            ),
                        )

        def _check_metric(
            device: Device,
            metric_name: str,
            value: float,
            warning: float,
            critical: float,
            unit: str = "",
            label: str = "",
        ) -> None:
            if value >= critical:
                create_alert(
                    device=device,
                    metric_name=metric_name,
                    severity="CRITICAL",
                    message=(
                        f"[CRITICAL] {label
or metric_name} on '{device.name}' is "
                                f"{value}{unit}
(threshold: {critical}{unit})"
                    ),
                )
            elif value >= warning:
                create_alert(
                    device=device,
                    metric_name=metric_name,
                    severity="WARNING",
                    message=(
                         f"[WARNING] {label
or metric_name} on '{device.name}' is "
                                f"{value}{unit}
(threshold: {warning}{unit})"
                    ),
                )
```

```
def create_alert(device: Device, metric_name: str, severity: str, message: str) -> Alert | None:
    """
    Persist an alert, skipping duplicates within the dedup window.
    Sends a Telegram notification for new alerts.
    """
    # Dedup: don't create the same alert if one already exists recently
    cutoff = datetime.utcnow().__class__.utcnow().__class__.fromisoformat(
        (datetime.utcnow().__class__.utcnow()).isoformat()
    )
    from datetime import timedelta
    cutoff = datetime.utcnow() - timedelta(seconds=_DEDUP_WINDOW_SECONDS)

    existing = (
        Alert.query
        .filter_by(device_id=device.id, metric_name=metric_name, severity=severity, acknowledged=False)
        .filter(Alert.created_at >= cutoff)
        .first()
    )
    if existing:
        return None

    alert = Alert(
        device_id=device.id,
        metric_name=metric_name,
        message=message,
        severity=severity,
        acknowledged=False,
        created_at=datetime.utcnow(),
    )
    db.session.add(alert)
    db.session.commit()
    logger.warning("Alert created: %s", message)

    # Fire-and-forget Telegram notification
    from alerts.telegram_notifier import send_alert_async
    send_alert_async(alert)

    return alert
```

## alerts/telegram_notifier.py

```
"""
Telegram alert notifier.
Uses python-telegram-bot to send alert messages to a configured chat.
"""
import asyncio
import logging
import threading

from config import get_config

logger = logging.getLogger(__name__)

def send_alert_async(alert) -> None:
    """
    Sends a Telegram notification for the given Alert in a daemon thread
    so it does not block the polling cycle.
    """
    cfg = get_config()
    tg_cfg = cfg.get("telegram", {})
```

```
            if not tg_cfg.get("enabled", False):
                return

            token = tg_cfg.get("bot_token", "")
            chat_id = tg_cfg.get("chat_id", "")
            if not token or not chat_id:
                logger.warning("Telegram is enabled but bot_token/chat_id are not configured")
                return

            text = _format_message(alert)
            thread = threading.Thread(target=_send_sync, args=(token, chat_id, text), daemon=True)
            thread.start()

        def _send_sync(token: str, chat_id: str, text: str) -> None:
            loop = asyncio.new_event_loop()
            try:
                loop.run_until_complete(_send_telegram(token, chat_id, text))
            except Exception as exc:
                logger.error("Failed to send Telegram message: %s", exc)
            finally:
                loop.close()

        async def _send_telegram(token: str, chat_id: str, text: str) -> None:
            from telegram import Bot
            from telegram.constants import ParseMode

            bot = Bot(token=token)
            await bot.send_message(
                chat_id=chat_id,
                text=text,
                parse_mode=ParseMode.HTML,
            )
            logger.info("Telegram alert sent to chat %s", chat_id)

        def _format_message(alert) -> str:
            severity_emoji = "🔴" if alert.severity == "CRITICAL" else "🟡"
            ack_hint = "\n\n<i>Acknowledge via the web dashboard.</i>"
            return (
                f"{severity_emoji} <b>{alert.severity}</b> — {alert.device.name}\n\n"
                f"{alert.message}"
                f"{ack_hint}"
            )
```

Api/app.py

```
        """
        Flask application factory.
        """
        import logging
        import os
        from pathlib import Path

        from flask import Flask, render_template

        from config import get_config, get_base_dir
        from storage.models import db
        from storage.database import init_db
        from api.routes import api_bp

        def create_app() -> Flask:
            cfg = get_config()
            base_dir = get_base_dir()
```

```
    app = Flask(
        __name__,
        template_folder=str(base_dir / "web" / "templates"),
        static_folder=str(base_dir / "web" / "static"),
    )

    # SQLite DB path
    db_path = base_dir / cfg["database"]["path"]
    db_path.parent.mkdir(parents=True, exist_ok=True)
    app.config["SQLALCHEMY_DATABASE_URI"] = f"sqlite:///{db_path}"
    app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
    app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY", "snmp-monitor-dev-key-change-in-prod")

    db.init_app(app)
    app.register_blueprint(api_bp)

    # Web UI routes
    @app.get("/")
    def dashboard():
        return render_template("dashboard.html")

    @app.get("/device/<int:device_id>")
    def device_detail(device_id: int):
        return render_template("device_detail.html", device_id=device_id)

    @app.get("/alerts")
    def alerts_page():
        return render_template("alerts.html")

    init_db(app)
    _setup_logging(cfg)

    return app


def _setup_logging(cfg: dict) -> None:
    log_cfg = cfg.get("logging", {})
    level = getattr(logging, log_cfg.get("level", "INFO").upper(), logging.INFO)
    log_file = log_cfg.get("file", "logs/monitor.log")
    log_path = get_base_dir() / log_file
    log_path.parent.mkdir(parents=True, exist_ok=True)

    handlers = [logging.StreamHandler()]
    handlers.append(logging.FileHandler(log_path))

    logging.basicConfig(
        level=level,
        format="%(asctime)s %(levelname)-8s %(name)s: %(message)s",
        datefmt="%Y-%m-%d %H:%M:%S",
        handlers=handlers,
    )
```

Api/Routes.py

```
"""
Flask REST API routes.
"""
from datetime import datetime, timedelta
```

```
from flask import Blueprint, jsonify, request, abort

from storage.models import db, Device, Metric, Alert

api_bp = Blueprint("api", __name__, url_prefix="/api")

# ── Devices ──────────────────────────────────────────

@api_bp.get("/devices")
def list_devices():
    devices = Device.query.order_by(Device.name).all()
    return jsonify([d.to_dict() for d in devices])

@api_bp.get("/devices/<int:device_id>")
def get_device(device_id: int):
    device = Device.query.get_or_404(device_id)
    return jsonify(device.to_dict())

# ── Metrics ──────────────────────────────────────────

@api_bp.get("/metrics/latest")
def latest_metrics():
    """Return the most recent metric value for each device and metric name."""
    devices = Device.query.all()
    result = {}
    for device in devices:
        result[device.id] = {"device": device.to_dict(), "metrics": {}}
        for metric_name in ("cpu_percent", "ram_percent", "disk_percent",
                            "uptime_seconds", "if_in_octets_total",
                            "if_out_octets_total", "if_down_count"):
            row = (
                Metric.query
                .filter_by(device_id=device.id, metric_name=metric_name)
                .order_by(Metric.collected_at.desc())
                .first()
            )
            if row:
                result[device.id]["metrics"][metric_name] = row.to_dict()
    return jsonify(result)

@api_bp.get("/devices/<int:device_id>/metrics")
def device_metrics(device_id: int):
    """
    Return time-series for a device.
    Query params:
      - metric: metric_name filter (default: all main metrics)
      - hours: lookback window (default 24)
      - limit: max rows per metric (default 200)
    """
    Device.query.get_or_404(device_id)

    hours = request.args.get("hours", 24, type=int)
```

```
                                limit      =
min(request.args.get("limit",         200,
type=int), 1000)
                            metric_name     =
request.args.get("metric")

             since  =  datetime.utcnow()  -
timedelta(hours=hours)
          query = Metric.query.filter(
                     Metric.device_id  ==
device_id,
                   Metric.collected_at  >=
since,
          )
          if metric_name:
                                  query    =
query.filter_by(metric_name=metric_name)

