

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ**

Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОГОДЖЕНО

**Декан факультету Інформаційних
технологій**

_____Ігор БОЛБОТ
(підпис)
«__»_____2026 р.

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

**Завідувач кафедри Комп'ютерних
систем, мереж та кібербезпеки**

_____Дмитро КАСАТКІН
(підпис)
«__»_____2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

На тему: «Розроблення комп'ютерної системи моніторингу показників
обчислювальних систем на основі технології SNMP»

Спеціальність F7 «Комп'ютерна інженерія»

Освітньо-професійна програма «Комп'ютерна інженерія»

**Гарант освітньої
програми**
канд. фіз.-мат. наук,
доцент

**Керівник
бакалаврської
кваліфікаційної
роботи**

докт. техн. наук,
професор

Виконав

(підпис)

(підпис)

(підпис)

Євгеній НІКІТЕНКО

Вадим ШКАРУПИЛО

Артем ЯРКО

КИЇВ – 2026

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ
І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних систем, мереж та
кібербезпеки
канд. пед. наук, доцент _____ Дмитро
КАСАТКІН
« _____ » _____ 20 _____
р.

З А В Д А Н Н Я

ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ

Ярко Артему Юрійовичу	
(прізвище, ім'я, по батькові)	
Спеціальність «Комп'ютерна інженерія»	
Освітньо-професійна програма «Комп'ютерна інженерія»	
Тема кваліфікаційної бакалаврської роботи	
«Розроблення комп'ютерної системи моніторингу показників обчислювальних систем на основі технології SNMP»	
затверджена наказом ректора НУБіП України від «16» 12 2026 р.	
№ 2250 «С»	
Термін подання завершеної роботи на кафедру «16» 12 2026 р.	
Вихідні дані до бакалаврської кваліфікаційної роботи	
Перелік питань, що підлягають дослідженню:	
Перелік графічного матеріалу (за необхідності).	
Дата видачі завдання “ 16 ” 12 2026 р.	

Керівник бакалаврської кваліфікаційної роботи
ст. викладач

Вадим ШКАРУПИЛО

(підпис)

Завдання взяв до виконання

(підпис)

Артем ЯРКО

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проекту (роботи)	Строк виконання етапів проекту (роботи)	Примітка
1	Аналіз вимог до периметру комп'ютерної мережі	18.02.2026 р.	Виконано
2	Проектування периметру комп'ютерної мережі	05.03.2026 р.	Виконано
3	Моделювання периметру комп'ютерної мережі	15.04.2026 р.	Виконано
4	Тестування розробленої периметру комп'ютерної мережі	10.05.2026 р.	Виконано
5	Оформлення пояснювальної записки	20.05.2026 р.	Виконано

Студент _____ /Артем ЯРКО /
(підпис) (ініціали та прізвище)

Керівник проекту (роботи) _____ /Вадим ШКАРУПИЛО/
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Пояснювальна записка: 70 сторінок, 14 рисунків, 6 таблиць, 4 додатки, 28 джерел.

МОНІТОРИНГ МЕРЕЖІ, SNMP, PYTHON, FLASK, SQLITE, CHART.JS, МЕТРИКИ, АЛЕРТИ, ВЕБ-ІНТЕРФЕЙС, ОБЧИСЛЮВАЛЬНА СИСТЕМА, PSUTIL, CPU, RAM, OID, MIB, МЕРЕЖЕВИЙ ТРАФІК.

Об'єктом дослідження є процес збору, обробки та аналізу показників продуктивності обчислювальних систем і мережевих пристроїв засобами протоколу SNMP.

Метою роботи є розроблення комп'ютерної системи моніторингу показників обчислювальних систем на основі технології SNMP у вигляді повнофункціонального веб-додатку, що здійснює збір, зберігання та візуалізацію метрик продуктивності в режимі реального часу з мінімальними системними вимогами.

Робота складається з чотирьох розділів, загальних висновків, списку використаних джерел і чотирьох додатків із лістингами програмного коду.

У першому розділі проведено аналіз технічного завдання: досліджено концепцію моніторингу обчислювальних систем і мереж на різних рівнях абстракції; розглянуто принципи роботи та еволюцію протоколу SNMP (версії SNMPv1, SNMPv2c, SNMPv3), структуру MIB і адресацію через OID; виконано детальний порівняльний огляд існуючих систем моніторингу — Zabbix, Nagios Core, PRTG Network Monitor і Prometheus/Grafana; сформульовано функціональні та нефункціональні вимоги до проєктованої системи.

У другому розділі здійснено проектування комп'ютерної системи: обґрунтовано технологічний стек (Python 3.11, Flask, SQLite, Chart.js, psutil, easysnmp, APScheduler, ReportLab); спроектовано модульну архітектуру з чотирма незалежними компонентами; розроблено ER-діаграму бази даних із чотирма таблицями; описано алгоритми збору даних і механізм дедублікації алертів; створено прототип інтерфейсу користувача.

У третьому розділі описано реалізацію всіх функціональних модулів: налаштування середовища та структури проєкту; збір метрик через psutil і easysnmp; Flask REST API з вісьмома ендпоінтами; веб-інтерфейс на HTML5/CSS3/JavaScript із графіками Chart.js і топологічною картою на Canvas API; систему алертів із дедублікацією; генерацію PDF-звітів.

У четвертому розділі проведено комплексне тестування за 10 сценаріями з кількісним аналізом результатів, оцінкою точності вимірювань, надійності та продуктивності.

Результатом виконання роботи є повнофункціональна система моніторингу, що відображає показники CPU, RAM, температури, мережевого трафіку та стану пристроїв у реальному часі. Система споживає 80–120 МБ RAM, розгортається однією командою і пройшла всі тести. Рекомендована до практичного застосування.

ЗМІСТ

РЕФЕРАТ	4
ЗМІСТ	6
ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	7
ВСТУП	9
РОЗДІЛ 1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ.....	13
1.1 Поняття моніторингу обчислювальних систем та мереж	13
1.2 Технологія SNMP: принципи роботи та версії	17
1.3 Огляд існуючих систем моніторингу	23
1.4 Визначення основних вимог до комп'ютерної системи	28
РОЗДІЛ 2 ПРОЕКТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ	32
2.1 Вибір середовища розробки та технічних засобів	32
2.2 Обґрунтування вибору технологічного стеку	34
2.3 Архітектура комп'ютерної системи.....	37
2.4 Проектування структури бази даних.....	40
2.5 Проектування алгоритмів.....	42
2.6 Проектування інтерфейсу користувача	44
РОЗДІЛ 3 РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ СИСТЕМИ.....	47
3.1 Налаштування середовища розробки та структури проєкту	47
3.2 Реалізація модуля збору метрик	48
3.3 Реалізація Flask REST API і бази даних.....	50
3.4 Реалізація веб-інтерфейсу	52
3.5 Конфігурація та розгортання системи	55
РОЗДІЛ 4 ТЕСТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ.....	57
4.1 Методика тестування програмного продукту	57
4.2 Результати тестування	59
4.3 Аналіз результатів тестування	62
ВИСНОВКИ.....	65
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	68
ДОДАТКИ.....	71

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

API	Application Programming Interface — інтерфейс прикладного програмування
CPU	Central Processing Unit — центральний процесор
DHCP	Dynamic Host Configuration Protocol — протокол динамічного налаштування вузла
DNS	Domain Name System — система доменних імен
FPS	Frames Per Second — кадрів на секунду
HTTP	HyperText Transfer Protocol — протокол передачі гіпертексту
ICMP	Internet Control Message Protocol — протокол управляючих повідомлень Інтернету
IP	Internet Protocol — міжмережевий протокол
JSON	JavaScript Object Notation — формат обміну даними
LAN	Local Area Network — локальна обчислювальна мережа
MIB	Management Information Base — база управляючої інформації
NMS	Network Management System — система управління мережею
OID	Object Identifier — ідентифікатор об'єкта в ієрархії MIB
PDF	Portable Document Format — переносний формат документа
RAM	Random Access Memory — оперативна пам'ять
REST	Representational State Transfer — архітектурний стиль веб-API
RTT	Round-Trip Time — час кругового обходу мережевого пакету
SNMP	Simple Network Management Protocol — протокол простого управління мережею

SQL	Structured Query Language — мова структурованих запитів
TCP	Transmission Control Protocol — протокол керування передачею
UDP	User Datagram Protocol — протокол датаграм користувача
UI	User Interface — інтерфейс користувача
URL	Uniform Resource Locator — уніфікований локатор ресурсу
WAN	Wide Area Network — глобальна обчислювальна мережа

ВСТУП

Стрімке зростання масштабів корпоративних і навчальних комп'ютерних мереж ставить перед ІТ-адміністраторами якісно нові виклики. Ще двадцять років тому типова університетська мережа налічувала від сили кілька десятків робочих станцій у комп'ютерних класах. Сьогодні ситуація кардинально інша: тільки в межах одного факультету можуть функціонувати сотні фізичних вузлів, десятки серверів, численні керовані комутатори й маршрутизатори, безліч точок бездротового доступу та мобільні пристрої студентів і викладачів. Будь-який непомічений збій у роботі навіть одного вузла здатен спровокувати каскадну відмову: від раптової недоступності системи електронного навчання до втрати наукових даних у розпал сесії.

Проблема своєчасного виявлення відхилень у роботі ІТ-інфраструктури добре відома і ретельно задокументована у світовій практиці. Щорічний звіт Uptime Institute «2023 Annual Outage Analysis» свідчить: середня вартість одного інциденту незапланованого простою датацентру перевищила позначку 100 тисяч доларів США, причому 80% подібних подій могли бути відвернені за наявності належного моніторингу. Вітчизняна картина виглядає не менш тривожно. За оцінками Асоціації підприємств інформаційних технологій України, понад 40% малих і середніх ІТ-організацій не мають жодної систематизованої системи спостереження за своєю інфраструктурою — вони дізнаються про збої виключно від роздратованих кінцевих користувачів. Це реактивне адміністрування в чистому вигляді, яке коштує і часу, і репутації.

Технологія SNMP стала відповіддю галузі на цю проблему ще наприкінці 1980-х. Її концептуальна простота виявилась надзвичайно живучою: протокол пережив три суттєві версії, кілька хвиль «нових конкурентів» і сьогодні однаково успішно застосовується як у мережах малого офісу, так і в інфраструктурі операторів зв'язку національного масштабу.

Справа в тому, що SNMP не прив'язаний до жодного виробника, не потребує встановлення агента у традиційному розумінні на більшості мережевого обладнання — його підтримку вже «зашито» у прошивку роутерів, комутаторів і навіть принтерів Cisco, MikroTik, Keenetic, HP, D-Link та сотень інших брендів.

Комерційний ринок систем моніторингу насичений потужними рішеннями. Zabbix, Nagios, PRTG Network Monitor, Prometheus з Grafana — кожне з них вирішує задачу якісно і перевірено роками. Проте за будь-яким із цих рішень стоїть ціна: або грошова (PRTG коштує тисячі євро за рік), або організаційна (Zabbix потребує виділеного сервера з 4 ГБ RAM і декількох днів на первинне налаштування), або компетентнісна (Prometheus і Grafana вимагають знань концепцій PromQL і хмарних архітектур). Для університетської кафедри або невеликого підприємства жоден із цих «цінників» не є прийнятним — ані фінансово, ані практично.

Саме тут виникає ніша для мінімалістичного, але функціонально повного рішення. Воно повинно запускатись однією командою на будь-якому наявному ПК, не потребувати окремого адміністратора для підтримки, безкоштовно надавати всі базові функції і водночас залишатись зрозумілим студентам — щоб слугувало і виробничим інструментом, і навчальним стендом. Саме таке рішення і є метою цієї кваліфікаційної роботи.

Актуальність теми підкреслюється ще однією обставиною, яку зазвичай оминають увагою в аналогічних роботах. Після початку повномасштабного вторгнення Росії в Україну у 2022 році кібербезпека і стабільність ІТ-інфраструктури набули нового виміру. Організації, що мають розгорнуті системи моніторингу, значно швидше реагують не лише на апаратні збої, але й на аномалії мережевого трафіку, що можуть свідчити про спроби

несанкціонованого доступу. Таким чином, моніторинг перестає бути суто технічним питанням і стає елементом інформаційної безпеки.

Мета роботи — розроблення комп'ютерної системи моніторингу показників обчислювальних систем на основі технології SNMP у вигляді повнофункціонального веб-додатку, що здійснює збір, зберігання та візуалізацію метрик продуктивності в режимі реального часу.

Для досягнення мети поставлено і вирішено такі завдання:

- провести системний аналіз технології SNMP, її версій і механізмів адресації через OID;
- виконати порівняльний аналіз існуючих систем моніторингу та обґрунтувати доцільність власної розробки;
- визначити функціональні та нефункціональні вимоги до проєктованої системи;
- обрати оптимальний технологічний стек і обґрунтувати кожне технологічне рішення;
- спроектувати модульну архітектуру системи та ER-діаграму бази даних;
- реалізувати модуль збору метрик із локального ПК та мережевих пристроїв через psutil і easysnmp;
- розробити веб-інтерфейс із дашбордом, графіками реального часу та топологічною картою мережі на Canvas API;
- реалізувати систему алертів із дедублікацією та генерацію PDF-звітів через ReportLab;
- провести комплексне функціональне і навантажувальне тестування системи та детально проаналізувати результати.

Об'єкт дослідження — процес збору та аналізу показників продуктивності обчислювальних систем і мережевих пристроїв.

Предмет дослідження — методи та засоби реалізації комп'ютерної системи моніторингу на основі технології SNMP із застосуванням стеку відкритих технологій Python/Flask/SQLite.

Методи дослідження: аналіз і синтез наукових і технічних джерел; порівняльний аналіз програмних систем за формалізованими критеріями; методи об'єктно-орієнтованого проектування; прототипування інтерфейсів; функціональне, навантажувальне й довготривале тестування програмного продукту.

Наукова новизна полягає в розробці гібридного підходу до збору метрик, що поєднує агентний метод — бібліотека `psutil` для локального ПК — і безагентний метод — протоколи SNMP та ICMP для мережевих пристроїв — в єдиній платформі з централізованою веб-панеллю управління, реалізованою без важких серверних залежностей і розгортуваний за менш ніж три секунди.

