

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами

Кафедра програмного забезпечення автоматизованих систем

Спеціальність 121 «Інженерія програмного забезпечення»

Освітня програма «Інженерія програмного забезпечення»

«Допущений до захисту»

Завідувач кафедри



— проф. Приходько С.Б.

«___» _____ 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття ступеня вищої освіти «бакалавр»

на тему: Розробка програмної системи для автоматизованого збору та обробки даних про вразливості програмного забезпечення з використанням конвеєра даних

Виконав: студент групи 4157зст3



Андреєв Артем Дмитрович

(підпис, ПІБ)

Керівник роботи:

доктор філософії



(посадка, науковий ступень вчене звання)

Трухов Артем Сергійович

(підпис, ПІБ)

Миколаїв – 2026 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет кораблебудування імені адмірала Макарова

Навчально-науковий інститут комп'ютерних наук та управління проєктами
 Кафедра програмного забезпечення автоматизованих систем
 Спеціальність 121 «Інженерія програмного забезпечення»
 Освітня програма «Інженерія програмного забезпечення»



«ЗАТВЕРДЖУЮ»

Гарант освітньої програми

доц. Макарова Л.М.

(підпис)

« 07 » 04 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ
на здобуття ступеня вищої освіти «бакалавр»

Студенту Андрєєву Артему Дмитровичу

(Прізвище, ім'я, по батькові)

1. Тема роботи: Розробка програмної системи для автоматизованого збору та обробки даних про вразливості програмного забезпечення з використанням конвеєра даних

Керівник роботи Артем Трухов Сергійович

Затверджені наказом ректора №275-уч від 07.04.2026 р.

2. Термін подання роботи: 01.06.2026 р.

3. Вихідні дані по роботі: _____

—

4. Перелік питань, що належать до розробки (найменування розділів)

Титульний аркуш; Завдання на кваліфікаційну роботу; Анотація (українською та англійською мовами); Зміст; Перелік умовних позначень, символів, одиниць та термінів (при необхідності); Вступ; Аналіз практичної задачі інженерії програмного забезпечення; Проєкт програмного забезпечення; Результати розробки програмного забезпечення; Розділ з охорони праці; Висновки; Список використаних джерел; Додатки (технічне завдання, текст програми, опис програми, інструкція користувача, програма та методика випробувань програмного забезпечення).

5. Перелік презентаційних матеріалів 1 Титульний слайд, 2 Мета та завдання дослідження, 3 Аналіз існуючого програмного забезпечення, 4 Моделювання вимог (Діаграма варіантів використання), 5 Архітектура конвеєра обробки даних, 6 Проєктування реляційної моделі даних, 7 Об'єктна структура програмної системи, 8 Технологічний стек проєкту, 9 Реалізація логіки конвеєра даних, 10 Результати функціонування та верифікація системи, 11 Результат прямого вивантаження зі сховища PostgreSQL, 12 Висновки за результатами роботи, 13 Фінальний слайд

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
4	Гурець Н.В., ст. викладач		

7. Дата видачі завдання 07.04.2026 р.

КАЛЕНДАРНИЙ ПЛАН

Номер	Назва етапів роботи	Терміни виконання	Примітка
1.	Підготовка розділу ВСТУП	30.03.2026	ПП*
2.	Аналіз практичної задачі інженерії ПЗ	06.04.2026	ПП*
3.	Аналіз вимог до ПЗ	13.04.2026	ПП*
4.	Розробка архітектури ПЗ	20.04.2026	
5.	Розробка проекту ПЗ	04.05.2026	
6.	Реалізація та тестування ПЗ	11.05.2026	
7.	Підготовка розділу Результати розробки ПЗ	15.05.2026	
8.	Підготовка розділу з охорони праці	18.05.2026	ПП*
9.	Підготовка розділу ВИСНОВКИ	22.05.2026	
10.	Оформлення списку використаних джерел та додатків	25.05.2026	
11.	Подання на кафедру ПЗАС тексту остаточного варіанту роботи, підписаного її керівником, у роздрукованому та електронному форматі разом із заявами щодо самостійності виконання роботи та ідентичності друкованої та електронної версій роботи (Додатки 1 і 2 «Порядку здійснення заходів з перевірки робіт на наявність текстових збігів/ідентичності/схожості із використанням програмно-технічних засобів», який введений в дію наказом ректора НУК за №20 від 20.01.2020 р.)	01.06.2026	не пізніше, ніж за 14 діб до захисту (згідно п.4.1 зазначеного Порядку
12.	Підготовка презентації та доповіді	05.06.2026	
13.	Попередній захист роботи на засіданні кафедри ПЗАС	08.06.2026	
14.	Подання на кафедру ПЗАС електронних версій наступних документів у форматі pdf: кваліфікаційної роботи; файлу-опису кваліфікаційної роботи (згідно Додатку до наказу ректора НУК за №287-уч від 19.05.2020 р.); презентації доповіді, а також Авторського договору з додатком	15.06.2026	

* - за результатами переддипломної практики (ПП), яка була з 02.02.2026 до 22.03.2026 р.

Студент

(підпис)

Андрєєв А.Д.

(ПІБ)

Керівник роботи

(підпис)

Трухов А.С.

(ПІБ)

АНОТАЦІЯ

У кваліфікаційній роботі розв'язано практичну задачу інженерії програмного забезпечення, пов'язану з автоматизацією збору, обробки та зберігання даних про кібервразливості. Метою роботи є розробка програмної системи, яка забезпечує інкрементальне отримання відомостей із National Vulnerability Database, очищення та трансформацію JSON-даних, збереження результатів у реляційній базі PostgreSQL і автоматизовану оркестрацію процесів за допомогою Apache Airflow.

У першому розділі проаналізовано предметну область моніторингу вразливостей, особливості NVD API та існуючі програмні аналоги. У другому розділі визначено функціональні й нефункціональні вимоги, побудовано діаграми проєктування, розроблено архітектуру ETL-конвеєра та реляційну модель даних. У третьому розділі описано практичну реалізацію модулів вилучення, трансформації та завантаження даних, налаштування DAG в Apache Airflow, а також результати тестування й верифікації роботи системи. Четвертий розділ присвячено питанням охорони праці під час роботи з екранними пристроями.

Об'єктом роботи є процес розробки програмного забезпечення для автоматизованого збору, обробки та зберігання даних про кібервразливості.

Кваліфікаційна робота викладена на 103 сторінках друкованого тексту та містить 16 рисунків, 3 таблиці, список використаних джерел з 20 найменувань і 5 додатків.

ABSTRACT

This qualification work addresses a practical software engineering problem related to the automation of collecting, processing, and storing data on cybersecurity vulnerabilities. The aim of the work is to develop a software system that provides incremental retrieval of vulnerability data from the National Vulnerability Database, cleaning and transformation of JSON data, storage of the results in the PostgreSQL relational database, and automated orchestration of the processing workflow using Apache Airflow.

The first section analyzes the subject area of vulnerability monitoring, the specifics of the NVD API, and existing software analogues. The second section defines functional and non-functional requirements, presents design diagrams, and describes the architecture of the ETL pipeline and the relational data model. The third section describes the practical implementation of the data extraction, transformation, and loading modules, the configuration of the Apache Airflow DAG, and the results of system testing and verification. The fourth section is devoted to occupational health and safety issues related to work with display screen equipment.

The object of the work is the process of developing software for the automated collection, processing, and storage of data on cybersecurity vulnerabilities.

The qualification work is presented on 103 pages of printed text and contains 16 figures, 3 tables, a list of 20 references, and 5 appendices.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ.....	8
ВСТУП.....	10
1 АНАЛІЗ ПРАКТИЧНОЇ ЗАДАЧІ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЗАЦІЇ ЗБОРУ ДАНИХ ЩОДО КІБЕРЗАГРОЗ.....	14
1.1 Аналіз практичної задачі інженерії програмного забезпечення.....	14
1.2 Аналіз аналогічного існуючого програмного забезпечення та обґрунтування необхідності розробки.....	18
1.3 Постановка задачі на розробку програмного забезпечення.....	26
2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА КОМПОНЕНТІВ ПРОГРАМНОЇ СИСТЕМИ АВТОМАТИЗОВАНОГО ЗБОРУ І ОБРОБКИ ДАНИХ.....	31
2.1 Аналіз вимог.....	31
2.2 Розробка архітектури програмного забезпечення.....	36
2.3 Розробка проєкту програмного забезпечення та моделювання бази даних.....	40
3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ МОНІТОРИНГУ КІБЕРВРАЗЛИВОСТЕЙ....	47
3.1 Вибір та обґрунтування технологічних засобів розробки.....	47
3.2 Розробка схеми бази даних та структури зберігання.....	48
3.3 Програмна реалізація модулів конвеєра даних (ETL).....	49
3.4 Налаштування оркестрації процесів в Apache Airflow.....	51
3.5 Тестування та перевірка працездатності системи.....	53
3.6 Результати експериментальних досліджень та верифікації працездатності системи.....	54
4 ОХОРОНА ПРАЦІ.....	58
4.1 Аналіз умов праці аналітика безпеки при роботі з екранними пристроями.....	58

4.2 Ергономічні вимоги до організації робочого місця та режиму праці.....	59
4.3 Електробезпека та пожежна безпека у приміщенні з обчислювальною технікою.....	60
ВИСНОВКИ.....	62
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	64
ДОДАТОК А ТЕХНІЧНЕ ЗАВДАННЯ НА РОЗРОБКУ ПРОГРАМНОЇ СИСТЕМИ.....	66
ДОДАТОК Б ТЕКСТ ПРОГРАМИ.....	73
ДОДАТОК В ОПИС ПРОГРАМИ.....	96
ДОДАТОК Г ІНСТРУКЦІЯ КОРИСТУВАЧА.....	98
ДОДАТОК Д ПРОГРАМА ТА МЕТОДИКА ВИПРОБУВАНЬ.....	102

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

ПС АЗОДВ – програмна система автоматизованого збору та обробки даних про вразливості.

API (Application Programming Interface) – програмний інтерфейс застосунку, набір правил та засобів для взаємодії між програмними системами.

CPE (Common Platform Enumeration) – стандартизована система ідентифікації програмного забезпечення, операційних систем та апаратних компонентів.

CVE (Common Vulnerabilities and Exposures) – міжнародний стандарт ідентифікації відомих вразливостей та загроз інформаційної безпеки.

CVSS (Common Vulnerability Scoring System) – стандартна система оцінювання рівня небезпеки та критичності вразливостей.

DAG (Directed Acyclic Graph) – орієнтований ациклічний граф, що використовується в Apache Airflow для опису послідовності виконання задач.

DBMS (Database Management System) – система керування базами даних.

ETL (Extract, Transform, Load) – процес вилучення, перетворення та завантаження даних.

HTTP (HyperText Transfer Protocol) – протокол передачі гіпертекстових даних у комп'ютерних мережах.

JSON (JavaScript Object Notation) – текстовий формат обміну структурованими даними.

NIST (National Institute of Standards and Technology) – Національний інститут стандартів і технологій США.

NVD (National Vulnerability Database) – Національна база даних вразливостей, що підтримується NIST.

ORM (Object-Relational Mapping) – технологія відображення об'єктів програмного коду на структури реляційної бази даних.

Pandas (Python Data Analysis Library) – програмна бібліотека мови Python

для аналізу та обробки даних.

REST (Representational State Transfer) – архітектурний стиль побудови вебсервісів.

SecOps (Security Operations) – підхід до організації процесів інформаційної безпеки, що поєднує експлуатацію інформаційних систем та заходи кіберзахисту.

UPSERT (Update + Insert) – операція бази даних, яка виконує вставку нового запису або оновлення існуючого у разі конфлікту ключів.

URL (Uniform Resource Locator) – уніфікований покажчик ресурсу в мережі Інтернет.

UTF-8 (Unicode Transformation Format 8-bit) – стандарт кодування символів Unicode.

Data Pipeline – програмний конвеєр автоматизованого переміщення та обробки даних між джерелами та сховищами.

Apache Airflow – програмна платформа для автоматизації, планування та моніторингу конвеєрів обробки даних.

ВСТУП

У сучасних умовах стрімкого розвитку інформаційних технологій та цифрової трансформації суспільства питання забезпечення інформаційної безпеки набувають особливої актуальності. Зростання кількості програмних продуктів, вебсервісів, хмарних платформ та розподілених інформаційних систем супроводжується постійним збільшенням кількості потенційних векторів атак і кіберзагроз. Унаслідок ускладнення архітектури програмного забезпечення та скорочення циклів його розробки регулярно виявляються нові вразливості, які можуть бути використані зловмисниками для несанкціонованого доступу до інформаційних ресурсів, компрометації конфіденційних даних, порушення цілісності інформації або відмови в обслуговуванні інформаційних систем.

Ефективна протидія сучасним кіберзагрозам потребує своєчасного отримання та аналізу інформації про виявлені вразливості програмного забезпечення. Для розробників програмного забезпечення, системних адміністраторів, фахівців із кібербезпеки та команд SecOps важливим завданням є постійний моніторинг актуальних відомостей про нові загрози та критичні вразливості. Одним із найбільш авторитетних джерел такої інформації є Національна база даних вразливостей (National Vulnerability Database, NVD), яка підтримується Національним інститутом стандартів і технологій США (NIST). База даних NVD містить відомості про відомі кібервразливості, кожній з яких присвоюється унікальний ідентифікатор відповідно до міжнародного стандарту Common Vulnerabilities and Exposures (CVE).

Практична задача інженерії програмного забезпечення, що розглядається у кваліфікаційній роботі, полягає в автоматизації процесу збору, обробки та зберігання великих обсягів інформації про кібервразливості з відкритих джерел даних. Дана задача характеризується комплексністю та невизначеністю умов, оскільки інформація надходить із зовнішнього джерела

у вигляді складних JSON-документів зі змінною структурою, а процес отримання даних супроводжується мережевими обмеженнями, змінами в API та необхідністю регулярної синхронізації великих масивів інформації. Ручна обробка таких даних є неефективною та не забезпечує своєчасного отримання актуальної інформації про нові загрози.

Незважаючи на наявність спеціалізованих систем управління вразливостями, таких як Tenable, Qualys, Rapid7 або OpenVAS, більшість із них орієнтовані на корпоративний сегмент ринку, потребують придбання ліцензій або не забезпечують достатньої гнучкості щодо інтеграції з власними аналітичними процесами організації. Крім того, багато рішень зосереджені переважно на виявленні вразливостей у локальній інфраструктурі, тоді як задача автоматизованого збору та накопичення відкритих даних про вразливості потребує окремого програмного забезпечення. У зв'язку з цим актуальним є розроблення спеціалізованої програмної системи, яка забезпечуватиме автоматизований збір, очищення, трансформацію та зберігання даних про кібервразливості з використанням сучасних технологій обробки даних.

Для вирішення зазначеної задачі доцільним є використання концепції конвеєрів даних (Data Pipeline), яка передбачає автоматизоване виконання етапів вилучення, перетворення та завантаження даних (Extract–Transform–Load, ETL). Використання такого підходу дозволяє забезпечити актуальність даних, зменшити вплив людського фактора, підвищити надійність процесу обробки інформації та створити централізоване сховище даних для подальшого аналізу.

Метою кваліфікаційної роботи є розробка програмної системи автоматизованого збору та обробки даних про кібервразливості з використанням конвеєра даних.

Для досягнення поставленої мети необхідно вирішити такі завдання:

- проаналізувати предметну область моніторингу кібервразливостей та дослідити особливості структури даних National Vulnerability Database;

- проаналізувати існуючі програмні рішення для збору та обробки інформації про вразливості та визначити їх переваги й недоліки;
- визначити функціональні та нефункціональні вимоги до програмної системи;
- розробити архітектуру програмної системи та структуру конвеєра даних;
- розробити реляційну модель бази даних для зберігання інформації про вразливості та пов'язані програмні продукти;
- реалізувати програмні модулі вилучення, перетворення та завантаження даних із використанням засобів Python та бібліотеки Pandas;
- реалізувати механізм інкрементального оновлення даних і автоматизовану оркестрацію процесів за допомогою Apache Airflow;
- провести тестування програмної системи та оцінити результати її функціонування;
- розробити комплект супровідної документації, що включає опис програми, інструкцію користувача та програму і методику випробувань.

Об'єктом дослідження є процес розробки програмного забезпечення для автоматизованого збору, обробки та зберігання даних про кібервразливості.

Предметом дослідження є методи, алгоритми та програмні засоби автоматизації процесів вилучення, трансформації та завантаження даних про кібервразливості на основі технологій Python, Pandas, PostgreSQL та Apache Airflow.

Практичне значення одержаних результатів полягає у створенні програмної системи, яка забезпечує автоматичне отримання актуальних даних про кібервразливості з відкритих джерел, їх очищення, структурування та збереження в реляційній базі даних. Використання розробленого рішення дозволяє зменшити вплив людського фактора, скоротити час обробки інформації, знизити навантаження на зовнішні ресурси завдяки механізму інкрементального оновлення та забезпечити можливість подальшого аналітичного опрацювання даних засобами SQL.

Результати кваліфікаційної роботи можуть бути використані для автоматизації процесів моніторингу кібервразливостей у навчальних, дослідницьких та виробничих проєктах, а також як основа для подальшого розвитку систем аналізу кіберзагроз.

1 АНАЛІЗ ПРАКТИЧНОЇ ЗАДАЧІ ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ АВТОМАТИЗАЦІЇ ЗБОРУ ДАНИХ ЩОДО КІБЕРЗАГРОЗ

1.1 Аналіз практичної задачі інженерії програмного забезпечення

У сучасному інформаційному суспільстві забезпечення стійкості корпоративних комп'ютерних мереж, серверних застосунків, вебсервісів і хмарних платформ до зовнішніх та внутрішніх кіберзагроз є одним із ключових напрямів інженерії програмного забезпечення. Розвиток цифрових сервісів, перехід підприємств до хмарних і гібридних інфраструктур, активне використання відкритих бібліотек, контейнеризації та мікросервісних архітектур істотно ускладнюють програмний ландшафт організацій. Чим складнішою стає програмна система, тим більшою є кількість потенційних дефектів, помилок конфігурації, несумісностей версій та вразливих залежностей.

Кібервразливість у програмному забезпеченні є дефектом або слабким місцем, яке може бути використане зловмисником для порушення конфіденційності, цілісності або доступності інформаційної системи. Такі вразливості можуть виникати на різних рівнях: у прикладному коді, сторонніх бібліотеках, операційних системах, сервісах автентифікації, мережевих компонентах, контейнерних образах або конфігураційних файлах. У разі несвоєчасного реагування навіть одна критична вразливість може стати початковою точкою атаки, що призведе до несанкціонованого доступу до даних, розповсюдження шкідливого програмного забезпечення, порушення роботи сервісів або повної зупинки бізнес-процесів.

Для побудови ефективної системи кіберзахисту недостатньо лише періодично перевіряти локальну інфраструктуру. Командам розробників, системним адміністраторам, фахівцям SecOps та аналітикам безпеки потрібне постійне джерело актуальної інформації про нові та змінені вразливості.

Одним із найважливіших відкритих джерел таких даних є National Vulnerability Database (NVD), яка надає доступ до відомостей про відомі кібервразливості через вебінтерфейс та програмний інтерфейс NVD API [1]. Кожному запису про вразливість відповідає ідентифікатор CVE, що використовується як загальноприйняте позначення конкретного дефекту безпеки [2].

Запис CVE у джерелі NVD не є простим текстовим повідомленням. Він може містити ідентифікатор, опис різними мовами, дату публікації, дату останньої модифікації, метрики критичності, посилання на додаткові матеріали, інформацію про конфігурації та перелік уразливих програмних продуктів. Для оцінювання рівня небезпеки застосовується система CVSS, яка дозволяє подати критичність вразливості у вигляді числової оцінки та якісного рівня небезпеки [3]. Для формального опису уразливих платформ і продуктів використовується CPE, що дає змогу ідентифікувати виробника, назву продукту, версію та інші параметри програмного або апаратного компонента [4].

Саме така насиченість даних робить задачу обробки вразливостей не лише інформаційною, а й інженерною. Дані NVD мають напівструктурований характер і передаються у форматі JSON. З одного боку, JSON є зручним форматом для обміну даними між програмними системами. З іншого боку, структура відповіді API містить вкладені об'єкти, масиви, альтернативні поля, списки мовних описів, різні версії метрик CVSS та складні конфігураційні блоки. Для людини така структура є незручною для ручного аналізу, а для програмної системи потребує спеціальної логіки деструктуризації, очищення, перевірки та нормалізації.

Практична складність полягає також у тому, що дані про вразливості не є статичними. Записи можуть не лише додаватися, а й оновлюватися після первинної публікації. Наприклад, для вразливості може бути уточнено опис, змінено оцінку CVSS, додано нові уразливі продукти або оновлено посилання на зовнішні джерела. Тому одноразове завантаження повного набору даних не

розв'язує задачу підтримання актуального сховища. Необхідно забезпечити регулярну синхронізацію з джерелом та зберігати інформацію про момент останнього успішного оновлення.

З цієї причини ключовим для розроблюваної програмної системи є інкрементальний підхід до збору даних. Його сутність полягає в тому, що система не завантажує весь обсяг інформації щоразу, а отримує лише нові або змінені записи за певний часовий проміжок. Такий підхід дає змогу зменшити обсяг мережевого трафіку, скоротити час виконання конвеєра, знизити навантаження на зовнішній API та мінімізувати ризик перевищення лімітів частоти запитів. Для цього програмна система повинна зберігати службову інформацію про останній успішний запуск і використовувати її під час формування наступного запиту до NVD API.

