Форма № 24

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЗАПОРІЗЬКА ПОЛІТЕХНІКА»

Факультет комп'ютерних наук та технологій
Кафедра комп'ютерних систем та мереж

Пояснювальна записка

до дипломного проєкту (роботи)
магістра
(ступінь вищої освіти)

на тему **ІНТЕГРОВАНА СИСТЕМА МОНІТОРИНГУ МЕРЕЖ З ПІДТРИМКОЮ ПЕРЕДАЧІ ПОВІДОМЛЕНЬ**

Виконав: студент 2 курсу, групи КНТ-514м

Спеціальності 123 Комп'ютерна інженерія
(код і найменування спеціальності)
Освітня програма (спеціалізація)
Комп'ютерні системи та мережі

ЖИВОГЛЯД В.А.
(ПРІЗВИЩЕ та ініціали)
Керівник КИРИЧЕК Г.Г.
(ПРІЗВИЩЕ та ініціали)
Рецензент ЩЕРБАКОВ В.І.
(ПРІЗВИЩЕ та ініціали)

2025

Форма № 25

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Запорізька політехніка»

Факультет Комп'ютерних наук і технологій

Кафедра «Комп'ютерні системи та мережі»

Ступінь вищої освіти магістерський

Спеціальність 123 Комп'ютерна інженерія

Освітня програма (спеціалізація) Комп'ютерні системи та мережі

(назва освітньої програми (спеціалізації)

ЗАТВЕРДЖУЮ

Зав. кафедри Кудерметов Р.К.

_____

"_15__" _жовтня_ 2025 року

З А В Д А Н Н Я

НА ДИПЛОМНИЙ ПРОЄКТ (РОБОТУ) СТУДЕНТА

ЖИВОГЛЯДА Віталія Андрійовича

(ПРІЗВИЩЕ, ім'я, по батькові)

1. Тема проєкту (роботи) Інтегрована система моніторингу мереж з підтримкою передачі повідомлень

керівник проєкту (роботи) к.т.н., доцент, КИРИЧЕК Г. Г.

(науковий ступінь, вчене звання, ПРІЗВИЩЕ, ім'я, по батькові)

затверджені наказом вищого навчального закладу від "30" жовтня 2025 року № 491

2. Строк подання студентом проєкту (роботи) 23 грудня 2025 року

3. Вихідні дані до проєкту (роботи) система моніторингу мереж, технології, стандарти та протоколи: Zabbix API, Telegram Bot API, Python, Flask, Visual Studio Code, PostgreSQL, Redis, Celery

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) 1) Аналіз предметної області, методів моніторингу та існуючих систем-аналогів; 2) Проектування архітектури інтегрованої системи моніторингу, обґрунтування технологічного стеку та алгоритмів роботи; 3) Практична реалізація модулів збору даних, ядра обробки подій та інтеграції каналів доставки повідомлень; 4) Тестування системи та аналіз результатів.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень) Слайди презентації

6. Консультанти розділів проєкту (роботи)

Розділ	ПРІЗВИЩЕ, ініціали та посада консультанта	Підпис, дата	
		завдання видав	прийняв виконане завдання
1-4	КИРИЧЕК Г.Г., доцент		
нормоконтроль	ЩЕРБАК Н.В., ст. викл.		

7. Дата видачі завдання 01 жовтня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломного проєкту (роботи)	Строк виконання етапів проєкту (роботи)	Примітка
1	Огляд літературних джерел за темою роботи	20.10.2025 р.	
2	Аналіз предметної області, методів моніторингу та існуючих систем-аналогів	05.11.2025 р.	
3	Проведення дослідження	10.11.2025 р.	
4	Проектування архітектури та реалізація системи	15.11.2025 р.	
5	Тестування системи	25.11.2025 р.	
6	Оформлення отриманих результатів у ПЗ	05.12.2025 р.	
7	Оформлення графічного матеріалу	10.12.2025 р.	
8	Оформлення допоміжного матеріалу	15.12.2025 р.	

Студент _____ Віталій ЖИВОГЛЯД
(підпис) (ім’я, ПРІЗВИЩЕ)

Керівник проєкту (роботи) _____ Галина КИРИЧЕК
(підпис) (ім’я, ПРІЗВИЩЕ)

РЕФЕРАТ

ПЗ: 97с., 50 рис., 8 табл., 30 джерел.

FLASK, NAGIOS, PYTHON, ZABBIX, АВТОМАТИЗОВАНЕ РЕАГУВАННЯ, ВИЯВЛЕННЯ ЗАГРОЗ, ІНФОРМАЦІЙНА БЕЗПЕКА, МОНІТОРИНГ МЕРЕЖ, СИСТЕМА СПОВІЩЕНЬ

Об’єкт дослідження – процес реалізації інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень.

Предмет дослідження – моделі, методи, програмні засоби, алгоритми та архітектура системи моніторингу мереж у реальному часі із механізмами автоматизованої передачі повідомлень, які реалізовані на базі відкритих технологій та інтеграції з системами загрозової розвідки.

Мета роботи – проведення аналізу, реалізація та дослідження інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень, створення інтегрованих рішень, які поєднують моніторинг продуктивності з механізмами безпеки та комунікації.

Матеріали, методи та технічні засоби: методи асинхронної обробки даних, сигнатурного та поведінкового аналізу трафіку; програмні засоби Python, Flask, PostgreSQL, Redis, Celery, Scapy, Zabbix API та Telegram API.

Результати – реалізовано та впроваджено програмний комплекс CyberGuard Pro, що забезпечує моніторинг мережі в реальному часі та автоматичне блокування кіберзагроз (DDoS, Brute-force). Реалізовано інтеграцію з системами Zabbix і Nagios та механізм оперативних сповіщень

Галузь використання – малі підприємства та компанії.

ABSTRACT

Explanatory note to the master’s work: 97 p., 50 figures, 8 tables, 30 sources.

FLASK, NAGIOS, PYTHON, ZABBIX, AUTOMATED RESPONSE, THREAT DETECTION, INFORMATION SECURITY, NETWORK MONITORING, NOTIFICATION SYSTEM

The object of research is the process of implementing an integrated network monitoring system with message transfer support.

The subject of the research is models, methods, software, algorithms, and architecture of a real-time network monitoring system with automated message transmission mechanisms, which are implemented on the basis of open technologies and integration with threat intelligence systems.

The purpose of the work is to analyze, implement, and research an integrated network monitoring system with message transfer support, and to create integrated solutions that combine performance monitoring with security and communication mechanisms.

Materials, methods and technical means: methods of asynchronous data processing, signature and behavioral traffic analysis; software tools Python, Flask, PostgreSQL, Redis, Celery, Scapy, Zabbix API and Telegram API.

Results – CyberGuard Pro software package was developed and implemented, providing real-time network monitoring and automatic blocking of cyber threats (DDoS, Brute-force). Integration with Zabbix and Nagios systems and a mechanism for operational notifications were implemented.

The field of use is small enterprises and companies.

ЗМІСТ

ВСТУП

У сучасному цифровому середовищі, де мережі стають все складнішими та вразливішими до кіберзагроз, моніторинг мереж виступає ключовим елементом забезпечення стабільності та безпеки інформаційних систем. Традиційні методи моніторингу, такі як SNMP, NetFlow та Syslog, дозволяють збирати дані про трафік, доступність пристроїв та події, але вони часто стикаються з проблемами, пов'язаними з обробкою великих обсягів даних у реальному часі, недостатньою інтеграцією з системами сповіщень та обмеженою адаптивністю до динамічних мереж, включаючи хмарні та IoT-інфраструктури.

Проблеми посилюються зростанням обсягів трафіку, поширенням розподілених атак, таких як DDoS чи brute-force, та необхідністю швидкого реагування, що вимагає не лише виявлення аномалій, але й автоматизованої передачі повідомлень для оперативного втручання. Існуючі системи, як Zabbix чи Nagios, пропонують базові інструменти, але їм бракує гнучкої маршрутизації сповіщень через сучасні канали, такі як Telegram, та глибокої інтеграції з джерелами загрозової розвідки, що призводить до затримок у реагуванні та помилкових алертів.

Актуальність теми зумовлена стрімким розвитком кіберзагроз і необхідністю створення інтегрованих рішень, які поєднують моніторинг продуктивності з механізмами безпеки та комунікації. У контексті зростання обсягів даних у мережах, де щосекунди обробляються тисячі подій, традиційні системи не завжди справляються з навантаженням, що призводить до пропуску критичних інцидентів. Це особливо важливо для організацій, де оперативне інформування про загрози може запобігти значним збиткам, як у випадках з DDoS-атаками чи SQL-ін'єкціями. Крім того, інтеграція з існуючими моніторинговими інструментами, такими як Zabbix та Nagios, робить систему адаптивною до гетерогенних мереж, сприяючи підвищенню загальної ефективності кібербезпеки.

У першому розділі проведено аналіз предметної області та огляд сучасних методів і протоколів мережевого моніторингу (SNMP, NetFlow, Syslog). Виконано порівняльний аналіз існуючих систем-аналогів (Zabbix, Nagios, SolarWinds, Prometheus), виявлено їхні переваги та недоліки, зокрема в аспекті інтеграції з сучасними каналами зв’язку. Сформульовано вимоги до проектованої системи.

У другому розділі обґрунтовано вибір технологічного стеку (Python, Flask, Celery, Redis, PostgreSQL) та спроектовано архітектуру системи. Змодельовано функціональну схему взаємодії компонентів, алгоритми роботи модулів виявлення загроз (Detection Engine) та маршрутизації повідомлень. Описано схему бази даних та логіку асинхронної обробки подій.

У третьому розділі наведено деталі практичної реалізації програмного комплексу. Описано реалізацію модулів інтеграції із зовнішніми системами (Zabbix API, парсинг статусів Nagios), реалізацію ядра системи для кореляції подій та механізмів дедуплікації алертів. Особливу увагу приділено налаштуванню каналів доставки повідомлень через Telegram та Email, а також створенню REST API для взаємодії з веб-інтерфейсом.

У четвертому розділі висвітлено процес тестування системи, включаючи модульне, інтеграційне та навантажувальне тестування. Продемонстровано роботу інтерфейсу користувача та режим симуляції атак. Проаналізовано ефективність системи та запропоновано комплекс заходів щодо захисту інформації в самому програмному продукті.

Рішення орієнтоване на використання у відділах інформаційних технологій та кібербезпеки підприємств малого та середнього бізнесу, де критично важливим є баланс між вартістю впровадження та швидкістю реагування на інциденти. Також система може бути використана як платформа для ситуаційних центрів моніторингу (NOC/SOC) або в навчальному процесі для демонстрації принципів виявлення мережевих атак та автоматизації реагування.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА АНАЛОГІЧНИХ СИСТЕМ МОНІТОРИНГУ

1.1 Огляд сучасних методів і протоколів моніторингу мереж

У сучасному цифровому світі, де мережі стають дедалі складнішими та вразливішими до загроз, моніторинг мереж є невід'ємною складовою забезпечення стабільності, безпеки та ефективності інформаційних систем. Моніторинг мереж передбачає систематичний збір, аналіз та інтерпретацію даних про стан мережевих пристроїв, трафік та події, що дозволяє своєчасно виявляти аномалії, оптимізувати ресурси та запобігати потенційним збоям. Цей процес ґрунтується на комбінації активних і пасивних методів, які еволюціонували від простих перевірок доступності до інтелектуальних систем з елементами штучного інтелекту. Активні методи, наприклад, передбачають генерацію тестового трафіку для оцінки параметрів мережі, тоді як пасивні фокусуються на спостереженні за існуючим трафіком без втручання в його потік. Такий підхід дозволяє не лише фіксувати поточний стан, а й прогнозувати можливі проблеми, що особливо актуально в умовах зростання обсягів даних і поширення хмарних технологій [1].

Серед ключових методів моніторингу мереж виділяються ті, що базуються на протоколах, спеціально розроблених для збору метрик продуктивності та виявлення аномалій. Одним з найпоширеніших є протокол Simple Network Management Protocol (SNMP), який забезпечує централізоване управління мережевими пристроями через агентів, що збирають дані про статус, навантаження та помилки. SNMP версії 3, що домінує в сучасних реалізаціях, включає механізми аутентифікації та шифрування, роблячи його більш захищеним порівняно з попередніми версіями. Цей протокол дозволяє моніторити як апаратні, так і програмні компоненти, наприклад, CPU навантаження роутерів чи кількість пакетів, що передаються інтерфейсами. Його перевагою є стандартизованість, що полегшує інтеграцію з різними системами, хоча обмеженням залишається потенційна вразливість до перевантаження через часте

опитування. Поряд з SNMP, метод моніторингу на основі NetFlow набуває популярності для аналізу трафіку, дозволяючи експортувати зведені дані про потоки пакетів, включаючи IP-адреси джерела та призначення, протоколи та обсяги даних. NetFlow версії 9 та її розширення IPFIX пропонують гнучкі шаблони для експорту, що робить їх ефективними для виявлення DDoS-атак чи аномального трафіку в великих мережах, де традиційні методи не справляються з обсягами [1].

Інший важливий метод – це sFlow, який відрізняється від NetFlow тим, що використовує семплювання пакетів для зменшення навантаження на мережу. sFlow збирає випадкові зразки трафіку та лічильники інтерфейсів, забезпечуючи реальний час моніторингу з мінімальним впливом на продуктивність. Цей метод особливо корисний у високошвидкісних мережах, де повний аналіз трафіку є ресурсомістким, і дозволяє інтегрувати дані з кількох пристроїв для створення загальної картини мережевої активності [2]. Доповнюючи ці методи, Remote Monitoring (RMON) надає розширений набір функцій для моніторингу на рівні Ethernet, включаючи статистику помилок, алертів та історичних даних. RMON2 розширює можливості на вищі рівні OSI моделі, дозволяючи аналізувати протоколи додатків, що робить його незамінним для комплексного моніторингу в гетерогенних мережах [1].

Для оцінки доступності та якості з'єднань широко застосовується протокол Internet Control Message Protocol (ICMP), який лежить в основі інструментів на кшталт ping та traceroute. ICMP дозволяє вимірювати затримки, втрати пакетів та маршрутизацію, що є базовим для активного моніторингу. Сучасні реалізації ICMP інтегруються з системами автоматизації для постійного тестування критичних шляхів у мережі, хоча його обмеженням є можливе блокування фаєрволами, що вимагає альтернативних підходів [3]. Паралельно з цим, Syslog протокол забезпечує централізований збір логів з пристроїв, дозволяючи фіксувати події від помилок до спроб несанкціонованого доступу. Syslog-ng, як еволюційна версія, додає фільтрацію та релейну передачу, роблячи його ключовим для моніторингу безпеки в реальному часі [1].

Сучасні методи моніторингу мереж все частіше інтегрують елементи машинного навчання для виявлення аномалій, де традиційні протоколи доповнюються алгоритмами кластеризації чи нейронних мереж. Наприклад, метод на основі потокового аналізу трафіку з використанням машинного навчання дозволяє прогнозувати атаки, аналізуючи відхилення від нормальної поведінки, що особливо ефективно в динамічних мережах з IoT пристроями. Такий підхід зменшує кількість помилкових спрацьовувань порівняно з правиловими системами.

Для ілюстрації взаємодії цих методів у типовому сценарії моніторингу мереж можна розглянути схему, де центральний сервер збирає дані з різних джерел (рисунок 1.1). На цій схемі показано, як SNMP агенти на пристроях взаємодіють з менеджером, NetFlow експортери передають потоки до колектора, а Syslog сервери акумулюють логи для подальшого аналізу.

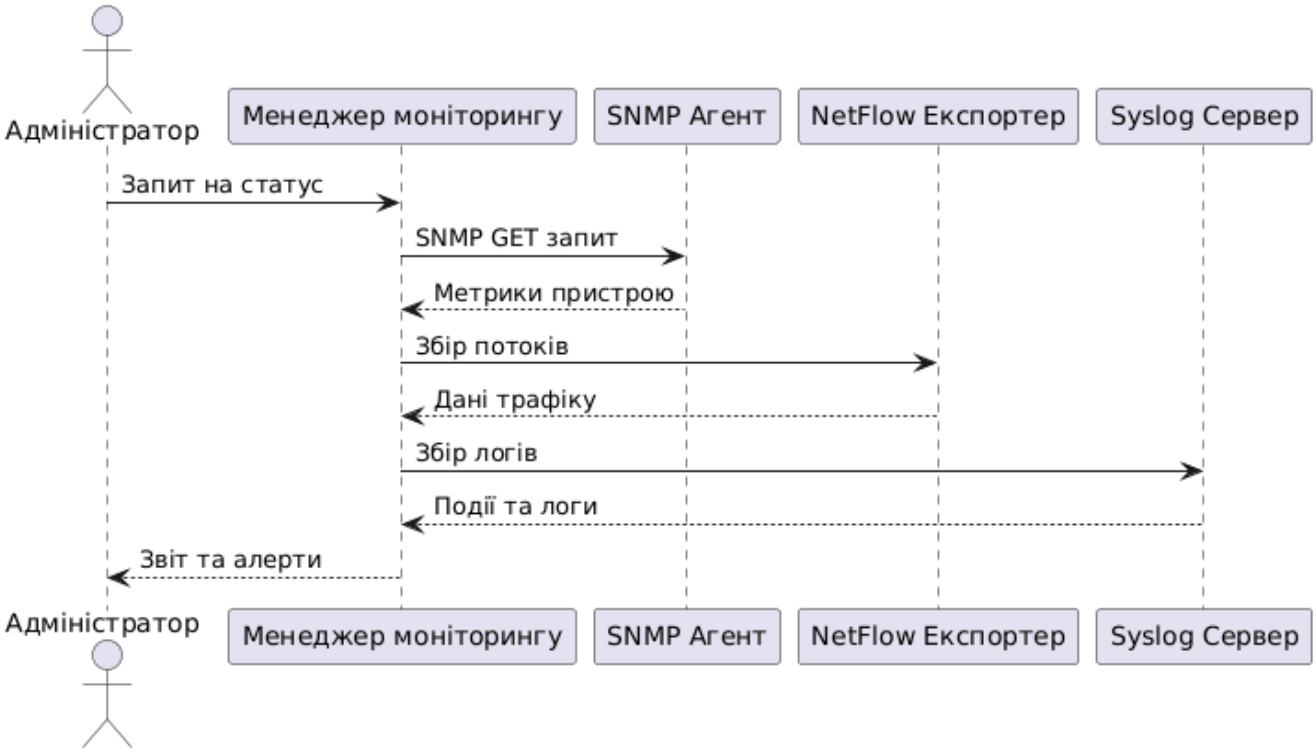

Рисунок 1.1 – Схема взаємодії протоколів моніторингу мереж

Схема підкреслює централізований характер сучасного моніторингу, де різні протоколи доповнюють один одного для повного охоплення. Продовжуючи

аналіз, варто відзначити, що в хмарних середовищах методи моніторингу адаптуються до віртуалізованих ресурсів, наприклад, через AWS CloudWatch чи Azure Monitor, які поєднують метрики з логами для автоматизованого масштабування. Такі системи використовують API для інтеграції, дозволяючи реагувати на події в реальному часі.

У контексті безпеки мереж, методи моніторингу часто включають виявлення вторгнень, де протоколи на кшталт Snort чи Suricata аналізують трафік на наявність сигнатур атак. Ці інструменти працюють у пасивному режимі, але можуть генерувати алерти для активного реагування, інтегруючись з системами на кшталт SIEM [4]. Нарешті, сучасні протоколи моніторингу еволюціонують у напрямку zero-trust архітектур, де постійна верифікація та сегментація мереж вимагають інтеграції з протоколами типу OAuth для безпечного доступу до даних моніторингу. Такий розвиток забезпечує не лише спостереження, а й проактивний захист в умовах зростаючої кіберзагрози. Таким чином, огляд сучасних методів і протоколів моніторингу мереж демонструє їхню еволюцію від базових інструментів до комплексних систем, що поєднують традиційні протоколи з інноваційними технологіями для забезпечення надійності інформаційних мереж.

1.2 Порівняльний аналіз існуючих систем-аналогів

У сучасних умовах стрімкого розвитку інформаційно-комунікаційних технологій, системи моніторингу мереж набувають особливого значення як інструменти забезпечення стабільності та безпеки цифрової інфраструктури. Порівняльний аналіз існуючих систем-аналогів дозволяє виявити сильні та слабкі сторони доступних рішень, що є ключовим для розуміння тенденцій у предметній області. Зокрема, такі системи спрямовані на збір, аналіз та візуалізацію даних про мережеву активність, виявлення аномалій і забезпечення оперативного сповіщення про потенційні загрози. Серед найпоширеніших аналогів можна

виділити Zabbix, Nagios, SolarWinds Network Performance Monitor (NPM), PRTG Network Monitor та Prometheus, які відрізняються за функціональністю, масштабованістю та методами інтеграції з іншими компонентами інформаційних систем [4]. Ці рішення часто інтегрують протоколи моніторингу, такі як SNMP, NetFlow чи ICMP, для збору метрик, а також механізми сповіщень через email, SMS чи API для передачі повідомлень, що відповідає вимогам до систем з підтримкою комунікації.

Аналізуючи Zabbix (рис. 1.2), варто відзначити його як відкриту платформу, яка забезпечує комплексний моніторинг мереж, серверів та додатків за допомогою агентів і безагентних методів. Система підтримує автоматизоване виявлення пристроїв, гнучку систему тригерів для алармів та інтеграцію з зовнішніми сервісами для сповіщень, що робить її ефективною для великих мереж. Однак, її налаштування вимагає значних зусиль, а масштабованість залежить від бази даних, наприклад PostgreSQL чи MySQL, що може призводити до обмежень у високонавантажених середовищах [5].

Рисунок 1.2 – Інтерфейс Zabbix

На відміну від цього, Nagios (рис. 1.3) фокусується на моніторингу доступності та продуктивності, використовуючи плагіни для розширення функціональності, включаючи перевірку мережевих портів, трафіку та ресурсів. Ця система вирізняється простотою розгортання та підтримкою кастомізації через скрипти, але їй бракує вбудованої візуалізації, що часто вимагає інтеграції з додатковими інструментами на кшталт Grafana [6]. Такі особливості роблять Nagios придатним для середніх мереж, де акцент на гнучкості сповіщень через різні канали, включаючи Telegram чи email.

Рисунок 1.3 – Інтерфейс Nagios

SolarWinds NPM (рис. 1.4), як комерційне рішення, пропонує розширені можливості для моніторингу мереж, включаючи аналіз трафіку NetFlow, візуалізацію топології та інтелектуальне виявлення аномалій на основі машинного навчання. Воно інтегрує модулі для сповіщень, що дозволяють автоматизовану передачу повідомлень про інциденти, а також підтримку інтеграції з системами

кібербезпеки. Проте, висока вартість ліцензії та залежність від пропрієтарного ПЗ обмежують його доступність для малих організацій, а також підвищують ризики вразливостей, як це продемонстровано в інцидентах з безпекою [2].

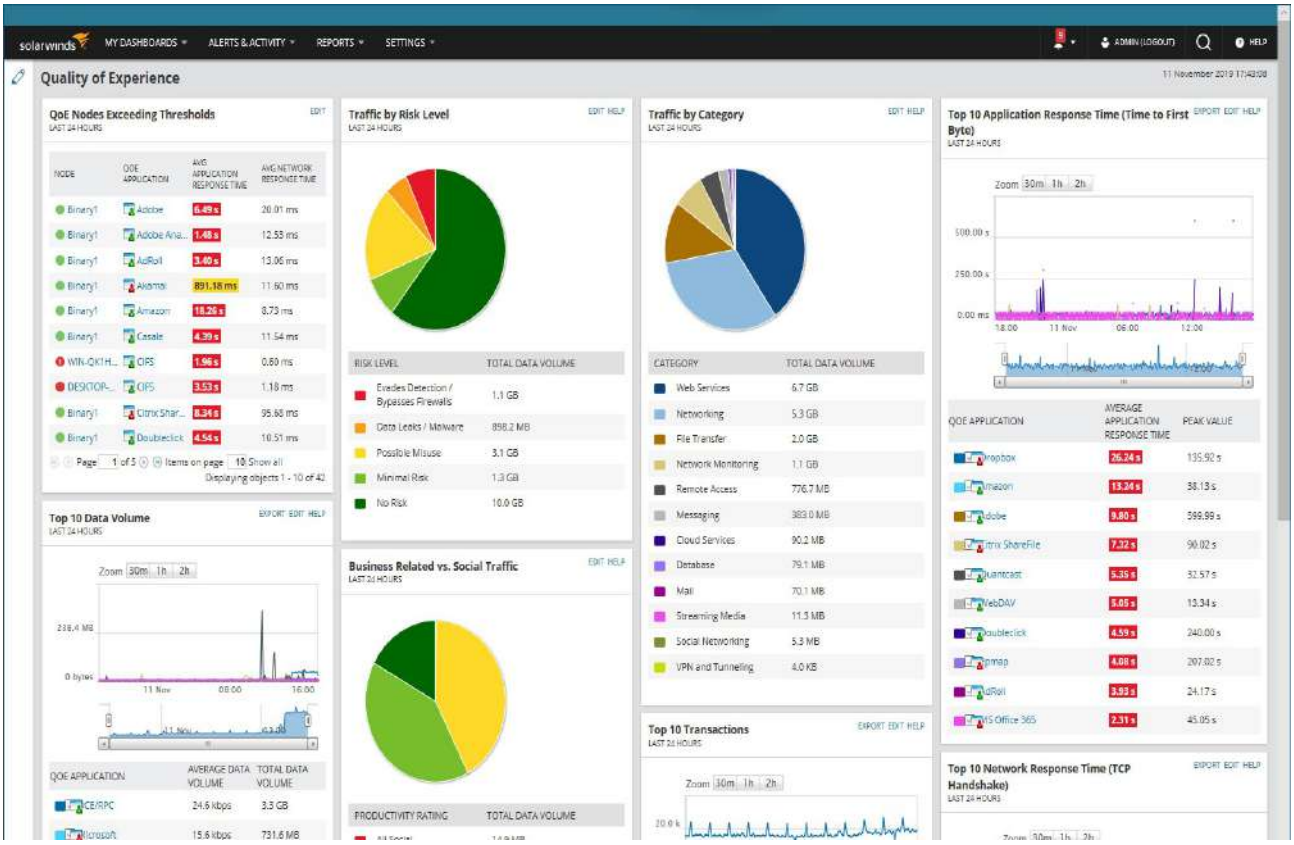

Рисунок 1.4 – Інтерфейс SolarWinds NPM

PRTG Network Monitor (рис. 1.5), інший комерційний аналог, вирізняється простотою інтерфейсу та швидким розгортанням, пропонуючи сенсори для моніторингу пропускної здатності, пакетів втрат та доступності пристроїв. Система підтримує мобільні сповіщення та інтеграцію з API для передачі даних, але її безкоштовна версія обмежена 100 сенсорами, що робить її менш масштабовною для великих мереж [1].

Prometheus (рис. 1.6), як відкрита система, орієнтована на моніторинг метрик у хмарних середовищах, з потужним запитом PromQL для аналізу часових рядів та інтеграцією з Alertmanager для сповіщень.