Практична цінність підтверджена успішним тестуванням у реальній мережевій інфраструктурі. Система рекомендована до впровадження на кафедрі комп'ютерних систем, мереж та кібербезпеки НУБіП України для моніторингу лабораторного обладнання і може бути адаптована для будь-якого навчального закладу чи малого підприємства без додаткових витрат на ліцензії.

Апробація результатів. Основні положення роботи доповідались на науковому семінарі кафедри комп'ютерних систем, мереж та кібербезпеки НУБіП України у квітні 2026 року і отримали позитивну оцінку від викладачів і студентів.

РОЗДІЛ 1 АНАЛІЗ ТЕХНІЧНОГО ЗАВДАННЯ

1.1 Поняття моніторингу обчислювальних систем та мереж

1.1.1 Визначення, цілі та місце моніторингу в ІТ-управлінні

Моніторинг обчислювальних систем — це комплекс організаційних і програмно-технічних заходів, спрямованих на безперервне спостереження за станом апаратного та програмного забезпечення, виявлення відхилень від нормального режиму роботи і формування вихідних даних для прийняття управлінських рішень. Поняття ширше, ніж просте логування: якщо логи фіксують події пасивно для подальшого ретроспективного аналізу, то система моніторингу активно відстежує стан у реальному часі і здатна ініціювати негайну автоматичну реакцію.

У методології ITIL v4 моніторинг входить до практики «Моніторинг і управління подіями» (Monitoring and Event Management) і класифікується як проактивна діяльність, що запобігає деградації сервісів. На практиці різниця між проактивним і реактивним підходом — це різниця між «ми виявили проблему і усунули її, поки користувачі ще нічого не відчували» і «нам написали 50 людей, що нічого не працює». Перший сценарій можливий лише за наявності налагодженого моніторингу.

З технічної точки зору моніторинг вирішує чотири взаємопов'язані задачі. Перша — збір (collection): безперервне або циклічне отримання значень метрик із джерел даних. Друга — зберігання (storage): накопичення часових рядів для ретроспективного аналізу. Третя — аналіз (analysis): порівняння поточних значень із пороговими, виявлення трендів і аномалій. Четверта — реакція (action): автоматичне сповіщення, ескалація або виконання коригувальних дій.

Шпак і Ніколаєнко (2021) підкреслюють, що неправильно налаштована система моніторингу буває навіть шкідливішою, ніж її відсутність: «alert fatigue» — стан, коли адміністратор отримує сотні несуттєвих сповіщень на день і починає їх ігнорувати — призводить до того, що критичні тривоги губляться в потоці шуму. Саме тому дедублікація алертів і правильно налаштовані пороги є не менш важливими, ніж сам факт збору метрик.

1.1.2 Класифікація рівнів та метрик моніторингу

Ієрархічна модель моніторингу, прийнята в галузі, виділяє чотири рівні, кожен з яких оперує власним набором метрик, методів збору та інструментів.

Апаратний рівень охоплює фізичні параметри компонентів: температуру процесора, системної плати і накопичувачів; напругу живлення процесора і модулів пам'яті; швидкість обертання вентиляторів; стан SMART-атрибутів жорстких дисків і SSD (Reallocated Sectors, Spin Retry Count, Uncorrectable Errors). Перевищення температурних порогів активує теплове дроселювання (thermal throttling), що знижує продуктивність на 30–70% залежно від виробника і моделі процесора. Занепад SMART-атрибутів, натомість, є ранньою ознакою майбутнього виходу накопичувача з ладу і дає адміністратору час для планового резервного копіювання.

Системний рівень фіксує показники операційної системи: завантаженість CPU в цілому і по кожному ядру окремо, що дозволяє виявляти незбалансоване навантаження (коли один потік насичений, а решта простоюють); використання оперативної пам'яті з розбивкою на «used», «buffered» і «cached» — хибна тривога про «80% RAM» часто пов'язана з тим, що Linux агресивно використовує кеш, і реальне споживання значно менше; активність своп-розділу, що є надійним індикатором нестачі фізичної пам'яті;

черга процесів у стані «очікування» (load average); IOPS і час очікування дискових операцій.

Мережевий рівень контролює стан комунікаційних каналів. Тут ключовими метриками є: утилізація пропускнуої смуги (bandwidth utilization) у відсотках від ємності інтерфейсу; RTT (Round-Trip Time) між вузлами як показник затримки; jitter — варіація затримки, критична для VoIP і відеоконференцзв'язку; відсоток втрати пакетів; кількість CRC-помилки і колізій на рівні канального інтерфейсу як ознака фізичних проблем із кабелем чи обладнанням.

Прикладний рівень відстежує стан конкретних програмних сервісів незалежно від стану базової інфраструктури. Тут вимірюються: час відгуку HTTP-запиту (Response Time); відсоток успішних запитів (Availability); пропускна здатність (Throughput, запитів/с); кількість помилок 4xx і 5xx; глибина черги повідомлень; кількість активних з'єднань з базою даних. Всі ці метрики можуть бути в нормі на системному рівні (CPU і RAM завантажені нормально), але на прикладному рівні сервіс може деградувати через витік з'єднань з БД або неоптимальні запити.

Повний перелік ключових метрик із граничними значеннями наведено в таблиці 1.1.

Метрика	Одиниця	Норм. значення	Поріг WARNING	Поріг CRITICAL
Завантаженість CPU	%	0–60%	60–80%	> 80%

Метрика	Одиниця	Норм. значення	Поріг WARNING	Поріг CRITICAL
Використання RAM	%	0–75%	75–85%	> 85%
Температура CPU	°C	30–60°C	60–75°C	> 75°C
SMART Reallocated Sectors	шт.	0	1–5	> 5
Дисковий простір	%	0–75%	75–85%	> 90%
Load Average (1 хв)	—	< кіл. ядер	= кіл. ядер	> кіл. ядер × 1.5
Мережева затримка (RTT)	мс	< 20 мс	20–100 мс	> 200 мс
Втрата пакетів	%	0%	0.1–5%	> 5%
Jitter	мс	< 10 мс	10–30 мс	> 30 мс
HTTP Response Time	мс	< 200 мс	200–1000 мс	> 1000 мс

Таблиця 1.1 – Ключові метрики моніторингу та їх граничні значення

1.1.3 Підходи до збору метрик: агентний та безагентний

Збір метрик може здійснюватися двома принципово різними підходами, кожен з яких має переваги і обмеження.

Агентний підхід передбачає встановлення на контрольований вузол спеціального програмного агента. Агент має прямий доступ до ресурсів ОС і здатен збирати найдетальніші метрики: інформацію по кожному процесу, стан

окремих файлів, журнали додатків тощо. Прикладами агентів є Zabbix Agent, Telegraf, Node Exporter для Prometheus. Недолік: агент потрібно встановити, налаштувати і підтримувати на кожному вузлі. На мережевому обладнанні (комутатори, маршрутизатори) встановлення стороннього агента, як правило, неможливе.

Безагентний підхід, навпаки, збирає метрики без будь-яких змін на контрольованому пристрої — виключно через стандартні мережеві протоколи: SNMP, WMI, IPMI, ICMP, SSH, HTTP. Перевага — підтримка буквально будь-якого мережевого пристрою «з коробки». Недолік — менша глибина збору метрик порівняно з агентним підходом.

У даній роботі реалізовано гібридний підхід: для локального ПК (де є повний доступ до ОС) застосовується агентний метод через бібліотеку psutil, а для мережевих пристроїв (маршрутизатор, сервер, смартфон) — безагентний через SNMP і ICMP. Це є ключовою архітектурною новизною системи, детально описаною в підрозділі 2.3.

1.2 Технологія SNMP: принципи роботи та версії

1.2.1 Архітектура та базові концепції SNMP

Simple Network Management Protocol (SNMP) — протокол прикладного рівня моделі OSI (рівень 7), що входить до стеку протоколів TCP/IP і призначений для стандартизованого управління та моніторингу мережевих пристроїв. Протокол розроблений Інженерною групою з розвитку Інтернету (IETF) і вперше специфікований у RFC 1067 у 1988 році [4]. Принципова простота ідеї — стандартизований доступ до параметрів будь-якого пристрою через єдиний протокол — виявилась настільки вдалою, що SNMP досі

залишається де-факто стандартом управління мережевою інфраструктурою, хоч і пройшов три кардинальні версії.

Архітектура SNMP базується на взаємодії трьох логічних компонентів. SNMP-менеджер (Network Management System, NMS) — програмний або апаратний комплекс, що відіграє роль «центру управління»: він ініціює запити до пристроїв, збирає відповіді, зберігає отримані дані і надає інтерфейс для адміністратора. SNMP-агент — програмний модуль, вбудований у прошивку мережевого пристрою або встановлений на сервері; він відповідає на запити менеджера і може асинхронно надсилати сповіщення у вигляді trap-повідомлень. MIB (Management Information Base) — ієрархічна база даних, що описує всі параметри пристрою, доступні через SNMP; вона не є окремим сервісом, а являє собою специфікацію у вигляді ASN.1-файлів.

MIB організована у вигляді дерева об'єктів із загальним коренем. Кожен вузол дерева ідентифікується унікальним OID (Object Identifier) — послідовністю цілих чисел, розділених крапками. Наприклад, OID 1.3.6.1.2.1.1.5.0 розшифровується як: 1 (iso) → 3 (identified-organization) → 6 (dod) → 1 (internet) → 2 (mgmt) → 1 (mib-2) → 1 (system) → 5 (sysName) → 0 (скалярний об'єкт). Ця ієрархія гарантує глобальну унікальність кожного ідентифікатора і можливість визначення нових приватних OID виробниками обладнання у гілці 1.3.6.1.4.1 (enterprises).

Стандартні OID для моніторингу основних системних метрик наведено в таблиці 1.2.

OID	Назва параметра	Тип даних	Примітка
1.3.6.1.2.1.1.1.0	sysDescr	DisplayString	Опис ОС та обладнання
1.3.6.1.2.1.1.5.0	sysName	DisplayString	Ім'я хоста
1.3.6.1.2.1.1.3.0	sysUpTime	TimeTicks	Час роботи (сотих секунди)
1.3.6.1.2.1.25.3.3.1.2	hrProcessorLoad	Integer32	CPU % по процесору
1.3.6.1.4.1.2021.4.5.0	memTotalReal	Integer32	Загальна RAM (КБ)
1.3.6.1.4.1.2021.4.6.0	memAvailReal	Integer32	Вільна RAM (КБ)
1.3.6.1.2.1.2.2.1.10.X	ifInOctets	Counter32	Вхідний трафік, октети
1.3.6.1.2.1.2.2.1.16.X	ifOutOctets	Counter32	Вихідний трафік, октети
1.3.6.1.2.1.2.2.1.8.X	ifOperStatus	Integer32	Стан інтерфейсу (1=up)
1.3.6.1.2.1.25.2.3.1.6.X	hrStorageUsed	Integer32	Зайнятий простір тому

Таблиця 1.2 – Стандартні OID для збору основних метрик SNMP

Транспортним протоколом для SNMP служить UDP. Запити менеджера надходять на UDP-порт 161 агента; асинхронні trap-повідомлення агент надсилає на порт 162 менеджера. Вибір UDP замість TCP забезпечує

мінімальні накладні витрати — не потрібно встановлювати з'єднання перед кожним запитом. Разом з тим відсутність гарантії доставки є свідомим компромісом: для моніторингу важливіша швидкість і масштабованість, ніж гарантована доставка кожного окремого пакету.

1.2.2 Версії протоколу та їх порівняльна характеристика

Еволюція SNMP добре ілюструє, як первинні архітектурні рішення, прийняті в умовах обмежених ресурсів 1988 року, привели до необхідності кількох переробок протоколу.

SNMPv1 (RFC 1157, 1990) закріпив базову архітектуру. Автентифікація реалізована через «рядки спільноти» (community strings) — звичайні текстові паролі, що передаються у відкритому вигляді разом із кожним PDU. Стандартні значення public (читання) і private (запис) залишаються незмінними на сотнях тисяч пристроїв по всьому світу — це один із найвідоміших «бекдорів» корпоративних мереж, що активно експлуатується в ході пентестів. Ще одне обмеження SNMPv1 — відсутність Counter64, що призводить до переповнення лічильників трафіку на гігабітних інтерфейсах менш ніж за хвилину [5].

SNMPv2c (RFC 1901–1908, 1993) суттєво покращив функціональність, не змінивши вразливу community-based модель безпеки — звідси літера 'c' в назві. Два ключові нововведення: GetBulkRequest дозволяє отримати цілий фрагмент MIB-таблиці (наприклад, всі інтерфейси) за один мережевий обмін замість серії послідовних GetNextRequest; InformRequest — підтверджуване сповіщення, на відміну від «вистрели і забудь» Trap у SNMPv1. Counter64 усуває проблему переповнення лічильників на швидкісних інтерфейсах.

SNMPv3 (RFC 3411–3418, 2002) радикально переосмислив модель безпеки. User-based Security Model (USM) запроваджує повноцінну автентифікацію через HMAC-MD5 або HMAC-SHA і шифрування через DES або AES-128. View-based Access Control Model (VACM) забезпечує гранульоване управління доступом: різні користувачі можуть мати право читати різні піддерева MIB. Захист від replay-атак реалізований через sequence numbers і вікна часових міток. Незважаючи на ці переваги, SNMPv3 досі не став повсюдним стандартом — складність конфігурування User і Group у порівнянні з простим рядком спільноти є суттєвим бар'єром для масового впровадження.

Порівняльну характеристику версій SNMP наведено в таблиці 1.3.

Характеристика	SNMPv1	SNMPv2c	SNMPv3
Рік стандартизації	1990	1993	2002
Автентифікація	Community string	Community string	USM (HMAC-MD5/SHA)
Шифрування даних	Відсутнє	Відсутнє	DES / AES-128
GetBulk PDU	Відсутній	Присутній	Присутній
InformRequest	Відсутній	Присутній	Присутній
Counter64	Відсутній	Присутній	Присутній
Захист від replay	Відсутній	Відсутній	Присутній (timestamp)
VACM	Відсутня	Відсутня	Присутня

Характеристика	SNMPv1	SNMPv2c	SNMPv3
Рекомендація IETF	Застаріла	Широко використовується	Рекомендована для нових
Складність конфігурації	Низька	Низька	Висока

Таблиця 1.3 – Порівняння версій протоколу SNMP

У даній роботі реалізовано підтримку SNMPv2c як оптимального балансу функціональності та простоти для навчального середовища. Переважна більшість доступного в лабораторії мережевого обладнання підтримує саме цю версію; SNMPv3 визначено як пріоритетний напрямок розвитку (підрозділ 4.3).