Додатковим фактором невизначеності є взаємодія із зовнішнім вебсервісом. На відміну від локального файлу або внутрішньої бази даних, API NVD не контролюється розробником програмної системи. Під час отримання даних можуть виникати мережеві затримки, таймаути, тимчасова недоступність сервера, неповні відповіді або обмеження частоти запитів. У разі перевищення допустимої кількості звернень сервер може повернути HTTP-статус 429 Too Many Requests. Така ситуація не повинна призводити до аварійного завершення всього процесу. Навпаки, система має коректно розпізнавати цей стан, виконувати очікування, повторювати запит і фіксувати всі важливі події в журналі виконання.

Не менш важливою є проблема якості даних. Навіть якщо API повернув коректну відповідь, окремі записи можуть мати неповні або нестандартні поля. У частині записів може бути відсутня оцінка CVSS, можуть відрізнятися версії метрик, у текстових описах можуть траплятися службові символи, переноси рядків або надлишкові пробіли. CPE-рядки також потребують додаткового розбору, оскільки інформація про виробника, продукт і версію зберігається в одному формалізованому рядку. Без попередньої валідації та очищення такі дані складно використовувати для подальшого SQL-аналізу.

З погляду інженерії програмного забезпечення практична задача полягає не просто у виконанні HTTP-запиту до зовнішнього API. Необхідно побудувати повний програмний конвеєр, який охоплює етапи вилучення, перетворення, завантаження та контролю результатів. На етапі вилучення система має сформувати коректні параметри запиту, отримати всі сторінки результатів і врахувати обмеження зовнішнього сервісу. На етапі перетворення необхідно привести сирий JSON до табличного вигляду, виділити основні поля CVE, розібрати CPE-рядки, очистити текстові значення та нормалізувати часові поля. На етапі завантаження потрібно зберегти дані в реляційній базі без дублювання, забезпечити транзакційність і підтримати оновлення вже наявних записів.

Ручне виконання описаних дій є неефективним з кількох причин. По-перше, обсяг даних є занадто великим для регулярного ручного аналізу. По-друге, інформація оновлюється нерівномірно, тому людина не може гарантувати своєчасне виявлення всіх змін. По-третє, ручне копіювання, фільтрація та перетворення JSON-даних створюють високий ризик помилок. По-четверте, без централізованого сховища складно будувати повторювані аналітичні запити, порівнювати результати в часі та контролювати повноту завантаження. Отже, задача потребує саме автоматизованого програмного рішення.

Для забезпечення практичної цінності програмна система повинна відповідати низці інженерних вимог. Вона має підтримувати інкрементальний збір даних, обробку Rate Limiting, очищення та нормалізацію JSON-структур, розбір CPE-рядків, збереження результатів у реляційній базі даних, журналювання виконання та можливість повторного запуску після помилки. Крім того, система повинна бути достатньо прозорою для супроводу: окремі етапи конвеєра мають бути розділені, а результат кожного запуску повинен фіксуватися у службовому журналі.

Важливим аспектом задачі є вибір форми зберігання результатів. Збереження даних у вигляді окремих JSON-файлів не забезпечує зручного

виконання аналітичних запитів, фільтрації за критичністю, групування за виробниками або пошуку за уразливими продуктами. Тому доцільним є використання реляційної бази даних, у якій основні відомості про CVE, уразливі продукти та журнал виконання конвеєра зберігаються у пов'язаних таблицях. Такий підхід дозволяє виконувати SQL-запити, будувати вибірки за датами, рівнем критичності, назвою продукту або виробником, а також контролювати дублювання записів.

Таким чином, практична задача інженерії програмного забезпечення в межах даної кваліфікаційної роботи полягає у проєктуванні та реалізації автоматизованої програмної системи, яка забезпечує регулярне отримання, очищення, структурування та збереження даних про кібервразливості з відкритого джерела NVD. Задача характеризується комплексністю через поєднання мережевої інтеграції, обробки напівструктурованих даних, забезпечення цілісності реляційного сховища та автоматизації регулярних процесів. Невизначеність умов проявляється у залежності від зовнішнього API, можливих змінах структури відповідей, обмеженнях частоти запитів, нестабільності мережі та нерівномірності оновлення записів про вразливості.

Отже, розробка програмної системи автоматизованого збору та обробки даних про кібервразливості є актуальною інженерною задачею. Її розв'язання дозволяє зменшити вплив людського фактора, підвищити оперативність отримання інформації про нові загрози, створити централізоване сховище структурованих даних і забезпечити основу для подальшого аналітичного використання накопиченої інформації.

1.2 Аналіз аналогічного існуючого програмного забезпечення та обґрунтування необхідності розробки

У сфері кібербезпеки існує значна кількість програмних рішень, призначених для виявлення, моніторингу, оцінювання та аналізу вразливостей програмного забезпечення. Частина таких систем орієнтована на активне

сканування комп'ютерних мереж та інфраструктури організації, інша частина забезпечує централізоване управління ризиками, а окремі інструменти призначені для локального накопичення відкритих даних про CVE-записи. Для визначення доцільності розробки власної програмної системи необхідно проаналізувати наявні аналоги, визначити їхні сильні та слабкі сторони, а також зіставити їх із практичною задачею, що розв'язується у кваліфікаційній роботі.

У межах даної роботи для порівняння було обрано три характерні програмні рішення: Tenable Nessus, Qualys VMDR та cve-search. Такий вибір є доцільним, оскільки ці системи належать до різних класів програмного забезпечення. Tenable Nessus представляє клас комерційних сканерів вразливостей, Qualys VMDR є прикладом корпоративної хмарної платформи управління вразливостями, а cve-search є відкритим програмним інструментом, найближчим за ідеєю до задачі локального пошуку та накопичення інформації про CVE. Завдяки цьому аналіз охоплює як готові комерційні системи, так і відкрите рішення, яке може бути використане як технічний орієнтир для розроблення власної програмної системи.

Першим розглянутим аналогом є Tenable Nessus. Це програмний продукт, призначений для оцінювання стану захищеності комп'ютерних мереж, серверів, робочих станцій, операційних систем, прикладних сервісів і вебзастосунків. Основний сценарій використання Nessus полягає в активному скануванні інфраструктури, виявленні відкритих портів, визначенні версій служб і зіставленні знайдених компонентів із базою відомих вразливостей. Приклад інтерфейсу аналітичної панелі системи Tenable Nessus наведено на рисунку 1.1 на наступній сторінці.

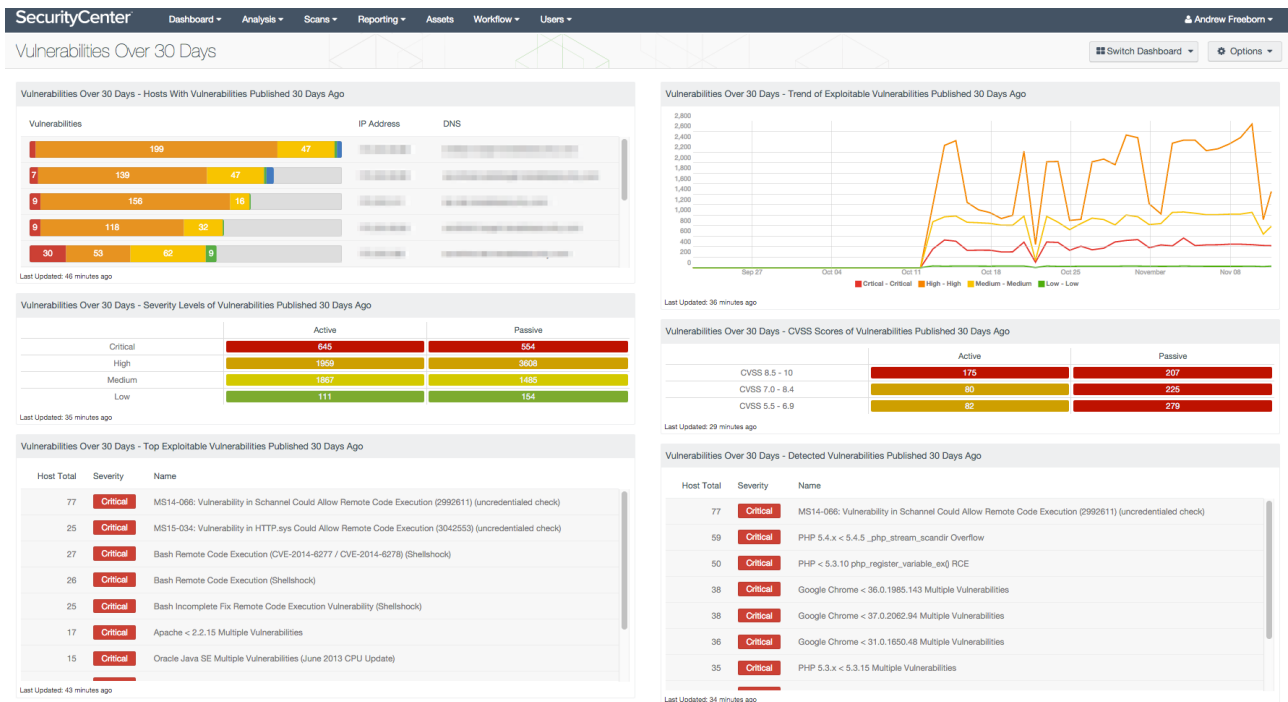


Рисунок 1.1 – Інтерфейс аналітичної панелі системи Tenable Nessus

До переваг Tenable Nessus можна віднести зрілий графічний інтерфейс, велику базу перевірок, підтримку різних сценаріїв сканування, можливість формування звітів і зручність використання для аудиту інформаційної безпеки. Такий інструмент є корисним для організацій, яким необхідно регулярно перевіряти власну інфраструктуру, оцінювати рівень ризику та отримувати практичні рекомендації щодо усунення виявлених проблем.

Водночас Tenable Nessus не повністю відповідає задачі, що розглядається в даній кваліфікаційній роботі. Його основна мета полягає у скануванні конкретної інфраструктури, а не у створенні власного централізованого сховища відкритих даних про кібервразливості. Користувач отримує результати перевірок у межах логіки самого продукту, однак не має повного контролю над внутрішньою структурою обробки даних. Для задачі побудови власного ETL-конвеєра важливо не тільки знайти вразливості в певній мережі, а й регулярно отримувати сирі дані з відкритого джерела, очищувати їх, нормалізувати та зберігати у реляційній базі даних для подальших SQL-запитів. Крім того, комерційна модель поширення Nessus може бути надмірною для навчального або дослідницького проєкту, де

важливими є відкритість, контрольованість і можливість модифікації алгоритмів.

Другим аналогом є Qualys VMDR. Це хмарна платформа, орієнтована на управління вразливостями, їх виявлення, оцінювання, пріоритезацію та підтримку процесів усунення ризиків у межах організації. На відміну від локального сканера, Qualys VMDR пропонує централізований підхід до управління кіберризиками, поєднуючи інвентаризацію активів, виявлення вразливостей, оцінювання критичності та формування аналітичних звітів. Приклад інтерфейсу хмарної платформи Qualys VMDR наведено на рисунку 1.2.



Рисунок 1.2 – Інтерфейс хмарної платформи Qualys VMDR

Перевагою Qualys VMDR є комплексність. Платформа дозволяє

організувати управління вразливостями на рівні підприємства, централізовано контролювати стан активів, аналізувати ризики та формувати звіти для фахівців із кібербезпеки. Такий підхід є ефективним для великих організацій, які мають значну кількість інформаційних систем, розподілену інфраструктуру та потребу в єдиному середовищі для моніторингу стану безпеки.

Однак у контексті даної роботи Qualys VMDR має низку обмежень. По-перше, це хмарна корпоративна платформа, використання якої передбачає залежність від інфраструктури постачальника. По-друге, користувач отримує готові результати аналізу, але не контролює повністю процес вилучення, трансформації та збереження первинних даних. По-третє, така система є надмірною для задачі побудови локального конвеєра обробки даних з відкритого API. У межах кваліфікаційної роботи необхідно не впровадити готову платформу управління ризиками, а спроектувати й реалізувати власну програмну систему, яка демонструє принципи інженерії даних, реляційного моделювання, транзакційного завантаження та автоматизованої оркестрації.

Третім розглянутим аналогом є cve-search. Це відкрите програмне рішення, призначене для локального пошуку відомостей про відомі вразливості. На відміну від Tenable Nessus і Qualys VMDR, цей інструмент ближчий до теми даної роботи, оскільки орієнтований саме на роботу з CVE-даними та може розгортатися локально. Структуру репозиторію відкритого проєкту cve-search наведено на рисунку 1.3 на наступній сторінці.

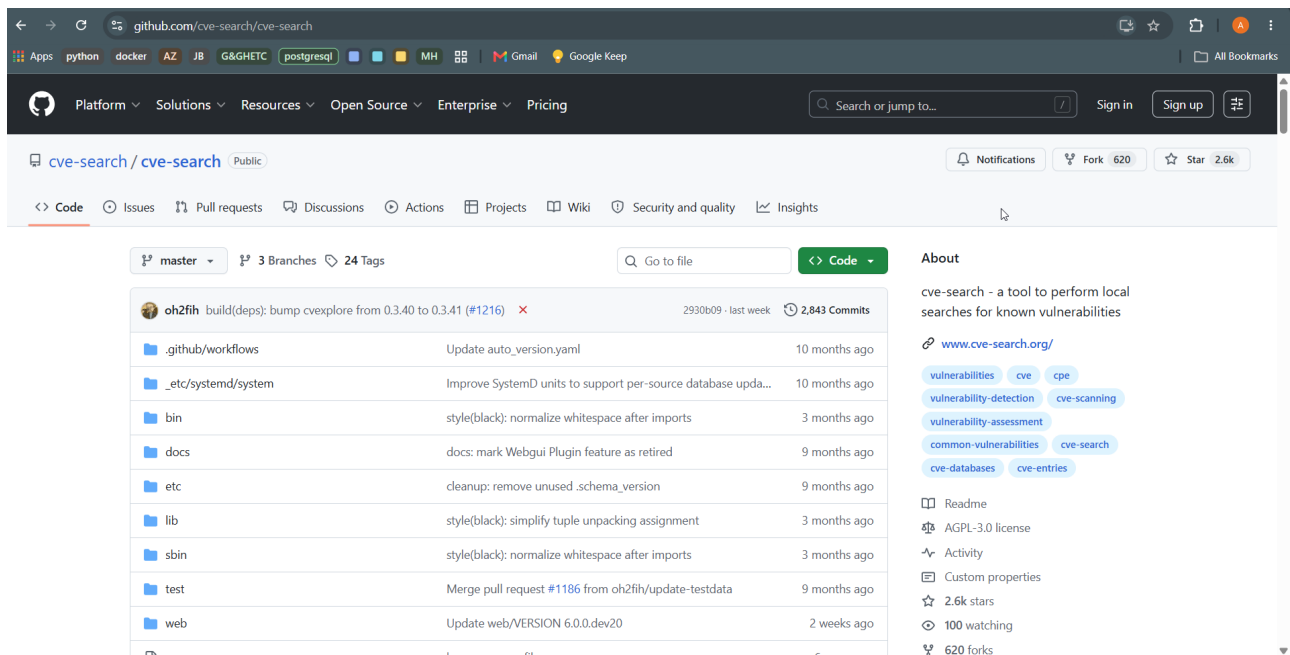


Рисунок 1.3 – Структура репозиторію відкритого проєкту cve-search

Основною перевагою cve-search є відкритий вихідний код і можливість локального використання. Такий інструмент може бути корисним для дослідників, аналітиків безпеки та розробників, яким потрібен локальний пошук за відомими CVE-записами. Відкритість коду дозволяє досліджувати внутрішню логіку роботи системи, змінювати окремі компоненти та адаптувати інструмент до певних сценаріїв використання.

Водночас cve-search також не повністю розв'язує задачу, поставлену в даній кваліфікаційній роботі. Його архітектура орієнтована на готовий інструмент локального пошуку, тоді як у роботі необхідно реалізувати саме навчально-практичну програмну систему із чітким поділом на етапи Extract, Transform і Load. Крім того, для цілей цієї роботи важливим є використання реляційного сховища PostgreSQL, що спрощує побудову аналітичних SQL-запитів, групування за виробниками, фільтрацію за критичністю та перевірку цілісності даних. У cve-search основний акцент зроблено на готовій системі пошуку, тоді як у розроблюваній системі акцент переноситься на керований конвеєр даних, прозору структуру бази, модульну архітектуру та оркестрацію виконання за допомогою Apache Airflow.

Для порівняння розглянутих систем було визначено низку критеріїв, що

мають безпосереднє значення для поставленої задачі. До таких критеріїв належать тип ліцензії, модель зберігання даних, спосіб оновлення інформації, наявність оркестрації процесів, можливість SQL-аналізу та здатність обробляти обмеження частоти запитів. Результати порівняння наведено в таблиці 1.1.

Таблиця 1.1 – Порівняльний аналіз систем обробки даних про вразливості

Критерій порівняння	Tenable Nessus	Qualys VMDR	cve-search	Розроблювана система
Тип ліцензії	Комерційна	Комерційна (SaaS)	Відкрита (Open Source)	Відкрита (Кастомна)
Модель зберігання	Пропрієтарна БД	Хмарне сховище	NoSQL (MongoDB)	Реляційна (PostgreSQL)
Спосіб оновлення	Повне/Плагіни	Автоматично хмарне	Лінійний скрипт	Інкрементальний ETL
Оркестрація процесів	Внутрішня закрита	Закрита хмарна	Відсутня (cron)	Apache Airflow (DAG)
Можливість SQL-аналізу	Відсутня	Обмежена через API	Відсутня	Повна (SQL, JOINS)
Обробка Rate Limiting	Не потрібна	Не потрібна	Слабкий таймаут	Повна автоматична

Проведений порівняльний аналіз показує, що кожен із розглянутих аналогів має власну сферу ефективного застосування. Tenable Nessus є сильним рішенням для активного сканування інфраструктури та виявлення вразливостей у конкретних вузлах мережі. Qualys VMDR є доцільним вибором для великих організацій, яким потрібна хмарна платформа управління

ризиками та централізованого контролю активів. cve-search є найближчим аналогом за ідеєю локальної роботи з CVE-даними, однак його використання не забезпечує повної відповідності вимогам даної роботи щодо реляційного аналізу, транзакційного завантаження та керованої оркестрації ETL-процесів.

Розроблювана програмна система відрізняється від наведених аналогів тим, що вона не намагається замінити повноцінну корпоративну платформу управління ризиками або мережевий сканер. Її основне призначення полягає в автоматизованому зборі, очищенні, нормалізації та зберіганні відкритих даних про кібервразливості у власному сховищі. Такий підхід дозволяє отримати контроль над усіма етапами обробки: від формування HTTP-запиту до NVD API до виконання аналітичного SQL-запиту в PostgreSQL.

Важливою перевагою власної реалізації є можливість адаптації структури бази даних до конкретних аналітичних завдань. У розроблюваній системі дані поділяються на основні записи CVE, пов'язані з ними уразливі програмні продукти та журнал виконання ETL-конвеєра. Це дає змогу не тільки зберігати інформацію, а й контролювати актуальність завантаження, перевіряти кількість оброблених записів, аналізувати критичні вразливості та виконувати вибірки за виробниками або продуктами.

Ще однією перевагою є використання Apache Airflow для оркестрації. На відміну від простого запуску скриптів вручну або через системний планувальник, Airflow дозволяє подати процес обробки у вигляді DAG, де кожна задача має власний статус, журнал виконання, правила повторного запуску та залежності від попередніх етапів. Це робить конвеєр більш прозорим, контрольованим і придатним для подальшого розвитку.

Отже, аналіз існуючих програмних рішень підтверджує доцільність розробки власної програмної системи. Комерційні продукти мають розвинені функції сканування та управління ризиками, але не забезпечують необхідної відкритості й контрольованості процесів обробки даних. Відкриті інструменти на зразок cve-search ближчі до поставленої задачі, але не повністю відповідають вимогам щодо реляційного зберігання, модульної ETL-

архітектури та автоматизованої оркестрації. Тому розробка власного рішення є обґрунтованою, оскільки дозволяє поєднати відкритість, гнучкість, контрольованість, реляційний аналіз і автоматизоване виконання конвеєра даних.

1.3 Постановка задачі на розробку програмного забезпечення

На основі проведеного аналізу предметної області, особливостей відкритих джерел інформації про кібервразливості та обмежень існуючих аналогічних систем сформульовано постановку задачі на розробку програмного забезпечення. Результати аналізу показали, що для ефективної роботи з даними про вразливості потрібен не окремий скрипт для завантаження JSON-файлів, а повноцінна програмна система, яка забезпечує регулярне отримання даних, їх очищення, нормалізацію, збереження та контроль результатів виконання.

Необхідно спроектувати та реалізувати програмну систему автоматизованого збору й обробки даних про кібервразливості з використанням конвеєра даних. Система має отримувати інформацію з NVD API, виконувати інкрементальне оновлення, перетворювати напівструктуровані JSON-дані у табличну форму, зберігати результати в реляційній базі PostgreSQL та забезпечувати автоматичну оркестрацію процесів за допомогою Apache Airflow.

Основним призначенням розроблюваної програмної системи є усунення ручної праці під час моніторингу оновлень відкритої бази даних вразливостей. Система повинна забезпечити можливість регулярного отримання актуальних CVE-записів, їх очищення від технічно зайвих або некоректних елементів, виділення ключових характеристик вразливостей і збереження у структурованому вигляді. Результатом роботи системи має бути база даних, придатна для виконання аналітичних SQL-запитів, формування звітів і подальшого використання аналітиком безпеки.

Вхідними даними для програмної системи є відповіді NVD API у форматі JSON. До складу таких відповідей входять ідентифікатори CVE, текстові описи, дати публікації та останньої модифікації, метрики критичності CVSS, конфігураційні блоки та CPE-рядки з описом уразливих продуктів. Оскільки структура JSON-відповіді має вкладений характер, система повинна виконувати її програмну деструктуризацію, відокремлюючи основні відомості про CVE від інформації про уразливі продукти.

