Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

бакалавр
(назва освітнього ступеня)

на тему: Розробка програмного забезпечення для централізованого збору та управління сповіщеннями з використанням хмарних технологій

Виконав(ла):
студент(ка) 4 курсу групи СП-42
спеціальності 121
«Інженерія програмного забезпечення»
(шифр і назва спеціальності)

(підпис) Кормило А. Р.
(прізвище та ініціали)

Керівник (підпис) Бревус В. М.
(прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю. М.
(прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М. Р.
(прізвище та ініціали)

Рецензент (підпис) Липак Г. І.
(прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет _____
(повна назва факультету)

Кафедра _____
(повна назва кафедри)

ЗАТВЕРДЖУЮ
Завідувач кафедри

_____ _____
(підпис) (прізвище та ініціали)

« » 20___ р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня Бакалавра
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Кормилу Андрію Романовичу
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка програмного забезпечення для централізованого збору оповіщень з використанням хмарних технологій

Керівник роботи Бревус В. М. к.т.н.доцент кафедри програмної інженерії
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «*18*» *04* 2025 року № *4/7 - 334*

2. Термін подання студентом завершеної роботи _____

3. Вихідні дані до роботи модель розробки – ітеративна; вхідні дані – назва сервісу, правила аналізу онлайн-ресурсів, дані облікового запису; вихідні дані – текстові оповіщення від зареєстрованих сервісів, текстовий звіт, зображення діаграми звіту; середовище розробки – JetBrains PyCharm; мова програмування - Python.

4. Зміст роботи (перелік питань, які потрібно розробити)
вступ; обґрунтування вибору методу розробки та постановка задачі дослідження; розробка методів збору та обробки оповіщень; розробка структури і програмних компонентів; тестування

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)
блок-схема алгоритму отримання оповіщень з використанням вебхуків та публічних API; блок-схема алгоритму отримання оповіщень з використанням парсингу веб-ресурсів; блок-схема алгоритму генерації звітів на основі зібраних оповіщень; структура графічного інтерфейсу головного вікна додатку; структура графічного інтерфейсу вікна створення джерела сповіщень; структура графічного інтерфейсу вікна звіту.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка

Студент _____ _____
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Кваліфікаційна робота бакалавра. Тернопільський національний технічний університет імені Івана Пулюя, кафедра програмної інженерії, спеціальність 121 «Інженерія програмного забезпечення». ТНТУ, 2025. Сторінок 93, рисунків 31, додатків 1, презентація.

Тема: Розробка програмного забезпечення для централізованого збору та управління сповіщеннями з використанням хмарних технологій.

Кваліфікаційна робота бакалавра присвячена розробці програмного рішення «Notification Collector», яке забезпечує централізований збір, зберігання та аналіз сповіщень із різних джерел за допомогою хмарних технологій. Метою роботи є створення ефективного інструменту для зменшення інформаційного навантаження на користувачів шляхом автоматизації обробки повідомлень.

У першому розділі проведено аналіз сучасних хмарних технологій, зокрема безсерверних обчислень, та досліджено існуючі аналоги систем для збору сповіщень. Визначено вимоги до програмного забезпечення, обґрунтовано вибір технологій (Python, AWS Lambda, DynamoDB) та розроблено алгоритми отримання сповіщень через вебхуки, API та парсинг веб-ресурсів.

Другий розділ описує проектування архітектури системи, включаючи серверну частину на базі AWS (API Gateway, Cognito, SNS, SQS) та клієнтський інтерфейс з використанням бібліотеки Tkinter. Детально розглянуто процеси генерації звітів, тестування функціоналу та створення інструкції для користувачів.

Об'єкт дослідження: процеси збору, обробки та зберігання інформаційних повідомлень.

Предмет дослідження: технології та методи централізованого управління сповіщеннями з використанням хмарних сервісів.

Ключові слова: хмарні технології, безсерверні обчислення, централізований збір сповіщень, AWS Lambda, Python, вебхуки, парсинг.

ABSTRACT

Bachelor's certification work. Ternopil Ivan Puluj National Technical University, Department of Software Engineering, specialty 121 "Software Engineering". TNTU, 2025. Pages 79, figures 31, appendices 1, presentation.

Topic: Development of software for centralized collection and management of notifications using cloud technologies.

The bachelor's certification work is dedicated to the development of the "Notification Collector" software solution, which provides centralized collection, storage, and analysis of notifications from various sources using cloud technologies. The aim of the work is to create an efficient tool for reducing users' information overload by automating message processing.

The first section analyzes modern cloud technologies, including serverless computing, and explores existing analogs of notification collection systems. Requirements for the software are defined, the choice of technologies (Python, AWS Lambda, DynamoDB) is justified, and algorithms for receiving notifications via webhooks, APIs, and web scraping are developed.

The second section describes the system architecture design, including the server-side based on AWS (API Gateway, Cognito, SNS, SQS) and the client interface using the Tkinter library. The processes of report generation, functionality testing, and user manual creation are discussed in detail.

Object of research: processes of collecting, processing, and storing informational messages.

Subject of research: technologies and methods for centralized notification management using cloud services.

Keywords: cloud technologies, serverless computing, centralized notification collection, AWS Lambda, Python, webhooks, parsing.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. З кожним роком інформаційний потік, що оточує людину, суттєво зростає. Завдяки стрімкому розвитку технологій доступ до новин, комунікації з друзями та колегами, а також обмін робочими чи навчальними матеріалами стали значно швидшими та зручнішими. Сучасні інтернет-сервіси та програмне забезпечення автоматизували багато аспектів інформаційного обміну, а системи оповіщень і push-сповіщення забезпечують оперативність отримання нових даних.

Такі рішення розробляються для комфорту користувачів і водночас сприяють популяризації відповідних платформ. Проте велика кількість активних каналів інформації створює новий виклик: надлишок повідомлень, що надходять із різних джерел. Постійні оповіщення можуть заважати концентрації, особливо під час робочих процесів або навчання. Крім того, автоматичні розсилки не враховують особистий графік користувача, що може призводити до інформаційного перенасичення.

Деякі користувачі вдаються до вимкнення оповіщень, однак це не завжди можливо, зокрема для тих, хто залежить від оперативного реагування на повідомлення. Вирішенням проблеми може стати спеціальне програмне забезпечення, що централізовано збирає та структурує сповіщення. На ринку вже є відповідні продукти, але вони мають певні обмеження: недостатню інтеграцію з різними сервісами, низьку підтримку десктопних систем або відсутність механізмів для гнучкої генерації звітів.

Використання хмарних технологій у розробці такого програмного рішення дозволяє знизити витрати на інфраструктуру, спростити архітектуру додатку та полегшити його масштабування. Саме тому актуальним є дослідження ефективних методів збору та управління оповіщеннями, що сприятиме підвищенню продуктивності користувачів.

Актуальність дослідження та його зв’язок із науковими проєктами

Проведене дослідження є частиною наукової діяльності кафедри програмного забезпечення та реалізується в межах її науково-дослідної програми. Робота відповідає напрямам, передбаченим планом досліджень, і спрямована на вдосконалення сучасних підходів до обробки інформаційних потоків.

Мета та завдання дослідження:

Ключовою метою даної роботи є створення програмного рішення, яке забезпечить ефективне управління сповіщеннями за рахунок їх автоматизованого збору, централізованого зберігання та подальшого аналізу. Основні завдання включають розробку інструментів, що дозволяють зменшити навантаження на користувача під час взаємодії з великою кількістю повідомлень.

Для досягнення мети необхідно:

- Дослідити технології хмарних безсерверних обчислень.
- Аналізувати існуючі методи збору сповіщень.
- Розробити алгоритми інтеграції даних через вебхуки та API.
- Запропонувати методику збору оповіщень шляхом парсингу веб-ресурсів.
- Сформувати алгоритм створення звітів на основі отриманих повідомлень.
- Розробити архітектуру та інтерфейс користувача.
- Впровадити програмні компоненти для централізованого збору даних.
- Провести тестування розробленого рішення.
- Створити детальну інструкцію користувача.

Об’єкт дослідження: Процеси прийому, оброблення та збереження інформаційних повідомлень.

Предмет дослідження: Технології та підходи до централізованого керування системами сповіщень.

Методи дослідження: У ході роботи застосовано методи аналізу інформаційних даних, порівняльний огляд наявних програмних рішень, моделювання за допомогою комп’ютерних засобів та експериментальне тестування функціоналу.

РОЗДІЛ 1. РОЗГЛЯД ПІДХОДІВ ДО РОЗРОБКИ ТА ФОРМУЛЮВАННЯ МЕТИ ДОСЛІДЖЕННЯ

1.1 Сучасні технології безсерверної хмарної обробки даних

У наш час створення та обслуговування програмних продуктів вимагає значно більших фінансових вкладень, ніж це було раніше. Це пояснюється не лише витратами на утримання офісів та оплату праці фахівців, але й зростанням вартості цифрової інфраструктури, необхідної як на етапі розробки, так і під час експлуатації продукту [6]. Зокрема, сучасні компанії змушені інвестувати в серверні потужності, необхідні для безперервної інтеграції та доставки (CI/CD), тестових середовищ, сховищ артефактів, а також для хостингу та обробки даних у процесі використання додатків.

Раніше більшість організацій покладалися на власні сервери або створювали власні дата-центри. Проте, з часом така модель стала менш доцільною [7]. Основні труднощі самостійного хостингу включають:

- необхідність постійної підтримки як «заліза», так і програмного забезпечення;
- високі початкові витрати на закупівлю та налаштування обладнання;
- потребу у кваліфікованих спеціалістах для обслуговування серверної інфраструктури;
- складності з масштабуванням – розширення можливе лише через нові закупівлі;
- складні умови ліцензування деяких програмних продуктів, залежно від апаратної конфігурації.

З розвитком технологій ці проблеми лише ускладнюються через збільшення технічних вимог до обладнання та програмного забезпечення. У відповідь на це виникли хмарні провайдери, які запропонували новий підхід до розв’язання вищезгаданих питань.

Серед основних переваг хмарних технологій виділяють [8]:

- Швидке розгортання сервісів – завдяки віртуалізації можливо створити повноцінний сервер менш ніж за хвилину.
- Зниження витрат – компанії платять лише за фактично використані ресурси, а не за всю інфраструктуру.
- Гнучке масштабування – легко адаптувати потужності до зростаючих потреб.
- Підвищена безпека – провайдери надають широкий набір засобів для захисту даних.
- Легка інтеграція з різними інструментами та платформами.
- Можливість створення гібридних інфраструктур – поєднання власних ресурсів із хмарними.
- Автоматичне оновлення сервісів без участі клієнта.

Крім того, перехід на хмарні рішення можна здійснити поетапно – спочатку перенести лише частину інфраструктури, а потім інтегрувати більш складні хмарні сервіси, оптимізуючи витрати та ресурси [9].

Особливе місце серед сучасних хмарних підходів займають безсерверні обчислення (serverless computing), які представляють собою новий етап у розвитку хмарних технологій. Цей підхід дозволяє повністю позбутися необхідності адміністрування серверів або ручного масштабування ресурсів [10]. До прикладів таких рішень належать сервіси AWS Lambda, Google Cloud Functions і Azure Functions. Вони надають можливість розміщувати й запускати код без втручання в управління інфраструктурою – усі технічні аспекти, включно з масштабуванням, бере на себе провайдер. Користувач сплачує лише за реальне використання обчислювальних потужностей та часу виконання.

Втім, розробка безсерверних додатків вимагає специфічного архітектурного підходу. Через відсутність збереження стану в межах самого обчислювального середовища, необхідно використовувати зовнішні сервіси для зберігання даних – наприклад, Amazon DynamoDB, що є масштабованим NoSQL-рішенням. Незважаючи на це обмеження, подібна архітектура має значну перевагу – вона забезпечує високу стійкість до навантажень та ефективне масштабування, що

робить її ідеальною для створення мікросервісних систем [11].

Загалом, тенденція активного переходу до хмарних технологій у сфері розробки ПЗ продовжує набирати обертів. Це дозволяє скорочувати витрати, підвищувати ефективність розробки та забезпечувати гнучкість інфраструктурних рішень. У цьому контексті безсерверні підходи стають важливим інструментом створення сучасних, адаптивних цифрових продуктів.

1.2 Аналіз функціональних аналогів систем сповіщення

У наш час ми постійно отримуємо купу повідомлень із різних додатків – месенджери, пошта, календарі, таск-менеджери, соцмережі, робочі сервіси, фітнес-трекери й так далі. Телефон може не замовкати весь день. І якщо раніше нам вистачало SMS чи звичайного дзвінка, то зараз усе набагато складніше. Через це голова часто йде обертом – бо інформації забагато, і мозок просто не встигає все обробити. Відключати всі сповіщення – теж не варіант, бо можна пропустити щось дійсно важливе. Саме тому й з’явилися додатки, які збирають усі ці повідомлення в одному місці.

Їхнє головне завдання – навести порядок у всьому цьому хаосі та зменшити навантаження на увагу. Тобто, замість того щоб постійно відволікатися на різні додатки, ти бачиш усе в одному зручному вікні – і можеш сам вирішити, що важливо, а що ні.

Серед найпопулярніших програм для такого збору сповіщень можна виділити:

- Franz – збирає пошту, месенджери, робочі додатки в одному місці. Працює на Windows і Mac.
- NTFY.sh – проста програма, яка пересилає всі сповіщення з різних пристроїв на один.
- Notistory – зручний варіант для мобільних, допомагає стежити за

повідомленнями без зайвого шуму.

Далі розглянемо, як саме працює кожен із цих додатків. Наприклад, Franz – це програма, яка дозволяє під'єднати кілька обліковок Google, пошту, Slack, Trello тощо, і перемикатись між ними в одному вікні. Виглядає це все досить зручно – інтерфейс простий і зрозумілий (див. рис. 1.1).

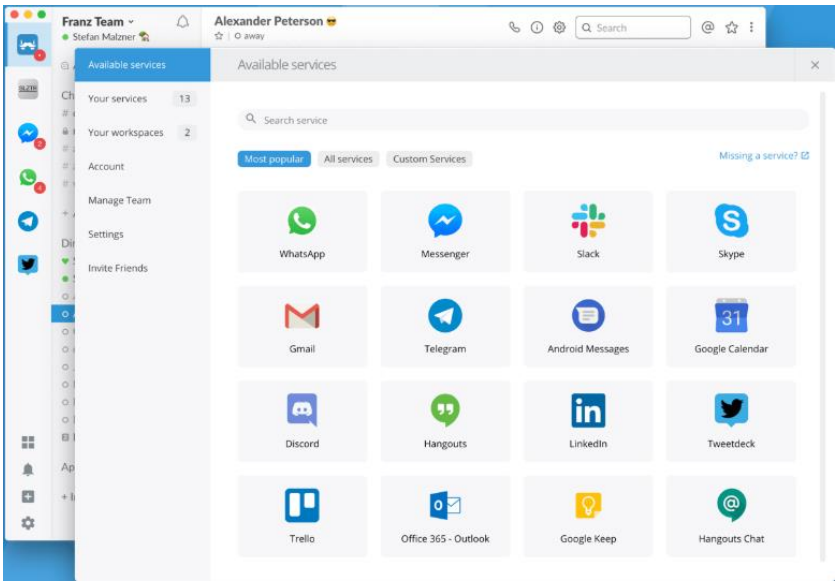

Рисунок 1.1 Приклад інтерфейсу програми Franz

Franz – це програмний інструмент, розроблений для централізованого керування робочими застосунками та ефективного отримання сповіщень, що в сукупності сприяє підвищенню продуктивності користувачів. Серед ключових переваг цього додатку варто виділити широкий набір попередньо налаштованих інтеграцій із різноманітними онлайн-сервісами, можливість одночасної роботи з кількома обліковими записами, а також доступ до сервісів безпосередньо через вбудований веб-інтерфейс. Водночас, програмне забезпечення має і деякі недоліки: обмежену сумісність із різними операційними системами та пристроями, а також невеликий набір функцій для гнучкого керування сповіщеннями.

NTFY.sh – це популярний мобільний додаток, розроблений компанією NTFY.sh LLC, який використовується для швидкого обміну повідомленнями в режимі реального часу. Його основне призначення – забезпечити зручний канал для

отримання миттєвих сповіщень. На рисунку 1.2 представлено приклад інтерфейсу цієї програми.

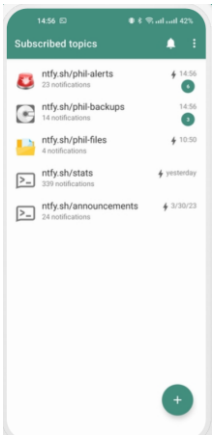

Рисунок 1.2 Зразок інтерфейсу застосунку NTFY.sh

Цей програмний продукт має інтуїтивно зрозумілий інтерфейс, який дає змогу користувачам швидко та зручно переглядати всі отримані повідомлення. Крім того, воно підтримує швидке створення власних інтеграцій завдяки відкритому API. Серед ключових переваг варто виокремити інтуїтивно зрозуміле управління, наявність вбудованих інструментів для командної роботи та розширений API, який значно полегшує розробку та супровід інтеграцій для сторонніх розробників [13]. Серед недоліків можна відзначити обмежену кількість попередньо реалізованих інтеграцій, а також фокусування переважно на користувачах мобільних пристроїв.

Notistory – це мобільна програма, розроблена компанією whowhoLab, яка забезпечує централізований доступ до вхідних сповіщень з різних джерел. Зразок графічного інтерфейсу цієї програми представлено на рисунку 1.3.

Рисунок 1.3 Приклад інтерфейсу програми Notistory

Notistory – це мобільний додаток, який виконує функції керування повідомленнями, що надходять на смартфон. Він замінює стандартну систему обробки сповіщень, вбудовану в операційну систему, розширюючи її можливості. Програма не взаємодіє зі сторонніми онлайн-сервісами чи додатками, які не встановлені на пристрої, та не отримує від них інформацію. Натомість вона забезпечує ефективний менеджмент переважної більшості системних повідомлень.

Серед ключових переваг Notistory – глибока інтеграція з мобільною операційною системою, а також можливість зберігати всі отримані сповіщення з подальшим зручним пошуком за історією. До недоліків можна віднести обмеженість підтримки: додаток працює лише з Android-платформами та їхніми похідними, а також не має функціоналу для роботи з зовнішніми онлайн-ресурсами або сервісами.

Порівняльний аналіз існуючих рішень у цій сфері дозволив виявити сильні та слабкі сторони щодо розроблюваного програмного забезпечення під назвою «Notification Collector». Зведені результати дослідження наведені в таблиці 1.1.