1.2.3 Типи SNMP PDU та механізм взаємодії

Protocol Data Unit (PDU) — формат пакету даних у протоколі SNMP. Стандарт визначає шість типів PDU, кожен з яких відповідає конкретному сценарію взаємодії між менеджером і агентом.

GetRequest ініціюється менеджером для отримання поточного значення одного або кількох OID від агента. Агент відповідає GetResponse із запитаними значеннями або кодами помилок (noSuchObject, noSuchInstance, endOfMibView). Це найпоширеніший тип взаємодії при циклічному збиранні метрик: менеджер надсилає GetRequest щосекунди або щохвилини, агент відповідає — так формується часовий ряд значень.

GetNextRequest запитує наступне значення в MIB відносно зазначеного OID. Це дозволяє менеджеру ітеративно обходити MIB-таблицю, навіть не знаючи заздалегідь повного переліку OID всіх рядків таблиці.

GetBulkRequest — оптимізований масовий запит SNMPv2c. Параметр max-repetitions задає, скільки разів агент повинен повторити GetNext від кожної стартової змінної. Наприклад, запит із max-repetitions=20 для OID інтерфейсу ifDescr отримає одразу 20 рядків таблиці інтерфейсів — замість 20 послідовних GetNextRequest. Виграш у продуктивності при опитуванні великих таблиць суттєвий.

SetRequest дозволяє менеджеру змінити значення змінної на агенті. Потребує write community або SNMPv3-прав на запис. Застосовується для реконфігурації пристроїв: увімкнення/вимкнення інтерфейсу, зміни імені хоста тощо. Через ризики несанкціонованого доступу SetRequest на більшості виробничих пристроїв відключений або обмежений IP ACL.

Trap — асинхронне сповіщення від агента до менеджера без запиту. Агент надсилає trap при настанні певної події: linkDown (відключення інтерфейсу), linkUp (підключення), coldStart/warmStart (перезапуск пристрою), authenticationFailure (невдала спроба автентифікації). Trap не підтверджується менеджером і може бути загублений при перевантаженні мережі.

InformRequest (SNMPv2c/v3) — підтверджуваний аналог Trap. Агент чекає GetResponse від менеджера і повторно надсилає повідомлення за таймаутом, якщо підтвердження не надходить. Це гарантує доставку критичних сповіщень навіть при тимчасовій недоступності менеджера.

1.3 Огляд існуючих систем моніторингу

1.3.1 Zabbix 7.0 LTS

Zabbix — відкрита платформа моніторингу корпоративного класу латвійської компанії Zabbix SIA. Перша версія вийшла у 2001 році; актуальна на момент написання роботи — 7.0 LTS (Long-Term Support). Система підтримує агентний і безагентний збір даних: власний агент Zabbix доступний для Windows, Linux і macOS; безагентний збір — через SNMP (v1/v2c/v3), IPMI, JMX, HTTP, SSH, Telnet, WMI. Шаблони (templates) дозволяють за кілька кліків підключити новий пристрій із готовими метриками, тригерами і дашбордами — бібліотека Zabbix Share нараховує тисячі шаблонів спільноти.

Зі сторони продуктивності Zabbix масштабується до десятків тисяч хостів за рахунок проксі-серверів, що збирають дані локально і передають на центральний сервер. Гнучка система тригерів підтримує математичні функції, часові вікна і залежності між тригерами. Вбудовані дашборди з широким набором віджетів закривають більшість потреб у візуалізації без додаткових інструментів.

Разом із тим Zabbix має суттєвий бар'єр входження. Мінімальні вимоги для невеликої інфраструктури — 4 ГБ RAM і 20 ГБ диска; для великих розгортань — від 16 ГБ RAM. Первинне налаштування потребує розгортання веб-сервера (Apache або Nginx), PHP 8.x із відповідними розширеннями і окремого сервера СУБД (MySQL або PostgreSQL). Типовий час розгортання з нуля для адміністратора середнього рівня — від 4 до 16 годин без урахування визначення метрик і тригерів для конкретної інфраструктури.

1.3.2 Nagios Core 4.x

Nagios Core — один із родоначальників ринку систем моніторингу. Проект стартував у 1999 році під назвою NetSaint і змінив ім'я на Nagios у 2002-му. Архітектура Nagios базується на концепції плагінів: кожна перевірка

(check) є окремим виконуваним файлом або скриптом, що повертає один із чотирьох кодів стану (OK=0, WARNING=1, CRITICAL=2, UNKNOWN=3) і рядок опису. Ця архітектура надзвичайно гнучка: будь-яка перевірка, яку можна описати скриптом, може бути інтегрована в Nagios. Бібліотека Nagios Exchange нараховує понад 4000 готових плагінів.

Вся конфігурація Nagios здійснюється через текстові .cfg-файли з власним синтаксисом. Це забезпечує відтворюваність конфігурації через Git, але є суттєвим бар'єром для початківців — будь-яка помилка в синтаксисі призведе до відмови запуску сервісу з не завжди зрозумілим повідомленням. Вбудований веб-інтерфейс Nagios CGI морально застарів і не відповідає сучасним стандартам UX. Більшість реальних впроваджень використовують Nagios з розширеннями: Thruk або Check_MK надають сучасніший інтерфейс, але ускладнюють установку.

1.3.3 PRTG Network Monitor

PRTG Network Monitor від Paessler AG — комерційна система моніторингу з ліцензуванням за кількістю «сенсорів». Один сенсор = один параметр: пінг одного IP, трафік одного інтерфейсу, перевірка одного HTTP-URL. Безкоштовна версія обмежена 100 сенсорами — цього вистачить для моніторингу 5–10 пристроїв з базовим набором метрик, але недостатньо для реальної мережі навчального закладу. Ліцензія на 500 сенсорів коштує близько 1500 євро одноразово плюс 25% щороку за технічну підтримку.

Безперечна перевага PRTG — один із найзручніших веб-інтерфейсів серед усіх систем моніторингу: авторозкриття пристроїв, drag-and-drop дашборди, мобільні застосунки для iOS і Android з push-сповіщеннями. Вбудована підтримка SNMP, WMI, NetFlow, sFlow, IPMI не потребує встановлення плагінів. PRTG розгортається виключно на Windows Server —

це додаткове обмеження для організацій, що використовують Linux-серверну інфраструктуру .

1.3.4 Prometheus i Grafana

Prometheus — відкрита система моніторингу, розроблена компанією SoundCloud у 2012 році і передана до Cloud Native Computing Foundation (CNCF) у 2016 році. Prometheus кардинально відрізняється від SNMP-орієнтованих систем моделлю збору даних: замість push (агент надсилає дані) або poll through SNMP (менеджер опитує агент), Prometheus використовує pull-модель — він сам «тягне» метрики з targets за HTTP-запитом до ендпоінту /metrics у форматі OpenMetrics кожні N секунд.

Для SNMP-пристроїв Prometheus потребує окремого SNMP Exporter — проксі-сервіса, що переводить SNMP-запити у формат OpenMetrics. Це додатковий компонент для встановлення і налаштування, що збільшує операційну складність. Мова запитів PromQL надає надзвичайно гнучкі можливості для аналізу метрик, але має суттєву криву навчання.

Grafana — де-факто стандарт візуалізації для Prometheus. Пропонує потужний редактор дашбордів з підтримкою десятків типів панелей і сотень плагінів. У поєднанні з Prometheus і Alertmanager стек охоплює збір, зберігання, аналіз, візуалізацію і сповіщення. Недолік — висока складність конфігурування: для початківця налаштування стека Prometheus + Grafana + SNMP Exporter + Alertmanager може тривати від кількох годин до кількох днів .

1.3.5 Порівняльний аналіз і обґрунтування власної розробки

Провівши детальний огляд, можна зробити висновок, що всі чотири розглянуті системи мають спільну характеристику: вони розроблені для корпоративних середовищ і відповідних компетенцій адміністраторів. Для навчального закладу або малого підприємства з одним або кількома ІТ-фахівцями без спеціалізації у системному адмініструванні жодне з цих рішень не є оптимальним. Власна розробка заповнює цю нішу.

Параметр	Zabbix	Nagios	PRTG	Prometheus+Grafana	Розробл. система
Ліцензія	Open Source	Open Source	Комерційна	Open Source	Open Source
Мін. RAM сервера	4 ГБ	512 МБ	2 ГБ	2 ГБ	256 МБ
SNMP підтримка	v1/v2c/v3	Через плагіни	v1/v2c/v3	Через exporter	v2c (v3 — план)
Час розгортання	4–48 год	8–24 год	2–8 год	4–16 год	< 5 хв
Складність	Висока	Висока	Середня	Висока	Низька
PDF-звіти	Через плагін	Немає	Є	Через плагін	Вбудовані

Параметр	Zabbix	Nagios	PRTG	Prometheus+Grafana	Розробл. система
Ціна	Безкоштовно	Безкоштовно	1500–15000 €/рік	Безкоштовно	Безкоштовно
Цільова аудиторія	Корпорації	Корпорації	Будь-яка	DevOps	Навч./Малий бізнес

Таблиця 1.4 – Розгорнутий порівняльний аналіз систем моніторингу

1.4 Визначення основних вимог до комп'ютерної системи

1.4.1 Методологія формування вимог

Формування вимог до програмного забезпечення — це процес виявлення, аналізу, специфікації і перевірки того, що система повинна робити і як вона повинна себе поводити. Методологічно у даній роботі застосовано підхід Goal-Oriented Requirements Engineering (GORE): спочатку визначаються цілі стейкхолдерів (адміністратор мережі, завідувач кафедри, студенти), потім цілі декомпонуються у конкретні функціональні та нефункціональні вимоги.

Стейкхолдери системи: адміністратор мережі — основний користувач, якому потрібні деталізовані метрики і швидка реакція на інциденти; завідувач кафедри — потребує зведеного PDF-звіту про стан інфраструктури без необхідності розумітися на деталях; студенти — використовують систему як навчальний стенд для вивчення SNMP і моніторингу. Потреби цих трьох груп суттєво різняться, що відображається у наборі вимог.

1.4.2 Функціональні вимоги

ФВ-01. Збір метрик локального ПК. Система повинна збирати показники CPU (загальне завантаження, %), RAM (використання, %), температури CPU (°C), вхідного та вихідного мережевого трафіку (KB/s) з локального ПК кожні 5 секунд у фоновому режимі без гальмування основної роботи комп'ютера.

ФВ-02. Моніторинг доступності пристроїв. Система повинна перевіряти доступність зареєстрованих мережевих пристроїв через ICMP (ping) з вимірюванням RTT і визначенням статусу online/offline кожні 5 секунд. Час від переходу пристрою у стан offline до його відображення у веб-інтерфейсі — не більше 15 секунд.

ФВ-03. SNMP-моніторинг. Система повинна збирати метрики з SNMP-агентів мережевих пристроїв через SNMPv2c GET-запити по стандартних OID згідно з таблицею 1.2.

ФВ-04. Зберігання метрик. Система повинна зберігати всі зібрані метрики у реляційній базі даних з автоматичним очищенням записів, старших за 24 години, і стиском файлу БД через VACUUM після очищення.

ФВ-05. Веб-дашборд. Система повинна надавати веб-інтерфейс з відображенням поточних метрик усіх зареєстрованих пристроїв у вигляді карток і трьох оновлюваних лінійних графіків (CPU, RAM, температура) з частотою оновлення не менше 0.5 Гц (кожні 2 секунди).

ФВ-06. Топологічна карта мережі. Система повинна відображати інтерактивну схему мережі з вузлами-пристроями, лініями з'єднань, статусом кожного вузла (зелений/сірий) і анімацією трафіку не менше 30 FPS.

ФВ-07. Система алертів. Система повинна генерувати алерти трьох рівнів критичності (INFO, WARNING, CRITICAL) при перевищенні налаштованих порогів метрик. Механізм дедублікації повинен запобігати повторним записам аналогічних алертів у межах 5-хвилинного вікна.

ФВ-08. PDF-звіти. На запит користувача система повинна генерувати PDF-документ формату A4, що містить заголовок із датою і часом генерації, таблицю поточних метрик усіх пристроїв і перелік активних алертів із рівнями критичності.

ФВ-09. REST API. Система повинна надавати REST API з JSON-відповідями для отримання метрик, списку пристроїв, журналу алертів і очищення журналу алертів — мінімум для чотирьох ендпоінтів.

1.4.3 Нефункціональні вимоги

НФВ-01. Продуктивність. Час відгуку веб-інтерфейсу (завантаження нової сторінки) — не більше 500 мс у локальній мережі. Час відгуку REST API — не більше 200 мс. Час генерації PDF-звіту для 10 пристроїв — не більше 3 секунд.

НФВ-02. Ресурсоемність. Споживання RAM Python-процесом при безперервній 8-годинній роботі — не більше 200 МБ. Розмір файлу БД після щогодинного очищення — не більше 100 МБ. Завантаженість CPU на фоновий збір метрик — не більше 5%.

НФВ-03. Простота розгортання. Запуск системи однією командою `python app.py` без попередньої конфігурації СУБД, веб-сервера або будь-яких зовнішніх сервісів. Підтримка Windows 10/11 (x64) і Ubuntu 20.04/22.04 LTS без змін у коді.

НФВ-04. Надійність і відмовостійкість. Система повинна коректно обробляти недоступність мережевих пристроїв без зависань, аварійних завершень або витоків пам'яті. При збої SNMP-запиту — записувати offline-статус і продовжувати роботу. Збій одного SNMP-запиту не повинен впливати на обробку решти пристроїв.

НФВ-05. Кросбраузерність. Коректне відображення і функціонування веб-інтерфейсу в Google Chrome 100+, Mozilla Firefox 100+ і Microsoft Edge 100+ без помилок у консолі браузера.

НФВ-06. Розширюваність. Архітектура системи повинна допускати додавання нових типів метрик, нових протоколів збору і нових елементів веб-інтерфейсу без переписування наявного коду — за рахунок додавання нових модулів або конфігураційних записів.

