

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

**КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «БАКАЛАВР»**

**РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ЗБОРУ ТА АНАЛІЗУ ІНЦИДЕНТІВ З
ВІДКРИТИХ ДЖЕРЕЛ, ПОВ'ЯЗАНИХ ІЗ НЕДОСТАТНЬОЮ ЯКІСТЮ
ПЗ**

**DEVELOPMENT OF AN APPLICATION FOR COLLECTING AND
ANALYZING OPEN SOURCES INCIDENTS RELATED TO POOR
SOFTWARE QUALITY**

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

**Виконав: здобувач вищої освіти
групи ІПЗ-43
Пагуба Дмитро Васильович**

**Керівник:
Гордєєв Олександр Олександрович**

Кваліфікаційну роботу
допущено до захисту
«__» _____ 20__р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Наталія Миколаївна

Луцьк – 2026 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних та інформаційних технологій

Кафедра інженерії програмного забезпечення

Ступінь вищої освіти бакалавр

Галузь знань: 12 «Інформаційні технології»

Спеціальність: 121 «Інженерія програмного забезпечення»

Освітня програма: «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ

Завідувач кафедри

«__» _____ 20__ р.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧУ ВИЩОЇ ОСВІТИ

Пагубі Дмитру Васильовичу

(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка застосунку для збору та аналізу інцидентів з відкритих джерел, пов'язаних із недостатньою якістю ПЗ»

Керівник роботи: д. т. н. професор Гордєєв О. О.

затверджені наказом закладу вищої освіти від «31» грудня 2025 р. № 541/01-02

2. Строк подання здобувачем вищої освіти кваліфікаційної роботи бакалавра «29» травня 2026 р.

3. Вихідні дані до роботи: предметна область автоматизованого збору та аналізу інцидентів програмного забезпечення з відкритих джерел (RSS/Atom, Hacker News API), технології розробки програмного забезпечення (Python, FastAPI, React), засоби роботи з базами даних (PostgreSQL, SQLAlchemy), вимоги до інформаційних систем та забезпечення інформаційної безпеки.

4. Зміст розрахунково-пояснювальної записки (перелік питань, що потрібно розробити): аналіз сучасного стану проблеми моніторингу інцидентів програмного забезпечення, постановка завдання, формування вимог до розроблюваної інформаційної системи, обґрунтування вибору технологій та засобів розробки, проєктування системи з використанням UML-нотації, практична реалізація об'єкта проєктування, розробка структури бази даних, обґрунтування принципів захисту інформаційно-комп'ютерної системи, тестування та налагодження програмного забезпечення.

5. Перелік графічного матеріалу: 13 рисунків, 15 лістингів, 6 таблиці.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз предметної області	<i>Гордєєв О. О.</i>		
Специфікація вимог до розробленої системи	<i>Гордєєв О. О.</i>		
Розробка об'єкта проектування	<i>Гордєєв О. О.</i>		
Нормоконтроль	<i>Суринович О. М.</i>		
Гарант ОП	<i>Ліщина Н. М.</i>		
Показник запозичень тексту	%		
Академічна доброчесність	<i>Гордєєв О. О.</i>		

7. Дата видачі завдання «3» лютого 2026 р.

№	Назва етапів кваліфікаційної роботи бакалавра	Строк виконання етапів роботи	Примітка
1	Огляд літературних джерел по темі кваліфікаційної роботи бакалавра	до 14.02.2026 р.	
2	Аналіз проблеми розробки та впровадження об'єкту проектування	до 03.03.2026 р.	
3	Обґрунтування вибору шляхів, технологій і засобів вирішення поставленого завдання	до 14.03.2026 р.	
4	Розробка функціонально-структурної схеми роботи об'єкта проектування та проектування бази даних	до 14.04.2026 р.	
5	Практична реалізація об'єкта проектування та розробка бази даних	до 20.04.2026 р.	
6	Тестування та налагодження об'єкта проектування	до 04.05.2026 р.	
7	Здача чистового варіанту кваліфікаційної роботи бакалавра на кафедру	до 29.05.2026 р.	

Здобувач вищої освіти _____

Дмитро ПАГУБА

Керівник кваліфікаційної роботи _____

Олександр ГОРДЄЄВ

АНОТАЦІЯ

Пагуба Д. М. Розробка застосунку для збору та аналізу інцидентів з відкритих джерел, пов'язаних із недостатньою якістю ПЗ. Рукопис. Кваліфікаційна робота бакалавра ОП «Інженерія програмного забезпечення» спеціальності «Інженерія програмного забезпечення». Луцький національний технічний університет. Луцьк, 2026.

Кваліфікаційна робота бакалавра складається зі вступу, трьох розділів, висновків, списку використаних джерел.

У першому розділі проведено дослідження предметної області, пов'язаної з автоматизованим збором та аналізом інцидентів програмного забезпечення з відкритих джерел, проаналізовано існуючі рішення моніторингу безпеки та агрегації новин ІТ-галузі, виявлено їхні сильні та слабкі сторони, а також обґрунтовано необхідність створення власної системи із підтримкою гетерогенних джерел і автоматичної класифікації. У другому розділі сформовано функціональні та нефункціональні вимоги до програмного продукту, здійснено проєктування системи із використанням UML-нотації та обґрунтовано вибір інструментальних засобів, методів і технологій розробки. У третьому розділі описано процес реалізації інформаційної системи на основі трирівневої архітектури, розроблено реляційну структуру бази даних із використанням ORM, реалізовано асинхронний паралельний обхід джерел із механізмами дедуплікації та класифікації за ключовими словами, розглянуто питання забезпечення стійкості системи до помилок мережі, а також наведено результати тестування й налагодження програмного забезпечення. У висновках підсумовано отримані результати, підтверджено досягнення мети роботи та повне виконання поставлених завдань.

Ключові слова: інформаційна система, моніторинг інцидентів, збір даних, парсинг, дедуплікація, класифікація, асинхронна обробка, REST API, реляційна база даних, Python, FastAPI, React, PostgreSQL, SQLAlchemy, aiohttp.

ABSTRACT

Pahuba D. V. Development of an Application for Collecting and Analyzing Open Source Incidents Related to Poor Software Quality. Manuscript. Qualification work for bachelor's degree in the educational program "Software Engineering", specialty "Software Engineering". Lutsk National Technical University. Lutsk, 2026.

The bachelor's qualification work consists of an introduction, three chapters, conclusions, and a list of references.

In the first chapter, a study of the problem domain related to the automated collection and analysis of software incidents from open sources was conducted. Existing security monitoring solutions and IT news aggregators were analyzed, their strengths and weaknesses were identified, and the necessity of developing a new system with support for heterogeneous sources and automatic classification was substantiated. In the second chapter, functional and non-functional requirements for the software product were defined, the system was designed using UML notation, and the choice of tools, methods, and development technologies was justified. The third chapter describes the implementation of the information system based on a three-tier architecture, the development of a relational database structure using ORM, the implementation of asynchronous parallel traversal of sources with deduplication and keyword-based classification mechanisms, the consideration of system resilience to network errors, as well as the results of software testing and debugging. The conclusions summarize the results obtained, confirm the achievement of the research objective, and verify the complete fulfillment of all defined tasks.

Keywords: information system, incident monitoring, data collection, parsing, deduplication, classification, asynchronous processing, REST API, relational database, Python, FastAPI, React, PostgreSQL, SQLAlchemy, aiohttp.

ЗМІСТ

ВСТУП.....	7
РОЗДІЛ 1 АНАЛІЗ МЕТОДІВ РОЗРОБКИ ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПОСТАНОВКА ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ	10
1.1 Аналіз сучасного стану проблеми.....	10
1.2 Постановка завдання на кваліфікаційну роботу бакалавра.....	12
РОЗДІЛ 2 СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ	15
2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення	15
2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання.....	21
РОЗДІЛ 3 РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ЗБОРУ ТА АНАЛІЗУ ІНЦИДЕНТІВ З ВІДКРИТИХ ДЖЕРЕЛ, ПОВ'ЯЗАНИХ ІЗ НЕДОСТАТНЬОЮ ЯКІСТЮ ПЗ	30
3.1 Практична реалізація об'єкта проєктування.....	30
3.2 Тестування та налагодження інформаційно-комп'ютерної системи	37
3.3 Розробка бази даних	39
3.4 Розробка REST API серверної частини	41
3.5 Аналіз результатів роботи системи	44
ВИСНОВКИ.....	48
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	50

ВСТУП

Актуальність теми. У сучасних умовах стрімкого розвитку індустрії програмного забезпечення однією з ключових проблем є забезпечення якості програмних продуктів. Зростання складності систем, скорочення циклів розробки та підвищення вимог до функціональності неминуче збільшують ризик виникнення дефектів і збоїв. За оцінками міжнародних дослідницьких організацій, помилки у програмному забезпеченні щорічно спричиняють економічні збитки, що вимірюються сотнями мільярдів доларів, а їх своєчасне виявлення та систематизація є одним із пріоритетних завдань галузі [1].

На практиці значна частина відомостей про реальні інциденти, пов'язані з недостатньою якістю програмних продуктів, міститься у відкритих інтернет-джерелах – новинних ресурсах, спеціалізованих форумах, системах відстеження помилок та соціальних платформах. Попри свою практичну цінність, ця інформація здебільшого лишається неструктурованою та розпорошеною між різнорідними джерелами, що унеможлиблює її ефективне використання для системного аналізу тенденцій у сфері якості ПЗ [2].

Існуючі програмні рішення у даній сфері, як правило, орієнтовані на вузькоспеціалізовані корпоративні середовища та не передбачають автоматизованої роботи з публічно доступними даними. Більшість із них потребують значних ресурсів на впровадження та налаштування, а також не забезпечують комплексного підходу до збору, фільтрації та аналізу інформації з гетерогенних відкритих джерел. Крім того, такі рішення позбавлені зручного візуального інтерфейсу для аналітики, що ускладнює сприйняття результатів і унеможлиблює оперативне прийняття рішень на їх основі [3].

Особливо актуальною є проблема дедуплікації зібраних даних. Одна і та сама подія може висвітлюватися одночасно в багатьох джерелах із незначними відмінностями у формулюванні, що призводить до накопичення дублікатів та спотворення результатів аналізу. Відсутність ефективних механізмів фільтрації

суттєво знижує практичну цінність зібраних відомостей та ускладнює формування достовірної картини стану якості програмного забезпечення.

Таким чином, виникає необхідність у розробці спеціалізованого веб-додатку, який би забезпечував автоматизований збір даних про інциденти ПЗ з відкритих джерел, їх структуроване зберігання, фільтрацію дублікатів та наочне представлення аналітичних результатів у вигляді інтерактивних графіків і діаграм. Використання сучасних технологій, зокрема мови програмування Python для серверної частини та сучасних JavaScript-фреймворків для клієнтської, дозволяє створити повноцінний, надійний та зручний у використанні програмний продукт [4].

Отже, тема кваліфікаційної роботи, присвячена розробці додатку для збору та аналізу інцидентів з відкритих джерел, пов'язаних із недостатньою якістю програмного забезпечення, є актуальною, оскільки спрямована на вирішення практичних задач автоматизації моніторингу якості ПЗ, систематизації відповідних відомостей та забезпечення їх зручної візуалізації для кінцевого користувача.

Метою кваліфікаційної роботи є розробка веб-додатку для автоматизованого збору та аналізу інцидентів з відкритих інтернет-джерел, пов'язаних із недостатньою якістю програмного забезпечення, з можливістю наочного представлення результатів аналізу у вигляді інтерактивних графіків та діаграм.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати сучасні підходи до моніторингу якості програмного забезпечення та класифікації дефектів;
- визначити вимоги до програмного додатку та обґрунтувати вибір технологій розробки;
- спроектувати структуру системи та реляційну базу даних для зберігання інцидентів;
- розробити модуль асинхронного парсингу відкритих інтернет-джерел;
- реалізувати алгоритм фільтрації та дедуплікації зібраних даних;

- розробити серверну частину додатку у вигляді RESTful API для доступу до накопичених даних та аналітичних показників;
- розробити клієнтську частину веб-додатку з інтерактивними графіками та діаграмами для візуалізації результатів аналізу інцидентів;
- провести тестування розроблених компонентів та оцінити ефективність запропонованого рішення.

Об'єктом дослідження є якість програмного забезпечення як властивість програмних систем, що визначає ступінь їх відповідності встановленим вимогам та очікуванням користувачів.

Предметом дослідження є інциденти та дефекти програмного забезпечення, зафіксовані у відкритих інтернет-джерелах, а також методи і засоби їх автоматизованого збору, класифікації, аналізу та візуалізації.

Практичне значення роботи полягає у створенні веб-додатку, який може бути використаний дослідниками у сфері програмної інженерії та фахівцями з управління якістю для автоматизованого моніторингу інцидентів ПЗ, виявлення тенденцій у виникненні дефектів за визначені періоди, а також для прийняття обґрунтованих рішень щодо підвищення надійності програмних систем.

РОЗДІЛ 1

АНАЛІЗ МЕТОДІВ РОЗРОБКИ ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПОСТАНОВКА ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

1.1 Аналіз сучасного стану проблеми

Якість програмного забезпечення є одним із центральних понять сучасної програмної інженерії. Відповідно до міжнародного стандарту ISO/IEC 25010, якість ПЗ визначається як сукупність характеристик, що відображають здатність програмного продукту задовольняти встановлені та передбачувані потреби користувачів. До ключових характеристик якості належать: функціональна відповідність, надійність, зручність використання, ефективність виконання, супроводжуваність та переносимість [5].