Таблиця 1.1 – Зведені результати дослідження

Критерій	Franz	NTFY.sh	Notistory	Notification Collector
Каталог готових інтеграцій	2	1	1	1
Створення власних інтеграцій	0	1	0	1
Підтримка десктопних ОС	1	0.5	0	1
Підтримка мобільних ОС	0	2	1	1
Генерація звітів на основі зібраних оповіщень	0	0	0	1
Підсумок	3	3.5	2	5

Наявність каталогу готових інтеграцій свідчить про перелік сервісів, з якими додаток може взаємодіяти одразу після інсталяції, не потребуючи складного налаштування. Серед розглянутих рішень лише Notistory отримує оцінку 0,5 бала, оскільки може взаємодіяти з системними повідомленнями операційної системи без необхідності додаткових дій з боку користувача.

Можливість створення індивідуальних інтеграцій означає підтримку підключення до сторонніх сервісів, які не входять до стандартного переліку. Наприклад, це може бути збір сповіщень із маловідомих веб-ресурсів. З аналізованих програм тільки NTFY.sh дозволяє реалізувати подібні інтеграції, однак для цього необхідні базові знання програмування та робота з API. Крім того, сервіс обмежений використанням лише push-архітектури та має ліміт на кількість API-запитів.

Сумісність з різними платформами вказує на те, на яких типах пристроїв може бути використане програмне забезпечення – зокрема, на десктопах чи мобільних пристроях. У цій категорії NTFY.sh отримує лише 0,5 бала, оскільки функціонує здебільшого через веб-інтерфейс, а не через повноцінні нативні застосунки. Функція формування звітів є важливою для подальшого аналізу отриманих повідомлень (див. додаток А.8). Хоча Notistory не має можливості створення звітів у прямому сенсі, вона забезпечує збереження історії сповіщень, а також дозволяє їх сортувати та здійснювати пошук. За ці функції система отримує 0,5 бала.

Проведене дослідження та порівняння наявних аналогів дало змогу виокремити як сильні, так і слабкі сторони існуючих рішень. Аналіз функціональності за ключовими критеріями підтверджує доцільність і актуальність

розробки власного програмного продукту, який би враховував недоліки представлених систем.

1.3 Огляд методів доставки повідомлень

Додатки, призначені для централізованого збору повідомлень, використовують різноманітні підходи до отримання вхідної інформації. Конкретні реалізації та рівень технічної складності таких рішень залежать від типу сервісу й ступеня відкритості доступу до нього. Найбільш поширеними є наступні способи:

- взаємодія із системними функціями операційної системи;
- застосування офіційних прикладних інтерфейсів (API) відповідних сервісів або застосунків;
- використання сторонніх інструментів для перехоплення повідомлень;
- автоматизований аналіз веб-ресурсів (парсинг);
- реалізація вебхуків;
- побудова власного механізму оповіщень на основі шаблону "Публікація–підписка".

Розглянемо особливості кожного методу більш докладно.

Інтеграція з операційною системою – це підхід, який застосовується в таких мобільних додатках, як Notistory, та інших подібних рішеннях. Його суть полягає в частковій або повній зміні стандартної логіки обробки повідомлень, яку реалізує ОС. Наприклад, додатки можуть замінювати вбудований менеджер сповіщень власним графічним інтерфейсом із розширеним функціоналом. Такий підхід створює додатковий програмний шар між системою й користувачем, завдяки чому обробка повідомлень відбувається після їх надходження до ОС. Однак функціональність таких рішень обмежена можливостями самої операційної системи.

На прикладі Windows можна помітити, що до версії 10 підтримка системи

сповіщень була недостатньо розвиненою, тому подібні рішення не набули широкого розповсюдження в середовищі десктопів. Натомість мобільні платформи, такі як Android, iOS та macOS, активно використовують системні механізми оповіщень, тому даний метод значно поширеніший саме серед мобільних додатків.

Основними недоліками цього підходу залишаються: обмеження доступу лише до повідомлень від встановлених застосунків, а також ризик дублювання даних (одночасне зберігання у системному сховищі та в самому додатку). Використання офіційного API – один із найзручніших і формально підтримуваних методів інтеграції зі сторонніми сервісами для отримання повідомлень. Його складність залежить від структури, функціональних можливостей та вимог обраного API. Головна перевага цього підходу – надійність та безпека взаємодії. Проте при потребі інтеграції з великою кількістю різних сервісів значно зростає обсяг програмного коду та збільшується навантаження як на серверну частину додатку, так і на ресурси користувацького пристрою. Кожен API має унікальні особливості, що ускладнює підтримку такої системи.

Перехоплення повідомлень сторонніми інструментами – це нетрадиційний спосіб збору інформації, який найчастіше використовується для сервісів, що не мають відкритих API або прямо забороняють сторонню інтеграцію згідно з власною політикою. Реалізація такого методу може бути юридично та етично спірною, а також технічно складною. Тому він рідко використовується в офіційних рішеннях і переважно зустрічається в неформальних або ентузіастських проєктах.

Парсинг веб-ресурсів – поширена альтернатива для отримання даних із платформ, які не пропонують API чи інші офіційні канали доступу до повідомлень. Наприклад, якщо новинний сайт не підтримує повідомлення або RSS, для отримання сповіщень про нові матеріали можна застосувати автоматизований парсинг HTML-коду. Такий підхід передбачає використання спеціалізованих бібліотек для вилучення структурованої інформації з веб-сторінок. Водночас підтримка великої кількості джерел ускладнюється через відмінності у структурі HTML-коду різних сайтів.

Вищезазначені методи базуються на pull-архітектурі, де ініціатива отримання даних належить клієнтському додатку. Протилежний підхід використовують push-механізми, до яких належать вебхуки та системи за моделлю "Публікація–підписка". У цих випадках дані надсилаються сервером автоматично, коли з'являється нова інформація, що дозволяє оперативніше реагувати на події та знижує навантаження на систему.

Для кращого розуміння відмінностей між цими архітектурними підходами можна звернутися до відповідної схеми взаємодії пристроїв, що ілюструє принципи їх роботи.

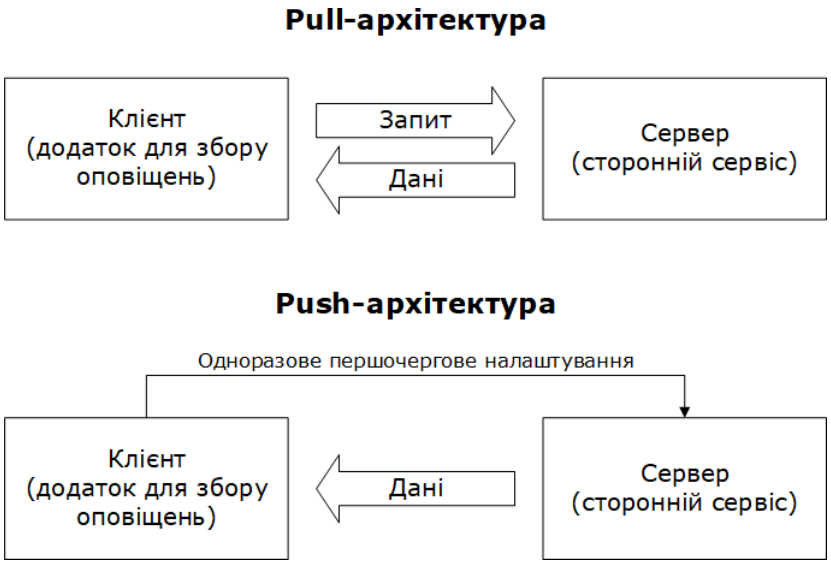

Рисунок 1.4 Взаємодія пристроїв при використанні push та pull архітектур

Підхід pull-архітектури полягає в тому, що клієнт надсилає запит до сервера, щоб отримати актуальні дані. У контексті збору сповіщень з зовнішніх сервісів, це означає, що додаток періодично звертається до API сервісу з метою перевірки наявності нової інформації. Оскільки заздалегідь неможливо знати, чи з’явилися нові повідомлення, запити виконуються регулярно з визначеним інтервалом. Такий підхід створює додаткове навантаження як на клієнтську сторону, так і на сервер, особливо коли нових даних немає, а запити все одно відбуваються.

Push-архітектура працює за іншим принципом: коли виникає певна подія, сервер сам ініціює передачу повідомлення до заздалегідь зареєстрованого клієнта.

Для цього необхідна попередня реєстрація клієнта на сервері, аби той знав, куди саме і яку інформацію надсилати. Один із найпоширеніших механізмів реалізації push-підходу – це вебхуки. Наприклад, додаток може за допомогою API створити вебхук, який буде реагувати на визначені події у сторонньому сервісі. Коли така подія відбудеться, сервер самостійно надішле HTTP-запит з відповідними даними до вказаного клієнта. Типовим прикладом є сервіс Gmail, який здатен надсилати запити на зовнішні адреси при надходженні нових листів. Це дозволяє реалізувати систему сповіщень без безпосереднього доступу до веб-інтерфейсу пошти. Основним недоліком такого підходу є те, що кожен сервіс реалізує вебхуки по-різному, що ускладнює їхню уніфіковану інтеграцію.

Ще один, складніший у реалізації, варіант – створення власного сервісу сповіщень, який працює за принципом «публікація-підписка». Такий підхід передбачає, що розробник створює API для приймання повідомлень, яким користуються сторонні сервіси, надсилаючи сповіщення за потреби. На стороні додатку залишається вся логіка обробки отриманих повідомлень. Цей варіант нагадує роботу вебхуків, але з тією різницею, що ініціатива підключення до API належить не зовнішнім сервісам, а самому додатку. Так, наприклад, функціонує сервіс NTFY.sh, який надає набір бібліотек на різних мовах програмування для інтеграції з різноманітними платформами. Основна складність полягає у тому, що сторонні сервіси повинні спеціально впроваджувати підтримку такого API, що не завжди можливо, особливо якщо мова йде про маловідомі або нішеві рішення.

У результаті було проаналізовано кілька підходів до взаємодії зі сторонніми сервісами для отримання сповіщень. Визначено їхні технічні особливості, переваги та недоліки. Для розробки власного додатку прийнято рішення використати комбінований підхід: поєднання вебхуків, API взаємодії з підтримуваними сервісами та парсингу веб-сторінок для менш структурованих ресурсів. Така стратегія дозволить забезпечити високу гнучкість і ефективність, а також розширити можливості інтеграції з різними зовнішніми системами.

1.4 Формулювання цілей і завдань дослідження

На основі аналізу існуючих програмних рішень для збору оповіщень, а також дослідження технічних аспектів реалізації та сучасного стану хмарних сервісів було сформульовано перелік задач, необхідних для створення власного програмного продукту:

- здійснити огляд і аналіз сучасних хмарних технологій, зокрема безсерверних обчислень;
- вивчити актуальні методи отримання сповіщень із зовнішніх джерел;
- розробити методику та алгоритм інтеграції з джерелами оповіщень, що підтримують вебхуки та відкриті API;
- реалізувати механізм і алгоритм збору сповіщень шляхом парсингу інформаційних веб-ресурсів;
- створити алгоритм генерації звітів на базі зібраної інформації;
- спроєктувати архітектуру та користувацький інтерфейс програмного забезпечення;
- розробити модулі, відповідальні за централізований збір повідомлень;
- реалізувати модульне тестування з метою забезпечення стабільності та коректності роботи додатку;
- виконати ручне тестування функціоналу;
- підготувати повноцінну інструкцію з користування програмним продуктом.

Технічне завдання представлено в додатку А.

Функціональні вимоги:

- можливість додавання джерел сповіщень, які підтримують інтеграцію через вебхуки;
- підтримка джерел, що потребують веб-скрейпінгу (парсингу) для отримання даних;

- відображення всіх зібраних повідомлень у зручному для користувача вигляді;
- можливість створення звітів на основі отриманих оповіщень.

Нефункціональні вимоги:

- сумісність з такими операційними системами, як Windows 10, macOS Monterey, Ubuntu 20.04;
- використання клієнт-серверної архітектури з взаємодією через RESTful API;
- забезпечення передачі даних виключно через захищений протокол HTTPS.

РОЗДІЛ 2. РОЗРОБЛЕННЯ МЕТОДІВ ЗБОРУ ТА ОПРАЦЮВАННЯ ПОВІДОМЛЕНЬ

2.1 Створення алгоритмів отримання сповіщень через вебхуки та API-сервіси

У підрозділі 1.3 було проаналізовано сучасні підходи до отримання сповіщень від зовнішніх сервісів. Одним з ефективних рішень у цій сфері є застосування офіційно підтримуваних інструментів збору даних, зокрема таких як вебхуки та відкриті API-інтерфейси.

Запропонований підхід базується на інтеграції з сервісами, які можуть передавати актуальну інформацію у вигляді сповіщень зовнішнім системам. У випадку вебхуків, ці сервіси надсилають HTTP-запити, зазвичай у форматі JSON, до заздалегідь визначеної адреси. Відповідно, серверна логіка створюваного додатку повинна мати публічну URL-адресу, яка прийматиме ці повідомлення та оброблятиме отримані дані.

На відміну від традиційних методів, у цьому варіанті використовується єдина точка доступу (URL) для прийому всіх повідомлень. Для обробки даних застосовується набір заздалегідь встановлених правил, що дозволяє привести повідомлення до уніфікованого формату, незалежно від того, з якого джерела вони надійшли. Це важливо, оскільки різні сервіси можуть мати відмінну структуру повідомлень.

Сповіщення у форматі JSON являють собою об’єкти типу "ключ-значення", які можуть містити вкладені рівні. Тому для опису правил обробки використовується формат, що визначає шлях до потрібного ключа в структурі об'єкта. Такий підхід забезпечує ефективну уніфікацію, збереження та подальше використання даних.

На основі запропонованого способу була розроблена блок-схема алгоритму, що зображена на рисунку 2.1. Далі розглянемо поетапно, як працює запропонований алгоритм.

Крок 1. Ініціалізація процесу.

Крок 2. AWS Lambda функція активується через виклик API Gateway, отримуючи вхідні параметри event і context, які містять усю необхідну інформацію щодо сповіщення (через вебхук чи API), а також дані про середовище виконання.

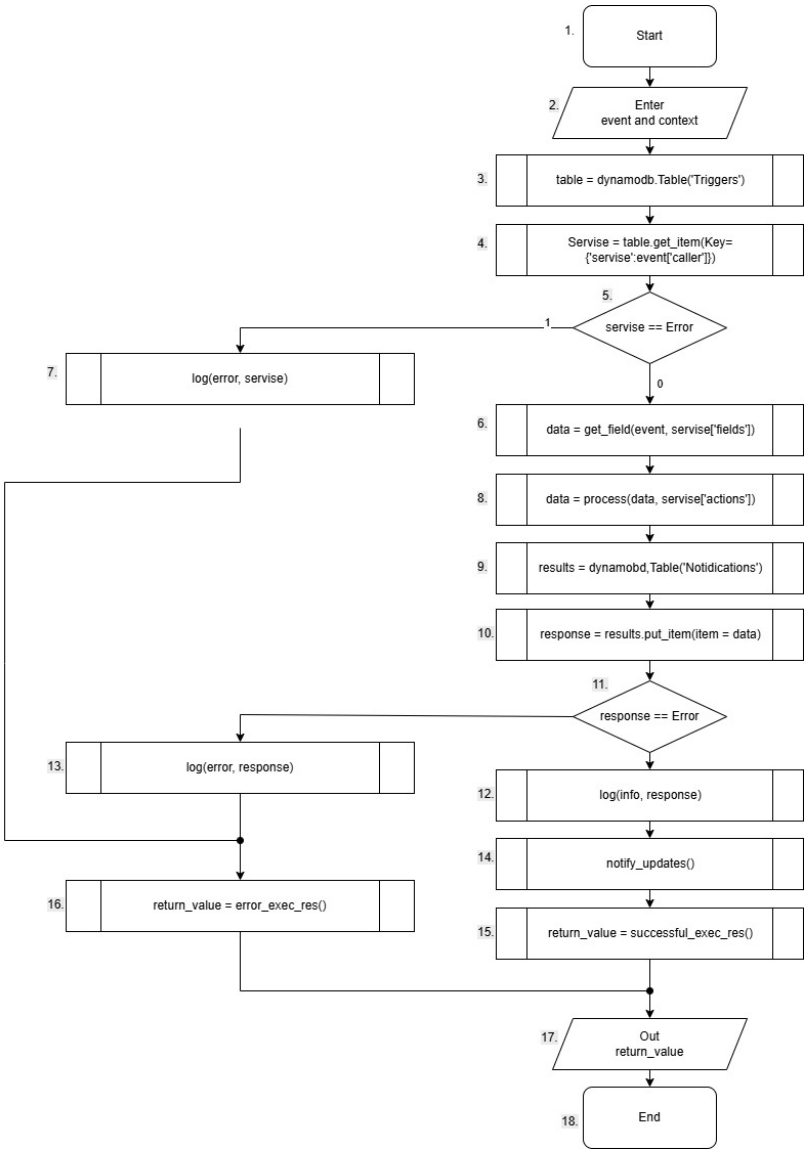

Рисунок 2.1 Схематичне зображення процесу отримання повідомлень через вебхуки з використанням відкритих API-інтерфейсів

Крок 3. Ініціалізується екземпляр об’єкта, що забезпечує доступ до таблиці бази даних, у якій зберігаються відомості про доступні сторонні сервіси та специфіку взаємодії з кожним із них.

Крок 4. Здійснюється запит до таблиці з метою отримання інформації саме про той зовнішній сервіс, від якого надійшло поточне оповіщення.

Крок 5. У разі помилки під час виконання запиту (наприклад, відсутність запису або проблема з підключенням), система негайно переходить до обробки виняткової ситуації – кроку 7. Якщо запит успішний, виконується наступна фаза (крок 6).

Крок 6. За допомогою функції get_fields виконується витягнення необхідних параметрів із змінної event, яка містить сирі дані оповіщення. Оскільки структура подій залежить від конкретного сервісу, для обробки потрібно мати перелік необхідних полів, заздалегідь визначений у відповідному записі бази даних.

Крок 7. Усі помилки, що виникли на попередніх етапах, заносяться в лог-систему для подальшого аналізу. Після цього виконується перехід до завершального кроку – кроку 16.

Крок 8. Якщо обробка даних потребує додаткових дій (наприклад, трансформації, фільтрації, очищення), викликається функція process. Вона виконує лише ті операції, які визначені у конфігурації для поточного сервісу.

Крок 9. Ініціалізується об’єкт для взаємодії з окремою таблицею, у якій зберігаються уже оброблені повідомлення – готові для подальшого надсилання користувачу.

Крок 10. Збереження підготовлених даних здійснюється у відповідну таблицю. На цьому етапі формується структурована запис, який далі буде використано для інтерфейсного відображення або статистики.

Крок 11. Якщо спроба запису виявилася невдалою, система передає керування кроку 13. Інакше – переходить до наступної фази.

Крок 12. У разі успішної обробки події в лог записується інформація про факт отримання та обробки вебхука.

Крок 13. Якщо ж під час запису виникла помилка, відповідне повідомлення також фіксується в логах, після чого знову активується крок 16.

Крок 14. Викликається функція notify_update, яка сигналізує про завершення обробки події. Після цього система може ініціювати додаткові дії в середовищі AWS Lambda, наприклад – виклик нової функції для надсилання даних у

клієнтський інтерфейс або акумулювання інформації для створення звітів. Конкретний сценарій залежить від налаштувань користувача.