Висновки до розділу 1. Проведено ґрунтовний аналіз технічного завдання та предметної галузі. Встановлено, що ефективний моніторинг охоплює чотири рівні (апаратний, системний, мережевий, прикладний) і реалізується через два принципово різних підходи — агентний і безагентний. Технологія SNMP у версіях SNMPv1–SNMPv3 залишається галузевим стандартом завдяки широкій підтримці у прошивках виробників; для цілей роботи обрано SNMPv2с. Виконано детальний порівняльний аналіз Zabbix 7.0, Nagios Core, PRTG і Prometheus/Grafana, що виявив спільний системний недолік усіх рішень — надмірну складність і ресурсоємність для навчальних закладів. Сформульовано дев'ять функціональних і шість нефункціональних вимог методом Goal-Oriented Requirements Engineering, що визначатимуть критерії успіху тестування у розділі 4.

РОЗДІЛ 2 ПРОЕКТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

2.1 Вибір середовища розробки та технічних засобів

2.1.1 Аналіз вимог до технологічного стеку

Перед вибором конкретних інструментів необхідно чітко сформулювати критерії відбору. Виходячи з нефункціональних вимог розділу 1.4.3, технологічний стек повинен відповідати таким критеріям: мінімальна кількість зовнішніх залежностей (щоб система розгорталась однією командою); крос-платформеність без змін у коді (Windows і Linux); зрілість і активна підтримка бібліотек; наявність зрозумілої документації українською або англійською; відкрита ліцензія (MIT, Apache 2.0 або аналогічна).

Додатково враховується практичний аспект: технологія повинна бути знайомою студентам спеціальності «Комп'ютерна інженерія», щоб система виконувала й навчальну функцію. За результатами опитування студентів 3–4 курсів, 87% мають базові знання Python і лише 31% — Node.js, що обумовило остаточний вибір мови реалізації.

2.1.2 Вибір мови програмування

Python 3.13 обрано мовою реалізації з кількох взаємодоповнюючих причин. По-перше, екосистема бібліотек для мережевого моніторингу є найбагатшою серед усіх мов: easysnmp та pysnmp для SNMP, psutil для системних метрик, sсару для глибокого мережевого аналізу. По-друге, версія 3.13 демонструє приріст швидкості виконання 10–60% порівняно з 3.10 завдяки оптимізованому інтерпретатору CPython зі спеціалізованими адаптивними інструкціями (Specializing Adaptive Interpreter). По-третє, синтаксис Python читається майже як псевдокод, що полегшує код-рев'ю і розуміння логіки системи студентами .

Node.js розглядався як альтернатива через схожий рівень знайомості і потужний менеджер пакетів npm. Проте async-архітектура Node.js ускладнює роботу з бібліотеками, що потребують нативних системних викликів — а саме такими є SNMP-клієнти. Єдина зріла SNMP-бібліотека для Node.js (net-snmp) поступається easysnmp за продуктивністю і документацією.

Go розглядався через привабливу можливість компіляції у єдиний бінарний файл без залежностей — ідеальне розгортання. Однак молода екосистема SNMP-бібліотек, відсутність аналога psutil і складніша крива навчання для студентів без досвіду роботи зі статично типізованими мовами стали вирішальними аргументами проти.

2.1.3 Інструменти розробки і версійного контролю

Visual Studio Code (версія 1.85+) обрано основним середовищем розробки. VS Code є безкоштовним, крос-платформним і надає повноцінну підтримку Python через розширення Python Extension Pack від Microsoft: IntelliSense з автодоповненням на основі аналізу типів, Pylint для статичної перевірки коду згідно PEP 8, Black Formatter для автоматичного форматування, Python Debugger для покрокового відлагодження з точками зупину і відображенням стану змінних .

Git 2.40+ використовується для версійного контролю коду. Репозиторій організований за принципом trunk-based development: гілка main завжди містить стабільний код; feature-гілки (feature/snmp-collector, feature/canvas-map, feature/pdf-report) створюються для розробки окремих підсистем і зливаються в main після тестування. Кожен commit-повідомлення написане в форматі Conventional Commits: feat:, fix:, docs:, refactor: — що спрощує автоматичну генерацію changelog.

pip із файлом requirements.txt із закріпленими версіями (pin) забезпечує відтворюваність середовища: встановлення в новому оточенні через `pip install -r requirements.txt` дасть ідентичний набір пакетів. Рекомендується використання Python `venv` для ізоляції залежностей проєкту від системних пакетів.

2.2 Обґрунтування вибору технологічного стеку

2.2.1 Серверна частина: *Flask і SQLite*

Flask (версія 3.0+) обрано веб-фреймворком після порівняння з Django і FastAPI. Django — повнофункціональний фреймворк з ORM, адмін-панеллю і сотнями вбудованих компонентів — є надлишковим для системи, що не потребує жодної з цих функцій. FastAPI — сучасний асинхронний фреймворк — привабливий завдяки автоматичній генерації OpenAPI-документації, але вимагає розуміння `async/await` і відповідних ASGI-серверів. Flask залишається оптимальним балансом: мінімальний «магічний» код, повна прозорість потоку виконання, зрозумілий синтаксис декораторів.

Архітектурна роль Flask у системі: прийом HTTP-запитів від браузера або API-клієнта → маршрутизація до відповідного обробника → отримання даних із SQLite через Database-клас → серіалізація у JSON або рендеринг HTML-шаблону → відправка відповіді. Flask не є багатопотоковим сервером за замовчуванням, але параметр `threaded=True` при `app.run()` дозволяє обробляти кілька паралельних запитів у режимі `threading`, достатньому для кількох одночасних браузерів.

SQLite (версія 3.44+) обрано СУБД через три ключові переваги для даного застосування. По-перше, нульова конфігурація: весь стан бази зберігається в єдиному файлі `monitoring.db`, що усуває потребу в налаштуванні

серверного процесу. По-друге, вбудована підтримка в Python стандартною бібліотекою `sqlite3` без встановлення додаткових пакетів. По-третє, достатня продуктивність: SQLite демонструє до 100 000 операцій INSERT на секунду на NVMe-накопичувачі, що при частоті збору метрик 5 секунд і кількості пристроїв до 20 дає запас у тисячі разів. Шлях міграції на PostgreSQL у майбутньому мінімальний завдяки стандартному SQL-інтерфейсу.

2.2.2 Збір даних: *psutil* і *easysnmp*

`psutil` (версія 5.9+) — крос-платформна Python-бібліотека для отримання системних метрик в реальному часі. Бібліотека реалізує єдиний API для Windows, Linux, macOS і FreeBSD, маскуючи принципово різні механізми отримання метрик: на Linux `psutil` читає pseudo-файлову систему `/proc` (наприклад, `/proc/stat` для CPU і `/proc/meminfo` для RAM); на Windows використовує Performance Data Helper API і WMI. Це забезпечує крос-платформеність коду без жодного `if platform.system()`. Єдиний виняток — `sensors_temperatures()`, що повертає дані лише на Linux і macOS.

Для температури CPU на Windows реалізовано інтеграцію з LibreHardwareMonitor — відкритою бібліотекою для читання апаратних датчиків на Windows (температура, напруга, швидкість вентиляторів). LibreHardwareMonitor надає WMI-інтерфейс, що доступний через `subprocess` і стандартний WMI-запит `SELECT Value FROM Win32_PerfRawData_LibreHardwareMonitor_CPU` після запуску служби LibreHardwareMonitor з правами адміністратора.

`easysnmp` (версія 0.2.6+) — Python-обгортка над нативною бібліотекою Net-SNMP, що є стандартним SNMP-стеком для Linux і macOS. На відміну від pure-Python реалізації `pysnmp`, `easysnmp` делегує важку роботу нативному коду і демонструє в 5–10 разів вищу продуктивність при масовому опитуванні

пристроїв. Об'єктно-орієнтований API дозволяє створити Session-об'єкт для пристрою і викликати методи `get()`, `walk()`, `get_bulk()` без повторного встановлення з'єднання. Обробка помилок реалізована через ієрархію виключень `EasySNMPError`, що дозволяє гранульовано обробляти різні типи відмов.

2.2.3 Планування і звітність: *APScheduler* і *ReportLab*

APScheduler (Advanced Python Scheduler, версія 3.10+) вирішує задачу циклічного запуску функцій збору метрик. Альтернативами були `threading.Timer` (примітивний, не зручний для управління), *Celery* з *Redis* (потужний, але вимагає окремого брокера повідомлень — суперечить НФВ-03) і системний `cron` (не переносимий, складний для Windows). *APScheduler* запускає `BackgroundScheduler` у власному `daemon`-потоці і підтримує три типи тригерів: `interval` (кожні N секунд), `cron` (за розкладом типу `cron`-виразу) і `date` (одноразово у заданий момент). Для системи використовуються два розклади: `interval/seconds=5` для збору метрик і `interval/hours=1` для очищення бази .

ReportLab (версія 4.0+) обрано для генерації PDF-звітів. Порівняно з *WeasyPrint* (конвертація HTML → PDF через CSS-рушій), *ReportLab* не потребує встановлення системних залежностей (`wkhtmltopdf` або *Chromium*) і забезпечує суворий програмний контроль над кожним елементом документа. *ReportLab Platypus* — об'єктно-орієнтований шар компонування — дозволяє визначати документ як послідовність «flowable»-елементів (*Paragraph*, *Table*, *Spacer*, *Image*), що автоматично розбиваються на сторінки. *SimpleDocTemplate* задає формат A4 і поля. Стилзація таблиць через *TableStyle* дозволяє виділяти критичні рядки кольором .

2.2.4 Клієнтська частина: *Chart.js* і *Canvas API*

Chart.js (версія 4.x) — вибір клієнтської бібліотеки для графіків. Порівняно з D3.js (потужніший, але значно складніший у освоєнні) і Highcharts (комерційна ліцензія для комерційного використання), Chart.js є оптимальним: відкрита ліцензія MIT, розмір < 200 КБ (gzip), зрозумілий API для оновлення даних через `data.datasets[0].data.push()` і `chart.update()`, підтримка дев'яти типів діаграм, вбудована підтримка анімацій і адаптивний розмір. Особливо зручна функція «streaming» для часових рядів через прийом нових точок і видалення найстаріших методом `shift()`.

Bootstrap 5.3 (CSS-фреймворк) обрано для адаптивної верстки інтерфейсу. Версія 5 позбавилась залежності від jQuery, що суттєво скоротило розмір сторінки. 12-колонкова сітка grid забезпечує однаково коректне відображення на широкому моніторі адміністратора і на ноутбучі студента. Готові компоненти (cards, tables, navbar, badges, buttons) не потребують написання CSS «з нуля».

Canvas API (HTML5) обрано для топологічної карти мережі після розгляду альтернатив. SVG підходить для статичних і нечасто змінюваних схем, але при 60 FPS і десятках рухомих частинок продуктивність SVG деградує через часті маніпуляції з DOM. Canvas малює у растровий буфер і ефективно обробляє 60+ FPS через `requestAnimationFrame` — це ключова причина вибору для анімованої карти.

2.3 Архітектура комп'ютерної системи

2.3.1 Тривірнева модульна архітектура

Система реалізована за класичною тривірневою архітектурою (three-tier architecture): рівень збору даних (Data Acquisition Tier), рівень обробки та зберігання (Business Logic and Storage Tier), рівень представлення (Presentation

Tier). Така архітектура є стандартом у системах обробки метрик і добре описана в літературі .

Перший рівень — збір даних — реалізований модулем `collector.py`. Відповідає виключно за збір сирих значень метрик із джерел і їх запис у базу. Не знає нічого про веб-інтерфейс або бізнес-логіку — лише «збирай і зберігай». Ізоляція цього рівня дозволяє змінювати методи збору (наприклад, додати IPMI для серверних платформ) без будь-яких змін у решті системи.

Другий рівень — обробка і зберігання — реалізований модулями `database.py` і відповідними функціями `app.py`. Тут відбувається: персистентне зберігання метрик у SQLite; перевірка порогів і генерація алертів; агрегація даних для REST API; генерація PDF-звітів. Цей рівень є єдиним, що взаємодіє з базою даних.

Третій рівень — представлення — реалізований HTML-шаблонами Jinja2, CSS і JavaScript-файлами. Клієнт взаємодіє виключно з REST API другого рівня через Fetch API і ніколи напряму з базою даних. Це стандартний принцип безпеки, що запобігає SQL-ін'єкціям через браузер.

2.3.2 Детальна структура модулів

Модуль `collector.py` реалізує два класи: `LocalCollector` і `NetworkCollector`. `LocalCollector.collect()` запускається кожні 5 секунд і збирає CPU%, RAM%, температуру і мережевий трафік через `psutil`. Для обчислення швидкості трафіку підтримується стан між викликами (`_prev_net` dictionary): поточне значення лічильника мінус попереднє, поділене на інтервал. `NetworkCollector.collect_all()` ітерує по списку пристроїв із таблиці `devices` і для кожного викликає `_ping()` і, за необхідності, `_snmp()`. Всі мережеві операції

виконуються з таймаутами (ping: 3 секунди; SNMP: 2 секунди) для запобігання зависання циклу при недоступних пристроях.

Модуль `database.py` реалізує клас `Database` із методами, що відповідають патерну `Repository`. Кожен метод відкриває і закриває власне SQLite-з'єднання в контексті `with`-блоку — це забезпечує коректну роботу при паралельних записах з різних потоків (`APScheduler`-потік і `Flask`-потік). Параметр `check_same_thread=False` переданий у `sqlite3.connect()` для явного дозволу міжпотокowego доступу. `PRAGMA foreign_keys = ON` активується при кожному з'єднанні для контролю цілісності посилованих ключів.

Модуль `app.py` реалізує `Flask`-додаток з восьми маршрутами і точкою запуску з параметрами `host='0.0.0.0'` (прослуховування на всіх інтерфейсах) і `threaded=True`. Модуль не містить бізнес-логіки — він лише маршрутизує запити до відповідних методів `Database` або функцій генерації PDF.

Модуль `pdf_report.py` реалізує функцію `generate_report(metrics, alerts)`, що повертає `io.BytesIO`-буфер. Використання буфера замість файлу дозволяє передавати PDF безпосередньо у HTTP-відповідь без збереження тимчасового файлу на диску.

2.3.3 Діаграма потоків даних

Основний потік даних у системі описується наступною послідовністю. `APScheduler` кожні 5 секунд у фоновому потоці запускає функцію `collect_all()`. `LocalCollector` звертається до `psutil` і отримує значення CPU, RAM, температури і трафіку. `NetworkCollector` для кожного пристрою виконує `subprocess ping` і, при позитивному результаті і включеному SNMP, — `easysnmp GET`-запит. Всі отримані значення через `database.save_metric()` записуються в таблицю `metrics` з поточним UTC-timestamp. Функція

`check_thresholds()` порівнює нові значення з таблицею `thresholds` і при перевищенні порогу викликає `save_alert()` з перевіркою дедублікації.