Вихідними даними системи є структуровані записи в реляційній базі даних PostgreSQL. Основні відомості про вразливості мають зберігатися в таблиці `cve_details`, інформація про уразливі продукти — у таблиці `vulnerable_products`, а службова інформація про виконання конвеєра — у таблиці `etl_log`. Такий поділ дозволяє уникнути надмірного дублювання даних, підтримувати логічний зв'язок між CVE-записами та продуктами, а також контролювати актуальність наступних інкрементальних запусків.

Функціональні вимоги до програмної системи:

- система повинна автоматично отримувати дані з NVD API за допомогою HTTP-запитів;
- система повинна підтримувати інкрементальний режим завантаження, за якого обробляються лише нові або змінені записи після останнього успішного запуску;
- система повинна підтримувати пагінацію результатів, оскільки API може повертати дані частинами;
- система повинна коректно обробляти обмеження частоти запитів, зокрема ситуацію отримання HTTP-статусу 429 Too Many Requests;
- система повинна виконувати повторні спроби запиту у разі тимчасових мережових помилок;
- система повинна очищувати текстові описи CVE від службових символів, переносів рядків і надлишкових пробілів;
- система повинна виділяти англomовні описи вразливостей як основні для подальшого аналізу;

- система повинна вилучати значення CVSS і якісний рівень критичності, якщо такі дані наявні у відповіді API;
- система повинна виконувати розбір CPE-рядків для визначення виробника, назви продукту та версії програмного забезпечення;
- система повинна формувати два набори табличних даних: набір основних CVE-записів і набір уразливих продуктів;
- система повинна завантажувати очищені дані до PostgreSQL із використанням транзакцій;
- система повинна запобігати дублюванню CVE-записів за допомогою операції вставки або оновлення;
- система повинна фіксувати результат кожного запуску в журналі ETL, зокрема дату запуску, статус, кількість оброблених записів і часову мітку останнього успішного оновлення;
- система повинна забезпечувати можливість виконання аналітичних SQL-запитів до накопичених даних.

Нефункціональні вимоги до програмної системи:

- система повинна бути модульною, тобто мати окремі компоненти для вилучення, трансформації, завантаження, доступу до бази даних та оркестрації;
- система повинна бути стійкою до тимчасових мережових помилок і не повинна втрачати вже збережені дані у разі збою окремого запуску;
- система повинна забезпечувати транзакційну цілісність під час запису даних у PostgreSQL;
- система повинна підтримувати можливість повторного запуску після помилки без дублювання вже наявних записів;
- система повинна зберігати конфігураційні параметри, паролі та API-ключі поза програмним кодом, зокрема у змінних оточення;
- система повинна бути придатною для запуску у середовищі Linux або WSL, що відповідає практичним умовам розробки;
- система повинна мати зрозумілу структуру коду, придатну для

супроводу, тестування та подальшого розширення;

- система повинна забезпечувати прийнятну швидкість під час обробки пакетів CVE-записів завдяки використанню табличних структур даних і пакетного завантаження в базу.

До програмних засобів, необхідних для реалізації системи, належать мова Python, бібліотека Pandas для обробки та нормалізації даних, бібліотека requests для HTTP-взаємодії з NVD API, драйвер psycopg2 для доступу до PostgreSQL, СКБД PostgreSQL для зберігання результатів і Apache Airflow для автоматизації запуску ETL-конвеєра. Такий набір технологій відповідає задачі розробки програмної системи, оскільки охоплює всі необхідні етапи: отримання даних, їх обробку, збереження та планування регулярного виконання.

У межах постановки задачі також визначено низку обмежень і припущень. По-перше, джерелом даних є NVD API, тому система залежить від доступності зовнішнього сервісу та коректності його відповідей. По-друге, у межах даної роботи не передбачається створення повноцінного графічного інтерфейсу користувача, оскільки основний акцент зроблено на автоматизації конвеєра даних, структурі сховища та можливості SQL-аналізу. По-третє, система не виконує активного сканування інфраструктури організації, а працює з відкритими даними про вже відомі вразливості. По-четверте, результати роботи системи призначені насамперед для аналітика безпеки або розробника, який може виконувати SQL-запити до бази PostgreSQL.

Критеріями успішного виконання поставленої задачі є можливість запуску ETL-конвеєра за розкладом, успішне отримання даних із NVD API, формування очищених табличних структур, збереження результатів у PostgreSQL, відсутність дублювання CVE-записів, наявність журналу виконання та можливість отримання аналітичних результатів за допомогою SQL-запитів. Додатковим критерієм є наочність роботи системи через вебінтерфейс Apache Airflow, де можна переглядати стан DAG, результат виконання окремих задач і журнали запусків.

Таким чином, постановка задачі передбачає створення програмної системи, яка реалізує повний цикл автоматизованої обробки даних про кібервразливості: від отримання сирих JSON-відповідей із NVD API до збереження нормалізованих записів у PostgreSQL і подальшого використання цих даних для аналітичних запитів. Розроблюване програмне забезпечення має усунути недоліки ручного моніторингу, забезпечити повторюваність процесу, підвищити актуальність інформації та створити основу для подальшого розвитку системи аналізу кіберзагроз.

2 ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА КОМПОНЕНТІВ ПРОГРАМНОЇ СИСТЕМИ АВТОМАТИЗОВАНОГО ЗБОРУ І ОБРОБКИ ДАНИХ

2.1 Аналіз вимог

Проектування програмної системи автоматизованого збору та обробки даних про кібервразливість потребує попереднього визначення вимог до системи, її меж, основних сценаріїв використання та очікуваних результатів функціонування. Оскільки розроблюване програмне забезпечення має працювати з відкритим зовнішнім джерелом даних, виконувати регулярну обробку напівструктурованих JSON-відповідей і зберігати результати у реляційному сховищі, вимоги до нього охоплюють як функціональні, так і нефункціональні аспекти.

Функціональні вимоги визначають дії, які система повинна виконувати для досягнення поставленої мети. Нефункціональні вимоги описують якісні характеристики програмної системи: надійність, продуктивність, масштабованість, безпеку, супроводжуваність та придатність до подальшого розвитку. Для даної системи особливо важливо забезпечити стійкість до помилок зовнішнього API, контроль цілісності даних у базі PostgreSQL і можливість автоматизованого запуску ETL-конвеєра за допомогою Apache Airflow.

З урахуванням постановки задачі та аналізу предметної області було визначено такі функціональні вимоги до програмної системи:

- Керування сесіями збору даних (F-1). Система повинна автоматично визначати часову мітку останнього успішного запуску конвеєра та використовувати її для формування наступного інкрементального запиту до NVD API. Це дозволяє не завантажувати весь масив даних повторно, а отримувати лише нові або змінені записи.

- Взаємодія з віддаленим API NVD (F-2). Програмний модуль вилучення

повинен надсилати HTTP-запити до NVD API, передавати параметри часового вікна, підтримувати пагінацію результатів і поступово накопичувати всі сторінки відповіді. У разі наявності API-ключа система повинна передавати його у відповідному заголовку запиту.

- Обробка мережових помилок і обмежень частоти запитів (F-3). Система повинна коректно реагувати на тимчасову недоступність джерела даних, таймаути, помилки серверної сторони та HTTP-статус 429 Too Many Requests. У таких випадках програмна система має виконувати очікування, повторювати запит до визначеної кількості разів і фіксувати помилки в журналі виконання.

- Парсинг та очищення даних (F-4). Модуль трансформації повинен приймати сирі JSON-об'єкти, виділяти основні атрибути CVE-записів, очищувати текстові поля від службових символів, нормалізувати пробіли й переноси рядків, а також відбирати англomовні описи як основні для подальшого аналізу.

- Валідація бізнес-правил (F-5). Система повинна перевіряти наявність обов'язкових полів, зокрема ідентифікатора CVE, контролювати допустимість числових значень CVSS у межах від 0.0 до 10.0, а також перевіряти коректність часових полів, які використовуються під час інкрементального завантаження.

- Розбір CPE-рядків (F-6). програмна система повинна виконувати синтаксичний розбір CPE-рядків версії 2.3 для виділення назви виробника, назви продукту та версії програмного забезпечення. Ці дані необхідні для формування окремої таблиці уразливих продуктів.

- Транзакційне завантаження даних у сховище (F-7). Модуль завантаження повинен записувати очищені дані до PostgreSQL у межах транзакції. У разі успішного виконання всі зміни мають фіксуватися операцією Commit, а у разі помилки — повністю відкочуватися операцією Rollback. Для уникнення дублювання CVE-записів необхідно використовувати операцію вставки або оновлення.

- Журналювання виконання конвеєра (F-8). Після кожного запуску

система повинна записувати службову інформацію до таблиці журналу: дату запуску, статус виконання, кількість оброблених записів і часову мітку останнього успішно обробленого оновлення. Ці дані використовуються для контролю роботи системи та формування наступного інкрементального запиту.

– Аналітичний доступ до результатів (F-9). Після завантаження даних у PostgreSQL аналітик безпеки повинен мати можливість виконувати SQL-запити до накопиченої інформації, зокрема фільтрувати вразливості за рівнем критичності, датою публікації, виробником, назвою продукту або кількістю пов'язаних уразливих компонентів.

Нефункціональні вимоги до системи визначено таким чином.

– Надійність (NF-1). Помилка під час обробки одного запису або тимчасова недоступність зовнішнього API не повинні призводити до пошкодження вже збережених даних. Система має зберігати узгоджений стан сховища навіть у разі аварійного завершення окремого запуску.

– Відмовостійкість (NF-2). Конвеєр повинен підтримувати повторні спроби виконання окремих задач, зокрема під час мережових збоїв або перевищення ліміту запитів до API. Оркестрація в Apache Airflow повинна забезпечувати контроль статусу задач і можливість повторного запуску.

– Продуктивність (NF-3). Обробка пакетів CVE-записів повинна виконуватися за прийнятний час завдяки використанню табличних структур даних, пакетних операцій і мінімізації повторних звернень до зовнішнього API.

– Масштабованість (NF-4). Архітектура програмної системи повинна дозволяти розширення функціональності без повної переробки системи. Зокрема, у майбутньому може бути додано нові джерела даних, додаткові аналітичні таблиці або інші механізми звітності.

– Безпека конфігураційних даних (NF-5). Паролі до бази даних, ключі API та інші чутливі параметри не повинні зберігатися безпосередньо у вихідному коді. Їх необхідно передавати через змінні оточення або

конфігураційні механізми середовища виконання.

- Супроводжуваність (NF-6). Код системи повинен бути структурований за логічними модулями, що відповідають етапам ETL-конвеєра: вилучення, трансформація, завантаження, доступ до бази даних та оркестрація. Такий поділ спрощує тестування, налагодження та подальший розвиток системи.

- Переносимість (NF-7). Система повинна бути придатною для розгортання у середовищі Linux або WSL, оскільки саме такі середовища є типовими для запуску Apache Airflow, PostgreSQL і серверних Python-застосунків.

- Спостережуваність (NF-8). Користувач або аналітик повинен мати можливість перевірити стан виконання конвеєра через журнали Airflow, таблицю `etl_log` і результати SQL-запитів до PostgreSQL.

Для формального моделювання взаємодії зовнішніх акторів із розроблюваною програмною системою та визначення меж її функціональності було побудовано діаграму варіантів використання. Розроблену діаграму варіантів використання наведено на рисунку 2.1 на наступній сторінці.

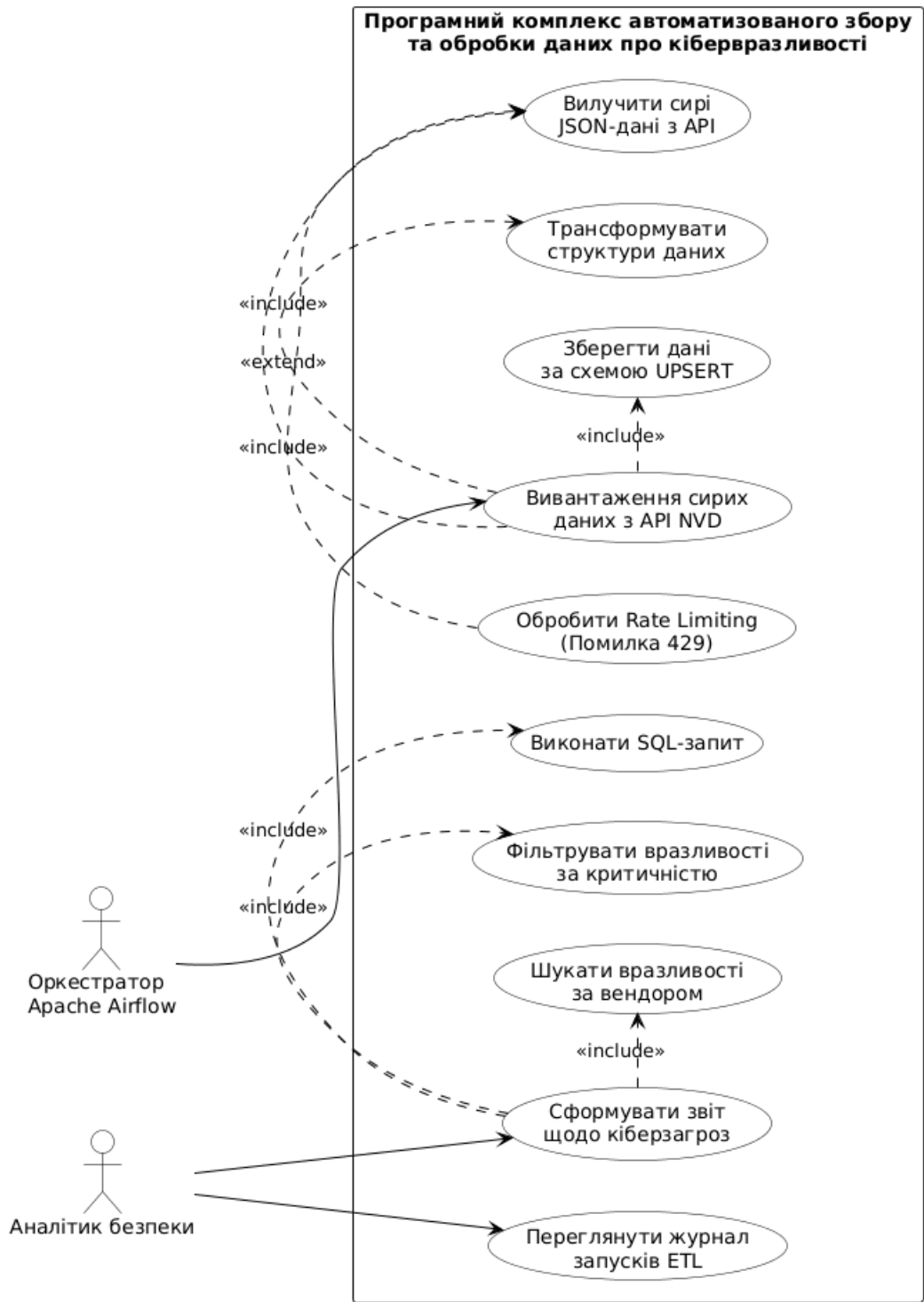


Рисунок 2.1 – Діаграма варіантів використання

На діаграмі варіантів використання виділено два ключові актори: «Оркестратор Apache Airflow» та «Аналітик безпеки». Перший актор є технічним і представляє механізм автоматичного запуску ETL-конвеєра за розкладом. Другий актор є кінцевим користувачем результатів роботи системи та використовує накопичені дані для аналітичних завдань.

Актор «Оркестратор Apache Airflow» ініціює основний сценарій роботи системи — «Вивантаження сирих даних з API NVD». Цей прецедент є комплексним і включає три обов'язкові підпроцеси: «Вилучити сирі JSON-дані з API», «Трансформувати структури даних» та «Зберегти дані за схемою UPSERT». Такий поділ відповідає логіці ETL-конвеєра, де кожен етап має власну відповідальність і виконується лише після успішного завершення попереднього.

Прецедент «Вилучити сирі JSON-дані з API» має розширення у вигляді прецеденту «Обробити Rate Limiting (Помилка 429)». Це означає, що додатковий сценарій очікування та повторного запиту активується не завжди, а лише за умови виникнення обмеження частоти звернень до NVD API. Така модель дозволяє відобразити винятковий сценарій, який є важливим для практичної роботи із зовнішнім вебсервісом.

Актор «Аналітик безпеки» не втручається у фонові процеси збору та трансформації даних, але використовує результати роботи системи. Для нього передбачено прецеденти «Переглянути журнал запусків ETL» і «Сформувати звіт щодо кіберзагроз». Перегляд журналу дозволяє контролювати актуальність сховища та статус останніх запусків. Формування звіту передбачає виконання SQL-запитів до PostgreSQL, пошук уразливостей за виробником або продуктом і фільтрацію записів за рівнем критичності.

Таким чином, діаграма варіантів використання на рисунку 2.1 чітко розмежовує автоматизовані фонові процеси збору та обробки даних і користувацькі сценарії доступу до результатів. Це дозволяє уточнити межі програмної системи, відокремити внутрішні процеси ETL-конвеєра від аналітичних дій користувача та сформувати основу для подальшого архітектурного проєктування.

2.2 Розробка архітектури програмного забезпечення

Архітектура розроблюваної програмної системи базується на концепції

конвеєра даних, що реалізує послідовність етапів Extract, Transform і Load. Такий підхід є доцільним для задачі автоматизованого збору даних про кібервразливості, оскільки вхідна інформація надходить із зовнішнього API у напівструктурованому форматі, потребує очищення та нормалізації, а після цього має бути збережена у реляційному сховищі для подальшого аналізу.

Загальна архітектура системи побудована за багаторівневим принципом. Це означає, що основні функції програмної системи розділено між окремими логічними шарами. Такий підхід зменшує зв'язаність компонентів, спрощує тестування та полегшує подальше розширення системи. У межах розроблюваного програмної системи виділено п'ять основних шарів: шар оркестрації, шар вилучення, шар трансформації, шар завантаження та шар бази даних.

Шар оркестрації відповідає за автоматичний запуск і контроль виконання конвеєра. У розроблюваній системі цю роль виконує Apache Airflow, де конвеєр подано у вигляді DAG. DAG описує порядок виконання задач, їхні залежності, правила повторного запуску, параметри затримки між повторними спробами та можливість моніторингу результатів через вебінтерфейс [15].

Шар вилучення відповідає за взаємодію із зовнішнім джерелом даних — NVD API. Його завданням є формування HTTP-запиту, передача параметрів часового вікна, підтримка пагінації, накопичення сторінок результатів і обробка виняткових ситуацій. Особливу увагу в цьому шарі приділено обробці Rate Limiting, оскільки зовнішній API може обмежувати частоту запитів.

Шар трансформації виконує перетворення сирих JSON-даних у структурований табличний формат. На цьому етапі відбувається виділення CVE-ідентифікатора, опису, дат публікації та модифікації, оцінки CVSS, рівня критичності та інформації про уразливі продукти. Оскільки JSON-відповідь NVD має вкладену структуру, трансформація включає розбір масивів, фільтрацію мовних описів, очищення текстових значень і парсинг CPE-рядків.

Шар завантаження відповідає за збереження очищених даних у

PostgreSQL. На цьому етапі виконується транзакційний запис основних CVE-записів, оновлення списку уразливих продуктів і фіксація результату запуску в таблиці журналу. Для уникнення дублювання використовується операція UPSERT, що дозволяє вставити новий запис або оновити наявний при конфлікті унікального ключа [8].

Шар бази даних забезпечує довготривале зберігання результатів і можливість аналітичного доступу до них. PostgreSQL використовується як реляційне сховище, у якому дані поділяються на три основні групи: відомості про CVE, інформацію про уразливі продукти та службовий журнал виконання ETL-конвеєра. Для оптимізації доступу до даних можуть використовуватися індекси, зокрема за датою останньої модифікації або ідентифікатором CVE [9].

Для узагальненого відображення основних потоків даних між зовнішнім джерелом, програмною системою, реляційним сховищем і користувачем було розроблено архітектурну схему системи. Вона показує, що джерелом вхідних даних є NVD API, обробка виконується автоматизованою системою, результати зберігаються у PostgreSQL, а аналітик безпеки отримує доступ до очищених даних через SQL-запити.

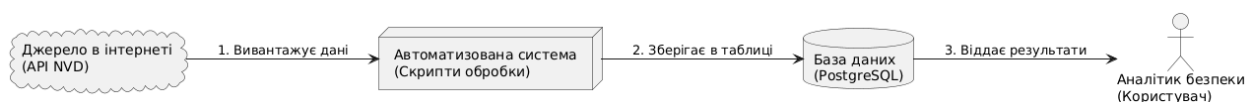


Рисунок 2.2 – Архітектурна схема системи

На рисунку 2.2 наведено архітектурну схему програмної системи автоматизованого збору та обробки даних про кібервразливості. Дана схема відображає загальний напрям руху інформації від зовнішнього джерела даних до кінцевого користувача та демонструє основні компоненти, які беруть участь у процесі обробки.

Початковим джерелом інформації є NVD API. З цього джерела програмна система отримує сирі відомості про кібервразливості у форматі JSON. Отримані дані передаються до автоматизованої системи, що містить

програмні модулі обробки та реалізує логіку ETL-конвеєра. На архітектурній схемі цей блок узагальнює внутрішні компоненти системи: API-клієнт, інкрементальний екстрактор, трансформер, валідатор, CPE-парсер, завантажувач і репозиторії доступу до бази даних.

Усередині автоматизованої системи виконуються основні етапи обробки даних: вилучення записів із зовнішнього API, очищення та нормалізація JSON-структур, виділення ключових атрибутів CVE, розбір CPE-рядків, формування структурованих наборів даних і підготовка результатів до збереження. Таким чином, автоматизована система виступає проміжним програмним шаром між зовнішнім джерелом даних і реляційним сховищем.