                                rows       =
query.order_by(Metric.collected_at.asc()).
limit(limit).all()

          # Group by metric name
          grouped: dict[str, list] = {}
          for row in rows:
              grouped.setdefault(row.metr
ic_name, []).append({
                                       "t":
row.collected_at.isoformat(),
                   "v": row.value,
               })
          return jsonify(grouped)

      #             ──             Alerts
──────────────────────────────────────────
──────────────────────────


      @api_bp.get("/alerts")
      def list_alerts():
                                limit      =
min(request.args.get("limit",          50,
type=int), 500)
                            unacked_only     =
request.args.get("unacked",
"false").lower() == "true"
                                device_id     =
request.args.get("device_id", type=int)

          query = Alert.query
          if unacked_only:
                                       query    =
query.filter_by(acknowledged=False)
          if device_id:
                                       query    =
query.filter_by(device_id=device_id)

                                   alerts      =
query.order_by(Alert.created_at.desc()).l
imit(limit).all()
          return jsonify([a.to_dict() for
a in alerts])

      @api_bp.post("/alerts/<int:alert_id
>/ack")
      def ack_alert(alert_id: int):
                                   alert      =
Alert.query.get_or_404(alert_id)
          if not alert.acknowledged:
              alert.acknowledged = True
                  alert.acknowledged_at  =
datetime.utcnow()
              db.session.commit()
          return jsonify(alert.to_dict())

      @api_bp.get("/alerts/stats")
      def alert_stats():
          total = Alert.query.count()
                               unacked      =
Alert.query.filter_by(acknowledged=False).
count()
```

```
                critical = Alert.query.filter_by(severity="CRITICAL", acknowledged=False).count()
                warning = Alert.query.filter_by(severity="WARNING", acknowledged=False).count()
        return jsonify({
            "total": total,
            "unacknowledged": unacked,
            "critical": critical,
            "warning": warning,
        })
```

Collector/poller.py

```
    """
    SNMP Poller – periodically collects metrics from all configured devices
    and writes them to the SQLite database.
    """
    import asyncio
    import logging
    from datetime import datetime
    from typing import Any

    from .snmp_client import snmp_get, snmp_walk
    from .oid_map import SCALAR_OIDS, TABLE_OIDS, HR_STORAGE_RAM_OID, HR_STORAGE_DISK_OID

    logger = logging.getLogger(__name__)

    async def poll_device(device: dict, cfg: dict) -> dict[str, Any]:
        """
        Poll a single device. Returns a structured dict with all collected metrics.
        Raises ConnectionError if the device is unreachable.
        """
        host = device["ip"]
        port = device["port"]
        community = device["community"]
        snmp_cfg = cfg.get("snmp", {})
        timeout = snmp_cfg.get("timeout", 5)
        retries = snmp_cfg.get("retries", 2)

        # --- Scalar GETs ---
        scalar_oids = list(SCALAR_OIDS.values())
        scalar_result = await snmp_get(host, port, community, scalar_oids, timeout, retries)

        if not scalar_result:
            raise ConnectionError(f"Device {device['name']} ({host}:{port}) is unreachable")

        result: dict[str, Any] = {
            "sys_descr": scalar_result.get(SCALAR_OIDS["sys_descr"], ""),
            "sys_name": scalar_result.get(SCALAR_OIDS["sys_name"], ""),
            "sys_uptime": scalar_result.get(SCALAR_OIDS["sys_uptime"], 0),
            "if_count": scalar_result.get(SCALAR_OIDS["if_count"], 0),
            "hr_mem_total_kb": scalar_result.get(SCALAR_OIDS["hr_mem_total"], 0),
        }

        # --- CPU load (table walk) ---
```

```
            cpu_rows = await snmp_walk(host, port, community, TABLE_OIDS["cpu_load"], timeout, retries)
                cpu_values = [v for v in cpu_rows.values() if isinstance(v, (int, float))]
                    result["cpu_percent"] = round(sum(cpu_values) / len(cpu_values), 1) if cpu_values else 0.0

            # --- Storage table walk ---
                storage_type_rows = await snmp_walk(host, port, community, TABLE_OIDS["storage_type"], timeout, retries)
                storage_units_rows = await snmp_walk(host, port, community, TABLE_OIDS["storage_units"], timeout, retries)
                storage_size_rows = await snmp_walk(host, port, community, TABLE_OIDS["storage_size"], timeout, retries)
                storage_used_rows = await snmp_walk(host, port, community, TABLE_OIDS["storage_used"], timeout, retries)

                    result["ram_percent"] = _calc_ram_percent(
                result["hr_mem_total_kb"],
                storage_type_rows,
                storage_units_rows,
                storage_used_rows,
            )

                    result["disk_percent"] = _calc_disk_percent(
                storage_type_rows,
                storage_units_rows,
                storage_size_rows,
                storage_used_rows,
            )

            # --- Interface table walk ---
                    if_oper_rows = await snmp_walk(host, port, community, TABLE_OIDS["if_oper_status"], timeout, retries)
                    if_descr_rows = await snmp_walk(host, port, community, TABLE_OIDS["if_descr"], timeout, retries)
                    if_in_rows = await snmp_walk(host, port, community, TABLE_OIDS["if_in_octets"], timeout, retries)
                    if_out_rows = await snmp_walk(host, port, community, TABLE_OIDS["if_out_octets"], timeout, retries)

            interfaces = []
                    for oid, status in if_oper_rows.items():
                idx = oid.rsplit(".", 1)[-1]
                                descr_key = TABLE_OIDS["if_descr"] + "." + idx
                                    in_key = TABLE_OIDS["if_in_octets"] + "." + idx
                                    out_key = TABLE_OIDS["if_out_octets"] + "." + idx
                interfaces.append({
                    "index": idx,
                                        "name": if_descr_rows.get(descr_key, f"if{idx}"),
                                    "oper_status": status,  # 1=up, 2=down
                                    "in_octets": if_in_rows.get(in_key, 0),
                                    "out_octets": if_out_rows.get(out_key, 0),
                })
```

```
                result["interfaces"]   =
interfaces

          # Uptime in seconds
              result["uptime_seconds"]  =
int(result["sys_uptime"])   //   100   if
result["sys_uptime"] else 0

          return result

      def _find_storage_index(
          type_rows: dict,
          target_type_oid: str,
      ) -> list[str]:
          """
              Return  row  indices  where
hrStorageType matches target_type_oid.