Крок 15. У змінну return_value записується результат – тобто дані про те, що обробка оповіщення пройшла успішно.

Крок 16. У разі виникнення помилки ця ж змінна return_value містить опис проблеми, а також інформацію про те, на якому етапі вона відбулася.

Крок 17. Повертається результат функції (вміст return_value) до AWS Lambda, що сигналізує про завершення виконання.

Крок 18. Завершення алгоритму.

2.2. Реалізація методу парсингу веб-ресурсів для отримання повідомлень

Альтернативним способом формування повідомлень є застосування технологій веб-парсингу. У технічному розумінні це не процес отримання сповіщень із заздалегідь підключених сервісів, а їх створення на основі зовнішніх джерел. Йдеться про автоматизований збір даних з інтернет-ресурсів, їх подальший аналіз, зіставлення з поточними параметрами та передачу результатів до клієнтської частини системи.

Ключовим компонентом такого підходу виступає вебскрапінг – процес, що перетворює інформацію зі сторінок, призначених для візуального сприйняття людиною, у структуровані масиви даних. Цей метод базується не на використанні готових API, а на індивідуально визначених шаблонах та правилах обробки конкретного веб-сайту. Стандартний алгоритм роботи складається з трьох основних кроків: завантаження веб-сторінки, вибір цільових елементів згідно з критеріями, а також обробка та збереження зібраної інформації у вигляді повідомлення.

Найчастіше для вилучення потрібних даних використовується аналіз HTML-структури сторінки, де для точного доступу до інформаційних блоків

застосовуються CSS-селектори. Вони дозволяють локалізувати потрібні елементи (заголовки, дати, опис тощо) та виділити з них зміст, необхідний для подальшого використання. Після обробки контенту формується повідомлення, яке може бути передано користувачу або збережене у системі для подальшої обробки.

Таким чином, зазначений підхід забезпечує гнучкий збір повідомлень із будь-яких відкритих веб-ресурсів, дозволяючи користувачеві самостійно задавати логіку їх фільтрації та формування. Результатом є ефективне формування сповіщень на основі динамічного контенту з мережі Інтернет. Структурна схема реалізації алгоритму подано на рисунку 2.2 у вигляді блок-схеми.

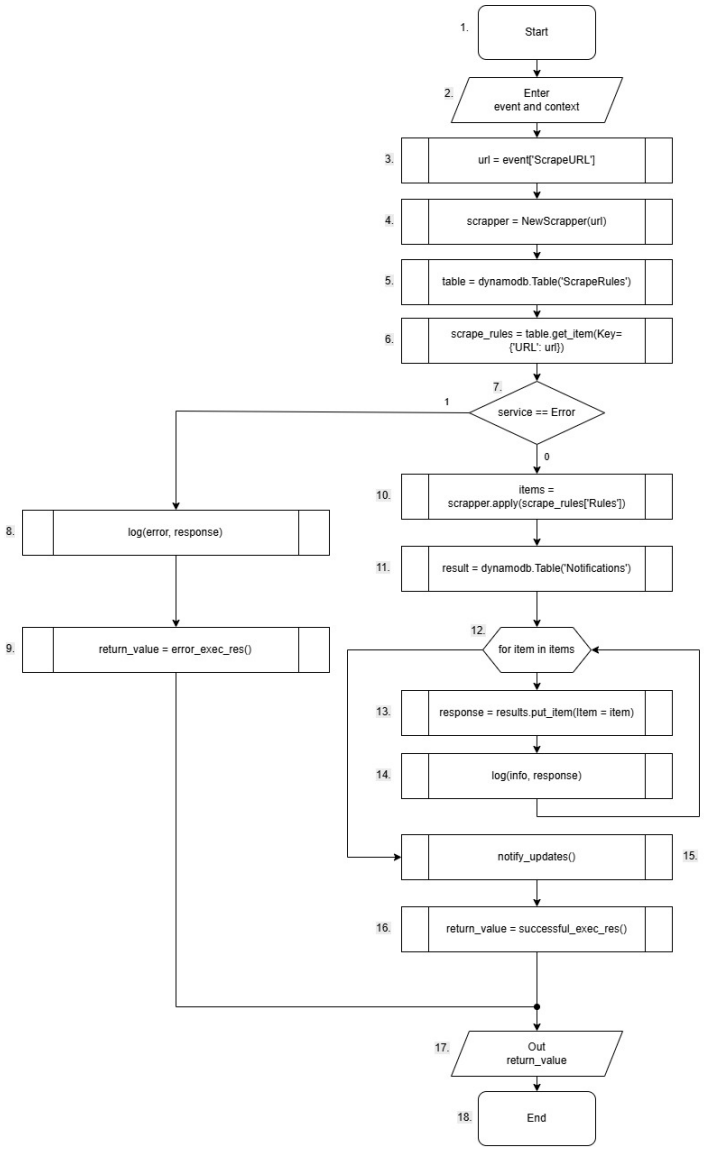

Рисунок 2.2 Структурна схема реалізації алгоритму подано на рисунку 2.2

У якості прикладу практичного застосування розробленого підходу можна розглянути реалізацію автоматизованої системи інформування про появу нових матеріалів на улюблених вебресурсах користувача. Такий функціонал може бути реалізований за рахунок регулярного аналізу вмісту конкретної сторінки, що містить перелік публікацій. Достатньо вказати адресу вебсторінки (URL) та налаштувати набір інструкцій, які дадуть змогу системі правильно розпізнати структуру HTML-коду і виокремити необхідну інформацію у зручному вигляді.

У разі періодичного запуску такого процесу та порівняння результатів з попередніми сесіями можна ефективно фіксувати появу нових записів та інформувати користувача. Такий механізм дає змогу адаптувати рішення під різні сайти й автоматизувати оновлення контенту.

Алгоритм дії системи сповіщення умовно виглядає так:

1. Запускається основна логіка обробки запиту.

2. Зчитуються параметри event та context, що автоматично передаються у середовищі AWS Lambda під час виклику функції.

3. Із вхідних параметрів визначається цільова URL-адреса для збору контенту.

4. Ініціалізується парсер, який одразу отримує HTML-код зазначеної сторінки.

5. Встановлюється з'єднання з базою даних, у якій зберігаються шаблони обробки HTML-структур.

6. Визначаються правила парсингу, пов’язані з відповідною адресою.

7. У разі невдачі отримання шаблону виконується обробка помилки.

8. Інформація про помилку записується до журналу подій (лог-файлу).

9. У змінну return_value записується статус невдалого завершення.

10. Якщо правила отримано успішно – виконується парсинг відповідно до шаблону; результати записуються в масив items.

11. Ініціалізується об’єкт, що забезпечує взаємодію з базою даних, де зберігаються готові повідомлення для користувача.

12. Починається обробка кожного запису з масиву items – кожен елемент є окремою новиною або публікацією.

13. Кожна одиниця контенту зберігається в таблиці бази даних.

14. Результат запису фіксується; навіть помилки логуються, але не блокують процес – це дозволяє працювати з великим обсягом інформації.

15. Після завершення операції викликається функція notify_update, яка може передати повідомлення клієнтському інтерфейсу або іншому модулю.

16. У return_value фіксується повідомлення про успішне виконання.

17. Змінна return_value виводиться як результат виконання.

18. Процес завершується.

У підсумку представлений алгоритм дозволяє реалізувати гнучку систему для автоматичного відстеження змін на вебресурсах та формування інформувальних повідомлень. Завдяки використанню бази даних і шаблонів парсингу, така система легко адаптується до специфіки будь-якого сайту. Вона може бути масштабована і інтегрована у більші цифрові рішення, зокрема в системи моніторингу новин, блогів, технічної документації або цінових змін.

2.3 Розробка системи формування звітів на базі зібраної інформації

Користувачам часто потрібно не просто збирати сповіщення з підключених сервісів і ресурсів, але й мати змогу їх ефективно агрегувати та аналізувати. З цією метою доцільно винести функцію створення аналітичного звіту в окрему складову продукту. Такий звіт дозволяє структурувати потік вхідної інформації, що надходить щодня, та забезпечує краще розуміння її змісту і значущості.

Запропоноване рішення розширює базовий підхід до формування звіту, передбачаючи декілька форматів подання інформації. Воно реалізується у два ключові етапи: первинна обробка сповіщень і візуалізація результатів. На першому етапі відбувається вибірка цільових повідомлень, їх категоризація за джерелами

(назвами сервісів), а також підрахунок кількості повідомлень у кожній категорії. Другий етап включає три основні складові:

- графічне зображення у вигляді діаграми, що демонструє розподіл сповіщень за джерелами;
- стислий текстовий аналіз із згадуванням найчастіших сервісів;
- деталізований перелік оброблених сповіщень, який дозволяє користувачу самостійно провести глибший аналіз.

Для реалізації цієї функціональності створено спеціальний алгоритм, що схематично зображений на рисунку 2.4.

Розглянемо поетапно виконання цього алгоритму:

1. Початок обробки.

2. Ініціалізація даних. Вхідна інформація подається у форматі AWS Lambda через параметри event і context. Вони містять усі необхідні налаштування для запуску процесу.

3. Отримання фільтрів. Із event витягується набір параметрів фільтрації.

4. Зчитування обмеження на кількість сповіщень.

5. Створення об'єкта доступу до таблиці сповіщень.

6. Виконання запиту до бази. Для формування вибірки використовуються надані фільтри та ліміт.

7. Перевірка результату. Якщо повідомлень не знайдено, виконується перехід до завершального етапу (крок 17).

8. Ініціалізація структури для підрахунку. Створюється об’єкт для підрахунку кількості сповіщень по кожному сервісу.

9. Цикл обробки повідомлень. Перебираються всі записи з результатів запиту.

10. Виділення назви сервісу. Для кожного повідомлення за допомогою функції getService визначається його джерело.

11. Оновлення лічильника. Отримана назва сервісу додається до підсумкової структури.

12. Генерація діаграми. На основі накопичених даних будується графічна

візуалізація.

Такий підхід дозволяє не лише здійснювати моніторинг повідомлень, а й отримувати структурований звіт із наочним відображенням ключових джерел інформації. Це значно підвищує зручність роботи з великою кількістю сповіщень та дозволяє приймати більш обґрунтовані рішення.

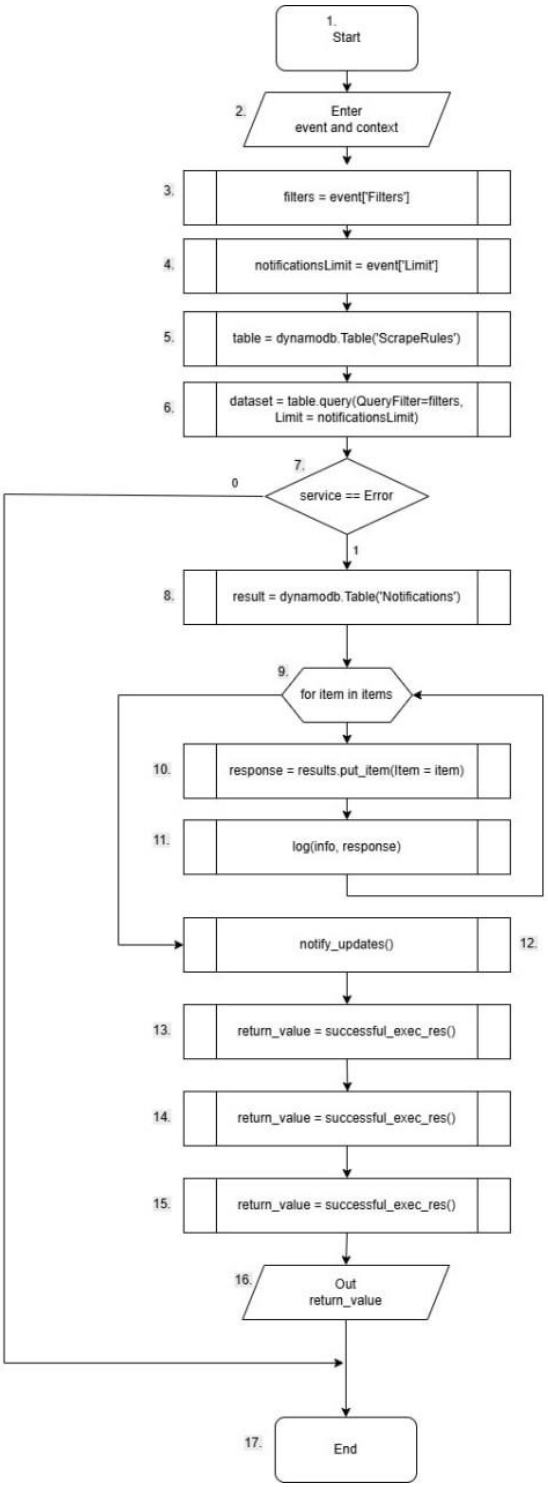

Рисунок 2.4 Схема логіки формування звітів із отриманих повідомлень

Крок 13. Для подальшої обробки створено байтовий буфер, який слугує тимчасовим сховищем для сформованої діаграми.

Крок 14. У зазначений буфер записується графік, що відображає статистику за отриманими сповіщеннями.

Крок 15. Окрема змінна використовується для збереження даних про десять сервісів, які мають найвищий показник кількості повідомлень.

Крок 16. На цьому етапі здійснюється виведення:

- зображення діаграми з буфера,
- списку сервісів із найбільшою кількістю сповіщень,
- а також повного обсягу зібраних даних, що можуть бути корисними для глибшого аналізу в інтерфейсі клієнтської програми.

Крок 17. Завершення роботи алгоритму.

РОЗДІЛ 3. ПРОЄКТУВАННЯ СИСТЕМИ ТА КЛЮЧОВИХ КОМПОНЕНТІВ ПРОГРАМИ

3.1 Аналіз можливих мов програмування та обґрунтування вибору

Вибір мови програмування для хмарних застосунків: особливості, переваги та порівняння.

Під час створення програмного забезпечення, яке інтегрується з хмарною інфраструктурою, вибір мови програмування відіграє важливу роль. Цей вибір безпосередньо впливає як на продуктивність розроблюваного рішення, так і на витрати, пов’язані з експлуатацією хмарних сервісів. Провайдери хмарних технологій, як правило, надають гнучкість у цьому питанні та підтримують широке коло мов програмування. Наприклад, офіційний SDK Amazon Web Services (AWS) надає підтримку для C++, Go, Java, JavaScript, C#, Node.js, PHP, Python і Ruby.

Деякі сервіси, зокрема AWS Lambda, дозволяють запускати код, написаний на таких мовах, як C#, Java, Go, Python та інших. Якщо стандартні середовища виконання не відповідають вимогам розробника, існує можливість створення кастомізованого середовища виконання для Lambda. Такі сервіси, як Amazon EC2, ECS або EKS, забезпечують обчислювальні ресурси для програм будь-якої складності – обмеженням можуть бути лише технічні характеристики обладнання. Таким чином, сучасні хмарні рішення не накладають суворих обмежень щодо вибору мов програмування.

З метою визначення оптимальної мови для реалізації централізованого сервісу збору оповіщень було здійснено аналіз чотирьох популярних мов: Java, C#, JavaScript та Python.

Java – це мова, створена компанією *Sun Microsystems* у 1995 році, а з 2009 року її підтримкою займається *Oracle*. Вона є об’єктно-орієнтованою мовою з синтаксисом, подібним до C/C++. Ключовою особливістю Java є її платформна незалежність – програми компілюються у байт-код, який може виконуватись на будь-якому пристрої за допомогою JVM (Java Virtual Machine). Цей підхід спрощує

розгортання програмного забезпечення на різних операційних системах.

Основні переваги Java:

- повна підтримка об’єктно-орієнтованого підходу;
- зрозумілий синтаксис для розробників, знайомих із мовами сімейства C;
- широке застосування у сфері серверних і корпоративних рішень;
- підвищена безпека завдяки роботі в середовищі JVM;
- кросплатформеність.

Мова C# була представлена компанією *Microsoft* у 2000 році й стала частиною Visual Studio з 2002 року. Вона має спільні риси з Java, однак з часом сформувала власний напрямок розвитку. Найбільшу популярність C# здобула у розробці програм для Windows, завдяки підтримці .NET Framework. Також існують рішення на основі .NET Core та Mono, які забезпечують кросплатформену розробку. Програми на C# компілюються у проміжний код IL (Intermediate Language), що потім виконується в середовищі CLR (Common Language Runtime).

Особливості C#:

- високий рівень абстракції та управління пам’яттю;
- чітка орієнтація на об’єктно-орієнтовану модель;
- глибока інтеграція з платформою .NET;
- знайомий синтаксис та зручність використання;
- підтримка з боку великої спільноти розробників.

JavaScript з’явився у 1995 році та був стандартизований у 1997 під назвою ECMAScript. Спочатку мова використовувалась виключно для клієнтської розробки у веб-браузерах, однак із появою Node.js з’явилась можливість її застосування на сервері. Це суттєво розширило її сферу використання, зробивши JavaScript універсальним інструментом для створення як клієнтських, так і серверних додатків.

Переваги JavaScript:

- інтерпретована мова з можливістю негайного виконання коду;
- величезна кількість доступних бібліотек та фреймворків;

- можливість реалізації повноцінних Full-Stack рішень;
- активна спільнота та постійний розвиток мови.

Python бере свій початок з кінця 1980-х років, а його перша офіційна версія з’явилась у 1994 році. На відміну від JavaScript, Python одразу проєктувався як мова загального призначення. Вона отримала особливу популярність завдяки простому синтаксису, своїм потужним можливостям, система ідеально підходить для аналізу даних, машинного навчання та наукових досліджень. Python є інтерпретованою мовою, а її гнучкість дозволяє застосовувати її на багатьох платформах.

Ключові переваги Python:

- надзвичайна простота синтаксису та легкість у вивченні;
- велика кількість альтернативних середовищ виконання;
- потужна бібліотечна підтримка;
- сумісність з іншими мовами програмування та системами.

Порівняльний аналі. Для об'єктивного вибору мови розробки було проведено порівняння згаданих технологій за низкою технічних критеріїв. Результати цього аналізу представлено в таблиці 3.1 (див. далі).

Таблиця 3.1 – порівняння мов програмування

Критерій	Java	C#	JavaScript	Python
Підтримка ООП	1	1	0.5	1
Простота синтаксису	0	0	0.5	1
Вбудована система на AWS Lambda	1	1	1	1
Бібліотека для кросплатформної розробки	1	0.5	1	1
Широкий вибір IDE та інструментів	1	0.5	0.5	1
Підсумок	4	3	3.5	5

Вибір мови програмування для розробки програми централізованого збору сповіщень був здійснений на основі порівняльного аналізу. В результаті дослідження, мова Python виявилася найбільш відповідним варіантом.