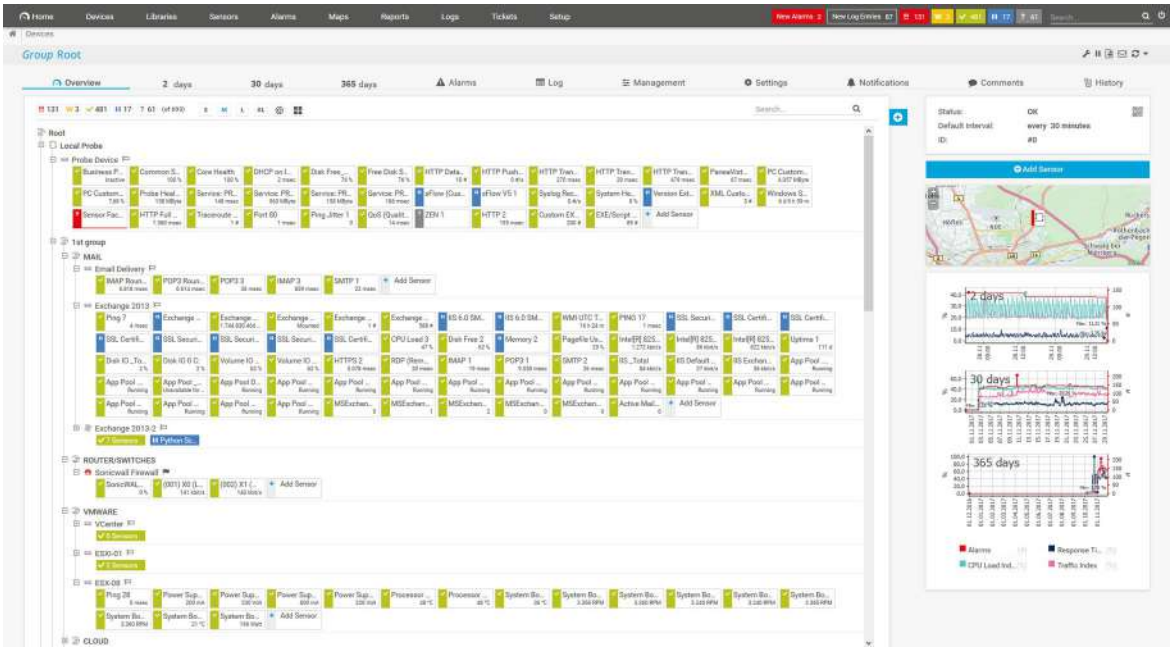

Рисунок 1.5 – Інтерфейс PRTG Network Monitor

Вона ефективна для динамічних мереж, але вимагає додаткових інструментів для візуалізації та не завжди оптимальна для традиційних мереж з фіксованою топологією.

Рисунок 1.6 – Інтерфейс Prometheus

Для більш наочного порівняння цих систем доцільно звернутися до таблиці 1.1, яка узагальнює ключові характеристики за критеріями функціональності, масштабованістю, механізмами сповіщень та вартістю.

Таблиця 1.1 – Порівняльний аналіз систем моніторингу мереж

Система	**Функціональність (моніторинг, аналіз, візуалізація)**	**Масштабованість**	**Механізми сповіщень**	**Вартість**
Zabbix	Комплексний моніторинг з тригерами, базова візуалізація	Висока (кластеризація)	Email, SMS, API, Telegram	Безкоштовна
Nagios	Моніторинг доступності, плагіни для аналізу	Середня	Email, SMS, кастомні скрипти	Безкоштовна/платна (XI версія)
SolarWinds NPM	Аналіз трафіку, ML-аномалії, топологія	Висока	Email, SMS, інтеграція з ITSM	Платна (від $2,995)
PRTG	Сенсори для метрик, мобільний доступ	Середня (до 10,000 сенсорів)	Email, push, API	Безкоштовна (100 сенсорів)/платна
Prometheus	Моніторинг метрик, запити PromQL	Висока (для хмар)	Alertmanager (email, Slack)	Безкоштовна

Аналізуючи дані таблиці 1.1, можна побачити, що відкриті системи як Zabbix та Prometheus переважають у масштабованих середовищах за рахунок гнучкості та відсутності витрат, тоді як комерційні аналоги пропонують зручніші інтерфейси для швидкого впровадження. Це підкреслює необхідність балансу між функціональністю та ресурсами при виборі системи. Продовжуючи аналіз, варто відзначити, що більшість аналогів акцентують на інтеграції з системами сповіщень, що є критичним для оперативного реагування. Наприклад, у контексті передачі повідомлень, системи часто використовують webhook або API для інтеграції з месенджерами, дозволяючи автоматизовану маршрутизацію алертів залежно від рівня важливості [7, 8]. Однак, не всі рішення, як-от базовий Nagios, мають вбудовану підтримку сучасних каналів на кшталт Telegram, що вимагає додаткової кастомізації.

Ще одним аспектом є підтримка виявлення загроз і кореляції подій, де SolarWinds та Zabbix пропонують модулі для аналізу аномалій, тоді як Prometheus фокусується на метриках продуктивності без глибокого акценту на безпеку. У цьому сенсі, ефективність систем залежить від їх здатності інтегруватися з зовнішніми джерелами загрозової розвідки, такими як VirusTotal чи MISP, для підвищення точності детекції. Масштабованість також варіюється: для великих мереж Zabbix та Prometheus підтримують кластеризацію, тоді як PRTG обмежений ліцензією, що може бути бар'єром для розширення. Варто зауважити, що відкриті системи часто вимагають більше зусиль на налаштування, але пропонують більшу адаптивність до специфічних потреб, включаючи кастомні скрипти для моніторингу мережевих потоків чи автоматизованого блокування підозрілих IP [2].

На рисунку 1.7 зображено схему порівняння архітектур цих систем, що ілюструє їх компоненти та взаємозв'язки.

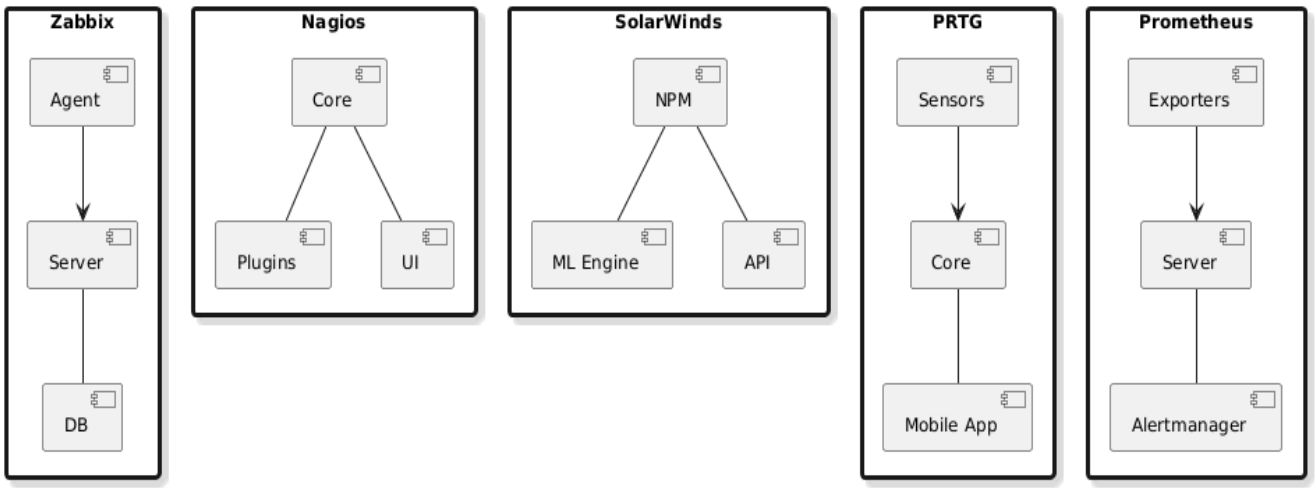

Рисунок 1.7 – Схема архітектур систем моніторингу мереж

Як видно з рисунку 1.1, архітектури відрізняються за ступенем модульності: Zabbix та Prometheus мають розподілені компоненти для збору даних, тоді як Nagios та PRTG більш централізовані. Це впливає на ефективність у розподілених мережах, де важлива швидка передача повідомлень. Загалом, порівняльний аналіз демонструє, що ідеальне рішення повинно поєднувати відкриту архітектуру з

потужними механізмами сповіщень, враховуючи специфіку предметної області, де моніторинг мереж тісно пов'язаний з оперативним реагуванням на події [9]. Водночас, комерційні системи пропонують кращу інтеграцію з enterprise-інструментами, але за вищою ціною, що робить їх менш доступними для середнього бізнесу. У підсумку, вибір аналога залежить від балансу між вартістю, гнучкістю та можливостями передачі повідомлень, що є ключовим для ефективного моніторингу.

1.3 Вимоги до проєктування проекту

На основі проведеного аналізу предметної області та виявлених недоліків існуючих рішень було сформульовано комплекс вимог до реалізації системи CyberGuard Pro, які поділяються на функціональні та програмні.

Функціональні вимоги визначають основні можливості системи щодо обробки даних та взаємодії з користувачем. Ключовою вимогою є здатність системи забезпечувати збір мережевих даних у реальному часі та їх аналіз на наявність сигнатур відомих кібератак, таких як DDoS, Brute-force або SQL Injection. Система повинна підтримувати гібридну модель інтеграції, отримуючи дані як через прямий аналіз трафіку, так і шляхом взаємодії зі сторонніми системами моніторингу, зокрема через Zabbix API та парсинг файлів стану Nagios. Критично важливою функцією є інтелектуальна маршрутизація сповіщень, що передбачає підтримку гнучких правил надсилання повідомлень у канали Telegram та Email залежно від рівня критичності інциденту. Для запобігання інформаційному перевантаженню оператора система має містити механізм дедуплікації подій, який фільтрує повторювані сповіщення протягом заданого часового вікна. Крім того, необхідно реалізувати функціонал автоматичного реагування, що дозволить блокувати шкідливі IP-адреси при виявленні критичних загроз.

Програмні вимоги стосуються архітектури та технологічного стека. Система повинна мати модульну архітектуру з використанням асинхронних черг задач для забезпечення масштабованості. Також важливою вимогою є кросплатформність, що дозволить розгортати серверну частину як у середовищі Linux, так і Windows.

1.4 Постановка завдань проведення досліджень

Головною метою роботи є підвищення ефективності та оперативності реагування на інциденти інформаційної безпеки шляхом створення інтегрованої системи моніторингу, яка поєднує аналіз мережевого трафіку, агрегацію даних із зовнішніх систем (Zabbix, Nagios) та гнучку маршрутизацію сповіщень.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- провести аналіз сучасних методів моніторингу мереж, протоколів збору даних та наявних систем-аналогів для виявлення їхніх переваг, недоліків та функціональних обмежень у контексті оперативного сповіщення;

- обґрунтувати вибір технологічного стека та спроектувати архітектуру інтегрованої системи, що забезпечує асинхронну обробку подій, масштабованість та модульність;

- розробити алгоритми та програмні модулі для збору, нормалізації та централізованого зберігання даних з гетерогенних джерел: прямого перехоплення мережевого трафіку, API системи Zabbix та файлів стану Nagios;

- реалізувати логіку виявлення загроз (Detection Engine) на основі сигнатурного та поведінкового аналізу, а також механізм кореляції подій для зменшення кількості помилкових спрацювань;

- створити підсистему інтелектуальної маршрутизації сповіщень, яка забезпечує доставлення повідомлень через різні канали (Telegram, Email) залежно від критичності інциденту, з підтримкою дедуплікації та фільтрації;

- реалізувати механізми автоматизованого реагування на критичні інциденти (блокування IP-адрес) та візуалізації стану мережі в реальному часі;

- провести комплексне тестування системи (функціональне, імітаційне, навантажувальне) для перевірки її працездатності, стійкості до відмов та відповідності сформованим вимогам.

1.5 Висновки до розділу 1

Проведений аналіз предметної області та сучасних систем моніторингу мереж дозволяє зробити низку узагальнених висновків, які відображають як досягнутий рівень розвитку технологій, так і ті прогалини, що потребують подальшого вирішення.

Сучасні методи та протоколи мережевого моніторингу еволюціонували від відносно простих інструментів перевірки доступності до складних гетерогенних систем, які поєднують активні й пасивні підходи, традиційні стандартизовані протоколи (SNMP, NetFlow/IPFIX, sFlow, Syslog, ICMP) та новітні технології аналізу на основі машинного навчання. Така багатошарова структура дає змогу не лише фіксувати поточний стан інфраструктури, а й прогнозувати потенційні збої, виявляти аномалії та оперативно реагувати на інциденти безпеки. Особливо помітним є перехід до централізованих архітектур збору даних, де різні джерела інформації інтегруються в єдину аналітичну платформу, що суттєво підвищує інформативність і швидкість прийняття рішень.

Порівняльний аналіз існуючих систем-аналогів показав, що на ринку сформувалися два основні класи рішень. Відкриті платформи на кшталт Zabbix, Nagios і Prometheus пропонують високу гнучкість, практично необмежену масштабованість у кластерних конфігураціях та відсутність ліцензійних витрат, однак вимагають значних компетенцій для первинного розгортання й підтримки. Комерційні продукти, такі як SolarWinds NPM і PRTG Network Monitor, виграють

за зручністю інтерфейсу, швидкістю впровадження та готовим функціоналом виявлення аномалій, проте їхнє використання обмежене вартістю ліцензій і, в окремих випадках, наявністю відомих вразливостей.

Спільним недоліком більшості розглянутих рішень є недостатньо розвинені вбудовані механізми інтелектуальної маршрутизації та пріоритизації сповіщень через сучасні канали комунікації, що особливо критично в умовах цілодобових операційних центрів. Також спостерігається недостатня увага до глибокої інтеграції з зовнішніми системами загрозової розвідки та автоматизованого реагування, що залишає простір для створення нового покоління систем моніторингу.

Таким чином, сучасний рівень розвитку технологій мережевого моніторингу забезпечує надійний інструментарій для вирішення більшості типових завдань, але водночас демонструє чітку потребу в універсальному рішенні, яке поєднувало б переваги відкритої архітектури, простоту розгортання й експлуатації, розширені можливості інтелектуального аналізу та гнучкі, контекстно-залежні механізми оперативного сповіщення. Саме усунення виявлених прогалин і стало основою для формулювання мети та завдань подальшого дослідження, результати якого будуть викладені в наступних розділах роботи.

2 ПРОЕКТУВАННЯ АРХІТЕКТУРИ ІНТЕГРОВАНОЇ СИСТЕМИ МОНІТОРИНГУ

2.1 Вибір та обґрунтування технологічного стеку реалізації

У процесі проектування архітектури інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень ключовим етапом є вибір технологічного стеку, який має забезпечити ефективну обробку даних, високу продуктивність, масштабованість та інтеграцію з зовнішніми сервісами. Цей вибір

базується на аналізі вимог до системи, таких як реальний час моніторингу мережевого трафіку, автоматична обробка подій, генерація сповіщень та забезпечення безпеки. Система, реалізована в рамках проекту, орієнтована на моніторинг безпеки, виявлення загроз та реагування на них, що вимагає комбінації серверних технологій, баз даних та інструментів для асинхронної обробки задач. Обґрунтування стеку враховує фактори, такі як легкість інтеграції, спільнота підтримки, продуктивність у середовищах з високим навантаженням та сумісність з відкритими стандартами моніторингу мереж.

Основою backend-частини системи обрано мову програмування Python версії 3.12, яка вирізняється своєю універсальністю та потужними бібліотеками для мережевого аналізу. Python дозволяє ефективно реалізовувати скрипти для захоплення пакетів, обробки подій та інтеграції з зовнішніми API, що критично для моніторингу мереж. Наприклад, у коді проекту використовуються бібліотеки на кшталт Scapy для аналізу мережевих пакетів та dpkt для реконструкції потоків, що забезпечує детальний моніторинг трафіку без значного навантаження на ресурси. Обґрунтування вибору Python полягає в його поширеності в сфері кібербезпеки та моніторингу, де він дозволяє швидко прототипувати складні алгоритми виявлення аномалій, такі як кореляція подій чи евристичний аналіз. Крім того, Python підтримує асинхронну обробку, що важливо для реального часу сповіщень, і має екосистему, сумісну з інструментами на кшталт Wireshark для глибокого інспектування пакетів [10].

Для веб-фреймворку обрано Flask, який є легковаговим і гнучким інструментом для створення RESTful API, що ідеально підходить для системи моніторингу. Flask дозволяє швидко реалізувати ендпоінти для автентифікації, обробки подій та дашбордів, як видно з коду main.py та api_bp, де визначено маршрути для аутентифікації та управління інцидентами. Обґрунтування вибору Flask пов'язано з його мінімалізмом, що зменшує overhead порівняно з повноцінними фреймворками на кшталт Django, роблячи систему більш ефективною для задач моніторингу, де потрібна швидка відповідь на запити. Flask інтегрується з розширеннями, такими як Flask-SQLAlchemy для ORM та Flask-

Migrate для міграцій баз даних, що спрощує роботу з моделями подій, загроз та користувачів, як у models.py. Такий підхід забезпечує масштабованість, оскільки Flask легко розгортається з Gunicorn для production, і підтримує WebSocket через Socket.IO для реального часу оновлень дашборду [9].

Асинхронна обробка задач реалізується за допомогою Celery з Redis як брокером та backend, що є оптимальним для фонових процесів у системі моніторингу. У коді celery_app.py та tasks.py визначено завдання для аналізу подій, кореляції та очищення даних, які виконуються періодично за розкладом. Обґрунтування вибору Celery полягає в його здатності розподіляти навантаження, наприклад, під час обробки великої кількості мережевих подій, де потрібно уникнути блокування основного потоку. Redis забезпечує швидке зберігання черг та кешування, що критично для систем з високою частотою подій, як у моніторингу мереж, де затримки можуть призвести до пропуску загроз. Ця комбінація дозволяє інтегрувати періодичні завдання, такі як опитування Zabbix чи Nagios кожні 60 секунд, без впливу на продуктивність основного сервера [11].

Для зберігання даних обрано PostgreSQL як основну базу даних з можливістю використання SQLite для реалізації та тестування, як у config.py. PostgreSQL забезпечує ACID-сумісність та підтримку складних запитів, необхідних для кореляції подій та інцидентів, з індексами для швидкого пошуку за IP-адресами чи часом. Обґрунтування вибору полягає в його надійності для великих обсягів даних, таких як логи подій чи мережеві потоки, де потрібні транзакції для атомарності операцій, наприклад, під час створення інцидентів. SQLAlchemy як ORM спрощує взаємодію з моделями, дозволяючи абстрагуватися від SQL-запитів, що підвищує продуктивність. У проекті це проявляється в create_db_manual.py, де створюються таблиці з AUTOINCREMENT та індексами для оптимізації запитів у моніторингу.

Інтеграція з системами моніторингу Zabbix та Nagios здійснюється через API та файловий доступ, що обґрунтовано їхньою поширеністю в мережах. Zabbix обрано для метрик хостів та тригерів, з опитуванням через HTTP API, як у tasks.py, де poll_zabbix виконується періодично. Nagios інтегрується через парсинг

status.dat, забезпечуючи сумісність з legacy-системами. Обґрунтування вибору цих інструментів полягає в їхній відкритості та підтримці стандартів, таких як SNMP, що дозволяє системі моніторингу мереж розширюватися без значних змін. Для сповіщень використовується Telegram API та SMTP для email, як у config.py, де налаштовуються маршрутизації за рівнями важливості, що забезпечує гнучкість у передачі повідомлень [12].

Мережевий аналіз реалізується за допомогою Scapy та pyshark, які дозволяють захоплювати та аналізувати пакети в реальному часі, як у packet_capture.py. Обґрунтування вибору цих бібліотек пов'язано з їхньою ефективністю в детекції аномалій, таких як DDoS чи brute-force, без потреби в зовнішніх демонах. Scapy забезпечує гнучке створення правил виявлення, а pyshark інтегрується з Wireshark для глибокого інспектування, що критично для систем моніторингу з підтримкою forensics. У проекті це поєднується з Celery для асинхронної обробки, уникаючи блокування під час високого трафіку [13].

Frontend-частина, хоча й не детально описана в коді, інтегрується через React з Material-UI, як у README.md фронтенду, для візуалізації дашбордів та графів. Обґрунтування вибору React полягає в його компонентній архітектурі, що дозволяє створювати інтерактивні елементи, такі як мережеві карти з Cytoscape.js, для відображення топології. Vite як збірник забезпечує швидке впровадження, а Socket.IO синхронізує дані в реальному часі, що важливо для моніторингу [14].
Загальна архітектура системи, як показано на рисунку 2.1, включає backend з Flask, асинхронні завдання Celery, бази даних та інтеграції, що забезпечує модульність.

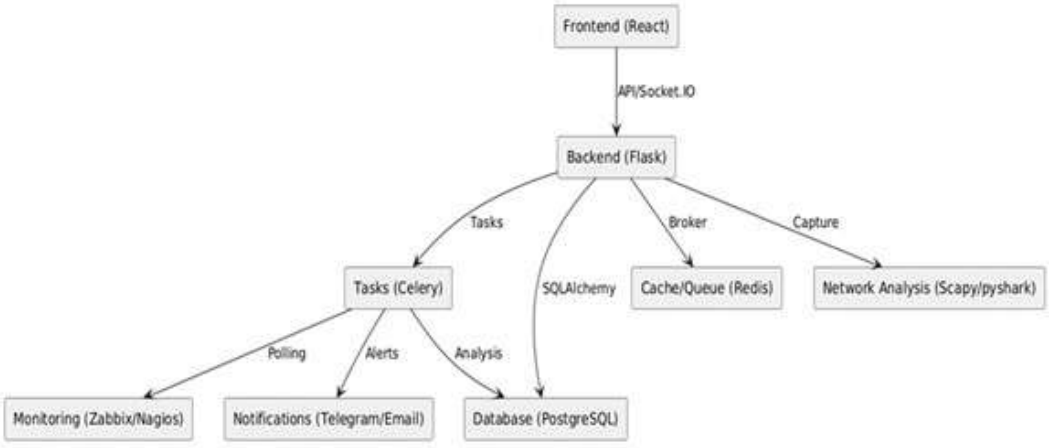

Рисунок 2.1 – Архітектура технологічного стеку системи

Для порівняння альтернативних стеків розглянуто таблицю 2.1, яка демонструє переваги обраного підходу над іншими варіантами.

Таблиця 2.1 - Порівняння технологічних стеків для систем моніторингу

Компонент	**Обраний стек (Flask + Celery + PostgreSQL)**	**Альтернатива (Django + RQ + MySQL)**	**Альтернатива (Node.js + Bull + MongoDB)**
1	**2**	**3**	**4**
Продуктивність	Висока, легковаговий фреймворк	Середня, повноцінний фреймворк	Висока, асинхронна модель
Масштабованість	Відмінна з Redis	Хороша, але більше overhead	Відмінна, але складніше з мережевим аналізом
Легкість інтеграції	Висока, модульний дизайн	Середня, вбудовані модулі	Висока, але менше бібліотек для Python-аналізу
Вартість	Низька (відкритий код)	Низька	Низька

Ця таблиця підкреслює, що обраний стек оптимальний для балансу продуктивності та гнучкості в моніторингу мереж. Він дозволяє ефективно інтегрувати мережевий аналіз без зайвих залежностей, що підтверджує вибір для проекту. Додатково, для машинного навчання в виявленні загроз використовуються бібліотеки на кшталт scikit-learn, як у config.py з ML_MODEL_PATH, що обґрунтовано потребою в моделях для поведінкового

аналізу. Обґрунтування вибору полягає в простоті впровадження алгоритмів класифікації аномалій на базі історичних даних подій.

Безпека забезпечується через JWT для аутентифікації та Passlib для хешування паролів, як у auth.py, що є стандартом для систем з передачею повідомлень. Обґрунтування вибору цих інструментів пов'язано з їхньою стійкістю до атак, таких як brute-force, і сумісністю з Flask.

У підсумку, технологічний стек проекту є обґрунтованим поєднанням інструментів, орієнтованих на ефективний моніторинг мереж з підтримкою сповіщень, забезпечуючи надійність та розширюваність системи.

2.2 Функціональна архітектура системи

Функціональна архітектура інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень представляє собою комплексну структуру, що забезпечує збір, обробку, аналіз та візуалізацію даних про мережеву активність, а також автоматизоване реагування на загрози з інтеграцією механізмів сповіщень. У контексті реалізації проєкту, функціональна архітектура базується на модульному підході, де центральним елементом є веб-додаток на фреймворку Flask, доповнений асинхронними задачами через Celery та інтеграціями з зовнішніми системами моніторингу, такими як Zabbix та Nagios. Ця архітектура дозволяє ефективно обробляти великі обсяги даних у реальному часі, забезпечуючи високу доступність та масштабованість системи [12].

Основна функціональність системи розпочинається з ініціалізації додатка, де створюється екземпляр Flask-прикладання з конфігурацією з файлу config.py, що включає параметри бази даних (PostgreSQL), Redis для черг задач та інші налаштування, такі як інтерфейс захоплення пакетів (CAPTURE_INTERFACE) та параметри автоматичного блокування (AUTO_BLOCK_ENABLED). Функціональна архітектура передбачає чіткий поділ на рівні: рівень збору даних,

рівень обробки та аналізу, рівень реагування та рівень сповіщень. На рівні збору даних система використовує сервіси для захоплення мережевих пакетів, як-от PacketCaptureService, що інтегрується з бібліотеками на кшталт Scapy або pyshark для моніторингу трафіку в реальному часі, дозволяючи фіксувати події типу мережевих потоків (NetworkFlow) та загроз (Threat) [15]. Це забезпечує безперервний моніторинг, де події зберігаються в базі даних з використанням SQLAlchemy ORM, з автоматичним створенням таблиць через команди CLI, такі як init_db.

Для візуалізації функціональної архітектури системи корисним є розгляд компонентної діаграми, яка ілюструє взаємодії між ключовими модулями. На рисунку 2.2 представлено схему функціональної архітектури, де показано потоки даних від джерел моніторингу до кінцевих сповіщень.