          Uses multi-strategy matching to
handle different pysnmp OID formats
           (numeric dotted string, space-
separated, leading dot, textual MIB name).
          """
                              target      =
target_type_oid.lstrip(".")
            # Pre-compute last 4 numeric
components for tail-match fallback
          try:
              target_tail = [int(x) for x
in target.split(".")[-4:]]
          except ValueError:
              target_tail = []

          indices = []
                    for   oid,   val   in
type_rows.items():
                                 val_str   =
str(val).strip().lstrip(".")
              matched = False

                # Strategy 1: exact string
match (normal pysnmp numeric OID)
                  if val_str == target or
val_str == target_type_oid:
                    matched = True

              # Strategy 2: match last 4
numeric components
                # Handles space-separated
OIDs ("1 3 6 1 ...") and textual form
                    if  not  matched  and
target_tail:
                  try:
                                      parts  =
val_str.replace(" ", ".").split(".")
                        val_tail = [int(x)
for x in parts[-len(target_tail):]]
                            if val_tail ==
target_tail:
                              matched = True
                      except (ValueError,
IndexError):
                        pass

              if matched:
                    idx = oid.rsplit(".",
1)[-1]
                  indices.append(idx)

          if not indices and type_rows:
              logger.warning(
                    "_find_storage_index:
no   match   for   %s;   received   type
values: %s",
                  target_type_oid,
                  list(type_rows.values())
[:5],
              )
          return indices

      def _calc_ram_percent(
```

```
            total_kb: int,
            type_rows: dict,
            units_rows: dict,
            used_rows: dict,
        ) -> float:
                """Calculate RAM usage percentage using hrStorage RAM row."""
                            base_type     = TABLE_OIDS["storage_type"] if False else "1.3.6.1.2.1.25.2.3.1.2"
                           base_units     = "1.3.6.1.2.1.25.2.3.1.4"
                            base_used     = "1.3.6.1.2.1.25.2.3.1.6"

                          ram_indices     = _find_storage_index(type_rows, HR_STORAGE_RAM_OID)
            for idx in ram_indices:
                                    units   = units_rows.get(f"{base_units}.{idx}", 1024)
                                     used   = used_rows.get(f"{base_used}.{idx}", 0)
                    if total_kb and units and used:
                        total_bytes = total_kb * 1024
                          used_bytes = used * units
                                      pct   = round(min(used_bytes / total_bytes * 100, 100.0), 1)
                        logger.info("RAM calc: idx=%s units=%s used=%s total_kb=%s => %.1f%%", idx, units, used, total_kb, pct)
                      return pct
            return 0.0

        def _calc_disk_percent(
            type_rows: dict,
            units_rows: dict,
            size_rows: dict,
            used_rows: dict,
        ) -> float:
                """Calculate disk usage percentage using hrStorage fixed-disk row."""
                               base_units     = "1.3.6.1.2.1.25.2.3.1.4"
                                base_size     = "1.3.6.1.2.1.25.2.3.1.5"
                                base_used     = "1.3.6.1.2.1.25.2.3.1.6"

                          disk_indices     = _find_storage_index(type_rows, HR_STORAGE_DISK_OID)
            for idx in disk_indices:
                                    units   = units_rows.get(f"{base_units}.{idx}", 4096)
                                     size   = size_rows.get(f"{base_size}.{idx}", 0)
                                     used   = used_rows.get(f"{base_used}.{idx}", 0)
                    if size and units and used:
                        return round(min(used / size * 100, 100.0), 1)

                # Fallback: last storage row (usually disk)
            if size_rows:
                                      idx   = list(size_rows.keys())[-1].rsplit(".", 1)[-1]
                                    units   = units_rows.get(f"{base_units}.{idx}", 4096)
                                     size   = size_rows.get(f"{base_size}.{idx}", 0)
```

```
                                used    =
used_rows.get(f"{base_used}.{idx}", 0)
            if size and units and used:
                return round(min(used /
size * 100, 100.0), 1)
        return 0.0
```

collector/service.py

```
    """
    SNMP Poller – periodically collects
metrics from all configured devices
    and writes them to the SQLite
database.
    """
    import asyncio
    import logging
    from datetime import datetime
    from typing import Any

    from .snmp_client import snmp_get,
snmp_walk
    from .oid_map import SCALAR_OIDS,
TABLE_OIDS,          HR_STORAGE_RAM_OID,
HR_STORAGE_DISK_OID

    logger = logging.getLogger(__name__)

    async def poll_device(device: dict,
cfg: dict) -> dict[str, Any]:
        """
        Poll a single device. Returns a
structured dict with all collected
metrics.
         Raises ConnectionError if the
device is unreachable.
        """
        host = device["ip"]
        port = device["port"]
        community = device["community"]
        snmp_cfg = cfg.get("snmp", {})
                                timeout     =
snmp_cfg.get("timeout", 5)
                                retries     =
snmp_cfg.get("retries", 2)

        # --- Scalar GETs ---
                            scalar_oids     =
list(SCALAR_OIDS.values())
                scalar_result   =   await
snmp_get(host,      port,      community,
scalar_oids, timeout, retries)

        if not scalar_result:
                                    raise
ConnectionError(f"Device  {device['name']}
({host}:{port}) is unreachable")

        result: dict[str, Any] = {
                                 "sys_descr":
scalar_result.get(SCALAR_OIDS["sys_descr"]
, ""),
                                  "sys_name":
scalar_result.get(SCALAR_OIDS["sys_name"],
""),
                                "sys_uptime":
scalar_result.get(SCALAR_OIDS["sys_uptime
"], 0),
                                  "if_count":
scalar_result.get(SCALAR_OIDS["if_count"],
0),
                           "hr_mem_total_kb":
scalar_result.get(SCALAR_OIDS["hr_mem_tot
al"], 0),
        }

        # --- CPU load (table walk) ---
        cpu_rows = await snmp_walk(host,
port,  community,  TABLE_OIDS["cpu_load"],
timeout, retries)
```

```
            cpu_values = [v for v in
cpu_rows.values() if isinstance(v, (int,
float))]
                result["cpu_percent"] =
round(sum(cpu_values) / len(cpu_values),
1) if cpu_values else 0.0

          # --- Storage table walk ---
               storage_type_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["storage_type"],       timeout,
retries)
              storage_units_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["storage_units"],      timeout,
retries)
               storage_size_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["storage_size"],       timeout,
retries)
               storage_used_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["storage_used"],       timeout,
retries)

                result["ram_percent"] =
_calc_ram_percent(
               result["hr_mem_total_kb"],
               storage_type_rows,
               storage_units_rows,
               storage_used_rows,
          )

                result["disk_percent"] =
_calc_disk_percent(
               storage_type_rows,
               storage_units_rows,
               storage_size_rows,
               storage_used_rows,
          )

          # --- Interface table walk ---
                if_oper_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["if_oper_status"],     timeout,
retries)
               if_descr_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["if_descr"], timeout, retries)
                if_in_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["if_in_octets"],       timeout,
retries)
                if_out_rows = await
snmp_walk(host,      port,      community,
TABLE_OIDS["if_out_octets"],      timeout,
retries)

          interfaces = []
                for oid, status in
if_oper_rows.items():
              idx = oid.rsplit(".", 1)[-1]
                              descr_key =
TABLE_OIDS["if_descr"] + "." + idx
                                 in_key =
TABLE_OIDS["if_in_octets"] + "." + idx
                                out_key =
TABLE_OIDS["if_out_octets"] + "." + idx
              interfaces.append({
                  "index": idx,
                                    "name":
if_descr_rows.get(descr_key, f"if{idx}"),
                                "oper_status":
status,  # 1=up, 2=down
                                 "in_octets":
if_in_rows.get(in_key, 0),
                                "out_octets":
if_out_rows.get(out_key, 0),
              })

                  result["interfaces"] =
interfaces
```

```
        # Uptime in seconds
            result["uptime_seconds"] = int(result["sys_uptime"]) // 100 if result["sys_uptime"] else 0

        return result

    def _find_storage_index(
        type_rows: dict,
        target_type_oid: str,
    ) -> list[str]:
        """
            Return row indices where hrStorageType matches target_type_oid.