Після завершення обробки очищені та структуровані дані зберігаються у базі даних PostgreSQL. У реляційному сховищі накопичуються основні відомості про вразливості, пов'язані з ними уразливі програмні продукти та журнал виконання ETL-процесів. Використання PostgreSQL дозволяє забезпечити цілісність даних, уникати дублювання записів і виконувати аналітичні SQL-запити.

Кінцевим користувачем результатів роботи системи є аналітик безпеки. Він не бере участі у внутрішніх етапах збору та трансформації даних, однак отримує доступ до результатів, збережених у PostgreSQL. На основі накопичених даних аналітик може формувати звіти, виконувати фільтрацію вразливостей за рівнем критичності, аналізувати уразливі програмні продукти та контролювати актуальність інформації.

Важливо, що рисунок 2.2 відображає саме архітектурну схему та загальний потік даних, а не детальну послідовність викликів методів. Деталізація внутрішньої структури програмної системи виконується в підрозділі 2.3 за допомогою логічної схеми моделі даних і UML-діаграми класів.

Отже, архітектурна схема демонструє загальну логіку роботи програмної системи: зовнішнє джерело надає сирі дані, автоматизована система виконує їх обробку, PostgreSQL забезпечує збереження

структурованої інформації, а аналітик безпеки використовує отримані результати для подальшого аналізу кіберзагроз.

2.3 Розробка проєкту програмного забезпечення та моделювання бази даних

Після визначення вимог і загальної архітектури програмної системи необхідно перейти до детального проєктування структури даних та об'єктної моделі системи. На цьому етапі формується логічна схема бази даних, визначаються сутності предметної області, зв'язки між ними, ключові поля, обмеження цілісності та класи програмної реалізації.

Оскільки розроблювана програмна система працює з даними про кібервразливості, структура сховища повинна відображати основні інформаційні об'єкти предметної області. До таких об'єктів належать CVE-запис, уразливий програмний продукт і службовий запис журналу ETL-конвеєра. Для забезпечення тривалого зберігання, цілісності та можливості швидкого аналізу інформації було розроблено логічну схему моделі даних, що складається з трьох основних сутностей: `cve_details`, `vulnerable_products` та `etl_log` які продемонстровані у логічній схемі моделі даних на наступній сторінці.

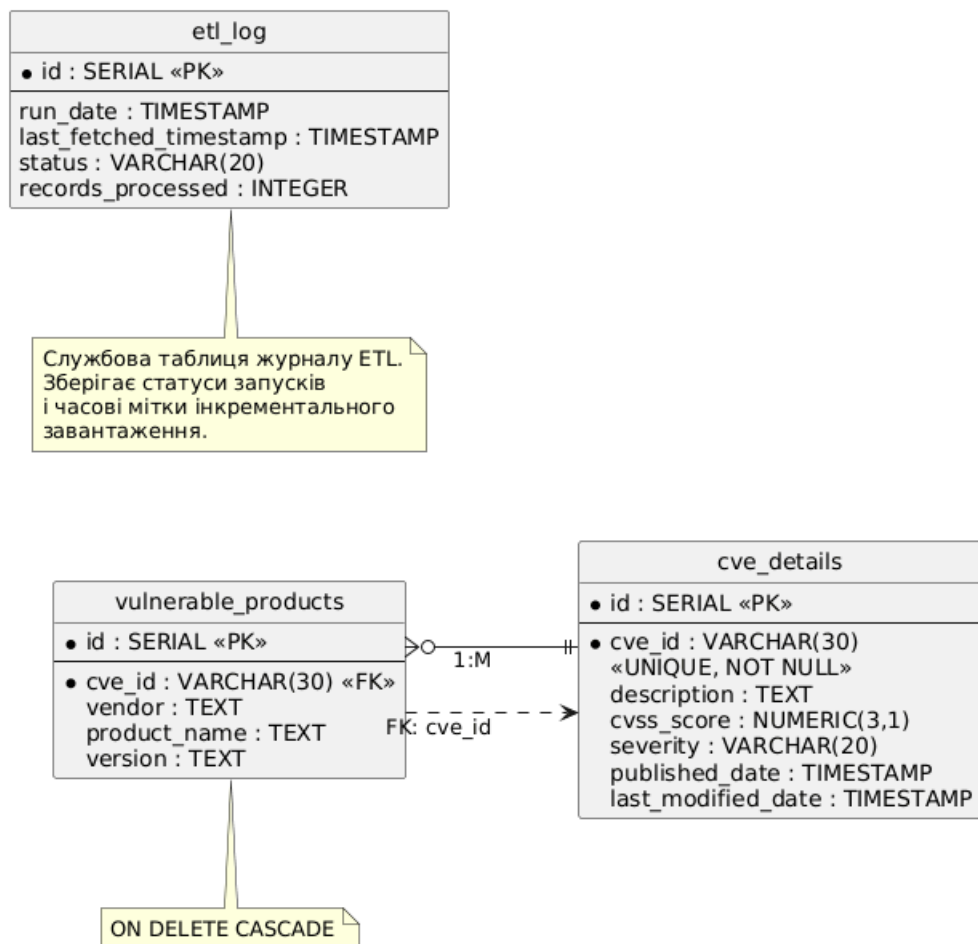


Рисунок 2.3 – Логічна схема моделі даних

Як показано на логічній схемі моделі даних (рис. 2.3), центральною сутністю сховища є `cve_details`. Вона призначена для зберігання основних відомостей про кібервразливості. Таблиця містить сурогатний первинний ключ `id`, унікальний бізнес-ідентифікатор `cve_id`, текстове поле `description`, числовий показник критичності `cvss_score`, якісний рівень небезпеки `severity`, а також часові поля `published_date` і `last_modified_date`.

Використання поля `id` як технічного первинного ключа дозволяє забезпечити внутрішню ідентифікацію записів у базі даних. Водночас поле `cve_id` зберігає предметний ідентифікатор вразливості та має обмеження `UNIQUE` і `NOT NULL`. Саме це поле використовується для визначення того, чи потрібно вставити новий CVE-запис, чи оновити вже наявний. Такий підхід зручний для інкрементального завантаження, оскільки одна й та сама

вразливість може бути отримана повторно після оновлення інформації в NVD.

Поле `description` використовується для зберігання очищеного англomовного опису вразливості. Поле `cvss_score` має тип `NUMERIC(3,1)`, що є достатнім для зберігання оцінок CVSS у діапазоні від 0.0 до 10.0. Поле `severity` дозволяє зберігати якісний рівень небезпеки, наприклад `LOW`, `MEDIUM`, `HIGH` або `CRITICAL`. Часові поля `published_date` і `last_modified_date` необхідні для аналізу динаміки появи вразливостей і визначення актуальності записів.

Сутність `vulnerable_products` моделює інформацію про програмні продукти, операційні системи або інші компоненти, на які поширюється відповідна вразливість. Між таблицями `cve_details` та `vulnerable_products` встановлено зв'язок типу «один-до-багатьох» через поле `cve_id`. Це означає, що одному CVE-запису може відповідати декілька уразливих продуктів або версій програмного забезпечення.

У таблиці `vulnerable_products` зберігаються поля `id`, `cve_id`, `vendor`, `product_name` та `version`. Поле `cve_id` виступає зовнішнім ключем і посиляється на унікальний бізнес-ідентифікатор у таблиці `cve_details`. Для підтримання реляційної цілісності використовується правило `ON DELETE CASCADE`, завдяки якому при видаленні CVE-запису всі пов'язані записи про продукти також автоматично видаляються. Водночас під час звичайного оновлення інформації про вразливість очищення застарілих зв'язків із продуктами виконується логікою модуля завантаження, після чого вставляється актуальний набір продуктів.

Сутність `etl_log` є службовою таблицею журналювання запусків конвеєра. Вона не має зовнішніх ключів до предметних таблиць, оскільки її призначення полягає не у зберіганні даних про вразливості, а у фіксації технічної інформації про виконання ETL-процесу. У таблиці зберігаються дата запуску `run_date`, часова мітка останнього успішно обробленого запису `last_fetched_timestamp`, статус виконання `status` і кількість оброблених записів `records_processed`.

Наявність таблиці `etl_log` є важливою для інкрементального режиму роботи системи. Під час нового запуску модуль вилучення звертається до цієї таблиці, визначає дату останнього успішного завантаження та формує наступне часове вікно запиту до NVD API. Це дозволяє уникнути повторного завантаження всього масиву даних і зменшити навантаження на зовнішній сервіс.

Запропонована логічна схема моделі даних відповідає задачі зберігання нормалізованої інформації про кібервразливості. Вона дозволяє відокремити основні CVE-записи від списку уразливих продуктів і службового журналу виконання. Завдяки цьому зменшується дублювання даних, забезпечується реляційна цілісність і створюється основа для аналітичних SQL-запитів.

На основі спроектованої логічної схеми моделі даних розробляється фізична структура бази даних у PostgreSQL. Фізичне створення таблиць `cve_details`, `vulnerable_products` та `etl_log`, а також накладання обмежень цілісності й індексів швидкодії реалізовано за допомогою SQL-скриптів. Повні лістинги створення та використання об'єктів бази даних наведено у Додатку Б.

Крім моделі даних, для реалізації програмної системи необхідно визначити об'єктну структуру системи. Об'єктна модель повинна відображати розподіл відповідальності між програмними класами, відповідати архітектурним шарам і забезпечувати можливість незалежної модифікації окремих компонентів. Для цього було побудовано UML-діаграму класів, наведену на рисунку 2.4 на наступній сторінці.

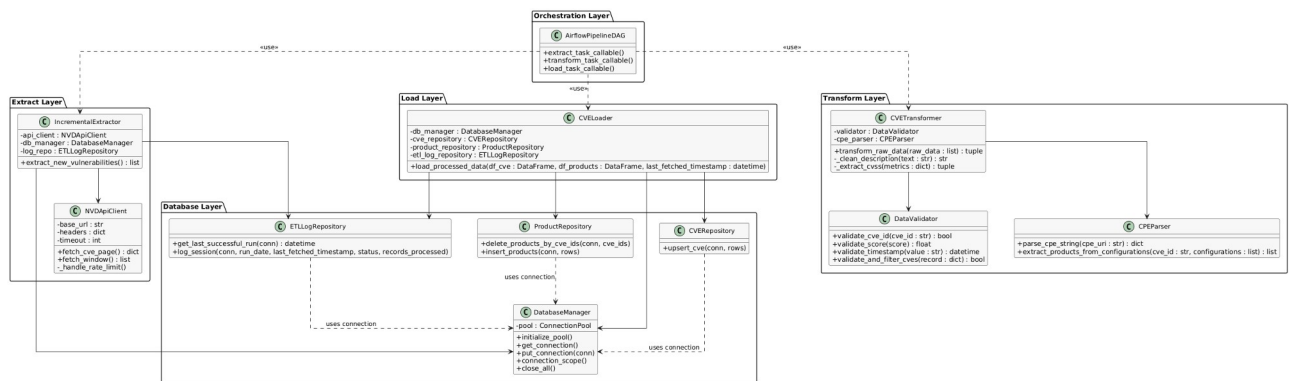


Рисунок 2.4 – UML-діаграма класів програмної системи

Діаграма класів (рис. 2.4) відображає статичну структуру програмної системи та групує класи за логічними пакетами: Database Layer, Extract Layer, Transform Layer, Load Layer і Orchestration Layer. Такий поділ відповідає архітектурі ETL-конвеєра та забезпечує чітке розмежування відповідальності між компонентами.

Пакет Database Layer містить клас DatabaseManager і класи-репозиторії CVERepository, ProductRepository та ETLLogRepository. Клас DatabaseManager відповідає за керування з'єднаннями з PostgreSQL, зокрема ініціалізацію пулу з'єднань, отримання активного з'єднання, повернення його до пулу та закриття ресурсів. Репозиторії інкапсулюють SQL-операції для роботи з відповідними таблицями бази даних і не містять логіки трансформації даних.

CVERepository відповідає за вставку та оновлення записів у таблиці cve_details. Саме цей клас реалізує доступ до операції UPSERT, що використовується для уникнення дублювання CVE-записів. ProductRepository відповідає за очищення застарілих зв'язків із продуктами та вставку актуального списку уразливих компонентів у таблицю vulnerable_products. ETLLogRepository забезпечує читання часової мітки останнього успішного запуску та запис результатів поточного виконання до таблиці etl_log.

Пакет Extract Layer містить класи NVDApiClient та IncrementalExtractor. Клас NVDApiClient інкапсулює HTTP-взаємодію з NVD API, формування параметрів запиту, передачу заголовків, підтримку пагінації та обробку ситуації Rate Limiting. Приватна логіка обробки обмеження частоти запитів

дає змогу приховати технічні деталі повторних звернень від інших компонентів системи.

Клас `IncrementalExtractor` координує процес вилучення даних. Він використовує `ETLLogRepository` для визначення часової мітки останнього успішного запуску, формує часові межі поточного запиту та викликає `NVDApiClient` для отримання нових або змінених записів. Таким чином, `IncrementalExtractor` не виконує безпосередню HTTP-логіку самостійно, а використовує API-клієнт як окремий сервіс.

Пакет `Transform Layer` містить класи `CVETransformer`, `DataValidator` та `CPEParser`. Центральним класом цього шару є `CVETransformer`, який перетворює сирі JSON-дані у два табличні набори: набір основних CVE-записів і набір уразливих продуктів. У процесі трансформації він очищує текстові описи, нормалізує дати, вилучає CVSS-метрики та формує структури, придатні для завантаження до PostgreSQL.

Клас `DataValidator` відповідає за перевірку коректності даних. Він контролює формат CVE-ідентифікатора, допустимість CVSS-оцінки та коректність часових значень. Клас `CPEParser` виконує розбір CPE-рядків і виділяє з них виробника, назву продукту та версію. Делегування цих задач окремим класам зменшує складність `CVETransformer` і робить код більш придатним до тестування.

Пакет `Load Layer` представлений класом `CVELoader`. Він координує фінальну фазу ETL-конвеєра: отримує очищені табличні дані, відкриває транзакцію через `DatabaseManager`, викликає методи репозиторіїв для вставки або оновлення CVE-записів, оновлює список уразливих продуктів і фіксує результат запуску в таблиці журналу. У разі помилки `CVELoader` повинен ініціювати відкат транзакції, щоб база даних не залишилася в частково оновленому стані.

Пакет `Orchestration Layer` містить клас `AirflowPipelineDAG`, який відображає логіку оркестрації конвеєра в Apache Airflow. Він не зберігає предметні дані та не виконує бізнес-логіку самостійно, а містить методи-

колбеки `extract_task_callable()`, `transform_task_callable()` та `load_task_callable()`. Ці методи послідовно викликають класи нижчих шарів і забезпечують інтеграцію програмних модулів у єдиний DAG.

Зв'язки між класами на рисунку 2.4 відповідають архітектурним залежностям системи. Оркестратор використовує класи `IncrementalExtractor`, `CVETransformer` і `CVELoader`, але не втручається в їхню внутрішню логіку. `IncrementalExtractor` залежить від `NVDApiClient` і `ETLLogRepository`, оскільки для інкрементального вилучення потрібні як HTTP-запити до NVD API, так і інформація про останній успішний запуск. `CVETransformer` використовує `DataValidator` і `CPEParser`, а `CVELoader` взаємодіє з репозиторіями та менеджером бази даних.

Описана об'єктна модель забезпечує низьку зв'язаність і високу внутрішню узгодженість компонентів. Кожен клас має чітко визначену відповідальність, що полегшує налагодження, модульне тестування й подальший супровід програмної системи. У разі зміни формату відповіді NVD API достатньо модифікувати класи шару вилучення або трансформації. У разі зміни структури бази даних основні зміни будуть зосереджені в репозиторіях і SQL-скриптах. Такий підхід робить архітектуру системи гнучкою та придатною для подальшого розвитку.

Таким чином, у другому розділі було визначено вимоги до програмної системи, побудовано модель варіантів використання, розроблено архітектурну схему системи, логічну схему моделі даних та UML-діаграму класів. Отримані проєктні рішення є основою для практичної реалізації програмних модулів, описаної у наступному розділі.

3 ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА РЕЗУЛЬТАТИ РОЗРОБКИ ПРОГРАМНОЇ СИСТЕМИ ДЛЯ МОНІТОРИНГУ КІБЕРВРАЗЛИВОСТЕЙ

3.1 Вибір та обґрунтування технологічних засобів розробки

Для успішного створення масштабованої та стійкої до відмов програмної системи автоматизованого збору інформації про вразливості (CVE) було проведено детальний аналіз сучасного стеку технологій. Ключовими критеріями вибору виступали: продуктивність при обробці великих масивів текстових даних, наявність розвинених бібліотек для роботи з мережевими протоколами, підтримка транзакційності на рівні баз даних та можливість гнучкої оркестрації розподілених завдань.

В якості базової мови програмування було обрано Python 3.11. Цей вибір зумовлений високою швидкістю розробки, лаконічністю синтаксису, а також наявністю екосистеми аналізу даних, що є критично важливим для ETL-інженерії. Версія 3.11 містить суттєві оптимізації швидкодії інтерпретатора (CPython) та покращену обробку винятків за допомогою механізму Exception Groups, що дозволяє ефективно керувати помилками у конкурентних мережесхемних потоках.

Для реалізації окремих архітектурних шарів системи було визначено такі компоненти:

- Шари вилучення даних (Extract) базуються на використанні бібліотеки requests для синхронної взаємодії з REST API NVD NIST. Вона забезпечує повноцінне керування HTTP-заголовками (передача API-ключа для підвищення лімітів частоти запитів), контроль таймаутів та автоматичну обробку серійних кодів відповідей.
- Шари перетворення та очищення (Transform) спираються на аналітичну бібліотеку pandas. Її структури даних DataFrame дозволяють у векторній формі здійснювати деструктуризацію вкладених JSON-масивів за допомогою функції `json_normalize()`, виконувати миттєву фільтрацію

невалідних значень та конвертацію типів даних (наприклад, перетворення рядків дат ISO-8601 у часові мітки `datetime64`) без використання повільних циклів мови Python.

- Шари завантаження та репозиторіїв (Load) реалізовано за допомогою низькорівневого драйвера `psycopg2` (або `psycopg` версії 3). Він забезпечує повноцінний доступ до можливостей СКБД PostgreSQL, дозволяє виконувати параметризовані запити, захищені від SQL-ін'єкцій, та підтримує механізм керування пулами з'єднань (ConnectionPool), що мінімізує накладні витрати на відкриття мережесетів при кожній ітерації конвеєра.

- Шар зберігання даних (Database) представлено об'єктно-реляційною СКБД PostgreSQL 15+. Вона обрана через її високу надійність, повну відповідність вимогам ACID та потужну підтримку операцій UPSERT за допомогою конструкції `ON CONFLICT DO UPDATE`, що є критичним для запобігання дублюванню записів CVE.

- Шар оркестрації процесів використовує платформу Apache Airflow 2.x, яка виступає ядром планування. Концепція розробки конвеєрів у вигляді спрямованих ациклічних графів (DAG) винятково мовою Python дозволяє безшовно інтегрувати модулі ETL, налаштовувати гнучкі розклади за допомогою виразів `cron`, ізолювати виконання окремих задач та моніторити їхній стан у реальному часі через інтегрований веб-інтерфейс.

3.2 Розробка схеми бази даних та структури зберігання

Фізична структура бази даних спроектована з урахуванням мінімізації надлишковості та забезпечення максимальної швидкості виконання аналітичних запитів.

1. Таблиця `cve_details` виступає центральним ядром схеми. Як технічний первинний ключ використано поле `id` типу `SERIAL`, що забезпечує внутрішню ідентифікацію записів у базі даних. Поле `cve_id` типу `VARCHAR(30)` є унікальним бізнес-ідентифікатором CVE-запису, на який

накладено обмеження UNIQUE та NOT NULL. Поле description визначено типом TEXT для збереження розгорнутих англomовних описів векторів атак та наслідків їхньої реалізації. Рейтинг критичності cvss_score має точний числовий тип NUMERIC(3,1), що є достатнім для зберігання оцінок CVSS у діапазоні від 0.0 до 10.0 та запобігає похибкам округлення, притаманним типам із плаваючою крапкою. Для прискорення пошуку вразливостей за датою їхньої останньої модифікації, на стовпець last_modified_date накладено B-Tree індекс, що оптимізує інкрементальне вибирання даних на наступних циклах ETL.

2. Таблиця vulnerable_products деструктуризує інформацію про програмне забезпечення, що піддається загрозам. Зв'язок із головною таблицею реалізовано за допомогою зовнішнього ключа FOREIGN KEY (cve_id) REFERENCES cve_details(cve_id) ON DELETE CASCADE. Каскадне видалення гарантує, що при видаленні запису CVE всі асоційовані записи про продукти будуть автоматично видалені, що унеможливорює появу "записів-сиріт" та підтримує реляційну цілісність сховища. Під час звичайного оновлення інформації про вразливість очищення застарілих зв'язків виконується логікою модуля завантаження. Унікальність записів контролюється складеним обмеженням UNIQUE (cve_id, vendor, product_name, version), що блокує дублювання ідентичних платформ для однієї уразливості.

3. Технічна таблиця etl_log служить для мета-координації та інкрементального збору. Вона фіксує мітку часу запуску run_date та значення last_fetched_timestamp, яке зберігає максимальну дату модифікації успішно імпортованих записів CVE. На основі цього поля модуль вилучення динамічно обчислює часове вікно для наступного запиту, що знижує навантаження на мережеві канали та сервер API NVD.

3.3 Програмна реалізація модулів конвеєра даних (ETL)

Програмна реалізація системи виконана відповідно до багаторівневої архітектури, розробленої у розділі 2. У практичній частині реалізація побудована на п'яти основних архітектурних шарах: шарі доступу до бази даних, шарі вилучення даних, шарі трансформації, шарі завантаження та шарі оркестрації. Додатково використано конфігураційний модуль `settings.py`, який централізовано зчитує параметри запуску з оточення. Така структура дозволяє зберегти слабку зв'язаність між компонентами та забезпечує відповідність реалізації UML-діаграмі класів.