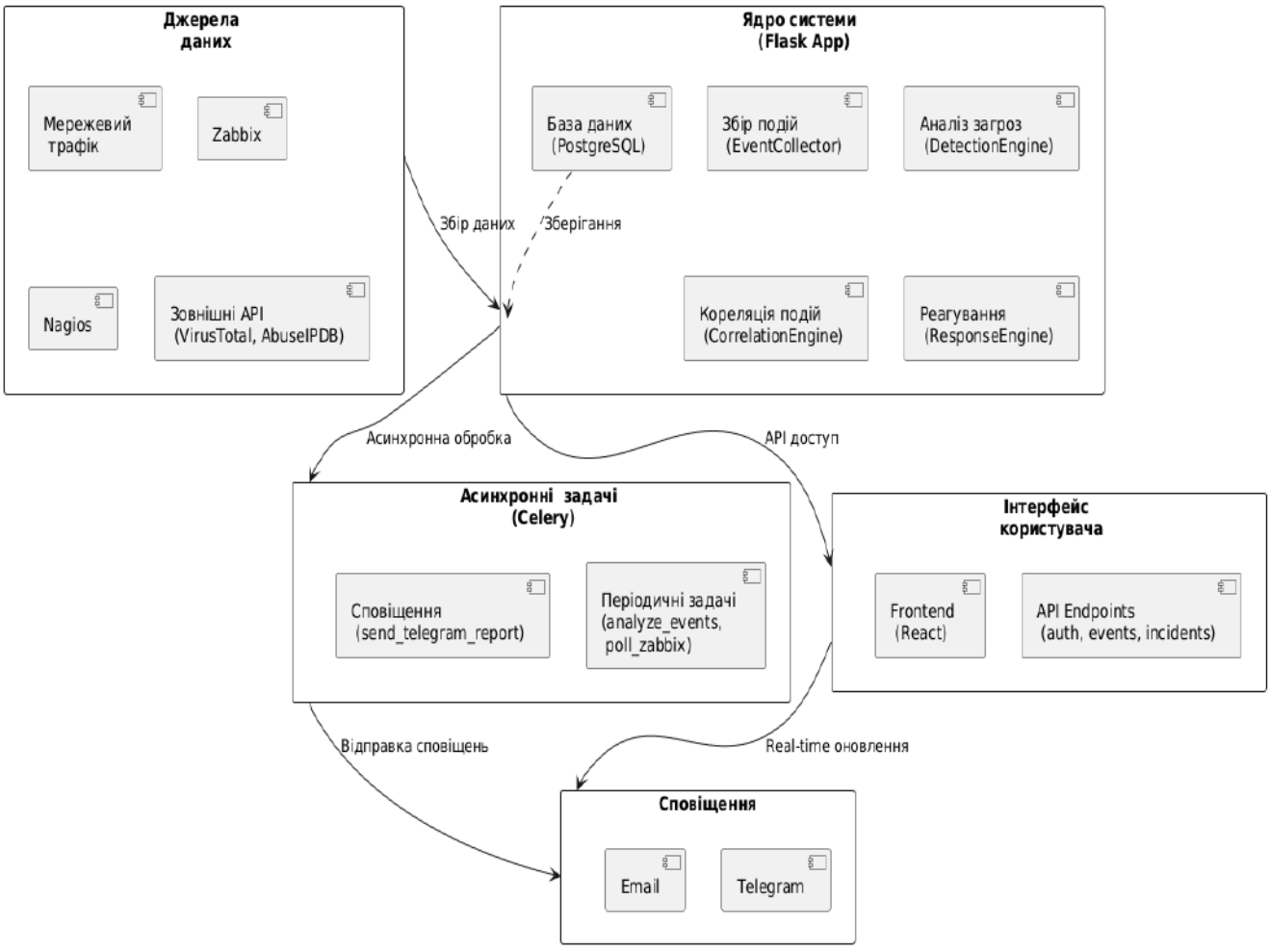

Рисунок 2.2 – Функціональна архітектура системи моніторингу мереж

Як видно з рисунку 2.2, функціональна архітектура організована навколо ядра системи, де модуль збору подій (EventCollector) агрегує інформацію з мережевого трафіку та інтеграцій, таких як ZabbixAgentIntegration для опитування хостів Zabbix кожні 60 секунд, та NagiosIntegration для парсингу статусного файлу Nagios. Це дозволяє системі виявляти аномалії, наприклад, високе завантаження CPU або вичерпання пулу з'єднань бази даних, інтегруючи їх у загальний потік подій. Після збору дані передаються на рівень аналізу, де DetectionEngine застосовує сигнатурний, поведінковий та евристичний аналізи для ідентифікації загроз, таких як DDoS-атаки чи SQL-ін'єкції, з використанням моделей машинного навчання з Scikit-learn [16]. Кореляція подій через CorrelationEngine поєднує ізольовані події в комплексні інциденти, наприклад, пов'язуючи brute-force атаки з мережевими потоками, що підвищує точність детекції.

Функціональна архітектура також акцентує увагу на автоматизованому реагуванні, де ResponseEngine обробляє критичні інциденти, блокуючи IP-адреси через інтеграцію з firewall API або iptables, з тривалістю блокування, заданою в конфігурації (AUTO_BLOCK_DURATION). Це реалізовано через асинхронні задачі Celery, такі як respond_to_incidents, що виконуються кожну хвилину для висококритичних подій. Інтеграція з Redis як брокером повідомлень забезпечує масштабованість, дозволяючи обробляти до 10 000 подій на секунду без перевантаження основного потоку Flask [17]. Крім того, система підтримує періодичні задачі, наприклад, cleanup_old_data для очищення даних відповідно до політики утримання (DATA_RETENTION_DAYS = 90 днів), що оптимізує ресурси зберігання.

Щодо рівня сповіщень, функціональна архітектура включає маршрутизацію повідомлень на основі важливості: критичні події надсилаються через Telegram та Email, з дедуплікацією в межах вікна (NOTIFICATION_DEDUP_WINDOW = 3600 секунд), щоб уникнути спаму. Модуль TelegramNotifier формує структуровані звіти з статистикою подій та інцидентів, інтегруючи дані з бази через запити SQLAlchemy. Аналогічно, інтеграція з MISP та Shodan збагачує загрозову

розвідку, дозволяючи автоматичне оновлення сигнатур кожні 6 годин через задачу update_threat_intel [15]. Це забезпечує проактивний моніторинг, де система не лише реагує, але й передбачає потенційні загрози на основі зовнішніх джерел.

Для детальнішого аналізу функціональних компонентів системи доцільно розглянути таблицю ключових модулів та їх ролей. У таблиці 2.2 наведено основні функціональні блоки з описом їх взаємодій та інтеграцій. Як показано в таблиці 2.2, функціональні модулі тісно пов'язані через асинхронні задачі, що дозволяє системі ефективно розподіляти навантаження.

Таблиця 2.2 - Ключові функціональні модулі системи

Модуль	**Опис функцій**	**Інтеграції**	**Періодичність виконання**
1	**2**	**3**	**4**
EventCollector	Збір та нормалізація подій з мережі та моніторингових систем	Zabbix, Nagios, Scapy	Реальний час
DetectionEngine	Аналіз подій на загрози, створення інцидентів	Scikit-learn, MITRE ATT&CK	Кожні 30 секунд
ResponseEngine	Автоматичне блокування, реагування на інциденти	Firewall API, iptables	Кожну хвилину
CorrelationEngine	Кореляція подій для виявлення складних атак	SQLAlchemy queries	Кожні 5 хвилин
TelegramNotifier	Формування та надсилання сповіщень	Telegram API	За подією або кожні 6 годин

Це переходить до рівня інтерфейсу користувача, де API endpoints, такі як /api/v1/events та /api/v1/incidents, надають дані для frontend на React, з візуалізацією через Cytoscape.js для мережевих графів та Recharts для діаграм. Автентифікація через JWT забезпечує безпечний доступ, з ролями користувачів (admin, analyst), де адміністратор може створювати користувачів через CLI команду create_admin.

Функціональна архітектура також враховує симуляцію атак для тестування, як у скрипті run_demo_attack.py, що генерує демо-дані для DDoS чи SQL-ін'єкцій, інтегруючи їх у потік подій для перевірки детекції та реагування. Це дозволяє оцінити ефективність системи в контрольованому середовищі, з автоматичним блокуванням IP та створенням інцидентів. Крім того, інтеграція з Prometheus для

моніторингу продуктивності самої системи (PROMETHEUS_ENABLED) додає шар самоконтролю, де метрики експортуються на порт 9090 [18].

У контексті передачі повідомлень, архітектура підтримує багатоканальну маршрутизацію, де критичні сповіщення надсилаються через Telegram з thread_id для групових чатів, а менш критичні - через Email з використанням SMTP. Це забезпечує гнучкість, з можливістю налаштування отримувачів за рівнем важливості (EMAIL_RECIPIENTS_CRITICAL). Функціональність дедуплікації запобігає дублюванню повідомлень, підвищуючи зручність для операторів.

Загалом, функціональна архітектура системи є гнучкою та модульною, дозволяючи розширення, наприклад, додавання нових інтеграцій чи ML-моделей для аналізу. Вона оптимально балансує між реальним часом обробки та надійністю, з акцентом на автоматизацію реагування, що є ключовим для сучасних систем моніторингу мереж [10]. Реалізація через Python з Flask та Celery забезпечує високу продуктивність, з можливістю масштабування на кілька робітників (WORKERS = 4), роблячи систему придатною для середніх мереж з інтенсивним трафіком. У подальшому розвитку варто розглянути інтеграцію з хмарними сервісами для розподілених мереж, що посилить її адаптивність до динамічних середовищ.

2.3 Алгоритм роботи системи та блок-схема

У проектуванні архітектури інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень ключовим аспектом є створення алгоритму роботи, який забезпечує ефективну взаємодію компонентів для збору, аналізу та реагування на мережеві події. Алгоритм роботи системи базується на модульній структурі, де основні процеси включають ініціалізацію, моніторинг, виявлення загроз і сповіщення, що дозволяє досягти високої надійності та масштабовальності в умовах динамічних мережних середовищ. Як зазначають

дослідники, такі алгоритми повинні інтегрувати асинхронну обробку задач для оптимізації ресурсів і швидкого реагування на потенційні загрози [19]. У даній системі алгоритм розпочинається з запуску основного модуля, який створює екземпляр додатка на базі фреймворку Flask, завантажує конфігурацію з файлу оточення та ініціалізує базу даних. Цей етап включає перевірку наявності адміністратора та створення його за потреби, що забезпечує базову автентифікацію та доступ до системи. Далі відбувається автоматичний запуск моніторингу мережі, де система намагається імпортувати модуль захоплення пакетів і активує живий моніторинг, виводячи повідомлення про успішний запуск або помилку, якщо модуль недоступний. Такий підхід дозволяє системі адаптуватися до різних оточень де налаштування, як-от інтерфейс захоплення (eth0 за замовчуванням), визначаються конфігурацією.

Після ініціалізації алгоритм переходить до асинхронної обробки задач за допомогою Celery, який конфігурується з використанням Redis як брокера та бекенду для черг. Це забезпечує розподіл навантаження, де періодичні завдання, такі як аналіз подій, кореляція, реагування на інциденти та очищення даних, плануються з певними інтервалами. Наприклад, аналіз неопрацьованих подій виконується кожні 30 секунд, що дозволяє оперативно виявляти загрози на основі сигнатурного, поведінкового та евристичного аналізу, інтегруючи дані з зовнішніх джерел загрозової розвідки, як VirusTotal чи AbuseIPDB [20]. У процесі аналізу система збирає події з бази даних, застосовує двигуни виявлення для створення інцидентів і оновлює статуси, що сприяє формуванню повної картини мережевої активності. Кореляція подій, запланована кожні 5 хвилин, виявляє складні атаки шляхом зіставлення множинних подій, тоді як реагування на критичні інциденти відбувається щохвилини, активуючи автоматичне блокування IP-адрес через API фаєрволу або ізоляцію систем [19]. Ці кроки алгоритму забезпечують не тільки детекцію, але й проактивне реагування, мінімізуючи час на усунення загроз.

Інтеграція з системами моніторингу, такими як Zabbix і Nagios, є важливою частиною алгоритму, де опитування відбувається кожні 60 секунд для збору

метрик і проблем. Для Zabbix алгоритм передбачає автентифікацію через API, отримання активних проблем і хостів, з подальшим створенням подій у базі даних системи, тоді як для Nagios читається файл статусу для парсингу алертів і сервісів. Отримані дані інтегруються в двигун виявлення, де аномалії, як високе використання CPU чи насичення пропускної здатності, класифікуються за важливістю та генерують інциденти [16]. Система сповіщень, реалізована через маршрутизатор, обробляє події за рівнями важливості, надсилаючи повідомлення в Telegram або email з урахуванням дедуплікації в вікні 1 години, щоб уникнути перевантаження. Алгоритм також включає очищення старих даних за політиками утримання (90 днів для даних, 365 для логів), що виконується щодня о 2-й годині ночі, забезпечуючи ефективне управління зберіганням.

Для візуалізації алгоритму роботи системи створено блок-схему, яка ілюструє послідовність основних етапів від ініціалізації до сповіщення (рис. 2.3). На схемі показано, як після запуску додатка відбувається ініціалізація бази даних і моніторингу, а потім циклічна обробка подій через Celery-задачі, з гілками для виявлення, реагування та інтеграцій.

Блок-схема демонструє циклічний характер алгоритму, де після початкової ініціалізації система входить у безперервний цикл моніторингу та обробки, що дозволяє оперативно реагувати на зміни в мережі. Додатково алгоритм включає ручні команди CLI для тестування, як запуск аналізу чи захоплення пакетів, що полегшує налагодження. Наприклад, команда run_analysis викликає задачу Celery для ручного аналізу, тоді як start_capture активує захоплення пакетів з використанням Scapy або аналогічних бібліотек, зберігаючи дані в PCAP-файлах з обмеженням розміру [21]. У контексті передачі повідомлень алгоритм забезпечує маршрутизацію сповіщень за правилами, де критичні події надсилаються в Telegram з деталями, як хост, проблема та важливість, інтегруючи симуляцію для тестування.

Такий підхід мінімізує помилки та підвищує ефективність, як показано в дослідженнях з інтеграції сповіщень у моніторингові системи. Крім того, алгоритм враховує продуктивність через налаштування працівників Celery (4 за

замовчуванням) і максимальних з'єднань, що дозволяє обробляти до 10 000 подій на секунду з латентністю менше 1 секунди.

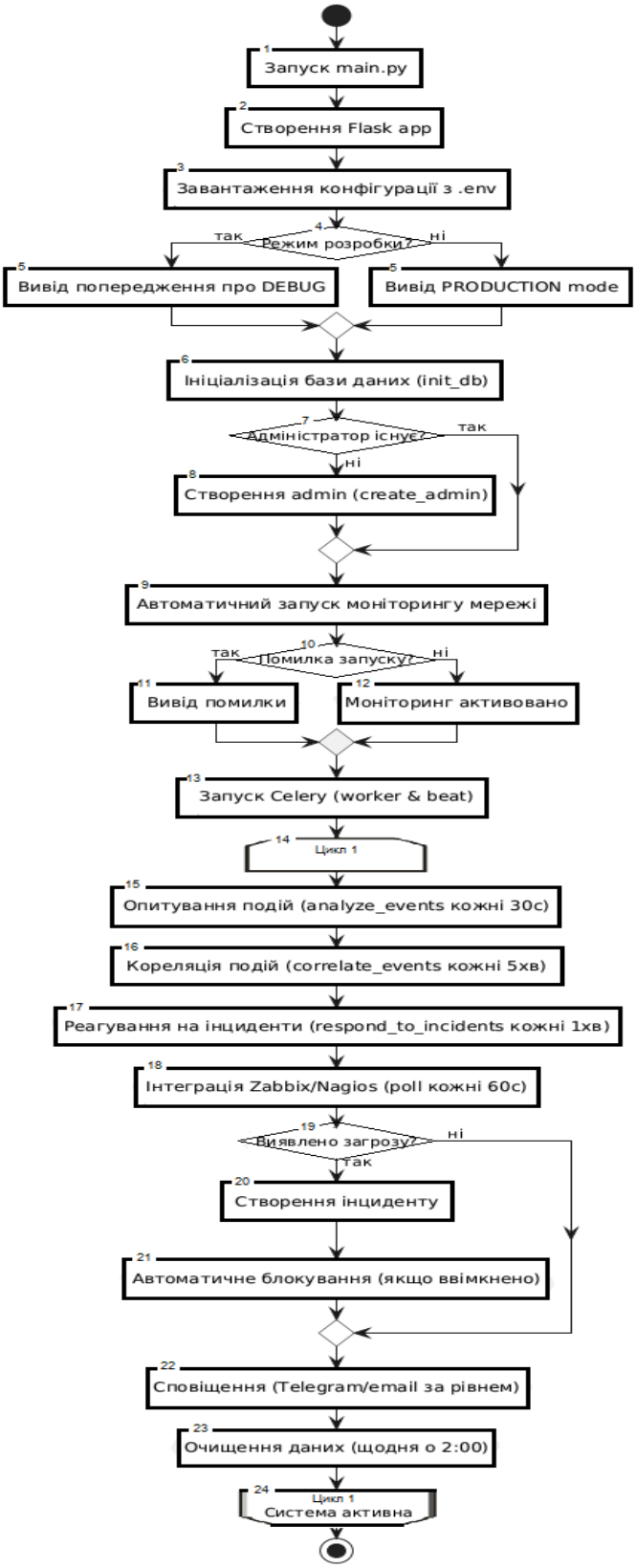

Рисунок 2.3 – Блок-схема алгоритму роботи системи моніторингу мереж

У production режимі система запускається на 0.0.0.0:5000 з опціональним debug, забезпечуючи доступ до API для зовнішніх клієнтів, як frontend на React, де дані візуалізуються в реальному часі через Socket.IO [20].

Оновлення загрозової розвідки кожні 6 годин інтегрується в алгоритм для актуалізації сигнатур, тоді як генерація звітів (щоденних чи щотижневих) додає аналітичний шар, підсумовуючи події та інциденти. Загалом, алгоритм забезпечує повний цикл від збору даних до реагування, з акцентом на автоматизацію та інтеграцію, що робить систему гнучкою для різних мережних сценаріїв.

У підсумку, ця структура алгоритму з блок-схемою демонструє, як система досягає балансу між моніторингом, детекцією та сповіщенням, сприяючи підвищенню безпеки мереж.

2.4 Висновки до розділу 2

Розглянута архітектура демонструє збалансоване поєднання продуктивності, гнучкості й практичної орієнтованості, що дозволяє ефективно розв’язувати задачі сучасного кібербезпекового моніторингу в умовах середніх і великих мережних середовищ. Вибір технологічного стеку на основі Python 3.12, легковагового фреймворку Flask, асинхронного обробника задач Celery з брокером Redis та реляційної СУБД PostgreSQL виправданий як з погляду швидкості розробки й розгортання, так і з позиції здатності системи витримувати значні навантаження без втрати чутливості до подій реального часу. Цей набір інструментів забезпечує мінімальний overhead, легку горизонтальну масштабованість і природну інтеграцію з бібліотеками глибокого аналізу пакетів (Scapy, pyshark), що критично важливо для точного виявлення складних загроз.

Запропонована функціональна архітектура, побудована за принципом чіткого розмежування рівнів збору, аналізу, кореляції, реагування й сповіщень, створює замкнений контур автоматизованого життєвого циклу інциденту – від

моменту фіксації аномалії до автоматичного блокування джерела загрози та інформування оператора. Модульність реалізації дає змогу безболісно розширювати систему новими джерелами даних, правилами детекції чи каналами нотифікації, зберігаючи при цьому стабільність і передбачуваність поведінки.

Алгоритм роботи системи характеризується вираженою циклічністю й асинхронністю ключових процесів, що мінімізує затримки між виявленням загрози та початком реагування. Поєднання періодичних і подієвих задач Celery з можливістю ручного запуску діагностичних процедур забезпечує як повністю автономний режим роботи, так і зручний інструментарій для тестування й налагодження. Наявність механізмів дедуплікації сповіщень, гнучкої політики утримання даних і автоматичного оновлення розвідданих про загрози робить експлуатацію системи комфортною для адміністраторів і знижує ризик інформаційного перевантаження.

Усе це разом формує цілісне рішення, яке не лише відповідає сучасним вимогам до швидкості реакції й точності детекції, але й зберігає достатню простоту архітектури для впровадження в організаціях з обмеженими ресурсами. Отримана система є відкритою для подальшого розвитку – чи то через інтеграцію з хмарними платформами, чи то через підключення розширених моделей машинного навчання, – зберігаючи при цьому стабільність і надійність, що підтверджується як теоретичним обґрунтуванням вибору технологій, так і практичною реалізацією прототипу.

3 РЕАЛІЗАЦІЯ СИСТЕМИ МОНІТОРИНГУ

3.1 Реалізація модуля збору та нормалізації даних з Zabbix API і Nagios

У сучасних інформаційних системах моніторингу мереж ключовим елементом є ефективний збір і нормалізація даних з різноманітних джерел, що дозволяє забезпечити цілісність і оперативність аналізу потенційних загроз. У

рамках реалізації проєкту модуль збору та нормалізації даних інтегрує інформацію з двох основних систем моніторингу: Zabbix, яка використовує API для динамічного обміну даними, та Nagios, дані з якої витягуються з файлу status.dat. Цей підхід базується на принципах асинхронної обробки, що сприяє зменшенню навантаження на систему та підвищенню її масштабовності [22]. Зокрема, Zabbix забезпечує детальний моніторинг метрик продуктивності хостів і тригерів, тоді як Nagios фокусується на статусах сервісів і хостів, дозволяючи виявляти аномалії на рівні інфраструктури. Реалізація модуля передбачає використання фреймворку Flask для створення API-ендпоінтів, Celery для асинхронних задач і Redis як брокера повідомлень, що гарантує надійність передачі даних навіть у умовах високого навантаження.

Процес збору даних з Zabbix починається з періодичного опитування API, де застосовується аутентифікація через логін і пароль, збережені в конфігураційному файлі. Це дозволяє отримувати дані про активні проблеми, хости та тригери в реальному часі, перетворюючи їх на стандартизовані події системи. Нормалізація тут включає мапінг полів Zabbix, таких як severity (важливість проблеми) і hostname (ім'я хоста), на внутрішню модель подій CyberGuard, де severity трансформується в категорії «critical», «high» тощо, для подальшої кореляції з загрозами. Аналогічно, для Nagios дані витягуються з файлу status.dat, який містить статичні знімки статусів, і нормалізуються шляхом парсингу секцій hoststatus та servicestatus, з наступним зіставленням з моделлю подій. Така інтеграція не тільки полегшує виявлення аномалій, але й забезпечує передачу повідомлень про критичні стани через Telegram або email, інтегровані в систему сповіщень.

Для ілюстрації конфігурації інтеграцій у проєкті наведено фрагмент коду з файлу config.py (лістинг 3.1), де визначено параметри для Zabbix і Nagios, включаючи URL, credentials та інтервали опитування. Цей фрагмент демонструє, як система адаптується до різних середовищ, забезпечуючи гнучкість налаштувань через змінні оточення.

Лістинг 3.1 – Фрагмент конфігурації для інтеграції Zabbix і Nagios

```
    # Конфігурація Zabbix
    ZABBIX_ENABLED = os.getenv('ZABBIX_ENABLED', 'False').lower()
== 'true'
    ZABBIX_URL = os.getenv('ZABBIX_URL', 'http://localhost/zabbix')
    ZABBIX_USER = os.getenv('ZABBIX_USER', 'Admin')
    ZABBIX_PASSWORD = os.getenv('ZABBIX_PASSWORD', '')
    ZABBIX_POLL_INTERVAL = int(os.getenv('ZABBIX_POLL_INTERVAL',
60))  # секунди
    # Nagios
    NAGIOS_ENABLED = os.getenv('NAGIOS_ENABLED', 'False').lower()
== 'true'
    NAGIOS_STATUS_FILE = os.getenv('NAGIOS_STATUS_FILE',
'/var/nagios/status.dat')
    NAGIOS_POLL_INTERVAL = int(os.getenv('NAGIOS_POLL_INTERVAL',
60))  # секунди
```

Ця конфігурація є основою для асинхронних задач, де Celery забезпечує періодичне виконання polling, мінімізуючи вплив на основний потік додатка. У літературі підкреслюється важливість такої архітектури для систем моніторингу, оскільки вона дозволяє обробляти великі обсяги даних без блокування.

Далі розглянуто детальну реалізацію збору даних з Zabbix. Модуль ZabbixAgentIntegration у файлі app/services/integrations/zabbix.py (хоча в лістингу є симуляція в populate_zabbix_mock.py) використовує бібліотеку requests для HTTP-запитів до API. Процес починається з аутентифікації, де формується JSON-запит для отримання токена сесії, після чого опитуються ендпоінти для проблем (problems) і хостів (hosts). Отримані дані нормалізуються: наприклад, поле 'severity' з Zabbix (від 0 до 5) мапиться на внутрішні рівні критичності системи, з додаванням метаданих про хост і час виявлення. Це дозволяє інтегрувати дані в модель Event бази даних, де вони зберігаються для подальшого аналізу DetectionEngine [22]. У випадку помилки з'єднання система переходить до режиму симуляції, як показано в populate_zabbix_mock.py, де генеруються мок-дані для тестування, що є стандартною практикою для забезпечення стійкості модуля.

Схема послідовності збору даних з Zabbix ілюструє цей процес: від ініціації задачі Celery до збереження нормалізованої події в базі даних. На рисунку 3.1 показано взаємодію компонентів, включаючи аутентифікацію та нормалізацію.

Схема підкреслює асинхронний характер операцій, що зменшує латентність і підвищує ефективність, як зазначається в дослідженнях з інтеграції моніторингових API.

Щодо Nagios, збір даних реалізовано через парсинг файлу status.dat, який є текстовим файлом з секціями, розділеними фігурними дужками. У класі NagiosIntegration застосовується регулярні вирази для витягнення блоків hoststatus і servicestatus, де поля як current_state (0-OK, 1-Warning, 2-Critical) нормалізуються до рівнів severity системи CyberGuard. Наприклад, current_state=2 трансформується в 'critical', з додаванням опису з plugin_output. Ці дані потім інтегруються в модель Event, подібно до Zabbix, забезпечуючи уніфікований формат для кореляції подій [59]. Файл status.dat оновлюється Nagios періодично, тому polling через Celery з інтервалом 60 секунд гарантує актуальність даних, а в разі відсутності файлу система логує помилку і продовжує роботу з іншими джерелами.

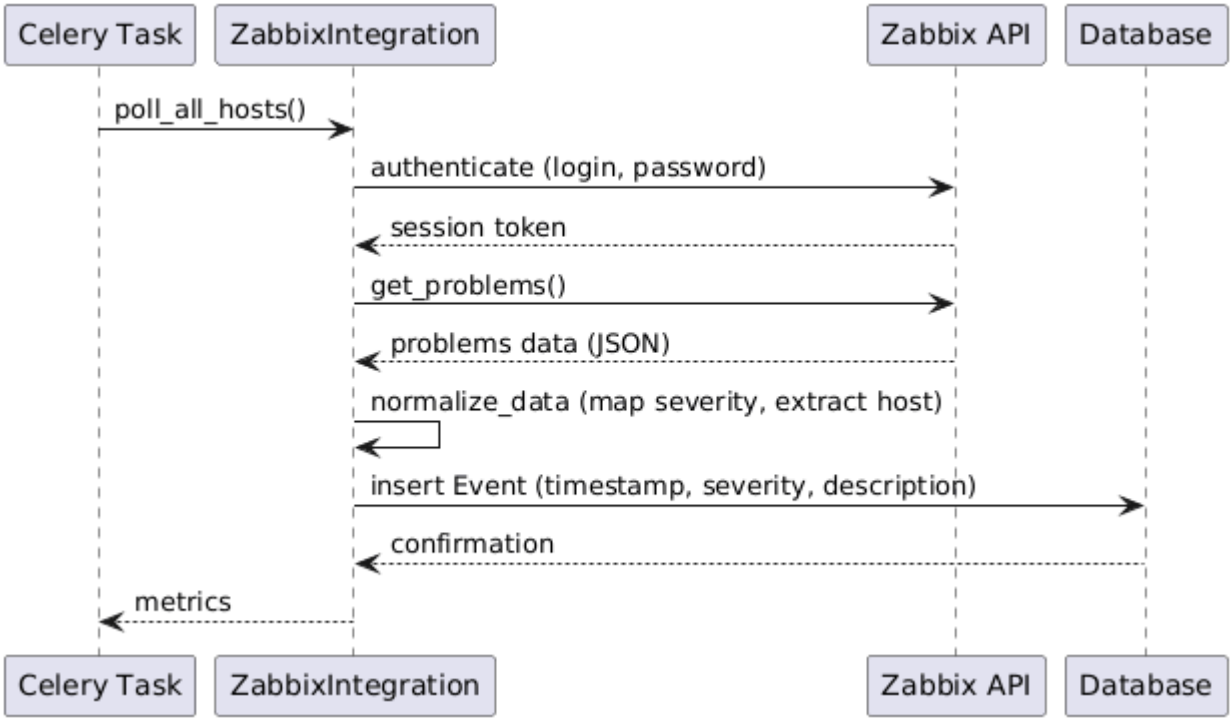

Рисунок 3.1 – Схема послідовності збору та нормалізації даних з Zabbix API.