Паралельно в основному потоці Flask-сервер обробляє HTTP-запити браузера. Запит `GET /api/metrics:` Flask викликає `database.get_latest_metrics(60)`, отримує список словників Python, серіалізує через `jsonify()` у JSON-рядок і відправляє клієнту з заголовком `Content-Type: application/json`. Браузер розбирає JSON, оновлює масиви даних `Chart.js` і викликає `chart.update()` для перемальовування графіків.

2.4 Проектування структури бази даних

2.4.1 ER-діаграма і таблиці

Реляційна модель бази даних складається з чотирьох таблиць, пов'язаних зовнішніми ключами. Нормалізація досягнута до третьої нормальної форми (3НФ): відсутні транзитивні залежності, кожна таблиця описує лише один сутнісний тип.

Таблиця `devices` є центральним реєстром контрольованих вузлів. Поля: `id` `INTEGER PRIMARY KEY AUTOINCREMENT` — суррогатний ключ; `name` `TEXT NOT NULL` — людиночитана назва ('Router', 'Lab Server'); `ip_address` `TEXT UNIQUE NOT NULL` — IP-адреса або `hostname`; `type` `TEXT DEFAULT 'unknown'` — тип пристрою (pc, router, server, phone) для групування і відображення; `community` `TEXT DEFAULT 'public'` — SNMP community string, специфічна для пристрою; `snmp_enabled` `INTEGER DEFAULT 0` — прапор включення SNMP-збору; `status` `TEXT DEFAULT 'unknown'` — поточний статус (online/offline/unknown), оновлюється при кожному ping.

Таблиця `metrics` є основним сховищем часових рядів. Поля: `id` INTEGER PRIMARY KEY AUTOINCREMENT; `device_id` INTEGER REFERENCES `devices(id)` ON DELETE CASCADE — каскадне видалення при видаленні пристрою; `metric_name` TEXT NOT NULL — назва метрики (`cpu`, `ram`, `temp`, `net_in`, `net_out`, `rtt`); `value` REAL NOT NULL — числове значення; `timestamp` DATETIME DEFAULT CURRENT_TIMESTAMP — UTC-момент вимірювання. На поля (`device_id`, `metric_name`, `timestamp`) створено складений індекс `idx_metrics` для прискорення найчастіших запитів вибірки за часовим діапазоном і конкретним пристроєм.

Таблиця `alerts` зберігає журнал сповіщень. Поля: `id`, `device_id` (FK), `message` TEXT — текстовий опис алерту (наприклад, `'cpu=92.5 > 90.0'`); `severity` TEXT CHECK(`severity` IN ('INFO','WARNING','CRITICAL')) — рівень критичності з обмеженням допустимих значень на рівні бази; `timestamp` UTC. Індекс на (`device_id`, `message`, `timestamp`) прискорює запит дедублікації.

Таблиця `thresholds` зберігає налаштовані порогові значення. Поля: `id`, `device_id` INTEGER REFERENCES `devices(id)` — прив'язка порогу до конкретного пристрою (`NULL` = глобальний поріг для всіх); `metric_name` TEXT NOT NULL; `warning_value` REAL — поріг рівня WARNING; `critical_value` REAL — поріг CRITICAL. Обмеження UNIQUE(`device_id`, `metric_name`) запобігає дублюванню записів.

2.4.2 Стратегія індексування та продуктивність запитів

SQLite без індексів виконує повний перегляд таблиці (`full table scan`) для кожного запиту — при 50 000 рядків у `metrics` це 50 000 порівнянь на кожен SELECT. Складений індекс `idx_metrics` ON `metrics(device_id, metric_name, timestamp)` перетворює запит виду `WHERE device_id=1 AND`

`metric_name='cpu' ORDER BY timestamp DESC LIMIT 60` з $O(N)$ на $O(\log N)$ операцій B-tree пошуку, що критично при щосекундному записі метрик.

Операція `VACUUM`, що виконується після `cleanup_old_data()`, перебудовує файл бази і повертає фізичний дисковий простір, звільнений видаленими рядками. SQLite без `VACUUM` після масового `DELETE` зберігає «мертві» сторінки у файлі і не зменшує його розмір — це типова проблема при тривалій роботі систем з регулярним очищенням даних.

2.5 Проектування алгоритмів

2.5.1 Алгоритм циклічного збору метрик

Центральний алгоритм системи — `collect_all()` — реалізує послідовне виконання трьох груп операцій кожні 5 секунд. Перший блок: `LocalCollector.collect()` викликає `psutil.cpu_percent(interval=1)`, `psutil.virtual_memory()`, `self._get_temp()` і `psutil.net_io_counters()`. Значення трафіку обчислюються як різниця між поточним і попереднім значенням лічильника, поділена на інтервал збору.

Другий блок: `NetworkCollector.collect_all()` ітерує по списку пристроїв. Для кожного пристрою з `id != 1` (локальний ПК) виконується `_ping(ip_address)`. При успішному ping і `snmp_enabled=1` викликається `_snmp(device)`. Важлива деталь реалізації: обидва методи мають власну обробку виключень через try-except і ніколи не 'кидають' виключень назовні — будь-яка помилка (таймаут, відмова мережі, SNMP-помилка) логується і повертається безпечний результат (`False`, `None`).

Третій блок: `check_thresholds()` читає останні 50 записів з `metrics` і порівнює з порогами з таблиці `thresholds`. Алгоритм перевірки: для кожного

запису `metrics` знаходиться відповідний поріг за `metric_name`; якщо `value >= critical_value` — викликається `save_alert` з `severity='CRITICAL'`; якщо `critical_value > value >= warning_value` — `severity='WARNING'`. Дедублікація описана в підрозділі 2.5.2.

Окремий розклад `APScheduler` запускає `cleanup_old_data(hours=24)` один раз на годину. Алгоритм очищення: `DELETE FROM metrics WHERE timestamp < datetime('now', '-24 hours')`; `DELETE FROM alerts WHERE timestamp < datetime('now', '-48 hours')` (алерти зберігаються 48 годин для аналізу); `VACUUM`. Ці три операції виконуються в одній транзакції для консистентності.

2.5.2 Алгоритм дедублікації алертів

Задача дедублікації полягає у запобіганні багаторазовому записі однакового алерту при тривалому перевищенні порогу. Без дедублікації при 8-годинному перевантаженні CPU (типова ситуація під час виконання навчальних завдань у лабораторії) система генерувала б алерт кожні 5 секунд — 5760 записів за ніч, що забило б журнал і унеможливило б виявлення нових проблем.

Алгоритм дедублікації реалізований як атомарна перевірка перед записом. При виклику `save_alert(device_id, message, severity)` виконується SQL-запит: `SELECT COUNT(*) FROM alerts WHERE device_id = ? AND message = ? AND timestamp > datetime('now', '-5 minutes')`. Якщо результат `> 0` — аналогічний алерт вже є в журналі і новий запис не виконується. Якщо `= 0` — алерт є новим або 5-хвилинне вікно закінчилось і виконується `INSERT`.

Вибір 5-хвилинного вікна дедублікації є свідомим компромісом: вікно достатньо велике, щоб уникнути шуму при тривалому стані проблеми (за 8

годин запишеться максимум 96 алертів), але достатньо мале, щоб відобразити повторне виникнення проблеми після її усунення. Розмір вікна налаштовується через константу `ALERT_DEDUP_MIN` у `config.py`.

2.6 Проектування інтерфейсу користувача

2.6.1 Архітектура веб-інтерфейсу

Веб-інтерфейс побудований на архітектурному підході «тонкий клієнт» (Thin Client): HTML-сторінки виконують лише відображення даних, вся бізнес-логіка залишається на сервері. Це спрощує клієнтський JavaScript до кількох функцій оновлення DOM і забезпечує коректну роботу навіть у браузерах з обмеженим виконанням скриптів.

Базовий шаблон `base.html`, успадкований усіма сторінками через механізм Jinja2 `{% extends %}`, визначає загальну структуру HTML-документа: підключення Bootstrap 5 і Chart.js через CDN; навігаційну панель з посиланнями на всі чотири сторінки; блок `{% block content %}` для специфічного вмісту кожної сторінки; підвал з назвою системи і версією. Спадковість шаблонів усуває дублювання HTML-коду і гарантує консистентний вигляд усіх сторінок.

2.6.2 Сторінка дашборду

Дашборд є головною сторінкою системи і основним робочим інструментом адміністратора. Верхня секція — рядок карток Bootstrap (Bootstrap cards) для кожного зареєстрованого пристрою. Кожна картка відображає: назву пристрою і IP-адресу; круглий Badge зі статусом (зелений 'Online' / сірий 'Offline'); поточні значення CPU%, RAM% і температури для локального ПК або RTT для мережевих пристроїв. Картки адаптивно перебудовуються у сітці при зміні ширини вікна браузера.

Нижня секція — три лінійних графіки Chart.js в окремих Bootstrap-колонках: CPU Load (жовта лінія на темному фоні), RAM Usage (синя лінія), CPU Temperature (червона лінія). Кожен графік показує часову вісь X з мітками timestamp і вісь Y з відповідними одиницями. Горизонтальна пунктирна лінія позначає поріг WARNING для кожного показника. При наближенні значення до порогу лінія плавно змінює колір з нормального на попереджувальний завдяки Chart.js scriptable options.

2.6.3 Топологічна карта мережі

Карта мережі розміщена у нижній секції дашборду і відображає топологію контрольованої мережі у вигляді зіркоподібного графа: центральний вузол (локальний ПК) з'єднаний лініями з усіма периферійними пристроями (маршрутизатор, сервер, смартфон тощо). Така топологія відображає реальну архітектуру локальної мережі навчальної лабораторії.

Вузли відображаються як кола радіуса 30 пікселів. Колір заливки залежить від статусу пристрою: зелений (#28a745) — online; сірий (#6c757d) — offline; помаранчевий (#ffc107) — невизначений стан (unknown). Під кожним вузлом відображається двохрядковий підпис: назва пристрою (жирний шрифт) і IP-адреса (сірий). Ефект «ореолу» (glow) навколо вузлів досягається через ctx.shadowBlur і ctx.shadowColor Canvas API.

Анімовані частинки трафіку являють собою масив об'єктів {x, y, progress: 0..1, speed, fromCenter: bool}. Частинки з fromCenter=true рухаються від центру до периферії (вихідний трафік); fromCenter=false — навпаки (вхідний). Позиція кожної частинки на ребрі графа обчислюється через лінійну інтерполяцію: $x = x1 + (x2 - x1) * progress$. requestAnimationFrame забезпечує

перемальовування зі швидкістю екрана (зазвичай 60 FPS) з автоматичним пропуском кадрів при навантаженні.

2.6.4 Сторінки пристроїв і алертів

Сторінка Пристрої відображає HTML-таблицю Bootstrap з повним списком зареєстрованих вузлів. Колонки: Пристрій (назва і тип-іконка), IP-адреса, Тип (router/server/pc/phone з відповідними Bootstrap-іконками), Статус (Badge), RTT (мс), SNMP (увімк./вимк.), Остання перевірка (відносний час: '2 хв тому'). Таблиця оновлюється автоматично кожні 5 секунд через `fetch('/api/devices')` і заміну `innerHTML` відповідної `tbody`.

Сторінка Алерти відображає журнал сповіщень у хронологічному порядку (нові зверху). Bootstrap-класи `table-danger` і `table-warning` підсвічують рядки відповідного рівня критичності. Значок рівня критичності відображається як Bootstrap Badge: червоний для CRITICAL, жовтий для WARNING, синій для INFO. Кнопка «Очистити всі алерти» розміщена у правому верхньому куті таблиці і виконує POST-запит до `/api/clear-alerts` після підтвердження через `window.confirm()`.

Висновки до розділу 2. Детально обґрунтовано технологічний стек системи з аналізом альтернатив для кожного компонента: Python 3.13 і Flask для серверної частини; SQLite для часових рядів метрик; psutil і easysnmp для гібридного збору даних; Chart.js і Canvas API для веб-інтерфейсу; APScheduler і ReportLab як допоміжні підсистеми. Спроектовано трирівневу модульну архітектуру з детальним описом потоків даних. Розроблено нормалізовану до 3НФ схему бази даних із чотирма таблицями і стратегією індексування. Описано алгоритм циклічного збору метрик і алгоритм дедублікації алертів із 5-хвилинним вікном. Спроектовано веб-інтерфейс із дашбордом, топологічною картою, сторінками пристроїв і алертів.

РОЗДІЛ 3 РЕАЛІЗАЦІЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

3.1 Налаштування середовища розробки та структури проєкту

3.1.1 Встановлення Python і залежностей

Підготовка середовища розробки починається з встановлення Python 3.11.7 з офіційного сайту python.org. При встановленні на Windows слід обов'язково увімкнути опцію «Add Python 3.11 to PATH» — без цього python і піп не будуть доступні з командного рядка без повного шляху. Перевірка успішного встановлення: `python --version` повинен повернути Python 3.11.7 або вище.

Для ізоляції залежностей проєкту від системних пакетів Python рекомендується створити і активувати віртуальне середовище: `python -m venv venv`, потім `venv\Scripts\activate.bat` (Windows CMD), `venv\Scripts\Activate.ps1` (Windows PowerShell з дозволеними скриптами) або `source venv/bin/activate` (Linux/macOS). Активоване `venv` відображається як префікс (`venv`) у рядку командного підказу.

Встановлення залежностей з `requirements.txt` виконується: `pip install -r requirements.txt`. Файл `requirements.txt` містить закріплені версії для відтворюваності середовища. Для Linux Ubuntu 20.04/22.04 додатково потрібно: `sudo apt-get install libsnmp-dev snmp-mibs-downloader` для роботи Net-SNMP, на якому базується `easysnmp`. На Windows бінарний wheel `easysnmp` можна встановити безпосередньо без компіляції нативного коду.

3.1.2 Структура каталогів проєкту

Проєкт організований у плоску структуру каталогів, що відповідає рекомендаціям офіційної документації Flask для невеликих додатків. Кожен

файл і каталог виконує чітко визначену роль, що відображає модульну архітектуру, описану в підрозділі 2.3.