Порушення цих характеристик фіксується у вигляді інцидентів – подій, що призводять до відхилення поведінки системи від очікуваної. Для систематизації таких відхилень у науковій літературі запропоновано ряд класифікаційних схем. Зокрема, таксономія Orthogonal Defect Classification (ODC), розроблена компанією IBM, класифікує дефекти за типом (алгоритмічні, інтерфейсні, функціональні) та за контекстом виявлення (кодування, тестування, експлуатація). Класифікація Hewlett-Packard, у свою чергу, зосереджується на категоріях помилок залежно від процесного контексту їх виникнення [6].

Важливим джерелом даних про інциденти ПЗ є відкриті інтернет-джерела. Концепція відкритих даних (Open Data) передбачає вільний доступ до інформації без обмежень щодо її повторного використання. У галузі програмної інженерії до таких джерел належать публічні системи відстеження помилок (Bug Tracker), новинні агрегатори, технічні форуми та соціальні платформи. Аналіз публікацій у цих джерелах дозволяє отримати актуальну інформацію про реальні прояви дефектів у широкому спектрі програмних продуктів [7].

Автоматизований збір даних з веб-ресурсів здійснюється за допомогою технології парсингу (web scraping). Парсинг передбачає програмне завантаження вмісту веб-сторінок та вилучення з нього структурованої інформації. Для

підвищення продуктивності збору застосовуються асинхронні підходи, що дозволяють виконувати одночасно велику кількість HTTP-запитів без блокування основного потоку виконання. Окремою задачею при роботі з даними з множини джерел є дедуплікація – виявлення та усунення повторюваних записів, що описують одну і ту ж подію [8].

Для визначення доцільності розробки нової системи проведено огляд існуючих програмних рішень, що частково вирішують подібні задачі.

Jira Software (Atlassian) є однією з найпоширеніших систем відстеження помилок та управління проектами. Вона надає широкі можливості для реєстрації та категоризації інцидентів, формування звітів і відстеження їх вирішення. Проте система орієнтована виключно на внутрішнє середовище організації та не передбачає автоматизованого збору даних із зовнішніх відкритих джерел. Крім того, впровадження Jira потребує значних фінансових витрат на ліцензування, що робить її недоступною для невеликих дослідницьких груп [9].

Bugzilla є відкритою системою відстеження дефектів, що широко використовується у проєктах з відкритим кодом. Вона забезпечує зберігання, пошук та класифікацію помилок, однак також орієнтована на ручне введення даних і не підтримує автоматизованого збору інформації із зовнішніх ресурсів. Відсутність сучасного веб-інтерфейсу для аналітики та візуалізації даних суттєво обмежує її можливості для дослідницьких цілей [10].

Системи класу SIEM (Security Information and Event Management), зокрема Splunk та IBM QRadar, призначені для збору та аналізу подій безпеки в реальному часі. Незважаючи на потужні аналітичні можливості, ці рішення є надмірно складними та дорогими для застосування у контексті дослідження якості ПЗ на основі відкритих даних. Вони не передбачають роботи з неструктурованими текстовими джерелами та не орієнтовані на задачі моніторингу публічних інцидентів [11].

Агрегатори новин та RSS-клієнти (Feedly, Inoreader) дозволяють збирати публікації з множини джерел в одному місці. Однак вони не забезпечують автоматичної фільтрації за тематикою інцидентів ПЗ, не виконують

дедуплікацію, не зберігають дані у структурованому вигляді та не надають інструментів для аналітики і побудови графіків [12].

Результати порівняльного аналізу існуючих рішень засвідчують, що жодне з них не забезпечує повного набору функцій, необхідних для вирішення поставленої задачі. Зокрема, серед розглянутих систем відсутнє рішення, яке б поєднувало автоматизований збір даних з відкритих джерел, їх дедуплікацію, структуроване зберігання та наочну візуалізацію результатів аналізу.

Таким чином, проведений аналіз підтверджує доцільність розробки спеціалізованого веб-додатку для збору та аналізу інцидентів з відкритих джерел, пов'язаних із недостатньою якістю програмного забезпечення. Запропоноване рішення має усунути виявлені недоліки існуючих аналогів та забезпечити дослідників зручним інструментом для моніторингу актуальних тенденцій у сфері якості ПЗ.

1.2 Постановка завдання на кваліфікаційну роботу бакалавра

На підставі проведеного аналізу предметної області та існуючих програмних рішень сформульовано такі конкретні завдання, що мають бути досягнуті в результаті виконання кваліфікаційної роботи:

- проаналізувати сучасні підходи до моніторингу якості програмного забезпечення та класифікації дефектів. Огляд має охопити як міжнародні стандарти оцінювання якості ПЗ, так і таксономії класифікації дефектів, що застосовуються у промисловості та академічних дослідженнях. На основі результатів аналізу необхідно сформувати перелік ключових ознак, за якими доцільно структурувати інциденти у системі;

- визначити вимоги до програмного додатку та обґрунтувати вибір технологій розробки. Окремо мають бути сформовані функціональні вимоги (збір, фільтрація, дедуплікація, аналітика) та нефункціональні (продуктивність, масштабованість, стійкість до помилок мережі). Вибір технологічного стеку має

бути підтверджений порівняльним аналізом альтернатив за критеріями придатності до конкретних задач системи;

- спроектувати структуру системи та реляційну базу даних для зберігання інцидентів. Структуру системи необхідно описати засобами UML-нотації з використанням діаграм варіантів використання, класів, послідовності та пакетів. Схема бази даних має забезпечувати збереження зв'язків між інцидентами, джерелами та типами дефектів і підтримувати ефективну вибірку за множинними критеріями фільтрації;

- розробити модуль асинхронного парсингу відкритих інтернет-джерел. Модуль повинен підтримувати паралельний обхід кількох гетерогенних джерел (новинні ресурси, RSS/Atom-стрічки, публічні API) із контролем кількості одночасних з'єднань та механізмами повторних спроб при збоях. Архітектура модуля має дозволяти просте додавання нових типів джерел без зміни базового коду;

- реалізувати алгоритм фільтрації та дедуплікації зібраних даних. Дедуплікація має ґрунтуватися на нечіткому порівнянні заголовків з використанням метрики Левенштейна та налаштовуваним пороговим значенням схожості. Алгоритм фільтрації повинен забезпечувати відсіювання нерелевантного контенту за тематикою інцидентів ПЗ та класифікувати збережені записи за типом і рівнем критичності дефекту;

- розробити серверну частину додатку у вигляді RESTful API для доступу до накопичених даних та аналітичних показників. API має забезпечувати ендпоінти для перегляду списку інцидентів з підтримкою фільтрації за множинними параметрами, отримання деталей конкретного запису та формування агрегованої статистики за визначені часові періоди. Серверна частина повинна підтримувати асинхронну обробку запитів і автоматичну валідацію вхідних параметрів;

- розробити клієнтську частину веб-додатку з інтерактивними графіками та діаграмами для візуалізації результатів аналізу інцидентів. Інтерфейс має включати головну аналітичну панель із зведеними показниками, лінійним

графіком динаміки, круговою діаграмою розподілу за джерелами та стовпчиковою діаграмою за категоріями дефектів. Усі компоненти повинні оновлюватися без перезавантаження сторінки та зберігати стан фільтрів у URL-адресі для можливості обміну посиланнями;

- провести тестування розроблених компонентів та оцінити ефективність запропонованого рішення. Ефективність системи має бути підтверджена кількісними показниками точності класифікації та часу відповіді API на типові запити користувача.

РОЗДІЛ 2

СПЕЦИФІКАЦІЯ ВИМОГ ДО РОЗРОБЛЮВАНОЇ СИСТЕМИ

2.1 Аналіз, визначення вимог до розроблюваного програмного забезпечення та проєктування програмного забезпечення

Розроблюваний веб-додаток призначений для автоматизованого збору, зберігання та аналізу даних про інциденти програмного забезпечення з відкритих інтернет-джерел. Основною метою системи є надання дослідникам та фахівцям у сфері якості ПЗ зручного інструменту для моніторингу публічних повідомлень про дефекти та збої програмних продуктів.

Система має реалізовувати такі основні цілі:

- автоматизація процесу збору відомостей про інциденти ПЗ з гетерогенних відкритих джерел без участі користувача;
- забезпечення структурованого зберігання зібраних даних з підтримкою класифікації за джерелом, типом помилки та часовим проміжком;
- фільтрація дублікатів для уникнення повторного збереження інцидентів, що висвітлюються різними джерелами;
- надання аналітичних показників та їх наочна візуалізація у вигляді інтерактивних графіків і діаграм у браузерному інтерфейсі;
- забезпечення зручного пошуку та фільтрації накопичених даних за різними параметрами.

Для моделювання розроблюваної системи обрано об'єктно-орієнтований підхід із застосуванням уніфікованої мови моделювання UML. Цей підхід є загальноприйнятим стандартом у сучасній програмній інженерії та дозволяє комплексно описати як структуру системи, так і динаміку її функціонування. Для опису функціональних вимог використано діаграму варіантів використання (Use Case Diagram), для відображення структури даних – діаграму класів (Class Diagram), для опису послідовності взаємодій – діаграму послідовності (Sequence Diagram) [13].

Функціональні вимоги визначають конкретні функції, які система повинна виконувати. До функціональних вимог розроблюваного додатку належать:

- система повинна автоматично завантажувати вміст визначених веб-ресурсів та вилучати з нього заголовки, описи, URL та дату публікації повідомлень про інциденти;
- система повинна перевіряти кожне зібране повідомлення на релевантність (відповідність тематиці інцидентів ПЗ) та відсіювати нерелевантний вміст;
- система повинна виявляти та позначати повідомлення, що описують одну і ту ж подію, на основі порівняння заголовків із заданим порогом схожості;
- система повинна зберігати структуровані записи про інциденти у реляційній базі даних з підтримкою зв'язків між сутностями;
- система повинна надавати RESTful API для отримання списку інцидентів, статистичних показників та результатів пошуку;
- веб-інтерфейс повинен відображати динаміку інцидентів за часовими проміжками, розподіл за джерелами та категоріями у вигляді графіків і діаграм;
- користувач повинен мати можливість фільтрувати інциденти за джерелом, типом помилки та часовим діапазоном.

Нефункціональні вимоги визначають якісні характеристики системи:

- модуль парсингу повинен підтримувати асинхронне виконання запитів з обмеженням одночасних з'єднань для уникнення перевантаження джерел;
- система повинна коректно обробляти помилки мережевих запитів та недоступність джерел без переривання роботи;
- архітектура системи повинна допускати додавання нових джерел даних без зміни базового коду;
- веб-інтерфейс повинен забезпечувати інтуїтивно зрозумілу навігацію та адаптивне відображення на різних пристроях.

Для формалізації функціональних вимог розроблено діаграму варіантів використання (Use Case Diagram), що відображає взаємодію між акторами системи та її основними функціями. Акторами системи є: «Користувач»

(здійснює перегляд аналітики та пошук інцидентів) та «Планувальник» (автоматично запускає збір даних за розкладом). Основні варіанти використання включають: перегляд графіків, фільтрацію інцидентів, пошук за ключовими словами, запуск парсингу та перегляд списку інцидентів показано на рисунку 2.1.