Конфігураційний модуль `'settings.py'` відповідає за централізоване зчитування параметрів запуску з оточення. До таких параметрів належать адреса NVD API, ключ доступу, параметри підключення до PostgreSQL, кількість повторних спроб, таймауту та робочий каталог для проміжних файлів. Завдяки цьому в коді не зберігаються паролі, рядки підключення або інші чутливі параметри.

Шар доступу до бази даних реалізований класом `'DatabaseManager'` та репозиторіями `'CVERepository'`, `'ProductRepository'`, `'ETLLogRepository'`. `'DatabaseManager'` керує пулом з'єднань PostgreSQL і надає контрольований доступ до активних сесій бази даних. Репозиторії інкапсулюють SQL-операції та відокремлюють логіку збереження від бізнес-логіки обробки даних. `'CVERepository'` виконує пакетне UPSERT-завантаження записів CVE, `'ProductRepository'` оновлює зв'язки між CVE та уразливими продуктами, а `'ETLLogRepository'` зчитує часову мітку останнього успішного запуску і фіксує результат поточного сеансу.

Модуль вилучення даних представлений класами `'NVDApiClient'` та `'IncrementalExtractor'`. `'NVDApiClient'` відповідає за HTTP-взаємодію з NVD API, підтримує пагінацію за параметром `'startIndex'`, обмеження `'resultsPerPage'`, обробку статусу 429 Too Many Requests та повторні спроби у разі тимчасових мережевих збоїв. `'IncrementalExtractor'` визначає часові межі інкрементального завантаження на основі таблиці `'etl_log'`, розбиває великий період на допустимі часові вікна та повертає сирий набір JSON-об'єктів разом

із кінцевою часовою міткою.

Шар трансформації реалізований класами `'CVETransformer'`, `'DataValidator'` та `'CPEParser'`. `'CVETransformer'` перетворює вкладену JSON-структуру NVD у два табличні набори даних: основну таблицю CVE та таблицю уразливих продуктів. Під час трансформації виконується вибір англomовного опису, очищення тексту від службових символів, отримання балу CVSS і рівня критичності, нормалізація дат та формування структур `'pandas.DataFrame'`. `'DataValidator'` перевіряє коректність CVE-ідентифікаторів, діапазонів CVSS та часових полів. `'CPEParser'` виконує розбір CPE-рядків і виділяє з них назву виробника, назву продукту та версію.

Шар завантаження представлений класом `'CVELoader'`. Він координує фінальний етап ETL-конвеєра і виконує запис даних у PostgreSQL у межах єдиної транзакції. Спочатку здійснюється UPSERT-завантаження основних відомостей про CVE, після чого для відповідних CVE очищуються застарілі зв'язки з продуктами і вставляються актуальні записи в таблицю `'vulnerable_products'`. Якщо будь-який етап завершується помилкою, транзакція відкочується, а в таблиці журналу фіксується статус `'FAILED'`.

Повні лістинги програмної реалізації наведено у Додатку Б. Вони включають модулі Python для шарів Extract, Transform, Load, клас керування DAG в Apache Airflow, репозиторії доступу до PostgreSQL, а також допоміжні SQL-запити для верифікації результатів завантаження та формування аналітичних звітів зі сховища даних.

3.4 Налаштування оркестрації процесів в Apache Airflow

Автоматизація запуску ETL-конвеєра реалізована за допомогою Apache Airflow. Для цього було створено DAG з ідентифікатором `'cve_vulnerability_pipeline'`, який описує послідовність виконання трьох основних задач: `'extract_task'`, `'transform_task'` та `'load_task'`.

Задача `'extract_task'` запускає клас `'IncrementalExtractor'`, визначає

часову мітку останнього успішного запуску, формує параметри запиту до NVD API та зберігає сирий JSON у проміжному файлі. Задача `transform_task` зчитує цей файл, викликає `CVETransformer`, виконує очищення та нормалізацію даних і створює два CSV-файли: `cve_clean.csv` та `products_clean.csv`. Задача `load_task` зчитує очищені дані, відкриває транзакцію PostgreSQL і викликає `CVELoader` для завантаження результатів у таблиці `cve_details`, `vulnerable_products` та `etl_log`.

Для підвищення відмовостійкості DAG налаштовано параметри `retries=3` та `retry_delay=timedelta(minutes=5)`. Це дозволяє автоматично повторювати виконання задачі у разі тимчасового мережевого збою або недоступності зовнішнього API. Послідовність задач задається оператором залежності `extract_task >> transform_task >> load_task`, що гарантує виконання наступного етапу лише після успішного завершення попереднього.

Стан налаштованого DAG у вебінтерфейсі Apache Airflow наведено на рисунку 3.1.

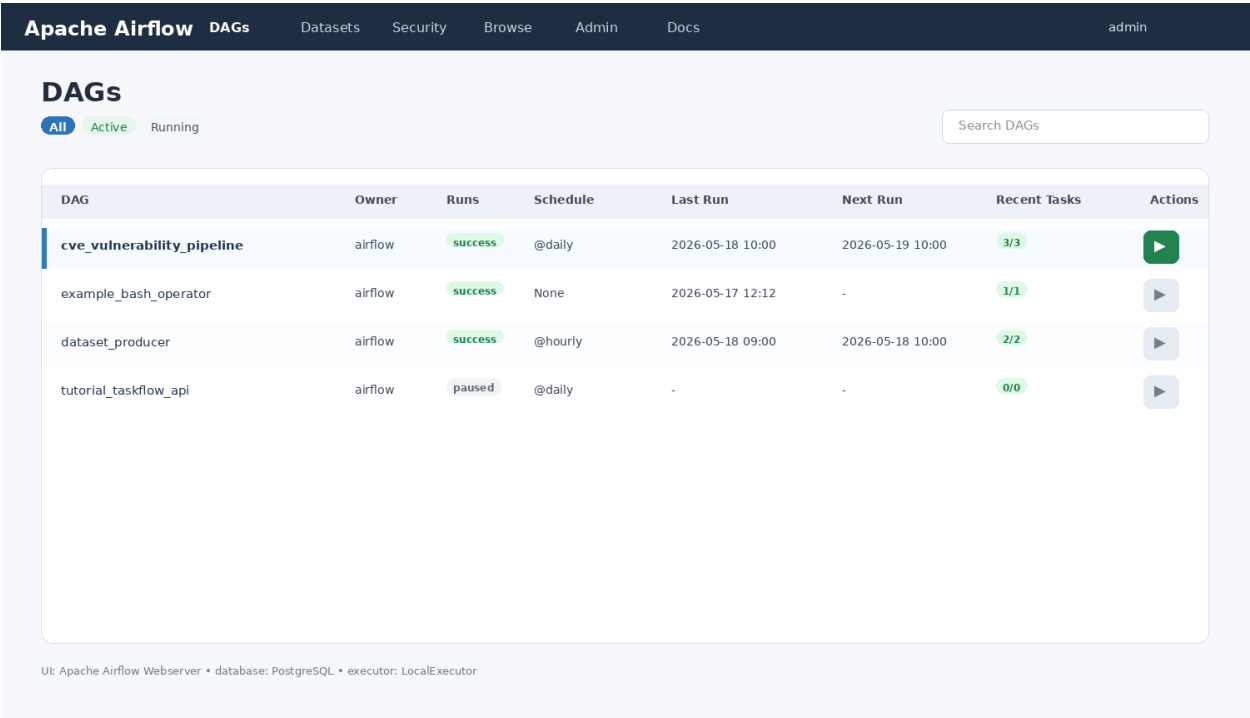


Рисунок 3.1 – Список DAG у вебінтерфейсі Apache Airflow

Графічне представлення успішного виконання задач конвеєра наведено на рисунку 3.2.

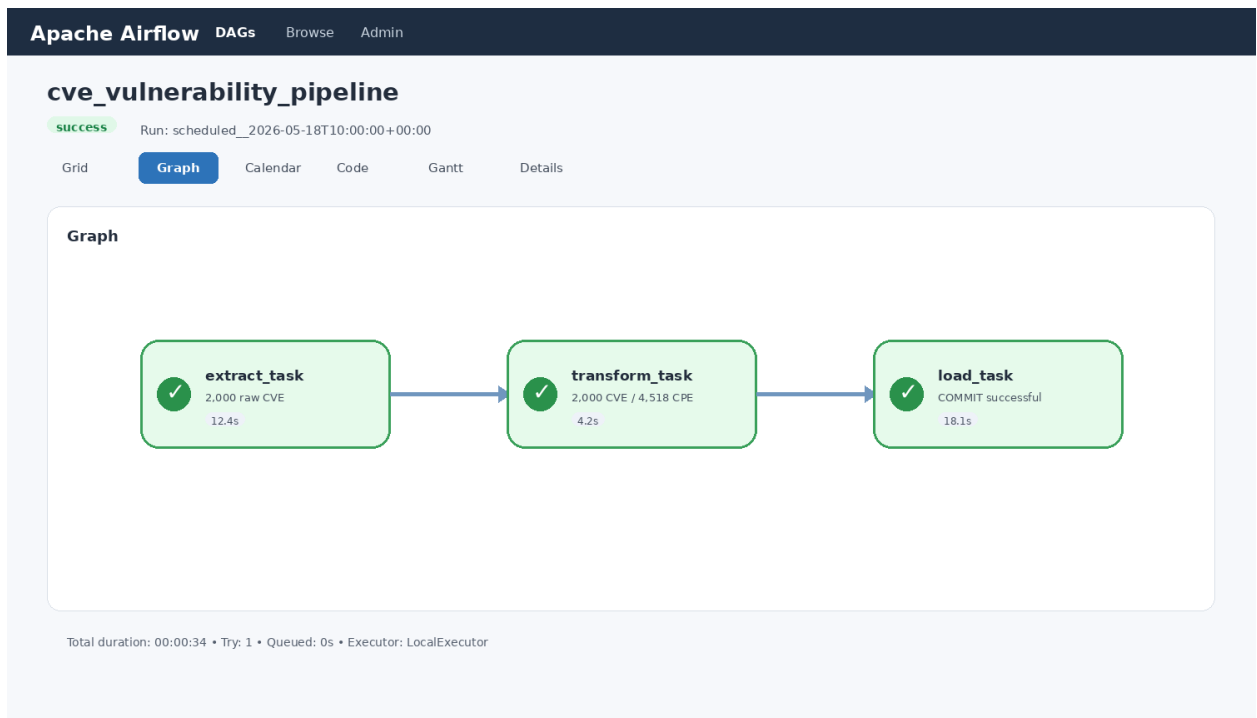


Рисунок 3.2 – Граф виконання задач DAG 'cve_vulnerability_pipeline'

3.5 Тестування та перевірка працездатності системи

Для перевірки працездатності програмної системи було проведено функціональне та інтеграційне тестування основних сценаріїв роботи системи. Метою тестування було підтвердження коректності інкрементального вилучення даних, стабільності роботи з NVD API, правильності трансформації JSON-структур, коректності розбору CPE-рядків і цілісності запису даних у PostgreSQL.

Першим тестовим сценарієм була перевірка інкрементального режиму. Для цього в таблицю 'etl_log' було внесено запис про попередній успішний запуск. Після запуску конвеєра модуль 'IncrementalExtractor' зчитав цю часову мітку і сформував запит до NVD API лише для нових або змінених записів. Така поведінка підтвердила виконання вимоги щодо зменшення обсягу

повторного завантаження даних.

Другим сценарієм була перевірка обробки обмеження Rate Limiting. У разі отримання відповіді HTTP 429 модуль `NVDApiClient` не завершує роботу аварійно, а переходить у режим очікування, після чого повторює запит. Це забезпечує стійкість системи до тимчасових обмежень із боку зовнішнього джерела.

Третім сценарієм була перевірка трансформації даних. На вхід модуля `CVETransformer` подавались JSON-об'єкти з вкладеними описами, метриками CVSS та конфігураціями CPE. У результаті було сформовано два нормалізовані набори даних: один для таблиці `cve\_details`, другий для таблиці `vulnerable\_products`. Окремо перевірялась коректність виділення полів `vendor`, `product\_name` і `version` із CPE-рядків.

Четвертим сценарієм була перевірка транзакційного завантаження та UPSERT-логіки. Повторне завантаження вже наявних CVE не створювало дублікатів у таблиці `cve\_details`, а оновлювало існуючі записи. Для дочірньої таблиці `vulnerable\_products` перед вставкою нових зв'язків виконувалось очищення попередніх записів за відповідними CVE, що дозволило уникнути накопичення застарілих або дубльованих прив'язок.

3.6 Результати експериментальних досліджень та верифікації працездатності системи

Для підтвердження працездатності розробленої програмної системи було виконано демонстраційний тестовий прогін ETL-конвеєра. Під час прогону перевірялося проходження повного циклу обробки даних: запуск DAG в Apache Airflow, вилучення даних із NVD API, трансформація JSON-структур у табличний вигляд, транзакційне завантаження в PostgreSQL та виконання контрольних SQL-запитів.

Після запуску DAG усі три задачі `extract\_task`, `transform\_task` та `load\_task` завершилися зі статусом `success`. На рисунку 3.2 видно, що задачі

виконувалися послідовно, без пропусків або аварійного завершення. Це підтверджує правильність налаштованих залежностей між етапами конвеєра.

Додатково було проаналізовано журнал виконання задачі `load\_task`, оскільки саме на цьому етапі відбувається запис даних у PostgreSQL і фіксація результату в журналі `etl\_log`. Фрагмент журналу виконання наведено на рисунку 3.3.

The screenshot displays the Apache Airflow interface for a DAG named 'load_task'. The task instance is marked as 'success' and was scheduled on 2026-05-18T10:00:00+00:00. The log shows the following sequence of events:

- 2026-05-18 10:00:23,114 | INFO | airflow.task | Executing PythonOperator load_task
- 2026-05-18 10:00:23,391 | INFO | load.loader | Beginning transactional load of processed ETL data
- 2026-05-18 10:00:24,728 | INFO | database.repositories | UPSERT completed for 2000 CVE rows
- 2026-05-18 10:00:27,019 | INFO | database.repositories | Deleted outdated products for 2000 CVE identifiers
- 2026-05-18 10:00:32,966 | INFO | database.repositories | Inserted 4518 product links
- 2026-05-18 10:00:35,451 | INFO | database.repositories | INSERT INTO etl_log status=SUCCESS records_processed=2000
- 2026-05-18 10:00:41,002 | INFO | load.loader | Load committed: CVE=2000 products=4518
- 2026-05-18 10:00:41,318 | INFO | airflow.task | Marking task as SUCCESS
- 2026-05-18 10:00:41,597 | INFO | airflow.dagrun | DagRun finished successfully

Рисунок 3.3 – Журнал виконання задачі `load\_task` в Apache Airflow

З журналу видно, що система виконала UPSERT-завантаження 2000 записів CVE, очистила застарілі зв'язки продуктів, додала 4518 записів до таблиці `vulnerable\_products` і завершила транзакцію командою commit. Завершення задачі зі статусом `SUCCESS` підтверджує, що запис у базу даних був виконаний без помилок.

Наступним етапом перевірки було виконання контрольного SQL-запиту до PostgreSQL. Запит об'єднує таблиці `cve\_details` та `vulnerable\_products` за полем `cve\_id` і повертає структурований перелік вразливостей разом із відповідними програмними продуктами. Результат виконання запиту наведено

на рисунку 3.4.

pgAdmin 4

Object Explorer

Query Tool

Servers

PostgreSQL 15

Databases

vuln_monitoring

Schemas

public

Tables

cve_details

vulnerable_products

etl_log

Query Editor

SELECT c.cve_id, c.cvss_score, c.published_date::date,
p.product_name AS product, p.vendor, p.version
FROM cve_details AS c
JOIN vulnerable_products AS p ON p.cve_id = c.cve_id
WHERE c.cvss_score >= 5.0
ORDER BY c.cvss_score DESC, c.published_date DESC
LIMIT 8;

Execute

Data Output

cve_id	cvss_score	published_date	product	vendor	version
CVE-2026-1024	9.8	2026-05-12	exchange_server	microsoft	2019_cu12
CVE-2026-1024	9.8	2026-05-12	exchange_server	microsoft	2022_cu02
CVE-2026-2107	9.1	2026-05-18	openssl	openssl	3.0.12
CVE-2026-1189	8.8	2026-05-14	linux_kernel	linux	5.15.0
CVE-2026-1189	8.8	2026-05-14	linux_kernel	linux	6.1.12
CVE-2026-2142	8.1	2026-05-18	nginx	nginx	1.25.4
CVE-2026-2001	7.5	2026-05-15	tomcat	apache	10.1.5
CVE-2026-2045	5.3	2026-05-17	chrome	google	121.0.0

Query returned successfully in 184 ms. 8 rows returned.

Рисунок 3.4 – Результат контрольного SQL-запиту до сховища PostgreSQL

Для перевірки цілісності сховища було виконано додаткові SQL-запити: підрахунок кількості записів у таблицях та пошук дублікатів у таблиці `vulnerable\_products`. Результати наведено на рисунку 3.5.

PostgreSQL verification queries

Контроль кількості записів після виконання ETL

SELECT 'cve_details' AS table_name, COUNT(*) AS row_count FROM cve_details
UNION ALL
SELECT 'vulnerable_products' AS table_name, COUNT(*) AS row_count FROM vulnerable_products
UNION ALL
SELECT 'etl_log' AS table_name, COUNT(*) AS row_count FROM etl_log;

table_name	row_count
cve_details	2000
vulnerable_products	4518
etl_log	17

Перевірка відсутності дублікатів у таблиці vulnerable_products

SELECT cve_id, vendor, product_name, version, COUNT(*) AS duplicates
FROM vulnerable_products
GROUP BY cve_id, vendor, product_name, version
HAVING COUNT(*) > 1;

0 rows returned — дублікати відсутні

Результати підтверджують коректність UPSERT-логіки та очищення дочірніх зв'язків перед завантаженням.

Рисунок 3.5 – Перевірка кількості записів і відсутності дублікатів у сховищі PostgreSQL

Отримані результати підтверджують, що розроблена програмна система коректно виконує повний цикл ETL-обробки. Дані успішно вилучаються із зовнішнього джерела, очищуються, нормалізуються та зберігаються у реляційній базі даних. Використання Apache Airflow забезпечує контрольованість і повторюваність запусків, а PostgreSQL надає можливість виконання аналітичних SQL-запитів до накопиченого сховища.

4 ОХОРОНА ПРАЦІ

4.1 Аналіз умов праці аналітика безпеки при роботі з екранними пристроями

Розробка та подальша експлуатація програмної системи автоматизованого збору даних про вразливості пов'язані з діяльністю аналітика з кібербезпеки, чие робоче місце оснащене персональним комп'ютером, рідкокристалічними моніторами та периферійними пристроями введення-виведення інформації. Праця аналітика класифікується як розумова діяльність із високим рівнем нервово-емоційного та зорового напруження. Відповідно до Закону України "Про охорону праці", роботодавець зобов'язаний створити на кожному робочому місці умови праці, що відповідають вимогам нормативно-правових актів.

Основними небезпечними та шкідливими виробничими факторами (фізичними та психофізіологічними) на робочому місці аналітика є:

- Незадовільні параметри мікроклімату, оскільки тривала робота серверного та комп'ютерного обладнання призводить до тепловиділення, що викликає підвищення температури повітря, зниження його відносної вологості та накопичення позитивно заряджених аероіонів.
- Нераціональне освітлення, де наявність прямих або відбитих відблисків на екрані монітора, недостатня освітленість робочої зони або надмірна яскравість екрана викликають швидку втому зорового аналізатора, зниження концентрації уваги та розвиток комп'ютерного зорового синдрому (CVS).
- Підвищений рівень шуму, де основними джерелами шуму в приміщенні є вентилятори систем охолодження ПК, кондиціонери та блоки безперебійного живлення. Тривалий вплив шуму понад норму призводить до подразнення нервової системи та зниження продуктивності праці.
- Електромагнітні поля та статична електрика, які утворюються

навколо моніторів та системних блоків і при тривалій дії можуть негативно впливати на серцево-судинну систему.

- Психофізіологічні фактори, такі як гіподинамія (тривале перебування в статичній позі сидячи), перенапруження зорового апарату під час аналізу лістингів коду вразливостей, висока щільність сприйняття інформації.

4.2 Ергономічні вимоги до організації робочого місця та режиму праці

Організація робочого місця аналітика безпеки повинна суворо відповідати "Вимогам щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями" (НПАОП 0.00-7.15-18) та ДСанПіН 3.3.2.007-98.

Основні геометричні та ергономічні параметри організації робочої зони:

- Робочий стіл та стілець: Висота робочої поверхні столу повинна встановлюватися в межах 680–800 мм. Робочий стілець має бути підйомно-поворотним, регульованим за висотою та кутами нахилу сидіння і спинки (оптимальний кут нахилу спинки — 90–110° відносно сидіння). Спинка стільця повинна мати анатомічний вигин для підтримки поперекового відділу хребта. За необхідності робоче місце комплектується підставкою для ніг шириною не менше 400 мм та довжиною 300 мм із регульованим кутом нахилу.

- Розміщення відеотермінала (монітора): Екран монітора повинен знаходитися на відстані 600–700 мм від очей оператора. Напрямок погляду має бути перпендикулярним до центру екрана, а кут зору у вертикальній площині відносно горизонтальної лінії очей повинен становити 15–20°. Оптимально використовувати монітори з матовим покриттям для уникнення дзеркальних відблисків від вікон чи світильників загального освітлення.