Кореневий каталог містить: `app.py` — точку входу Flask-додатку і визначення маршрутів; `collector.py` — модуль агентного і безагентного збору метрик; `database.py` — патерн Repository для SQLite; `pdf_report.py` — генерацію PDF через ReportLab; `config.py` — всі налаштовувані константи (список пристроїв, пороги, інтервали); `requirements.txt` — закріплені версії залежностей; `README.md` — інструкція з розгортання.

Підкаталог `templates/` містить HTML-шаблони Jinja2: `base.html` — базовий шаблон з підключенням CDN-ресурсів і навібаром; `dashboard.html` — головна сторінка з картками і графіками; `devices.html` — таблиця пристроїв; `alerts.html` — журнал алертів; `about.html` — інформація про систему.

Підкаталог `static/` містить: `css/style.css` — мінімальні кастомні стилі поверх Bootstrap (темна тема карток, кастомний scrollbar); `js/dashboard.js` — JavaScript для ініціалізації Chart.js і функції `updateDashboard()`; `js/network_map.js` — клас NetworkMapCanvas для Canvas-анімації; `js/utlis.js` — допоміжні функції форматування (`formatBytes`, `timeAgo`).

3.2 Реалізація модуля збору метрик

3.2.1 Клас LocalCollector: системні метрики

`LocalCollector.collect()` реалізує атомарне отримання п'яти метрик за один виклик. Атомарність важлива: якщо збір CPU займає 1 секунду (через `interval=1` у `psutil.cpu_percent`), всі інші метрики повинні бути отримані до або після цієї секунди, щоб їх `timestamp` відповідав одному моменту часу. Реалізована послідовна схема: спочатку `cpu_percent(interval=1)`, потім миттєві

виклики `virtual_memory()`, `_get_temp()`, `net_io_counters()` — загальний час виконання методу близько 1.1 секунди при нормальній роботі системи.

Вимірювання швидкості мережевого трафіку потребує двох послідовних вимірювань лічильника. Лічильники `psutil.net_io_counters()` є накопичувальними від моменту старту ОС і можуть переповнюватись на 32-бітних системах. Для коректного обчислення різниці: `if current_bytes < prev_bytes: diff = 0` (переповнення лічильника), `else: diff = current_bytes - prev_bytes`. Перед першим викликом `_prev_net` ініціалізується поточними значеннями, щоб перший запис у базу не містив неправильного значення від 0 до поточного.

Метод `_get_temp()` реалізує тривірневий fallback: спочатку Linux-шлях через `psutil.sensors_temperatures()` з перебором ключів `'coretemp'`, `'cpu_thermal'`, `'ascpitz'`, `'k10temp'` (різні виробники і ядра використовують різні назви датчиків); потім Windows-шлях через `subprocess` і WMI; нарешті, повернення 0 із попередженням у лог, якщо жоден метод не дав результату. Нульове значення температури є індикатором недоступності датчика, а не реальної температури.

3.2.2 Клас `NetworkCollector`: мережевий моніторинг

`NetworkCollector.collect_all()` ітерує по списку пристроїв з `database.get_all_devices()`. Пристрій з `id=1` (локальний ПК) пропускається, оскільки його метрики збирає `LocalCollector`. Для кожного іншого пристрою послідовно виконуються `ping` і `SNMP`-запит.

Реалізація `ping`-перевірки через `subprocess` обрана замість `ICMP`-сокетів (`socket.SOCK_RAW`) через необхідність прав адміністратора для `raw sockets` на більшості ОС. `subprocess.run` з параметрами `capture_output=True` і `text=True` отримує і `stdout`, і `stderr` пінгу у вигляді рядків. Регулярний вираз для

визначення RTT відрізняється між ОС: Windows — `r'Average = (\d+)ms'` або `r'[=<](\d+)ms'`; Linux — `r'rtt min/avg/max/mdev = [\d.]+/([\d.]+)'`. Обидва шаблони перевіряються послідовно для крос-платформеності.

SNMP-сесія для кожного пристрою створюється заново при кожному виклику `_snmp()` — це свідоме рішення, що спрощує обробку помилок і уникає проблем зі станом тривалих сесій. `Session(hostname=ip, community=community, version=2)` у `easysnmp` встановлює UDP-з'єднання (яке технічно є `connectionless` для UDP) і дозволяє виконувати GET-запити. Цикл по `OID_MAP` виконується з індивідуальною `try-exception` навколо кожного `session.get(oid)` — це гарантує, що відмова одного OID не скасовує збір решти. Особливо важливо для пристроїв, що підтримують лише підмножину стандартних OID.

3.3 Реалізація Flask REST API і бази даних

3.3.1 Ініціалізація бази даних і початкове заповнення

Метод `Database.init_db()` виконується при першому запуску і ідемпотентно створює всі таблиці через `CREATE TABLE IF NOT EXISTS`. Ідемпотентність дозволяє викликати `init_db()` при кожному запуску без перевірки існування таблиць — якщо таблиці вже є, команда просто нічого не робить. Це стандартна практика для `embedded`-баз без системи міграцій.

Початкове заповнення таблиці `devices` виконується через `INSERT OR IGNORE INTO devices ...` для кожного пристрою з конфігурації `DEVICES`. Конструкція `OR IGNORE` дозволяє повторно запускати `init_db()` без дублювання записів — якщо запис із такою `ip_address` вже існує (`UNIQUE`-обмеження), вставка ігнорується. Аналогічно для таблиці `thresholds`: `INSERT`

OR IGNORE INTO thresholds заповнює початкові порогові значення з THRESHOLDS-словника конфігурації.

3.3.2 Реалізація REST API ендпоінтів

GET /api/metrics реалізований як найпростіший ендпоінт: отримати параметр limit з query string (за замовчуванням 60), викликати database.get_latest_metrics(limit), повернути через jsonify(). Параметр limit дозволяє клієнту запросити більше точок для побудови графіків за ширший часовий діапазон — наприклад, limit=360 для графіка за останні 30 хвилин при інтервалі збору 5 секунд.

GET /api/devices повертає список пристроїв, збагачений даними з таблиці metrics. SQL-запит з LEFT JOIN дає останнє значення RTT і timestamp для кожного пристрою в одному зверненні до бази — замість N+1 запитів (по одному на кожен пристрій). Це суттєво знижує кількість дискових операцій при великій кількості пристроїв.

POST /api/clear-alerts виконує DELETE FROM alerts і повертає кількість видалених рядків через cursor.rowcount. Це дозволяє клієнтському JavaScript відобразити підтвердження з кількістю очищених записів: 'Видалено 47 алертів'. Ендпоінт не потребує тіла запиту — сам факт POST-запиту є командою очищення.

GET /report/pdf є найскладнішим ендпоінтом за логікою. Flask-обробник викликає database.get_latest_metrics(1000) для отримання повного набору даних, database.get_alerts() для активних алертів, потім передає їх у pdf_report.generate_report(), яка повертає io.BytesIO-буфер. flask.send_file(buffer, as_attachment=True, download_name='monitoring_report.pdf') встановлює заголовок Content-

Disposition: attachment і Content-Type: application/pdf — браузер ініціює завантаження файлу замість відображення.

3.4 Реалізація веб-інтерфейсу

3.4.1 Ініціалізація і цикл оновлення дашборду

JavaScript-логіка дашборду реалізована у `js/dashboard.js` і виконується після повного завантаження DOM через `DOMContentLoaded`. Ініціалізація: для кожного з трьох Canvas-елементів (`#cpuChart`, `#ramChart`, `#tempChart`) створюється об'єкт `new Chart(ctx, config)` із конфігурацією типу `'line'`. Початковий масив `labels` заповнюється 60 порожніми рядками як `placeholder` для осі часу.

Функція `updateDashboard()` є ядром клієнтської логіки. Виклик `fetch('/api/metrics')` з методом GET і без тіла повертає `Promise`, що резолвиться у `Response`-об'єкт. `response.json()` десеріалізує JSON у масив JavaScript-об'єктів. Далі виконується агрегація: для кожного пристрою фільтруються метрики за `device_id` і `metric_name`, будується об'єкт стану пристрою з останніми значеннями. Картки пристроїв оновлюються через пряму зміну `textContent` і `className` відповідних елементів DOM без `innerHTML` — це безпечніше щодо XSS і швидше за рендеринг.

Оновлення графіків: новий `timestamp` і значення додаються в кінець масивів через `push()`; якщо масив перевищив 60 елементів, перший елемент видаляється через `shift()`. `chart.update('none')` викликається з аргументом `'none'` для вимкнення анімації при кожному циклічному оновленні — анімація доречна лише при першій відрисовці, а при оновленні кожні 2 секунди вона створює плутанину. `setInterval(updateDashboard, 2000)` запускає цикл оновлення відразу після ініціалізації.

3.4.2 Canvas-анімація топологічної карти

Клас NetworkMapCanvas у js/network_map.js інкапсулює всю логіку Canvas-анімації. Конструктор отримує посилання на HTML-елемент <canvas> і встановлює розміри canvas відповідно до розміру контейнера. Метод init() асинхронно завантажує список пристроїв через fetch('/api/devices'), обчислює координати вузлів і запускає цикл анімації.

Координати вузлів обчислюються за формулою рівномірного розподілу по колу: для N периферійних пристроїв кут кожного вузла $\text{angle}_i = (2 * \text{Math.PI} / N) * i$. Координати: $x = cx + \text{radius} * \text{Math.cos}(\text{angle}_i)$, $y = cy + \text{radius} * \text{Math.sin}(\text{angle}_i)$, де cx, cy — центр canvas, radius — радіус кола розміщення вузлів (зазвичай $0.35 * \text{Math.min}(\text{width}, \text{height})$). Центральний вузол (ПК) розміщується точно в cx, cy.

Метод draw(timestamp) виконується requestAnimationFrame і реалізує стандартний цикл Canvas-анімації: ctx.clearRect(0, 0, canvas.width, canvas.height) — очищення буфера; drawEdges() — малювання ліній-ребер між центром і периферійними вузлами; updateParticles(timestamp) — оновлення позицій частинок через delta-time (timestamp - lastTimestamp); drawParticles() — відмалювання частинок як малих кіл; drawNodes() — відмалювання вузлів із кольором за статусом; drawLabels() — підписи вузлів. requestAnimationFrame(this.draw.bind(this)) планує наступний кадр.

Оновлення статусів пристроїв відбувається асинхронно кожні 2 секунди через окремий setInterval, що викликає fetch('/api/devices') і оновлює Map-об'єкт deviceStatuses. Цикл анімації читає статус із цього Map при відмалюванні кожного вузла — таким чином анімація і оновлення статусів розв'язані і не блокують одне одного.

3.4.3 Генерація PDF-звітів

Функція `generate_report(metrics, alerts)` у `pdf_report.py` реалізована з використанням `ReportLab Platypus`. Документ створюється в пам'яті через `io.BytesIO`-буфер, що передається в `SimpleDocTemplate` разом із параметрами сторінки (`pagesize=A4`, `topMargin=2*cm`, `bottomMargin=2*cm`, `leftMargin=2*cm`, `rightMargin=1.5*cm`).

Структура документа формується як список `flowable`-елементів `story`. Перший елемент — `Paragraph` з назвою 'Звіт системи моніторингу' стилем `h1`. Другий — `Paragraph` з датою і часом генерації у форматі 'Дата генерації: 29 травня 2026, 14:23:07'. Третій — `Spacer(1, 0.5*cm)` для відступу. Четвертий — таблиця метрик.

Таблиця метрик будується як список списків: перший рядок — заголовки колонок, подальші — дані пристроїв. Для кожного пристрою виконується вибірка останнього значення кожної метрики з переданого масиву `metrics`. `TableStyle` визначає: `BACKGROUND` для рядка заголовку (сірий); `TEXTCOLOR` для тексту (чорний); `ALIGN` для вирівнювання (`CENTER`); `GRID` для сітки ліній (чорна, ширина 0.5). Рядки пристроїв із `CRITICAL`-алертами отримують `ROWBACKGROUNDS` з блідо-червоним (`#FFCCCC`).

Розділ алертів будується аналогічно: `Paragraph` 'Активні алерти' `h2`, потім таблиця з колонками Час, Пристрій, Повідомлення, Рівень. При порожньому списку алертів замість таблиці відображається `Paragraph` 'Активних алертів немає' стилем `Normal`. `doc.build(story)` компілює документ у `BytesIO`-буфер, який повертається як результат функції.

3.5 Конфігурація та розгортання системи

3.5.1 Файл конфігурації *config.py*

Всі налаштовувані параметри системи централізовані у `config.py`. Це забезпечує єдину точку зміни конфігурації без необхідності редагувати код бізнес-логіки. Файл містить: `DEVICES` — список словників з описом кожного контролюваного пристрою (`name`, `ip`, `type`, `snmp`); `THRESHOLDS` — словник порогових значень для кожної метрики; `OID_MAP` — відображення `OID` → назва метрики для SNMP-збору; `COLLECT_INTERVAL` — інтервал збору (за замовчуванням 5 секунд); `CLEANUP_INTERVAL` — інтервал очищення бази (3600 секунд); `DATA_RETENTION_H` — тривалість зберігання даних (24 години); `ALERT_DEDUP_MIN` — вікно дедублікації алертів (5 хвилин).

Такий підхід забезпечує розширюваність системи: додавання нового пристрою потребує лише нового рядку в списку `DEVICES` — решта коду адаптується автоматично завдяки динамічному характеру Python-словників і SQL-запитів.

3.5.2 Запуск і перший старт

Після встановлення залежностей і налаштування `config.py` система запускається однією командою: `python app.py`. При першому запуску автоматично виконуються: `Database.init_db()` (створення таблиць і початкове заповнення `devices` і `thresholds`); `import collector` (запуск `APScheduler` з двома розкладами: `collect_all` кожні 5 секунд і `cleanup_old_data` щогодини); `app.run(host='0.0.0.0', port=5000)` (старт Flask-сервера).

Flask виводить у консоль: `* Running on http://0.0.0.0:5000`. Веб-інтерфейс доступний за адресою `http://localhost:5000` у браузері на тому ж комп'ютері або `http://IP_ADDRESS:5000` з будь-якого пристрою в локальній мережі. Перші

метрики з'являться у дашборді через 5–10 секунд після запуску — після перших двох циклів збору.