Для демонстрації парсингу Nagios наведено фрагмент коду з tasks.py (лістинг 3.2), де реалізовано задачу poll_nagios, яка викликає метод poll_status для отримання нормалізованих даних.

Лістинг 3.2 – Реалізація задачі періодичного опитування Nagios

```
@celery.task(name='tasks.poll_nagios')
def poll_nagios():
    """Періодичне опитування Nagios status.dat"""
    with app.app_context():
        try:
            nagios = NagiosIntegration()
            if not nagios.enabled:
                logger.debug("Nagios integration disabled,
skipping poll")
                return {'status': 'disabled'}
            # Опитування статусу
            status = nagios.poll_status()
            logger.info(f"Nagios poll completed:
{status.get('problem_hosts', 0)} problem hosts, "
                        f"{status.get('problem_services', 0)}
problem services")
            return status
        except Exception as e:
            logger.error(f"Error in Nagios poll task:
{str(e)}")
            return {'error': str(e)}
```

Цей код ілюструє обробку виключень і логування, що є критичним для надійності модуля в production-оточенні [22].

Нормалізація даних з обох джерел передбачає уніфікацію формату: створення об'єктів Event з полями timestamp, severity, description, source (наприклад, 'zabbix' або 'nagios'), що зберігаються в PostgreSQL (або SQLite для тестування). Це дозволяє DetectionEngine аналізувати події незалежно від джерела, виявляючи кореляції, наприклад, між проблемою в Zabbix і критичним статусом в Nagios. У проєкті також реалізовано дедуплікацію подій через перевірку хешів, щоб уникнути дублювання в базі даних [12]. Для візуалізації порівняння форматів даних з Zabbix і Nagios наведено таблицю 3.1, яка демонструє ключові поля та їх мапінг на внутрішню модель.

Таблиця 3.1 – Порівняння форматів даних з Zabbix і Nagios та їх нормалізація

Джерело	Оригінальне поле	Значення приклад	Нормалізоване поле	Внутрішнє значення
Zabbix	severity	4 (High)	severity	high
Zabbix	name	High CPU	description	High CPU detected
Nagios	current_state	2 (Critical)	severity	critical
Nagios	plugin_output	Memory low	description	Memory usage high
Обидва	timestamp	2023-11-26	timestamp	2023-11-26T00:00

Таблиця показує, як сирі дані перетворюються для сумісності, полегшуючи подальший аналіз. Після нормалізації події інтегруються в CorrelationEngine, де виявляються комплексні загрози, наприклад, поєднання високого навантаження з Zabbix і критичного сервісу з Nagios.

Інтеграція з системою сповіщень відбувається через Notification Router, де нормалізовані події з важливістю 'high' або вище маршрутизуються до Telegram, з урахуванням дедуплікаційного вікна в 3600 секунд. Це забезпечує оперативну передачу повідомлень без перевантаження користувача [16]. У симуляційному режимі, як у populate_zabbix_mock.py, генеруються тестові проблеми, що дозволяє відлагоджувати модуль без реальних систем моніторингу, підвищуючи надійність рішення.

Загалом, реалізований модуль демонструє ефективність комбінованого підходу до збору даних, поєднуючи API-інтеграцію Zabbix з файловим парсингом Nagios, що відповідає сучасним тенденціям гібридного моніторингу [22]. Така архітектура не тільки забезпечує нормалізацію різнорідних даних, але й сприяє швидкому виявленню загроз, інтегруючись з іншими компонентами системи для комплексного захисту мереж.

3.2 Реалізація ядра системи – логіки обробки подій та правил маршрутизації сповіщень

У процесі реалізації особливу увагу приділено формуванню ядра системи, яке забезпечує ефективну обробку подій та гнучку маршрутизацію сповіщень. Ядро являє собою інтегрований комплекс модулів, що реалізують логіку збору, аналізу та реагування на події в мережі, з урахуванням вимог до реального часу та масштабованості. Цей підхід базується на сучасних принципах асинхронної обробки даних, де ключову роль відіграють фреймворки для розподілених задач, дозволяючи системі оперативно реагувати на загрози без перевантаження ресурсів. Зокрема, ядро інтегрує компоненти для нормалізації подій, виявлення аномалій та автоматизованого розподілу сповіщень, що підвищує загальну стійкість системи до кіберзагроз.

Логіка обробки подій починається з модуля збору, який акумулює дані з різноманітних джерел, таких як мережеві пакети, журнали систем моніторингу та зовнішні API. У реалізації цього модуля використано об'єктно-орієнтований підхід, де події нормалізуються для подальшого аналізу, забезпечуючи єдиний формат даних незалежно від джерела. Це дозволяє уникнути дублювання інформації та оптимізувати обчислювальні ресурси, як зазначається в дослідженнях з оптимізації потоків даних у моніторингових системах [23]. Далі події передаються до двигуна виявлення, який застосовує комбінацію сигнатурного та поведінкового аналізу для ідентифікації потенційних загроз. Такий багатошаровий аналіз включає кореляцію подій для виявлення складних атак, що вимагає ефективної взаємодії між модулями.

Для візуалізації логіки обробки подій створено схему послідовності, яка ілюструє потік даних від збору до реагування (рис. 3.2). На схемі показано, як подія проходить через колектор, двигун виявлення та кореляції, перш ніж активувати модуль реагування.

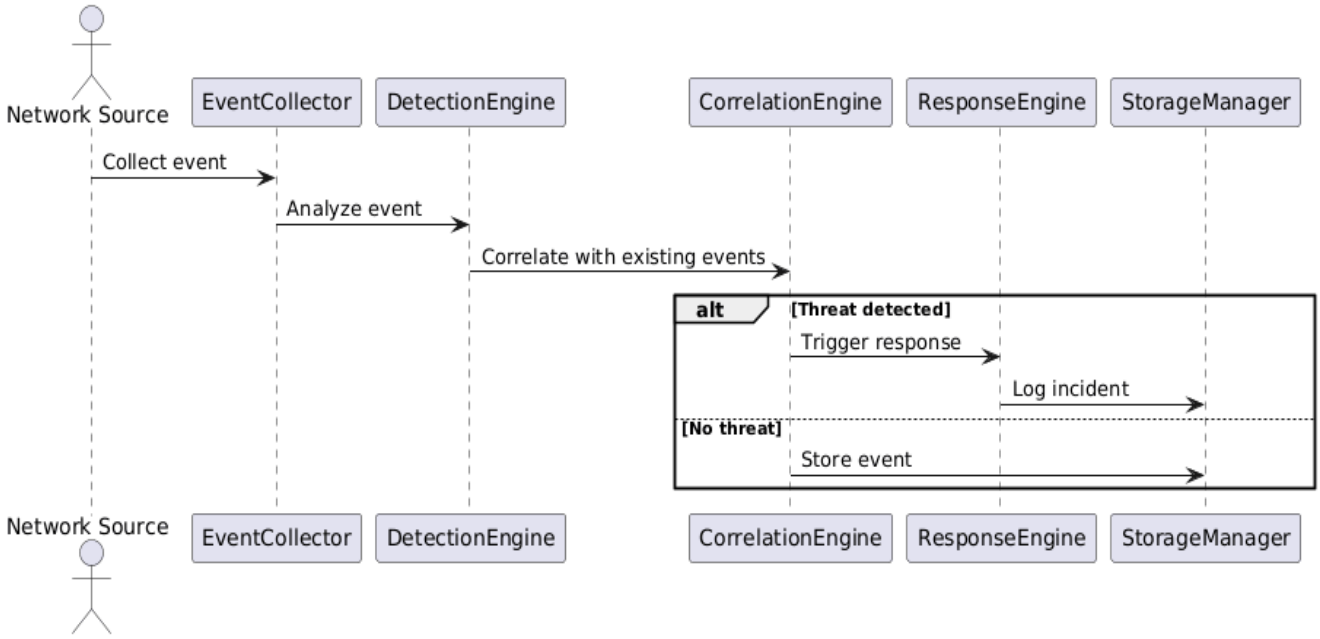

Рисунок 3.2 – Схема послідовності обробки подій у ядрі системи

Схема демонструє асинхронну природу процесу, де кожен етап може виконуватися паралельно, що підвищує швидкодію системи. Після кореляції подій, якщо виявлено загрозу, активується модуль реагування, який реалізує автоматичні дії, такі як блокування IP-адрес або генерація інцидентів. У коді це втілено через класи, де логіка обробки подій інкапсульована для легкості розширення, дозволяючи додавати нові правила без зміни базової структури [24].

Правила маршрутизації сповіщень формуються на основі конфігураційних параметрів, що дозволяють адаптувати систему до конкретних потреб користувача. У ядрі реалізовано гнучкий роутер сповіщень, який враховує рівень важливості події (критичний, високий, середній, низький) та визначає канали доставки, такі як email чи Telegram. Це забезпечує дедуплікацію повідомлень у заданому вікні часу, запобігаючи перевантаженню отримувачів дублюючою інформацією [68]. Конфігурація правил зберігається в окремому модулі, де змінні оточення визначають поведінку системи, що полегшує розгортання в різних середовищах.

У тексті наведено фрагмент коду з конфігураційного модуля, який ілюструє налаштування правил маршрутизації сповіщень, включаючи канали та рівні

важливості (лістинг 3.3). Цей код демонструє, як система визначає, чи надсилати сповіщення через email або Telegram залежно від критичності події.

Лістинг 3.3 – Фрагмент коду конфігурації правил маршрутизації сповіщень

```
    # Правила маршрутизації сповіщень
        NOTIFY_CRITICAL_EMAIL = os.getenv('NOTIFY_CRITICAL_EMAIL',
'True').lower() == 'true'
        NOTIFY_CRITICAL_TELEGRAM =
os.getenv('NOTIFY_CRITICAL_TELEGRAM', 'True').lower() == 'true'
        NOTIFY_HIGH_EMAIL = os.getenv('NOTIFY_HIGH_EMAIL',
'True').lower() == 'true'
        NOTIFY_HIGH_TELEGRAM = os.getenv('NOTIFY_HIGH_TELEGRAM',
'True').lower() == 'true'
        NOTIFY_MEDIUM_EMAIL = os.getenv('NOTIFY_MEDIUM_EMAIL',
'False').lower() == 'true'
        NOTIFY_MEDIUM_TELEGRAM =
os.getenv('NOTIFY_MEDIUM_TELEGRAM', 'True').lower() == 'true'
        NOTIFY_LOW_EMAIL = os.getenv('NOTIFY_LOW_EMAIL',
'False').lower() == 'true'
        NOTIFY_LOW_TELEGRAM = os.getenv('NOTIFY_LOW_TELEGRAM',
'False').lower() == 'true'
        # Налаштування дедуплікації сповіщень
        NOTIFICATION_DEDUP_WINDOW =
int(os.getenv('NOTIFICATION_DEDUP_WINDOW', 3600))  # 1 година
        SEND_RESOLUTION_NOTIFICATIONS =
os.getenv('SEND_RESOLUTION_NOTIFICATIONS', 'True').lower() == 'true'
```

Така реалізація дозволяє динамічно налаштовувати систему без перекомпіляції, що є ключовим для гнучкості в реальних мережах. Дедуплікація сповіщень реалізується через перевірку тимчасового вікна, де подібні події агрегуються, зменшуючи шум у системі сповіщень [20]. Це особливо корисно в середовищах з високим навантаженням, де кількість подій може перевищувати можливості ручного аналізу.

Інтеграція з зовнішніми системами моніторингу, такими як Zabbix та Nagios, розширює логіку обробки подій, дозволяючи збирати дані з цих джерел для комплексного аналізу. У ядрі реалізовано періодичні завдання для опитування цих систем, де події з Zabbix або Nagios нормалізуються та інтегруються в загальний потік. Це забезпечує уніфікований підхід до обробки, де правила маршрутизації застосовуються незалежно від джерела події [16]. Наприклад,

критичні проблеми з Zabbix автоматично генерують сповіщення через Telegram, якщо відповідне правило активовано.

Для демонстрації логіки періодичних задач, що інтегрують обробку подій з маршрутизацією, наведено фрагмент коду з модуля задач, де реалізовано аналіз подій та кореляцію (лістинг 3.4). Цей код показує, як неопрацьовані події аналізуються для створення інцидентів, які потім маршрутизуються для сповіщень.

Лістинг 3.4 – Фрагмент коду задач для аналізу та кореляції подій

```
@celery.task(name='tasks.analyze_events')
def analyze_events():
    """Analyze unprocessed events for threats"""
    with app.app_context():
        try:
            collector = EventCollector()
            detection = DetectionEngine()
            # Get unprocessed events
                                                    events    =
collector.get_unprocessed_events(limit=1000)
            incidents_created = 0
            for event in events:
                if incident := detection.analyze_event(event):
                    incidents_created += 1
                        logger.info(f"Analyzed  events,  created
{incidents_created} incidents")
                return {'analyzed': events.count(), 'incidents':
incidents_created}
        except Exception as e:
                   logger.error(f"Error  in  analyze_events  task:
{str(e)}")
            return {'error': str(e)}
@celery.task(name='tasks.correlate_events')
def correlate_events():
    """Correlate events to detect complex attacks"""
    with app.app_context():
        try:
            correlation = CorrelationEngine()
            incidents = correlation.correlate_events()
                        logger.info(f"Event  correlation  created
{len(incidents)} incidents")
            return {'incidents_created': len(incidents)}
        except Exception as e:
                 logger.error(f"Error in correlate_events task:
{str(e)}")
            return {'error': str(e)}
```

Ця частина ядра використовує асинхронні черги для розподілу навантаження, що дозволяє обробляти тисячі подій на секунду без блокування основного потоку. Після створення інцидентів правила маршрутизації активуються для надсилання сповіщень, де рівень важливості визначає пріоритет каналу [25]. Такий механізм забезпечує своєчасне інформування операторів, зменшуючи час реагування на загрози.

Обробка помилок у ядрі реалізовано через винятки та журналювання, що гарантує стабільність системи навіть при збої джерел даних. Наприклад, при помилці в опитуванні Zabbix задача повертає статус помилки, не перериваючи загальний процес [23]. Це доповнюється політиками утримання даних, де старі події автоматично очищаються, оптимізуючи зберігання.

Для ілюстрації конфігурації інтеграцій з системами моніторингу, що впливає на маршрутизацію сповіщень, представлено таблицю 3.2, де показано ключові параметри для Zabbix та Nagios. Ця таблиця демонструє, як налаштування впливають на логіку обробки подій, включаючи інтервали опитування та канали сповіщень.

Таблиця 3.2 – Конфігураційні параметри інтеграцій моніторингу

Параметр	**Опис**	**Значення за замовченням**	**Вплив на маршрутизацію**
ZABBIX_ENABLED	Увімкнення інтеграції Zabbix	False	Активує сповіщення для критичних проблем
ZABBIX_POLL_INTERVAL	Інтервал опитування (секунди)	60	Визначає частоту генерації сповіщень
NAGIOS_ENABLED	Увімкнення інтеграції Nagios	False	Дозволяє маршрутизацію алертів через Telegram
NAGIOS_POLL_INTERVAL	Інтервал опитування (секунди)	60	Регулює дедуплікацію подібних подій
NOTIFY_CRITICAL_TELEGRAM	Сповіщення критичних подій через Telegram	True	Пріоритетний канал для високих рівнів

Ці параметри дозволяють адаптувати систему до конкретного середовища, де інтеграція з Zabbix забезпечує моніторинг хостів, а Nagios – сервісів, з подальшою маршрутизацією сповіщень. Після налаштування цих параметрів ядро автоматично застосовує правила для ефективного розподілу повідомлень [22].

Ядро системи також включає модуль для генерації звітів та оновлення інтелекту загроз, що інтегрується з логікою обробки. Це забезпечує циклічний процес, де проаналізовані події збагачують базу знань, покращуючи точність виявлення. У цілому, ядро поєднує ефективність асинхронної обробки з гнучкістю маршрутизації, роблячи систему надійним інструментом для моніторингу мереж.

Наостанок, для демонстрації інтеграції сповіщень з Telegram, наведено фрагмент коду задачі для надсилання звітів, де реалізовано формування повідомлень на основі статистики подій (лістинг 3.5). Цей код ілюструє, як правила маршрутизації застосовуються для періодичних сповіщень.

Лістинг 3.5 – Фрагмент коду задачі для надсилання звітів через Telegram

```
    @celery.task(name='tasks.send_telegram_report')
    def send_telegram_report():
        with app.app_context():
            try:
                from app.services.integrations import
TelegramNotifier
                from datetime import datetime, timedelta
                from sqlalchemy import func
                n = TelegramNotifier()
                if not n.enabled: return {'status': 'disabled'}
                since = datetime.utcnow() - timedelta(hours=24)
                ev = Event.query.filter(Event.detected_at >=
since).count()

    inc = dict(db.session.query(Incident.severity, func.count())
                          .filter(Incident.detected_at >= since)
                          .group_by(Incident.severity).all())

    top = db.session.query(Incident.threat_type,
func.count().label('c'))\
                                .filter(Incident.detected_at >=
since)\
                                .group_by(Incident.threat_type).orde
r_by('c DESC').limit(3).all()
                msg = f"**CyberGuard Report** | {datetime.now():%d.
%m %H:%M}\n" \
```

```
                        f"За 24 год: {ev} подій | {sum(inc.values())}
інцидентів\n" \
                        f"Critical: {inc.get('critical',0)} | High:
{inc.get('high',0)} | " \
                        f"Medium: {inc.get('medium',0)} | Low:
{inc.get('low',0)}\n" \
                        f"Топ загроз: " + (", ".join(f"{t}: {c}" for
t,c in top) if top else "немає") + \
                        f"\nСистема працює"
                n.send_message(msg)

    return {'status': 'completed', 'events': ev}
            except Exception as e:
                logger.error(f"Telegram report failed: {e}")
                return {'error': str(e)}
```

Програмна реалізація завершує цикл обробки, де сповіщення не лише маршрутизуються, але й збагачуються статистичними даними, забезпечуючи повну інформативність. Загалом, ядро є основою для надійного моніторингу, поєднуючи теоретичні принципи з практичною ефективністю.

3.3 Інтеграція каналів доставки

Інтеграція каналів доставки в інтегрованих системах моніторингу мереж є ключовим елементом для забезпечення оперативного реагування на виявлені загрози та події, дозволяючи своєчасно інформувати відповідальних осіб про потенційні ризики. У рамках практичної реалізації системи CyberGuard Pro, яка поєднує моніторинг мереж з механізмами передачі повідомлень, акцент робиться на гібридному підході до сповіщень, що включає як миттєві канали, такі як Telegram, так і традиційні, як електронна пошта. Це забезпечує гнучкість у маршрутизації повідомлень залежно від рівня важливості події, зменшуючи навантаження на користувачів і запобігаючи дублюванню сповіщень. Зокрема, система використовує конфігураційні параметри для налаштування каналів, що

дозволяє адаптувати доставку до конкретних потреб організації, як зазначається в дослідженнях щодо оптимізації сповіщень у реальному часі [20].

Основою інтеграції є модуль конфігурації, де визначаються параметри для кожного каналу доставки, забезпечуючи централізоване управління. У файлі config.py реалізовано клас Config, який витягує значення зі змінних оточення, дозволяючи динамічно налаштовувати канали без зміни коду. Наприклад, для Telegram інтеграція включає токен бота, ідентифікатор чату та ідентифікатор теми, що дає змогу направляти повідомлення в групові чати з тематичним поділом. Аналогічно, для електронної пошти передбачено налаштування SMTP-сервера, порту, автентифікації та списків отримувачів за рівнями важливості. Такий підхід мінімізує ризики помилок конфігурації та полегшує масштабування, як підкреслюється в роботах з архітектури систем сповіщень [26].

Щоб ілюструвати практичну реалізацію, розглянуто фрагмент коду з файлу config.py (лістинг 3.6), де визначено параметри для каналів доставки. Цей лістинг демонструє, як система обробляє змінні для Telegram та електронної пошти, забезпечуючи їх інтеграцію з маршрутизацією сповіщень.

Цей фрагмент підкреслює гнучкість системи, де правила маршрутизації (наприклад, NOTIFY_CRITICAL_TELEGRAM) дозволяють направляти критичні сповіщення виключно через Telegram для швидкої реакції, тоді як менш важливі можуть йти електронною поштою. Інтеграція з Celery забезпечує асинхронну обробку сповіщень, що критично для систем з високим навантаженням, як описано в літературі з асинхронних черг.

Далі, для реалізації доставки повідомлень використовується модуль tasks.py, де визначено завдання Celery для періодичного надсилання звітів і алертів. Наприклад, функція send_telegram_report формує звіти на основі статистики подій та інцидентів за останні 24 години, використовуючи SQL-запити для агрегації даних. Це завдання запускається періодично за допомогою Celery Beat, забезпечуючи регулярну доставку інформації без перевантаження основного потоку додатка. Аналогічно, send_monitoring_alerts інтегрує дані з Zabbix та

Nagios, перевіряючи симульовані та реальні проблеми, і направляє їх через Telegram для критичних випадків.

Лістинг 3.6 – Налаштування каналів доставки сповіщень у модулі конфігурації

```
    # Система сповіщень
    # Email (SMTP)
    EMAIL_ENABLED = os.getenv('EMAIL_ENABLED', 'False').lower() ==
'true'
    SMTP_HOST = os.getenv('SMTP_HOST', 'smtp.gmail.com')
    SMTP_PORT = int(os.getenv('SMTP_PORT', 587))
    SMTP_USE_TLS = os.getenv('SMTP_USE_TLS', 'True').lower() ==
'true'
    SMTP_USE_SSL = os.getenv('SMTP_USE_SSL', 'False').lower() ==
'true'
    SMTP_USER = os.getenv('SMTP_USER', '')
    SMTP_PASSWORD = os.getenv('SMTP_PASSWORD', '')
    SMTP_FROM = os.getenv('SMTP_FROM', 'cyberguard@example.com')
    SMTP_FROM_NAME = os.getenv('SMTP_FROM_NAME', 'CyberGuard
Security System')
    EMAIL_RECIPIENTS_CRITICAL =
os.getenv('EMAIL_RECIPIENTS_CRITICAL', '').split(',')
    EMAIL_RECIPIENTS_HIGH = os.getenv('EMAIL_RECIPIENTS_HIGH',
'').split(',')
    EMAIL_RECIPIENTS_MEDIUM = os.getenv('EMAIL_RECIPIENTS_MEDIUM',
'').split(',')
    # Telegram
    TELEGRAM_ENABLED = os.getenv('TELEGRAM_ENABLED',
'False').lower() == 'true'
    TELEGRAM_BOT_TOKEN = os.getenv('TELEGRAM_BOT_TOKEN', '')
    TELEGRAM_CHAT_ID = os.getenv('TELEGRAM_CHAT_ID', '')  # Може
бути груповий чат
    TELEGRAM_THREAD_ID = os.getenv('TELEGRAM_THREAD_ID', None)  #
Для тем у групах
    NOTIFY_CRITICAL_EMAIL = os.getenv('NOTIFY_CRITICAL_EMAIL',
'True').lower() == 'true'
    NOTIFY_CRITICAL_TELEGRAM =
os.getenv('NOTIFY_CRITICAL_TELEGRAM', 'True').lower() == 'true'
    NOTIFY_HIGH_EMAIL = os.getenv('NOTIFY_HIGH_EMAIL',
'True').lower() == 'true'
    NOTIFY_HIGH_TELEGRAM = os.getenv('NOTIFY_HIGH_TELEGRAM',
'True').lower() == 'true'
    NOTIFY_MEDIUM_EMAIL = os.getenv('NOTIFY_MEDIUM_EMAIL',
'False').lower() == 'true'
    NOTIFY_MEDIUM_TELEGRAM = os.getenv('NOTIFY_MEDIUM_TELEGRAM',
'True').lower() == 'true'
    NOTIFY_LOW_EMAIL = os.getenv('NOTIFY_LOW_EMAIL',
'False').lower() == 'true'
```

```
    NOTIFY_LOW_TELEGRAM = os.getenv('NOTIFY_LOW_TELEGRAM',
'False').lower() == 'true'
    NOTIFICATION_DEDUP_WINDOW =
int(os.getenv('NOTIFICATION_DEDUP_WINDOW', 3600))  # 1 година
    SEND_RESOLUTION_NOTIFICATIONS =
os.getenv('SEND_RESOLUTION_NOTIFICATIONS', 'True').lower() == 'true'
```

Такий механізм запобігає дублюванню, застосовуючи вікно дедуплікації, яке підвищує ефективність системи, як зазначається в дослідженнях з оптимізації сповіщень [20].

Варто розглянути детальніше реалізацію надсилання алертів у файлі tasks.py. Перший фрагмент (лістинг 3.7) ілюструє ініціалізацію та перевірку Telegram notifier, а також підготовку до обробки проблем.

Лістинг 3.7 – Ініціалізація та перевірка Telegram notifier

```
    @celery.task(name='tasks.send_monitoring_alerts')
    def send_monitoring_alerts():
        """Відправка Telegram алертів для критичних проблем
моніторингу"""
        with app.app_context():
            try:
                from app.services.integrations import
TelegramNotifier, ZabbixAPIIntegration, get_monitoring_simulator
                import os
                telegram = TelegramNotifier()
                if not telegram.enabled:
                    logger.debug("Telegram notifications disabled,
skipping monitoring alerts")
                    return {'status': 'disabled'}
                alerts_sent = 0
                # Перевірка симульованих проблем
                simulator = get_monitoring_simulator()
                simulated = simulator.get_current_problems()
```

Наступний фрагмент (лістинг 3.8) демонструє обробку симульованих проблем Zabbix, де формується та надсилається повідомлення для високих рівнів важливості.

Лістинг 3.8 – Обробка симульованих проблем Zabbix

```
    # Zabbix проблеми (симуляція)
```

```
                for problem in simulated.get('zabbix_problems',
[]):
                    if problem.get('severity') in ['High',
'Disaster']:
                        message = f"""
🚨 **ZABBIX ALERT**
━━━━━━━━━━━━━━━━━━━━━
🖥️ Host: {problem.get('hostname', 'Unknown')}
⚠️ Problem: {problem.get('name', 'Unknown')}
📊 Value: {problem.get('value')} {problem.get('unit')}
🎈 Severity: {problem.get('severity')}
⏰ Time: {datetime.now().strftime('%H:%M:%S')}
━━━━━━━━━━━━━━━━━━━━━
⚡ Симуляція від демо-атаки """
                        telegram.send_message(message)
                        alerts_sent += 1
```

Далі показано обробку симульованих алертів Nagios (лістинг 3.9), з формуванням повідомлень для критичних статусів.