- Параметри мікроклімату та шуму: Температура повітря у приміщенні з обчислювальною технікою повинна підтримуватися в межах 22

–25 °С у теплий період року та 20–22 °С у холодний період при відносній вологості 40–60%. Швидкість руху повітря не повинна перевищувати 0.1–0.2 м/с. Еквівалентний рівень шуму на робочому місці аналітика відповідно до ДСН 3.3.6.037-99 не повинен перевищувати 50 дБА.

– Режим праці та відпочинку: При 8-годинному робочому дні для запобігання розвитку втоми та підтримки працездатності встановлюються регламентовані перерви тривалістю 10–15 хвилин через кожні 45–50 хвилин безперервної роботи за монітором. Під час перерв аналітику рекомендується виконувати комплекси вправ для очей (зміна фокусної відстані, кругові рухи) та фізичні вправи для зняття статичного напруження з м'язів шиї, спини та плечового поясу.

4.3 Електробезпека та пожежна безпека у приміщенні з обчислювальною технікою

Приміщення, в якому експлуатується комп'ютерна техніка та сервери збору даних, відноситься до категорій приміщень без підвищеної небезпеки ураження електричним струмом (за умови сухого повітря, нормальної температури та відсутності струмопровідної підлоги). Проте, велика кількість одночасно підключеного обладнання створює потенційні ризики електричного та пожежного характеру.

Заходи забезпечення електробезпеки:

– Усі металеві корпуси персональних комп'ютерів, серверних стійок, КВП та кондиціонерів повинні бути надійно заземлені шляхом підключення до загального контуру захисного заземлення будівлі з опором розтікання струму не більше 4 Ом.

– Підключення обладнання до електромережі напругою 220 В із частотою 50 Гц повинно здійснюватися виключно через трипровідні кабелі із заземлювальною жилою та євророзетки із заземлювальними контактами.

– В обов'язковому порядку на розподільних щитах живлення

встановлюються пристрої захисного вимкнення (ПЗВ), які миттєво знеструмлюють лінію при виникненні струму витoku на корпус понад 30 мА.

- Профілактичне вимірювання опору ізоляції дротів та перевірка цілісності заземлювальних провідників повинні проводитися спеціалізованим персоналом не рідше одного разу на 12 місяців.

Заходи забезпечення пожежної безпеки:

- Основними причинами пожеж у приміщеннях з обчислювальною технікою є короткі замикання в імпульсних блоках живлення, перевантаження кабельних ліній та іскріння в місцях ненадійних контактів. Для захисту від перевантажень лінії живлення захищаються автоматичними вимикачами, розрахованими на номінальний струм споживання.

- Забороняється використання саморобних подовжувачів, експлуатація обладнання з пошкодженою ізоляцією та захаращення вентиляційних отворів на корпусах комп'ютерів.

- Приміщення повинно бути обладнане автоматичною системою пожежної сигналізації з димовими оптико-електронними сповіщувачами (наприклад, СПД-3).

- На випадок виникнення загоряння приміщення забезпечується первинними засобами пожежогасіння — вуглекислотними вогнегасниками класу ВВ-2 або ВВ-5. Використання пінних або водних вогнегасників для гасіння електроустановок під напругою суворо заборонено. Шляхи евакуації та евакуаційні виходи повинні утримуватися вільними та маркуватися світловими покажчиками "ВИХІД".

ВИСНОВКИ

У дипломній роботі успішно розв'язано актуальну науково-практичну задачу, яка полягає в автоматизації процесів збору, структурування та тривалого зберігання інформації про кібервразливості (CVE). Це дозволило суттєво підвищити загальну ефективність моніторингу загроз інформаційній безпеці сучасних ІТ-інфраструктур. На основі проведеного детального аналізу існуючих методів та технологій агрегації даних було обґрунтовано доцільність побудови автоматизованого конвеєра даних (ETL) на базі відкритих API-інтерфейсів, що повністю замінює ручну працю та мінімізує вплив людського фактора.

Мету кваліфікаційної роботи, яка полягала в розробці програмної системи автоматизованого збору та обробки даних про кібервразливості з використанням конвеєра даних, досягнуто повністю.

У ході виконання роботи було вирішено всі поставлені завдання, а саме:

- проаналізовано предметну область моніторингу кібервразливостей та досліджено особливості структури даних National Vulnerability Database;
- проаналізовано існуючі програмні рішення для збору та обробки інформації про вразливості, визначено їх переваги й недоліки;
- визначено функціональні та нефункціональні вимоги до програмної системи;
- розроблено архітектуру програмної системи та структуру конвеєра даних;
- розроблено реляційну модель бази даних для зберігання інформації про вразливості та пов'язані програмні продукти;
- реалізовано програмні модулі вилучення, перетворення та завантаження даних із використанням засобів Python та бібліотеки Pandas;
- реалізовано механізм інкрементального оновлення даних і автоматизовану оркестрацію процесів за допомогою Apache Airflow;
- проведено тестування програмної системи та оцінено результати її

функціонування;

- розроблено комплект супровідної документації, що включає опис програми, інструкцію користувача та програму і методику випробувань.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. National Vulnerability Database. NVD Vulnerability APIs. URL: <https://nvd.nist.gov/developers/vulnerabilities> (дата звернення: 16.03.2026).
2. CVE Program. Common Vulnerabilities and Exposures. URL: <https://www.cve.org> (дата звернення: 16.03.2026).
3. FIRST. Common Vulnerability Scoring System SIG. URL: <https://www.first.org/cvss/> (дата звернення: 16.03.2026).
4. National Vulnerability Database. Official Common Platform Enumeration Dictionary. URL: <https://nvd.nist.gov/products/cpe> (дата звернення: 16.03.2026).
5. Tenable Nessus. Vulnerability Assessment Solution. URL: <https://www.tenable.com/products/nessus> (дата звернення: 20.03.2026).
6. Qualys VMDR. Vulnerability Management, Detection and Response. URL: <https://www.qualys.com/apps/vulnerability-management-detection-response/> (дата звернення: 20.03.2026).
7. cve-search. GitHub repository. URL: <https://github.com/cve-search/cve-search> (дата звернення: 20.03.2026).
8. PostgreSQL Documentation. INSERT. URL: <https://www.postgresql.org/docs/current/sql-insert.html> (дата звернення: 04.05.2026).
9. PostgreSQL Documentation. Indexes. URL: <https://www.postgresql.org/docs/current/indexes.html> (дата звернення: 04.05.2026).
10. Python Software Foundation. Python 3 Documentation. URL: <https://docs.python.org/3/> (дата звернення: 04.05.2026).
11. Requests: HTTP for Humans. Documentation. URL: <https://requests.readthedocs.io/> (дата звернення: 04.05.2026).
12. pandas documentation. API reference. URL: <https://pandas.pydata.org/docs/> (дата звернення: 04.05.2026).

13. McKinney W. Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter. 3rd ed. O'Reilly Media, 2022. 576 p.
14. psycopg2 documentation. URL: <https://www.psycopg.org/docs/> (дата звернення: 11.05.2026).
15. Apache Airflow Documentation. DAGs. URL: <https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html> (дата звернення: 11.05.2026).
16. PlantText UML Editor. URL: <https://www.planttext.com> (дата звернення: 18.05.2026).
17. Про охорону праці : Закон України від 14.10.1992 № 2694-XII. Відомості Верховної Ради України. 1992. № 49. Ст. 668.
18. Про затвердження Вимог щодо безпеки та захисту здоров'я працівників під час роботи з екранними пристроями : наказ Міністерства соціальної політики України від 14.02.2018 № 207. Офіційний вісник України. 2018. № 37. Ст. 1316.
19. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин : ДСанПіН 3.3.2.007-98 : затв. постановою Головного державного санітарного лікаря України від 10.12.1998 № 7.
20. Санітарні норми виробничого шуму, ультразвуку та інфразвуку : ДСН 3.3.6.037-99 : затв. постановою Головного державного санітарного лікаря України від 01.12.1999 № 37.

ДОДАТОК А ТЕХНІЧНЕ ЗАВДАННЯ НА РОЗРОБКУ ПРОГРАМНОЇ СИСТЕМИ

Вступ

Додаток містить Технічне завдання на розробку програмної системи автоматизованого збору та обробки даних про вразливості програмного забезпечення з використанням конвеєра даних. Цей документ є визначальним інженерним базисом для проектування архітектури, реалізації модулів трансформації та інтеграції усіх компонентів системи. Сферою застосування розроблюваної програмної системи є інженерія програмного забезпечення та кібербезпека, зокрема процеси автоматизованого моніторингу та оперативної аналітики актуальних цифрових загроз.

1 Найменування та область застосування

Цей документ поширюється на програмне забезпечення системи автоматизованого збору та обробки даних про вразливості програмного забезпечення з використанням конвеєра даних. Скорочене найменування розробки — ПС АЗОДВ.

Головною областю застосування створеного програмного продукту є сфера інженерії програмного забезпечення та кібербезпеки. Система призначена для використання аналітиками з комп'ютерної безпеки, системними адміністраторами та командами розробників з метою автоматизації моніторингу виходу нових світових уразливостей та оперативного аналізу потенційних загроз для інфраструктури організації.

2 Підстави для розробки

Підставою для розробки цього програмного забезпечення є завдання на виконання кваліфікаційної роботи бакалавра, затверджене кафедрою програмного забезпечення автоматизованих систем Національного

університету кораблебудування імені адмірала Макарова. Розробка виконується відповідно до затвердженого плану-графіку та вимог сучасних стандартів до проєктування автоматизованих систем обробки слабкоструктурованої інформації.

3 Призначення розробки

Програмна система призначена для повного усунення ручної праці під час моніторингу оновлень Національної бази даних вразливостей США (NVD). Вона забезпечує регулярне, автоматичне та безпечне вивантаження нових записів про дефекти коду, їх трансформацію з ієрархічного вигляду JSON у пласкі табличні структури, очищення від зайвої чи некоректної інформації, а також збереження у централізоване реляційне сховище. Кінцевою метою використання системи є надання користувачам можливості швидкого пошуку критичних багів безпеки та побудови аналітичних звітів за допомогою стандартних SQL-запитів.

4 Технічні вимоги

4.1 Вимоги до функціональних характеристик

Програмна система АЗОДВ повинна забезпечувати стабільне виконання трьох основних етапів обробки даних, що відповідають класичній концепції інженерії даних ETL: вилучення, перетворення та завантаження. Для цього в архітектурі системи мають бути чітко реалізовані окремі функціональні модулі та конвеєри керування.

В межах підсистеми вилучення даних програма повинна в автоматичному режимі надсилати запити до відкритого програмного інтерфейсу бази даних NVD. Система має обов'язково підтримувати механізм інкрементального завантаження. При кожному новому запуску модуля необхідно аналізувати часову мітку останнього успішного запису в локальній базі даних і формувати параметри HTTP-запиту до API таким чином, щоб отримувати лише ті записи CVE, які були створені або змінені з моменту цього

останнього оновлення. Також цей модуль повинен містити логіку обробки обмежень сервера на кількість запитів за хвилину, роблячи контрольовані паузи між викликами, щоб уникнути блокування IP-адреси системи.

Підсистема перетворення даних повинна базуватися на можливостях мови Python та бібліотеки Pandas. Головною функцією цього модуля є розбір складних вкладених JSON-документів, які надходять від сервера. Програма повинна очищати отримані масиви від порожніх рядків та приводити типи даних до стандартних форматів. Зокрема, текстові описи вразливостей мають очищатися від службових символів, дати повинні перетворюватися на уніфікований формат реляційної бази даних, а числові бали небезпеки за системою оцінки CVSS мають коректно округлятися. Окремою функцією цього модуля є розбір списків вразливого обладнання та ПЗ за стандартом CPE з подальшим виділенням назв вендорів та версій у окремі табличні структури для збереження логічних зв'язків.

Підсистема завантаження повинна забезпечувати надійний запис обробленої інформації до реляційної бази даних PostgreSQL. Функціонал модуля має контролювати унікальність записів за номерами CVE, щоб уникнути дублювання інформації при повторному аналізі оновлених вразливостей. Запис даних повинен виконуватися з урахуванням цілісності зв'язків між головними таблицями опису загроз та дочірніми таблицями списків уразливих продуктів.

Підсистема оркестрації та управління конвеєром повинна функціонувати під керуванням платформи Apache Airflow версії 2.x або новішої сумісної версії. Вона зобов'язана виконувати регулярний запуск усього ланцюжка завдань за визначеним у конфігурації розкладом, наприклад, один раз на добу в автоматичному режимі. Головною функцією оркестратора є відслідковування успішності кожного кроку, ведення детальних логів роботи системи та запуск механізму повторних спроб виконання завдань у разі виникнення тимчасових помилок мережі або недоступності зовнішнього сервера.

4.2 Вимоги до надійності

Програмна система має бути спроектована з урахуванням високого рівня відмовостійкості, оскільки вона функціонує в умовах мережевої невизначеності. У випадку повної відсутності інтернет-з'єднання під час чергового сеансу зв'язку з API, система не повинна аварійно завершувати роботу всієї платформи оркестрації. Модуль вилучення має зробити кілька заздалегідь налаштованих спроб підключення з інтервалом у кілька хвилин, і якщо зв'язок не відновився, зафіксувати помилку в логах, пропустити поточний цикл та зберегти стабільність системи до наступного запланованого запуску.

База даних PostgreSQL повинна зберігати свою цілісність навіть у випадках раптового вимкнення живлення комп'ютера або примусової зупинки контейнера завдяки використанню стандартних механізмів транзакцій системи керування базами даних. Будь-які операції запису нових пачок даних від конвеєра повинні або виконуватися повністю, або повністю скасовуватися, не залишаючи в таблицях частково оброблених чи розірваних записів.

4.3 Умови експлуатації

Експлуатація та розробка програмної системи ПС АЗОДВ повинні відбуватися з використанням сучасних технологій ізоляції робочого середовища, а саме інструменту Dev Containers на базі платформи Docker. Це накладає вимогу, за якою на комп'ютері користувача чи розробника має бути встановлена система керування контейнерами Docker Desktop та середовище розробки Visual Studio Code з відповідним офіційним розширенням.

Вся конфігурація середовища розробки, включаючи інтерпретатор Python, версії бібліотек обробки даних та налаштування баз даних, повинна автоматично розгортатися з конфігураційних файлів контейнера під час запуску проєкту. Користувачеві забороняється вносити ручні зміни в системні бібліотеки контейнера в обхід файлів конфігурації, щоб не порушити

відтворюваність системи на інших комп'ютерах.

4.4 Вимоги до складу та параметрів технічних засобів

Для забезпечення стабільної роботи конвеєра даних, бази даних та системи оркестрації Apache Airflow комп'ютер користувача повинен задовольняти мінімальним апаратним характеристикам. Необхідно використовувати центральний процесор з архітектурою x86-64, який має не менше чотирьох фізичних ядер та підтримує технології апаратної віртуалізації, необхідні для стабільної роботи Docker.

Об'єм оперативної пам'яті має становити щонайменше 8 гігабайт, оскільки бібліотека Pandas виконує всі аналітичні операції з трансформації великих масивів JSON безпосередньо в ОЗУ, а сама платформа оркестрації потребує постійного виділення ресурсів для своїх внутрішніх процесів. Вільне місце на твердотільному накопичувачі SSD має бути об'ємом не менше 10 гігабайт для розміщення образів Docker-контейнерів, локального сховища реляційної бази даних PostgreSQL та накопичуваних файлів текстових логів роботи конвеєра. Також обов'язковою умовою є наявність постійного підключення до мережі інтернет зі швидкістю передачі даних не менше 10 Мегабіт на секунду для забезпечення стабільного завантаження оновлень через API NVD.

4.5 Вимоги до інформаційної та програмної сумісності

Програмне забезпечення системи повинно мати чітко визначену сумісність із базовими системними компонентами та інструментами розробки. В межах ізольованого контейнера Docker система має працювати під керуванням стабільного дистрибутива операційної системи Linux, наприклад, Ubuntu чи Debian. Основним інструментом написання коду обробки виступає мова програмування Python версії 3.11 або вище. Для швидкої трансформації великих масивів даних у пам'яті обов'язковою є сумісність із бібліотекою Pandas останніх стабільних версій. Як сховище даних повинна

використовуватися реляційна система PostgreSQL версії 15 або новіша. Керування розкладом та оркестрація завдань конвеєра повинні спиратися на архітектуру Apache Airflow третьої версії або вище.

На рівні інформаційної сумісності вхідними даними для системи є структуровані файли та відповіді серверів у форматі JSON, що відповідають офіційній специфікації NVD API. Вихідними інформаційними потоками є очищені дані, які передаються до PostgreSQL через стандартні протоколи підключення баз даних і зберігаються у вигляді таблиць, сумісних зі стандартом мови запитів SQL.

5 Стадії та етапи розробки

Головні етапи розробки наведено у таблиці А.1.

Таблиця А.1 – Основні етапи виконання робіт

№	Назва проектного етапу	Термін імплементації
1.	Системний аналіз практичної задачі та дослідження предметної області розробки	24.02.2026
2.	Формулювання, аналіз та систематизація вимог до програмного забезпечення	08.03.2026
3.	Проектування загальної архітектури та структурних компонентів системи	18.03.2026
4.	Детальне моделювання та розробка проєкту програмного забезпечення	03.04.2026
5.	Програмна реалізація модулів, верифікація працездатності та тестування системи	01.05.2026

6 Порядок контролю і приймання

Контроль якості та приймання розробленого програмного забезпечення здійснюється під час демонстрації працездатності системи керівнику кваліфікаційної роботи. Критеріями успішності розробки є: – успішний запуск усього середовища проєкту через Dev Containers у Visual Studio Code без виникнення помилок конфігурації; – коректне автоматичне викачування тестової пачки даних про вразливості з API NVD за визначений проміжок часу; – відсутність помилок у логах Apache Airflow під час виконання всього

ланцюжка ETL-завдань; – наявність правильно заповнених таблиць у базі даних PostgreSQL, де інформація рознесена по відповідних сутностях без втрати зв'язків та дублювання номерів CVE; – успішне виконання демонстраційних аналітичних SQL-запитів користувача, які виводять точну інформацію про кількість знайдених критичних уразливостей у базі даних.

ДОДАТОК Б ТЕКСТ ПРОГРАМИ

Лістинг 1 – Модуль конфігурації системи (settings.py)

```

from __future__ import annotations
import os
from dataclasses import dataclass
from typing import Optional

def env(name: str, default: Optional[str] = None) -> str:
    value = os.getenv(name, default)
    if value is None:
        raise RuntimeError(f"Environment variable {name} is required")
    return value

@dataclass(slots=True)
class AppConfig:
    """Параметри запуску ETL-конвеєра."""
    nvdapibaseurl: str = env(
        "NVDAPIBASEURL",
        "https://services.nvd.nist.gov/rest/json/cves/2.0",
    )
    nvdapiskey: Optional[str] = os.getenv("NVDAPIKEY")
    nvddtimeoutseconds: int = int(os.getenv("NVDDTIMEOUTSECONDS", "30"))
    nvddresultsperpage: int = int(os.getenv("NVDDRESULTSPERPAGE", "2000"))
    nvddinitialdaysback: int = int(os.getenv("NVDDINITIALDAYSBACK", "30"))
    nvddmaxbackoffseconds: int = int(os.getenv("NVDDMAXBACKOFFSECONDS",
"90"))
    nvddmaxretriesperpage: int = int(os.getenv("NVDDMAXRETRIESPERPAGE",
"5"))

    postgreshost: str = env("POSTGRESHOST", "localhost")
    postgresport: int = int(os.getenv("POSTGRESPOST", "5432"))
    postgresdb: str = env("POSTGRESDB", "vulnmonitoring")
    postgresuser: str = env("POSTGRESUSER", "postgres")
    postgrespassword: str = env("POSTGRESPASSWORD", "postgres")
    postgresminconn: int = int(os.getenv("POSTGRESMINCONN", "1"))
    postgresmaxconn: int = int(os.getenv("POSTGRESMAXCONN", "5"))
    localworkdir: str = os.getenv("LOCALWORKDIR", "/tmp/cvepipeline")
    loglevel: str = os.getenv("LOGLEVEL", "INFO")

```

Лістинг 2 – Модуль керування підключенням до PostgreSQL (databasemanager.py)

```

from __future__ import annotations
import logging
from contextlib import contextmanager
from typing import Generator
from psycopg2.pool import SimpleConnectionPool
from config.settings import AppConfig
logger = logging.getLogger(__name__)
class DatabaseManager:
    """Менеджер пулу з'єднань PostgreSQL."""
    def __init__(self, config: AppConfig) -> None:
        self.config = config
        self.pool: SimpleConnectionPool | None = None
    def initializepool(self) -> None:
        """Ініціалізація пулу з'єднань."""
        if self.pool is not None:
            return
        logger.info("initializing PostgreSQL connection pool")
        self.pool = SimpleConnectionPool(
            minconn=self.config.postgresminconn,
            maxconn=self.config.postgresmaxconn,
            host=self.config.postgreshost,
            port=self.config.postgresport,
            dbname=self.config.postgresdb,
            user=self.config.postgresuser,
            password=self.config.postgrespassword,
        )
    def getconnection(self):
        """Отримання з'єднання з пулу."""
        if self.pool is None:
            self.initializepool()
        if self.pool is None:
            raise RuntimeError("Connection pool is not available")
        conn = self.pool.getconn()
        conn.autocommit = False
        return conn

```

```

def putconnection(self, conn) -> None:
    """Повернення з'єднання до пулу."""
    if conn is None:
        return
    if self.pool is None:
        conn.close()
        return
    self.pool.putconn(conn)
@contextmanager
def connectionscope(self) -> Generator:
    """Контекстний менеджер для безпечної роботи зі з'єднанням."""
    conn = None
    try:
        conn = self.getconnection()
        yield conn
    finally:
        self.putconnection(conn)
def closeall(self) -> None:
    """Закриття всіх активних з'єднань."""
    if self.pool is not None:
        self.pool.closeall()
        self.pool = None