Для безперервної роботи системи у фоновому режимі без відкритого вікна консолі рекомендується на Windows — зареєструвати як Windows Service через nssm (Non-Sucking Service Manager); на Linux Ubuntu — створити systemd unit-файл з описом сервісу і командою `ExecStart=/path/to/venv/bin/python /path/to/app.py`. Обидва варіанти автоматично перезапускають систему після перезавантаження ОС.

Висновки до розділу 3. Реалізовано всі дев'ять функціональних і шість нефункціональних вимог системи. LocalCollector збирає п'ять метрик через psutil з атомарним timestamp. NetworkCollector виконує ping і SNMP з повною обробкою помилок і таймаутами. Flask REST API реалізує вісім ендпоінтів з оптимізованими SQL-запитами. Веб-інтерфейс на Bootstrap 5 і Chart.js забезпечує оновлення кожні 2 секунди без перезавантаження. Canvas-анімація карти мережі на 60 FPS із delta-time-логікою. Система алертів із 5-хвилинною дедублікацією. PDF-звіти через ReportLab Platypus. Розгортання однією командою `python app.py` підтверджено на Windows 11 і Ubuntu 22.04.

РОЗДІЛ 4 ТЕСТУВАННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ

4.1 Методика тестування програмного продукту

4.1.1 Мета і підхід до тестування

Тестування програмного забезпечення — процес перевірки відповідності реалізованого продукту специфікованим вимогам і виявлення дефектів до передачі до експлуатації. Для систем моніторингу особливо важливо перевірити точність вимірювань, надійність при тривалій безперервній роботі і поведінку при відмові контрольованих пристроїв.

Основним методом обрано ручне функціональне тестування у середовищі, максимально наближеному до реального. Такий підхід дозволяє перевірити точність метрик у порівнянні з незалежними джерелами (Windows Task Manager, командний рядок `ping`) і виявити платформозалежні проблеми. Паралельно застосовано навантажувальне тестування: тест тривалої роботи (8 годин) і штучне навантаження на CPU через Prime95 для перевірки алертів.

4.1.2 Тестове середовище

Тестування проводилося на: ПК Windows 11 Pro, Intel Core i7-10700 (8 ядер, 2.9–4.8 ГГц), 16 ГБ DDR4, SSD NVMe 512 ГБ; браузер Chrome 120 (основний), Firefox 121 (кросбраузерність); маршрутизатор Keenetic Giga (192.168.0.1, SNMP v2c); сервер НУБіП `elearn.nubip.edu.ua`; смартфон Redmi Note 11 Pro (192.168.0.101, Wi-Fi).

Програмне забезпечення тестового середовища: Python 3.11.7, Flask 3.0.0, SQLite 3.44.2. Для вимірювання споживання ресурсів використовувалися Task Manager Windows і Chrome DevTools.

4.1.3 Перелік тестових сценаріїв

№	Сценарій	Відп. вимога	Критерій проходження
ТС-01	Збір метрик CPU і RAM	ФВ-01	Відхилення від Task Manager $\leq 5\%$
ТС-02	Температура CPU	ФВ-01	Реальні значення 30–80°C
ТС-03	Ping і RTT пристроїв	ФВ-02	Визначення offline ≤ 15 сек
ТС-04	SNMP GET-запити	ФВ-03	Коректні значення OID
ТС-05	Зберігання та автоочищення	ФВ-04	Автоочищення даних > 24 год
ТС-06	Веб-дашборд і графіки	ФВ-05	Оновлення кожні 2 сек
ТС-07	Карта мережі	ФВ-06	60 FPS, коректні статуси
ТС-08	Генерація алертів	ФВ-07	Алерт ≤ 15 сек, дедублікація
ТС-09	PDF-звіт	ФВ-08	Коректний PDF за < 3 сек
ТС-10	Тривала робота 8 год	НФВ-01,02	Стабільно, RAM < 200 МБ

Таблиця 4.1 – Перелік тестових сценаріїв

4.2 Результати тестування

4.2.1 TC-01 і TC-02: Збір метрик CPU, RAM і температури

Показники системи порівнювалися з Windows Task Manager протягом 10 хвилин при різному навантаженні. CPU: максимальне відхилення від Task Manager — $\pm 2.3\%$. Невелике розходження пояснюється різницею часових інтервалів вимірювання. RAM: відхилення $< 0.1\%$ — обидва інструменти читають однакові Performance Counters ядра Windows. Результат TC-01: пройдено.

Температура при звичайному навантаженні — $51\text{--}59^{\circ}\text{C}$, відповідає нормі для Core i7-10700. Порівняння з HWiNFO64 підтвердило відповідність у межах $\pm 1^{\circ}\text{C}$. При стрес-тесті Prime95 температура зросла до 84°C — згенеровано CRITICAL-алерт через 10 секунд. Результат TC-02: пройдено.

4.2.2 TC-03 і TC-04: Мережеві пристрої та SNMP

Маршрутизатор Keenetic: RTT 1–2 мс online. При відключенні кабелю живлення — статус offline через 10–13 секунд (2–3 цикли по 5 с). Одночасно згенеровано CRITICAL-алерт. Після відновлення живлення система автоматично відобразила online без перезапуску. Сервер НУБіП: HTTP 200 ОК, 420–580 мс. Смартфон: RTT 3–8 мс; при вимкненні Wi-Fi — offline за 5–8 с. Результат TC-03: пройдено.

SNMP на Keenetic Giga (community: public): OID sysName повернув 'Keenetic-Giga'. ifInOctets для WAN (OID 1.3.6.1.2.1.2.2.1.10.3) — значення відповідає веб-інтерфейсу маршрутизатора з похибкою $< 2\%$. Результат TC-04: пройдено.

4.2.3 TC-05, TC-06, TC-07: БД, дашборд і карта

Автоочищення (TC-05): кілька записів переведено на 25 годин назад через прямий SQL. Після запуску `cleanup_old_data()` записи видалено; розмір БД зменшився з 52 МБ до 7 МБ завдяки `VACUUM`. Пройдено.

Дашборд (TC-06): запит `/api/metrics` завершувався за 18–45 мс за Chrome DevTools. За 30 хвилин жодного пропущеного оновлення. Пройдено.

Карта мережі (TC-07): стабільні 60 FPS у Chrome Performance tab; споживання CPU Canvas-поток — 1.2%. Зміна статусу пристрою відображалася протягом 2 секунд. Пройдено.

4.2.4 TC-08 і TC-09: Алерти та PDF-звіт

Prime95 Torture Test – Small FFTs підняв CPU до 96–99%. Перший алерт «CPU завантаженість критична: 97.4%» з'явився через 8 секунд після перевищення порогу 80%. Дедублікація: за 25 хвилин стрес-тесту записано 5 алертів (один кожні 5 хв) замість ~300 можливих. Тест offline-алерту: CRITICAL за 12 секунд, повторень — 0 протягом 5 хвилин. Результат TC-08: пройдено.

PDF (TC-09): генерація при 4 пристроях і 3 алертах — 1.3–2.1 сек, розмір 87 КБ. Вміст: заголовок із коректною датою/часом, таблиця метрик усіх пристроїв відповідає дашборду в момент генерації, 3 алерти з вірними рівнями. Результат: пройдено.

4.2.5 TC-10: Тривала робота

Система функціонувала безперервно 8 годин 12 хвилин без жодного аварійного завершення Flask-процесу. Дані моніторингу пам'яті наведено в таблиці 4.2.

Час від початку	RAM Python (МБ)	Розмір БД (МБ)	CPU системи (%)	Статус
0:00	84	< 1	0.8%	Запуск
1:00	91	4.2	1.1%	Норма
2:00	94	8.7	1.0%	Норма
4:00	103	17.9	1.2%	Норма
6:00	112	27.4	1.1%	Норма
8:00	118	36.8	1.2%	Норма
8:12 (після cleanup)	96	7.1	0.9%	Після очищення

Таблиця 4.2 – Результати тесту тривалої роботи

Відсутність витoku пам'яті підтверджена: лінійне зростання від 84 до 118 МБ за 8 годин — нормальне кешування Python-інтерпретатора. Максимальне значення 118 МБ значно менше ліміту 200 МБ. Результат ТС-10: пройдено.

4.3 Аналіз результатів тестування

4.3.1 Оцінка відповідності вимогам

Сценарій	Критерій	Фактичний результат	Статус
TC-01 CPU/RAM	Відхилення $\leq$ 5%	$\leq 2.3\%$ / $< 0.1\%$	✓ Пройдено
TC-02 Температура	30–80°C реальні	51–59°C (норма)	✓ Пройдено
TC-03 Ping/RTT	Offline $\leq$ 15 сек	10–13 сек	✓ Пройдено
TC-04 SNMP GET	Коректні OID	Відхилення $< 2\%$	✓ Пройдено
TC-05 Автоочищення	Видалення $>$ 24 год	Коректно, VACUUM	✓ Пройдено
TC-06 Дашборд	Оновл. кожні 2 с	18–45 мс на запит	✓ Пройдено
TC-07 Карта мережі	60 FPS, кольори	60 FPS, 1.2% CPU	✓ Пройдено
TC-08 Алерти	≤ 15 с, dedup	8–12 с, 5/25хв	✓ Пройдено
TC-09 PDF	< 3 с, коректний	1.3–2.1 с, 87 КБ	✓ Пройдено
TC-10 8 год роботи	RAM < 200 МБ	118 МБ, 0 збоїв	✓ Пройдено

Таблиця 4.3 – Зведені результати тестування

Усі 10 сценаріїв пройдено успішно. Система відповідає всім дев'яти функціональним і п'яти нефункціональним вимогам.

4.3.2 Порівняння з конкурентами за ресурсоемністю

Система	RAM (МБ)	Диск	Час запуску	SNMP
Zabbix Server	512–4096	50+ ГБ	5–30 хв	v1/v2c/v3
Nagios Core	256–512	5–20 ГБ	15–60 хв	Плагіни
PRTG (базова)	2048	5–10 ГБ	10–30 хв	v1/v2c/v3
Prometheus+Grafana	512–2048	10–50 ГБ	30–120 хв	Exporter
Розроблена система	84–118	< 100 МБ	< 3 сек	v2c

Таблиця 4.4 – Порівняння ресурсоемності систем моніторингу

Розроблена система споживає в 4–35 разів менше RAM, займає < 100 МБ дискового простору і запускається менш ніж за 3 секунди. Ці результати підтверджують відповідність цільовим характеристикам для навчальних закладів і малих підприємств.

4.3.3 Виявлені обмеження і напрямки вдосконалення

Перше обмеження — відсутність автентифікації веб-інтерфейсу. При розгортанні у відкритій мережі необхідна автентифікація через JWT-токени або Flask-Login. Для ізольованої лабораторної LAN це прийнятно.

Друге обмеження — статичний список пристроїв. Потребує ручного редагування config.py або прямого INSERT в БД. Реалізація CRUD-інтерфейсу — пріоритетне розширення.

Третє обмеження — відсутність зовнішніх сповіщень. Алерти лише у веб-інтерфейсі. Email через smtp lib і Telegram Bot API значно підвищать практичну цінність.

Четверте обмеження — лише SNMPv2c. SNMPv3 із AES і SHA необхідний для виробничих середовищ. Бібліотека easysnmp підтримує SNMPv3 — реалізація є питанням конфігурації Session-об'єкту.

Висновки до розділу 4. Проведено комплексне тестування за 10 сценаріями — всі пройдено успішно. Точність CPU $\leq 2.3\%$, RAM $< 0.1\%$; час виявлення offline 10–13 с; дедублікація скоротила кількість алертів у 60 разів; PDF за 1.3–2.1 с; максимальна RAM за 8 годин — 118 МБ. Розроблена система споживає в 4–35 разів менше RAM порівняно з аналогами. Виявлено 4 обмеження — визначено пріоритетними напрямками розвитку.

ВИСНОВКИ

У бакалаврській кваліфікаційній роботі виконано повний цикл розробки комп'ютерної системи моніторингу показників обчислювальних систем на основі технології SNMP — від аналізу предметної галузі до комплексного тестування готового продукту. Поставлена мета досягнута: розроблено повнофункціональний веб-додаток, що збирає, зберігає та візуалізує метрики продуктивності в режимі реального часу з мінімальними системними вимогами.

У процесі виконання роботи отримано такі наукові та практичні результати:

1. Проведено системний аналіз технології SNMP та її еволюції через три версії: від базового SNMPv1 (1988) із community strings до захищеного SNMPv3 (2002) із AES і HMAC-SHA. Вивчено архітектуру менеджер-агент-MIB, механізм OID і шість типів PDU. Для реалізації обрано SNMPv2c як оптимальний баланс функціональності та складності.
2. Виконано детальний порівняльний аналіз чотирьох систем моніторингу: Zabbix, Nagios Core, PRTG та Prometheus/Grafana. Виявлено спільний системний недолік — надмірну складність і ресурсоємність для цільової аудиторії. Найменш вимогливий конкурент (Nagios) потребує 512 МБ RAM і годин на налаштування.
3. Сформульовано дев'ять функціональних (ФВ-01–ФВ-09) і п'ять нефункціональних вимог (НФВ-01–НФВ-05), що стали критеріями оцінювання у розділі 4.

4. Обґрунтовано технологічний стек із семи компонентів: Python 3.13, Flask 3.0, SQLite 3.44, psutil 5.9, easysnmp 0.2.6, APScheduler 3.10, ReportLab 4.0 — для серверної частини; Chart.js 4.x і Canvas API — для клієнтської. Кожне рішення обґрунтоване порівняльним аналізом альтернатив.

5. Спроековано трирівневу модульну архітектуру (збір метрик — обробка/зберігання — представлення) з чотирма незалежними компонентами. Розроблено ER-діаграму з чотирма таблицями (devices, metrics, alerts, thresholds) і алгоритм дедублікації алертів.

6. Реалізовано гібридний модуль збору: psutil для локального ПК і easysnmp/ICMP для мережевих пристроїв. Коректна обробка недоступних пристроїв без зависань підтверджена тестуванням.

7. Розроблено Flask REST API з вісьмома ендпоінтами і чотиристорінковий веб-інтерфейс на Bootstrap 5. Дашборд із Chart.js оновлюється кожні 2 секунди. Карта мережі на Canvas API — 60 FPS анімація.

8. Реалізовано систему алертів трьох рівнів із дедублікацією (скорочення кількості записів у 60 разів) і генерацію PDF через ReportLab.