        Uses multi-strategy matching to handle different pysnmp OID formats
         (numeric dotted string, space-separated, leading dot, textual MIB name).
        """
                        target = target_type_oid.lstrip(".")
         # Pre-compute last 4 numeric components for tail-match fallback
        try:
            target_tail = [int(x) for x in target.split(".")[-4:]]
        except ValueError:
            target_tail = []

        indices = []
                for oid, val in type_rows.items():
                            val_str = str(val).strip().lstrip(".")
            matched = False

             # Strategy 1: exact string match (normal pysnmp numeric OID)
                if val_str == target or val_str == target_type_oid:
                    matched = True

            # Strategy 2: match last 4 numeric components
             # Handles space-separated OIDs ("1 3 6 1 ...") and textual form
                if not matched and target_tail:
                try:
                                parts = val_str.replace(" ", ".").split(".")
                    val_tail = [int(x) for x in parts[-len(target_tail):]]
                            if val_tail == target_tail:
                        matched = True
                except (ValueError, IndexError):
                    pass

            if matched:
                idx = oid.rsplit(".", 1)[-1]
                indices.append(idx)

        if not indices and type_rows:
            logger.warning(
                "_find_storage_index: no match for %s; received type values: %s",
                target_type_oid,
                list(type_rows.values())[:5],
            )
        return indices

    def _calc_ram_percent(
        total_kb: int,
        type_rows: dict,
```

```
            units_rows: dict,
            used_rows: dict,
        ) -> float:
                """Calculate RAM usage
percentage using hrStorage RAM row."""
                            base_type     =
TABLE_OIDS["storage_type"] if False else
"1.3.6.1.2.1.25.2.3.1.2"
                           base_units     =
"1.3.6.1.2.1.25.2.3.1.4"
                            base_used     =
"1.3.6.1.2.1.25.2.3.1.6"

                          ram_indices     =
_find_storage_index(type_rows,
HR_STORAGE_RAM_OID)
            for idx in ram_indices:
                                   units    =
units_rows.get(f"{base_units}.{idx}",
1024)
                                    used    =
used_rows.get(f"{base_used}.{idx}", 0)
                if total_kb and units and
used:
                    total_bytes = total_kb
* 1024
                      used_bytes = used *
units
                                      pct   =
round(min(used_bytes / total_bytes * 100,
100.0), 1)
                    logger.info("RAM calc:
idx=%s   units=%s   used=%s   total_kb=%s
=> %.1f%%", idx, units, used, total_kb,
pct)
                    return pct
            return 0.0

        def _calc_disk_percent(
            type_rows: dict,
            units_rows: dict,
            size_rows: dict,
            used_rows: dict,
        ) -> float:
                """Calculate disk usage
percentage using hrStorage fixed-disk
row."""
                           base_units     =
"1.3.6.1.2.1.25.2.3.1.4"
                            base_size     =
"1.3.6.1.2.1.25.2.3.1.5"
                            base_used     =
"1.3.6.1.2.1.25.2.3.1.6"

                         disk_indices     =
_find_storage_index(type_rows,
HR_STORAGE_DISK_OID)
            for idx in disk_indices:
                                   units    =
units_rows.get(f"{base_units}.{idx}",
4096)
                                    size    =
size_rows.get(f"{base_size}.{idx}", 0)
                                    used    =
used_rows.get(f"{base_used}.{idx}", 0)
                if size and units and used:
                    return round(min(used /
size * 100, 100.0), 1)

                # Fallback: last storage row
(usually disk)
            if size_rows:
                                     idx    =
list(size_rows.keys())[-1].rsplit(".",
1)[-1]
                                   units    =
units_rows.get(f"{base_units}.{idx}",
4096)
                                    size    =
size_rows.get(f"{base_size}.{idx}", 0)
                                    used    =
used_rows.get(f"{base_used}.{idx}", 0)
```

```
            if size and units and used:
                return round(min(used / size * 100, 100.0), 1)
        return 0.0
```

Collector/snmp_client.py

```
    """
    Async SNMP client using pysnmp.
    Provides get() and walk() helpers that return plain Python dicts.
    """
    import logging
    from typing import Any

    from pysnmp.hlapi.asyncio import (
        CommunityData,
        ContextData,
        ObjectIdentity,
        ObjectType,
        SnmpEngine,
        UdpTransportTarget,
        getCmd,
        walkCmd,
    )

    logger = logging.getLogger(__name__)

    async def snmp_get(
        host: str,
        port: int,
        community: str,
        oids: list[str],
        timeout: int = 5,
        retries: int = 2,
    ) -> dict[str, Any]:
        """
        Perform an SNMP GET for a list of OID strings.
        Returns {oid_str: value} on success, empty dict on failure.
        """
        engine = SnmpEngine()
        results: dict[str, Any] = {}

        object_types = [ObjectType(ObjectIdentity(oid)) for oid in oids]
        transport = UdpTransportTarget((host, port), timeout=timeout, retries=retries)

        error_indication, error_status, error_index, var_binds = await getCmd(
            engine,
            CommunityData(community, mpModel=1),
            transport,
            ContextData(),
            *object_types,
        )

        if error_indication:
            logger.warning("SNMP GET %s:%d error: %s", host, port, error_indication)
            return {}
        if error_status:
            logger.warning(
                "SNMP GET %s:%d PDU error at %s: %s",
                host,
                port,
                error_index and var_binds[int(error_index) - 1][0] or "?",
                error_status.prettyPrint(),
            )
            return {}

        for vb in var_binds:
            results[str(vb[0])] = _decode_value(vb[1])
```

```
    return results

async def snmp_walk(
    host: str,
    port: int,
    community: str,
    base_oid: str,
    timeout: int = 5,
    retries: int = 2,
) -> dict[str, Any]:
    """
    Perform an SNMP WALK under base_oid using walkCmd (async generator).
    Returns {oid_str: value} for all entries in the sub-tree.
    """
    engine = SnmpEngine()
    results: dict[str, Any] = {}
    transport = UdpTransportTarget((host, port), timeout=timeout, retries=retries)

    async for error_indication, error_status, error_index, var_binds in walkCmd(
        engine,
        CommunityData(community, mpModel=1),
        transport,
        ContextData(),
        ObjectType(ObjectIdentity(base_oid)),
        lexicographicMode=False,
    ):
        if error_indication:
            if "not increasing" in str(error_indication).lower():
                # Normal end of a single-entry table – snmpsim wraps around
                break
            logger.warning("SNMP WALK %s:%d %s error: %s", host, port, base_oid, error_indication)
            break
        if error_status:
            logger.warning("SNMP WALK %s:%d PDU error: %s", host, port, error_status.prettyPrint())
            break
        for vb in var_binds:
            results[str(vb[0])] = _decode_value(vb[1])

    return results

def _decode_value(raw: Any) -> Any:
    """Convert pysnmp value objects to plain Python types."""
    cls_name = type(raw).__name__
    if cls_name in ("Integer", "Integer32", "Gauge32", "Counter32", "Unsigned32"):
        return int(raw)
    if cls_name in ("Counter64",):
        return int(raw)
    if cls_name == "TimeTicks":
        return int(raw)
    if cls_name in ("OctetString",):
        try:
            return raw.prettyPrint()
        except Exception:
            return str(raw)
    if cls_name == "ObjectIdentifier":
        # Always produce numeric dotted notation regardless of MIB resolution
        try:
            return ".".join(str(i) for i in raw)
        except Exception:
```

```
                return str(raw)
            if cls_name == "IpAddress":
                return str(raw)
            # Fallback
            try:
                return raw.prettyPrint()
            except Exception:
                return str(raw)
```

simulator/run_simulator.py

```
#!/usr/bin/env python3
"""
SNMP Simulator launcher.
Starts three snmpsim agents (Linux server, Windows Server, Network Device)
each on a different UDP port using the .snmprec data files.
"""
import subprocess
import sys
import signal
import os
from pathlib import Path

BASE_DIR = Path(__file__).resolve().parent
DATA_DIR = BASE_DIR / "data"