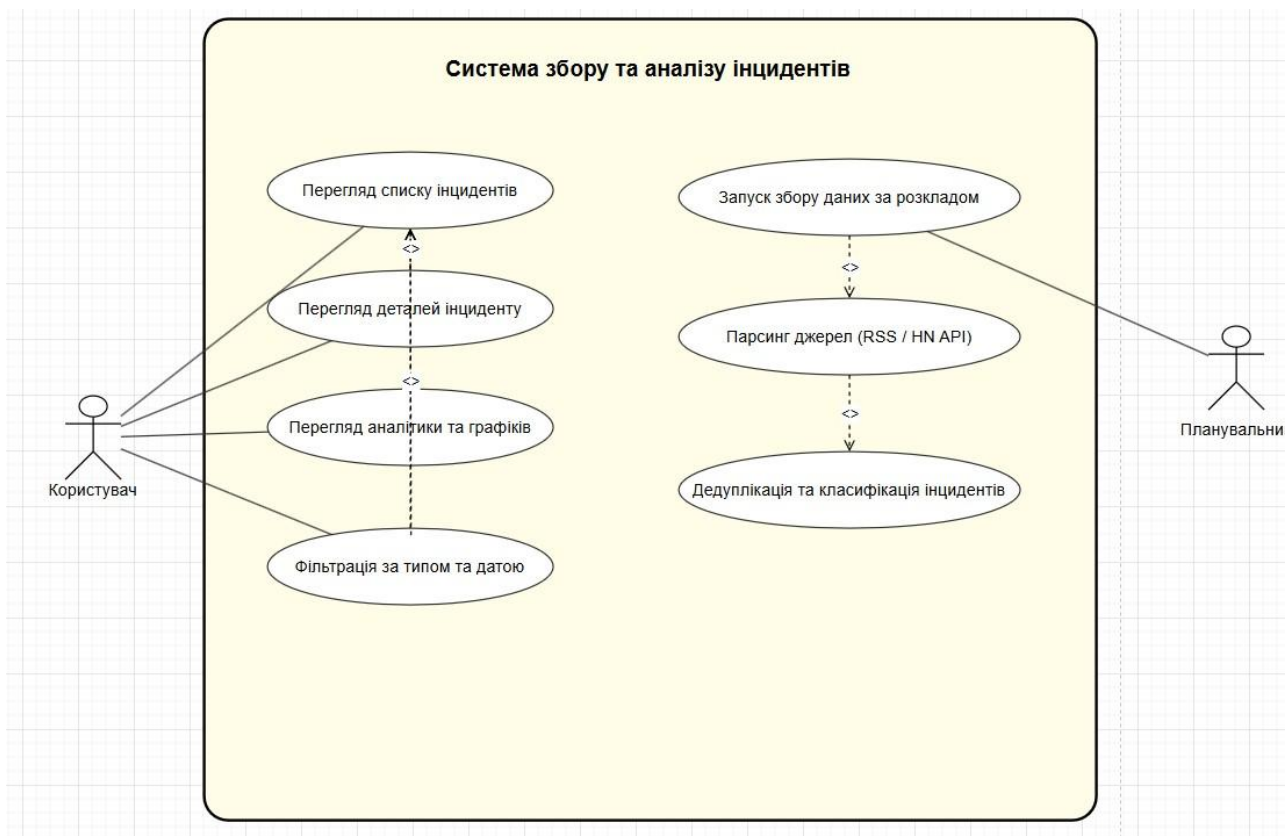


Рисунок 2.1 – Діаграма варіантів використання системи

Структуру даних системи відображає діаграма класів (рис. 2.2). Основними класами є: Incident (інцидент), що містить атрибути ідентифікатора, заголовку, опису, URL, дати публікації та ознаки дубліката; Source (джерело), що зберігає назву, URL та категорію ресурсу; IncidentType (тип інциденту), що описує категорію та критичність дефекту. Між класами Incident і Source визначено відношення асоціації «багато-до-одного», між Incident і IncidentType – відношення «багато-до-багатьох» через зв'язкову таблицю.

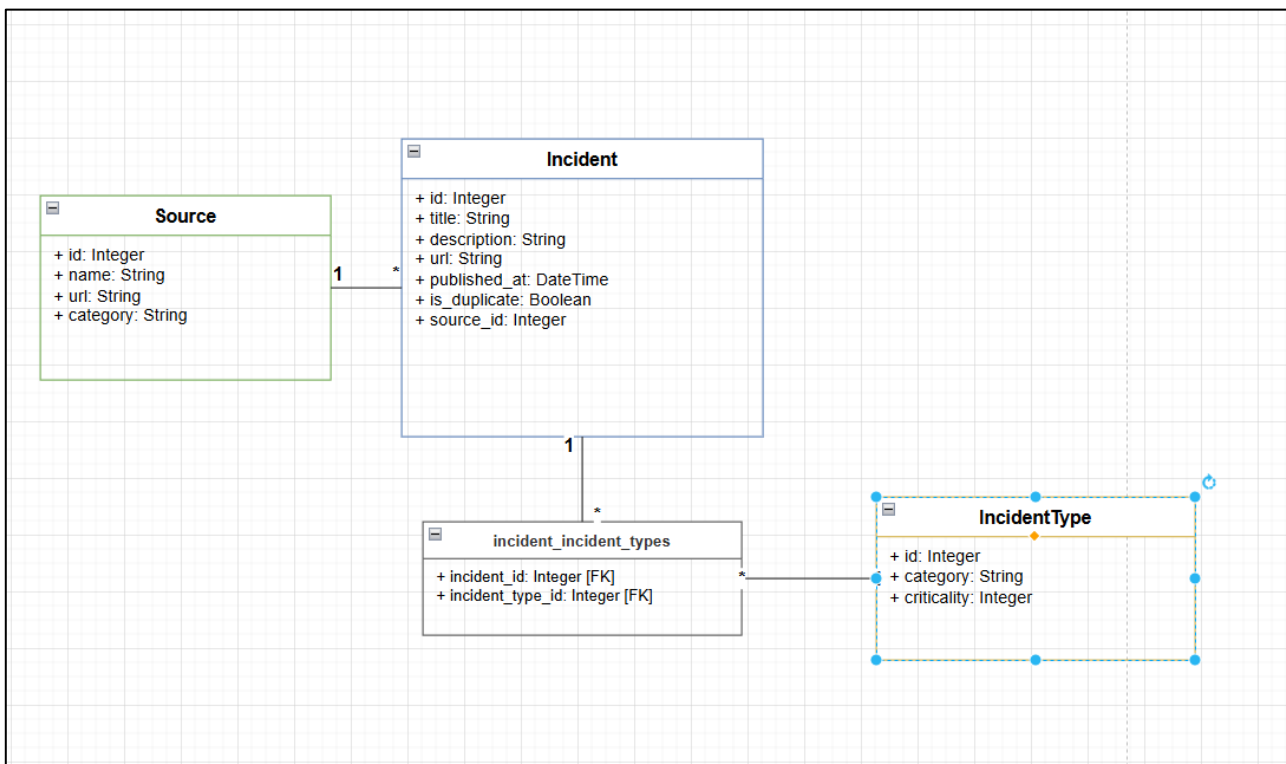


Рисунок 2.2 – Діаграма класів системи

Динаміку взаємодії компонентів системи під час збору та збереження інцидентів відображає діаграма послідовності (рис. 2.3). Процес розпочинається з виклику `run_parser()` планувальником, який за встановленим розкладом передає керування модулю парсингу. Модуль парсингу формує асинхронний HTTP GET-запит до зовнішнього джерела; у разі успішної відповіді (200 OK) джерело повертає вміст у форматі HTML або XML, після чого модуль виконує розбір документа та витяг записів про інциденти.

Для кожного витягнутого запису модуль звертається до дедуплікатора з викликом `check_duplicate(title)`. Дедуплікатор, у свою чергу, направляє запит `SELECT by title_hash` до бази даних та отримує булевий результат щодо унікальності запису. Подальша поведінка системи визначається умовним фрагментом `alt`: якщо інцидент є унікальним, модуль парсингу виконує операцію `INSERT incident` у базі даних, отримує згенерований ідентифікатор `incident.id` та викликає `update_index(incident)` для оновлення індексу дедуплікатора; якщо ж запис ідентифіковано як дублікат, дедуплікатор повертає відповідний прапор і інцидент пропускається без збереження. Після

опрацювання всіх записів поточного джерела модуль парсингу повертає планувальнику результат `parsing_complete(saved_count)` із кількістю збережених інцидентів.

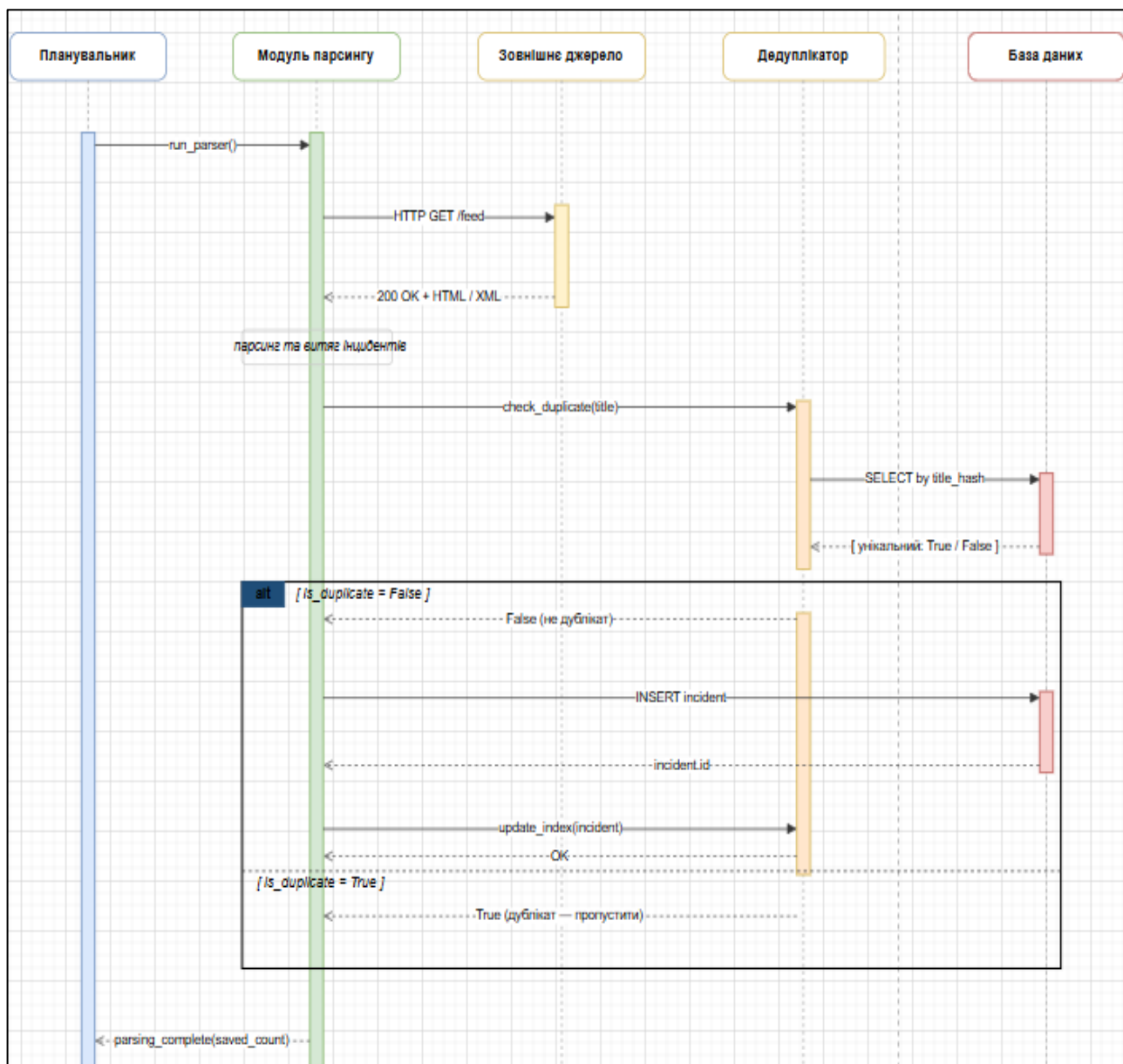


Рисунок 2.3 – Діаграма послідовності процесу збору інцидентів

Для опису сценаріїв взаємодії користувача з системою складено таблиці типу «подія-відгук». Основний сценарій перегляду аналітики передбачає такі кроки: користувач відкриває головну сторінку веб-додатку; система виконує запит до API (`/api/stats`) та отримує агреговані показники; клієнтська частина відображає графіки динаміки інцидентів за останні 30 днів, кругову діаграму

розподілу за джерелами та стовпчикову діаграму розподілу за типами помилок. Сценарій фільтрації передбачає введення користувачем параметрів фільтру (джерело, часовий діапазон), після чого система повертає відфільтрований список інцидентів.

Проектування інтерфейсу користувача здійснено на основі принципів UX/UI-дизайну з урахуванням потреб цільової аудиторії – дослідників та аналітиків у сфері якості ПЗ. Веб-інтерфейс складається з таких основних екранів: головна аналітична панель з графіками та ключовими показниками; сторінка перегляду списку інцидентів з пошуком та фільтрацією; сторінка детального перегляду окремого інциденту.

Головна аналітична панель (рис. 2.4) містить: лінійний графік динаміки кількості інцидентів за обраний часовий період; кругову діаграму розподілу інцидентів за джерелами; стовпчикову діаграму розподілу за категоріями помилок; зведені числові показники (загальна кількість інцидентів, кількість унікальних, відсоток дублікатів, кількість активних джерел).