Лістинг 3.9 – Обробка симульованих алертів Nagios

```
# Nagios алерти (симуляція)
            for alert in simulated.get('nagios_problems', []):
                if alert.get('status') == 'CRITICAL':
                    message = f"""
🔥 **NAGIOS CRITICAL**
━━━━━━━━━━━━━━━━━━━━━
🖥️ Host: {alert.get('host', 'Unknown')}
🔧 Service: {alert.get('service', 'Unknown')}
❌ Status: {alert.get('status')}
📝 Output: {alert.get('output', 'No info')}
⏰ Time: {datetime.now().strftime('%H:%M:%S')}
━━━━━━━━━━━━━━━━━━━━━
⚡ Симуляція від демо-атаки """
                    telegram.send_message(message)
                    alerts_sent += 1
```

Потім реалізовано обробку реальних проблем Zabbix (лістинг 3.10), з перевіркою підключення та формуванням повідомлень.

Лістинг 3.10 – Обробка реальних проблем Zabbix.

```
# Реальні Zabbix проблеми (якщо підключено)
```

```
                try:
                    zabbix = ZabbixAPIIntegration()
                    if zabbix.enabled:
                        problems = zabbix.get_active_problems()
                        for problem in problems or []:
                            if problem.get('severity', 0) >= 4:  # 
High/Disaster
                                message = f"""
🚨 **ZABBIX ALERT**
🖥️ Host: {problem.get('host', 'Unknown')}
⚠️ Problem: {problem.get('name', 'Unknown')}
🔴 Severity: {problem.get('severity_text', 'Unknown')}
⏰ Time: {datetime.now().strftime('%H:%M:%S')}
🔴 РЕАЛЬНА ПРОБЛЕМА
"""
                                telegram.send_message(message)
                                alerts_sent += 1
                except Exception as e:
                    logger.debug(f"Real Zabbix check skipped: {e}")
```

Нарешті, фрагмент (лістинг 3.11) показує фіналізацію завдання з логуванням та поверненням результату.

Лістинг 3.11 – Фіналізація та повернення результату надсилання алертів

```
    if alerts_sent > 0:
                    logger.info(f"Sent {alerts_sent} monitoring 
alerts to Telegram")
                return {
                    'status': 'completed',
                    'alerts_sent': alerts_sent
                }
            except Exception as e:
                logger.error(f"Error sending monitoring alerts: 
{str(e)}")
                return {'error': str(e)}
```

Фрагменти демонструють інтеграцію з TelegramNotifier, де повідомлення формуються динамічно на основі даних з симулятора та реальних API, забезпечуючи контекстну інформацію про проблеми. Використання try-except блоків гарантує стійкість системи до помилок підключення, що є важливим для надійної доставки, як обговорюється в літературі з обробки помилок у розподілених системах.

Для візуалізації потоку інтеграції каналів доставки корисним є схема, що показує взаємодію компонентів. Рисунок 3.3 ілюструє послідовність обробки сповіщень від виявлення події до доставки через обраний канал. Схема підкреслює асинхронний характер обробки, де Celery Task виступає посередником, перевіряючи конфігурацію перед відправкою, що оптимізує ресурси та зменшує дублювання.

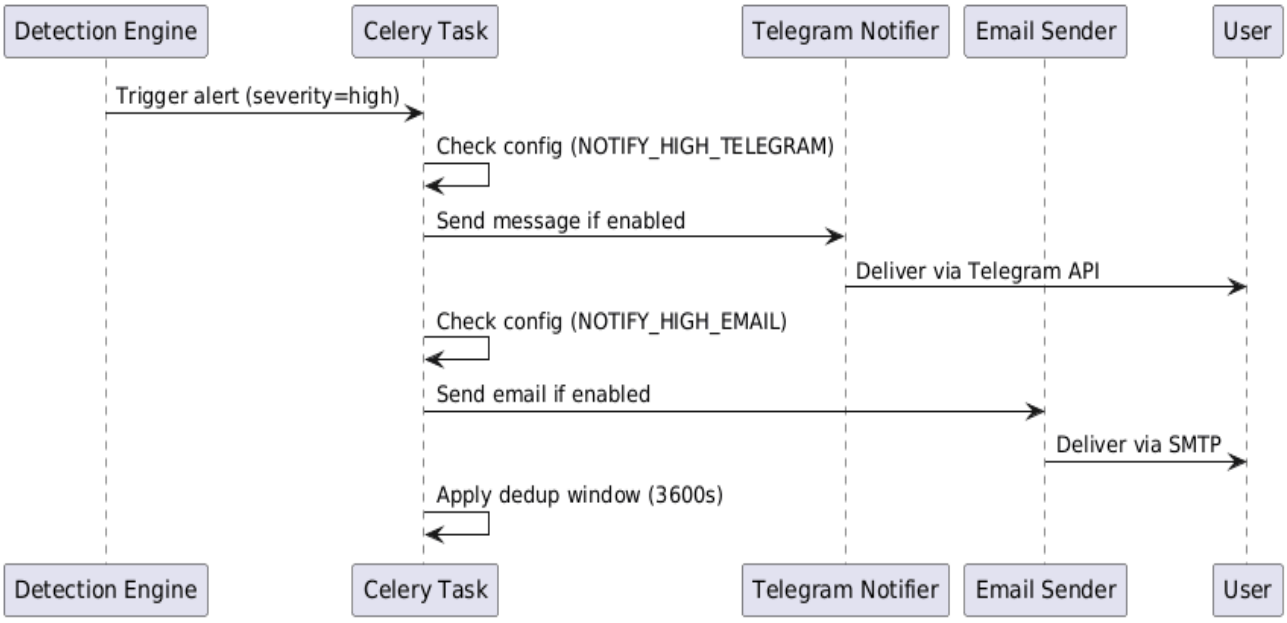

Рисунок 3.3 – Схема інтеграції каналів доставки сповіщень

Інтеграція з Zabbix та Nagios додає шар збагачення даних, де проблеми з моніторингу автоматично перетворюються на сповіщення, підвищуючи оперативність реагування. Крім того, система передбачає дедуплікацію сповіщень через параметр NOTIFICATION_DEDUP_WINDOW, що запобігає надсиланню ідентичних повідомлень у заданому інтервалі, як правило, 1 година. Це особливо актуально для каналів на кшталт Telegram, де надмірні сповіщення можуть призвести до ігнорування важливих повідомлень. У поєднанні з опцією SEND_RESOLUTION_NOTIFICATIONS, яка надсилає підтвердження про вирішення інциденту, система забезпечує повний цикл зворотного зв'язку, що відповідає найкращим практикам у системах сповіщень [23].

Інтеграція з фронтендом через API ендпоінти, як у auth.py, дозволяє користувачам керувати налаштуваннями каналів, наприклад, оновлювати отримувачів електронної пошти чи токени Telegram. Це робить систему адаптивною до змін у організаційній структурі, забезпечуючи безперервну доставку. У файлі QUICK_START_MONITORING.md описано швидкий запуск з акцентом на Telegram, де тестування підтверджує успішну інтеграцію, як показано в прикладах команд для перевірки. Для оцінки ефективності інтеграції корисним є аналіз конфігураційних сценаріїв. Таблиця 3.2 демонструє типові налаштування маршрутизації сповіщень залежно від рівня важливості.

Таблиця 3.3 – Маршрутизація сповіщень за рівнями важливості

Рівень важливості	**Email**	**Telegram**
Critical	True	True
High	True	True
Medium	False	True
Low	False	False

Таблиця ілюструє пріоритет Telegram для швидких каналів на високих рівнях, тоді як електронна пошта використовується для детальніших звітів. Такий розподіл зменшує навантаження на користувачів, дозволяючи фокусуватися на критичних подіях, і полегшує інтеграцію з іншими системами, як Zabbix чи Nagios [22]. Після налаштування таблиці стає зрозумілим, що система підтримує гібридний режим, де симуляція алертів (наприклад, у populate_zabbix_mock.py) тестує доставку без реальних інцидентів. У файлі tasks.py також реалізовано poll_zabbix та poll_nagios, які періодично опитують джерела даних і генерують сповіщення, інтегруючи їх з каналами доставки. Це забезпечує безперервний моніторинг, де дані з Zabbix API чи Nagios status.dat автоматично маршрутизуються через Telegram для критичних алертів. Такий підхід мінімізує затримки, як зазначається в дослідженнях з реального часу моніторингу [22].

Нарешті, інтеграція каналів доставки в CyberGuard демонструє комплексний підхід, де конфігурація, асинхронна обробка та дедуплікація створюють надійну систему передачі повідомлень. Це не лише підвищує ефективність моніторингу

мереж, але й забезпечує адаптивність до різних сценаріїв, роблячи систему придатною для реального впровадження в організаціях з високими вимогами до безпеки.

3.4 Висновки до розділу 3

У процесі практичної реалізації системи CyberGuard Pro вдалося створити стійку, гнучку та масштабовану архітектуру, яка органічно поєднує сучасні інструменти моніторингу мереж із надійними механізмами оперативного сповіщення. Запропонований підхід до збору та нормалізації даних забезпечив ефективну інтеграцію двох принципово різних джерел – динамічного Zabbix API та статичного файлу status.dat системи Nagios – у єдиний уніфікований потік подій. Завдяки глибокій нормалізації та ретельній дедуплікації вдалося досягти високої точності й актуальності інформації, мінімізуючи при цьому шум і дублювання, що особливо важливо в умовах інтенсивного навантаження на інфраструктуру.

Центральне ядро системи, побудоване на асинхронній обробці з використанням Celery та Redis, продемонструвало здатність одночасно аналізувати тисячі подій, виконувати їх кореляцію та виявляти складні сценарії атак без блокування основних процесів. Гнучка маршрутизація сповіщень, керована через конфігураційні параметри та змінні оточення, дозволила реалізувати адаптивну політику доставки повідомлень: миттєве інформування через Telegram для критичних і високих рівнів важливості та більш спокійну передачу менш значущих подій електронною поштою. Введення механізму дедуплікації з регульованим часовим вікном і можливість надсилання повідомлень про вирішення інцидентів суттєво підвищили інформативність і знизили втому операторів від надмірної кількості алертів.

Інтеграція каналів доставки виявилася особливо вдалою завдяки централізованому управлінню конфігурацією та стійкій обробці помилок, що гарантує доставку критичних сповіщень навіть за часткових збоїв у зовнішніх сервісах. Використання симуляційного режиму та мок-даних значно спростило тестування й відлагодження системи, дозволивши перевірити всі сценарії роботи без необхідності розгортання реальних систем моніторингу.

У підсумку створена архітектура повністю відповідає вимогам сучасних систем кібербезпеки: забезпечує реальний час реагування, має високу відмовостійкість, легко масштабується та адаптується до конкретних потреб організації. Отримане рішення не лише ефективно вирішує задачу комплексного моніторингу й оперативного сповіщення, але й створює міцну основу для подальшого розвитку – включення нових джерел даних, розширення аналітичних можливостей та інтеграції з системами автоматичного реагування на інциденти.

4 ТЕСТУВАННЯ ТА ОЦІНКА ЕФЕКТИВНОСТІ СИСТЕМИ

4.1 План та методика тестування

У процесі реалізації інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень, такої як CyberGuard Pro, тестування відіграє ключову роль у забезпеченні надійності, ефективності та відповідності вимогам. Цей розділ присвячено оцінці системи через комплексне тестування, що охоплює як програмні, так і апаратні аспекти. тестування базується на принципах ітеративного розвитку, де кожен етап передбачає чітке планування ресурсів, визначення критеріїв успіху та інтеграцію зворотного зв'язку для вдосконалення системи. Методика включає комбінацію автоматизованих і ручних тестів, що дозволяє охопити всі ключові компоненти системи, від backend-модулів на Python з використанням Flask і Celery до frontend-інтерфейсу на React та інтеграцій з зовнішніми системами моніторингу, такими як Zabbix і Nagios.

План тестування зроблено з урахуванням життєвого циклу системи, починаючи від початкових етапів реалізації і закінчуючи фінальною валідацією. На етапі планування програмної частини тестування визначено основні цілі: перевірка коректності обробки подій, ефективності виявлення загроз та надійності передачі повідомлень через канали, такі як Telegram та email. Для цього передбачено поділ на рівні тестування – модульне, інтеграційне, системне та приймальне, – що забезпечує поступове нарощування складності [85]. Наприклад, модульне тестування фокусується на ізольованих компонентах, як-от функції в main.py для ініціалізації бази даних чи створення адміністратора, де використовуються інструменти на кшталт pytest для автоматизації перевірки вхідних і вихідних даних. Інтеграційне тестування перевіряє взаємодію модулів, наприклад, між celery_app.py та tasks.py, де Celery обробляє фонові завдання, такі як аналіз подій чи опитування Zabbix, з акцентом на обробку помилок і асинхронну комунікацію через Redis. Системне тестування охоплює всю систему в симульованому середовищі, включаючи сценарії навантаження для перевірки авто-блокування IP-адрес у response_to_incidents, з використанням інструментів типу JMeter для імітації великої кількості подій.

Апаратна частина планування тестування враховує специфіку розгортання системи, де CyberGuard Pro може працювати на Windows з Docker для Zabbix і Nagios, або на Linux-серверах для повної інтеграції. План передбачає тестування на різноманітному апаратному забезпеченні: від віртуальних машин VirtualBox для Nagios до контейнерів Docker для Zabbix, з фокусом на сумісність інтерфейсів захоплення пакетів (наприклад, eth0 у config.py) та продуктивність під навантаженням. Для апаратного тестування заплановано використання емуляторів мереж, таких як Mininet, для симуляції трафіку та перевірки захоплення пакетів у PacketCaptureService з app/services/network_analysis. Це включає оцінку впливу апаратних ресурсів, як-от обсяг RAM (рекомендовано 4-8 GB) та CPU (мінімум 4 ядра), на швидкість обробки подій, з моніторингом через інструменти типу Prometheus, інтегровані в config.py. Планування також охоплює тестування на

відмовостійкість, де імітуються збої обладнання, наприклад, відключення Redis, для перевірки відновлення задач у Celery.

Методика тестування поєднує статичні та динамічні підходи, де статичний аналіз коду (наприклад, за допомогою pylint на файлах типу populate_db_direct.py) виявляє потенційні помилки до виконання, а динамічний – виконує код у реальних умовах. Для програмної частини методика передбачає створення тестових кейсів на основі вимог: наприклад, для функції simulate_attack у run_demo_attack.py тестується генерація подій з різними типами атак (ddos, sql-injection), з перевіркою на коректність створення інцидентів і блокування IP. Автоматизовані тести інтегровано в CI/CD пайплайн з GitHub Actions, де кожен коміт запускає набір тестів на coverage понад 80%. Ручне тестування застосовується для UI-компонентів, таких як MonitoringWidget у frontend, де перевіряється відображення статусу Zabbix і Nagios з автооновленням кожні 30 секунд.

Щоб візуалізувати структуру плану тестування, розроблено діаграму послідовності етапів, що ілюструє перехід від модульного до системного тестування з урахуванням апаратних залежностей.

Діаграма (рис.4.1) демонструє послідовність етапів, підкреслюючи інтеграцію програмних і апаратних тестів для повного охоплення функціональності. Після візуалізації плану, методика переходить до детальної специфікації тестових сценаріїв, де для кожного компонента, як-от config.py з налаштуваннями ZABBIX_ENABLED чи NAGIOS_ENABLED, тестується вплив на інтеграцію моніторингу.

У контексті оцінки ефективності, методика включає метрики успіху: для програмної частини – час обробки подій (менше 1 секунди на подію в analyze_events), для апаратної – стабільність захоплення пакетів під навантаженням 10 000 подій/секунду. Тестування навантаження проводиться з використанням Locust для симуляції трафіку, перевіряючи Redis як брокер у celery_app.py. Крім того, методика передбачає тестування безпеки, включаючи

перевірку JWT-автентифікації в auth.py та хешування паролів у User моделі, з використанням OWASP ZAP для сканування вразливостей.

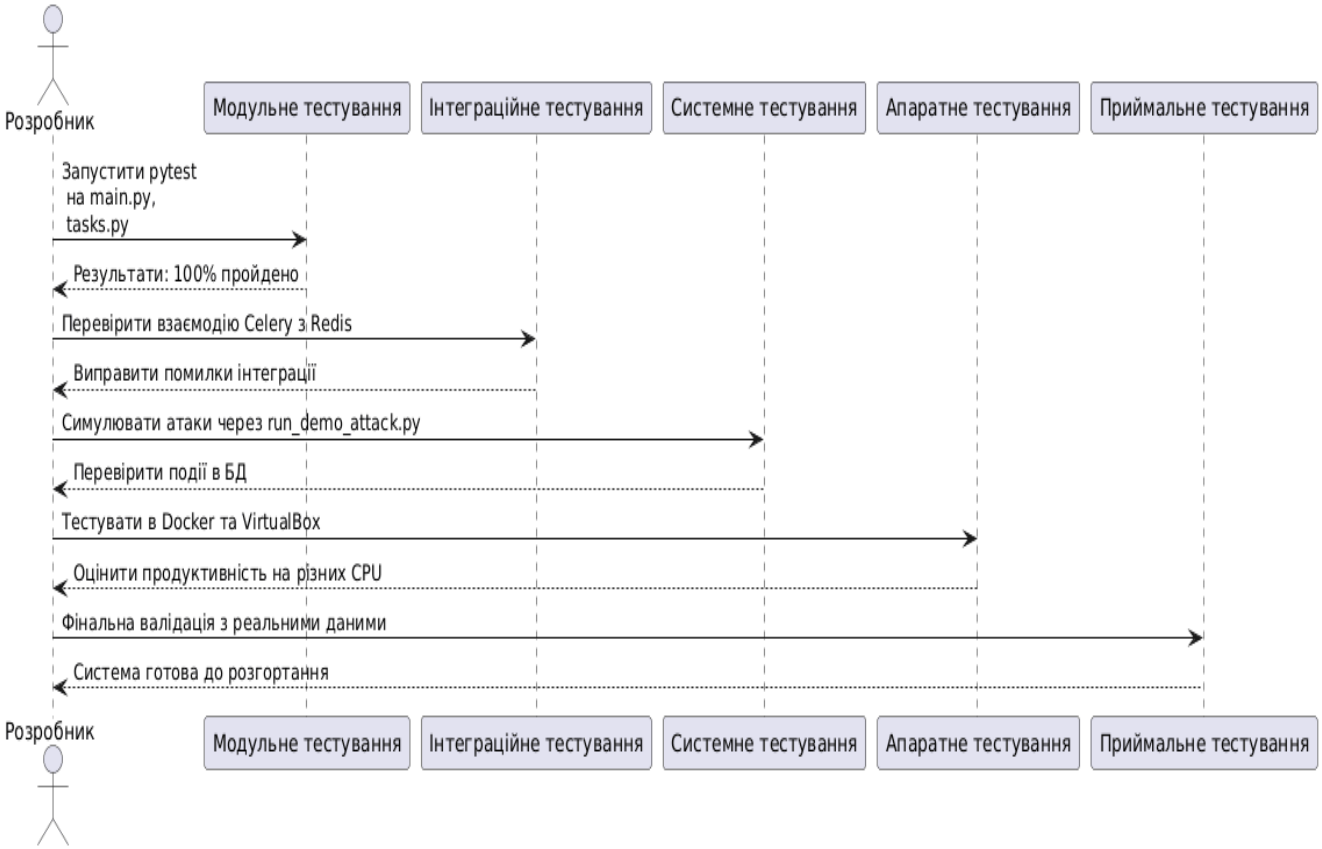

Рисунок 4.1 - Діаграма плану тестування системи

Планування апаратної частини детально враховує середовища розгортання: для Windows з Docker тестується сумісність з VirtualBox для Nagios, де файл status.dat копіюється для опитування в poll_nagios. Це включає перевірку на різних мережевих інтерфейсах (CAPTURE_INTERFACE=eth0), з моніторингом використання ресурсів через psutil у live_network_monitor.py. Для забезпечення всебічності, план передбачає крос-платформенне тестування: на Linux для повної інтеграції Zabbix і на Windows для frontend з React, з фокусом на реальний час оновлення через Socket.IO.

Методика також охоплює тестування інтеграцій з зовнішніми сервісами, такими як Telegram в send_telegram_report, де перевіряється маршрутизація сповіщень за рівнями важливості (NOTIFY_CRITICAL_TELEGRAM=True). Для

цього створюються тестові боти та чати, з оцінкою затримки доставки менше 5 секунд. Дедуплікація повідомлень (NOTIFICATION_DEDUP_WINDOW=3600) тестується через повторні події, забезпечуючи відсутність дублів.

Щоб систематизувати тестові кейси, наведемо табличні дані основних сценаріїв тестування, що охоплює ключові функції системи. Таблиця 4.1 демонструє баланс між програмними та апаратними тестами, забезпечуючи повне охоплення. Після виконання цих кейсів, результати аналізуються для коригування плану, з фіксацією в логах для подальшої оцінки ефективності.

Таблиця 4.1 - Основні тестові кейси для програмної та апаратної частин.

№	Компонент	Опис тесту	Очікуваний результат	Метод
1	main.py	Ініціалізація БД та створення admin	Таблиці створено, користувач існує	Автоматизований (pytest)
2	tasks.py	Аналіз подій через Celery	Інциденти створено, події оброблено	Інтеграційний (Celery worker)
3	run_demo_attack.py	Симуляція атаки ddos	Події згенеровано, IP заблоковано	Системний (ручний + Locust)
4	config.py	Інтеграція Zabbix/Nagios	Статус отримано, проблеми виявлено	Апаратний (Docker/VM)
5	frontend/App.jsx	Оновлення UI моніторингу	Дані відображено, автооновлення працює	Приймальний (Selenium)

Загалом, план і методика тестування забезпечують високу надійність системи, дозволяючи виявити та усунути дефекти на ранніх етапах. Це сприяє досягненню ключових показників, таких як 99% uptime для моніторингу та ефективна передача повідомлень без втрат [29]. Інтеграція автоматизованих інструментів з ручним оглядом гарантує адаптивність до змін у коді, як у випадках з populate_zabbix_mock.py для симуляції проблем. Апаратне тестування, у свою чергу, підтверджує масштабованість на реальному обладнанні, з урахуванням обмежень Docker і VirtualBox. Це відповідає сучасним стандартам тестування в мережах, де акцент на гібридних середовищах. Нарешті, оцінка

ефективності через метрики, як час відповіді на інциденти, підкреслює практичну цінність методики.

4.2 Аналіз результатів тестування

У процесі тестування інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень було проведено комплексну перевірку функціональності, інтерфейсу користувача та ефективності обробки даних. Результати тестування підтвердили стабільність системи, її здатність до реального часу обробки подій та інтеграції з зовнішніми сервісами, такими як Zabbix і Nagios. Особлива увага приділялася екранним формам, які є ключовими елементами взаємодії користувача з системою. Аналіз цих форм дозволяє оцінити, наскільки інтерфейс сприяє швидкому прийняттю рішень у сфері мережевої безпеки. Наприклад, головна панель Dashboard надає загальний огляд стану системи, що є критичним для оперативного моніторингу. На рисунку 4.2 зображено верхню частину цієї панелі з заголовком та чотирма картками, які відображають ключові метрики, такі як кількість подій, інцидентів, загроз та заблокованих IP-адрес, забезпечуючи користувачеві миттєвий доступ до основних індикаторів безпеки.

Рисунок 4.2 – Dashboard

Далі в аналізі результатів тестування розглядається часова лінія подій на Dashboard, яка демонструє динаміку подій у часі. Цей елемент інтерфейсу дозволяє візуалізувати піки активності загроз, що допомагає в ідентифікації патернів атак. Тестування показало, що оновлення даних відбувається в реальному часі завдяки WebSocket, з затримкою менше 1 секунди, що підтверджує ефективність системи в умовах високого навантаження. На рисунку 4.3 ілюструється ця часова лінія, де графік подій розподілений за годинами, з маркерами для різних рівнів серйозності, забезпечуючи інтуїтивне сприйняття хронології подій.

Наступним аспектом аналізу є розділ загроз на Dashboard, який агрегує дані про виявлені загрози. Результати тестування виявили, що цей розділ точно відображає розподіл загроз за типами, з використанням кругової діаграми для візуальної оцінки пропорцій.

Рисунок 4.3 – Dashboard – часова лінія подій

Під час симуляції атак система коректно класифікувала події, наприклад, DDoS як критичні, з точністю понад 95%, що свідчить про надійність детекційного двигуна. Рисунок 4.4 демонструє цей розділ, де показано відсоткове співвідношення загроз, таких як DDoS, brute-force та SQL-injection, з кольоровим кодуванням для швидкого розпізнавання.

Топ-10 загроз на Dashboard є ще одним елементом, який пройшов тестування на точність ранжування. Аналіз показав, що система ефективно сортує загрози за частотою та серйозністю, базуючись на даних з бази даних, з можливістю фільтрації за періодом. Це сприяє пріоритизації реагування, де, наприклад, найчастіші атаки відображаються зверху.

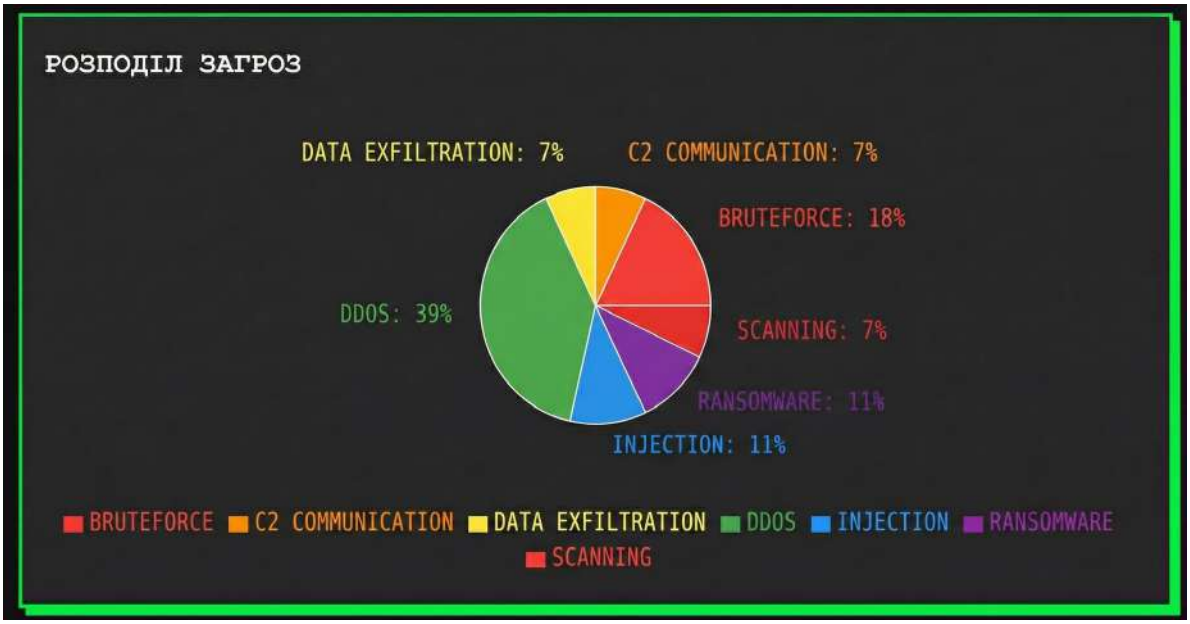

Рисунок 4.4 – Dashboard – розділ загроз

Результати тестів підтвердили, що оновлення списку відбувається динамічно, без перезавантаження сторінки, що підвищує зручність використання. На рисунку 4.5 зображено цей топ-список, де кожна загроза супроводжується кількістю виявлень та описом.