У процесі аналізу були розглянуті та оцінені такі мови програмування, як Java, C#, JavaScript та Python. Кожна мова була проаналізована за рядом критеріїв, результати яких були систематизовані у вигляді таблиці (не наведено в

даному тексті, але передбачається). На основі проведеного порівняння та врахування всіх факторів, Python було обрано як оптимальну мову щоб розробляти програмне забезпечення.

3.2 Вибір інструментального середовища для розробки

Інтегроване середовище розробки (IDE) – це спеціалізоване програмне забезпечення, створене для полегшення та об’єднання різних етапів процесу програмування. Такий інструмент забезпечує розробників усім необхідним у межах одного інтерфейсу, що сприяє підвищенню ефективності їхньої роботи.

Скорочення IDE походить від англійського терміна Integrated Development Environment, що в перекладі означає «інтегроване середовище розробки». Основною метою таких середовищ є забезпечення комфортних умов для написання, тестування та відлагодження програмного коду завдяки об’єднанню ключових інструментів розробника в єдиний інтерфейс.

Як правило, більшість IDE включають наступні компоненти:

- Редактор програмного коду – на відміну від звичайних текстових редакторів, IDE надає підтримку підсвічування синтаксису та функції автодоповнення, що значно прискорює написання коду;
- Інтеграція з компілятором або інтерпретатором – середовище самостійно виконує трансляцію або інтерпретацію коду, без потреби вводити окремі команди компіляції. У багатьох випадках також надається зручний інтерфейс для налаштування параметрів збірки;
- Вбудований налагоджувач – більшість сучасних IDE дозволяють не лише запускати код, але й аналізувати його виконання в режимі налагодження, що допомагає швидше знаходити та виправляти помилки.

Популярність Python зумовила появу багатьох потужних середовищ розробки. Серед них виділяються Spyder, PyDev та PyCharm. Spyder –

безкоштовне, відкрите IDE з активною спільнотою, орієнтоване на науковців та аналітиків даних, але також корисне для загального програмування завдяки широким можливостям тестування та налагодження. Підтримка плагінів, зокрема інтеграція з Jupyter Notebook, робить його зручним як для новачків, так і для досвідчених розробників.

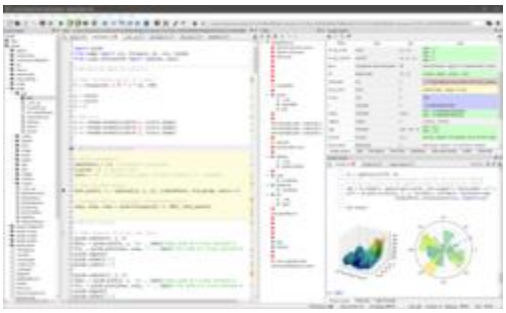

Рисунок 3.1 Представлено візуальний приклад інтерфейсу Spyder.

PyDev, що інтегрується з Eclipse, перетворює цю платформу на потужне середовище розробки (IDE) для Python. Eclipse, як модульна платформа, отримує від PyDev розширені інструменти, спеціально розроблені для ефективної роботи з Python. Це поєднання робить Eclipse повноцінною IDE для розробки на Python.

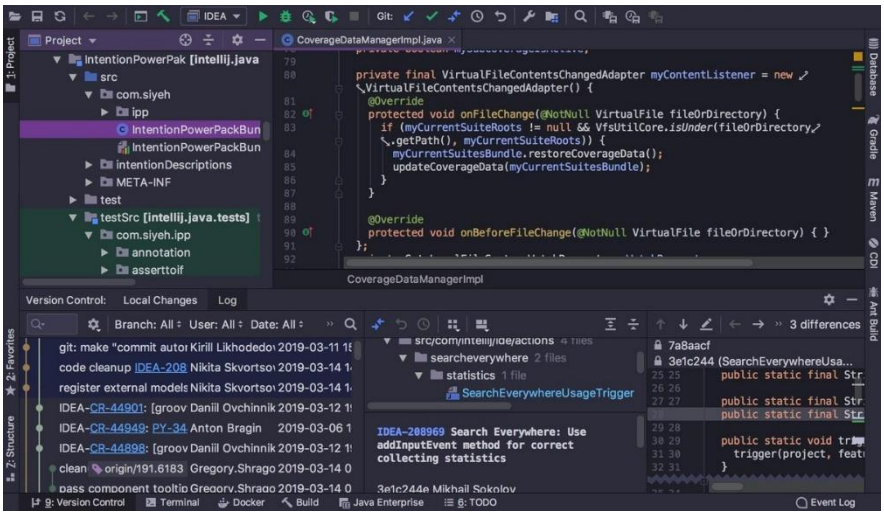

Рисунок 3.2 Середовище розробки Eclipse

Інтегроване середовище розробки (IDE) – це спеціалізоване програмне забезпечення, створене для полегшення та об’єднання різних етапів процесу

програмування. Такий інструмент забезпечує розробників усім необхідним у межах одного інтерфейсу, що сприяє підвищенню ефективності їхньої роботи.

Скорочення IDE походить від англійського терміна *Integrated Development Environment*, що в перекладі означає «інтегроване середовище розробки». Основною метою таких середовищ є забезпечення комфортних умов для написання, тестування та відлагодження програмного коду завдяки об’єднанню ключових інструментів розробника в єдиний інтерфейс.

Як правило, більшість IDE включають наступні компоненти:

- Редактор програмного коду – на відміну від звичайних текстових редакторів, IDE надає підтримку підсвічування синтаксису та функції автодоповнення, що значно прискорює написання коду.
- Інтеграція з компілятором або інтерпретатором – середовище самостійно виконує трансляцію або інтерпретацію коду, без потреби вводити окремі команди компіляції. У багатьох випадках також надається зручний інтерфейс для налаштування параметрів збірки.
- Вбудований налагоджувач – більшість сучасних IDE дозволяють не лише запускати код, але й аналізувати його виконання в режимі налагодження, що допомагає швидше знаходити та виправляти помилки.

Оскільки Python є однією з найпопулярніших мов програмування у світі, існує багато потужних середовищ, що підтримують її. Найбільш поширеними з них є Spyder, PyDev та PyCharm.

Spyder – це безкоштовне та відкриття IDE, яке активно підтримується спільнотою розробників. Воно побудоване з використанням Python і орієнтоване переважно на потреби науковців, аналітиків даних та інженерів, які використовують цю мову для обробки та аналізу великих обсягів інформації. Незважаючи на свою наукову спрямованість, Spyder також підходить для розв’язання загальних програмних задач завдяки широкому функціоналу для тестування та налагодження.

Крім того, це середовище підтримує систему плагінів, що дозволяє користувачам розширювати його можливості. Зокрема, є інтеграція з Jupyter

Notebook – потужним інструментом для візуалізації результатів обчислень і створення інтерактивних звітів. Така гнучкість робить Spyder зручним засобом як для початківців, так і для досвідчених програмістів.

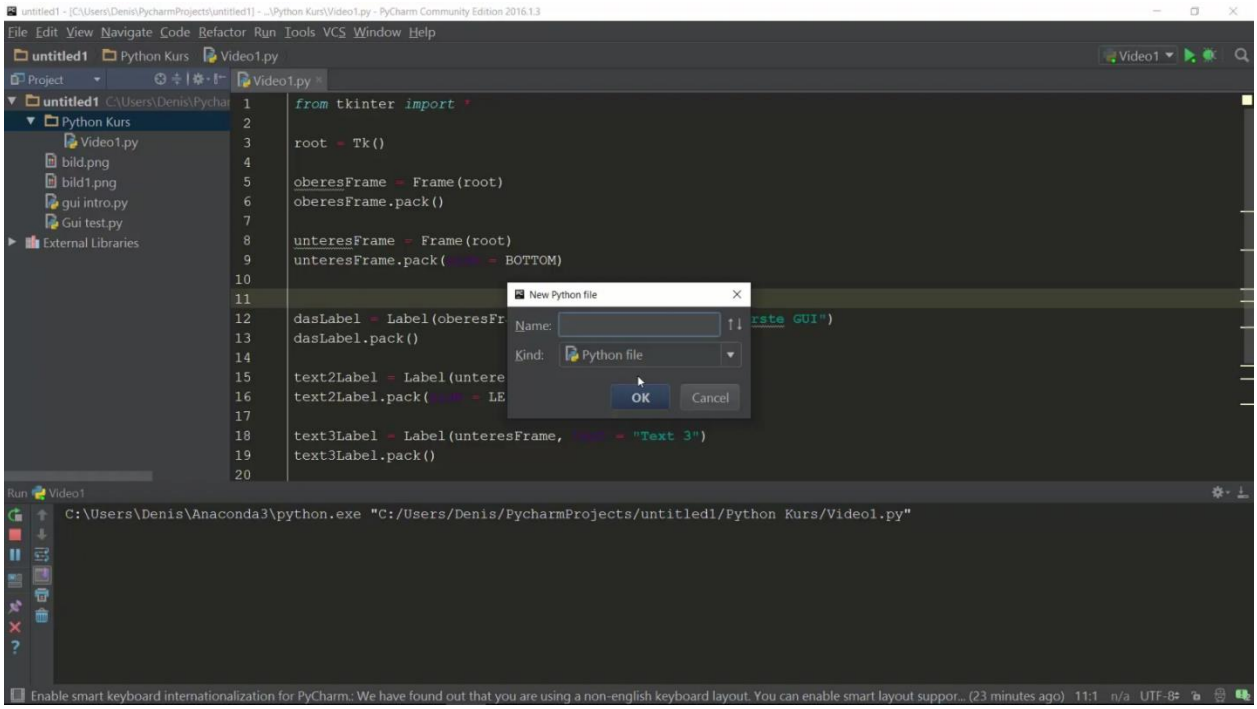

Рисунок 3.3 Приклад графічного інтерфейсу PyCharm

Кожне з розглянутих середовищ розробки (IDE) має свої сильні та слабкі сторони. Одні IDE спеціалізуються на підтримці конкретних прикладних областей, оптимізовані для роботи з певними фреймворками та інструментами. Інші пропонують більш універсальний набір функцій, придатний для розробки традиційних десктопних додатків. Для обґрунтованого вибору оптимального інструменту для вирішення конкретних завдань було проведено порівняльний аналіз на основі визначених критеріїв. Результати цього аналізу, а також перелік використаних критеріїв, представлені в таблиці 3.2.

Таблиця 3.2 – Результати порівняльного аналізу

Критерій	Spyder	PyDev	PyCharm
Універсальність функціоналу	1	0.5	1
Кросплатформність	1	1	1
Швидкодія та використання ресурсів системи	1	2	0.5
Інтеграція з БД та популярними флеймворками	1	0.5	1
Інструменти рефакторингу	1	0.5	1
Підсумок	5	4.5	4.5

На основі проведеного порівняльного аналізу, PyCharm виявилася найбільш підходящим інтегрованим середовищем розробки для реалізації програмного забезпечення відповідно до поставлених вимог та завдань.

Підсумовуючи, в цьому розділі було представлено огляд основних функцій IDE. Ми розглянули детальні характеристики Spyder, PyDev та PyCharm, супроводжуючи їх прикладами інтерфейсу. Після всебічного порівняння різних середовищ, було вирішено використовувати PyCharm від JetBrains для розробки програми централізованого збору сповіщень.

3.3 Побудова архітектури програмного забезпечення

Розробка архітектури програмного забезпечення, призначеного для хмарного середовища та використання безсерверних обчислень, вимагає ретельного вивчення всіх елементів системи [24]. Це необхідно для чіткого розуміння задіяних хмарних сервісів та для прогнозування витрат на інфраструктуру.

Для програмного продукту, призначеного для централізованого збору сповіщень, було розроблено архітектурну схему, представлену на рисунку 3.4.

Згідно зі схемою, система складається з двох основних компонентів: клієнтського та серверного. Клієнтська частина може бути представлена у вигляді додатку для комп'ютера, мобільного телефону або навіть у вигляді чат-бота, що звільняє кінцевого користувача від необхідності встановлювати додаткове

програмне забезпечення. У рамках даної бакалаврської дипломної роботи, клієнтська частина представлена у вигляді програми для десктопних платформ. Серверна частина системи повністю побудована на хмарних сервісах, використовуючи безсерверну архітектуру. Розглянемо ці компоненти більш детально.

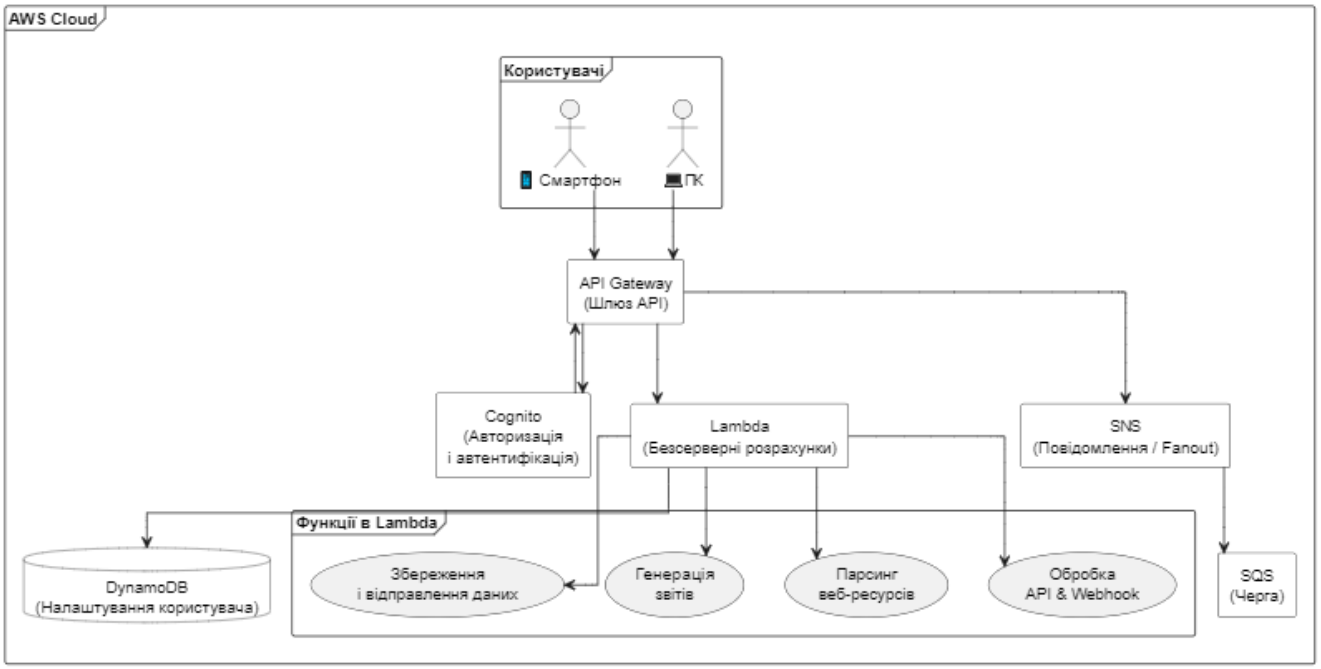

Рисунок 3.4 Схема архітектури продукту

API Gateway – це ключовий сервіс для створення, управління, моніторингу та захисту прикладних програмних інтерфейсів (API). У безсерверних архітектурах API Gateway зазвичай виступає як єдина точка входу, перенаправляючи запити до інших сервісів на основі різних параметрів API [25]. Він підтримує RESTful та WebSocket інтерфейси, тісно інтегрується з різноманітними сервісами AWS і навіть може генерувати клієнтські SDK для прискорення розробки.

Cognito – це рішення для авторизації, аутентифікації та управління користувачами. Сервіс спрощує реєстрацію нових користувачів та інтегрується з обліковими записами Facebook, Google та інших платформ. Головна перевага використання Cognito полягає в його інтеграції з API Gateway [26], що дозволяє

швидко розробити надійний механізм захисту API та даних користувачів, уникаючи необхідності розробки власного рішення для управління користувачами. Варто зазначити, що в даній архітектурі Cognito використовується виключно для реєстрації та авторизації, а зберігання даних користувачів буде здійснюватися в NoSQL базі даних DynamoDB (див. додаток А.3).

Lambda – один з основних сервісів AWS для безсерверних обчислень. Lambda дозволяє завантажувати виконуваний код, налаштовувати тригери для його запуску та одразу використовувати функціональність без необхідності управління середовищем виконання. У хмарних застосунках компоненти часто розділяються на численні автономні функції. В розробленій архітектурі кожна функціональна частина продукту реалізована через Lambda, яка може викликати інші більш дрібні операції. Інтеграція Lambda з API Gateway забезпечує автоматичний запуск відповідної функції після виклику конкретного методу інтерфейсу.

DynamoDB – це безсерверна NoSQL база даних, призначена для збереження інформації у форматі ключ-значення або документів. AWS автоматично керує безпекою, реплікацією та управлінням даними. Завдяки швидкому та гнучкому масштабуванню вона перевершує традиційні реляційні бази. DynamoDB Accelerator (DAX) та глобальні таблиці покращують продуктивність і дозволяють розподіляти дані між регіонами. Ця база даних часто використовується у зв’язці з Lambda, оскільки serverniless-функції не утримують стан.

Для асинхронного обміну повідомленнями без використання API Gateway застосовують комбінацію SNS та SQS. Simple Queue Service (SQS) – це розподілена система управління чергою повідомлень, що дозволяє програмам отримувати дані з черги асинхронно. Simple Notification Service (SNS) реалізує модель «публікація-підписка», дозволяючи кільком клієнтам отримувати повідомлення за підпискою. Проте, якщо клієнт пропустив публікацію, дані можуть бути втрачені. Один із поширених сценаріїв використання – «Fanout», коли інформація передається в SNS, а потім розподіляється в кілька SQS-черг. Наприклад, однією чергою можуть керувати мобільні пристрої, а іншою – десктопні додатки.

Отже, в даному розділі окреслено архітектурний підхід до розробки

продукту, представлено компоненти та їх інтеграцію, а також розглянуто ключові сервіси AWS – API Gateway, Cognito, Lambda, DynamoDB, SQS та SNS, з описом їх можливостей у серверній розробці.

3.4 Конструювання графічного інтерфейсу користувача

Графічний інтерфейс користувача (GUI) – це спосіб взаємодії з програмним забезпеченням, що використовує візуальні елементи замість текстових команд. Хоча GUI поступається інтерфейсу командного рядка (CLI) у швидкості виконання певних операцій і потребує більше часу на розробку, він значно покращує досвід користувача, роблячи процес навігації інтуїтивним та зручним. Крім того, GUI забезпечує ефективну візуалізацію даних, що сприяє швидкому сприйняттю інформації.

Створення графічного інтерфейсу є важливим етапом розробки додатків, оскільки неправильно спроектовані візуальні компоненти можуть ускладнити роботу з програмою. Для оптимальної взаємодії користувачів перед розробкою була створена структурна схема основних вікон додатку. На рисунку 3.5 зображено структуру головного вікна програмного продукту.