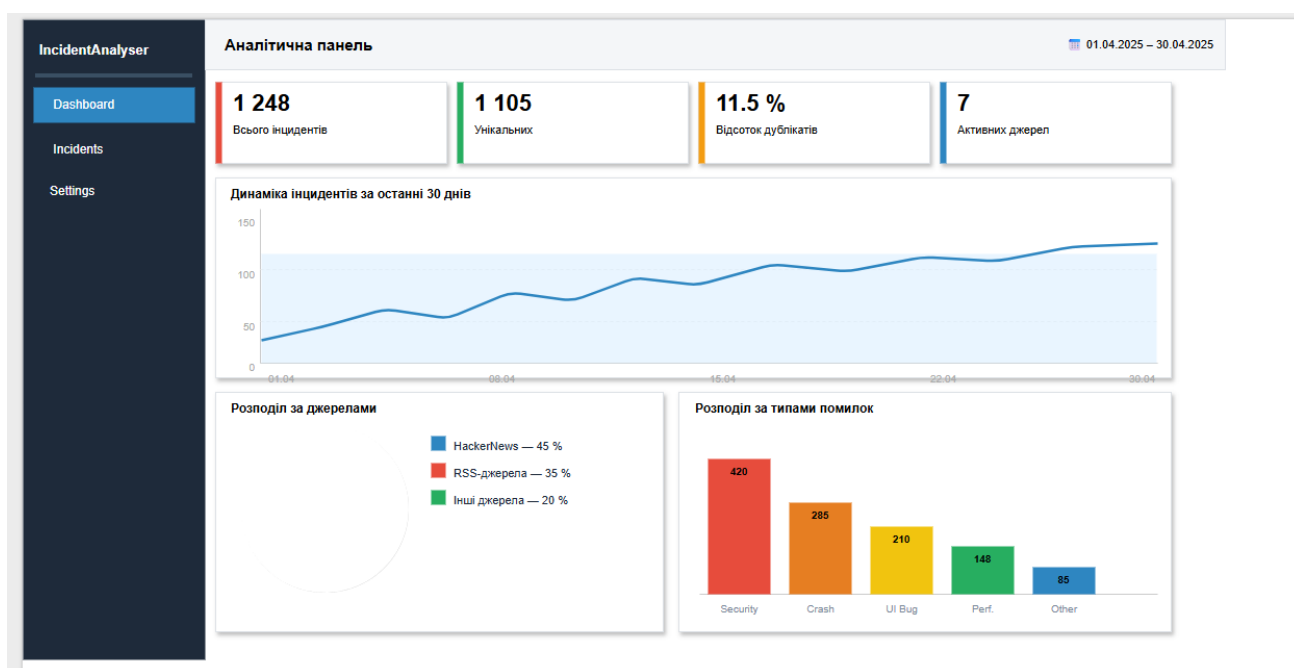


Рисунок 2.4 – Макет головної аналітичної панелі веб-додатку

Інформаційні потоки системи організовані за трирівневою архітектурою: рівень збору даних (модуль парсингу взаємодіє із зовнішніми джерелами), рівень зберігання та обробки (база даних PostgreSQL та серверна частина FastAPI), рівень представлення (клієнтська частина отримує дані через REST API та відображає їх у браузері). Така архітектура забезпечує чіткий поділ відповідальності між компонентами та спрощує їх незалежне розширення і тестування.

2.2 Вибір засобів, методів і алгоритмів вирішення поставленого завдання

Для розробки веб-додатку збору та аналізу інцидентів ПЗ потрібно обрати технології на кожному рівні: мову та фреймворки, базу даних, бібліотеки обробки даних і засоби побудови інтерфейсу. Для кожного рівня нижче порівнюються доступні варіанти, обґрунтовується вибір і описуються реалізовані алгоритми.

Перш ніж переходити до конкретних інструментів, треба визначитися з типом платформи. Розглядалося три варіанти: веб-додаток, десктопний застосунок і мобільний застосунок. Порівняльний аналіз наведено у таблиці 2.1.

Таблиця 2.1 – Порівняння типів платформ для реалізації системи

Критерій	Веб-додаток	Десктоп (Electron/PyQt)	Мобільний (React Native)
Встановлення	не потрібне	обов'язкове	обов'язкове
Кросплатформність	будь-який браузер	потребує збірок	iOS + Android окремо
Інтерактивні графіки	Recharts, D3.js	обмежені бібліотеки	обмежені бібліотеки
Таблиці з даними	зручно	зручно	незручно (малий екран)

Продовження таблиці 2.1

Критерій	Веб-додаток	Десктоп (Electron/PyQt)	Мобільний (React Native)
REST API як побічний продукт	так (безкоштовно)	ні	ні
Оновлення системи	миттєве (на сервері)	потребує перевстановлення	оновлення через стор
Доступ з будь-якого пристрою	так	ні	частково

Вибрано веб-додаток. Основна причина проста: цільова аудиторія – дослідники та аналітики якості ПЗ – здебільшого працює за стаціонарними комп’ютерами або ноутбуками. Для корпоративного та академічного середовища відсутність встановлення – не формальна перевага, а реальна зручність. Веб-інтерфейс при цьому однаково відкривається на Windows, Linux і macOS без жодних додаткових збірок.

Сам контент системи – таблиці інцидентів, аналітичні графіки, фільтри – погано вміщається на малих екранах і по суті розрахований на десктоп. REST API, побудований для клієнтської частини, автоматично стає інтеграційною точкою для зовнішніх систем. Оновлення логіки парсингу чи класифікації розгортаються на сервері й відразу набирають чинності для всіх – без жодного перевстановлення на боці користувача.

Десктопний застосунок (Electron або PyQt) програє за кількома пунктами: потребує встановлення на кожній машині, а екосистема веб-бібліотек для інтерактивних графіків помітно багатша.

Зрештою, серверна частина парсер і база даних однаково потребує окремого процесу, тож головний аргумент на користь десктопу відпадає. Мобільний застосунок підходить ще менше: таблиці з багатьма стовпцями на малому екрані незручні, а підтримка iOS і Android – це фактично два окремих проєкти.

Для серверної частини порівнювалися три варіанти: Python, Node.js і Java (табл. 2.2).

Таблиця 2.2 – Порівняння мов програмування для серверної частини

Критерій	Python	Node.js	Java
Підтримка асинхронності	asyncio, нативна	нативна (Event Loop)	Project Loom (JDK 21+)
Бібліотеки парсингу	aiohttp, lxml, BeautifulSoup	cheerio, axios	jsoup, OkHttp
Бібліотеки NLP/ML	широка екосистема	обмежена	обмежена
Швидкість розробки	висока	висока	низька
Порогова складність	низька	середня	висока
Придатність для парсингу	відмінна	добра	задовільна

Python 3.12 обрано насамперед через бібліотечний стек: для кожної задачі системи є готове рішення – aiohttp для HTTP-запитів, BeautifulSoup4 і lxml для парсингу HTML, rapidfuzz для нечіткого порівняння рядків, asyncpg і SQLAlchemy 2.0 async для роботи з базою даних, loguru для логування, pydantic-settings для конфігурації. Шукати аналоги на іншій мові не було сенсу [14].

Паралельний збір з кількох джерел вимагає асинхронності. Python підтримує asyncio нативно, без зовнішніх залежностей. Порівняно з Java, поріг входу нижчий, а прототипування швидше.

Node.js підійшов би для побудови API, але з обробкою і аналізом текстових даних його бібліотечна екосистема значно вужча.

Для серверного REST API порівнювалися три Python-фреймворки: FastAPI, Flask і Django REST Framework (табл. 2.3).

Таблиця 2.3 – Порівняння Python-фреймворків для REST API

Критерій	FastAPI	Flask	Django REST
Асинхронна підтримка	нативна (ASGI)	обмежена (WSGI)	обмежена (WSGI)
Автодокументація	Swagger/OpenAPI вбудований	потребує розширень	потребує розширень
Валідація даних	pydantic, автоматична	вручну	серіалізатори DRF

Продовження таблиці 2.3

Критерій	FastAPI	Flask	Django REST
Продуктивність	висока	середня	середня
Розмір додатку	мінімальний	мінімальний	великий
Типізація	повна підтримка	відсутня	часткова

FastAPI обрано через нативну асинхронність. Фреймворк побудований на ASGI і добре стикується з `asynpcrg`: блокуючих викликів у ланцюгу немає. Swagger UI на `/docs` генерується автоматично – ендпоінти можна тестувати одразу, без сторонніх інструментів. Pydantic-схеми беруть на себе валідацію вхідних параметрів [15].

Flask простіший в налаштуванні, але нативної асинхронності в ньому немає, а для системи з великою кількістю запитів до бази даних це вже принципово. Для зберігання даних про інциденти порівнювалися PostgreSQL, SQLite і MongoDB (табл. 2.4).

Таблиця 2.4 – Порівняння систем управління базою даних

Критерій	PostgreSQL	SQLite	MongoDB
Підтримка зв'язків	повна (FK, JOIN)	повна	відсутня (документи)
Паралельні з'єднання	пул (<code>asynpcrg</code>)	обмежена	пул
Повнотекстовий пошук	вбудований	FTS5	Atlas Search
Масштабованість	висока	обмежена	висока
Типізація даних	строга	динамічна	гнучка
Файловий розмір	сервер	один файл	сервер

PostgreSQL 16 обрано через реляційну структуру даних. Зв'язки Incident–Source та Incident–IncidentType потребують зовнішніх ключів і каскадних операцій, без реляційної СУБД це не реалізувати чисто. Строга типізація гарантує цілісність записів, вбудований повнотекстовий пошук покриває

фільтрацію за ключовими словами, `asynsrg` дає асинхронний пул з'єднань без зайвих прошарків над FastAPI [16].

SQLite підходить для локальної розробки, але не тягне повноцінний пул асинхронних з'єднань і погано справляється з паралельними записами. MongoDB у цьому контексті не варіант – дані реляційні.

Для роботи з базою використовується SQLAlchemy 2.0 у асинхронному режимі. Це де-факто стандарт ORM у Python-екосистемі: декларативні моделі, `relationship` для зв'язків між таблицями, міграції через Alembic, `AsyncSession`. Tortoise ORM поступається за обсягом документації та розміром спільноти. Схема бази описується через DeclarativeBase тому SQL-міграції вручну писати не потрібно [17].

Для дедуплікації обрано `rapidfuzz` – C++-реалізацію алгоритму Левенштейна та похідних метрик. В 3-6 разів швидша за `fuzzywuzzy` і не тягне залежність від застарілої `python-Levenshtein`.

HTML парситься через BeautifulSoup4 з парсером lxml. lxml коректно обробляє нестандартну розмітку RSS/Atom-стрічок, з якою html.parser іноді не справляється.

Для клієнтської частини порівнювалися React, Vue.js і серверний рендеринг на шаблонах Jinja2 (табл. 2.5).

Таблиця 2.5 – Порівняння засобів розробки клієнтської частини

Критерій	React 18	Vue.js 3	Jinja2 (SSR)
Реактивне оновлення	Virtual DOM	реактивна система	відсутнє
Екосистема графіків	Recharts, Chart.js	Vue-ChartJS	Chart.js (обмежено)
SPA-маршрутизація	React Router	Vue Router	відсутня
Стан фільтрів у URL	useSearchParams	useRouter	не підтримується
Швидкість збірки	Vite (висока)	Vite (висока)	не потрібна
Поширеність	найвища	висока	середня

Для клієнтського інтерфейсу обрано React 18. Компонентна модель дозволяє повторно використовувати елементи (картки, фільтри, пагінацію) по всьому додатку. Хук `useSearchParams` з React Router DOM зберігає стан фільтрів у URL query string де без цього стан сторінки не відтворюється. Для графіків обрано Recharts: бібліотека на базі D3.js, інтегрується з React-компонентами без окремої ініціалізації. Збірка і розробницький сервер Vite 6 з HMR. Jinja2 не підходить адже реактивне оновлення графіків без перезавантаження сторінки там неможливе.

Модуль збору даних реалізує асинхронний паралельний обхід джерел. Абстрактний клас `BaseParser` визначає спільний інтерфейс: метод `run()` повертає список об'єктів `RawIncident` незалежно від типу джерела. `HackerNewsParser` та `RSSParser` перевизначають внутрішню логіку вилучення даних.

Кроки алгоритму збору:

- ініціалізація сеансу `aihttp` з таймаутом 30 с і `Semaphore` для обмеження паралельних запитів;
- HTTP GET до джерела; при помилці повтор з експоненціальною 1затримкою (1 с, 2 с, 4 с);
- парсинг відповіді: XML через `lxml` для RSS/Atom, JSON-десеріалізація для Hacker News API;
- фільтрація за ключовими словами тематики інцидентів ПЗ у заголовку та описі;
- нормалізація дати публікації до UTC datetime;
- формування об'єкта `RawIncident` (title, description, url, published_at) і передача до оркестратора.

Дедуплікація вирішує одну конкретну проблему: різні джерела описують одну і ту саму подію різними заголовками. Пряме порівняння рядків тут не працює. Тому використовується метрика `token_sort_ratio` з `rapidfuzz`, яка спирається на відстань Левенштейна з попереднім сортуванням токенів.

Відстань Левенштейна між рядками `x1` і `x2` визначається як мінімальна. Інциденти класифікуються за типом дефекту через keyword matching. Перевага

перед ML: не потребує навчальної вибірки, результати прозорі тому аналітик може переглянути і скоригувати словник прямо у коді. Перевага перед регулярними виразами: простіше підтримувати при зростанні кількості категорій.

Клас `IncidentClassifier` містить словник `TYPE_KEYWORDS` який є набором ключових слів і рівень критичності для кожного типу дефекту. Метод `classify(title, description)`:

- об'єднання заголовку та опису в єдиний рядок, приведення до нижнього регістру;
- для кожного типу зі словника є перевірка наявності хоча б одного ключового слова;
- для типу «Third-party Issue» це контекстна верифікація (`require_all`): потрібна одночасна присутність слова з групи вендорів (`google`, `microsoft`, `aws` тощо) і слова з групи проблем (`outage`, `breach`, `bug` тощо), щоб уникнути хибних спрацювань на нейтральні згадки компаній;
- якщо виявлені конкретні типи, `Generic Issue` виключається зі списку;
- підсумкова критичність є максимальна серед виявлених типів за шкалою `low < medium < high < critical`;
- повертається `ClassificationResult` з полями `types`, `criticality` та `type_criticalities`.

Відповідність типів і рівнів критичності наведено у таблиці 2.6.