```

Лістинг 3 – Репозиторії доступу до таблиць бази даних (repositories.py)

```

from __future__ import annotations
import logging
from datetime import datetime
from typing import Iterable, Sequence
from psycopg2.extras import execute_values
logger = logging.getLogger(__name__)
class ETLLogRepository:
    """Репозиторій журналу запусків ETL."""
    def getlastsuccessfulrun(self, conn) -> datetime | None:
        query = """
            SELECT lastfetchedtimestamp
            FROM etllog
            WHERE status = 'SUCCESS'

```

```

        ORDER BY rundate DESC
        LIMIT 1;
    """
    with conn.cursor() as cursor:
        cursor.execute(query)
        row = cursor.fetchone()
        return row[0] if row else None
def logsession(
    self,
    conn,
    rundate: datetime,
    lastfetchtimestamp: datetime | None,
    status: str,
    recordsprocessed: int,
) -> None:
    query = """
        INSERT INTO etllog (
            rundate,
            lastfetchtimestamp,
            status,
            recordsprocessed
        )
        VALUES (%s, %s, %s, %s);
    """
    with conn.cursor() as cursor:
        cursor.execute(query, (rundate, lastfetchtimestamp,
status, recordsprocessed))
class CVERepository:
    """Репозиторій основної таблиці CVE."""
    def upsertcve(self, conn, rows: Sequence[tuple]) -> None:
        if not rows:
            logger.info("No CVE rows for UPSERT")
            return
        query = """
            INSERT INTO cvedetails (
                cveid,
                description,

```

```

        cvssscore,
        severity,
        publisheddate,
        lastmodifieddate
    )
VALUES %s
ON CONFLICT (cveid) DO UPDATE SET
    description = EXCLUDED.description,
    cvssscore = EXCLUDED.cvssscore,
    severity = EXCLUDED.severity,
    publisheddate = EXCLUDED.publisheddate,
    lastmodifieddate = EXCLUDED.lastmodifieddate;
"""

with conn.cursor() as cursor:
    execute_values(cursor, query, rows, pagesize=1000)
    logger.info("UPSERT completed for %s CVE rows", len(rows))
class ProductRepository:
    """Репозиторій таблиці уразливих продуктів."""
    def deleteproductsbycveids(self, conn, cveids: Iterable[str]) ->
None:
        cveids = list(dict.fromkeys(cveids))
        if not cveids:
            return
        query = "DELETE FROM vulnerableproducts WHERE cveid = ANY(%s);"
        with conn.cursor() as cursor:
            cursor.execute(query, (cveids,))
        logger.info("Deleted outdated products for %s CVE identifiers",
len(cveids))
    def insertproducts(self, conn, rows: Sequence[tuple]) -> None:
        if not rows:
            return
        query = """
            INSERT INTO vulnerableproducts (
                cveid,
                vendor,
                productname,
                version

```

```

    )
    VALUES %s
    ON CONFLICT (cveid, vendor, productname, version) DO NOTHING;
"""
with conn.cursor() as cursor:
    execute_values(cursor, query, rows, pagesize=1000)
logger.info("Inserted %s product links", len(rows))

```

Лістинг 4 – Клієнт NVD API (nvdclient.py)

```

from __future__ import annotations
import logging
import time
from datetime import datetime, timezone
from typing import Any
import requests
from config.settings import AppConfig
logger = logging.getLogger(__name__)
class NVDApiClient:
    """HTTP-клієнт для NVD CVE API."""
    def __init__(self, config: AppConfig) -> None:
        self.baseurl = config.nvdapibaseurl
        self.timeout = config.nvdtimeoutseconds
        self.resultsperpage = config.nvdresultsperpage
        self.maxbackoffseconds = config.nvdmaxbackoffseconds
        self.maxretriesperpage = config.nvdmaxretriesperpage
        self.headers: dict[str, str] = {}
        if config.nvdapikey:
            self.headers["apiKey"] = config.nvdapikey
        self.session = requests.Session()
        self.session.headers.update(self.headers)
    @staticmethod
    def tonvddatetime(value: datetime) -> str:
        value = value.astimezone(timezone.utc)
        return value.strftime("%Y-%m-%dT%H:%M:%S.000")
    def handleratelimit(self, response: requests.Response, attempt: int)
-> None:
        retryafter = response.headers.get("Retry-After")

```

```

        if retryafter and retryafter.isdigit():
            sleepseconds = int(retryafter)
        else:
            sleepseconds = min(2 ** attempt, self.maxbackoffseconds)
            logger.warning("HTTP 429 from NVD API. Sleep %s seconds",
sleepseconds)
            time.sleep(sleepseconds)
    def fetchcvepage(
        self,
        startdate: datetime,
        enddate: datetime,
        startindex: int,
    ) -> dict[str, Any]:
        """Отримання однієї сторінки результатів."""
        params = {
            "lastModStartDate": self.tonvddatetime(startdate),
            "lastModEndDate": self.tonvddatetime(enddate),
            "resultsPerPage": self.resultsperpage,
            "startIndex": startindex,
        }
        for attempt in range(1, self.maxretriesperpage + 1):
            try:
                response = self.session.get(
                    self.baseurl,
                    params=params,
                    timeout=self.timeout,
                )
                if response.status_code == 429:
                    self.handleratelimit(response, attempt)
                    continue
                if 500 <= response.status_code < 600:
                    logger.warning("Temporary server error: %s",
response.status_code)
                    time.sleep(min(2**attempt, self.maxbackoffseconds))
                    continue
                response.raise_for_status()
                return response.json()

```

```

except requests.RequestException as exc:
    logger.warning("Network error on attempt %s: %s",
attempt, exc)

    if attempt == self.maxretriesperpage:
        raise
        time.sleep(min(2**attempt, self.maxbackoffseconds))
    raise RuntimeError("NVD API request failed")
def fetchwindow(self, startdate: datetime, enddate: datetime) ->
list[dict[str, Any]]:
    """Отримання всіх сторінок у межах одного часового вікна."""
    allrecords: list[dict[str, Any]] = []
    startindex = 0
    while True:
        payload = self.fetchcvepage(
            startdate=startdate,
            enddate=enddate,
            startindex=startindex,
        )
        vulnerabilities = payload.get("vulnerabilities", [])
        totalresults = payload.get("totalResults", 0)
        allrecords.extend(vulnerabilities)
        startindex += len(vulnerabilities)
        logger.info("Fetched %s/%s vulnerability records",
startindex, totalresults)
        if not vulnerabilities or startindex >= totalresults:
            break
        time.sleep(1.5)
    return allrecords

```

Лістинг 5 – Модуль інкрементального вилучення даних
(incrementalextractor.py)

```

from __future__ import annotations
import logging
from datetime import datetime, timedelta, timezone
from config.settings import AppConfig
from database.databasesmanager import DatabaseManager
from database.repositories import ETLogRepository

```

```

from extract.nvdclient import NVDApiClient
logger = logging.getLogger(__name__)
class IncrementalExtractor:
    """Координатор інкрементального вивантаження CVE."""
    MAXNVDWINDOWDAYS = 120
    def __init__(
        self,
        apiclient: NVDApiClient,
        dbmanager: DatabaseManager,
        etllogrepository: ETLLogRepository,
        config: AppConfig,
    ) -> None:
        self.apiclient = apiclient
        self.dbmanager = dbmanager
        self.etllogrepository = etllogrepository
        self.config = config
    def resolvestarttimestamp(self) -> datetime:
        with self.dbmanager.connectionscope() as conn:
            lastsucces = self.etllogrepository.getlastsuccessfulrun(conn)
            if lastsucces is not None:
                return lastsucces.astimezone(timezone.utc)
            return datetime.now(timezone.utc) -
timedelta(days=self.config.nvdinitialdaysback)
    def extractnewvulnerabilities(self) -> tuple[list[dict], datetime]:
        start = self.resolvestarttimestamp()
        now = datetime.now(timezone.utc)
        result: list[dict] = []
        windowstart = start
        lastfetched = start
        while windowstart < now:
            windowend = min(
                windowstart + timedelta(days=self.MAXNVDWINDOWDAYS),
                now,
            )
            logger.info("Extraction window: %s - %s", windowstart,
windowend)
            records = self.apiclient.fetchwindow(

```

```

        startdate=windowstart,
        enddate=windowend,
    )
    result.extend(records)
    lastfetched = windowend
    windowstart = windowend
    return result, lastfetched

```

Лістинг 6 – Модуль валідації даних (validator.py)

```

from __future__ import annotations
import logging
import re
from datetime import datetime
from typing import Any
import pandas as pd
logger = logging.getLogger(__name__)
class DataValidator:
    """Перевірка бізнес-правил для CVE, CVSS та дат."""
    CVEIDPATTERN = re.compile(r"^CVE-\d{4}-\d{4,}$")
    def validatecveid(self, cveid: str | None) -> bool:
        return bool(cveid and self.CVEIDPATTERN.match(cveid))
    def validatescore(self, score: Any) -> float | None:
        if score in (None, ""):
            return None
        try:
            score = float(score)
        except (TypeError, ValueError):
            logger.warning("Invalid CVSS score: %s", score)
            return None
        if not 0.0 <= score <= 10.0:
            logger.warning("Out-of-range CVSS score: %s", score)
            return None
        return score
    def validatetimestamp(self, value: str | None) -> datetime | None:
        if not value:
            return None
        try:

```

```

        return pd.to_datetime(value, utc=True).to_pydatetime()
    except Exception:
        logger.warning("Invalid timestamp: %s", value)
        return None

def validateandfiltercves(self, record: dict[str, Any]) -> bool:
    cveid = record.get("cveid")
    if not self.validatecveid(cveid):
        logger.warning("Record skipped due to invalid CVE ID: %s",
cveid)

        return False
    return True

```

Лістинг 7 – Модуль розбору CPE-рядків (cpeparser.py)

```

from __future__ import annotations
import logging
from typing import Any
logger = logging.getLogger(__name__)
class CPEParser:
    """Парсер CPE 2.3."""
    @staticmethod
    def normalizecomponent(value: str) -> str | None:
        if value in {"", "-", ""}:
            return None
        return value
    def parsecpestring(self, cpeuri: str) -> dict[str, str | None] |
None:
        if not cpeuri or not cpeuri.startswith("cpe:2.3:"):
            return None
        parts = cpeuri.split(":")
        if len(parts) < 6:
            logger.warning("Malformed CPE string: %s", cpeuri)
            return None
        return {
            "vendor": self.normalizecomponent(parts[3]),
            "productname": self.normalizecomponent(parts[4]),
            "version": self.normalizecomponent(parts[5]),
        }

```

```

def extractproductsfromconfigurations(
    self,
    cveid: str,
    configurations: list[dict[str, Any]],
) -> list[dict[str, str | None]]:
    products: list[dict[str, str | None]] = []
    def walknodes(nodes: list[dict[str, Any]]) -> None:
        for node in nodes:
            for cpematch in node.get("cpeMatch", []):
                if cpematch.get("vulnerable", True) is False:
                    continue
                criteria = cpematch.get("criteria", "")
                parsed = self.parsecpestring(criteria)
                if parsed:
                    products.append(
                        {
                            "cveid": cveid,
                            "vendor": parsed["vendor"],
                            "productname": parsed["productname"],
                            "version": parsed["version"],
                        }
                    )
            children = node.get("children", [])
            if children:
                walknodes(children)
    for config in configurations:
        walknodes(config.get("nodes", []))
    return products

```

Лістинг 8 – Модуль трансформації даних (transformer.py)

```

from __future__ import annotations
import logging
import re
from typing import Any
import pandas as pd
from transform.cpeparser import CPEParser
from transform.validator import DataValidator

```

```

logger = logging.getLogger(__name__)
class CVETransformer:
    """Клас трансформації CVE-записів."""
    CLEANTEXTPATTERN = re.compile(r"[\r\n\t]+")
    MULTISPACEPATTERN = re.compile(r"\s{2,}")
    def __init__(self, validator: DataValidator, cpeparser: CPEParser)
-> None:
        self.validator = validator
        self.cpeparser = cpeparser
    @staticmethod
    def selectenglishdescription(descriptions: list[dict[str, Any]]) ->
str:
        return next(
            (item.get("value", "") for item in descriptions if
item.get("lang") == "en"),
            "",
        )
    def cleandescription(self, text: str) -> str:
        text = self.CLEANTEXTPATTERN.sub(" ", text or "")
        text = self.MULTISPACEPATTERN.sub(" ", text)
        return text.strip()
    @staticmethod
    def extractcvss(metrics: dict[str, Any]) -> tuple[float | None, str
| None]:
        candidates = [
            metrics.get("cvssMetricV31", []),
            metrics.get("cvssMetricV30", []),
            metrics.get("cvssMetricV2", []),
        ]
        for candidate in candidates:
            if candidate:
                entry = candidate[0]
                score = entry.get("cvssData", {}).get("baseScore")
                severity = entry.get("cvssData", {}).get("baseSeverity")
or entry.get("baseSeverity")
                return score, severity
        return None, None

```

```

def transformrawdata(
    self,
    rawdata: list[dict[str, Any]],
) -> tuple[pd.DataFrame, pd.DataFrame]:
    cverows: list[dict[str, Any]] = []
    productrows: list[dict[str, Any]] = []
    for item in rawdata:
        cvenode = item.get("cve", {})
        cveid = cvenode.get("id")
        rawscore, severity = self.extractcvss(cvenode.get("metrics",
{}))

        flatcverow = {
            "cveid": cveid,
            "description": self.cleandescription(
                self.selectenglishdescription(cvenode.get("descript
ions", []))
            ),
            "cvssscore": self.validator.validatescore(rawscore),
            "severity": severity,
            "publisheddate":
self.validator.validatetimestamp(cvenode.get("published")),
            "lastmodifieddate":
self.validator.validatetimestamp(cvenode.get("lastModified")),
        }
        if not self.validator.validateandfiltercves(flatcverow):
            continue
        cverows.append(flatcverow)
        productrows.extend(
            self.cpeparser.extractproductsfromconfigurations(
                cveid=cveid,
                configurations=cvenode.get("configurations", []),
            )
        )
    dfcve = pd.DataFrame(
        cverows,
        columns=[
            "cveid",

```

```

        "description",
        "cvssscore",
        "severity",
        "publisheddate",
        "lastmodifieddate",
    ],
)
dfproducts = pd.DataFrame(
    productrows,
    columns=["cveid", "vendor", "productname", "version"],
)
if not dfcve.empty:
    dfcve =
dfcve.drop_duplicates(subset=["cveid"]).reset_index(drop=True)
if not dfproducts.empty:
    dfproducts = (
        dfproducts.drop_duplicates(subset=["cveid", "vendor",
"productname", "version"])
        .reset_index(drop=True)
    )
    logger.info("Transformed %s CVE rows and %s product rows",
len(dfcve), len(dfproducts))
    return dfcve, dfproducts

```

Лістинг 9 – Модуль транзакційного завантаження (loader.py)

```

from __future__ import annotations
import logging
from datetime import datetime, timezone
import pandas as pd
from database.databasemanager import DatabaseManager
from database.repositories import CVERepository, ETLLogRepository,
ProductRepository
logger = logging.getLogger(__name__)
class CVELoader:
    """Координатор фінального етапу Load."""
    def __init__(
        self,

```

```

        dbmanager: DatabaseManager,
        cverepository: CVERepository,
        productrepository: ProductRepository,
        etllogrepository: ETLLogRepository,
    ) -> None:
        self.dbmanager = dbmanager
        self.cverepository = cverepository
        self.productrepository = productrepository
        self.etllogrepository = etllogrepository

    @staticmethod
    def frametotuples(df: pd.DataFrame) -> list[tuple]:
        if df.empty:
            return []
        return [tuple(row) for row in df.where(pd.notna(df),
None).to_numpy()]

    def logfailedrun(self, recordsprocessed: int) -> None:
        try:
            with self.dbmanager.connectionscope() as conn:
                self.etllogrepository.logsession(
                    conn,
                    rundate=datetime.now(timezone.utc),
                    lastfetchtimestamp=None,
                    status="FAILED",
                    recordsprocessed=recordsprocessed,
                )
            conn.commit()
        except Exception:
            logger.exception("Failed to write FAILED status to etllog")

    def loadprocesseddata(
        self,
        dfcve: pd.DataFrame,
        dfproducts: pd.DataFrame,
        lastfetchtimestamp: datetime,
    ) -> None:
        cverows = self.frametotuples(dfcve)
        productrows = self.frametotuples(dfproducts)
        recordsprocessed = len(cverows)

```

```

        with self.dbmanager.connectionscope() as conn:
            try:
                self.cverepository.upsertcve(conn, cverows)
                if not dfproducts.empty:
                    ids = dfproducts["cveid"].dropna().unique().tolist()
                    self.productrepository.deleteproductsbycveids(conn,
ids)
                    self.productrepository.insertproducts(conn,
productrows)

                self.etllogrepository.logsession(
                    conn,
                    rundate=datetime.now(timezone.utc),
                    lastfetchedtimestamp=lastfetchedtimestamp,
                    status="SUCCESS",
                    recordsprocessed=recordsprocessed,
                )
                conn.commit()
                logger.info("Load committed: CVE=%s products=%s",
len(cverows), len(productrows))
            except Exception:
                conn.rollback()
                logger.exception("Load failed; transaction rolled back")
                self.logfailedrun(recordsprocessed)
                raise

```

Лістинг 10 – Модуль оркестрації Apache Airflow (cverpipelinedag.py)

```

from __future__ import annotations
import json
import logging
import os
from datetime import datetime, timedelta, timezone
import pandas as pd
from airflow import DAG
from airflow.operators.python import PythonOperator
from config.settings import AppConfig
from database.databasemanager import DatabaseManager
from database.repositories import CVERepository, ETLLogRepository,

```

ProductRepository

```

from extract.incrementalextractor import IncrementalExtractor
from extract.nvdclient import NVDApiClient
from load.loader import CVELoader
from transform.cpeparser import CPEParser
from transform.transformer import CVETransformer
from transform.validator import DataValidator
logger = logging.getLogger(__name__)
class AirflowPipelineDAG:
    """Клас-обгортка для callable-функцій Airflow."""
    def __init__(self) -> None:
        config = AppConfig()
        self.workdir = config.localworkdir
        os.makedirs(self.workdir, exist_ok=True)
        self.dbmanager = DatabaseManager(config)
        self.dbmanager.initializepool()
        self.etllogrepository = ETLLogRepository()
        self.cverepository = CVERepository()
        self.productrepository = ProductRepository()
        self.extractor = IncrementalExtractor(
            apiclient=NVDApiClient(config),
            dbmanager=self.dbmanager,
            etllogrepository=self.etllogrepository,
            config=config,
        )
        self.transformer = CVETransformer(
            validator=DataValidator(),
            cpeparser=CPEParser(),
        )
        self.loader = CVELoader(
            dbmanager=self.dbmanager,
            cverepository=self.cverepository,
            productrepository=self.productrepository,
            etllogrepository=self.etllogrepository,
        )
        def extracttaskcallable(self, context) -> dict:
            rawrecords, lastfetched = self.extractor.extractnewvulnerabilities()

```

```

        rawpath = os.path.join(self.workdir, "rawcvepayload.json")
        metapath = os.path.join(self.workdir, "extractmeta.json")
        with open(rawpath, "w", encoding="utf-8") as fh:
            json.dump(rawrecords, fh, ensureascii=False)
        with open(metapath, "w", encoding="utf-8") as fh:
            json.dump({"lastfetchtimestamp":
lastfetched.isoformat()}, fh)
            logger.info("Extract task wrote %s raw records",
len(rawrecords))
            return {"rawpath": rawpath, "metapath": metapath}
def transformtaskcallable(self, context) -> dict:
    ti = context["ti"]
    payload = ti.xcompull(taskids="extracttask")
    with open(payload["rawpath"], "r", encoding="utf-8") as fh:
        rawrecords = json.load(fh)
    dfcve, dfproducts = self.transformer.transformrawdata(rawrecords)
    cvecsvpath = os.path.join(self.workdir, "cveclean.csv")
    productscsvpath = os.path.join(self.workdir,
"productsclean.csv")
    dfcve.to_csv(cvecsvpath, index=False)
    dfproducts.to_csv(productscsvpath, index=False)
    return {
        "cvecsvpath": cvecsvpath,
        "productscsvpath": productscsvpath,
        "metapath": payload["metapath"],
    }
def loadtaskcallable(self, context) -> None:
    ti = context["ti"]
    payload = ti.xcompull(taskids="transformtask")
    dfcve = pd.read_csv(payload["cvecsvpath"])
    dfproducts = pd.read_csv(payload["productscsvpath"])
    if not dfcve.empty:
        dfcve["publisheddate"] =
pd.to_datetime(dfcve["publisheddate"], utc=True, errors="coerce")
        dfcve["lastmodifieddate"] =
pd.to_datetime(dfcve["lastmodifieddate"], utc=True, errors="coerce")
        with open(payload["metapath"], "r", encoding="utf-8") as fh:

```

```

        meta = json.load(fh)

        lastfetched =
datetime.fromisoformat(meta["lastfetchedtimestamp"])
        if lastfetched.tzinfo is None:
            lastfetched = lastfetched.replace(tzinfo=timezone.utc)
        self.loader.loadprocesseddata(
            dfcve=dfcve,
            dfproducts=dfproducts,
            lastfetchedtimestamp=lastfetched,
        )
    pipeline = AirflowPipelineDAG()
    defaultargs = {
        "owner": "airflow",
        "depends_on_past": False,
        "retries": 3,
        "retry_delay": timedelta(minutes=5),
    }
    with DAG(
        dag_id="cvevulnerabilitypipeline",
        description="Automated ETL pipeline for collecting and processing
NVD CVE data",
        schedule="@daily",
        startdate=datetime(2026, 1, 1),
        catchup=False,
        default_args=default_args,
        tags=["cve", "nvd", "etl", "postgresql"],
    ) as dag:
        extracttask = PythonOperator(
            task_id="extracttask",
            python_callable=pipeline.extracttaskcallable,
        )
        transformtask = PythonOperator(
            taskid="transformtask",
            pythoncallable=pipeline.transformtaskcallable,
        )
        loadtask = PythonOperator(
            taskid="loadtask",