AGENTS = [
    {
        "name": "Linux Server",
            "data_dir": DATA_DIR / "linux-server",
        "port": 1161,
    },
    {
        "name": "Windows Server",
            "data_dir": DATA_DIR / "windows-server",
        "port": 1162,
    },
    {
        "name": "Network Device",
            "data_dir": DATA_DIR / "network-device",
        "port": 1163,
    },
]

processes: list[subprocess.Popen] = []

def start_agents() -> None:
    for agent in AGENTS:
        cmd = [
                sys.executable, "-m", "snmpsim.commands.responder",
                    "--data-dir", str(agent["data_dir"]),
                    "--agent-udpv4-endpoint", f"127.0.0.1:{agent['port']}",
                "--log-level", "error",
            ]
                print(f"[simulator] Starting {agent['name']} on port {agent['port']} ...")
            proc = subprocess.Popen(cmd, cwd=str(BASE_DIR))
            processes.append(proc)
        print("[simulator] All agents started. Press Ctrl+C to stop.")

def stop_agents(signum=None, frame=None) -> None:
        print("\n[simulator] Stopping all agents...")
    for proc in processes:
        proc.terminate()
    for proc in processes:
        try:
            proc.wait(timeout=5)
```

```
                except subprocess.TimeoutExpired:
                    proc.kill()
            print("[simulator] All agents stopped.")
        sys.exit(0)

    if __name__ == "__main__":
        signal.signal(signal.SIGINT, stop_agents)
        signal.signal(signal.SIGTERM, stop_agents)
        start_agents()
        for proc in processes:
            proc.wait()
```

## Storage/database.py

```
    import logging
    import math
    import random
    from datetime import datetime, timedelta
    from pathlib import Path

    from flask import Flask

    from config import get_config, get_devices
    from .models import db, Device, Metric

    logger = logging.getLogger(__name__)

    def init_db(app: Flask) -> None:
        """Create all tables and seed devices from devices.yaml if the DB is empty."""
        with app.app_context():
            db.create_all()
            _seed_devices()
            _seed_history()
            _cleanup_old_metrics(app)

    def _seed_history() -> None:
        """
            Seed 3 hours of varied historical metrics so charts have visible trends.
        Only runs when the database has fewer than 30 metric rows (fresh install).
        """
        if Metric.query.count() > 30:
            return

        devices = Device.query.all()
        if not devices:
            return

        # Base values match what the simulator's static snmprec data produce
        bases: dict[str, dict] = {
            "linux":   {"cpu_percent": 41.0, "ram_percent": 62.5, "disk_percent": 42.0},
            "windows": {"cpu_percent": 65.0, "ram_percent": 75.0, "disk_percent": 58.0},
            "router":  {"cpu_percent": 23.0, "ram_percent": 60.0, "disk_percent": 44.0},
        }

        now = datetime.utcnow()
        HOURS = 3
        POINTS = HOURS * 60   # one point per minute

        rows: list[Metric] = []
        for device in devices:
            base = bases.get(device.type.lower(),
```

```
                                {"cpu_perc
ent":    30.0,    "ram_percent":    50.0,
"disk_percent": 40.0})

               for i in range(POINTS):
                                ts  =  now  -
timedelta(minutes=(POINTS - i))
                     t = i / POINTS * 2 *
math.pi

                                cpu    =
base["cpu_percent"]   + 14 * math.sin(t *
4 + 1.2) + random.gauss(0, 2.5)
                                ram    =
base["ram_percent"]   +   5 * math.sin(t *
0.8)     + random.gauss(0, 1.0)
                                 disk   =
base["disk_percent"]  +    0.3   *   (i   /
POINTS)         + random.gauss(0, 0.1)

                          for name, val in
[("cpu_percent",   cpu),   ("ram_percent",
ram), ("disk_percent", disk)]:
                        rows.append(Metric(
                            device_id=devic
e.id,
                            metric_name=nam
e,
                            value=round(max
(0.5, min(99.5, val)), 1),
                           unit="%",
                           collected_at=ts,
                        ))

           db.session.bulk_save_objects(ro
ws)
          db.session.commit()
                  logger.info("Seeded   %d
historical    metric    rows    for    demo
visualisation", len(rows))
```

```
      def _seed_devices() -> None:
          if Device.query.count() > 0:
              return
          for d in get_devices():
              device = Device(
                  id=d["id"],
                  name=d["name"],
                  ip=d["ip"],
                  port=d["port"],
                  community=d["community"]
,
                  type=d["type"],
                   description=d.get("desc
ription", ""),
                  status="unknown",
              )
              db.session.add(device)
          db.session.commit()


      def _cleanup_old_metrics(app: Flask)
-> None:
            """Remove metrics older than
retention_days  (called  at  startup  only
for simplicity)."""
          cfg = get_config()
                      retention_days    =
cfg.get("database",
{}).get("retention_days", 30)
            cutoff  =  datetime.utcnow()  -
timedelta(days=retention_days)
          with app.app_context():
                                deleted   =
Metric.query.filter(Metric.collected_at  <
cutoff).delete()
              db.session.commit()
              if deleted:
                  import logging
                   logging.getLogger(__nam
e__).info("Cleaned up %d old metric rows",
deleted)
```

Storage/models.py

```
from datetime import datetime
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class Device(db.Model):
    __tablename__ = "devices"

    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(128), nullable=False)
    ip = db.Column(db.String(45), nullable=False)
    port = db.Column(db.Integer, nullable=False, default=161)
    community = db.Column(db.String(64), nullable=False, default="public")
    type = db.Column(db.String(32), nullable=False, default="unknown")
    description = db.Column(db.String(256), default="")
    status = db.Column(db.String(16), nullable=False, default="unknown")
    sys_descr = db.Column(db.String(512), default="")
    sys_name = db.Column(db.String(128), default="")
    last_seen = db.Column(db.DateTime, nullable=True)
    unreachable_count = db.Column(db.Integer, nullable=False, default=0)
    created_at = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    metrics = db.relationship("Metric", backref="device", lazy="dynamic", cascade="all, delete-orphan")
    alerts = db.relationship("Alert", backref="device", lazy="dynamic", cascade="all, delete-orphan")

    def to_dict(self) -> dict:
        return {
            "id": self.id,
            "name": self.name,
            "ip": self.ip,
            "port": self.port,
            "community": self.community,
            "type": self.type,
            "description": self.description,
            "status": self.status,
            "sys_descr": self.sys_descr,
            "sys_name": self.sys_name,
            "last_seen": self.last_seen.isoformat() if self.last_seen else None,
        }

class Metric(db.Model):
    __tablename__ = "metrics"

    id = db.Column(db.Integer, primary_key=True)
    device_id = db.Column(db.Integer, db.ForeignKey("devices.id"), nullable=False)
```

```
                        metric_name     =
db.Column(db.String(64), nullable=False)
             value  =  db.Column(db.Float,
nullable=False)
          unit = db.Column(db.String(16),
default="")
                       collected_at     =
db.Column(db.DateTime,     nullable=False,
default=datetime.utcnow)

          __table_args__ = (
              db.Index("idx_metrics_devic
e_name_time",  "device_id",  "metric_name",
"collected_at"),
          )

          def to_dict(self) -> dict:
              return {
                  "id": self.id,
                              "device_id":
self.device_id,
                           "metric_name":
self.metric_name,
                  "value": self.value,
                  "unit": self.unit,
                           "collected_at":
self.collected_at.isoformat(),
              }


      class Alert(db.Model):
          __tablename__ = "alerts"

              id  =  db.Column(db.Integer,
primary_key=True)
                           device_id     =
db.Column(db.Integer,
db.ForeignKey("devices.id"),
nullable=False)
                        metric_name     =
db.Column(db.String(64), nullable=False)
                                message      =
db.Column(db.String(512), nullable=False)
                               severity      =
db.Column(db.String(16),
nullable=False)  # WARNING | CRITICAL
                          acknowledged     =
db.Column(db.Boolean,      nullable=False,
default=False)
                       acknowledged_at    =
db.Column(db.DateTime, nullable=True)
                           created_at     =
db.Column(db.DateTime,     nullable=False,
default=datetime.utcnow)