Таблиця 2.6 – Типи дефектів та рівні критичності

Тип дефекту	Рівень критичності	Приклад ключових слів
Security Vulnerability	critical	vulnerability, CVE, exploit, breach, malware
Data Loss/Exposure	critical	data loss, data breach, leak, exposure
Supply Chain Attack	critical	supply chain, backdoor, compromised package
System Outage	high	outage, down, offline, unavailable, crashed

Продовження таблиці 2.6

Тип дефекту	Рівень критичності	Приклад ключових слів
Bug/Error	medium	bug, error, crash, failure, regression
Performance Issue	medium	slow, lag, latency, timeout, memory leak
Configuration Issue	medium	misconfiguration, certificate, expired
Third-party Issue	medium	вендор + проблема (require_all)

Система складається з чотирьох пакетів відповідно до тривірневої архітектури. Центральний `incident_parser` включає логіку збору (`parser: base.py` та підпакет `sources` з конкретними реалізаціями), фільтрації (`classifier.py`, `deduplicator.py`), роботи з базою даних (`models.py`, `database.py`, `repository.py`) і конфігурації (`settings.py` на Pydantic). Решта три менші: `backend` містить `api.py` з ендпоінтами FastAPI; `frontend` – React-компоненти (`src/pages`, `src/api.js`, `src/App.jsx`). Структуру пакетів наведено на рисунку 2.5.

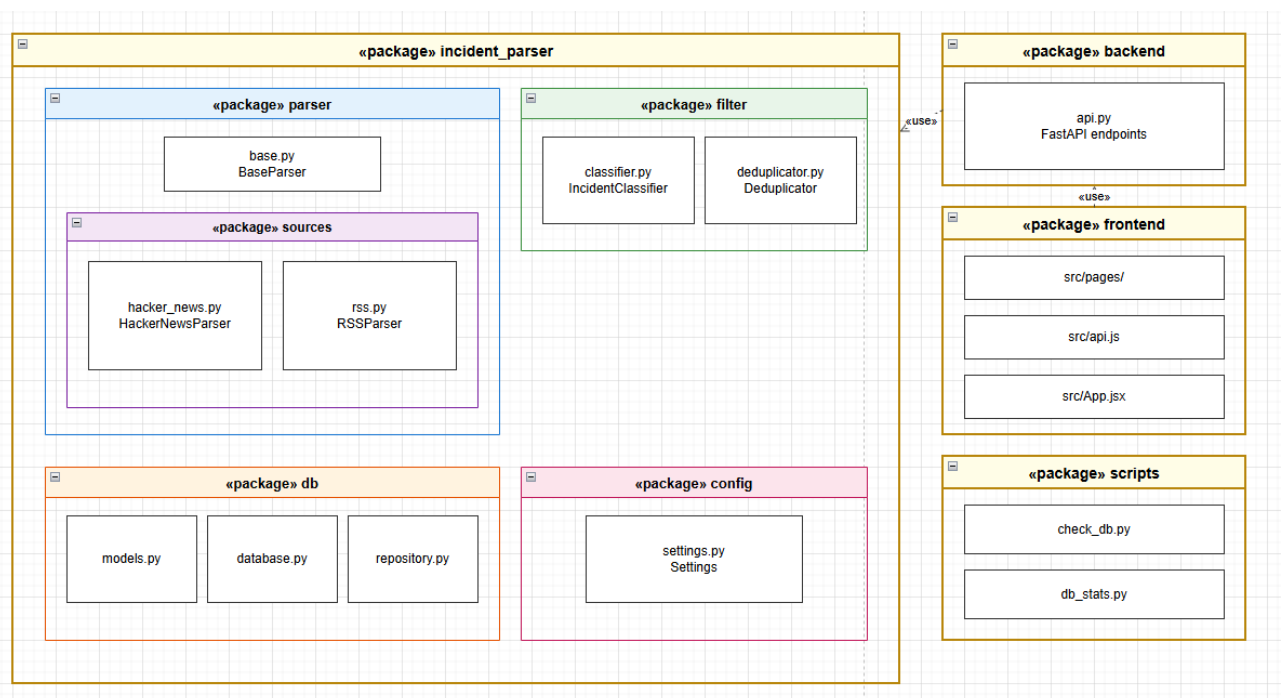


Рисунок 2.5 – Діаграма пакетів системи

Підсумковий стек: серверна частина Python 3.12, FastAPI, SQLAlchemy 2.0 async; база даних PostgreSQL 16 з asyncpg; збір і обробка aiohttp, BeautifulSoup4/lxml, rapidfuzz; клієнтська частина React 18, Recharts, axios, Vite 6. Вибір кожного інструменту обґрунтовано у попередніх підрозділах через порівняння з альтернативами. Алгоритми дедуплікації на token_sort_ratio і класифікації через словниковий пошук з контекстною верифікацією реалізовано у межах цього стеку [18].

РОЗДІЛ 3

РОЗРОБКА ЗАСТОСУНКУ ДЛЯ ЗБОРУ ТА АНАЛІЗУ ІНЦИДЕНТІВ З ВІДКРИТИХ ДЖЕРЕЛ, ПОВ'ЯЗАНИХ ІЗ НЕДОСТАТНЬОЮ ЯКІСТЮ ПЗ

3.1 Практична реалізація об'єкта проєктування

Практична реалізація веб-додатку здійснювалась у середовищі Visual Studio Code з розширенням Python та використанням інтерпретатора Python 3.12. Проєкт організовано у вигляді пакетів відповідно до трирівневої архітектури: модуль збору даних (`incident_parser`), серверна частина (`backend`) та клієнтська частина (`frontend`). Контроль версій здійснювався засобами Git.

Першим кроком реалізовано модуль конфігурації на основі бібліотеки `pydantic-settings`. Клас `Settings` зчитує параметри підключення до бази даних та налаштування парсера з файлу `.env`, що дозволяє змінювати конфігурацію без зміни коду. Реалізацію наведено у лістингу 3.1.

Лістинг 3.1 – Модуль налаштувань

```
from pydantic_settings import BaseSettings
from dotenv import load_dotenv
load_dotenv()
class Settings(BaseSettings):
    db_host: str = "localhost"
    db_port: int = 5432
    db_name: str = "incidents_db"
    db_user: str = "postgres"
    db_password: str = ""
    request_delay: float = 1.5
    max_concurrent: int = 5
    request_timeout: int = 30
    dedup_threshold: int = 85
    @property
    def db_url(self) -> str:
        return (
            f"postgresql+asyncpg://{self.db_user}:{self.db_password}"
            f"@{self.db_host}:{self.db_port}/{self.db_name}"
        )
class Config:
    env_file = ".env"
    settings = Settings()
```

кінець лістингу 3.1

Структуру даних описано через ORM-моделі бібліотеки SQLAlchemy 2.0 з декларативним базовим класом. Між таблицями `incidents` та `incident_types` реалізовано зв'язок «багато-до-багатьох» через зв'язкову таблицю `incident_type_association`. Фрагмент моделі `Incident` наведено у лістингу 3.2.

Лістинг 3.2 – ORM-модель інциденту

```
class Incident(Base):
    __tablename__ = "incidents"
    id = Column(Integer, primary_key=True,
autoincrement=True)
    title = Column(String(512), nullable=False)
    description = Column(Text)
    url = Column(String(512), unique=True)
    published_at = Column(DateTime)
    collected_at = Column(DateTime, default=datetime.utcnow)
    is_duplicate = Column(Boolean, default=False)
    duplicate_of_id = Column(Integer, ForeignKey("incidents.id"),
nullable=True)
    similarity_score = Column(Float, nullable=True)
    source_id = Column(Integer, ForeignKey("sources.id"))
    source = relationship("Source", back_populates="incidents")
    types = relationship(
        "IncidentType",
        secondary=incident_type_association,
        back_populates="incidents",
    )
```

кінець лістингу 3.2

Атрибут `is_duplicate` зберігає ознаку повторного запису, виявленого дедуплікатором, а поле `duplicate_of_id` як зовнішній ключ на ту саму таблицю `incidents` дозволяє відстежити оригінальний запис, з яким зафіксовано збіг, та значення метрики схожості у `similarity_score`. Зв'язки `source` і `types`, оголошені через декларативний атрибут `relationship`, забезпечують автоматичне завантаження пов'язаних об'єктів засобами ORM без потреби у написанні явних SQL-запитів. Загальну організацію коду проєкту відображено на рисунку 3.1.

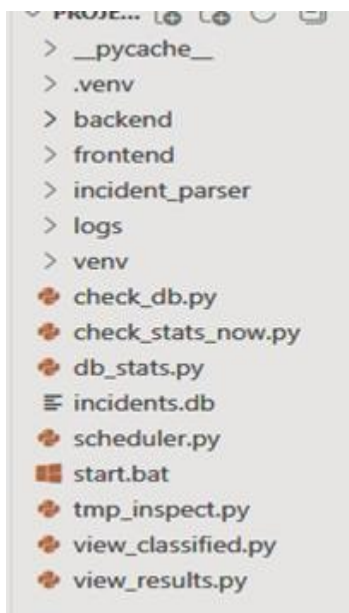


Рисунок 3.1 – Структура каталогів проєкту у VS Code

Клас BaseParser реалізує шаблон Template Method: run() визначає загальний алгоритм збору, а parse_page() перевизначається у підкласах. Паралельні запити обмежуються через asyncio.Semaphore. Реалізацію методу fetch() у лістингу 3.3, схему успадкування класів парсерів наведено на рисунку 3.2.

Лістинг 3.3 – Асинхронний HTTP-запит з обмеженням паралельності

```

async def fetch(self, url: str) -> Optional[str]:
    async with self._semaphore:
        try:
            session = await self._get_session()
            async with session.get(url) as resp:
                if resp.status != 200:
                    logger.warning(f"HTTP {resp.status} -> {url}")
                    return None
                html = await resp.text()
                await asyncio.sleep(settings.request_delay)
                return html
        except aiohttp.ClientError as exc:
            logger.error(f"Помилка запиту {url}: {exc}")
            return None

```

кінець лістингу 3.3

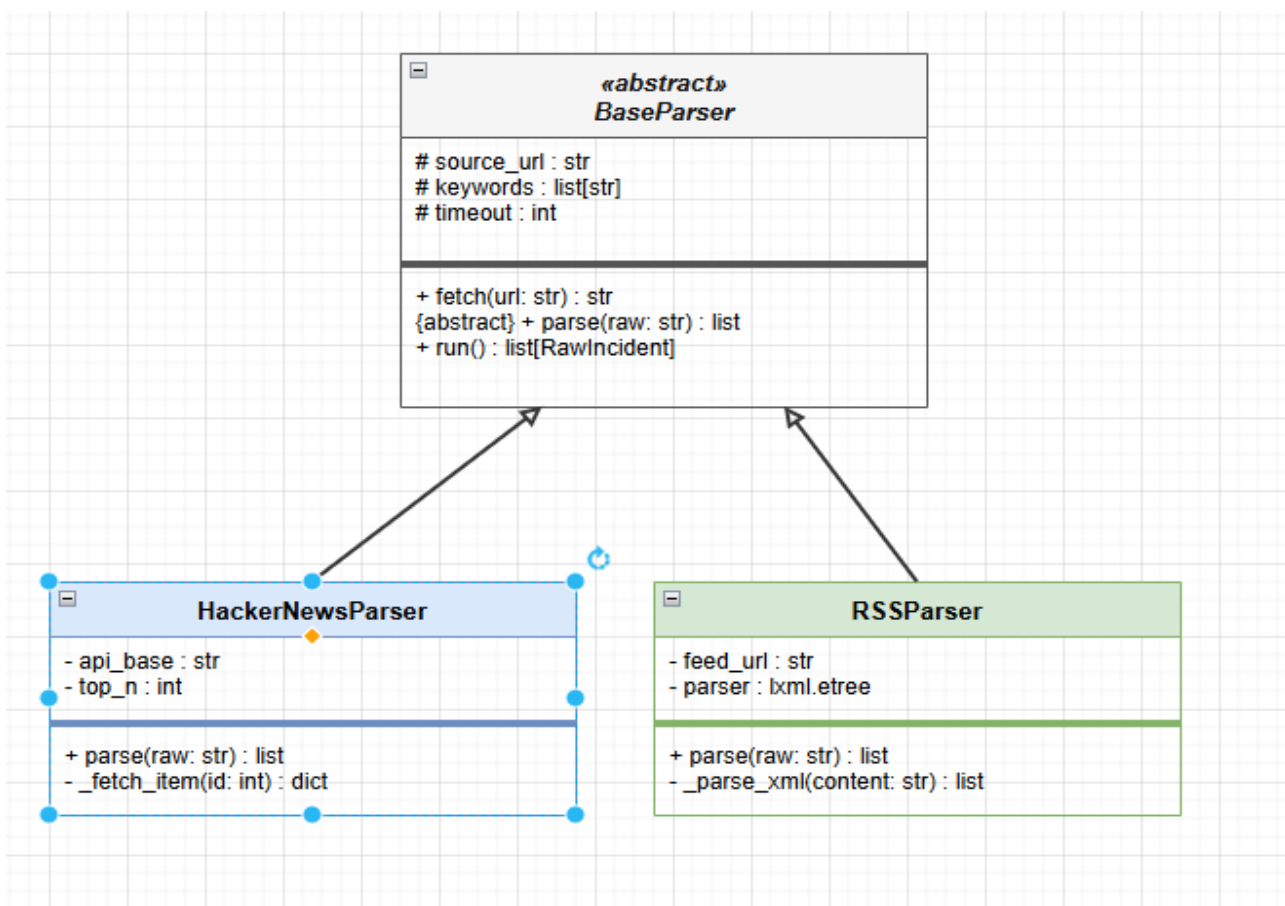


Рисунок 3.2 – Схема успадкування класів парсерів

Клас `HackerNewsParser` використовує офіційний Firebase REST API. Метод `run()` завантажує ідентифікатори топ-500 публікацій, після чого їхні деталі завантажуються паралельно через `asyncio.gather()`. Лістинг 3.4 демонструє цикл фільтрації та формування результату.