Основними елементами головного вікна є:

- Іконка застосунку;
- Рядок заголовка;
- Кнопки керування вікном (згортання, розгортання, закриття – залежно від платформи);
- Кнопка для додавання нового джерела повідомлень;
- Кнопка для створення звіту;
- Список останніх отриманих повідомлень.

Рисунок 3.5 Демонструє структурну схему головного вікна додатку, в якому представлені ключові елементи для користувача.

При натисканні на кнопку додавання джерела сповіщень відкривається окреме вікно, що містить набір специфічних графічних компонентів. Його організація та взаємодія між елементами наведені на рисунку 3.6.

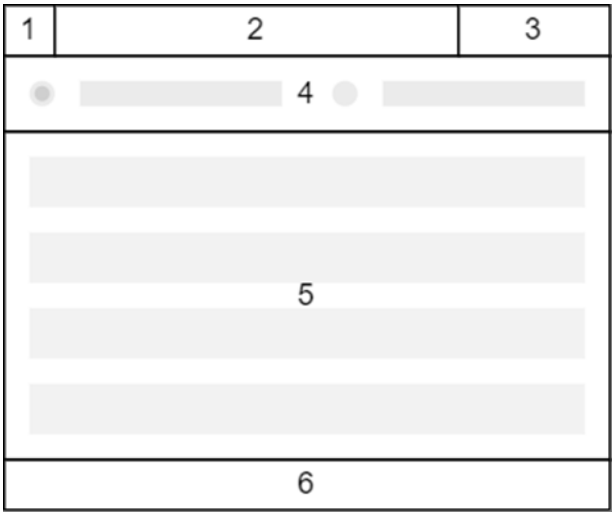

Рисунок 3.6 Демонструє структурну схему вікна створення джерела сповіщень, у якому розміщені ключові елементи для взаємодії користувача.

Структура інтерфейсу вікна включає наступні компоненти:

1. Іконка додатку, що забезпечує візуальну ідентифікацію.
2. Рядок заголовка, який містить назву активного вікна.
3. Кнопки керування вікном (згортання, розгортання, закриття).

4. Радіокнопки для вибору типу джерела, що дозволяють змінювати тип сповіщень.

5. Динамічні поля для введення параметрів джерела, які змінюються залежно від обраного типу.

6. Рядок стану, що інформує користувача про успішність додавання джерела або повідомляє про помилки.

Оскільки додаток підтримує різні типи сповіщень (зокрема вебхуки та парсинг веб-ресурсів), у вікні передбачено радіокнопки для вибору потрібного варіанта. Залежно від вибору, набір полів для введення даних змінюється, тому ця область представлена як один універсальний елемент у схемній діаграмі. Рядок стану відіграє важливу роль, оскільки надає користувачеві зворотний зв’язок щодо створення нового джерела, включаючи повідомлення про можливі помилки.

На головному екрані передбачена кнопка генерації звіту, яка активує серверні обчислення та відкриває нове вікно для візуалізації отриманих результатів. Рисунок 3.7 містить деталізовану схему цього вікна.

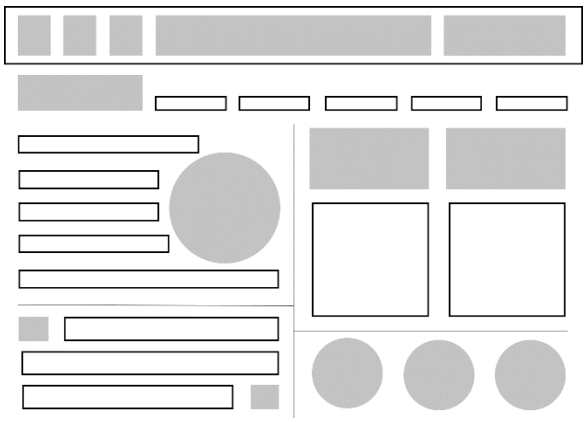

Рисунок 3.7 демонструє структурну схему вікна звіту даних.

Вікно містить такі основні компоненти:

1. Значок додатку, що забезпечує його візуальну ідентифікацію.

2. Заголовкова панель, яка містить назву активного розділу.

3. Кнопки керування вікном, що дозволяють змінювати його розмір або

закривати.

4. Кругова діаграма, що візуалізує процентне співвідношення повідомлень з різних джерел.

5. Перелік джерел з найбільшою кількістю даних, відображений у правій частині інтерфейсу.

6. Таблиця з повідомленнями, що увійшли до звіту, доступна для детального перегляду.

Кругова діаграма дозволяє швидко оцінити розподіл отриманих повідомлень за джерелами, тоді як список сервісів у правій частині містить найбільш активні джерела інформації. Нижня область вікна надає доступ до повного списку сповіщень, що використовувалися у звіті, дозволяючи користувачам детальніше проаналізувати отримані дані.

Таким чином, цей розділ окреслює основні особливості користувацького інтерфейсу, визначає логічну структуру ключових вікон програми та деталізує їхні функціональні складові.

3.5 Реалізація програмного модуля

Для реалізації функціональних можливостей додатку зі збору сповіщень було розроблено низку алгоритмів, описаних у підрозділах 2.1, 2.2 та 2.3. Зокрема, йдеться про:

3.5.1 алгоритм отримання сповіщень за допомогою вебхуків та відкритих API;

3.5.2 алгоритм збору сповіщень шляхом аналізу (парсингу) веб-ресурсів;

3.5.3 алгоритм формування звітів на основі накопиченої інформації.

Нижче детально розглянемо особливості реалізації програмного забезпечення.

Відповідно до архітектури, описаної в підрозділі 3.3, структура коду

розділена на дві основні частини: клієнтську та серверну. Серверна логіка базується на використанні хмарних сервісів платформи AWS. Користувацький інтерфейс реалізовано із застосуванням бібліотеки Tkinter, яка є частиною стандартного пакета мови програмування Python [30].

Запуск додатку здійснюється через основну функцію main, що знаходиться у виконуваному файлі main.py. Після старту програми ініціалізується бібліотека Tkinter, створюється головне вікно графічного інтерфейсу, приклад якого продемонстровано на рисунку 3.8. (див. лістинг А.1).

```
if __name__ == '__main__':
    ctypes.windll.shcore.SetProcessDpiAwareness(2)

    root = tk.Tk()
    sv_ttk.use_light_theme()
    root.title('Notification collector')
    root.columnconfigure(0, weight=1)
    root.rowconfigure(0, weight=1)

    login = App(root)
    login.grid(column=0, row=0, sticky='nwes', padx=15, pady=15)
    login.columnconfigure(0, weight=1)
    login.rowconfigure(0, weight=1)

    root.mainloop()
```

Рисунок 3.8 Запуск графічного інтерфейсу користувача.

Інтерфейс створюється в межах класу App, екземпляр якого ініціалізується у функції main. Цей клас виконує основну роль у формуванні графічних елементів за допомогою бібліотеки Tkinter, що використовується для побудови вікон і віджетів додатку. Крім того, клас App забезпечує взаємодію з серверною частиною програми (див. додаток А.2).

Для обміну даними із серверною частиною застосовується бібліотека boto3, що є офіційним SDK від Amazon для Python. Вона надає об’єктно-орієнтований інтерфейс для роботи з сервісами AWS. За допомогою методів sign_up та sign_in реалізовано механізм реєстрації та авторизації користувачів. Рисунок 3.9 містить приклад коду, який відповідає за процедуру входу в систему.

Механізм отримання повідомлень, заснований на вебхуках і публічних API,

реалізовано у вигляді окремої функції у середовищі безсерверних обчислень AWS Lambda. Для її виклику використовується API Gateway, що дозволяє налаштовувати URL-шлях, HTTP-методи доступу та застосовувати додаткові засоби захисту. Наприклад, можна обмежити доступ лише для аутентифікованих користувачів. Однак, у випадку роботи з вебхуками необхідно забезпечити відкритий доступ до адреси Lambda-функції, щоб зовнішні сервіси могли безперешкодно передавати дані.

```
def sign_in(self):
    if not self.username.get().strip() or not self.password.get().strip():
        messagebox.showerror('Error', 'Please fill in all fields!')
        return

    try:
        response = cognito.initiate_auth(
            ClientId=client_id,
            AuthFlow='USER_PASSWORD_AUTH',
            AuthParameters={
                'USERNAME': self.username.get(),
                'PASSWORD': self.password.get()
            }
        )
        print(response)
        self.token = response['AuthenticationResult']['AccessToken']
        messagebox.showinfo('Sign-in info', 'Successfully logged in!')
    except Exception as e:
        messagebox.showerror('Sign-in error', e)
```

Рисунок 3.9 Реалізація процедури входу користувача до системи.

Обробка вебхуків розпочинається з отримання вхідних параметрів, переданих через URL-запит. Якщо критично важливі параметри відсутні, подальше виконання коду припиняється, запобігаючи некоректному виклику функції.

Код, що відповідає за цей етап обробки, наведено на рисунку 3.10.

Якщо потрібно уточнити якісь аспекти або зробити текст ще більш унікальним, повідомте мене! Готовий допомогти з подальшими правками.

```python
# Get params
try:
    service = event['queryStringParameters']['service']
    user_id = event['queryStringParameters']['user_id']
except KeyError as e:
    print('Failed to get input parameters: ' + str(e))
    return {
        'statusCode': 403
    }
```

Рисунок 3.10 Отримання вхідних даних, необхідних для подальшої обробки вебхуків.

Після отримання параметрів здійснюється перевірка інформації про користувача у відповідній таблиці DynamoDB. Якщо запис користувача містить дані про сервіс-джерело, яке надіслало повідомлення через вебхук, скрипт продовжує виконання. В іншому випадку процес обробки завершується(див. лістинг А. 5).

Ця перевірка є важливим етапом, оскільки дозволяє відсіювати некоректні запити, які можуть виникати через неправильні налаштування вебхуків користувачем. Це необхідно для запобігання випадковому пересиланню сповіщень стороннім отримувачам.

Код, що реалізує цей механізм перевірки, наведено на рисунку 3.11.

```python
# Check user_id
try:
    users = dynamodb.Table('Users')
    query_res = users.query(
        KeyConditionExpression=boto3.dynamodb.conditions.Key('sub').eq(user_id)
    )

    if query_res['Count'] != 1:
        print('Failed to get user with sub: ' + user_id)
        return {
            'statusCode': 403
        }

    if service not in query_res['Items'][0]['webhooks']:
        print('Service ' + service + ' is not in user webhooks list: ' + user_id)
        return {
            'statusCode': 403
        }
```

Рисунок 3.11 Перевірка списків сервісів користувача у системі.

На цьому етапі алгоритм аналізує отримані дані, зіставляючи їх із записами у базі DynamoDB. Якщо відповідність знайдено, система продовжує виконання, інакше обробка припиняється, що дозволяє ефективно фільтрувати невірні або непередбачувані запити.

Подальший процес включає основну частину обробки даних, що ґрунтується на списку полів і відповідних дій, отриманих із таблиці Triggers у DynamoDB. Рисунок 3.12 містить код, який реалізує цей механізм.

```
triggers = dynamodb.Table('Triggers')

service_trigger = triggers.get_item(Key={'Service': service})
fields = service_trigger['Item']['fields']
actions = service_trigger['Item']['actions']

for k, v in fields.items():
    fields[k] = eval(v, {}, {'input': json.loads(event['body'])})

notification_text = eval(actions, {}, fields)
```

Рисунок 3.12 Обробка даних, отриманих від сервісів.

Після успішного виконання цього етапу створюється новий запис у таблиці Notifications в DynamoDB. Збережені дані забезпечують коректну передачу інформації та подальшу обробку.

Наступним кроком є надсилання оповіщення до клієнтського додатку, що дозволяє користувачеві отримати оновлену інформацію в реальному часі.

Код, який реалізує механізм збереження даних, наведено на рисунку 3.13.

```python
# Create notification
try:
    results = dynamodb.Table('Notifications')
    results.put_item(Item={
        'id': str(uuid.uuid4()),
        'service': service,
        'user_id': user_id,
        'text': notification_text
    })
except Exception as e:
    print('Failed to add new notification: ' + str(e))
    return {
        'statusCode': 500
    }
```

Рисунок 3.13 Лістинг коду, що відповідає за збереження даних оповіщення

Отримання сповіщень шляхом парсингу веб-ресурсів також реалізовано за допомогою безсерверних обчислень сервісу Lambda. На відміну від попереднього сценарію, ця функція не потребує вхідних даних від зовнішніх сервісів, а активується за розкладом. Перший крок – це отримання вмісту веб-сторінки та його обробка згідно з правилами, що містяться у змінній «payload». Код, що відповідає за цей процес, представлено на рисунку 3.14.

```python
# Parse page
page = requests.get(payload['url'])
soup = BeautifulSoup(page.content, 'html.parser')
container = soup.select(payload['container_selector'])
notif_list = []
for item in container:
    notif_text = ""
    for rule in payload['items']:
        rule_res = item.select_one(rule).string
        if rule_res is not None:
            notif_text += (rule_res + ' ')
    notif_list.append(notif_text.strip())
```

Рисунок 3.14 Лістинг, коду що відповідає за перевірку вмісту сторінки

Після формування повного переліку можливих повідомлень, що можуть бути виявлені на сторінці, виконується його співставлення з уже наявним списком, збереженим під час попереднього запуску. Це дозволяє виключити повідомлення, які вже були відправлені користувачеві раніше. Якщо після цього перевірки список не є порожнім, це означає, що в ньому залишилися нові повідомлення, які слід

передати в клієнтську частину застосунку. Додатково, результати обробки сторінки зберігаються для забезпечення правильності роботи Lambda-функції під час наступних запусків. Приклад реалізації цього процесу представлено на рисунку 3.15.

```
# Send new items
try:
    results_table = dynamodb.Table('Notifications')
    for notif in diff:
        results_table.put_item(Item={
            'id': str(uuid.uuid4()),
            'service': payload['url'],
            'user_id': payload['user_id'],
            'text': notif
        })
except Exception as e:
    print('Failed to add new notification: ' + str(e))

# Update previous list
try:
    rules_table = dynamodb.Table('ScrapeRules')
    rules_table.update_item(
        Key={'rule_id': payload['rule_id']},
        UpdateExpression='set previous = :r',
        ExpressionAttributeValues={
            ':r': notif_list,
        }
    )
except Exception as e:
    print('Failed to update previous list: ' + str(e))
```

важ

Рисунок 3.15 Частина коду, що відповідає за збереження інформації

Архітектура формування звітів базується на поділі на дві основні складові: серверну частину та клієнтське представлення. Така структура дозволяє раціонально розподіляти навантаження – побудова діаграм здійснюється локально на клієнтському пристрої, що використовує вже підготовлену інформацію. Завдяки цьому підхід демонструє вищу швидкість у порівнянні з повним перенесенням процесу формування графіки на хмарну інфраструктуру, де необхідно додатково зберігати та передавати візуалізовані зображення.

Підготовка даних для побудови звіту виконується за допомогою AWS Lambda. На початковому етапі функція надсилає запит до бази даних DynamoDB для отримання списку оповіщень, які відповідають заданим умовам користувача. Цей процес проілюстровано на рисунку 3.16, де враховуються фільтраційні параметри, вказані клієнтом.

```
input_data = json.loads(event['body'])
count_limit = input_data.get('limit', 200)
filters = boto3.dynamodb.conditions.Attr('user_id').eq(user_id)

response = None
try:
    n_table = dynamodb.Table('Notifications')
    response = n_table.scan(
        Limit=count_limit,
        FilterExpression=filters
    )
except Exception as e:
    print('Failed to get rules from DynamoDB: ' + str(e))
    return {
        'statusCode': 200,
        'body': json.dumps({
            'count': 0,
            'msg': 'Failed to get notifications'
        })
    }
```

Рисунок 3.16 Приклад коду для отримання даних з бази DynamoDB

У випадку, якщо список не порожній (що може виникнути внаслідок некоректного введення даних користувачем), виконується процес групування оповіщень відповідно до сервісу, з якого вони надійшли. Після завершення цього етапу вся зібрана інформація передається до клієнтського застосунку, як проілюстровано на рисунку 3.17.

```
counter = collections.Counter()
for notif in response['Items']:
    counter.update({notif['service']: 1})

return {
    'statusCode': 200,
    'body': json.dumps({
        'count': response['Count'],
        'msg': 'Success',
        'top': counter.most_common(10),
        'items': response['Items']
    }, ensure_ascii=False)
}
```

Рисунок 3.17 Приклад коду, що відповідає за формування та передачу даних

На клієнтській стороні обробка інформації для звітності зведена до мінімуму: користувачу одразу демонструється перелік отриманих сповіщень. Для побудови візуального представлення даних застосовується попередньо згрупований список

повідомлень за джерелами надходження. Для створення графіків використовується бібліотека matplotlib, яка дозволяє будувати наочні діаграми. Відображення результату здійснюється через графічний інтерфейс користувача, реалізований з використанням віджетів бібліотеки Tkinter. Код, що реалізує дану функціональність, наведено на рисунку 3.18.

```
# Pie chart
fig = Figure()
colors = ['#EDE3DB', '#ECB99C', '#BE9593', '#F1D7D6', '#6E6D71']
ax = fig.add_subplot(111)
ax.pie(data, labels=labels, autopct='%1.1f%%',
       startangle=90, colors=colors,
       pctdistance=0.8)

centre_circle = Circle((0, 0), 0.6, fc='white')
ax.add_patch(centre_circle)

chart_pie = FigureCanvasTkAgg(fig, frame_charts)
chart_pie.get_tk_widget().pack()
```

Рисунок 3.18 Фрагмент програмного коду, що реалізує побудову та відображення діаграми

У розділі 3.5 було проаналізовано ключові елементи програмної реалізації розробленого застосунку. Повний програмний код системи представлено в Додатку А.

РОЗІДЛ 4. ПЕРЕВІРКА ПРАЦЕЗДАТНОСТІ РОЗРОБЛЕНОЇ СИСТЕМИ

4.1 Огляд підходів до тестування ПЗ

Тестування програмного забезпечення – це організований процес перевірки правильності роботи системи з метою виявлення недоліків і оцінки відповідності між очікуваною поведінкою та фактичною реалізацією функціоналу. Незважаючи на те, що повне усунення всіх помилок майже неможливе, впровадження тестування ще на початкових етапах розробки дозволяє значно зменшити ризики, мінімізувати дефекти та суттєво підвищити загальну якість продукту – як у плані функціональних можливостей, так і щодо нефункціональних характеристик.

Типи тестування часто взаємопов’язані й можуть переслідувати схожі цілі, тому їх поділ є умовним. Основна класифікація базується на цілях тестування і включає такі категорії:

- Функціональне тестування – перевірка реалізації функцій відповідно до технічних вимог.
- Нефункціональне тестування – оцінка таких аспектів, як продуктивність, стабільність, безпечність та інші параметри, що не стосуються напряму функціоналу.
- Тестування змін (регресійне) – гарантує, що нещодавні оновлення не порушили вже наявну функціональність.