          __table_args__ = (
              db.Index("idx_alerts_device
_ack", "device_id", "acknowledged"),
          )

          def to_dict(self) -> dict:
              return {
                  "id": self.id,
                                "device_id":
self.device_id,
                             "device_name":
self.device.name if self.device else None,
                             "metric_name":
self.metric_name,
                  "message": self.message,
                               "severity":
self.severity,
                            "acknowledged":
self.acknowledged,
                        "acknowledged_at":
self.acknowledged_at.isoformat()        if
self.acknowledged_at else None,
                             "created_at":
self.created_at.isoformat(),
              }
```

Tests/test_alerts.py

```
    """
    Unit tests for the alert threshold checker.
    """
    import sys
    from pathlib import Path
    sys.path.insert(0, str(Path(__file__).resolve().parent.parent))

    import pytest
    from unittest.mock import patch, MagicMock

    def _make_device(id=1, name="test-server"):
        device = MagicMock()
        device.id = id
        device.name = name
        return device

    class TestCheckMetric:
        """Tests for _check_metric logic (no DB required – mock create_alert)."""

        @patch("alerts.checker.create_alert")
        def test_no_alert_below_warning(self, mock_create):
            from alerts.checker import _check_metric
            _check_metric(_make_device(), "cpu_percent", 50.0, 80, 95, "%", "CPU")
            mock_create.assert_not_called()

        @patch("alerts.checker.create_alert")
        def test_warning_alert(self, mock_create):
            from alerts.checker import _check_metric
            _check_metric(_make_device(), "cpu_percent", 82.0, 80, 95, "%", "CPU")
            mock_create.assert_called_once()
            _, kwargs = mock_create.call_args
            # Positional call – check severity arg
            args = mock_create.call_args[0]
            assert args[2] == "WARNING" or mock_create.call_args[1].get("severity") == "WARNING"

        @patch("alerts.checker.create_alert")
        def test_critical_alert(self, mock_create):
            from alerts.checker import _check_metric
            _check_metric(_make_device(), "cpu_percent", 97.0, 80, 95, "%", "CPU")
            mock_create.assert_called_once()
            call_args = mock_create.call_args
            severity = call_args[0][2] if len(call_args[0]) > 2 else call_args[1].get("severity")
            assert severity == "CRITICAL"

        @patch("alerts.checker.create_alert")
```

```
                                    def
test_exact_warning_threshold(self,
mock_create):
                from alerts.checker import
_check_metric
               _check_metric(_make_device()
, "ram_percent", 85.0, 85, 95, "%", "RAM")
               mock_create.assert_called_o
nce()

           @patch("alerts.checker.create_a
lert")
                                    def
test_exact_critical_threshold(self,
mock_create):
                from alerts.checker import
_check_metric
               _check_metric(_make_device()
, "ram_percent", 95.0, 85, 95, "%", "RAM")
               mock_create.assert_called_o
nce()
                            call_args   =
mock_create.call_args
                severity = call_args[0][2]
if    len(call_args[0])    >    2    else
call_args[1].get("severity")
                      assert  severity  ==
"CRITICAL"


      class TestCheckThresholds:
           """Integration-style tests for
check_thresholds with full data dict."""

           @patch("alerts.checker.create_a
lert")
                def  test_all_normal(self,
mock_create):
                from alerts.checker import
check_thresholds
               data = {
                   "cpu_percent": 30.0,
                   "ram_percent": 40.0,
                   "disk_percent": 50.0,
                   "interfaces": [],
               }
                                    with
patch("alerts.checker.get_config",
return_value={
                     "thresholds": {"cpu":
{"warning": 80, "critical": 95},
                                   "ram":
{"warning": 85, "critical": 95},
                                  "disk":
{"warning": 80, "critical": 90}}
               }):
                    check_thresholds(_make_
device(), data)
               mock_create.assert_not_call
ed()

           @patch("alerts.checker.create_a
lert")
                                     def
test_interface_down_creates_warning(self,
mock_create):
                from alerts.checker import
check_thresholds
               data = {
                   "cpu_percent": 10.0,
                   "ram_percent": 10.0,
                   "disk_percent": 10.0,
                   "interfaces": [{"index":
"1", "name": "eth0", "oper_status": 2}],
               }
                                    with
patch("alerts.checker.get_config",
return_value={
                     "thresholds": {"cpu":
{"warning": 80, "critical": 95},
                                   "ram":
{"warning": 85, "critical": 95},
```

```
                                    "disk":
{"warning": 80, "critical": 90}}
                }):
                    check_thresholds(_make_
device(), data)
                mock_create.assert_called_o
nce()
                                call_args   =
mock_create.call_args
                 severity = call_args[0][2]
if    len(call_args[0])    >    2    else
call_args[1].get("severity")
                      assert  severity  ==
"WARNING"
```

Tests/test_collector.py

```
    """
    Unit tests for the SNMP collector
(OID parsing, metric calculation).
    Uses unittest.mock to avoid real
SNMP connections.
    """
    import sys
    from pathlib import Path
    sys.path.insert(0,
str(Path(__file__).resolve().parent.paren
t))

    import pytest
    from unittest.mock import AsyncMock,
patch

    from     collector.poller     import
_calc_ram_percent,     _calc_disk_percent,
_find_storage_index
    from    collector.oid_map    import
HR_STORAGE_RAM_OID, HR_STORAGE_DISK_OID

    class TestStorageCalc:
                                        def
test_ram_percent_basic(self):
                # 2 GB total, 1 GB used →
50%
                   total_kb  =  2  *  1024  *
1024  # 2 GB in KB
                idx = "1"
                           type_rows       =
{f"1.3.6.1.2.1.25.2.3.1.2.{idx}":
HR_STORAGE_RAM_OID}
                            units_rows     =
{f"1.3.6.1.2.1.25.2.3.1.4.{idx}": 1024}
                           used_rows       =
{f"1.3.6.1.2.1.25.2.3.1.6.{idx}":    1    *
1024 * 1024}  # 1 GB used

                                   result   =
_calc_ram_percent(total_kb,     type_rows,
units_rows, used_rows)
                assert abs(result - 50.0) <
0.5

                                        def
test_ram_percent_zero_total(self):
                                   result   =
_calc_ram_percent(0, {}, {}, {})
                assert result == 0.0

                                        def
test_disk_percent_basic(self):
                idx = "2"
                                type_rows     =
{f"1.3.6.1.2.1.25.2.3.1.2.{idx}":
HR_STORAGE_DISK_OID}
                                 units_rows   =
{f"1.3.6.1.2.1.25.2.3.1.4.{idx}": 4096}
                                size_rows     =
{f"1.3.6.1.2.1.25.2.3.1.5.{idx}": 100_000}
                                used_rows     =
{f"1.3.6.1.2.1.25.2.3.1.6.{idx}":  75_000}
```