```

```

        pythoncallable=pipeline.loadtaskcallable,
    )
extracttask >> transformtask >> loadtask

```

Лістинг 11 – Допоміжні SQL-запити для контролю та аналітики (reportqueries.sql)

```

-- 1. Останні успішні запуски ETL-конвеєра.
SELECT
    rundate,
    lastfetchedtimestamp,
    status,
    recordsprocessed
FROM etllog
ORDER BY rundate DESC
LIMIT 20;

-- 2. Невдалі запуски ETL-конвеєра.
SELECT
    rundate,
    status,
    recordsprocessed
FROM etllog
WHERE status = 'FAILED'
ORDER BY rundate DESC;

-- 3. Контрольна кількість рядків у таблицях сховища.
SELECT 'cvedetails' AS tablename, COUNT(*) AS rowcount FROM cvedetails
UNION ALL
SELECT 'vulnerableproducts' AS tablename, COUNT(*) AS rowcount FROM
vulnerableproducts
UNION ALL
SELECT 'etllog' AS tablename, COUNT(*) AS rowcount FROM etllog;

-- 4. Останні критичні вразливості.
SELECT
    cveid,
    severity,

```

```

        cvssscore,
        publisheddate,
        lastmodifieddate
FROM cvedetails
WHERE severity = 'CRITICAL'
ORDER BY lastmodifieddate DESC
LIMIT 50;

```

-- 5. Контрольне вивантаження даних для демонстраційної таблиці.

```

SELECT
    c.cveid,
    c.cvssscore,
    c.publisheddate::date AS publisheddate,
    p.productname AS product,
    p.vendor,
    p.version
FROM cvedetails AS c
JOIN vulnerableproducts AS p
    ON p.cveid = c.cveid
ORDER BY c.publisheddate DESC, c.cveid, p.vendor, p.productname
LIMIT 100;

```

-- 6. Топ виробників за кількістю критичних уразливостей.

```

SELECT
    p.vendor,
    COUNT(DISTINCT c.cveid) AS criticalcvecount
FROM cvedetails AS c
JOIN vulnerableproducts AS p
    ON p.cveid = c.cveid
WHERE c.cvssscore >= 9.0
GROUP BY p.vendor
ORDER BY criticalcvecount DESC, p.vendor
LIMIT 20;

```

-- 7. CVE, оновлені за останні 7 днів.

```

SELECT
    cveid,

```

```

        severity,
        cvssscore,
        lastmodifieddate
FROM cvedetails
WHERE lastmodifieddate >= NOW() - INTERVAL '7 days'
ORDER BY lastmodifieddate DESC;

```

-- 8. Перевірка дублікатів у vulnerableproducts.

```

SELECT
    cveid,
    vendor,
    productname,
    version,
    COUNT(*) AS duplicates
FROM vulnerableproducts
GROUP BY cveid, vendor, productname, version
HAVING COUNT(*) > 1;

```

-- 9. CVE без пов'язаних продуктів.

```

SELECT
    c.cveid,
    c.severity,
    c.lastmodifieddate
FROM cvedetails AS c
LEFT JOIN vulnerableproducts AS p
    ON p.cveid = c.cveid
WHERE p.cveid IS NULL
ORDER BY c.lastmodifieddate DESC
LIMIT 50;

```

-- 10. Продуктивність ETL за днями.

```

SELECT
    DATE(rundate) AS runday,
    SUM(recordsprocessed) AS processedtotal
FROM etllog
WHERE status = 'SUCCESS'
GROUP BY DATE(rundate) ORDER BY runday DESC;

```

ДОДАТОК В ОПИС ПРОГРАМИ

1 Загальні відомості

Програмна система автоматизованого збору та обробки даних про кібервразливості призначен для забезпечення безперервного циклу імпорту та структурування даних у сфері інформаційної безпеки. Програма реалізує логіку сервісного рівня та орієнтована на автономне функціонування у фоновому режимі серверної операційної системи під управлінням середовища Docker контейнеризації.

Повні лістинги програмної реалізації наведено у Додатку Б. Вони включають модулі Python для шарів Extract, Transform, Load, клас керування DAG в Apache Airflow, репозиторії доступу до PostgreSQL, а також допоміжні SQL-запити для верифікації результатів завантаження та формування аналітичних звітів зі сховища даних.

2 Функціональна структура та опис логічних модулів

Програмне забезпечення побудовано за модульним архітектурним принципом та складається з чотирьох ключових елементів. Компонент оркестрації забезпечує створення спрямованого ациклічного графа задач в Apache Airflow, контролює послідовність викликів інженерних скриптів, керує станами виконання та здійснює передачу метаданих між кроками за допомогою механізму внутрішніх комунікацій. Модуль екстракції відповідає за низькорівневу взаємодію з віддаленим сервером за протоколом HTTP REST API, реалізує логіку інкрементального сканування, зчитує системний лог попередніх запусків та розраховує динамічний часовий фільтр, а також містить вбудований інтервальний таймер затримок для дотримання політики обмежень джерела. Модуль очищення та декомпозиції здійснює розбір текстових масивів JSON в оперативній пам'яті за допомогою структур даних DataFrame бібліотеки Pandas, виконує нормалізацію описів регулярними виразами, валідує числові бали CVSS та проводить деструктуризацію рядків

CPE для відокремлення атрибутів розробника, назви ПЗ та його версій. Модуль транзакційного завантаження забезпечує з'єднання з СКБД PostgreSQL через драйвер, реалізує операції злиття даних за принципом оновлення конфліктів для запобігання дублюванню первинних ключів і гарантує збереження реляційної цілісності даних.

3 Технічні вимоги до програмного середовища

Для успішного виконання та розгортання розробленого програмного забезпечення необхідна наявність стабільного системного оточення. Базовою операційною системою має виступати Linux або Windows із підсистемою WSL другої версії. Середовище контейнеризації повинно спиратися на Docker Engine версії 24 або новішої, а також Docker Compose версії 2.20 або вище. Як основний інтерпретатор використовується Python версії 3.11 із встановленими сторонніми бібліотеками обробки даних Pandas, клієнтом Requests та драйвером бази даних Psycopg2.

ДОДАТОК Г ІНСТРУКЦІЯ КОРИСТУВАЧА

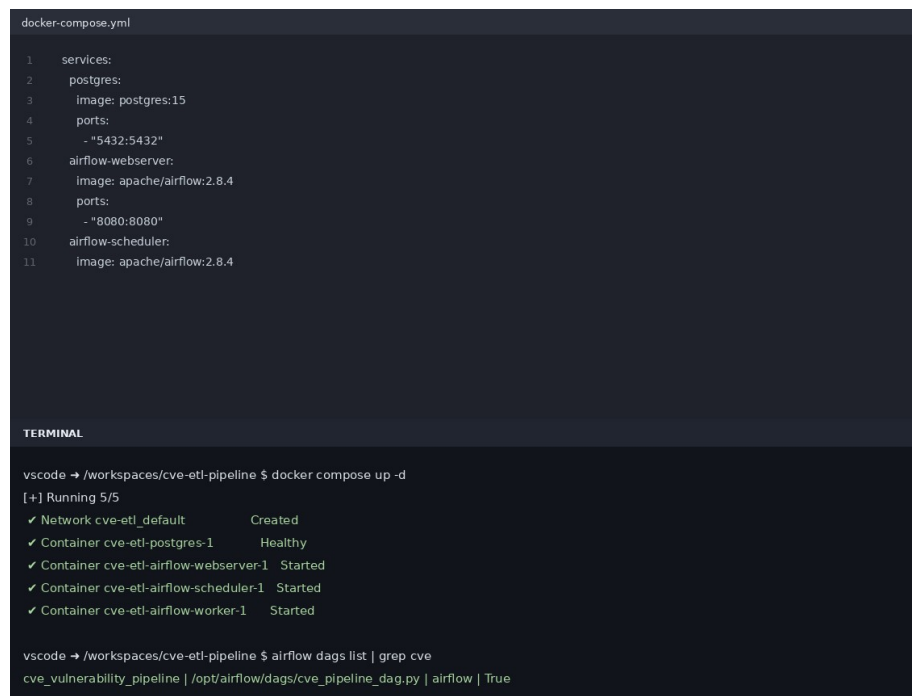
1 Порядок розгортання та запуску системи

Програмна система постачається у вигляді проєкту, який може бути запуснений у контейнеризованому середовищі. Для первинного розгортання користувач повинен встановити Visual Studio Code, Docker Desktop та розширення Dev Containers. Після цього необхідно відкрити папку проєкту у Visual Studio Code і підтвердити запуск середовища в контейнері.

Після відкриття проєкту користувач запускає локальні сервіси командою:

```
docker compose up -d
```

У результаті мають бути запущені контейнери PostgreSQL, Apache Airflow Webserver, Apache Airflow Scheduler та робочий процес Airflow Worker. Приклад успішного запуску середовища розробки наведено на рисунку Г.1.



```
docker-compose.yml
1  services:
2    postgres:
3      image: postgres:15
4      ports:
5        - "5432:5432"
6    airflow-webserver:
7      image: apache/airflow:2.8.4
8      ports:
9        - "8080:8080"
10   airflow-scheduler:
11     image: apache/airflow:2.8.4

TERMINAL

vscode → /workspaces/cve-etl-pipeline $ docker compose up -d
[+] Running 5/5
✓ Network cve-etl_default      Created
✓ Container cve-etl-postgres-1  Healthy
✓ Container cve-etl-airflow-webserver-1 Started
✓ Container cve-etl-airflow-scheduler-1 Started
✓ Container cve-etl-airflow-worker-1 Started

vscode → /workspaces/cve-etl-pipeline $ airflow dags list | grep cve
cve_vulnerability_pipeline | /opt/airflow/dags/cve_pipeline_dag.py | airflow | True
```

Рисунок Г.1 – Запуск контейнеризованого середовища

Після успішного запуску контейнерів вебінтерфейс Apache Airflow стає

доступним у браузері за локальною адресою:

<http://localhost:8080>

Сховище PostgreSQL доступне на порту 5432, а для виконання SQL-запитів можна використовувати pgAdmin, DBeaver, DataGrip або інший SQL-клієнт.

2 Робота з графічним інтерфейсом Apache Airflow

Для контролю роботи автоматизованого конвеєра користувач переходить до вебінтерфейсу Apache Airflow і відкриває список DAG. У списку необхідно знайти DAG з назвою `cve\_vulnerability\_pipeline`. Саме цей граф відповідає за виконання повного циклу ETL-обробки даних про кібервразливості.

У вебінтерфейсі користувач може переглянути статус останнього запуску, час наступного запуску, стан окремих задач і загальну історію виконання. Для ручного запуску поза розкладом потрібно натиснути кнопку запуску DAG у колонці Actions. Приклад списку DAG та запуску конвеєра наведено на рисунку Г.2.

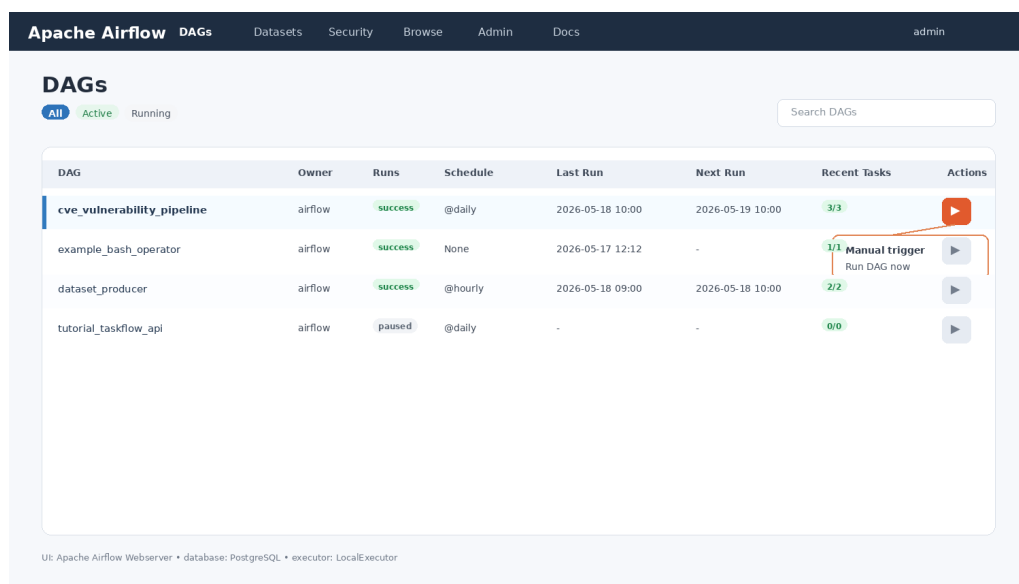


Рисунок Г.2 – Ручний запуск DAG `cve\_vulnerability\_pipeline` в Apache Airflow

Після запуску користувач може перейти до режиму Graph View. У цьому

режимі відображається послідовність задач `extract\_task`, `transform\_task` та `load\_task`. Зелений статус задач означає, що відповідний етап завершився успішно. Приклад графа виконання наведено на рисунку Г.3.

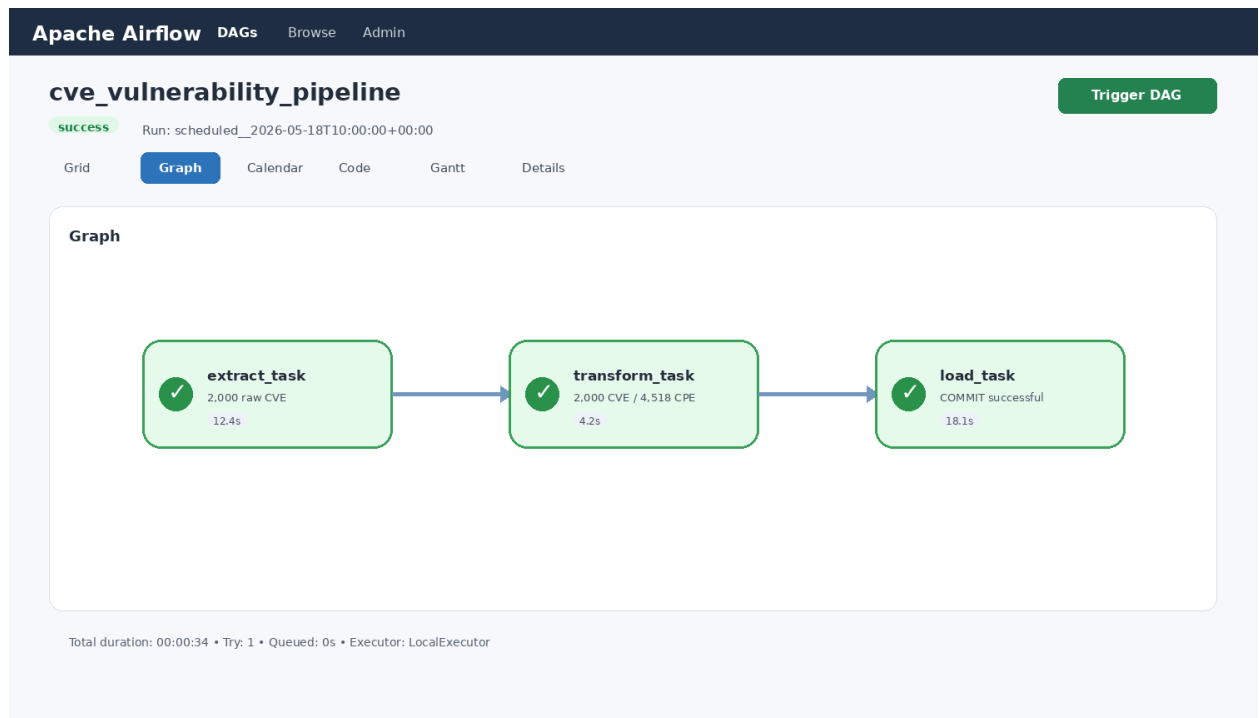


Рисунок Г.3 – Перегляд послідовності задач ETL-конвеєра в Apache Airflow

Для діагностики помилок користувач може натиснути на конкретну задачу та відкрити її журнал виконання. У журналі фіксуються повідомлення про підключення до NVD API, кількість отриманих записів, результати трансформації, кількість завантажених рядків і статус завершення транзакції.

3 Отримання аналітичних даних аналітиком безпеки

Після завершення роботи конвеєра користувач може виконувати аналітичні SQL-запити до PostgreSQL. Для цього необхідно відкрити SQL-клієнт, підключитися до бази даних `vuln\_monitoring` і виконати запити до таблиць `cve\_details`, `vulnerable\_products` та `etl\_log`.

Найтипівішим сценарієм є пошук критичних вразливостей для конкретних програмних продуктів або виробників. Для цього

використовується об'єднання таблиць `cve\_details` та `vulnerable\_products` за полем `cve\_id`. Приклад виконання аналітичного SQL-запиту наведено на рисунку Г.4.

The screenshot shows the pgAdmin 4 interface. On the left is the Object Explorer with a tree view showing the database structure: Servers > PostgreSQL 15 > Databases > vuln_monitoring > Schemas > public > Tables > cve_details, vulnerable_products, etl_log. The main area is the Query Tool, which contains a Query Editor and a Data Output section.

Query Editor

SELECT c.cve_id, c.cvss_score, c.published_date::date,
p.product_name AS product, p.vendor, p.version
FROM cve_details AS c
JOIN vulnerable_products AS p ON p.cve_id = c.cve_id
WHERE c.cvss_score >= 5.0
ORDER BY c.cvss_score DESC, c.published_date DESC
LIMIT 8;

Data Output

cve_id	cvss_score	published_date	product	vendor	version
CVE-2026-1024	9.8	2026-05-12	exchange_server	microsoft	2019_cu12
CVE-2026-1024	9.8	2026-05-12	exchange_server	microsoft	2022_cu02
CVE-2026-2107	9.1	2026-05-18	openssl	openssl	3.0.12
CVE-2026-1189	8.8	2026-05-14	linux_kernel	linux	5.15.0
CVE-2026-1189	8.8	2026-05-14	linux_kernel	linux	6.1.12
CVE-2026-2142	8.1	2026-05-18	nginx	nginx	1.25.4
CVE-2026-2001	7.5	2026-05-15	tomcat	apache	10.1.5
CVE-2026-2045	5.3	2026-05-17	chrome	google	121.0.0

Query returned successfully in 184 ms. 8 rows returned.

Рисунок Г.4 – Виконання аналітичного SQL-запиту до PostgreSQL

Таким чином, користувач може не лише контролювати автоматичне виконання ETL-конвеєра через Apache Airflow, а й отримувати практичні аналітичні результати безпосередньо з реляційного сховища PostgreSQL.

ДОДАТОК Д ПРОГРАМА ТА МЕТОДИКА ВИПРОБУВАНЬ

1 Об'єкт та мета випробувань

Об'єктом випробувань є програмна система автоматизованого збору та обробки даних про кібервразливості, реалізована на базі мови Python, СКБД PostgreSQL та платформи Apache Airflow. Метою проведення випробувань є експериментальна перевірка відповідності розробленого програмного забезпечення технічним вимогам, підтвердження працездатності механізмів інкрементального завантаження даних, стійкості системи до інфраструктурних збоїв та обмежень сервера джерела, а також оцінка швидкодії модулів очищення даних.

2 Склад та методика проведення тестових сценаріїв

Для комплексного оцінювання системи розроблено та виконано серію функціональних та інтеграційних тестів, результати яких зведені у таблицю.

Таблиця Д.1 – Журнал виконання та результатів випробувань програмної системи

№ тесту	Опис тестового сценарію	Очікуваний інженерний результат	Фактичний результат випробування	Статус
1	2	3	4	5
T-01	Первинне підключення до API за порожньої бази даних.	Вилучення повної поточної сторінки записів та успішне створення таблиць сховища.	Конвеєр успішно створив структуру таблиць та завантажив перші дві тисячі записів.	Успішно
T-02	Перевірка працездатності інкрементального режиму збору.	Програма зчитує мітку часу та запитує з API лише оновлені за останню добу записи.	Надіслано запит із часовим фільтром. Отримано та додано лише сорок два нові записи.	Успішно
T-03	Симуляція перевищення ліміту запитів від сервера джерела.	Модуль переходить у режим очікування на шістдесят секунд без аварійної зупинки.	Отримано відповідь HTTP 429. Скрипт призупинив виконання, зачекав хвилину та продовжив роботу.	Успішно

Кінець таблиці Д.1

1	2	3	4	5
T-04	Валідація та очищення введеного тексту із символами табуляції.	Очищення тексту від службових символів та декомпозиція рядків платформ на атрибути.	Текст опису повністю нормалізовано. CPE-рядки коректно розбито на атрибути.	Успішно
T-05	Інтеграційний тест злиття при повторному завантаженні наявної у базі CVE.	Оновлення текстового опису та балів CVSS без дублювання запису у таблиці.	Дані успішно оновлено в наявному рядку бази. Кількість унікальних ключів не змінилася.	Успішно
T-06	Стрес-тест швидкодії на пакеті з двох тисяч складних записів.	Обробка, трансформація та приведення типів у пам'яті мають тривати менше п'яти секунд.	Середній час обробки великого пакета в Pandas склав 4,18 с.	Успішно

3 Висновки за результатами випробувань

Комплексні випробування розробленого програмного забезпечення підтвердили його повну працездатність, надійність та відповідність усім вимогам, затвердженим у Технічному завданні. Механізми автоматичної обробки мережових помилок та інкрементального наповнення бази даних функціонують коректно, що підтверджує повну готовність програмної системи до практичної експлуатації в підрозділах кібербезпеки організації.