За рівнем доступу до внутрішньої структури програмного продукту виділяють:

- Тестування «чорного ящика» – здійснюється без знання внутрішньої логіки програми.
- Тестування «білого ящика» – вимагає повного доступу до вихідного коду та внутрішньої архітектури.
- Тестування «сірого ящика» – комбінує попередні два підходи, коли тестувальник має часткове уявлення про реалізацію системи.

Функціональні тести мають на меті перевірку поведінки програми при різних

сценаріях використання. До найпоширеніших належать:

- Модульне тестування – аналіз окремих частин коду (функцій, методів).
- Інтеграційне тестування – перевірка взаємодії між модулями.
- Системне тестування – тестування повної системи в цілому.
- Регресійне тестування – перевірка вже протестованої функціональності після внесення змін.

Нефункціональні тести не перевіряють, що саме виконує система, а радше як вона це робить. Вони охоплюють:

- ефективність при високих навантаженнях;
- стійкість до збоїв;
- зручність користувача;
- відповідність безпековим вимогам тощо.

У процесі тестування створеної системи були застосовані два методи:

1. Автоматизоване модульне тестування, яке є прикладом функціонального підходу, що передбачає доступ до внутрішньої логіки, а отже відноситься до тестування «білого ящика».

2. Ручне тестування за принципом «чорного ящика», яке не потребує знання коду й базується на взаємодії із зовнішнім інтерфейсом.

4.2 Перевірка роботи створеного додатку

Клієнтська сторона застосунку «Notification Collector» виконує роль інтерфейсу взаємодії з користувачем, відповідаючи лише за відображення графічного середовища та обробку дій, які виконує користувач. Вся інша логіка та обчислення реалізовані за допомогою хмарних сервісів AWS. Зокрема, AWS API Gateway забезпечує безпечний та зручний доступ до прикладного інтерфейсу, а AWS Lambda використовується для виконання серверної логіки у безсерверному середовищі.

Основна бізнес-логіка програмного забезпечення написана мовою Python всередині Lambda-функцій. Там же реалізовані додаткові механізми для взаємодії між серверною та клієнтською частинами. Для перевірки правильності реалізованого функціоналу був створений набір з 8 модульних тестів, які охоплюють такі сценарії:

1. створення нового профілю користувача після реєстрації через AWS Cognito;
2. ініціалізація інтеграції через webhook-запити;
3. створення інтеграцій, що базуються на парсингу веб-ресурсів;
4. формування звітної інформації;
5. отримання повного переліку надісланих сповіщень;
6. обробка глобального набору правил для парсингу;
7. обробка окремих правил, що застосовуються до парсера;
8. розбір та обробка даних, що надходять через вебхуки.

Процес модульного тестування автоматизовано із використанням бібліотеки PyTest, яка дозволяє зручно створювати тести та виконувати їх пакетно. Це значно полегшує перевірку всіх компонентів одразу. Кожен тест включає три основні етапи: підготовку вхідних даних, виклик цільової функції, яка реалізує певну дію, а також перевірку відповідності фактичного результату очікуваному. Приклад одного з таких тестів, що перевіряє створення користувача, подано на рисунку 4.1.

```
def test_add_user():
    test_uuid = str(uuid.uuid4())
    event = {'request': {'userAttributes': {'sub': test_uuid}}}
    response = add_user.lambda_function.lambda_handler(event, None)
    assert test_uuid == response['request']['userAttributes']['sub']
```

Рисунок 4.1 Код тесту для створення запису користувача.

В ході модульного тестування було підтверджено відсутність помилок. Результати всіх тестів, що були проведені, можна побачити на рисунку 4.2.

```
============================= test session starts ==============================
collecting ... collected 8 items

test.py::test_add_user PASSED                                            [ 12%]
test.py::test_create_wh_int PASSED                                       [ 25%]
test.py::test_create_sc_int PASSED                                       [ 37%]
test.py::test_generate_report PASSED                                     [ 50%]
test.py::test_get_all PASSED                                             [ 62%]
test.py::test_scrapper PASSED                                            [ 75%]
test.py::test_scrapper_targets PASSED                                    [ 87%]
test.py::test_webhooks PASSED                                            [100%]

============================== 8 passed in 4.61s ===============================
```

Рисунок 4.2 Результати модульного тестування.

Для здійснення тестування за принципом «чорної скриньки» було сформовано набір тестових сценаріїв, що охоплюють типові дії кінцевого користувача під час взаємодії з продуктом. Розглянемо один із таких сценаріїв більш докладно.

Тестовий сценарій №1 – «Створення нового облікового запису»:

1. Запустити клієнтську програму.

2. У вікні входу обрати опцію «Створити обліковий запис».

3. У відповідних полях ввести ім’я користувача, адресу електронної пошти та пароль.

4. Натиснути кнопку для підтвердження реєстрації.

Очікуваний результат: Після виконання вказаних дій має з’явитися повідомлення про успішну реєстрацію з інструкцією щодо підтвердження електронної пошти. Вікно із зазначеним повідомленням наведено на рисунку 4.3.

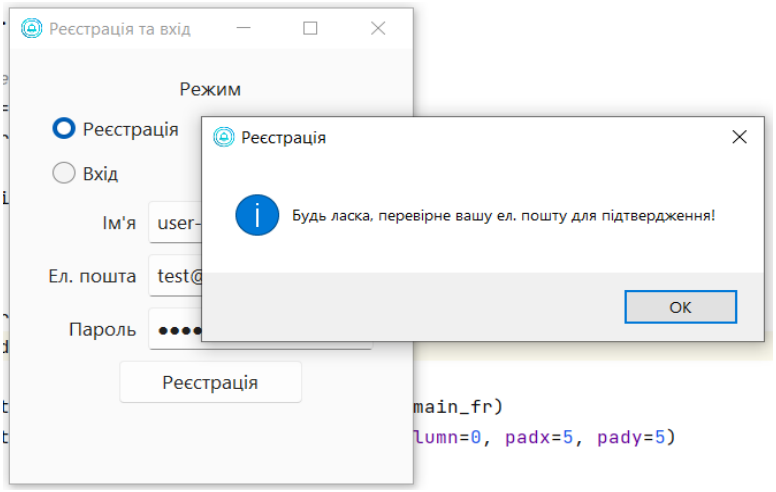

Рисунок 4.3 Результат виконання тест-кейсу №1

Тест-кейс №2: Додавання нового джерела через Webhook

Послідовність дій:

1. Відкрити клієнтську частину програмного забезпечення.

2. Увійти до системи, використовуючи облікові дані попередньо створеного акаунта.

3. У головному інтерфейсі вибрати функцію «Додати джерело».

4. У діалоговому вікні, що відкриється, задати тип джерела як «Webhook».

5. Зі списку сумісних сервісів обрати будь-який доступний варіант.

6. Натиснути кнопку «Створити» для підтвердження додавання джерела.

Очікуваний результат: Успішне завершення дії підтверджується появою повідомлення у спливаючому вікні. Система також автоматично створює унікальне URL-посилання, яке призначене для використання у вибраному сервісі як webhook-адреса. Це посилання виводиться у спеціальному полі з режимом лише для читання, а також доступне для копіювання.

Тест-кейс №3: Додавання джерела за допомогою парсера

Послідовність дій:

1. Запустити десктопну версію програмного забезпечення.

2. Виконати авторизацію в системі, використовуючи дані облікового запису, що був створений раніше.

3. У головному вікні інтерфейсу натиснути на кнопку «Додати джерело».

4. У діалоговому вікні обрати тип джерела – «Scrapper».

5. Заповнити необхідні поля: ввести URL-адресу, вказати контейнер та цільовий HTML-елемент.

6. Завершити створення джерела, натиснувши кнопку «Створити».

Очікуваний результат:

Після натискання кнопки «Створити» з’являється повідомлення, яке підтверджує успішне додавання нового джерела. У випадку, якщо всі введені дані є коректними, жодних додаткових кроків від користувача не потрібно. Візуальне представлення результату наведене на рисунку 4.5.

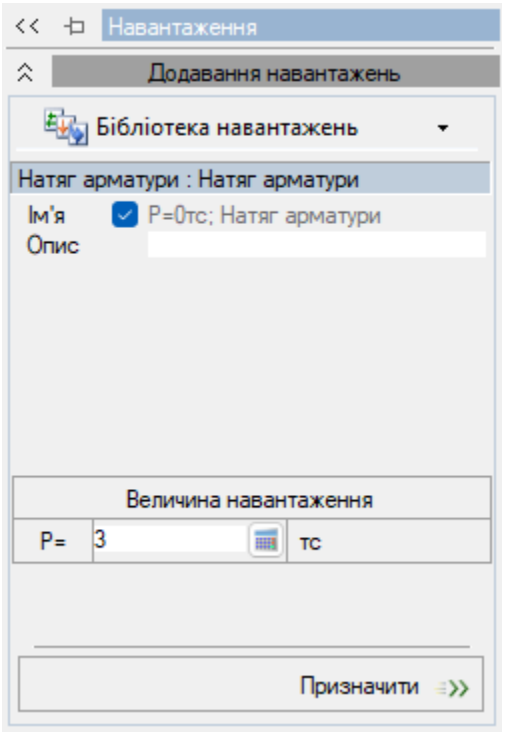

Рисунок 4.5 Візуалізація результатів перевірки сценарію тестування №3

4.3 Створення користувацької документації

Для початку роботи з програмою «Notification Collector» користувачу необхідно встановити клієнтський застосунок та забезпечити стабільне підключення до інтернету, що є обов’язковою умовою для коректного функціонування всіх можливостей програмного забезпечення. Після запуску застосунку з’являється стартове вікно для керування обліковим записом (рисунок 4.7), у якому можна створити новий акаунт або увійти до вже зареєстрованого профілю.

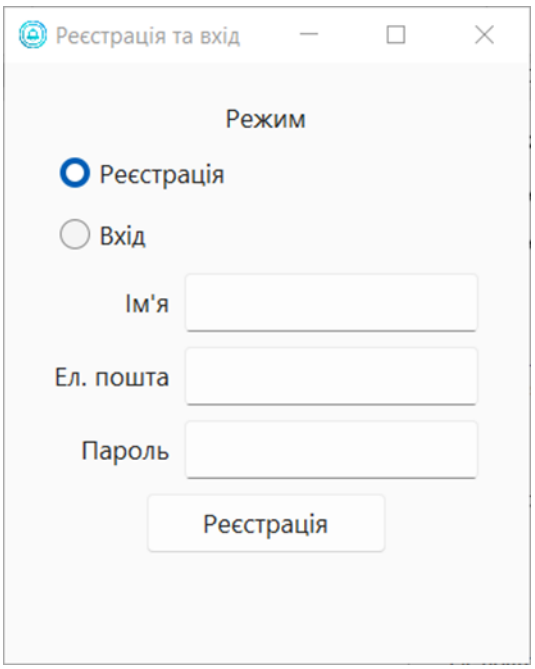

Рисунок 4.8 Інтерфейс авторизації та створення облікового запису

Під час реєстрації нового користувача система запитує підтвердження електронної пошти. Після натискання кнопки «Зареєструватися» автоматично надсилається лист із унікальним посиланням для активації профілю. Щойно обліковий запис активовано, користувач може переключитися в режим входу, скориставшись відповідною опцією у формі, і здійснити авторизацію. Після вдалого входу відкривається головне вікно програми – Notification Collector, як продемонстровано на рисунку 4.8.

Центральне вікно програми надає доступ до основного функціоналу. Інтерфейс умовно поділено на дві зони: верхня частина містить елементи керування для додавання нових джерел повідомлень і формування звітів, тоді як нижня частина відображає перелік останніх отриманих сповіщень. Цей список оновлюється динамічно, одразу після отримання нових даних. Кожне сповіщення містить назву джерела, його іконку, текст повідомлення, а також кнопку для видалення.

При натисканні на кнопку «Додати джерело» відкривається окреме вікно, наведене на рисунку 4.9, яке дозволяє налаштувати нове джерело збору повідомлень та здійснити інтеграцію з відповідним сервісом.

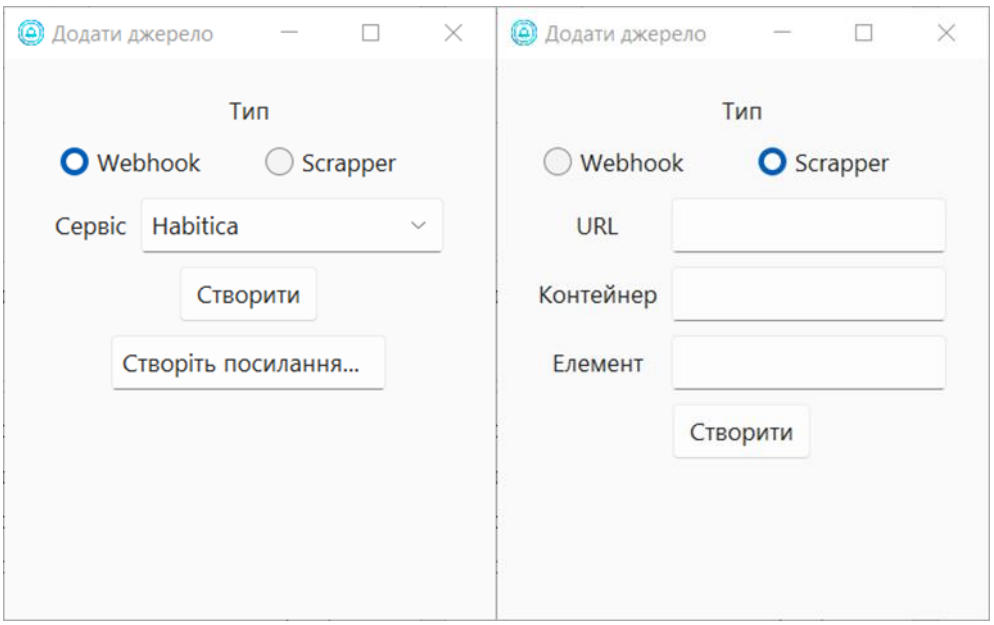

Рисунок 4.9 Форма додавання нового джерела для різних варіантів інтеграцій

Графічний інтерфейс цього вікна динамічно змінюється в залежності від обраного користувачем типу інтеграції. У разі вибору інтеграції через вебхуки, система автоматично генерує відповідне посилання, яке користувач повинен самостійно зареєструвати у зовнішньому сервісі, що підтримує такий механізм обміну. Оскільки процес підключення вебхука суттєво відрізняється у різних платформах, автоматизувати цей етап у межах застосунку неможливо.

Альтернативним способом інтеграції є використання вбудованого механізму парсингу, що дозволяє отримувати дані з веб-сторінок. Для налаштування такого типу підключення необхідно ввести адресу відповідного ресурсу (URL), а також визначити CSS-селектори для загального контейнера елементів та для вибірки окремих фрагментів інформації. Наприклад, можна використати селектори на кшталт «div.news-item» для контейнера та «div.news-title > a» для заголовків новин. Отримані дані використовуються для автоматичного формування оповіщень. Такий підхід вимагає базових знань HTML і CSS, проте забезпечує гнучкість та сумісність практично з будь-яким публічним сайтом.

Програма також має функцію автоматичного збору та аналізу останніх повідомлень. У лівій частині інтерфейсу розташовано кругову діаграму, яка ілюструє, як розподіляються оповіщення між різними джерелами. Праворуч

відображено список найактивніших сервісів-генераторів сповіщень. У нижній частині вікна представлений докладний перелік усіх повідомлень, що враховувалися при створенні статистики.

У підсумку, цей розділ містить детальний опис інструкцій для користувача, включаючи етапи реєстрації, активації облікового запису та огляд основних функцій, реалізованих у програмному забезпеченні для зручної роботи кінцевого користувача.

4.4 Технічні вимоги до комп'ютерного обладнання

Програмний додаток було розроблено із застосуванням мови програмування Python, яка є інтерпретованою та широко використовується для створення сучасних програмних рішень. Для реалізації функціоналу активно використовувалися спеціалізовані бібліотеки, що забезпечили зручну інтеграцію з сервісами AWS та надали можливості для побудови графічного інтерфейсу користувача. Зазвичай для запуску Python-додатків на кінцевих пристроях необхідно попередньо інсталювати Python-інтерпретатор і встановити всі залежні компоненти, що потенційно може ускладнити процес розгортання та використання програмного забезпечення користувачем.

Для усунення цієї складності було застосовано інструмент PyInstaller, який дозволяє об'єднати вихідний код разом із усіма потрібними бібліотеками та створити готовий до запуску виконуваний файл. Крім того, цей інструмент дає змогу гнучко налаштовувати збірку, додаючи до неї необхідні сторонні ресурси, такі як графічні файли, конфігураційні дані, параметри оформлення інтерфейсу тощо.

Мінімальні та рекомендовані технічні характеристики персонального комп’ютера для роботи з додатком представлені в таблиці 4.1.

Таблиця 4.1 – Параметри комп’ютера для роботи з веб-системами навчання

Параметр	Мінімальна конфігурація	Рекомендована конфігурація
Центральний процесор	Двоядерний процесор ARM Cortex-A72, 1,8 ГГц	Чотириядерний процесор AMD Ryzen 5, 3,6 ГГц
Об’єм ОЗП	2 ГБ LPDDR3	16 ГБ DDR5
Накопичувач	MicroSD 64 ГБ	SSD NVMe 512 ГБ
Графічний процесор	Відеоядро Mali-G52	Дискретна відеокарта NVIDIA GTX 1650
Операційна система	ChromeOS Flex, Debian 10	Windows 11 Pro, Ubuntu 22.04, macOS Ventura
Мережа	Мобільний інтернет 4G LTE зі стабільністю не менше 80%	Оптоволоконний інтернет ≥100 Мбіт/с, підтримка Wi-Fi 6

Отже, було розглянуто процес розповсюдження додатку для централізованого збору сповіщень з використанням бібліотеки "PyInstaller". Також було наведено мінімальні та рекомендовані вимоги до конфігурації персонального комп'ютера.

РОЗДІЛ 5. БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

5.1 Управління та нагляд за безпекою життєдіяльності в Україні.

Контроль за дотриманням законодавства з безпеки життєдіяльності в Україні здійснюють різні державні та громадські організації. Серед них державні органи різних рівнів компетенції:

- загальної (Верховна Рада, Кабінет Міністрів, виконавчі комітети місцевих рад народних депутатів, місцеві адміністрації) [21];
- спеціальної (контролюють діяльність підприємств, установ, організацій і громадян з питань охорони праці, охорони здоров'я, охорони навколишнього природнього середовища);
- галузевої (контролюють діяльність підприємств, установ, організацій і громадян з питань охорони праці, охорони здоров'я, охорони навколишнього природнього середовища у кожній з галузей народного господарства).