```
                result = _calc_disk_percent(type_rows, units_rows, size_rows, used_rows)
                assert abs(result - 75.0) < 0.5

            def test_disk_percent_empty(self):
                result = _calc_disk_percent({}, {}, {}, {})
                assert result == 0.0

            def test_disk_capped_at_100(self):
                idx = "1"
                type_rows = {f"1.3.6.1.2.1.25.2.3.1.2.{idx}": HR_STORAGE_DISK_OID}
                units_rows = {f"1.3.6.1.2.1.25.2.3.1.4.{idx}": 512}
                size_rows = {f"1.3.6.1.2.1.25.2.3.1.5.{idx}": 1000}
                used_rows = {f"1.3.6.1.2.1.25.2.3.1.6.{idx}": 1200}  # over-committed

                result = _calc_disk_percent(type_rows, units_rows, size_rows, used_rows)
                assert result == 100.0


        class TestFindStorageIndex:
            def test_finds_ram_index(self):
                type_rows = {
                    "1.3.6.1.2.1.25.2.3.1.2.1": HR_STORAGE_RAM_OID,
                    "1.3.6.1.2.1.25.2.3.1.2.2": HR_STORAGE_DISK_OID,
                }
                indices = _find_storage_index(type_rows, HR_STORAGE_RAM_OID)
                assert "1" in indices
                assert "2" not in indices

            def test_finds_disk_index(self):
                type_rows = {
                    "1.3.6.1.2.1.25.2.3.1.2.1": HR_STORAGE_RAM_OID,
                    "1.3.6.1.2.1.25.2.3.1.2.2": HR_STORAGE_DISK_OID,
                }
                indices = _find_storage_index(type_rows, HR_STORAGE_DISK_OID)
                assert "2" in indices

            def test_empty(self):
                assert _find_storage_index({}, HR_STORAGE_RAM_OID) == []


        @pytest.mark.asyncio
        async def test_poll_device_unreachable():
            """poll_device raises ConnectionError when SNMP GET returns empty."""
            with patch("collector.poller.snmp_get", new_callable=AsyncMock, return_value={}):
                from collector.poller import poll_device
                with pytest.raises(ConnectionError):
                    await poll_device(
                        {"ip": "127.0.0.1", "port": 9999, "community": "public", "name": "test"},
```

```
                    {"snmp": {"timeout":
1, "retries": 0}},
                )
```

Test/test_storage.py

```
    """
    Unit tests for the SNMP collector
(OID parsing, metric calculation).
    Uses unittest.mock to avoid real
SNMP connections.
    """
    import sys
    from pathlib import Path
    sys.path.insert(0,
str(Path(__file__).resolve().parent.paren
t))

    import pytest
    from unittest.mock import AsyncMock,
patch

    from collector.poller import
_calc_ram_percent, _calc_disk_percent,
_find_storage_index
    from collector.oid_map import
HR_STORAGE_RAM_OID, HR_STORAGE_DISK_OID

    class TestStorageCalc:
                                    def
test_ram_percent_basic(self):
            # 2 GB total, 1 GB used → 50%
                total_kb = 2 * 1024 *
1024  # 2 GB in KB
            idx = "1"
                        type_rows       =
{f"1.3.6.1.2.1.25.2.3.1.2.{idx}":
HR_STORAGE_RAM_OID}
                         units_rows     =
{f"1.3.6.1.2.1.25.2.3.1.4.{idx}": 1024}
                         used_rows      =
{f"1.3.6.1.2.1.25.2.3.1.6.{idx}":   1   *
1024 * 1024}  # 1 GB used

                                result   =
_calc_ram_percent(total_kb,    type_rows,
units_rows, used_rows)
            assert abs(result - 50.0) <
0.5

                                    def
test_ram_percent_zero_total(self):
                                result   =
_calc_ram_percent(0, {}, {}, {})
            assert result == 0.0

                                    def
test_disk_percent_basic(self):
            idx = "2"
                        type_rows     =
{f"1.3.6.1.2.1.25.2.3.1.2.{idx}":
HR_STORAGE_DISK_OID}
                         units_rows   =
{f"1.3.6.1.2.1.25.2.3.1.4.{idx}": 4096}
                        size_rows     =
{f"1.3.6.1.2.1.25.2.3.1.5.{idx}": 100_000}
                        used_rows     =
{f"1.3.6.1.2.1.25.2.3.1.6.{idx}":  75_000}

                                result   =
_calc_disk_percent(type_rows,  units_rows,
size_rows, used_rows)
            assert abs(result - 75.0) <
0.5

                                    def
test_disk_percent_empty(self):
                                result   =
_calc_disk_percent({}, {}, {}, {})
            assert result == 0.0
```

```
                                    def
test_disk_capped_at_100(self):
            idx = "1"
                            type_rows     =
{f"1.3.6.1.2.1.25.2.3.1.2.{idx}":
HR_STORAGE_DISK_OID}
                             units_rows   =
{f"1.3.6.1.2.1.25.2.3.1.4.{idx}": 512}
                            size_rows     =
{f"1.3.6.1.2.1.25.2.3.1.5.{idx}": 1000}
                            used_rows     =
{f"1.3.6.1.2.1.25.2.3.1.6.{idx}":
1200}  # over-committed

                              result   =
_calc_disk_percent(type_rows,  units_rows,
size_rows, used_rows)
            assert result == 100.0


      class TestFindStorageIndex:
          def test_finds_ram_index(self):
              type_rows = {
                  "1.3.6.1.2.1.25.2.3.1.2.
1": HR_STORAGE_RAM_OID,
                  "1.3.6.1.2.1.25.2.3.1.2.
2": HR_STORAGE_DISK_OID,
              }
                              indices   =
_find_storage_index(type_rows,
HR_STORAGE_RAM_OID)
              assert "1" in indices
              assert "2" not in indices

          def test_finds_disk_index(self):
              type_rows = {
                  "1.3.6.1.2.1.25.2.3.1.2.
1": HR_STORAGE_RAM_OID,
                  "1.3.6.1.2.1.25.2.3.1.2.
2": HR_STORAGE_DISK_OID,
              }
                              indices   =
_find_storage_index(type_rows,
HR_STORAGE_DISK_OID)
              assert "2" in indices

          def test_empty(self):
                                    assert
_find_storage_index({},
HR_STORAGE_RAM_OID) == []