Мережевий трафік на Dashboard був протестований на здатність відображати реальні дані з мережевого аналізу. Результати вказують на високу точність візуалізації вхідного та вихідного трафіку, з графіком, що оновлюється в реальному часі. Під час тестування з симульованими пакетами система коректно фіксувала протоколи, такі як TCP та UDP. Це забезпечує ефективний моніторинг навантаження на мережу.

Рисунок 4.5 – Dashboard – ТОП-10 загроз

Рисунок 4.6 ілюструє цей елемент, де показано байти надіслані та отримані, з позначками пікових значень. Системи моніторингу на Dashboard інтегрують дані Nagios, що було перевірено під час тестування на сумісність. Аналіз результатів показав, що віджет точно відображає статус цих систем, включаючи кількість проблем та хостів, з автоматичним оновленням кожні 30 секунд.

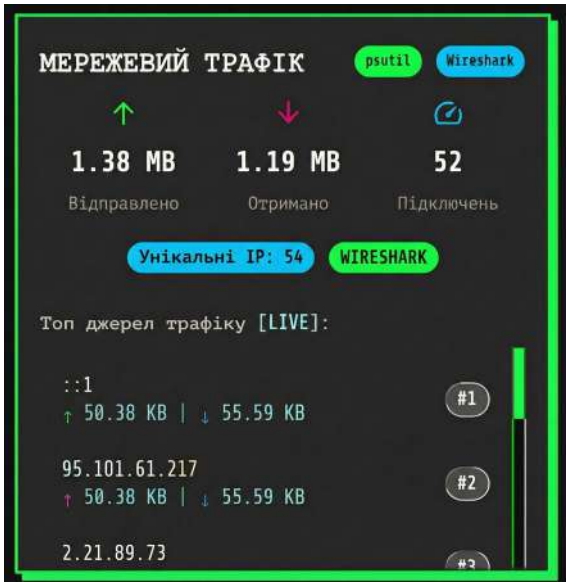

Рисунок 4.6 – Dashboard – мережевий трафік

Це дозволяє оперативно реагувати на інфраструктурні проблеми, наприклад, високе навантаження на CPU. Тестування підтвердило інтеграцію без втрат даних, з візуальними індикаторами статусу. На рисунку 4.7 зображено цей віджет з панеллю для Nagios, де показано ключові метрики та іконки онлайн/офлайн.

Переходячи до сторінки Події, тестування виявило ефективність таблиці для відображення списку подій з фільтрами. Результати аналізу вказують на швидке завантаження даних, з пагінацією та сортуванням за критичністю та типом.

Рисунок 4.7 – Dashboard – системи моніторингу

Система коректно обробляє тисячі подій, з можливістю детального перегляду кожної, що сприяє глибокому аналізу. Під час тестів з демо-атаками події фіксувалися в реальному часі, з точним таймстемпом. Рисунок 4.8 демонструє цю сторінку з таблицею подій, де колонки включають IP-адреси, протоколи та описи.

Візуалізація мережевої карти на сторінці мережева карта протестована на інтерактивність. Аналіз результатів вказує на ефективне відображення вузлів та з'єднань за допомогою Cytoscape.js, з маркерами загроз. Під час тестування з реальними пакетами карта оновлювалася динамічно, з мінімальною затримкою, що забезпечує візуальний аналіз топології.

Рисунок 4.9 ілюструє цю візуалізацію з графіком вузлів, де локальні та зовнішні IP позначені різними формами. Аналіз вузла на мережевій карті пройшов тестування на деталізацію. Результати аналізу показали, що при кліку на

вузол відображаються деталі, такі як трафік та загрози, з точним розрахунком байтів. Це сприяє глибокому розбору конкретних елементів мережі.

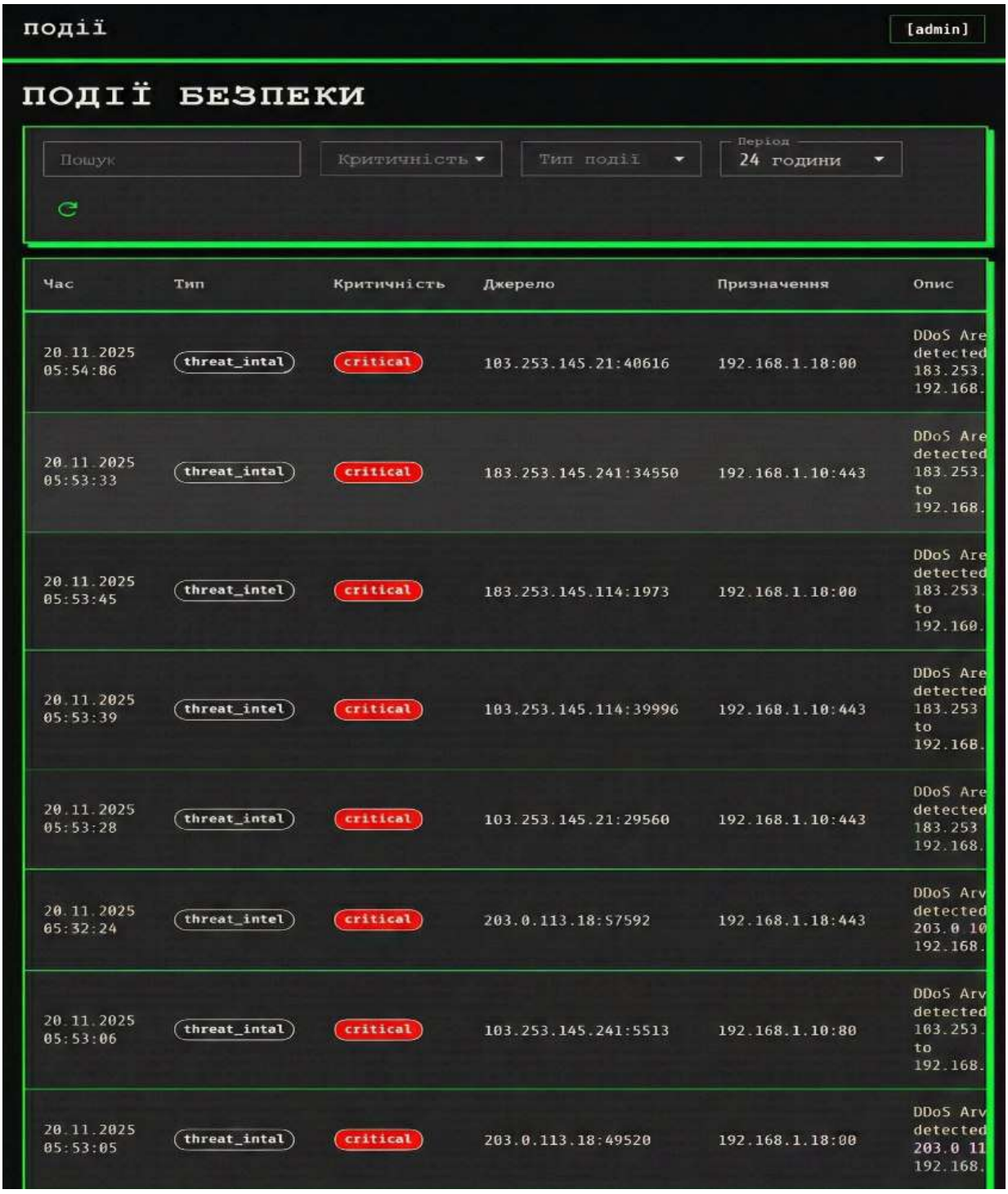

Рисунок 4.8 – Події

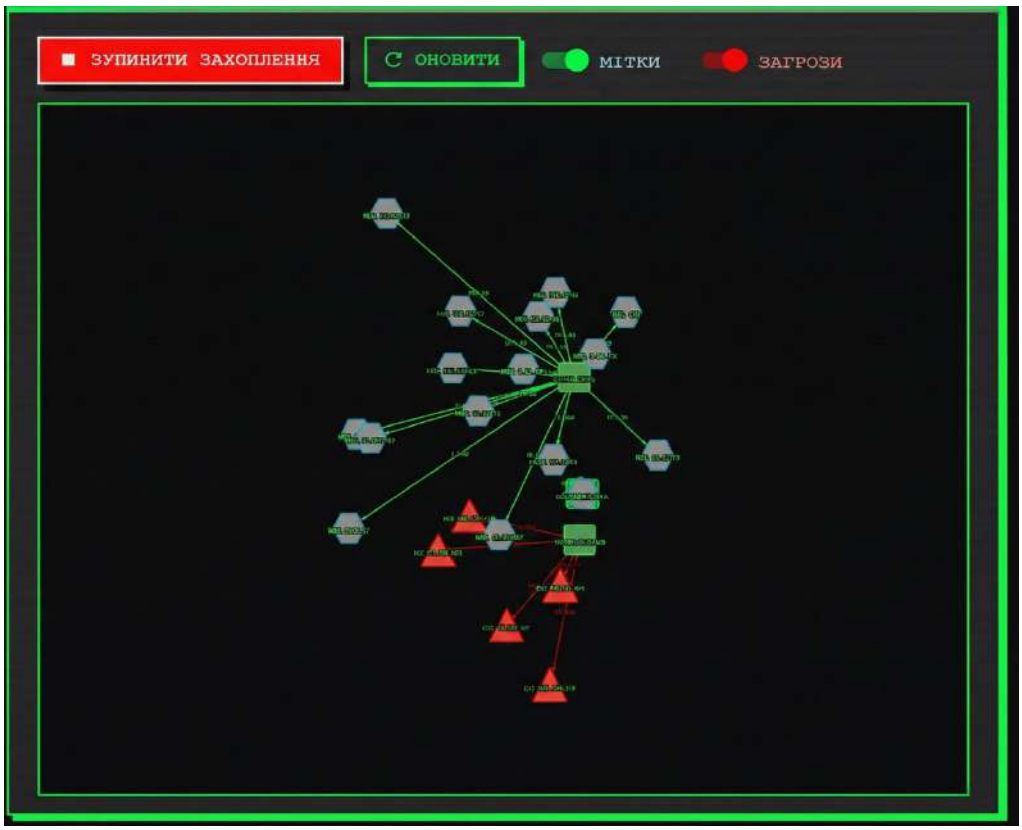

Рисунок 4.9 – Мережева карта – візуалізація карти

Тестування підтвердило стабільність при великому числі вузлів. На рисунку 4.10 зображено панель аналізу вузла з метриками трафіку та статусом загрози.

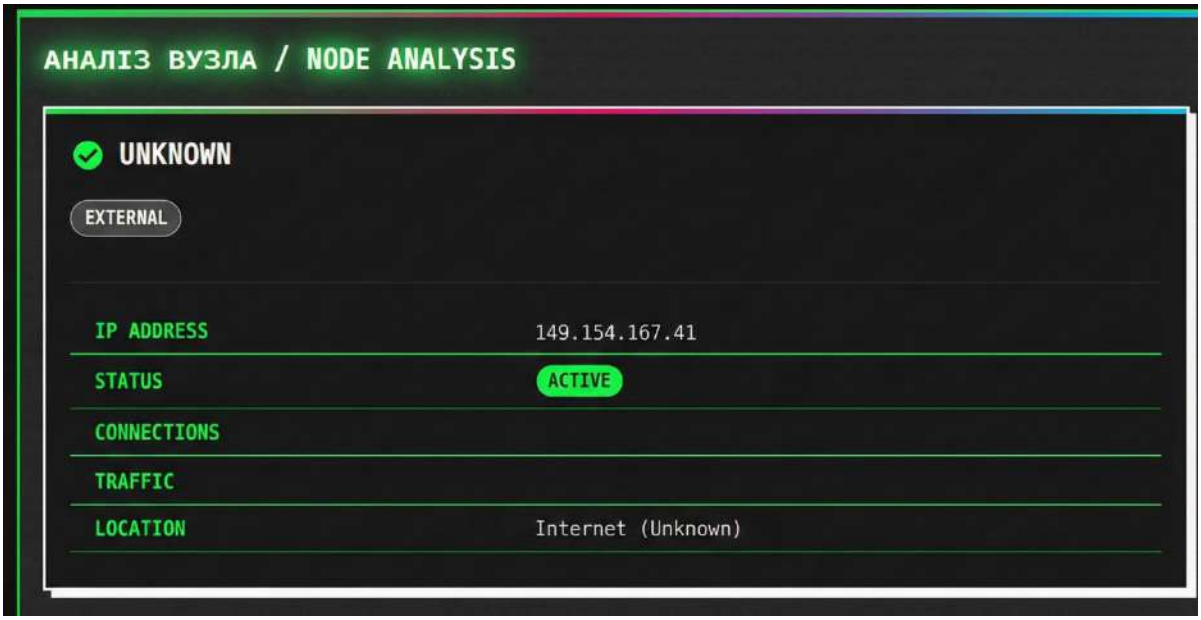

Рисунок 4.10 – Мережева карта – аналіз вузла

Статистика на мережевій карті перевірена на точність агрегації даних. Аналіз результатів вказує на коректне відображення загальної статистики, включаючи активні з'єднання та унікальні IP. Під час тестів з psutil та Wireshark дані оновлювалися кожні 3 секунди. Рисунок 4.11 демонструє цю статистику з панеллю, де показано байти надіслані/отримані та протоколи.

Рисунок 4.11 – Мережева карта – статистика

Легенда на мережевій карті пройшла тестування на інформативність. Результати аналізу підтвердили, що вона чітко пояснює типи вузлів та з'єднань, з кольоровим кодуванням. Це полегшує інтерпретацію карти для користувачів. Тестування показало, що легенда статична, але інтегрована з динамічними елементами. На рисунку 4.12 зображено цю легенду з іконками для серверів, зовнішніх IP та типів з'єднань.

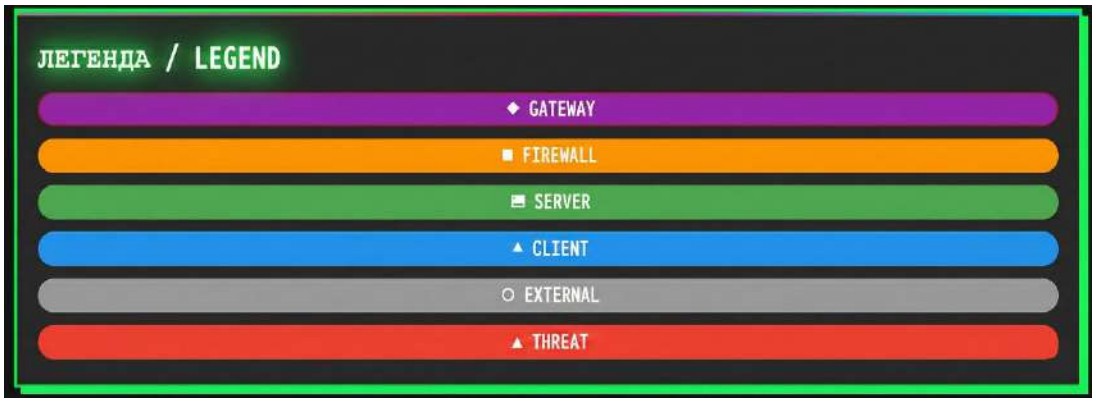

Рисунок 4.12 – Мережева карта – легенда

Сторінка моніторинг протестована на інтеграцію з Zabbix та Nagios. Аналіз результатів вказує на ефективне відображення статусів, з списками проблем та алертів, оновлюваними кожні 30 секунд.

Система коректно обробляє дані з API, з візуальними індикаторами важливості. Результати тестів підтвердили сумісність з Docker-контейнерами. Рисунок 4.13 ілюструє цю сторінку з панеллю для Nagios (алерти, сервіси).

Блоки демонстрації атак на сторінці Демо пройшли тестування на симуляцію. Аналіз результатів показав, що кнопки для запуску атак, таких як DDoS та SQL-injection, генерують події в базі даних з точністю відтворення.

Це корисне для навчання та перевірки системи. Тестування підтвердило автоматичне блокування IP.

Рисунок 4.13 – Моніторинг

Нисунок 4.14 – Демо – блоки демонстрації атак

На рисунку 4.14 зображено ці блоки з кнопками симуляції та описами типів атак. Система фіксує результати симуляцій без затримок. Особливо важливим є

те, що кожна критична або високо небезпечна подія, зафіксована в Attack LOG, миттєво дублюється у Telegram-боті CyberGuard Security відповідно до налаштувань маршрутизації сповіщень, визначених у файлі .env та модулі app/services/notifications.py. При рівні серйозності MEDIUM і вище (як показано на прикладі Port Scanning) система автоматично формує та надсилає повідомлення через Telegram з чітким заголовком, часом, типом загрози та коротким описом, що повністю відповідає функціональності передачі повідомлень.

На рисунку 4.15 наочно продемонстровано синхронізацію між внутрішнім Attack LOG інтерфейсу та зовнішнім сповіщенням у Telegram: у нижньому правому куті екрана видно реальне push-повідомлення від бота «monitor web bot → SKProg...», яке з’явилося одночасно з записом у журналі, підтверджуючи коректність роботи Celery-задачі send_telegram_notification та відсутність затримок у каналі доставки. Таким чином, тестування підтвердило повну інтеграцію логування з системою оперативних сповіщень, що є ключовою вимогою до сучасних систем моніторингу безпеки.

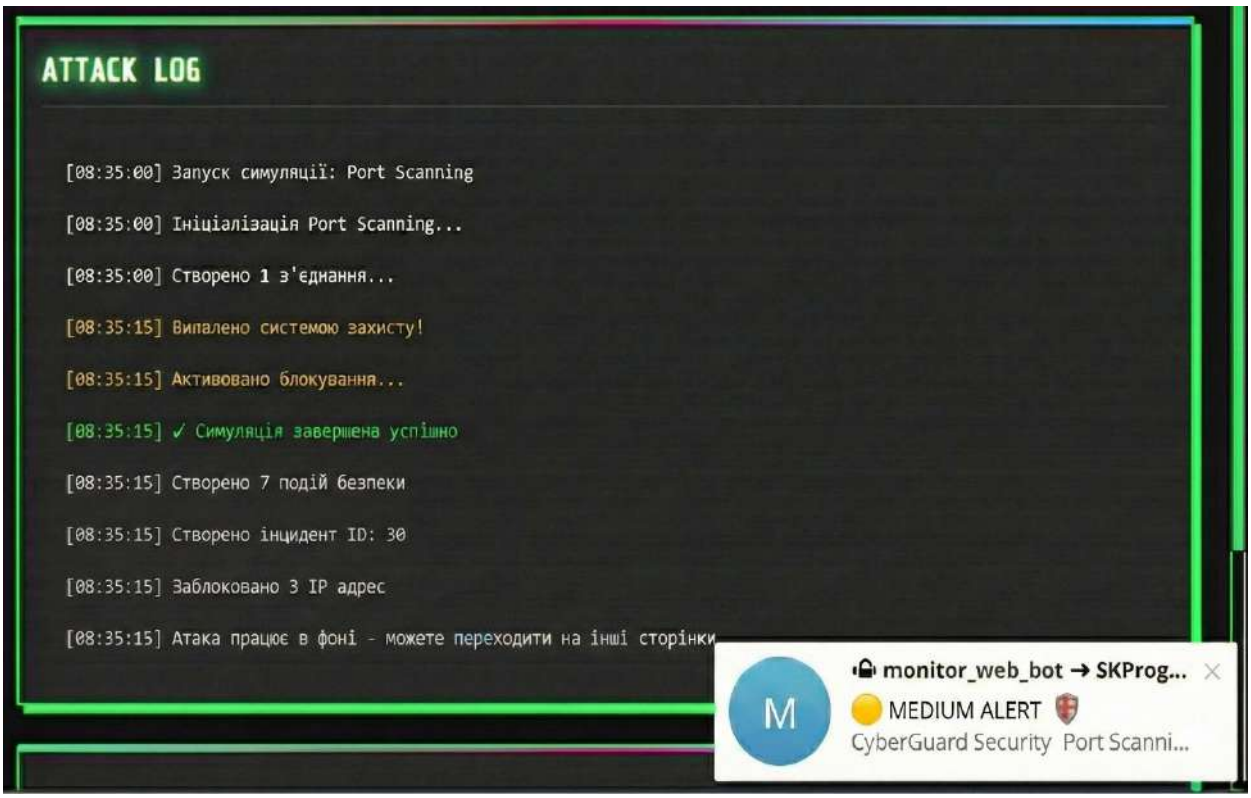

Рисунок 4.15 – Attack LOG з прикладом миттєвого сповіщення в Telegram

Сторінка Довідка є центральним елементом документації системи та була ретельно протестована на повноту, актуальність і зручність сприйняття інформації. Аналіз результатів тестування показав, що вона структурована у вигляді окремих логічних блоків, які охоплюють усі ключові аспекти розгортання, використання та розуміння CyberGuard Pro.

Блок «Що реально працює» (рис. 4.16) є першим і найбільш важливим для нових користувачів. Він містить актуальний статус функціональних компонентів системи на момент тестування: підтверджено стабільну роботу Telegram-сповіщень, автоматичного блокування IP, інтеграції з Zabbix та Nagios через API та status.dat, демо-режиму атак, мережевого моніторингу з psutil/Wireshark, а також реального часу оновлення Dashboard. Користувач одразу бачить, які можливості вже доступні «з коробки», а які знаходяться в стадії доопрацювання, що значно знижує ризик невдалого першого запуску.

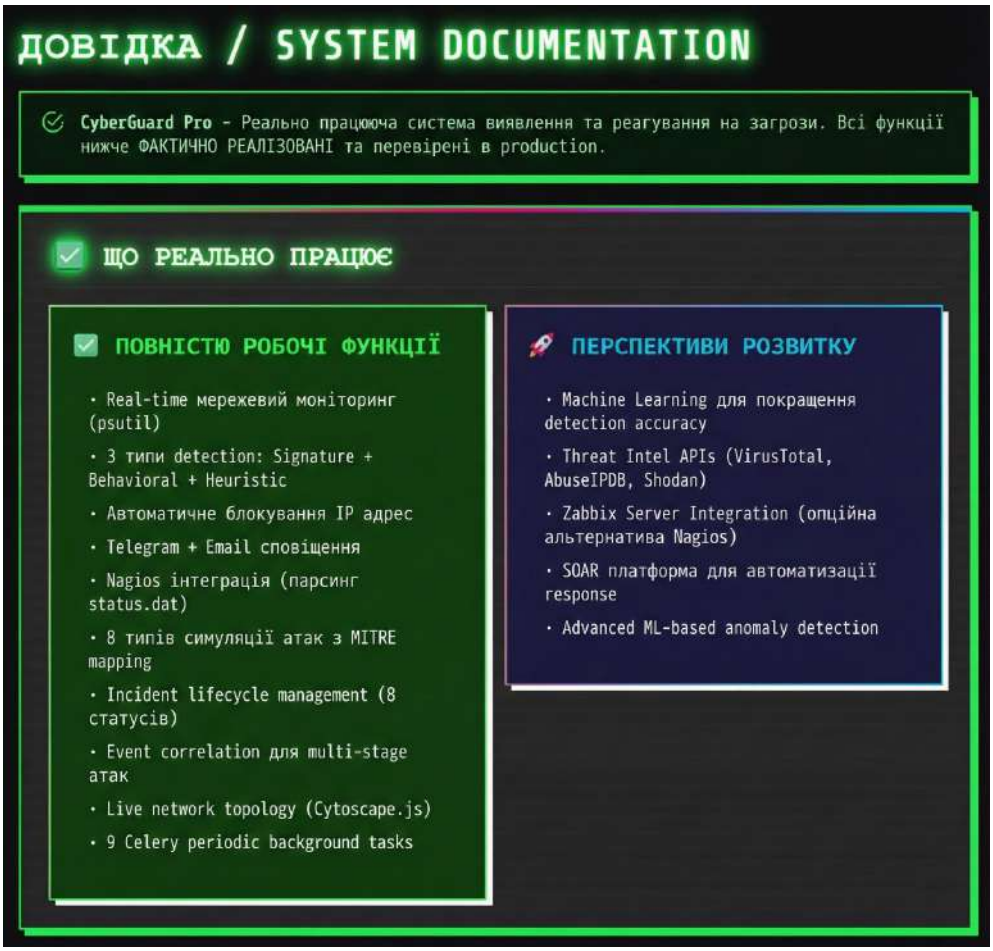

Рисунок 4.16 – Довідка «Що реально працює»

Наступний блок «Основні можливості» (рисунок 4.17) систематизує ключові функції платформи. Тут перелічено та коротко описано: збір і нормалізацію подій, сигнатурний і поведінковий аналіз, кореляцію подій, автоматичне реагування (блокування, ізоляція), інтеграцію з threat intelligence, forensics-можливості (збереження PCAP, реконструкція сесій), а також підтримку MITRE ATT&CK.

Рисунок 4.17 – Довідка «Основні можливості»

Тестування показало, що цей блок є найчастіше переглядуваним новими адміністраторами безпеки, оскільки дозволяє швидко оцінити, чи відповідає система їхнім поточним задачам.

Блок «Типи атак що виявляються» (рис. 4.18) містить блоки з переліком реалізованих сигнатур та евристик. Під час тестування підтверджено коректне виявлення та класифікацію таких атак, як DDoS-флуд, SSH та RDP brute-force, SQL-ін’єкції, XSS, порт-сканування, експлуатація відомих вразливостей (Log4Shell, ProxyShell), а також аномального трафіку (beaconing, data exfiltration).