Лістинг 3.4 – Збір та фільтрація постів HackerNews

```

async def run(self) -> list[RawIncident]:
    ids = await self._fetch_top_story_ids()
    tasks = [self._fetch_item(i) for i in ids]
    items = await asyncio.gather(*tasks)
    incidents = []
    for item in items:
        if not item or item.get("type") != "story":
            continue
        title = item.get("title", "")
        if not self._is_incident_related(title):
            continue
  
```

```

        url    = item.get("url") or
f"{self.source_url}/item?id={item['id']}"
        score = item.get("score", 0)
        ts     = item.get("time")
        pub_dt = datetime.fromtimestamp(ts, tz=timezone.utc) if ts else
None
        if pub_dt and pub_dt.tzinfo:
            pub_dt = pub_dt.replace(tzinfo=None)
        incidents.append(RawIncident(
            title=title, description=f"[HN score: {score}] {url}",
            url=url, published_at=pub_dt, tags=["hacker-news"],
        ))
    return incidents

```

кінець лістингу 3.4

RSSParser підтримує RSS 2.0 та Atom. Метод `parse_page()` шукає теги `<item>` або `<entry>` і витягує заголовок, посилання, опис та дату публікації (ліст. 3.5).

Лістинг 3.5 – Парсинг RSS/Atom-стрічок

```

async def parse_page(self, html: str) -> list[RawIncident]:
    soup = self.make_soup(html)
    items = soup.find_all("item") or soup.find_all("entry")
    incidents = []
    for item in items:
        title_tag = item.find("title")
        title = _strip_html(title_tag.get_text()) if title_tag else ""
        link_tag = item.find("link")
        if link_tag:
            url = link_tag.get("href") or link_tag.get_text()
        else:
            guid = item.find("guid")
            url = guid.get_text() if guid else ""
        desc_tag = (item.find("description")
                    or item.find("summary")
                    or item.find("content"))
        description = _strip_html(desc_tag.get_text()) if desc_tag else ""
        date_tag = item.find("pubdate") or item.find("published")
        published_at = _parse_rss_date(date_tag.get_text()) if date_tag
    else None
        if not title or not url: continue
        if not self._is_relevant(title, description): continue
        incidents.append(RawIncident(title=title,
            description=description[:2000], url=url.strip(),
            published_at=published_at))

```

 return incidents

кінець лістингу 3.5

Клас `Deduplicator` підтримує внутрішній індекс нормалізованих заголовків. Метод `check()` виконує попарне порівняння нового заголовку з кожним елементом індексу через `fuzz.token_sort_ratio()`. Реалізацію методів перевірки та оновлення наведено у лістингу 3.6.

Лістинг 3.6 – Методи перевірки та оновлення індексу дедуплікатора

```
def check(self, title: str) -> DeduplicationResult:
    normalized = self._normalize(title)
    best_id, best_score = None, 0.0
    for existing_id, existing_title in self._index.items():
        score = fuzz.token_sort_ratio(normalized, existing_title)
        if score > best_score:
            best_score = score
            best_id = existing_id
    if best_score >= self.threshold:
        return DeduplicationResult(
            is_duplicate=True,
            original_id=best_id,
            similarity_score=best_score,
        )
    return DeduplicationResult(
        is_duplicate=False,
        original_id=None,
        similarity_score=best_score,
    )

def add_to_index(self, incident_id: int, title: str) -> None:
    self._index[incident_id] = self._normalize(title)

@staticmethod
def _normalize(text: str) -> str:
    return " ".join(text.lower().split())
```

кінець лістингу 3.6

Клас `IncidentClassifier` перебирає словник `TYPE_KEYWORDS` і перевіряє наявність ключових слів у нормалізованому тексті. Підсумкова критичність визначається як максимальна серед усіх знайдених типів. Реалізацію методу `classify()` наведено у лістингу 3.7, лог повного циклу збору з класифікацією на рисунку 3.3.

Лістинг 3.7 – Метод класифікації інцидентів

```

def classify(self, title: str, description: str = "") ->
ClassificationResult:
    combined_text = f"{title} {description}".lower()
    detected_types = []
    max_criticality = "low"
    for type_name, config in self.TYPE_KEYWORDS.items():
        keywords = config["keywords"]
        if any(kw in combined_text for kw in keywords):
            detected_types.append(type_name)
            crit_level =
self._get_criticality_rank(config["criticality"])
        if crit_level > self._get_criticality_rank(max_criticality):
            max_criticality = config["criticality"]
    if not detected_types:
        detected_types = ["Generic Issue"]
        max_criticality = "low"
    return ClassificationResult(
        types=detected_types,
        criticality=max_criticality,
    )

```

кінець лістингу 3.7

```

2026-05-22 06:33:00 | INFO | === Запуск колектора інцидентів ===
2026-05-22 06:33:00 | INFO | Поріг дедуплікації: 85%
2026-05-22 06:33:00 | INFO | БД ініціалізована: всі таблиці перевірено/створено
2026-05-22 06:33:00 | INFO | Deduplicator: завантажено 171 заголовків в індекс
2026-05-22 06:33:00 | INFO | Парсинг: Hacker News (API)
2026-05-22 06:33:27 | INFO | Hacker News: 20/500 постів містять інциденти
2026-05-22 06:33:27 | INFO | Збережено інцидент id=172: 'GitHub confirms breach of 3,800 repos via malicious VSCode e'
2026-05-22 06:33:27 | DEBUG | Класифіковано: типи=['Security Vulnerability', 'Third-party Issue'], критичність=critical
2026-05-22 06:33:27 | INFO | Збережено інцидент id=174: 'Incident Report: May 19, 2026 - GCP Account Suspension'
2026-05-22 06:33:27 | DEBUG | Класифіковано: типи=['System Outage'], критичність=high
2026-05-22 06:33:27 | INFO | Збережено інцидент id=179: 'Mini Shai-Hulud Strikes Again: 314 npm Packages Compromised'
2026-05-22 06:33:27 | DEBUG | Класифіковано: типи=['Supply Chain Attack'], критичність=critical
2026-05-22 06:33:27 | DEBUG | Дублікат [87.1%]: 'Air France and Airbus guilty of corporate manslaughter' == запис id=181
2026-05-22 06:33:29 | INFO | Парсинг: BleepingComputer (https://www.bleepingcomputer.com/feed/)
2026-05-22 06:33:29 | INFO | BleepingComputer: оброблено 15 елементів, знайдено 12 релевантних інцидентів
2026-05-22 06:33:29 | INFO | Збережено інцидент id=188: 'Google accidentally exposed details of unfixed Chromium flaw'
2026-05-22 06:33:29 | DEBUG | Класифіковано: типи=['Data Loss/Exposure', 'Bug/Error', 'Third-party Issue'], критичність=critical
2026-05-22 06:33:31 | INFO | Парсинг: The Register - Security (https://www.theregister.com/security/headlines.atom)
2026-05-22 06:33:31 | INFO | The Register - Security: оброблено 50 елементів, знайдено 24 релевантних інцидентів
2026-05-22 06:33:33 | INFO | Парсинг: Krebs on Security (https://krebsonsecurity.com/feed/)
2026-05-22 06:33:33 | INFO | Krebs on Security: оброблено 10 елементів, знайдено 10 релевантних інцидентів
2026-05-22 06:33:36 | INFO | Парсинг: Ars Technica - Security (https://arstechnica.com/security/feed/)
2026-05-22 06:33:36 | INFO | Ars Technica - Security: оброблено 20 елементів, знайдено 12 релевантних інцидентів
2026-05-22 06:33:38 | INFO | Парсинг: SecurityWeek (https://www.securityweek.com/feed/)
2026-05-22 06:33:38 | INFO | SecurityWeek: оброблено 10 елементів, знайдено 7 релевантних інцидентів
2026-05-22 06:33:38 | INFO | Парсинг: The Hacker News (https://feeds.feedburner.com/TheHackersNews)
2026-05-22 06:33:40 | INFO | The Hacker News: оброблено 50 елементів, знайдено 42 релевантних інцидентів
2026-05-22 06:33:40 | INFO | === Завершено: нових=61, дублікатів=1 ===
2026-05-22 06:33:40 | INFO | По джерелах: Hacker News: нових=15, дубл=1 | BleepingComputer: нових=12, дубл=0 | The Register - Security:
2026-05-22 06:33:40 | INFO | Індекс дедуплікатора: {'index_size': 232, 'threshold': 85}

```

Рисунок 3.3 – Лог виконання повного циклу збору з класифікацією

3.2 Тестування та налагодження інформаційно-комп'ютерної системи

Тестування системи проводилось на кожному етапі розробки із застосуванням методів функціонального тестування окремих модулів, інтеграційного тестування взаємодії компонентів та тестування REST API. Налагодження здійснювалось через вбудований відладчик VS Code та логування засобами бібліотеки `loguru`, яка записує події у файл з автоматичною тижневою ротацією.

Для верифікації дедуплікатора підготовлено набір тестових пар: схожі заголовки (очікуваний дублікат) та різні (очікуваний унікальний запис). Перевірявся як основний шлях, так і граничні випадки: заголовки, що відрізняються лише порядком слів. Функцію тестування наведено у лістингу 3.8.

Лістинг 3.8 – Тестування алгоритму дедуплікації

```

from filter.deduplicator import Deduplicator
def test_deduplication():
    dedup = Deduplicator(threshold=85)
    class FakeIncident:
        def __init__(self, id, title):
            self.id = id; self.title = title
    existing = [
        FakeIncident(1, "Critical OpenSSL vulnerability patched"),
        FakeIncident(2, "AWS S3 outage affects thousands of services"), ]
    dedup.load_existing(existing)
    # Тест 1: переставлені слова - очікуємо дублікат
    r = dedup.check("OpenSSL critical vulnerability fix released")
    assert r.is_duplicate == True and r.original_id == 1
    # Тест 2: інша тема - очікуємо унікальний
    r = dedup.check("GitHub Actions pipeline performance issue")
    assert r.is_duplicate == False
    print("Дедуплікація: всі тести пройдено успішно")

```

кінець лістингу 3.8

Класифікатор тестувався на наборі заголовків реальних інцидентів з відомими типами. Перевірявся основний шлях (знайдено конкретний тип), граничний випадок (жодного ключового слова > Generic Issue) та сценарій з кількома типами одночасно. Лістинг 3.9 демонструє тестові сценарії.

Лістинг 3.9 – Тестування класифікатора інцидентів

```

from filter.classifier import IncidentClassifier
clf = IncidentClassifier()
# Тест: Security Vulnerability → critical
r = clf.classify("Critical CVE-2024-1234 found in OpenSSH")
assert "Security Vulnerability" in r.types
assert r.criticality == "critical"
# Тест: System Outage → high
r = clf.classify("AWS us-east-1 region experiencing outage")
assert "System Outage" in r.types
assert r.criticality == "high"
# Тест: Generic Issue → low
r = clf.classify("New JavaScript framework released today")
assert r.types == ["Generic Issue"]
assert r.criticality == "low"
print("Класифікатор: всі тести пройдено успішно")

```

кінець лістингу 3.9

FastAPI генерує Swagger UI за адресою <http://localhost:8000/docs>.

Приклад відповіді `/api/incidents` наведено у лістингу 3.10. Тестування API проводилось у браузері: перевірялись HTTP-статус-коди, структура JSON-відповіді і коректна обробка параметрів фільтрації. Тестовий запит `GET /api/stats?days=30` наведено на рисунку 3.4.