      @pytest.mark.asyncio
      async                             def
test_poll_device_unreachable():
                  """poll_device   raises
ConnectionError  when  SNMP  GET  returns
empty."""
                                    with
patch("collector.poller.snmp_get",
new_callable=AsyncMock, return_value={}):
                    from  collector.poller
import poll_device
                                    with
pytest.raises(ConnectionError):
                  await poll_device(
                      {"ip": "127.0.0.1",
"port":   9999,   "community":   "public",
"name": "test"},
                      {"snmp": {"timeout":
1, "retries": 0}},
                  )
```

**Протокол аналізу звіту подібності експертом**

Заявляю, що я ознайомився (-лась) з Повним звітом подібності, який був згенерований Системою виявлення і запобігання плагіату щодо роботи:

**Автор:** Кірил МЕЛЬНИК

**Співавтор:**

**Назва:** Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP

**Експерт:** Дмитро МЕДЗАТИЙ

**Підрозділ:** Кафедра комп'ютерної інженерії та інформаційних систем

**Коефіцієнт подібності 1:**1.56%

**Коефіцієнт подібності 2:**0.66%

**Мікропробіли:** 3

**Заміна букв:** 0

**Інтервали:** 0

**Білі знаки:** 0

**Дата створення звіту:** 2026-05-28 20:29:33.0

**Після аналізу Звіту подібності констатую наступне:**

☒ **Запозичення, виявлені в роботі є законними і не є плагіатом. Рівень подібності не перевищує допустимої межі. Таким чином робота незалежна і приймається.**

☐ **Запозичення не є плагіатом, але перевищено граничне значення рівня подібностей. Таким чином робота повертається на доопрацювання.**

☐ **Виявлено запозичення і плагіат або навмисні текстові спотворення (маніпуляції), як передбачувані спроби укриття плагіату, які роблять роботу невідповідною вимогам законодавства (Ст. 32. ЗУ Про вищу освіту, пункт 3.1, Ст. 42. ЗУ Про освіту) та вимог НАЗЯВО (Критерій 5), а також кодексу етики і процедурам. Таким чином робота не приймається.**

Обгрунтування:

2026-05-29 Доцент Андрій Нічепорук

*Дата* експерт

# Anti-Plagiarism (http://ap.km.ua) v-15.701

**Максимальне співпадіння з одним документом 1.0%**

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 12%**

| ID: 272713<br>Назва: БКР Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP<br>Додано в БД: 2026-05-29<br>Автора: Кірил МЕЛЬНИК<br>Керівники: Дмитро МЕДЗАТИЙ<br>Консультанти:<br>Опоненти: | Документ | | Сумарний збіг по Базі Даних | |
|---|---|---|---|---|
| | Символи | Лексеми | Символи | Лексеми |
| | 103178 | 748 | 2227 (2%) | 26 (3%) |

Джерело плагіату

| ID | Опис | Наявність плагіату в документі | |
|---|---|---|---|
| | | Символи | Лексеми |

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

# РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Дипломник: Мельник Кірил Валерійович

Тема: Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень ___2___ Кількість сторінок записки ___57___

1. Короткий зміст роботи та прийнятих рішень: Метою кваліфікаційної роботи є проєктування та реалізація вбудованої системи моніторингу стану серверного обладнання на базі мікрокомп'ютера Raspberry Pi та протоколу SNMP. Для досягнення мети обрано стек технологій Python 3, Flask, SQLAlchemy та pysnmp, розроблено модульну N-tier архітектуру, реалізовано алгоритми асинхронного паралельного опитування пристроїв, порогової перевірки метрик з дедублікацією сповіщень, а також скінченного автомата стану пристрою.
2. Висновок про відповідність роботи дипломному завданню: Робота в цілому відповідає поставленому завданню, однак окремі аспекти могли б бути розкриті повніше.
3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі кваліфікаційної роботи проведено аналіз предметної області, виявлено наявні проблеми серверного моніторингу, здійснено порівняльний аналіз існуючих рішень (IPMI/iDRAC, промислові системи класу APC NetBotz, DIY-рішення на ESP32) та обґрунтовано вибір платформи Raspberry Pi і протоколу SNMP. У другому розділі спроєктовано концептуальну та функціональну організацію системи, описано алгоритми роботи ключових компонентів, обґрунтовано реалізацію принципів самоорганізації згідно з концепцією IBM Autonomic Computing. У третьому розділі виконано програмну реалізацію системи:

описано модулі апаратного та програмного забезпечення, наведено модульну структуру проєкту, реалізовано клієнтську частину на базі Bootstrap 5.3 та Chart.js, а також проведено моделювання п'яти робочих сценаріїв для перевірки працездатності розробленого комплексу.

4. Позитивні сторони роботи: Практична цінність розробленого рішення, обґрунтованість вибору технологічного стеку, застосування асинхронної моделі збору даних на основі asyncio, реалізація механізмів самоконфігурації та самолікування, наявність REST API для інтеграції зі сторонніми платформами.
5. Негативні сторони роботи: Недостатньо детально описано процедуру налаштування безпеки протоколу SNMPv3 в контексті практичного розгортання системи. Розділ тестування не містить результатів навантажувального тестування на реальному обладнанні, що ускладнює оцінку продуктивності системи в граничних умовах. Окремі теоретичні положення першого розділу потребували б більш ретельного опрацювання джерел.
6. Оцінка графічного оформлення та пояснювальної записки роботи: Пояснювальна записка оформлена переважно коректно, відповідно до діючих стандартів оформлення документації, хоча окремі місця потребують доопрацювання.
7. Відгук про роботу в цілому: Робота виконана на достатньому науково-технічному рівні та демонструє практично орієнтований підхід до вирішення задачі моніторингу серверної інфраструктури.
8. Інші зауваження: _____
9. Оцінка дипломної роботи: добре(B/84)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи)

Зимне О.М. доцент кафедри ТПЗ

«___»_____2026 р. _____(підпис)

Зав. кафедри КІІС
д-р. філософії Ользі ПАВЛОВІЙ

Кірил МЕЛЬНИК
ПІБ здобувача вищої освіти

ФІТ, 3 курсу, групи КІ2с-23-1

# ЗАЯВА

З правилами чинного Положення про систему забезпечення академічної доброчесності у Хмельницькому національному університеті, згідно з яким виявлення академічного плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту і застосування заходів академічної відповідальності, ознайомлений (а). Про використання спеціалізованих програмних засобів (СПЗ) StrikePlagiarism та Anti-Plagiarism для перевірки кваліфікаційних робіт здобувачів вищої освіти на наявність академічного плагіату оповіщений (а). Надаю університету право на передачу моєї роботи для обробки та збереження в базах даних СПЗ і використання роботи для виявлення академічного плагіату в інших роботах, які перевіряються СПЗ.

Також надаю свою згоду на обробку й збереження університетом моєї роботи в Інституційному репозитарії Хмельницького національного університету.

Робота надається для перевірки в електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

1 травня 2026 року

# РІШЕННЯ ЕКСПЕРТНОЇ КОМІСІЇ

## КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОРМАЦІЙНИХ СИСТЕМ ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Назва кваліфікаційної роботи Вбудована система моніторингу стану серверного обладнання на базі Raspberry Pi та протоколу SNMP
Автор Кірил МЕЛЬНИК
Освітня програма Комп'ютерна інженерія та програмування
Рівень вищої освіти перший (бакалаврський)
Спеціальність 123 Комп'ютерна інженерія
Науковий керівник: к.т.н., доцент Дмитро МЕДЗАТИЙ

На основі аналізу кваліфікаційної роботи на дотримання вимог академічної доброчесності (у т.ч. відсутності ознак академічного плагіату) з урахуванням результатів перевірки роботи спеціалізованим програмним засобом(ами) комісія зробила такий висновок:

| № | Висновок | Позначка про відповідність |
|---|---|---|
| 1 | Ознаки академічного плагіату | |
| 1.1 | Запозичення, виявлені в роботі, є законними і не є академічним плагіатом (далі – зазначаються підстави віднесення запозичень до правомірних, якщо потрібно). Робота приймається до захисту. | відповідає |
| 1.2 | Виявлені запозичення не є академічним плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи (далі – зазначаються детальні та аргументовані підстави віднесення запозичень до правомірних). Робота приймається до захисту, але має бути відкоригована. | |
| 1.3 | Виявлені запозичення не є академічним плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. Робота може бути допущена до захисту після того як буде відкоригована та доопрацьована і успішно пройде повторну перевірку на академічний плагіат. | |
| 1.4 | Робота містить навмисні текстові спотворення, передбачувані спроби укриття текстових запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту. | |
| 2 | Інші види порушень академічної доброчесності | |

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) усі запозичення фрагментарні, або мають належним чином оформленні посилання;

2) окремі виявлені збіги є загальновживаними фразами або виразами, про що свідчить посилання системи на збіг з джерелами на один фрагмент речення;

3) всі зафіксовані системою ознаки модифікації тексту відносяться до комбінування латинських символів зі україномовними скороченнями індексів в формулах, що не є модифікацією тексту.

4) значна частина знайденого плагіату відноситься до списку використаних джерел

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості StrikePlagiarism, складає 1,56%; та системою Anti-Plagiarism складає 1%, що, з урахуванням наведених обґрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

01.06.2026

Завідувач кафедри — Підпис — Ольга ПАВЛОВА (Ім'я, ПРІЗВИЩЕ)

Гарант освітньої програми — Підпис — Андрій НІЧЕПОРУК (Ім'я, ПРІЗВИЩЕ)

Керівник кваліфікаційної роботи — Підпис — Дмитро МЕДЗАТИЙ (Ім'я, ПРІЗВИЩЕ)