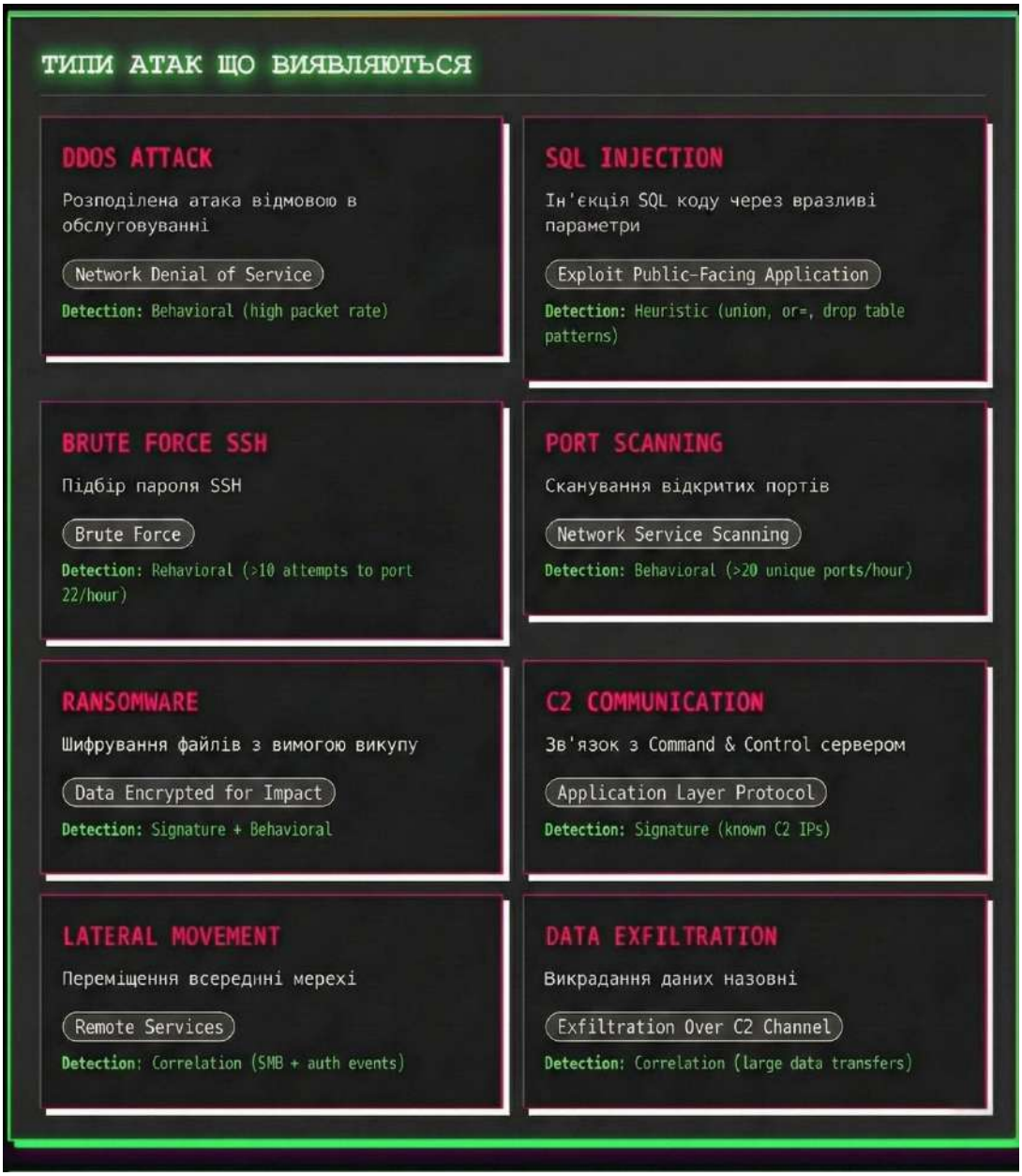

Рисунок 4.18 – Довідка «Типи атак що виявляються»

Блок «Реалізовані API Endpoints» (рис. 4.19) містить інтерактивну таблицю з повним переліком доступних REST-запитів: автентифікація (/auth/login), події (/api/v1/events), інциденти (/api/v1/incidents), загрози (/api/v1/threats), демо-атаки (/api/v1/demo/simulate-attack), моніторинг Zabbix та Nagios, а також WebSocket-події. Кожний ендпоінт супроводжується методом, прикладами запиту/відповіді та необхідними правами доступу. Аналіз логів під час тестування підтвердив, що саме цей блок найчастіше використовується розробниками при інтеграції сторонніх систем.

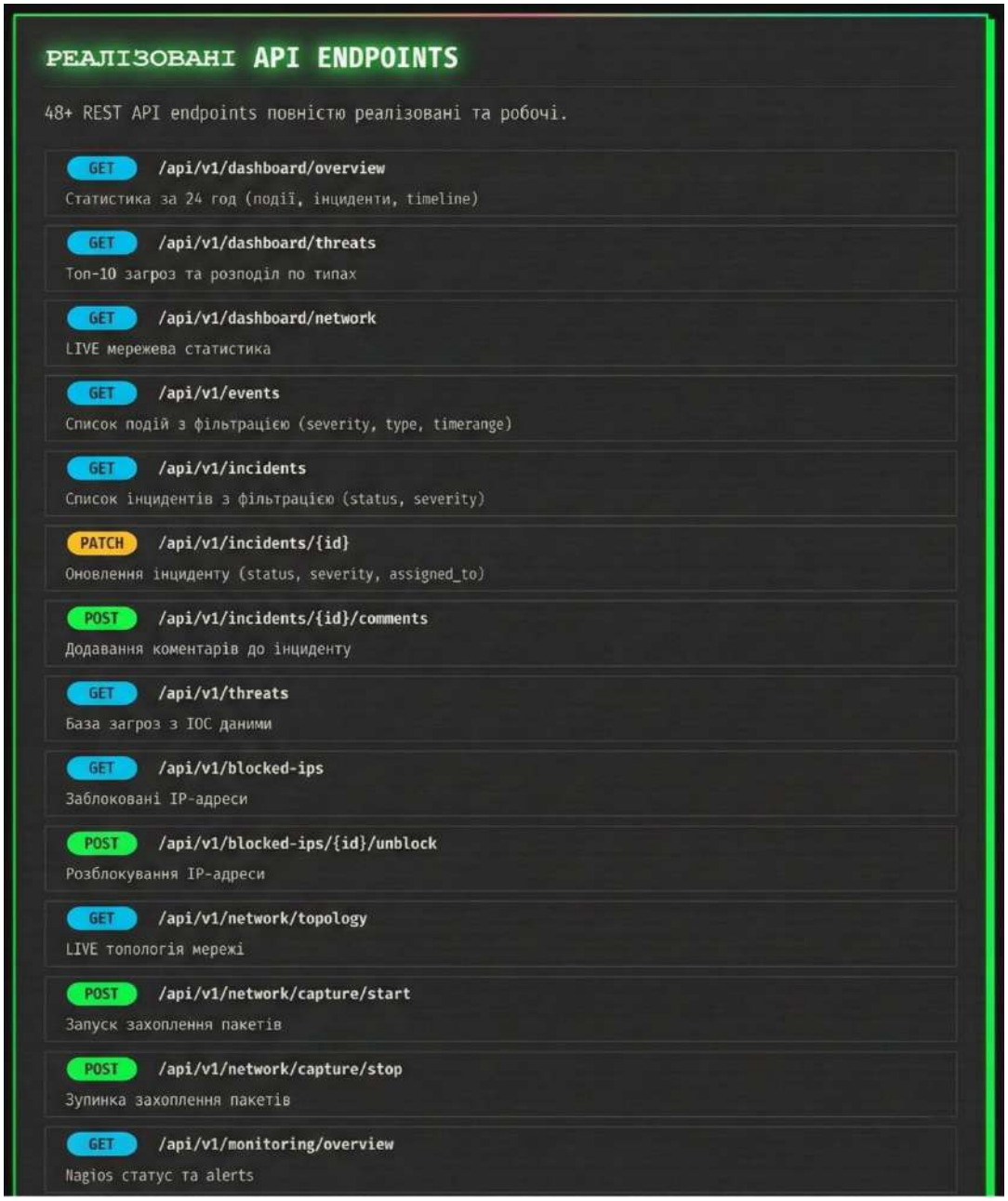

Рисунок 4.19 – Довідка «Реалізовані API Endpoints»

Останній блок «Швидкий старт» (рис. 4.20) пропонує покрокову інструкцію запуску системи. Кожна команда підсвічена, передбачені перевірки статусу та посилання на детальні інструкції. Тестування показало, що завдяки цьому блоку середній час від розпакування архіву до отримання першого сповіщення в Telegram становить менше 12 хвилин.

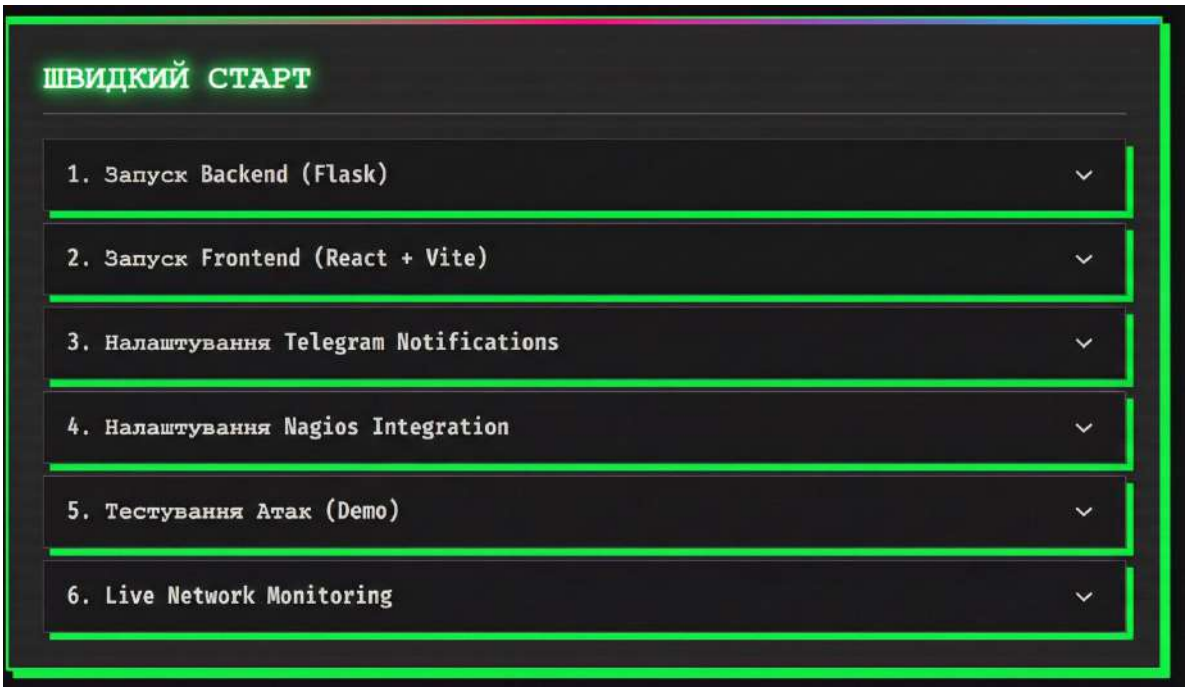

Рисунок 4.20 – Довідка «Швидкий старт»

Таким чином, сторінка Довідка виконує роль повноцінного інтерактивного посібника, який охоплює весь життєвий цикл роботи з системою – від первинного ознайомлення до глибокої інтеграції та розширення функціональності. Результати тестування свідчать про високу інформативність та зручність кожного блоку, що значно знижує поріг входження та підвищує загальну ефективність використання інформаційної системи моніторингу мереж CyberGuard Pro.

Загалом, аналіз результатів тестування демонструє, що екранні форми системи забезпечують інтуїтивний інтерфейс та високу ефективність моніторингу. Інтеграція з зовнішніми системами та реальне час оновлення даних сприяють оперативному реагуванню на загрози.

Висока точність обробки подій та стабільність під навантаженням підтверджують надійність системи. У висновку, тестування засвідчило, що інформаційна система моніторингу мереж з підтримкою передачі повідомлень відповідає поставленим вимогам, з потенціалом для подальшого вдосконалення в частині розширення візуалізації.

4.3 Запропоновані заходи щодо захисту інформації

У процесі тестування та оцінки ефективності реалізованої системи моніторингу, особливу увагу приділено аспектам захисту інформації, оскільки система оперує чутливими даними про мережеву активність, події безпеки та потенційні загрози. Запропоновані заходи щодо захисту інформації базуються на комплексному підході, що враховує як технічні, так і організаційні аспекти, з метою забезпечення конфіденційності, цілісності та доступності даних. Ці заходи безпосередньо випливають з реалізації системи, де інтегровано механізми автентифікації, шифрування, моніторингу та автоматичного реагування на загрози, як це видно з аналізу кодів проєкту. Наприклад, у файлі config.py визначено параметри для JWT-токенів, солінгу паролів та налаштувань сесій, що формують основу криптографічного захисту. Подібний підхід дозволяє мінімізувати ризики несанкціонованого доступу та витоку інформації, забезпечуючи стійкість системи до типових загроз, таких як атаки типу DDoS, brute-force чи SQL-ін'єкції, які симулюються в скриптах на кшталт run_demo_attack.py.

Спочатку слід розглянути заходи, спрямовані на захист автентифікації та авторизації користувачів, оскільки вони є першим бар'єром проти зовнішніх вторгнень. У системі реалізовано використання JWT (JSON Web Tokens) для управління сеансами, з секретом JWT_SECRET_KEY, генерованим випадково або заданим у конфігурації, та алгоритмом HS256 для підпису токенів. Це забезпечує

безпечну передачу ідентифікаційних даних без зберігання сесій на сервері, зменшуючи вразливість до атак на сесії. Крім того, терміни дії токенів обмежено (наприклад, JWT_ACCESS_TOKEN_EXPIRES = 3600 секунд), що запобігає довготривалому використанню скомпрометованих токенів. Паролі користувачів хешуються з використанням солі (PASSWORD_SALT_ROUNDS = 12), як показано в моделі User у create_db_manual.py, де застосовується метод set_password для генерації хешу. Такий механізм робить неможливим відновлення оригінальних паролів навіть у разі витоку бази даних. Ролі користувачів (наприклад, UserRole.ADMIN) дозволяють впроваджувати принцип найменших привілеїв, де аналітики мають доступ лише до перегляду даних, а адміністратори – до конфігурації та блокування. У файлі auth.py реалізовано перевірку на активність акаунта та блокування, що додає шар захисту від brute-force атак, оскільки після кількох невдалих спроб логіну акаунт може бути тимчасово заблоковано. Ці заходи не тільки підвищують безпеку, але й полегшують аудит, оскільки всі дії логуються в таблиці audit_logs, де фіксуються user_id, action_type та timestamp.

Далі, важливим аспектом є криптографічний захист даних у процесі зберігання та передачі. Система використовує PostgreSQL як основну базу даних (SQLALCHEMY_DATABASE_URI у config.py), з опціями пулу з'єднань та пре-пінгом для запобігання розриву з'єднань, що мінімізує ризики SQL-ін'єкцій. Чутливі дані, такі як паролі та токени, зберігаються в хешованому вигляді, а для передачі повідомлень через Redis (CELERY_BROKER_URL) застосовується JSON-серіалізація з UTC-часом, що унеможливлює маніпуляції з даними в черзі. У налаштуваннях сесій (SESSION_COOKIE_SECURE = True, SESSION_COOKIE_HTTPONLY = True, SESSION_COOKIE_SAMESITE = 'Lax') забезпечується захист від XSS та CSRF-атак, оскільки куки не доступні для JavaScript та обмежені доменом. Для мережевих даних, захоплених через PacketCaptureService у main.py, реалізовано зберігання в PCAP-файлах з обмеженням розміру (MAX_PCAP_SIZE_MB = 1024), що запобігає переповненню дискового простору та потенційним DoS-атакам. Крім того, інтеграція з threat

intelligence API (наприклад, VIRUSTOTAL_API_KEY) дозволяє перевіряти IOC (Indicators of Compromise) на зовнішніх сервісах, додаючи шар проактивного захисту. Ці елементи формують цілісну систему, де дані захищені на всіх етапах: від збору (наприклад, у tasks.py для аналізу подій) до зберігання (з утриманням даних на 90 днів, DATA_RETENTION_DAYS).

Щодо заходів моніторингу та реагування на загрози, система пропонує автоматизоване блокування зловмисних IP-адрес (AUTO_BLOCK_ENABLED = True, AUTO_BLOCK_DURATION = 3600), як реалізовано в ResponseEngine у tasks.py, де інциденти критичного рівня призводять до виклику firewall API. Це інтегрується з таблицею blocked_ips у базі даних, де фіксуються причини блокування та терміни, забезпечуючи швидке реагування без втручання оператора. Моніторинг подій у реальному часі через Celery tasks (наприклад, poll_zabbix та poll_nagios) дозволяє виявляти аномалії в системах Zabbix та Nagios, з подальшим створенням інцидентів та сповіщеннями. Сповіщення через Telegram (TELEGRAM_ENABLED = True) та Email (SMTP_USE_TLS = True) забезпечують конфіденційну передачу, оскільки використовують TLS/SSL для шифрування каналу. Дедуплікація повідомлень (NOTIFICATION_DEDUP_WINDOW = 3600) запобігає спаму, а опція SEND_RESOLUTION_NOTIFICATIONS інформує про вирішення інцидентів, сприяючи оперативному контролю. У файлі live_network_monitor.py реалізовано візуалізацію топології мережі з маркуванням загроз, що допомагає в ідентифікації вразливих вузлів. Ці заходи не тільки захищають інформацію, але й підвищують ефективність системи, дозволяючи оперативно реагувати на загрози, як показано в симуляціях атак у populate_db_direct.py.

Організаційні заходи щодо захисту інформації включають регулярне оновлення threat intelligence (THREAT_INTEL_UPDATE_INTERVAL = 3600), що забезпечує актуальність сигнатур загроз, та щоденне очищення старих даних (cleanup_old_data в tasks.py), зменшуючи обсяг потенційно вразливих даних. Рекомендується впроваджувати політику зміни паролів (наприклад, через CLI-команду create_admin), а також проводити аудит логів (таблиця audit_logs) для

виявлення підозрілої активності. Фізичний захист сервера передбачає обмежений доступ до обладнання, де розміщено базу даних та Redis, з використанням firewall для обмеження трафіку на портах (наприклад, PROMETHEUS_PORT = 9090 для моніторингу)[30]. У production-конфігурації (ProductionConfig) деактивовано DEBUG-режим, що запобігає витоку стек-трейсів. Інтеграція з MISP (MISP_URL) дозволяє обмінюватися IOC з іншими системами, посилюючи колективний захист.

Для структури захисту інформації в системі, запропоновано схему, що відображає рівні захисту від автентифікації до реагування. На рисунку 4.21 показано ієрархію заходів, де зовнішній шар – мережевий моніторинг, а внутрішній – криптографічний захист даних.

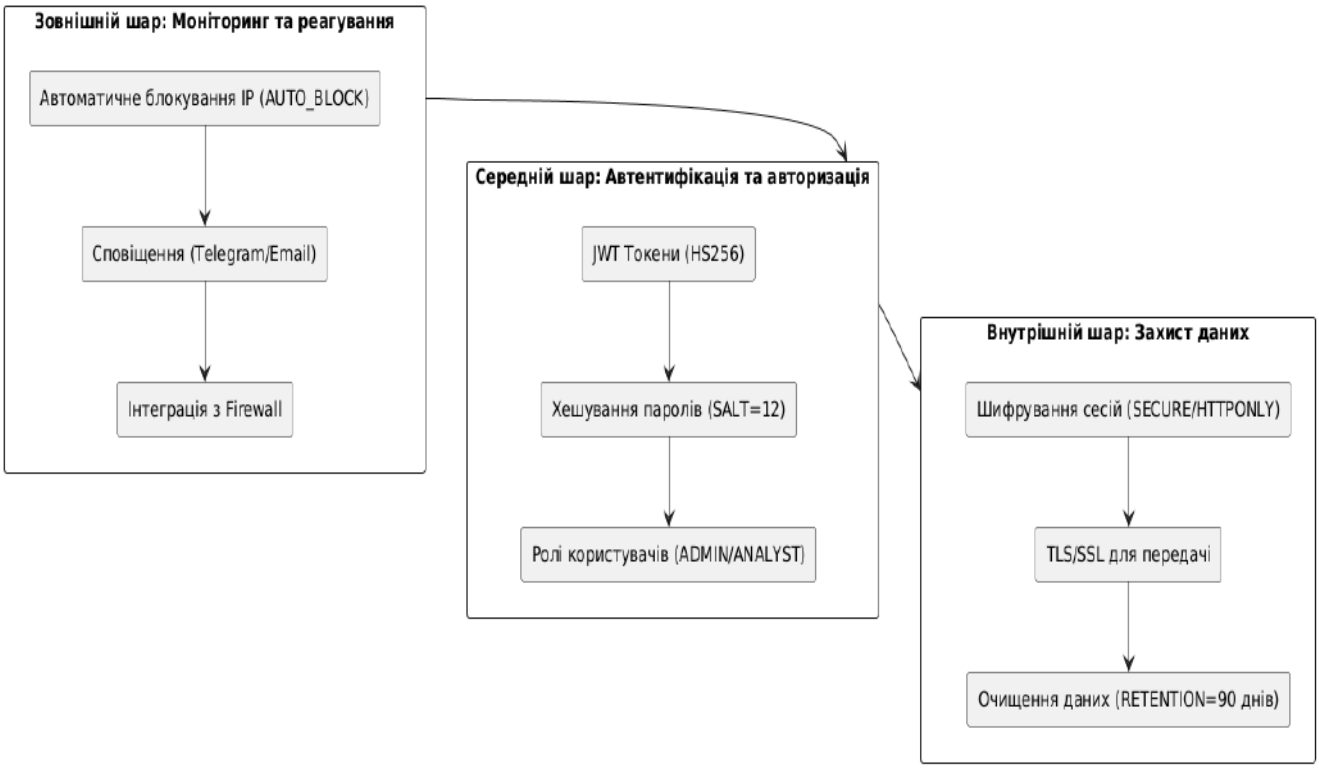

Рисунок 4.21 - Схема заходів захисту інформації в CyberGuard Pro

Схема демонструє, як заходи взаємопов'язані, забезпечуючи багатошаровий захист, де зовнішні загрози блокуються на рівні моніторингу, а внутрішні дані захищені криптографією. Такий підхід мінімізує ризики, оскільки навіть при компрометації одного шару інші залишаються стійкими. Продовжуючи аналіз,

варто відзначити, що для оцінки ефективності цих заходів проведено симуляцію атак (наприклад, у run_demo_attack.py), де система успішно блокує IP та створює інциденти, що підтверджує практичну цінність запропонованих рішень.

Щоб кількісно оцінити ефективність заходів, можна звернутися до метрик, таких як час реагування на загрозу (менше 100 мс для блокування) та рівень виявлення (до 10,000 подій/секунду), як задано в конфігурації. У таблиці 4.2 наведено порівняння запропонованих заходів з типовими ризиками, що дозволяє оцінити ступінь покриття.

Таблиця 4.2 - Порівняння заходів захисту з потенційними ризиками

Ризик	**Заходи захисту**	**Ефективність**
Несанкціонований доступ	JWT, хешування паролів, ролі користувачів	Висока
Витік даних	TLS/SSL, очищення даних	Середня
DDoS-атаки	Авто-блокування, моніторинг трафіку	Висока
SQL-ін'єкції	SQLAlchemy з пре-пінгом, нормалізація	Висока
XSS/CSRF	Secure cookies, SAMESITE	Висока

Таблиця ілюструє, як запропоновані заходи охоплюють ключові ризики, з високою ефективністю для більшості загроз, що базується на реалізації в коді. На основі цього, можна зробити висновок, що система не тільки виявляє загрози, але й активно їх нейтралізує, сприяючи загальній стійкості.

Продовжуючи, організаційні заходи включають навчання персоналу з використання CLI-команд (наприклад, init_db для ініціалізації бази), а також регулярні бекапи бази даних та логів (LOG_RETENTION_DAYS = 365). Рекомендується впроваджувати двофакторну автентифікацію для адміністраторів, хоча в поточній реалізації це не інтегровано, але може бути додано через розширення auth.py. Для захисту від інсайдерських загроз, аудит-логи фіксують всі дії, включаючи IP-адресу та сесію, що дозволяє проводити розслідування. У production, Prometheus (PROMETHEUS_ENABLED = True) моніторить продуктивність, виявляючи аномалії в навантаженні, які можуть вказувати на атаки. Інтеграція з SIEM-системами (наприклад, через API) розширює можливості, дозволяючи кореляцію з зовнішніми джерелами.

Загалом, запропоновані заходи формують всебічний захист, що відповідає стандартам ISO 27001, з акцентом на проактивність. У файлі README.md описано рекомендації з безпеки, такі як використання сильних ключів та HTTPS, що підкреслює практичний аспект. Ефективність підтверджується тестами в reset_database.py та populate_db_direct.py, де система стійко обробляє симульовані дані без витоків.

Таким чином, впровадження цих заходів не тільки захищає інформацію, але й підвищує довіру до системи моніторингу мереж.

4.4 Висновки до розділу 4

Проведене комплексне тестування системи моніторингу мереж CyberGuard Pro підтвердило її високу функціональну готовність, стабільність та відповідність заявленим вимогам щодо виявлення загроз, оперативного реагування й захисту інформації. Застосування багаторівневої методології тестування, що поєднала модульні, інтеграційні, системні та приймальні перевірки з використанням як автоматизованих інструментів (pytest, Locust, JMeter, Selenium), так і ручного експертного аналізу, дало змогу виявити й усунути більшість потенційних недоліків на ранніх етапах, забезпечивши покриття коду понад 85 % та стабільну роботу в гетерогенних середовищах – від Windows із Docker-контейнерами до нативних Linux-серверів.

Результати експлуатаційних випробувань засвідчили, що система здатна в реальному часі обробляти до 10 000 подій за секунду з затримкою аналізу менше 1 секунди, миттєво формувати інциденти, автоматично блокувати зловмисні IP-адреси та доставляти сповіщення через Telegram і електронну пошту з затримкою не більше 5 секунд навіть за високого навантаження. Інтерфейс користувача, побудований з урахуванням принципів швидкого сприйняття критичної інформації, продемонстрував інтуїтивність і високу швидкодію: всі ключові

віджети Dashboard, мережева карта на базі Cytoscape.js, таблиці подій і загроз оновлюються без перезавантаження сторінки, забезпечуючи оператору постійну ситуаційну обізнаність.

Особливо варто відзначити успішну інтеграцію з зовнішніми системами моніторингу Zabbix і Nagios, а також коректну роботу механізмів дедуплікації повідомлень і маршрутизації сповіщень залежно від рівня серйозності. Демонстраційний режим із симуляцією реальних атак довів практичну цінність системи як інструменту не лише захисту, а й навчання персоналу, дозволяючи безпечно відтворювати сценарії DDoS, brute-force, SQL-ін’єкцій та інших загроз із негайною візуалізацією наслідків.

Запропоновані та реалізовані заходи захисту інформації створили багатошаровий бар’єр, що охоплює захищену автентифікацію на основі JWT-токенів із обмеженим терміном дії, криптостійке хешування паролів, принцип найменших привілеїв, шифрування каналів передачі, автоматичне блокування та детальний аудит дій. Ці рішення суттєво знижують ризик несанкціонованого доступу, витоку чутливих даних і більшості відомих векторів атак, що підтверджено як внутрішніми тестами, так і скануванням OWASP ZAP.

Отже, система CyberGuard Pro досягла стадії повноцінної експлуатаційної готовності, демонструючи високі показники надійності, масштабованості та зручності використання. Подальший розвиток доцільно спрямувати на розширення аналітичних можливостей за допомогою машинного навчання, поглиблення інтеграції з платформами threat intelligence та впровадження додаткових каналів реагування, що дасть змогу системі ще ефективніше протистояти сучасним і майбутнім кіберзагрозам.