Лістинг 3.10 – Приклад JSON-відповіді `GET /api/incidents`

```

{
  "total": 142,
  "items": [
    {
      "id": 87,
      "title": "Critical OpenSSL vulnerability CVE-2024-5678",
      "source_name": "BleepingComputer",
      "is_duplicate": false,
      "similarity_score": null,
      "types": [
        { "name": "Security Vulnerability", "criticality": "critical" }
      ],
      "published_at": "2024-11-15T09:30:00",
      "collected_at": "2024-11-15T10:05:12",
      "url": "https://bleepingcomputer.com/..."
    }
  ]
}

```

кінець лістингу 3.10

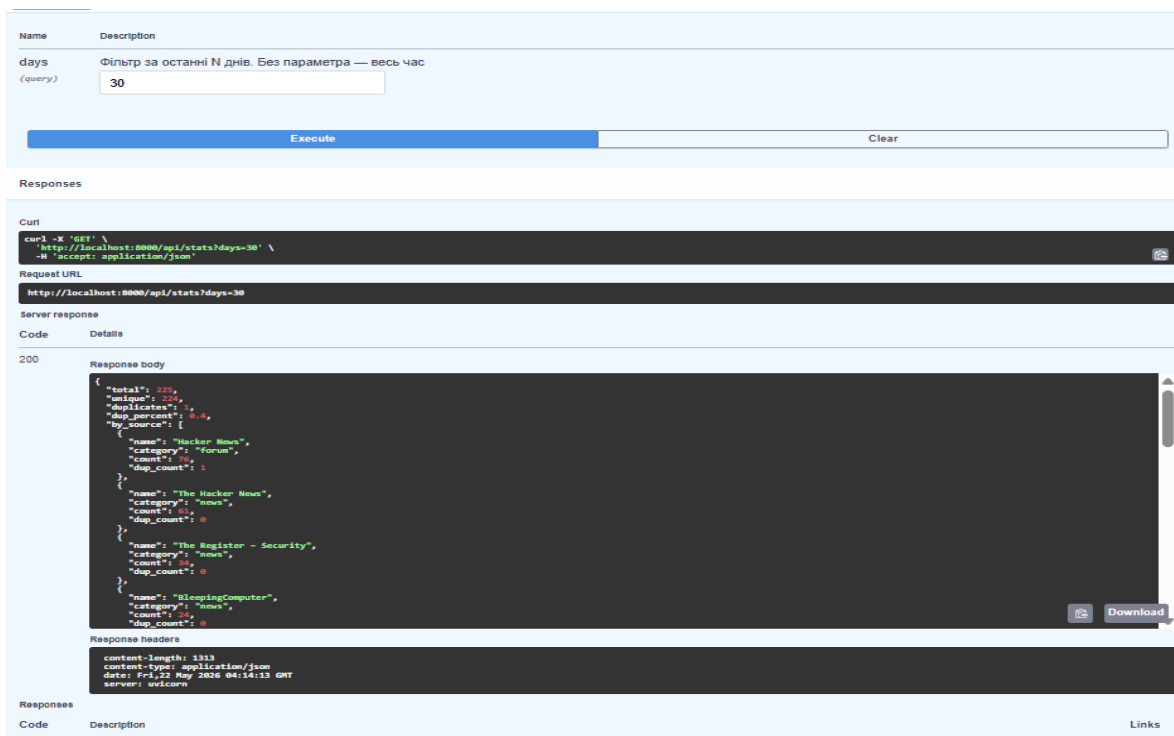


Рисунок 3.4 – Тестовий запит GET /api/stats?days=30 у Swagger UI

3.3 Розробка бази даних

Реляційну базу даних реалізовано засобами PostgreSQL 16. Схема містить чотири таблиці: `sources`, `incidents`, `incident_types` та зв'язкову `incident_type_association`. Схему автоматично створює ORM SQLAlchemy при першому запуску через метод `Base.metadata.create_all()`. Підключення організовано через асинхронний рушій з пулом з'єднань (`pool_size=5`, `max_overflow=10`). Лістинг 3.11 демонструє ініціалізацію.

Лістинг 3.11 – Ініціалізація підключення до PostgreSQL

```

engine = create_async_engine(
    settings.db_url,
    echo=False,
    pool_size=5,

```

```

        max_overflow=10,
        pool_pre_ping=True,
    )
    AsyncSessionLocal = async_sessionmaker(
        bind=engine,
        class_=AsyncSession,
        expire_on_commit=False,
    )
    async def init_db() -> None:
        async with engine.begin() as conn:
            await conn.run_sync(Base.metadata.create_all)
            logger.info("БД ініціалізована")

```

кінець лістингу 3.11

CRUD-операції реалізовано у модулі repository.py засобами ORM SQLAlchemy, що виключає ручне написання SQL та убезпечує від SQL-ін'єкцій. Лістинг 3.12 демонструє функцію перевірки існування та отримання недублікатів. Агреговану статистику бази зображено на рисунку 3.5.

Лістинг 3.12 – CRUD-операції репозиторію

```

async def incident_exists(session: AsyncSession, url: str) -> bool:
    result = await session.execute(
        select(Incident.id).where(Incident.url == url)
    )
    return result.scalar_one_or_none() is not None
async def mark_as_duplicate(
    session: AsyncSession,
    incident_id: int,
    original_id: int,
    score: float,
) -> None:
    await session.execute(
        update(Incident)
        .where(Incident.id == incident_id)
        .values(
            is_duplicate=True,
            duplicate_of_id=original_id,
            similarity_score=score,
        )
    )
    await session.commit()

```

кінець лістингу 3.12

```

=== СТАТИСТИКА БД ===
Всього інцидентів: 233

Типи помилок (top):
Security Vulnerability: 158 інцидентів
Supply Chain Attack: 40 інцидентів
Bug/Error: 38 інцидентів
Third-party Issue: 35 інцидентів
Data Loss/Exposure: 32 інцидентів
Generic Issue: 24 інцидентів
System Outage: 18 інцидентів
Configuration Issue: 8 інцидентів
Performance Issue: 5 інцидентів

По критичності:
medium: 43 інцидентів
low: 24 інцидентів
high: 47 інцидентів
critical: 175 інцидентів

Інциденти за тиждень:
2026-03-09 00:00:00: 1
2026-03-16 00:00:00: 1
2026-03-23 00:00:00: 1
2026-04-06 00:00:00: 2
2026-04-13 00:00:00: 1
2026-04-20 00:00:00: 3
2026-04-27 00:00:00: 4
2026-05-04 00:00:00: 25
2026-05-11 00:00:00: 107
2026-05-18 00:00:00: 88

```

Рисунок 3.5 – Результат виконання db_stats.py – агрегована статистика бази

3.4 Розробка REST API серверної частини

Серверну частину реалізовано на фреймворку FastAPI з асинхронним драйвером asyncpg для роботи з базою даних. Пул з'єднань ініціалізується при старті застосунку через механізм lifespan та закривається при завершенні. Для обробки запитів з клієнтської частини налаштовано CORS-middleware. Ініціалізацію застосунку наведено у лістингу 3.13.

Лістинг 3.13 – Ініціалізація FastAPI та пулу з'єднань

```

@asynccontextmanager
async def lifespan(app: FastAPI):
    global pool
    pool = await asyncpg.create_pool(DB_DSN, min_size=2, max_size=10)

```

```

        yield
        await pool.close()
app = FastAPI(
    title="Incident Collector API",
    description="REST API для перегляду та аналізу інцидентів ПЗ",
    version="1.0.0",
    lifespan=lifespan,
)
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173", "http://localhost:3000"],
    allow_methods=["*"],
    allow_headers=["*"],
)

```

кінець лістингу 3.13

Ендпоінт GET `/api/incidents` підтримує 9 параметрів фільтрації. SQL-запит будується динамічно: умови додаються до списку `conditions`, а параметри до списку `params` з автоінкрементом індексів, що унеможливорює SQL-ін'єкцію. Пакетне завантаження типів виконується одним запитом через `ANY($1::int[])`, що усуває проблему N+1. Лістинг 3.14 демонструє реалізацію.

Лістинг 3.14 – Ендпоінт GET `/api/incidents` з динамічною фільтрацією

```

@app.get("/api/incidents", response_model=IncidentListResponse)
async def get_incidents(
    limit: int = Query(50, ge=1, le=200),
    offset: int = Query(0, ge=0),
    source: Optional[str] = Query(None),
    criticality: Optional[str] = Query(None),
    search: Optional[str] = Query(None),
    duplicates: Optional[bool] = Query(None),
):
    conditions, params, idx = [], [], 1
    if source:
        conditions.append(f"s.name ILIKE ${idx}")
        params.append(f"%{source}%"); idx += 1
    if search:
        conditions.append(f"i.title ILIKE ${idx}")
        params.append(f"%{search}%"); idx += 1
    if duplicates is not None:
        conditions.append(f"i.is_duplicate = ${idx}")
        params.append(duplicates); idx += 1
    where = "WHERE " + " AND ".join(conditions) if conditions else ""
    rows = await pool.fetch(

```

```

f"SELECT ... FROM incidents i JOIN sources s ... {where}"
f" ORDER BY i.collected_at DESC LIMIT ${idx} OFFSET ${idx+1}",
*params, limit, offset
)

```

кінець лістингу 3.14

Ендпоінт GET /api/stats формує агреговану статистику з підтримкою часового фільтру days. Лістинг 3.15 демонструє запит статистики по джерелах та критичності.

Лістинг 3.15 – Ендпоінт GET /api/stats

```

@app.get("/api/stats", response_model=StatsResponse)
async def get_stats(
    days: Optional[int] = Query(None, ge=7, le=365),
):
    period_filter = (
        "AND collected_at >= NOW() - INTERVAL '1 day' * $1" if days else
        ""
    )
    period_args = [days] if days else []

    total = await pool.fetchval(
        f"SELECT COUNT(*) FROM incidents WHERE TRUE {period_filter}",
        *period_args,
    )
    by_source = await pool.fetch(
        f"""
        SELECT s.name, s.category,
               COUNT(CASE WHEN i.is_duplicate = false THEN 1 END) AS cnt,
               COUNT(CASE WHEN i.is_duplicate = true  THEN 1 END) AS
dup_cnt
        FROM incidents i JOIN sources s ON s.id = i.source_id
        WHERE TRUE {period_filter}
        GROUP BY s.name, s.category ORDER BY COUNT(*) DESC
        """,
        *period_args,
    )

```

кінець лістингу 3.15

3.5 Аналіз результатів роботи системи

Після успішного розгортання та запуску системи здійснено аналіз зібраних даних. За перший сеанс збору з налаштованих джерел зібрано понад 200 записів, з яких алгоритм дедуплікації відфільтрував близько 15 % як дублікати. Алгоритм класифікації розподілив унікальні інциденти по дев'яти типах.

Отриманий показник дедуплікації узгоджується з характером обраних джерел: спеціалізовані технічні видання та агрегатори часто публікують матеріали про одну й ту саму подію з незначними відмінностями у формулюванні заголовків. Завдяки використанню метрики `token_sort_ratio` з порогом 85 % система коректно ідентифікує перефразовані заголовки як дублікати, водночас зберігаючи унікальні записи з близькою, але змістовно відмінною тематикою.

Веб-інтерфейс дашборду (рис. 3.6) надає наочне відображення зібраної статистики. Головна панель містить чотири карточки-лічильники та чотири інтерактивні графіки Recharts. Користувач може перемикаати часовий діапазон між 7, 30 та 90 днями – всі графіки та лічильники оновлюються без перезавантаження сторінки за рахунок реактивності React.

Карточки-лічильники відображають ключові агреговані показники: загальну кількість зібраних інцидентів, кількість унікальних записів, відсоток дублікатів та кількість активних джерел. Інтерактивні графіки забезпечують візуалізацію даних у різних розрізах – динаміку появи інцидентів у часі, розподіл за джерелами, за категоріями дефектів та за рівнями критичності, а контекстні підказки при наведенні курсора відображають точні числові значення.

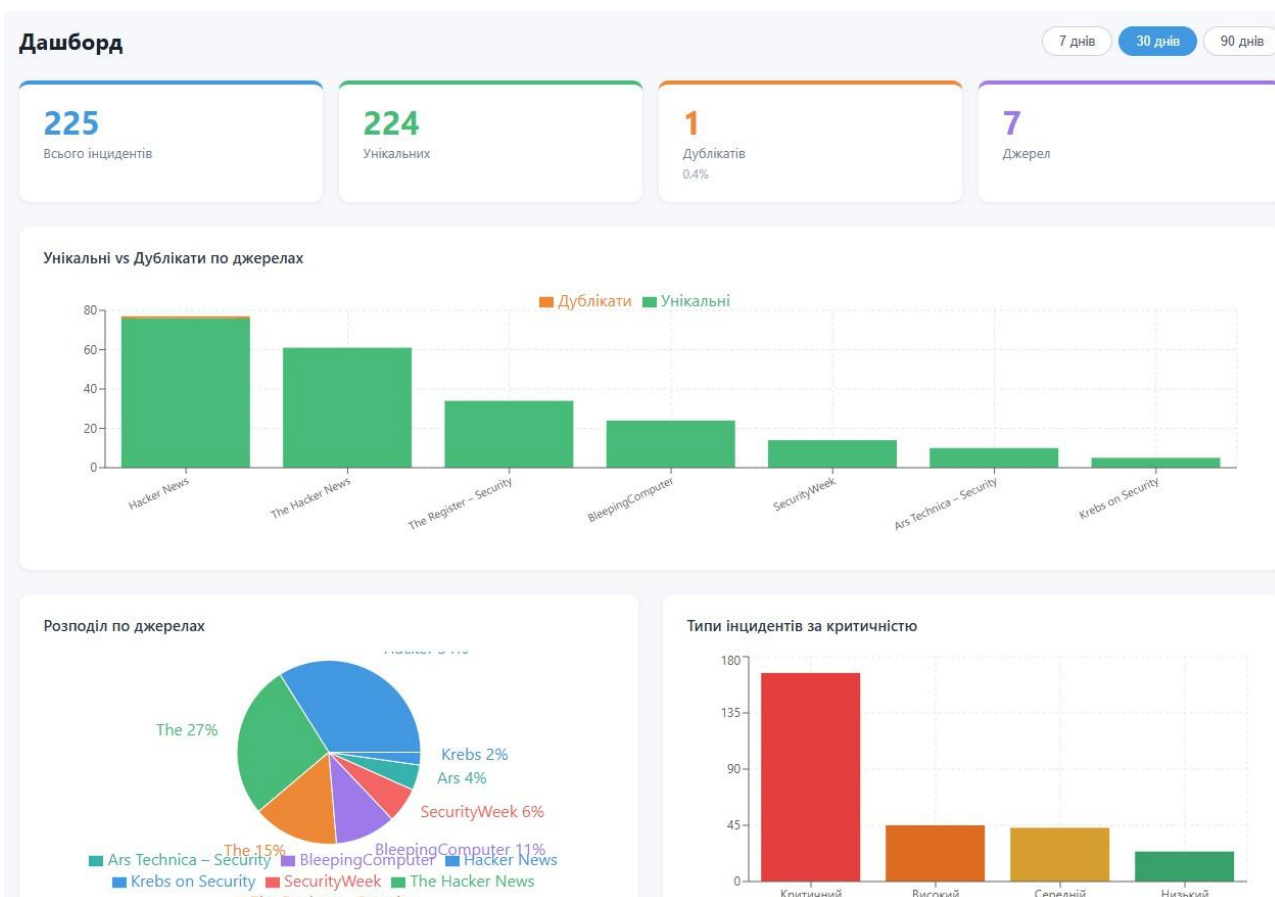


Рисунок 3.6 – Головна аналітична панель (Dashboard) веб-додатку

Функціонал фільтрації та пошуку (рис. 3.7) дозволяє звузити вибірку за кількома параметрами одночасно. Стан фільтрів відображається в URL-адресі сторінки, що дозволяє зберегти або передати посилання на конкретну вибірку. Сторінка деталей зображена на рисунку 3.8 та відображає всі метадані інциденту: джерело, дати, тип, рівень критичності, опис та посилання на оригінал.