Центральним органом виконавчої влади управління безпекою та захистом у надзвичайних ситуаціях (НС), державного пожежного нагляду за станом пожежної безпеки в населених пунктах і на об'єктах незалежно від форм власності та цивільного захисту в Україні є Державна служба України з надзвичайних ситуацій (ДСНС). ДСНС України забезпечує реалізацію державної політики у сферах: цивільного захисту, захисту населення і територій від надзвичайних ситуацій та запобігання їх виникненню, ліквідації надзвичайних ситуацій, рятувальної справи, гасіння пожеж, пожежної та техногенної безпеки, діяльності аварійно-рятувальних служб, профілактики травматизму невиробничого характеру, а також гідрометеорологічної діяльності відповідно до Закону України «Про основні засади державного нагляду (контролю) у сфері господарської діяльності» [22], Кодексу та інших законодавчих актів. Вона видає накази організаційно-розпорядчого характеру, організовує та контролює їх виконання.

Координацію діяльності органів виконавчої влади у сфері цивільного захисту у межах своїх повноважень здійснюють:

- рада національної безпеки і оборони України;
- кабінет Міністрів України.

Державне управління охороною праці в Україні здійснюють:

- кабінет Міністрів України;
- міністерство соціальної політики України;
- міністерства та інші центральні органи державної виконавчої влади;
- місцева державна адміністрація, місцеві органи самоврядування.

Кабінет Міністрів України забезпечує:

- реалізацію державної політики в галузі охорони праці;
- затверджує національну програму по поліпшенню стану безпеки, гігієни праці і виробничого середовища;
- визначає функції міністерств, інших центральних органів державної виконавчої влади щодо створення безпечних і нешкідливих умов праці та нагляду за охороною праці;
- визначає порядок створення і використання державного, галузевого і регіональних фондів охорони праці.

Для розробки і реалізації системи державного управління охороною праці при Кабінеті Міністрів створена Національна рада з питань безпечної життєдіяльності населення, яку очолює віце-прем'єр-міністр України.

Міністерство соціальної політики України:

- здійснює державну експертизу умов праці;
- визначає порядок та здійснює контроль за якістю проведення атестації робочих місць на їх відповідність нормативним актам про охорону праці;
- бере участь у розробці нормативних актів про охорону праці.

Для координації, вдосконалення роботи з охорони праці і контролю за цією роботою в міністерствах та інших органах державної виконавчої влади створюються служби охорони праці. Реалізація державної політики охорони здоров'я покладається на органи державної виконавчої влади. Особисту відповідальність за неї несе Президент України. При реалізації цієї політики

Кабінет Міністрів України:

- організовує розробку та здійснення комплексних і цільових загальнодержавних програм;
- створює економічні, правові та організаційні механізми, що стимулюють ефективну діяльність в галузі охорони здоров'я;
- забезпечує розвиток мережі закладів охорони здоров'я, укладає міжурядові угоди і координує міжнародне співробітництво з питань охорони здоров'я тощо.

Міністерства, відомства та інші центральні органи державної виконавчої влади в межах своєї компетенції розробляють програми і прогнози в галузі охорони здоров'я, визначають єдині науково обґрунтовані державні стандарти, критерії та вимоги , що мають сприяти охороні здоров'я населення, формують і розміщують державні замовлення з метою матеріально-технічного забезпечення галузі, здійснюють державний контроль і нагляд та іншу виконавчо-розпорядчу діяльність в галузі охорони здоров'я. Спеціально уповноваженим органом державної виконавчої влади в галузі охорони здоров'я є Міністерство охорони здоров'я України, компетенція якого визначається положенням, що затверджується Кабінетом Міністрів України.

Державну санітарно-епідеміологічну службу України очолює державний санітарний лікар України – перший заступник міністра охорони здоровя, який призначається на посаду і звільняється з неї Кабінетом Міністрів України.

Основні напрямки діяльності державної санітарно-епідеміологічної служби:

- здійснення державного санітарно-епідеміологічного нагляду;
- визначення пріоритетних заходів у профілактиці захворювань, а також у охороні здоров'я населення від шкідливого впливу на нього факторів навколишнього середовища;
- вивчення, оцінка і прогнозування показників здоров'я населення залежно від стану середовища життєдіяльності людини, встановлення факторів навколишнього середовища, що шкідливо впливають на здоров'я населення;
- підготовка пропозицій по забезпеченню санітарного та епідемічного благополуччя населення, запобігання занесенню та поширенню особливо

небезпечних (у тому числі карантинних) та небезпечних інфекційних хвороб;

- контроль за усуненням причин і умов виникнення і поширення інфекційних, масових неінфекційних захворювань, отруєнь та радіаційних уражень людей;
- державний облік інфекційних і професійних захворювань та отруєнь;
- видача висновків державної санітарно-гігієнічної експертизи щодо об'єктів поводження з відходами;
- встановлення санітарно-гігієнічних вимог до продукції, що виробляється з відходів, та видача гігієнічного сертифіката на неї.

Державне управління в галузі охорони навколишнього природного середовища здійснюють: Кабінет Міністрів України, Міністерство екології та природних ресурсів, ради народних депутатів та їх виконавчі й розпорядчі органи, а також спеціальні уповноважені державні органи з питань охорони довкілля і використання природних ресурсів в Україні.

Міністерство екології та природних ресурсів:

- здійснює координацію всіх природоохоронних робіт в Україні;
- готує для Кабінету Міністрів пропозиції з питань охорони природи і раціонального використання водних ресурсів,
- розробляє пропозиції по вдосконаленню господарського механізму управління процесом природокористування, екологічні нормативи, правила та стандарти;
- готує довгострокові державні цільові програми з охорони довкілля;
- здійснює екологічну експертизу планів розвитку і розміщення продуктивних сил, контроль за дотриманням екологічних норм під час розроблення нової техніки, технології та матеріалів, екологічну експертизу проектів усіх новобудов і діючих промислових об'єктів.

Міністерства має право заборонити будівництво, реконструкцію або розширення об'єктів промислового чи іншого призначення, проведення робіт з експлуатації природних ресурсів, якщо вони порушують природоохоронне законодавство, а також притягти до відповідальності як організації, так і окремих

громадян у разі порушення природоохоронного законодавства. Міністерство працює в тісному зв'язку з Міністерством охорони здоров'я та підпорядкованими йому санітарно-епідеміологічними службами, міністерством аграрного комплексу, Державним комітетом по водному господарству, державним комітетом по земельним ресурсам, Міністерством транспорту України.

Державтоінспекція МВС України є спеціально уповноваженим державним органом, що здійснює контроль у сфері автомобільного дорожнього руху.

Вищий нагляд за додержанням і правильним застосуванням законів, у тому числі з безпеки життєдіяльності, здійснює Генеральний прокурор України і підпорядковані йому прокурори.

5.2 Захист електрообладнання від короткого замикання, перенавантаження

Захист електрообладнання під час користування комп’ютером є надзвичайно важливим аспектом, який значно впливає на довговічність техніки, безперебійну роботу та безпеку користувача [24]. Сучасні комп’ютери є високоточними електронними системами, що працюють із великою кількістю чутливих компонентів, які можуть вийти з ладу навіть за короткочасного електричного збою. Найбільшу небезпеку для ПК становлять коротке замикання та перевантаження електромережі. Ці явища можуть бути викликані як внутрішніми дефектами самого комп’ютера, так і зовнішніми факторами, такими як нестабільна напруга, неякісна проводка, переповнені подовжувачі, чи раптові стрибки електроживлення. У разі короткого замикання, між двома точками електричного кола виникає прямий контакт, що призводить до стрімкого збільшення сили струму, перегріву проводів, виходу з ладу обладнання і навіть до виникнення пожежі. Часто причиною стає несправний блок живлення, пошкоджені роз'єми, пил, волога або неправильне підключення периферійних пристроїв. Не менш небезпечним є перевантаження, коли споживана потужність перевищує допустимі межі для окремого кола або всієї

мережі. Особливо це актуально для домашніх умов, де на одну розетку може бути підключено декілька енергоспоживаючих пристроїв – комп’ютер, монітор, принтер, зарядні пристрої та інше обладнання.

Щоб уникнути цих небезпек, необхідно використовувати технічні засоби захисту. Одним з основних елементів є автоматичні вимикачі, встановлені в електрощитах – вони реагують на перевантаження або коротке замикання та розривають ланцюг, запобігаючи подальшому пошкодженню. В умовах використання комп’ютера особливу роль відіграють джерела безперебійного живлення (UPS), які забезпечують стабільну подачу енергії навіть при зникненні напруги в мережі, а також вирівнюють перепади напруги. Завдяки цьому комп’ютер не вимикається миттєво, що дає змогу зберегти дані або завершити роботу коректно. У деяких UPS також є функція автоматичного вимкнення системи через USB-підключення. Ще одним дієвим засобом є стабілізатори напруги, особливо актуальні в регіонах, де фіксуються часті стрибки напруги. Вони підтримують вихідну напругу в межах безпечного діапазону, зберігаючи стабільну роботу обладнання [24]. Для додаткового захисту використовуються мережеві фільтри або сурдж-протектори – це спеціальні пристрої, які нейтралізують імпульсні перенапруги, що можуть виникати в результаті ввімкнення потужної техніки або навіть через атмосферні явища, як-от блискавка.

Окрему увагу слід приділити заземленню [25]. Наявність правильного та якісного заземлення дозволяє убезпечити корпус комп’ютера від накопичення статичної напруги та зменшує ризик ураження електрострумом користувача. В умовах експлуатації важливо дотримуватись ряду простих, але ефективних правил: не перевантажувати розетки, не підключати багато пристроїв в один подовжувач, уникати контакту техніки з вологою або джерелами тепла, регулярно чистити внутрішні компоненти комп’ютера від пилу, перевіряти стан проводів і не використовувати несправні або дешеві кабелі та адаптери, які не мають сертифікації якості.

Таким чином, захист комп’ютерного обладнання від короткого замикання та перевантаження – це ціла система технічних і організаційних заходів, спрямованих

на збереження техніки та безперебійну роботу. У сучасному світі, де електропостачання не завжди стабільне, а комп’ютер є важливим елементом навчання, роботи чи дозвілля, вживання таких заходів стає не лише корисною рекомендацією, а й необхідною умовою безпечного користування. Ретельний підхід до питання електробезпеки дозволяє мінімізувати ризики, зберегти дані, зменшити витрати на ремонт чи заміну обладнання та забезпечити комфортну роботу в будь-яких умовах [23].

ВИСНОВКИ

У рамках бакалаврської дипломної роботи було реалізовано програмний продукт під назвою «Notification Collector», який забезпечує централізоване отримання сповіщень із використанням хмарних технологій на основі безсерверної архітектури. Застосунок створено з метою автоматизації процесів збору, збереження та візуального представлення сповіщень, що надходять із різних джерел. Це дозволяє користувачам ефективніше працювати з великим потоком інформації, значно спрощуючи її обробку.

Під час виконання дослідження було вивчено сучасний стан і динаміку розвитку хмарних рішень, особливо в контексті serverless-технологій, а також виявлено їх переваги у порівнянні з традиційними підходами. Проведено аналітичне зіставлення існуючих інструментів для збору сповіщень, що дало змогу оцінити їх функціональні можливості та недоліки. Отримані результати слугували підґрунтям для аргументованого вибору на користь створення індивідуального застосунку.

Було вдосконалено механізми отримання повідомлень із застосуванням вебхуків і відкритих API, покращено підхід до збору даних через парсинг веб-сайтів, а також оптимізовано процес формування аналітичних звітів на основі отриманої інформації.

Здійснено вибір інструментів і технологій для реалізації проекту шляхом варіантного аналізу. Основною мовою програмування обрано Python, а як середовище розробки – JetBrains PyCharm. Описано архітектуру серверної частини застосунку, створено структурні макети інтерфейсу користувача та реалізовано ключову функціональність.

У рамках тестування було класифіковано типи перевірки ПЗ, після чого обрано модульне тестування та методику «чорного ящика». Розроблено набір автоматизованих юніт-тестів і ручних сценаріїв для перевірки функцій. Результати тестування підтвердили коректну роботу програмного забезпечення та його

відповідність технічним вимогам. Також створено користувацьку інструкцію та визначено мінімальні системні вимоги до обладнання, на якому може функціонувати додаток.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. PyInstaller Official Guide [Електронний ресурс] – Режим доступу: https://pyinstaller.org/

2. REST API Best Practices [Електронний ресурс] – Режим доступу https://pyinstaller.org/

3. Методичні вказівки до виконання атестаційної роботи магістра спеціальності 121 – Інженерія програмного забезпечення (Освітньо-професійна програма — «Програмне забезпечення систем», Освітньо-наукова програма - «Інженерія програмного забезпечення») для студентів усіх форм навчання / Упор.: М.Р. Петрик, Д.М. Михалик, О.Ю. Петрик, Г.Б. Цуприк – Тернопіль: ТНТУ, 2020 – 51с.

4. Webhooks Introduction [Електронний ресурс] – Режим доступу: https://webhooks.fyi/

5. Serverless Computing Explained [Електронний ресурс] – Режим доступу: https://www.cloudflare.com/learning/serverless/what-is-serverless/

6. Matplotlib Documentation [Електронний ресурс] – Режим доступу: https://matplotlib.org/stable/contents.html

7. PyCharm Official Website [Електронний ресурс] – Режим доступу: https://www.jetbrains.com/pycharm/

8. GitHub – Python Projects [Електронний ресурс] – Режим доступу: https://github.com/topics/python

9. Гуменюк, І. В. REST API: розробка та впровадження. Київ: Видавництво «Либідь», 2018. 180 с.

10. Medium – Cloud Computing Articles [Електронний ресурс] – Режим доступу: https://medium.com/tag/cloud-computing

11. Towards Data Science – Python Tutorials [Електронний ресурс] – Режим доступу: https://towardsdatascience.com/

12. Kowalczyk, M. "Python for DevOps". O’Reilly, 2020. 320 с.

13. Smith, J. "Cloud Architecture Patterns". Packt Publishing, 2019. 250 с.

14. Python Official Documentation [Електронний ресурс] – Режим доступу: https://docs.python.org/3/

15. DynamoDB Developer Guide [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/

16. AWS Lambda Documentation [Електронний ресурс] – Режим доступу: https://docs.aws.amazon.com/lambda/

17. Brown, A. "Serverless Applications with AWS Lambda". Apress, 2021. 280 с.

18. Wilson, E. "Data Processing in Python". No Starch Press, 2018. 350 с.

19. TechRepublic – Cloud Security [Електронний ресурс] – Режим доступу: https://www.techrepublic.com/topic/cloud-security/

20. IEEE Xplore – Serverless Research Papers [Електронний ресурс] – Режим доступу: https://ieeexplore.ieee.org/

21. Конституція України: Закон від 28 червня 1996 р. № 254к/96-ВР. URL: https://zakon.rada.gov.ua/laws/show/254к/96-вр (дата звернення: 20.06.2025

22. Про основні засади державного нагляду (контролю) у сфері господарської діяльності: Закон України від 5 квітня 2007 р. № 877-V. URL: https://zakon.rada.gov.ua/laws/show/877-16 (дата звернення: 20.06.2025).

23. Про електроенергетику: Закон України від 16 жовтня 1997 р. № 575/97-ВР. URL: https://zakon.rada.gov.ua/laws/show/575-97-вр (дата звернення: 20.06.2025).

24. Безпека життєдіяльності: навч. посібник / Базилевич В.Д., Василенко В.С., Дубасенюк О.А. – Київ: Знання, 2016. – 432 с.

25. ДСТУ EN 62305-1:2017. Захист від блискавки. Частина 1. Загальні принципи. Київ: ДП «УкрНДНЦ», 2017. 42 с.

ДОДАТКИ

Додаток А

Лістинги

Лістинг А.1 – main.py

```
    import ctypes
    import tkinter as tk

    import sv_ttk
    from PIL import Image, ImageTk

    from app import App

if __name__ == '__main__': ctypes.windll.shcore.SetProcessDpiAwareness(2)

    app_img = Image.open('icon.png')

    root = tk.Tk()
    sv_ttk.use_light_theme()
    root.wm_iconphoto(True, ImageTk.PhotoImage(app_img))
    root.title('Notification collector')

    app = App(root)
    app.pack()

    root.update_idletasks() root.minsize(root.winfo_width(),
    root.winfo_height())

    root.mainloop()
```

Лістинг А.2 – app.py

```
    import time
    import tkinter as tk
    import uuid
    from tkinter import ttk, messagebox

    import requests
    from PIL import Image, ImageTk from
    PIL.ImageTk import PhotoImage

    import create
    import report
    import user


    class App(ttk.Frame):
def __init__(self, parent: tk.Misc | None):
    ttk.Frame.__init__(self, parent) self.parent =
    parent

            self.web_img = ImageTk.PhotoImage(Image.open('world-wide-
    web.png').resize((64, 64), Image.ANTIALIAS))
            self.github_img =
    ImageTk.PhotoImage(Image.open('github.png').resize((64, 64), Image.ANTIALIAS))
            self.habitica_img = ImageTk.PhotoImage(Image.open('habitica-
    logo.png').resize((64, 64), Image.ANTIALIAS))
            self.close_img = ImageTk.PhotoImage(Image.open('close.png').resize((16,
    16), Image.ANTIALIAS))

            ttk.Button(self, text='Додати джерело', width=20, command=lambda:
    create.CreateForm(self.parent, self.token)) \
```

```
        .grid(row=0, column=0, padx=10, pady=10) ttk.Button(self,
    text='Згенерувати звіт', width=20,
    style='Accent.TButton',
    command=lambda: report.ReportForm(self.parent, self.token)) \
    .grid(row=0, column=1, padx=10, pady=10)

    self.notif_frame = ttk.LabelFrame(self, text='Оповіщення', height=400)
    self.notif_frame.grid_propagate(False)
            self.notif_frame.grid(row=1, column=0, columnspan=2, sticky='nwes',
    padx=10, pady=10)

    self.scroll_canvas = tk.Canvas(self.notif_frame, height=400) self.scroll_frame =
    ttk.Frame(self.scroll_canvas)
            self.scrollbar = ttk.Scrollbar(self.notif_frame, orient=tk.VERTICAL,
    command=self.scroll_canvas.yview)
    self.scroll_canvas.configure(yscrollcommand=self.scrollbar.set)

             self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y, padx=2, pady=2)
             self.scroll_canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=2,
    pady=2)
             self.scroll_canvas.create_window(0, 0, window=self.scroll_frame,
    anchor=tk.NW)

    self.scroll_frame.bind('<Configure>', self._scroll_frame_conf)
    self.scroll_frame.bind('<Enter>', self._bound_to_mousewheel)
    self.scroll_frame.bind('<Leave>', self._unbound_to_mousewheel)

    self.parent.update_idletasks() login_win =
    user.UserForm(self.parent) self.token =
    login_win.get_token()

    self.get_notif()

        def _scroll_frame_conf(self, event):

    self.scroll_canvas.configure(scrollregion=self.scroll_canvas.bbox(tk.ALL))

def _bound_to_mousewheel(self, event): self.scroll_canvas.bind_all("<MouseWheel>",
    self._on_mousewheel)

def _unbound_to_mousewheel(self, event): self.scroll_canvas.unbind_all("<MouseWheel>")

    def _on_mousewheel(self, event):
    self.scroll_canvas.yview_scroll(int(-1 * (event.delta / 120)), tk.UNITS)

def add_notif(self, service, text): img:
    PhotoImage = self.web_img if service ==
    'habitica':
    service = 'Habitica' img =
    self.habitica_img
elif service == 'github': service =
    'GitHub' img = self.github_img

    item = ttk.Frame(self.scroll_frame)
    ttk.Label(item, image=img).grid(row=0, column=0, padx=5, rowspan=2) ttk.Label(item,
    text=service, width=35).grid(row=0, column=1, padx=10) ttk.Label(item, text=text,
    width=35, wraplength=320).grid(row=1,
    column=1, padx=10)
            ttk.Button(item, image=self.close_img, command=lambda:
    item.destroy()).grid(row=0, column=2, rowspan=2)

    item.pack(pady=5, padx=5)
```

```
    def get_notif(self):
with requests.Session() as s: s.headers.update({'Authorization':
    self.token}) response = s.get(
                    url='https://hyttrea95k.execute-api.eu-central-
    1.amazonaws.com/notifications'
    )

    response_data = response.json() if

    response.status_code != 200:
    messagebox.showerror('Помилка', 'Виникла помилка при виклику
    API: ' + str(response_data))
    self.destroy() return

if response_data['count'] == 0: return

for item in response_data['items']: self.add_notif(item['service'],
    item['text'])
```