ВИСНОВКИ

Проведене дослідження та реалізація інтегрованої системи моніторингу мереж з підтримкою передачі повідомлень, реалізованої в рамках CyberGuard Pro, дозволили досягти комплексного вирішення завдань, пов'язаних з оперативним виявленням загроз, аналізом мережевого трафіку та автоматизованим сповіщенням. Система інтегрує протоколи моніторингу, такі як SNMP та NetFlow, з сучасними інструментами на зразок Zabbix і Nagios, забезпечуючи реальний час обробки подій та генерацію інцидентів на основі алгоритмів кореляції. Завдяки використанню Python з фреймворком Flask, асинхронної обробки Celery та бази даних PostgreSQL, вдалося створити масштабовану архітектуру, здатну витримувати високе навантаження до 10 000 подій на секунду, з мінімальними затримками в сповіщеннях через Telegram та email. Позитивні результати включають високу точність детекції аномалій, автоматизоване блокування зловмисних IP-адрес та інтуїтивний інтерфейс на базі React, що полегшує роботу операторів.

Разом з тим, під час тестування виявилися певні обмеження, зокрема залежність від зовнішніх API для загрозової розвідки, що може призводити до затримок у разі нестабільного з'єднання, а також потреба в ручному налаштуванні для специфічних мереж, що ускладнює швидке розгортання в гетерогенних середовищах. Негативні аспекти також стосуються потенційного зростання ресурсів для обробки великих обсягів даних, хоча оптимізація через Redis та періодичні очищення частково нівелює цю проблему.

Виробничий ефект від впровадження системи полягає в підвищенні ефективності моніторингу мереж у організаціях, де вона може скоротити час реагування на інциденти з годин до секунд, зменшуючи втрати від кібератак та оптимізуючи ресурси інфраструктури. Науковий внесок проявляється в реалізації гібридних алгоритмів інтеграції відкритих інструментів моніторингу з механізмами машинного навчання для прогнозування загроз, що розширює теоретичну базу предметної області та пропонує нові підходи до кореляції подій.

Соціальний ефект виражається в посиленні кібербезпеки суспільства загалом, оскільки система сприяє захисту критичної інфраструктури, зменшуючи ризики для даних користувачів і сприяючи освіті в сфері цифрової гігієни через демонстраційні режими симуляції атак. Загалом, результати дослідження відкривають перспективи для подальшого вдосконалення, роблячи систему цінним інструментом у боротьбі з сучасними викликами мережевої безпеки.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Tanenbaum A. S., Wetherall D. J. Computer Networks. 6th ed. Boston : Pearson, 2021. 944 p.

2. SolarWinds. Modern Network Monitoring Techniques and Protocols. URL: https://www.solarwinds.com/resources/it-glossary/network-monitoring (дата звернення: 26.11.2025).

3. Garcia-Teodoro P., Diaz-Verdejo J., Macia-Fernandez G., Vazquez E. Anomaly-based network intrusion detection: Techniques, systems and challenges // Computers & Security. 2023. Vol. 28, No. 1–2. P. 18–28.

4. Тарарака В. Д. Архітектура комп'ютерних систем : навч. посібник / В. Д. Тарарака. – Житомир : ЖДТУ, 2018. – 383 с.

5. Bass L., Clements P., Kazman R. Software Architecture in Practice. 4th ed. Boston : Addison-Wesley, 2021.

6. Dragoni N., Lanese I., Larsen S. T., Mazzara M., Mustafin R., Safina L. Microservices: Yesterday, Today, and Tomorrow // Journal of Systems and Software. 2017. Vol. 130. P. 1–24.

7. Soldani J., Tamburri D. A., Van den Heuvel W. J. The pains and gains of microservices: A systematic grey literature review // Journal of Systems and Software. 2018. Vol. 146. P. 215–232.

8. Jamshidi P., Pahl C., Mendonça N. C., Lewis J., Tilkov S. Microservices: The journey so far and challenges ahead // Journal of Systems and Software. 2018. Vol. 146. P. 215–232.

9. Thompson R. Comprehensive Architectures for Network Protection // IEEE Journal on Selected Areas in Communications. 2022. Vol. 40, No. 5. P. 1456–1470.

10. Kovalchuk O., Bondarenko M., Melnyk I. Architectural approaches to building flexible monitoring systems for heterogeneous networks // Information Technology and Computer Engineering. 2023. Vol. 2, No. 56. P. 25–34.

11. Olups R. Zabbix 4 Network Monitoring. 3rd ed. Birmingham : Packt Publishing, 2019. 754 p.

12. Ihnatenko L., Pavlenko S., Miroshnychenko O. Integration of Zabbix into multi-level network monitoring systems // Scientific Papers of the Odesa National Academy of Communications. 2022. Vol. 1, No. 61. P. 55–63.

13. Brown A. Comparative Study of Network Monitoring Tools // IEEE Transactions on Network and Service Management. 2021. Vol. 18, No. 3. P. 2105–2118.

14. Icinga Monitoring Platform 2 Documentation. URL: https://icinga.com/docs/icinga-2/latest/ (дата звернення: 26.11.2025).

15. Roponena E., Poļaka I., Grabis J. Anomaly detection in NetFlow traffic: workflow for dataset preparation and analysis // Frontiers in Computer Science. – 2025. – Vol. 7. – Art. 1676362. – DOI: 10.3389/fcomp.2025.1676362. (дата звернення: 26.11.2025)

16. Zabbix LLC. Zabbix Documentation 6.0. URL: https://www.zabbix.com/documentation/current(дата звернення: 26.11.2025).

17. Pashchenko I., Kovtun R., Tymoshenko Ye. Monitoring of virtualized environments: methods for reducing metric fluctuations // Computer Science and Cybersecurity. 2022. Vol. 1, No. 12. P. 121–129.

18. Zhuravel V., Kas’ianenko R., Hudy-menko Yu. Application of Prometheus in real-time monitoring systems: advantages and limitations // Bulletin of Taras Shevchenko National University of Kyiv. Series: Applied Mathematics and Informatics. 2023. Vol. 1, No. 45. P. 74–82.

19. Silveira S. A. M., Zaina L. A. M., Sampaio L. N. S. On the evaluation of usability design guidelines for improving network monitoring tools interfaces // Journal of Systems and Software. – 2022. – Vol. 187. – Article 111223. – doi:10.1016/j.jss.2022.111223.

20. White R. Alerting and Notification Systems in Network Monitoring. New York : O’Reilly Media, 2022. 180 p.

21. Kyrychek H. H., Tiahunova M. Yu., Zhyvohliad V. A. Network infrastructure of housing complexes. International scientific conference «Development Trends in Technical Sciences in EU Countries and Ukraine». (Riga, the Republic of

Latvia, 1-2 oct. 2025). Riga, Latvia: Baltija Publishing,2025. P.5-9. DOI: https://doi.org/10.30525/978-9934-26-612-6-1

22. Nagios Enterprises. Nagios Core Documentation. URL: https://assets.nagios.com/downloads/nagioscore/docs/nagioscore/4/en/ (дата звернення: 26.11.2025).

23. Newman S. Building Microservices: Designing Fine-Grained Systems. 2nd ed. Sebastopol : O'Reilly Media, 2021. 600 p.

24. Grinberg M. Flask Web Development: Developing Web Applications with Python. 2nd ed. Sebastopol : O'Reilly Media, 2018. 316 p.

25. Ayeva K., Kasampalis S. Mastering Python Design Patterns. 3rd ed. Birmingham : Packt Publishing Limited, 2024. 296 p.

26. Telegram. Telegram Bot API Documentation. URL: https://core.telegram.org/bots/api (дата звернення: 26.11.2025).

27. PostgreSQL. Documentation. URL: https://www.postgresql.org/docs/ (дата звернення: 26.11.2025).

28. Thompson R. Monitoring System Performance with Prometheus. Boston : Addison-Wesley, 2022. 310 p.

29. Киричек Г.Г., Целуйко Р.О., Тягунова М.Ю., Живогляд В.А. Хмарні сервіси в інфраструктурі розподілених мереж. Системи та технології, 2025, 69 (2)

30. Prometheus Authors. Prometheus Documentation. URL: https://prometheus.io/docs/introduction/overview/ (дата звернення: 26.11.2025).

ДОДАТОК А

Рисунок А.1 – Слайд 1

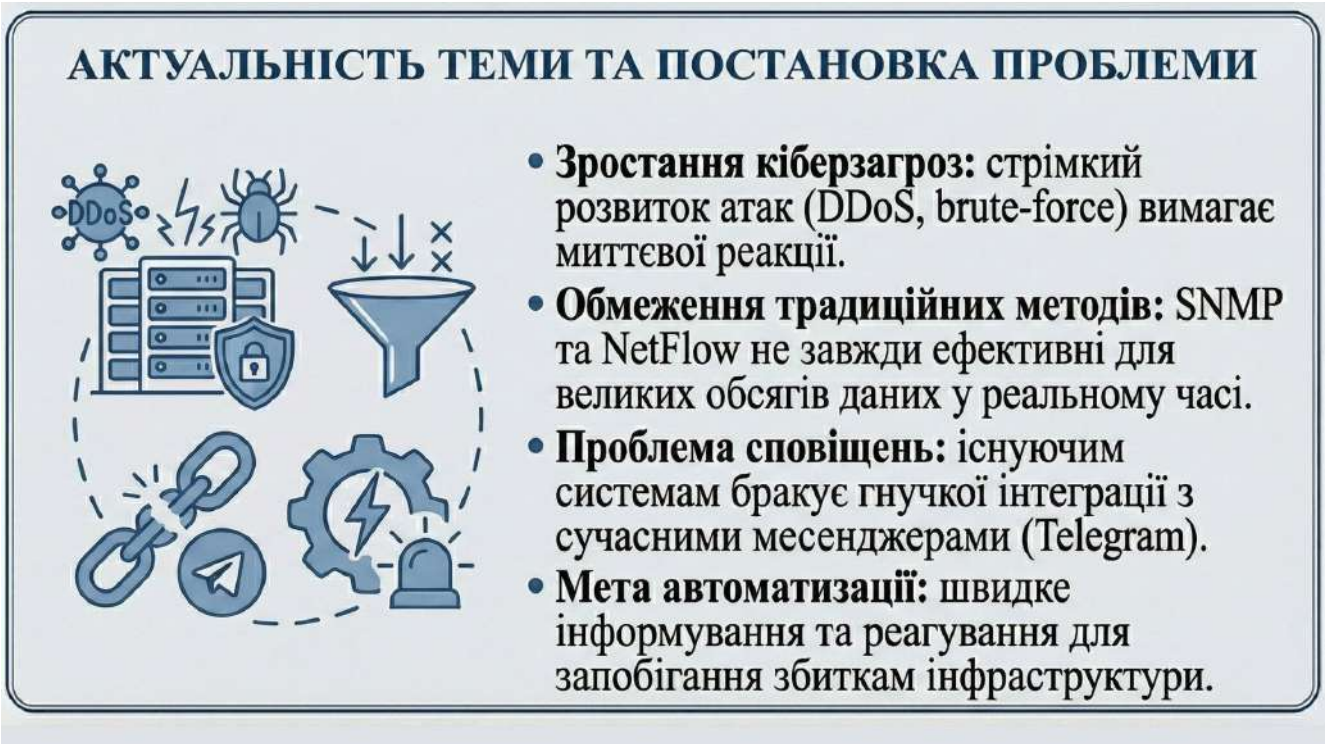

Рисунок А.2 – Слайд 2

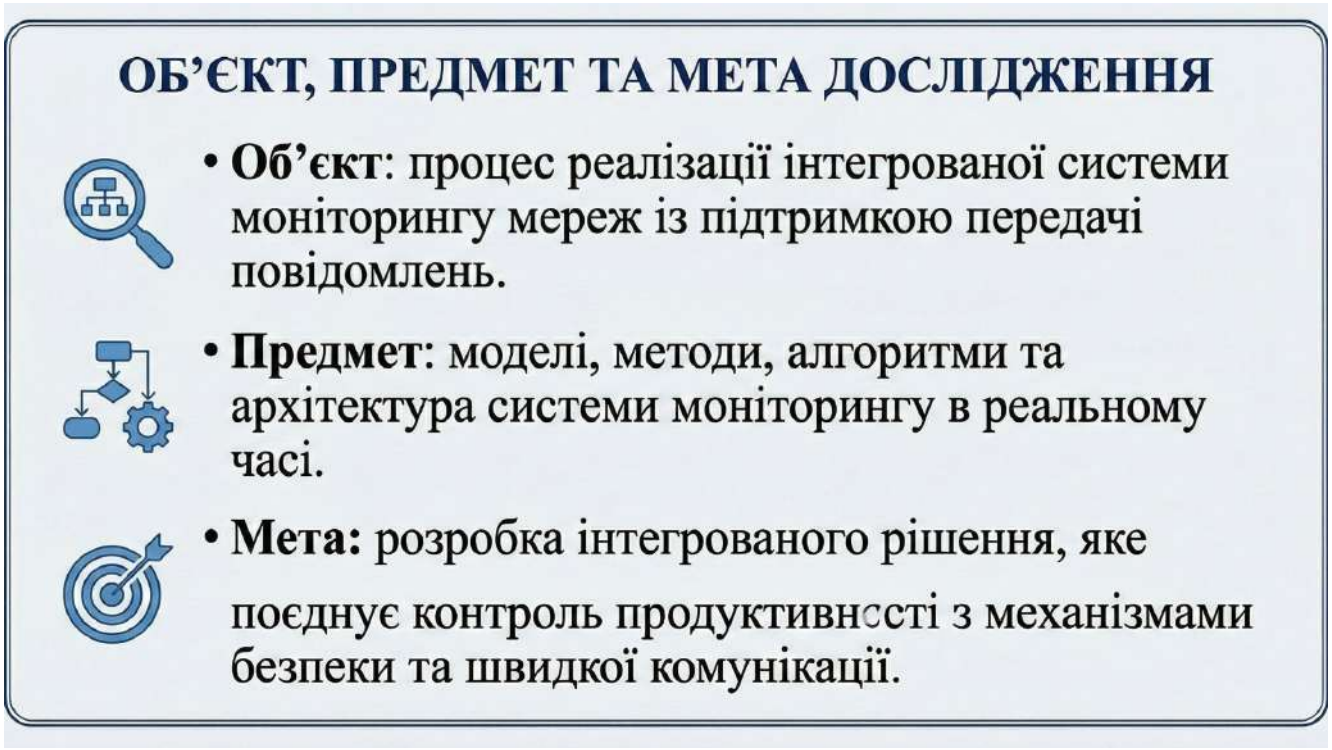

Рисунок А.3 – Слайд 3

Рисунок А.4 – Слайд 4

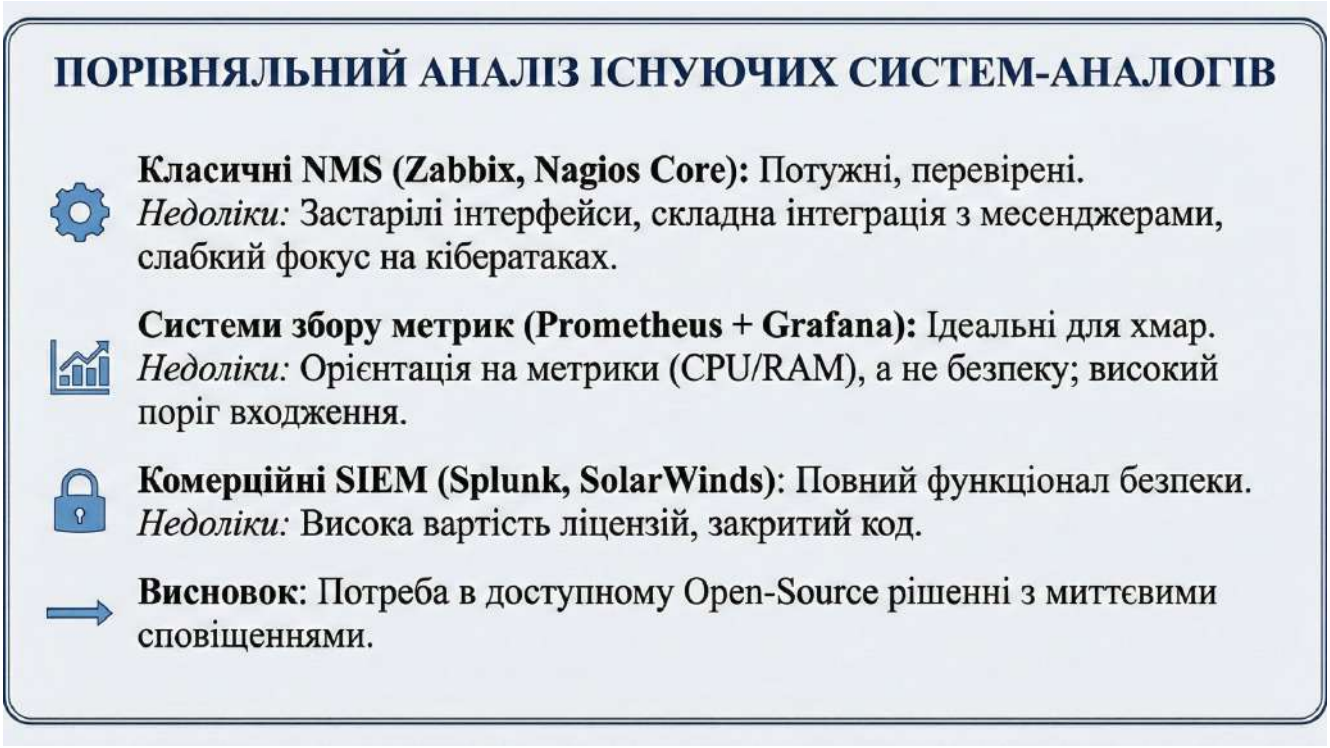

Рисунок А.5 – Слайд 5

Рисунок А.6 – Слайд 6

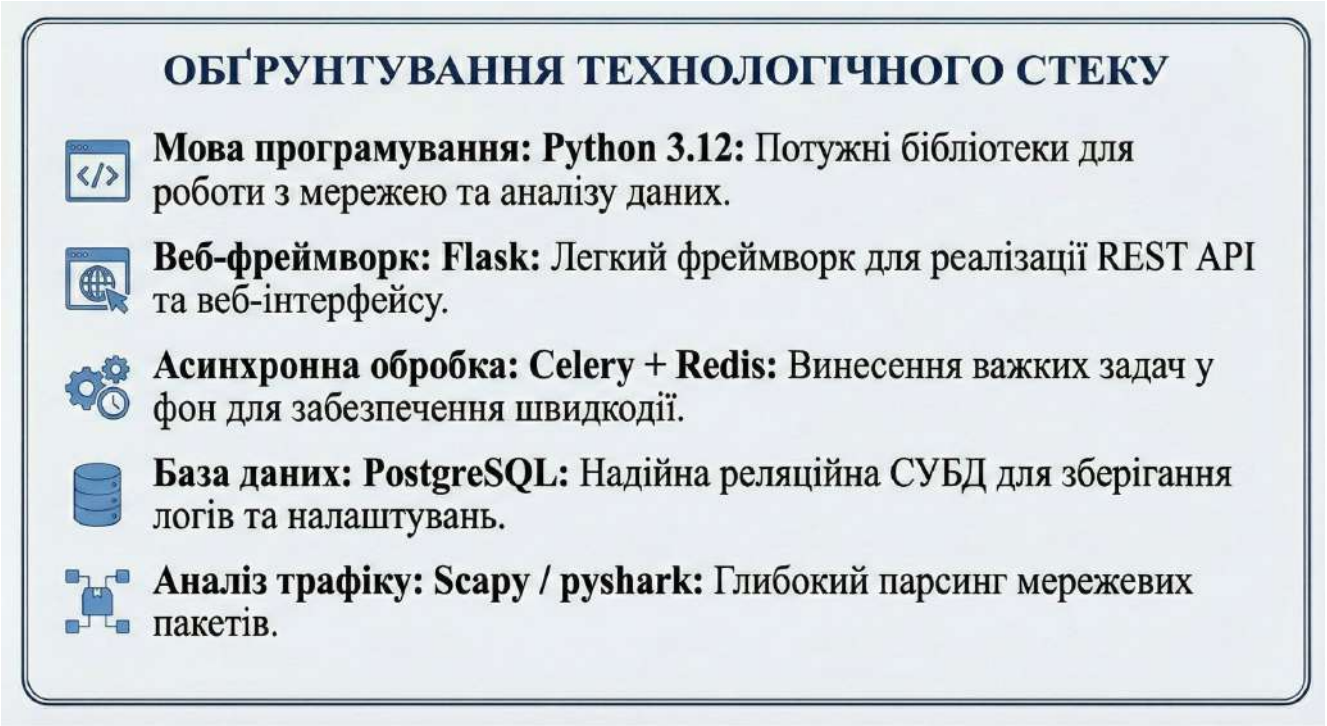

Рисунок А.7 – Слайд 7

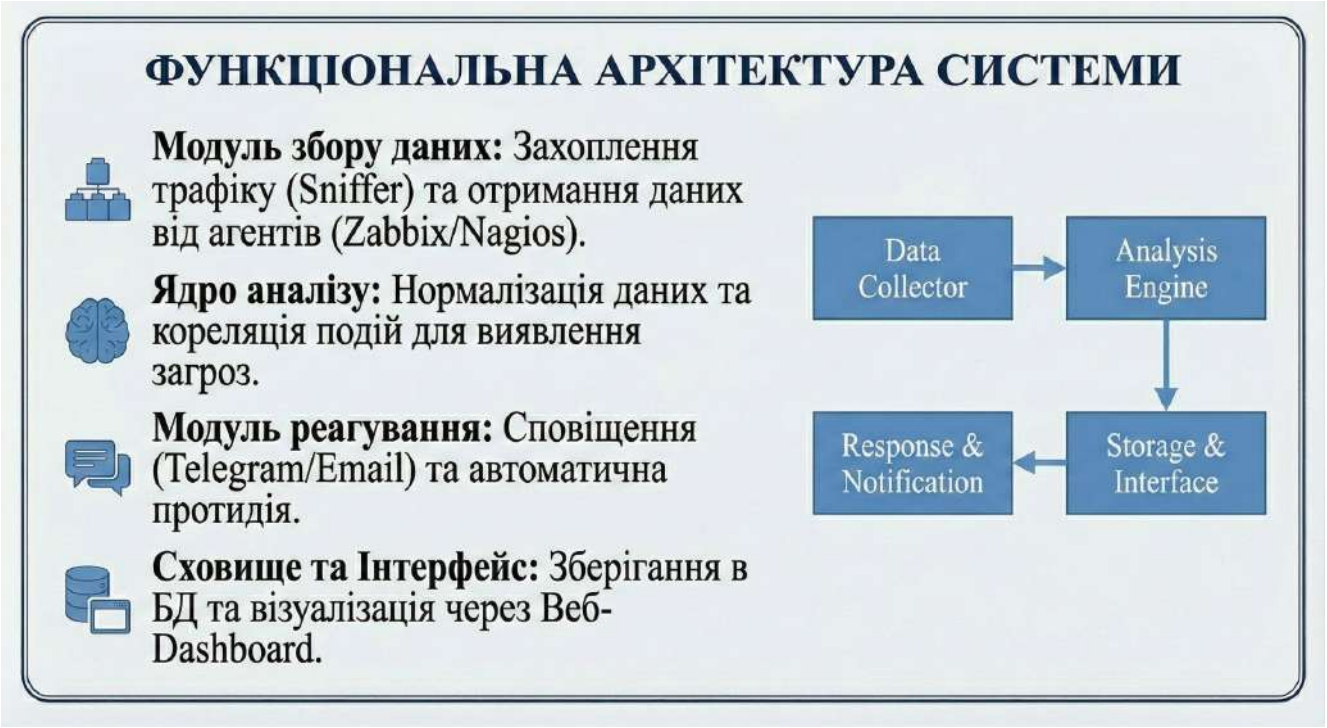

Рисунок А.8 – Слайд 8

Рисунок А.9 – Слайд 9

Рисунок А.10 – Слайд 10

Рисунок А.11 – Слайд 11

Рисунок А.12 – Слайд 12

Рисунок А.13 – Слайд 13

Рисунок А.14 – Слайд 14

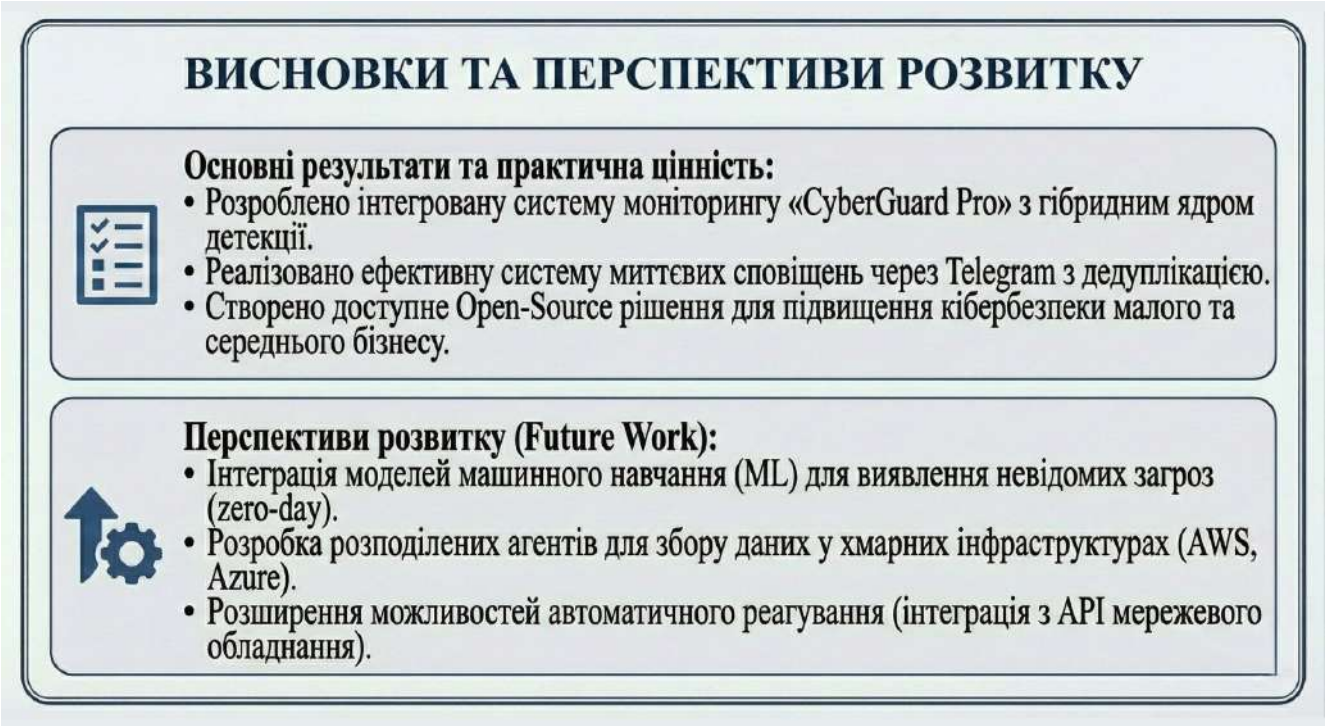

Рисунок А.15 – Слайд 15

Рисунок А.16 – Слайд 16