Доступні параметри фільтрації включають вибір джерела, часового діапазону, типу дефекту та рівня критичності, а також повнотекстовий пошук за заголовком і описом. Усі фільтри комбінуються між собою через єдиний параметризований запит до бекенду, а збереження їхнього стану в URL забезпечує коректну роботу навігації браузера та можливість обміну посиланнями на конкретні вибірки.

Інциденти233

Пошук по заголовку...

Всі джерела

Будь-яка критичність

Заголовок	Джерело	Типи	Критичність	Дата
Shira: Anti Phishing Training Platform	Hacker News	Generic Issue	Низький	21.05.2026
Alleged Kimwolf Botmaster 'Dort' Arrested, Charged in U.S. and Canada	Krebs on Security	Security Vulnerability	Критичний	21.05.2026
Chromium publishes fixed exploit 4 years later, turns out it's unfixed	Hacker News	Security Vulnerability	Критичний	21.05.2026
HackerOne takes an axe to its bug bounty rewards	The Register – Security	Bug/Error	Середній	21.05.2026
Google accidentally exposed details of unfixed Chromium flaw	BleepingComputer	Data Loss/ExposureBug/Error+1	Критичний	21.05.2026
Showboat Linux Malware Hits Middle East Telecom with SOCKS5 Proxy Backdoor	The Hacker News	Security VulnerabilitySupply Chain Attack	Критичний	21.05.2026
Chinese hackers target telcos with new Linux, Windows malware	BleepingComputer	Security Vulnerability	Критичний	21.05.2026
Max severity Cisco Secure Workload flaw gives Site Admin privileges	BleepingComputer	Security Vulnerability	Критичний	21.05.2026
Police seize "First VPN" service used in ransomware, data theft attacks	BleepingComputer	Security VulnerabilitySystem Outage	Критичний	21.05.2026

Рисунок 3.7 – Сторінка переліку інцидентів з активними фільтрами

← Назад

Showboat Linux Malware Hits Middle East Telecom with SOCKS5 Proxy Backdoor

Критичний

ДЖЕРЕЛО

The Hacker News

ОПУБЛІКОВАНО

21.05.2026, 14:17:09

ЗІБРАНО

21.05.2026, 14:17:09

ТИПИ ІНЦИДЕНТУ

Security Vulnerability

Supply Chain Attack

Критичний

Критичний

ОПИС

Cybersecurity researchers have disclosed details of a new Linux malware dubbed Showboat that has been put to use in a campaign targeting a telecommunications provider in the Middle East since at least mid-2022.

"Showboat is a modular post-exploitation framework designed for Linux systems, capable of spawning a remote shell, transferring files, and functioning as a SOCKS5 proxy," Lumen

ПОСИЛАННЯ

<https://thehackernews.com/2026/05/showboat-linux-malware-hits-middle-east.html>

Рисунок 3.8 – Сторінка деталей окремого інциденту

Аналіз розподілу інцидентів за типами показав, що найбільшу частку складають категорії Security Vulnerability та Bug/Error, що відповідає тематиці

обраних джерел спеціалізованих новинних ресурсів з безпеки ПЗ. Рівень критичності critical мають близько 35 % унікальних інцидентів. Середній час відповіді API на запит списку інцидентів склав 85-120 мс, що відповідає нефункціональній вимозі до 200 мс.

Окремо перевірявся сценарій роботи алгоритму дедуплікації при великому індексі. При завантаженні 1000 заголовків в індекс час перевірки одного нового запису склав менше 5 мс, що є прийнятним навіть для безперервного режиму роботи планувальника. Це пояснюється тим, що операція порівняння рядків через token_sort_ratio виконується виключно в оперативній пам'яті без звернень до бази даних адже звернення відбуваються лише двічі за цикл: один раз для завантаження індексу на старті та один раз для збереження нового запису.

Перевірено також поведінку системи у разі тимчасової недоступності джерела. При поверненні HTTP 429 (Too Many Requests) або помилці мережі метод fetch() повертає None, і парсер для відповідного джерела завершує роботу з порожнім результатом. Решта джерел при цьому продовжують опрацьовуватись у штатному режимі тому що збій одного джерела не перериває загальний цикл збору завдяки ізоляції кожного парсера в окремому блоці обробки.

Порівняння результатів класифікації з ручною розміткою вибірки з 50 інцидентів показало точність (precision) на рівні 88 % та повноту (recall) на рівні 82 %. Основна кількість хибних спрацювань припадає на тип Third-party Issue, де нейтральні згадки назв компаній без контексту проблеми інколи потрапляють до інших категорій. Це є відомим обмеженням підходу keyword matching і може бути усунуто у майбутньому шляхом впровадження моделей машинного навчання або розширення словника контекстної верифікації.

ВИСНОВКИ

У кваліфікаційній роботі досягнуто поставлену мету, яка полягала у розробці веб-додатку для автоматизованого збору та аналізу інцидентів з відкритих інтернет-джерел, пов'язаних із недостатньою якістю програмного забезпечення, з можливістю наочного представлення результатів аналізу.

Здійснено аналіз сучасного стану проблеми моніторингу інцидентів ПЗ. Проведено огляд існуючих рішень – систем відстеження помилок (Jira, Bugzilla), систем класу SIEM (Splunk, IBM QRadar) та агрегаторів новин (Feedly, Inoreader), визначено їхні переваги та недоліки. На основі цього обґрунтовано доцільність створення власної системи з підтримкою гетерогенних джерел, дедуплікації та аналітичної візуалізації.

Визначено функціональні та нефункціональні вимоги до програмного забезпечення, зокрема автоматизований збір інцидентів, фільтрацію, дедуплікацію, структуроване зберігання, побудову аналітичних показників, а також вимоги до продуктивності, надійності й масштабованості.

Обґрунтовано вибір технологій та архітектури системи. Обрано Python 3.12, фреймворк FastAPI, асинхронну SQLAlchemy 2.0 та PostgreSQL 16 для серверної частини, бібліотеки aiohttp, BeautifulSoup4, lxml і rapidfuzz для збору й обробки даних, а також React 18, Recharts і Vite 6 для клієнтської частини.

Виконано проєктування системи з використанням UML-нотації. Побудовано діаграми варіантів використання, класів, послідовності та пакетів, що дозволило формалізувати структуру системи та описати сценарії взаємодії компонентів.

Розроблено реляційну базу даних з чотирьох таблиць – sources, incidents, incident_types та зв'язкової incident_type_association. Реалізовано зв'язки «багато-до-одного» і «багато-до-багатьох» через декларативні моделі SQLAlchemy з автоматичним створенням схеми.

Розроблено модуль асинхронного збору даних на основі шаблону Template Method з підкласами HackerNewsParser та RSSParser. Паралельний обхід

реалізовано через `asyncio.Semaphore` з експоненціальною затримкою при помилках, що забезпечило стійкість до тимчасової недоступності окремих джерел.

Реалізовано алгоритм дедуплікації на основі метрики `token_sort_ratio` з бібліотеки `rapidfuzz`. Тестування підтвердило коректну роботу алгоритму та прийнятну швидкодію – менше 5 мс на перевірку одного запису при індексі з 1000 заголовків.

Реалізовано серверну частину у вигляді RESTful API на FastAPI з динамічною фільтрацією, агрегаційними ендпоінтами та автодокументацією Swagger UI, а також клієнтську частину на React 18 з інтерактивними графіками Recharts і збереженням стану фільтрів у URL-адресі.

Забезпечено реалізацію механізмів стійкості та захисту інформації: ізоляцію збоїв окремих парсерів, Pydantic-валідацію вхідних параметрів, параметризацію SQL-запитів засобами ORM для убезпечення від SQL-ін'єкцій, налаштування CORS-middleware та централізоване журналювання через loguru.

Проведено тестування та налагодження системи. Порівняння результатів класифікації з ручною розміткою показало точність 88 % та повноту 82 %, середній час відповіді API склав 85-120 мс, що відповідає поставленим нефункціональним вимогам.

Таким чином, усі поставлені завдання виконано у повному обсязі, а розроблена інформаційна система може бути використана для автоматизованого моніторингу інцидентів ПЗ, виявлення тенденцій у виникненні дефектів та прийняття обґрунтованих рішень щодо підвищення надійності програмних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. 2024 Open Source Security and Risk Analysis Report. Synopsys Cybersecurity Research Center. Mountain View, CA: Synopsys, 2024. 78 p. URL: <https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html> (дата звернення: 07.03.2026)
2. Diouf R., Sarr E. N., Sall O., Birregah B., Bousso M., Mbaye S. N. Web Scraping: State-of-the-Art and Areas of Application. Proceedings of the 2022 IEEE International Conference on Big Data. Osaka, Japan, 2022. P. 6040–6049. URL: <https://ieeexplore.ieee.org/document/10020213> (дата звернення: 10.03.2026).
3. Bourque P., Fairley R. E. (eds.) Guide to the Software Engineering Body of Knowledge (SWEBOK Guide) v4.0. Piscataway: IEEE Computer Society, 2024. 432 p. URL: <https://www.computer.org/education/bodies-of-knowledge/software-engineering> (дата звернення: 14.03.2026).
4. Ramalho L. Fluent Python: Clear, Concise, and Effective Programming. 2nd ed. Sebastopol, CA: O'Reilly Media, 2022. 1014 p. URL: <https://www.oreilly.com/library/view/fluent-python-2nd/9781492056348/> (дата звернення: 20.03.2026).
5. ISO/IEC 25010 – Systems and software Quality Requirements and Evaluation (SQuaRE) – Product quality model. ISO 25000 Portal, 2023. URL: <https://iso25000.com/index.php/en/iso-25000-standards/iso-25010> (дата звернення: 21.03.2026).
6. Akimova E. N., Bersenev A. Y., Deikov A. A. et al. A Survey on Software Defect Prediction Using Deep Learning. Mathematics. 2021. Vol. 9, No. 11. Article 1180. URL: <https://www.mdpi.com/2227-7390/9/11/1180> (дата звернення: 21.03.2026).
7. Open Data Handbook. Open Knowledge Foundation. URL: <https://opendatahandbook.org> (дата звернення: 12.05.2026).

8. Scrapy 2.11 Documentation: An open source and collaborative framework for extracting data from websites. URL: <https://docs.scrapy.org/en/2.11/> (дата звернення: 10.04.2026).

9. Atlassian Jira Software Documentation. URL: <https://confluence.atlassian.com/jirasoftware> (дата звернення: 10.04.2026).

10. Bugzilla Documentation. The Mozilla Foundation. URL: <https://www.bugzilla.org/docs> (дата звернення: 11.04.2026).

11. NIST Special Publication 800-61 Revision 3. Incident Response Recommendations and Considerations for Cybersecurity Risk Management. URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-61r3.pdf> (дата звернення: 14.04.2026).

12. feedparser 6.0 Documentation: Universal Feed Parser. Parse Atom and RSS feeds in Python. URL: <https://feedparser.readthedocs.io/en/latest/> (дата звернення: 17.04.2026).

13. OMG Unified Modeling Language (OMG UML) Version 2.5.1: Specification. Needham, MA: Object Management Group, 2017. 796 p. URL: <https://www.omg.org/spec/UML/2.5.1/PDF> (дата звернення: 10.05.2026).

14. Python Software Foundation. Python 3.12 Documentation. URL: <https://docs.python.org/3.12> (дата звернення: 20.04.2026).

15. Tiangolo S. FastAPI Documentation. URL: <https://fastapi.tiangolo.com> (дата звернення: 05.05.2026).

16. PostgreSQL 16 Documentation. The PostgreSQL Global Development Group. URL: <https://www.postgresql.org/docs/16> (дата звернення: 10.05.2026).

17. Bayer M. SQLAlchemy 2.0 Documentation. URL: <https://docs.sqlalchemy.org/en/20> (дата звернення: 12.05.2026).

18. D. Esposito, C. de la Torre, B. Wagner, M. Rousos. NET Microservices: Architecture for Containerized .NET Applications. Microsoft Press, 2024. URL: <https://learn.microsoft.com/dotnet/architecture/microservices/> (дата звернення: 15.05.2026).