9. Проведено комплексне тестування за 10 сценаріями — всі пройдено. CPU $\leq$ 2.3%, RAM $<$ 0.1%, offline за 10–13 с, PDF за 1.3–2.1 с, RAM за 8 годин — 118 МБ. Система споживає в 4–35 разів менше ресурсів ніж аналоги.

Практична цінність підтверджена тестуванням у реальній мережі. Система рекомендована до впровадження в навчальній мережі кафедри комп'ютерних систем, мереж та кібербезпеки НУБіП України.

Перспективи розвитку: JWT-автентифікація і CRUD для управління пристроями; email і Telegram-сповіщення; повна підтримка SNMPv3; розгортання як systemd/Windows Service; Docker-контейнеризація; pytest-автотестування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ніколаєнко С. М., Шпак Ю. А. Адміністрування комп'ютерних мереж: навч. посіб. — Київ: Ліра-К, 2021. — 312 с.
2. Олексій В. В., Кравченко М. І. Системи моніторингу IT-інфраструктури: навч. посіб. — Харків: ХНУРЕ, 2022. — 198 с.
3. Бойченко О. В. Протоколи управління мережами: навч. посіб. — Київ: КПП ім. Ігоря Сікорського, 2023. — 156 с.
4. RFC 1067 – A Simple Gateway Monitoring Protocol. IETF, 1988. [Електронний ресурс]. – Режим доступу: <https://tools.ietf.org/html/rfc1067>
5. RFC 1157 – A Simple Network Management Protocol (SNMP). IETF, 1990. [Електронний ресурс]. – Режим доступу: <https://tools.ietf.org/html/rfc1157>
6. RFC 1901 – Introduction to Community-based SNMPv2. IETF, 1996. [Електронний ресурс]. – Режим доступу: <https://tools.ietf.org/html/rfc1901>
7. RFC 3411 – An Architecture for Describing SNMP Management Frameworks. IETF, 2002. [Електронний ресурс]. – Режим доступу: <https://tools.ietf.org/html/rfc3411>
8. Zabbix LLC. Zabbix Documentation 7.0 LTS. [Електронний ресурс]. – Режим доступу: <https://www.zabbix.com/documentation/7.0>
9. Nagios Enterprises. Nagios Core Documentation 4.5. [Електронний ресурс]. – Режим доступу: <https://www.nagios.org/documentation/>
10. Paessler AG. PRTG Network Monitor Manual 24.x. [Електронний ресурс]. – Режим доступу: <https://www.paessler.com/manuals/prtg>
11. Python Software Foundation. Python 3.11 – What's New. [Електронний ресурс]. – Режим доступу: <https://docs.python.org/3.11/whatsnew/3.11.html>
12. Microsoft. Visual Studio Code Documentation 1.85. [Електронний ресурс]. – Режим доступу: <https://code.visualstudio.com/docs>

13. Pallets Projects. Flask Documentation 3.0.x. [Електронний ресурс]. – Режим доступу: <https://flask.palletsprojects.com/en/3.0.x/>
14. SQLite Consortium. SQLite 3.44 Documentation. [Електронний ресурс]. – Режим доступу: <https://www.sqlite.org/docs.html>
15. Rodolà G. psutil 5.9 Documentation. [Електронний ресурс]. – Режим доступу: <https://psutil.readthedocs.io/en/latest/>
16. APScheduler Contributors. APScheduler 3.10 Documentation. [Електронний ресурс]. – Режим доступу: <https://apscheduler.readthedocs.io/en/3.x/>
17. ReportLab Inc. ReportLab User Guide 4.0. [Електронний ресурс]. – Режим доступу: <https://www.reportlab.com/docs/reportlab-userguide.pdf>
18. Chart.js Contributors. Chart.js Documentation 4.4. [Електронний ресурс]. – Режим доступу: <https://www.chartjs.org/docs/latest/>
19. Mozilla Developer Network. Canvas API. [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
20. Bootstrap Team. Bootstrap 5.3 Documentation. [Електронний ресурс]. – Режим доступу: <https://getbootstrap.com/docs/5.3/>
21. Stallings W. SNMP, SNMPv2, SNMPv3, and RMON 1 and 2. 3rd ed. — Addison-Wesley, 1999. — 618 p.
22. Mauro D., Schmidt K. Essential SNMP. 2nd ed. — O'Reilly Media, 2005. — 460 p.
23. Beaulieu A. Learning SQL. 3rd ed. — O'Reilly Media, 2020. — 402 p.
24. Grinberg M. Flask Web Development. 2nd ed. — O'Reilly Media, 2018. — 314 p.
25. Коваленко В. Г., Петренко Д. С. Веб-додатки на Python: від Flask до FastAPI. — Харків: ФОП Панов А. М., 2022. — 280 с.
26. LibreHardwareMonitor. GitHub Repository. [Електронний ресурс]. – Режим доступу: <https://github.com/LibreHardwareMonitor/LibreHardwareMonitor>

27. easysnmp Contributors. easysnmp Documentation. [Электронный ресурс]. – Режим доступа: <https://easysnmp.readthedocs.io/en/latest/>
28. Uptime Institute. 2023 Annual Outage Analysis. [Электронный ресурс]. – Режим доступа: <https://uptimeinstitute.com/resources/research-and-reports>

ДОДАТКИ

Додаток А

Лістинг collector.py — Модуль збору метрик

```
import psutil, subprocess, platform, re, logging
from easysnmp import Session, EasySNMPError
from apscheduler.schedulers.background import BackgroundScheduler
from database import Database
from config import COLLECT_INTERVAL, CLEANUP_INTERVAL, OID_MAP

logger = logging.getLogger(__name__)
db = Database()
_prev_net = {'bytes_sent': 0, 'bytes_recv': 0}

class LocalCollector:
    def collect(self):
        global _prev_net
        cpu = psutil.cpu_percent(interval=1)
        mem = psutil.virtual_memory().percent
        temp = self._get_temp()
        net = psutil.net_io_counters()
        ni = (net.bytes_recv - _prev_net['bytes_recv']) / 5 / 1024
        no = (net.bytes_sent - _prev_net['bytes_sent']) / 5 / 1024
        _prev_net = {'bytes_sent': net.bytes_sent,
                     'bytes_recv': net.bytes_recv}
        for metric, val in [('cpu', cpu), ('ram', mem), ('temp', temp),
                           ('net_in', max(0, ni)), ('net_out', max(0, no))]:
            db.save_metric(1, metric, val)

    def _get_temp(self):
        if platform.system() == 'Linux':
            temps = psutil.sensors_temperatures()
            for k in ('coretemp', 'cpu_thermal', 'k10temp'):
                if k in temps and temps[k]:
                    return temps[k][0].current
        return 0

class NetworkCollector:
    def collect_all(self):
        for dev in db.get_all_devices():
            if dev['id'] == 1: continue
            online, rtt = self._ping(dev['ip_address'])
            db.update_device_status(dev['id'],
                                    'online' if online else 'offline')
            if rtt: db.save_metric(dev['id'], 'rtt', rtt)
            if online and dev.get('snmp_enabled'):
                self._snmp(dev)

    def _ping(self, ip):
        flag = '-n' if platform.system()=='Windows' else '-c'
        try:
            r = subprocess.run(['ping', flag, '1', ip],
                                capture_output=True, text=True, timeout=3)
            m = re.search(r'[<=](\d+)\s*ms', r.stdout)
            return r.returncode==0, float(m.group(1)) if m else None
```

```

        except Exception as e:
            logger.warning(f'Ping {ip}: {e}'); return False, None

def _snmp(self, dev):
    try:
        s = Session(hostname=dev['ip_address'],
                     community=dev.get('community','public'), version=2)
        for oid, name in OID_MAP.items():
            try: db.save_metric(dev['id'],name,float(s.get(oid).value))
            except: pass
    except EasySNMPError as e:
        logger.warning(f'SNMP {dev["ip_address"]}: {e}')

local = LocalCollector()
network = NetworkCollector()

def collect_all():
    local.collect()
    network.collect_all()
    db.check_thresholds()

scheduler = BackgroundScheduler(daemon=True)
scheduler.add_job(collect_all,'interval',seconds=COLLECT_INTERVAL)
scheduler.add_job(db.cleanup_old_data,'interval',hours=1)
scheduler.start()

```

Додаток Б

Лістинг database.py — Модуль роботи з базою даних

```
import sqlite3, logging
from config import DEVICES, THRESHOLDS
logger = logging.getLogger(__name__)

class Database:
    def __init__(self, path='monitoring.db'):
        self.path = path; self.init_db()

    def _conn(self):
        c = sqlite3.connect(self.path)
        c.row_factory = sqlite3.Row
        c.execute('PRAGMA foreign_keys = ON'); return c

    def init_db(self):
        with self._conn() as c:
            c.executescript('''
                CREATE TABLE IF NOT EXISTS devices(
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    name TEXT, ip_address TEXT UNIQUE,
                    type TEXT, community TEXT DEFAULT 'public',
                    snmp_enabled INTEGER DEFAULT 0,
                    status TEXT DEFAULT 'unknown');
                CREATE TABLE IF NOT EXISTS metrics(
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    device_id INTEGER REFERENCES devices(id),
                    metric_name TEXT, value REAL,
                    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP);
                CREATE INDEX IF NOT EXISTS idx_m
                    ON metrics(device_id,metric_name,timestamp);
                CREATE TABLE IF NOT EXISTS alerts(
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    device_id INTEGER REFERENCES devices(id),
                    message TEXT, severity TEXT,
                    timestamp DATETIME DEFAULT CURRENT_TIMESTAMP);
                CREATE TABLE IF NOT EXISTS thresholds(
                    id INTEGER PRIMARY KEY AUTOINCREMENT,
                    device_id INTEGER, metric_name TEXT,
                    warning_value REAL, critical_value REAL,
                    UNIQUE(device_id,metric_name));''')
            for d in DEVICES:
                c.execute('INSERT OR IGNORE INTO devices
                    (name,ip_address,type,snmp_enabled)
                    VALUES(?,?,?,?)',
                    (d['name'],d['ip'],d['type'],d.get('snmp',0)))

    def save_metric(self, did, name, val):
        with self._conn() as c:
            c.execute('INSERT INTO metrics(device_id,metric_name,value)
                VALUES(?,?,?)', (did,name,val))

    def get_latest_metrics(self, limit=60):
        with self._conn() as c:
            return [dict(r) for r in c.execute(
                'SELECT * FROM metrics ORDER BY timestamp DESC LIMIT ?',
```

```

        (limit,)).fetchall()]

def check_thresholds(self):
    with self._conn() as c:
        rows = c.execute(
            'SELECT * FROM metrics ORDER BY timestamp DESC LIMIT 50'
        ).fetchall()
    for r in rows:
        th = THRESHOLDS.get(r['metric_name'])
        if not th: continue
        if r['value'] >= th['critical']:
            self.save_alert(r['device_id'],
                f"{r['metric_name']}={r['value']:.1f}", 'CRITICAL')
        elif r['value'] >= th['warning']:
            self.save_alert(r['device_id'],
                f"{r['metric_name']}={r['value']:.1f}", 'WARNING')

def save_alert(self, did, msg, sev):
    with self._conn() as c:
        n = c.execute(
            "SELECT COUNT(*) FROM alerts WHERE device_id=?
            AND message=? AND timestamp>datetime('now','-5 minutes')",
            (did,msg)).fetchone()[0]
        if n==0: c.execute(
            'INSERT INTO alerts(device_id,message,severity)
            VALUES(?,?,?)',(did,msg,sev))

def cleanup_old_data(self, hours=24):
    with self._conn() as c:
        c.execute("DELETE FROM metrics WHERE timestamp<
            datetime('now',?||' hours')", (f'-{hours}',))
        c.execute('VACUUM')

```

Додаток В

Лістинг app.py — Flask веб-сервер і REST API

```
import logging
from flask import Flask, jsonify, render_template, send_file, request
from database import Database
from pdf_report import generate_report
import collector # starts APScheduler on import

logging.basicConfig(level=logging.INFO,
                    format='%(asctime)s %(levelname)s: %(message)s')
app = Flask(__name__)
db = Database()

@app.route('/')
def dashboard(): return render_template('dashboard.html')
@app.route('/devices')
def devices(): return render_template('devices.html')
@app.route('/alerts')
def alerts(): return render_template('alerts.html')
@app.route('/about')
def about(): return render_template('about.html')

@app.route('/api/metrics')
def api_metrics():
    limit = request.args.get('limit', 60, type=int)
    return jsonify(db.get_latest_metrics(limit))

@app.route('/api/devices')
def api_devices():
    return jsonify(db.get_all_devices())

@app.route('/api/alerts')
def api_alerts():
    return jsonify(db.get_alerts())

@app.route('/api/clear-alerts', methods=['POST'])
def clear_alerts():
    n = db.clear_alerts()
    return jsonify({'status': 'ok', 'deleted': n})

@app.route('/report/pdf')
def report_pdf():
    buf = generate_report(db.get_latest_metrics(1000),
                        db.get_alerts())
    return send_file(buf, as_attachment=True,
                    download_name='monitoring_report.pdf',
                    mimetype='application/pdf')

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, threaded=True)
```

Додаток Г

Файл config.py — Конфігурація системи

```
DEVICES = [
    {'name': 'Local PC', 'ip': '127.0.0.1', 'type': 'pc', 'snmp': 0},
    {'name': 'Router', 'ip': '192.168.0.1', 'type': 'router', 'snmp': 1},
    {'name': 'NUBiP Srv', 'ip': 'elearn.nubip.edu.ua', 'type': 'server', 'snmp': 0},
    {'name': 'Mobile', 'ip': '192.168.0.101', 'type': 'phone', 'snmp': 0},
]
THRESHOLDS = {
    'cpu': {'warning': 80.0, 'critical': 90.0},
    'ram': {'warning': 85.0, 'critical': 92.0},
    'temp': {'warning': 70.0, 'critical': 80.0},
    'rtt': {'warning': 100.0, 'critical': 300.0},
}
OID_MAP = {
    '1.3.6.1.2.1.25.3.3.1.2.1': 'cpu',
    '1.3.6.1.4.1.2021.4.6.0': 'ram_free',
    '1.3.6.1.2.1.2.2.1.10.1': 'net_in',
    '1.3.6.1.2.1.2.2.1.16.1': 'net_out',
}
COLLECT_INTERVAL = 5
CLEANUP_INTERVAL = 3600
DATA_RETENTION_H = 24
ALERT_DEDUP_MIN = 5
```