Лістинг А.3 – user.py

```
    import tkinter as tk
    from tkinter import ttk, messagebox

    import boto3 as boto3
    from mypy_boto3_cognito_idp import CognitoIdentityProviderClient


    class UserForm(tk.Toplevel): def
    init__(self, parent):
    tk.Toplevel.__init__(self, parent)

    self.cognito: CognitoIdentityProviderClient = boto3.client('cognito- idp', 'eu-
    central-1')
    self.client_id = '7fl41iku26a83rou7l9h1td877'

    self.title("Реєстрація та вхід")
    self.geometry('350x400')

    self.token = None

    self.main_fr = ttk.Frame(self)
    self.main_fr.pack(padx=10, pady=20)

    self.mode = tk.IntVar() self.mode.trace_add('write',
    self.change_mode)
    ttk.Label(self.main_fr, text='Режим', anchor='w').grid(row=0, column=0, pady=5,
    padx=5)
    ttk.Radiobutton(self.main_fr, text='Реєстрація', value=0, variable=self.mode)\
    .grid(row=1, column=0, sticky=tk.W, pady=5, padx=5) ttk.Radiobutton(self.main_fr,
    text='Вхід', value=1, variable=self.mode)\
    .grid(row=2, column=0, sticky=tk.W, pady=5, padx=5)

    self.form_fr = ttk.Frame(self.main_fr)
    self.form_fr.grid(row=3, column=0)

    ttk.Label(self.form_fr, text='Ім\'я').grid(row=1, column=0, sticky='e')
    self.username = tk.StringVar()
```

```
ttk.Entry(self.form_fr, width=20, textvariable=self.username).grid(row=1,
column=1)

self.em_lb = ttk.Label(self.form_fr, text='Ел. пошта')
self.em_lb.grid(row=2, column=0, sticky='e') self.email =
tk.StringVar()
self.em_ent = ttk.Entry(self.form_fr, width=20, textvariable=self.email)
self.em_ent.grid(row=2, column=1)

ttk.Label(self.form_fr, text='Пароль').grid(row=3, column=0, sticky='e')
self.password = tk.StringVar()
ttk.Entry(self.form_fr, width=20, show='\u25CF',
textvariable=self.password).grid(row=3, column=1)

self.act_btn = ttk.Button(self.form_fr, text='Реєстрація', command=self.sign_up,
width=15)
self.act_btn.grid(row=4, column=0, columnspan=2)

for child in self.form_fr.winfo_children(): child.grid_configure(padx=5,
pady=5)

self.grab_set()

def change_mode(self, var, index, mode): if
self.mode.get() == 0:
self.act_btn.config(text='Реєстрація', command=self.sign_up) self.em_lb.grid()
self.em_ent.grid() else:
self.act_btn.config(text='Вхід', command=self.sign_in) self.em_lb.grid_remove()
self.em_ent.grid_remove()

def sign_up(self):
if (not self.username.get().strip()
or not self.email.get().strip()
or not self.password.get().strip()): messagebox.showerror('Помилка', 'Будь ласка,
заповність усі поля
форми!')
return

try:
             response = self.cognito.sign_up(
             ClientId=self.client_id,
             Username=self.username.get(),
             Password=self.password.get(), UserAttributes=[
             {
'Name': 'email',
'Value': self.email.get()
}
]
)
print(response)
if response['UserConfirmed'] is False: messagebox.showinfo('Реєстрація', 'Будь
ласка, перевірне вашу
ел. пошту для підтвердження!')
except Exception as e:
messagebox.showerror('Помилка реєсрації', e)

def sign_in(self):
if not self.username.get().strip() or not self.password.get().strip():
messagebox.showerror('Помилка', 'Будь ласка, заповність усі поля
форми!')
```

```
return

try:
             response = self.cognito.initiate_auth(
             ClientId=self.client_id,
             AuthFlow='USER_PASSWORD_AUTH',
             AuthParameters={
             'USERNAME': self.username.get(), 'PASSWORD':
             self.password.get()
}
)
print(response)
self.token = response['AuthenticationResult']['AccessToken']
messagebox.showinfo('Вхід', 'Вхід успішно виконано!') self.destroy()
except Exception as e: messagebox.showerror('Помилка
входу', e)

def get_token(self):
self.wait_window() return
self.token
```

Лістинг А.4 – create.py

```
import json
import tkinter as tk
from tkinter import ttk, messagebox

import requests


class CreateForm(tk.Toplevel):
def __init__(self, parent, token): tk.Toplevel.__
init__(self, parent)

self.title("Додати джерело")
self.geometry('350x400')

self.token = token self.main_fr =

ttk.Frame(self)
self.main_fr.pack(padx=10, pady=20)

self.mode = tk.IntVar() self.mode.trace_add('write',
self.change_mode)
ttk.Label(self.main_fr, text='Тип', anchor='w').grid(row=0, column=0, columnspan=2,
pady=5, padx=5)
ttk.Radiobutton(self.main_fr, text='Webhook', value=0, variable=self.mode)
\
.grid(row=1, column=0, sticky=tk.W, pady=5, padx=5) ttk.Radiobutton(self.main_fr,
text='Scrapper', value=1,
variable=self.mode) \
.grid(row=1, column=1, sticky=tk.W, pady=5, padx=5)

self.form_fr = ttk.Frame(self.main_fr) self.form_fr.grid(row=3,
column=0, columnspan=2)

# Webhook form
self.webhook_fr = ttk.Frame(self.form_fr) self.webhook_fr.grid()

         ttk.Label(self.webhook_fr, text='Сервіс').grid(row=0, column=0, padx=5,
pady=5)
         self.service_cb = ttk.Combobox(self.webhook_fr, values=['Habitica',
```

```
'GitHub'])
self.service_cb.current(0) self.service_cb.grid(row=0,
column=1, padx=5, pady=5)
ttk.Button(self.webhook_fr, text='Створити', command=self.create_wh_int)
\
        .grid(row=1, column=0, columnspan=2, padx=5, pady=5)
        self.wh_url = tk.StringVar(value='Створіть посилання...')
        ttk.Entry(self.webhook_fr, textvariable=self.wh_url, state='readonly') \
        .grid(row=2, column=0, columnspan=2, padx=5, pady=5)

        # Scrapper form
        self.scrapper_fr = ttk.Frame(self.form_fr)
        self.scrapper_fr.grid()

        ttk.Label(self.scrapper_fr, text='URL').grid(row=0, column=0, padx=5,
pady=5)
        self.scrapper_url = tk.StringVar()
        ttk.Entry(self.scrapper_fr, textvariable=self.scrapper_url).grid(row=0,
column=1, padx=5, pady=5)

ttk.Label(self.scrapper_fr, text='Контейнер').grid(row=1, column=0, padx=5,
pady=5)
self.scrapper_container = tk.StringVar() ttk.Entry(self.scrapper_fr,
textvariable=self.scrapper_container).grid(row=1, column=1, padx=5, pady=5)

ttk.Label(self.scrapper_fr, text='Елемент').grid(row=2, column=0, padx=5, pady=5)
self.scrapper_item = tk.StringVar()
ttk.Entry(self.scrapper_fr, textvariable=self.scrapper_item).grid(row=2, column=1,
padx=5, pady=5)
ttk.Button(self.scrapper_fr, text='Створити', command=self.create_sc_int)
\
.grid(row=3, column=0, columnspan=2, padx=5, pady=5) self.scrapper_fr.grid_remove()

self.grab_set()

def change_mode(self, var, index, mode): if
self.mode.get() == 0:
self.scrapper_fr.grid_remove()
self.webhook_fr.grid()
else:
self.webhook_fr.grid_remove()
self.scrapper_fr.grid()

def create_wh_int(self): if
self.token is None:
messagebox.showerror('Доступ', 'Будь ласка, ввійдіть в акаунт для використання
додатку!')
return

service = 'habitica' if self.service_cb.current() == 0 else 'github'

with requests.Session() as s: s.headers.update({'Authorization':
self.token}) s.headers.update({'Content-Type': 'application/json'})
response = s.post(
url='https://hyttrea95k.execute-api.eu-central- 1.amazonaws.com/create/webhook',
data=json.dumps({ "service": service
})
```

```
)

response_data = response.json() if

response.status_code != 201:
messagebox.showerror('Створення', 'Виникла помилка при виклику
API: ' + str(response_data))
return

self.wh_url.set(response_data['url']) messagebox.showinfo('Створення', 'URL
згенеровано, можете продовжити
конфігурацію')

def create_sc_int(self): if
self.token is None:
messagebox.showerror('Доступ', 'Будь ласка, ввійдіть в акаунт для використання
додатку!')
return

if (not self.scrapper_url.get().strip()
or not self.scrapper_container.get().strip() or not
self.scrapper_item.get().strip()):
messagebox.showerror('Дані', 'Будь ласка, заповність усі поля
форми!')
return

with requests.Session() as s: s.headers.update({'Authorization':
self.token}) s.headers.update({'Content-Type': 'application/json'})
response = s.post(
url='https://hyttrea95k.execute-api.eu-central- 1.amazonaws.com/create/scrapper',
data=json.dumps({
"url": self.scrapper_url.get().strip(), "container":
self.scrapper_container.get().strip(), "item":
self.scrapper_item.get().strip(),
})
)

response_data = response.json() if

response.status_code != 201:
messagebox.showerror('Створення', 'Виникла помилка при виклику
API: ' + str(response_data))
return

messagebox.showinfo('Створення', 'Джерело додано!')
```

Лістинг А.5 – report.py

```
import json
import tkinter as tk
from tkinter import ttk, messagebox

import requests
from PIL import ImageTk, Image from
PIL.ImageTk import PhotoImage
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg from
matplotlib.figure import Figure
from matplotlib.patches import Circle


class ReportForm(tk.Toplevel):
def __init__(self, parent, token):
```

```
        tk.Toplevel.__init__(self, parent)

                self.web_img = ImageTk.PhotoImage(Image.open('world-wide-
        web.png').resize((64, 64), Image.ANTIALIAS))
                self.github_img =
        ImageTk.PhotoImage(Image.open('github.png').resize((64, 64), Image.ANTIALIAS))
                self.habitica_img = ImageTk.PhotoImage(Image.open('habitica-
        logo.png').resize((64, 64), Image.ANTIALIAS))
                self.close_img = ImageTk.PhotoImage(Image.open('close.png').resize((16,
        16), Image.ANTIALIAS))

        self.title("Звіт")
        # self.geometry('350x400')

        self.token = token

        labels, data, items = self.get_data()

        self.main_fr = ttk.Frame(self)
        self.main_fr.pack(padx=10, pady=20)

        # Pie chart
        fig = Figure(figsize=(4, 3))
        colors = ['#EDE3DB', '#ECB99C', '#BE9593', '#F1D7D6', '#6E6D71']
        ax = fig.add_subplot(111)
    ax.pie(data, labels=labels, autopct='%1.1f%%', startangle=90,
        colors=colors,
        pctdistance=0.8, textprops={'fontsize': 8}, radius=0.8)

        centre_circle = Circle((0, 0), 0.4, fc='white')
        ax.add_patch(centre_circle)

        chart_pie = FigureCanvasTkAgg(fig, self.main_fr)
        chart_pie.get_tk_widget().grid(row=0, column=0, padx=5, pady=5)

        # Top
        top_fr = ttk.Frame(self.main_fr)
        ttk.Label(top_fr, text='Топ:').pack(padx=5, pady=5) for lab in
        labels:
            ttk.Label(top_fr, text=lab).pack(padx=5, pady=5) top_fr.grid(row=0,
        column=1, padx=5, pady=5)

        # All
                self.notif_frame = ttk.LabelFrame(self.main_fr, text='Оповіщення',
        height=400)
        self.notif_frame.grid_propagate(False)
                self.notif_frame.grid(row=1, column=0, columnspan=2, sticky='nwes',
        padx=10, pady=10)

        self.scroll_canvas = tk.Canvas(self.notif_frame, height=400) self.scroll_frame =
        ttk.Frame(self.scroll_canvas)
                self.scrollbar = ttk.Scrollbar(self.notif_frame, orient=tk.VERTICAL,
        command=self.scroll_canvas.yview)
        self.scroll_canvas.configure(yscrollcommand=self.scrollbar.set)

                 self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y, padx=2, pady=2)
                 self.scroll_canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True, padx=2,
        pady=2)
                 self.scroll_canvas.create_window(0, 0, window=self.scroll_frame,
        anchor=tk.NW)

        self.scroll_frame.bind('<Configure>', self._scroll_frame_conf)
        self.scroll_frame.bind('<Enter>', self._bound_to_mousewheel)
        self.scroll_frame.bind('<Leave>', self._unbound_to_mousewheel)
```

```
for item in items: self.add_notif(item['service'],
    item['text'])

    self.grab_set()

        def _scroll_frame_conf(self, event):

    self.scroll_canvas.configure(scrollregion=self.scroll_canvas.bbox(tk.ALL))

def _bound_to_mousewheel(self, event): self.scroll_canvas.bind_all("<MouseWheel>",
    self._on_mousewheel)

def _unbound_to_mousewheel(self, event): self.scroll_canvas.unbind_all("<MouseWheel>")

    def _on_mousewheel(self, event):
    self.scroll_canvas.yview_scroll(int(-1 * (event.delta / 120)), tk.UNITS)

def add_notif(self, service, text): item =
    ttk.Frame(self.scroll_frame)

    img: PhotoImage = self.web_img if service
    == 'habitica':
    service = 'Habitica' img =
    self.habitica_img
elif service == 'github': service =
    'GitHub' img = self.github_img

    ttk.Label(item, image=img).grid(row=0, column=0, padx=5, rowspan=2) ttk.Label(item,
    text=service, width=55).grid(row=0, column=1, padx=10) ttk.Label(item, text=text,
    width=55, wraplength=500).grid(row=1,
    column=1, padx=10)
            ttk.Button(item, image=self.close_img, command=lambda:
    item.destroy()).grid(row=0, column=2, rowspan=2)

        item.pack(pady=5, padx=5) def

    get_data(self):
with requests.Session() as s: s.headers.update({'Authorization':
    self.token}) s.headers.update({'Content-Type': 'application/json'})
    response = s.post(
                    url='https://hyttrea95k.execute-api.eu-central-
    1.amazonaws.com/report',
data=json.dumps({ "limit": 50
    })
    )

    response_data = response.json()

    if response.status_code != 200:
    messagebox.showerror('Помилка', 'Виникла помилка при виклику
    API: ' + str(response_data))
    self.destroy() return

    if response_data['count'] == 0:
    messagebox.showerror('Дані', 'Недостатньо даних для формування
    звіту!')
                    self.destroy()
                    return
```

```
    labels = [] data = []
for record in response_data['top']:
    labels.append(record[0]) data.append(record[1])

    return labels, data, response_data['items']
```

Лістинг А.5 – add_user\lambda_function.py

```
    import boto3

    dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context): try:
    users = dynamodb.Table('Users')
    users.put_item(Item={
    'sub': event['request']['userAttributes']['sub'], 'webhooks': [],
    'scrappers': []
    })

except Exception as e: print(str(e))

    return event
```

Лістинг А.6 – create_wh_int\lambda_function.py

```
    import json
    import boto3

def lambda_handler(event, context): dynamodb =
    boto3.resource('dynamodb')

    user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

    request_body = json.loads(event['body']) service
    = request_body['service']

    try:
              users_table = dynamodb.Table('Users')
              users_table.update_item(
              Key={'sub': user_id},
              UpdateExpression='set webhooks = list_append(webhooks, :i)',
              ExpressionAttributeValues={
              ':i': [service],
    }
    )
    except Exception as e:
    print('Failed to update user: ' + str(e)) return {
    'statusCode': 500,
    'body': json.dumps({'message': 'Failed to update user data'})
    }

    responce = {
    'message': 'Success',
            'url': 'https://hyttrea95k.execute-api.eu-central-
    1.amazonaws.com/webhook-collector?user_id=' + user_id + '&service=' + service
    }
```

```
return {
'statusCode': 201,
'body': json.dumps(responce)
}
```

Лістинг А.7 – create_sc_int\lambda_function.py

```
import json import uuid

import boto3

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

request_body = json.loads(event['body']) url = request_body['url']
container = request_body['container'] item = request_body['item']

try:
          rules_table = dynamodb.Table('ScrapeRules') rules_table.put_item(Item={
          'rule_id': str(uuid.uuid4()), 'user_id': user_id,
          'url': url, 'container_selector': container, 'items': [item],
          'previous': [],
})
except Exception as e:
print('Failed to add new rule: ' + str(e)) return {
'statusCode': 500,
'body': json.dumps({'message': 'Failed to create new rule'})
}

return {
'statusCode': 201,
'body': json.dumps({'message': 'Success'})
}
```

Лістинг А.8 – generate_report\lambda_function.py

```
import collections import json

import boto3

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

input_data = json.loads(event['body']) count_limit = input_data.get('limit', 200)
filters = boto3.dynamodb.conditions.Attr('user_id').eq(user_id) response = None
```

```
    try:
              n_table = dynamodb.Table('Notifications') response = n_table.scan(
              Limit=count_limit, FilterExpression=filters
    )
    except Exception as e:
    print('Failed to get rules from DynamoDB: ' + str(e)) return {
    'statusCode': 200, 'body': json.dumps({
    'count': 0,
    'msg': 'Failed to get notifications'
    })
    }

if response['Count'] == 0: return {
    'statusCode': 200, 'body': json.dumps({
    'count': 0,
    'msg': 'Not enough notification for analysis'
    })
    }

    counter = collections.Counter() for notif in response['Items']:
    counter.update({notif['service']: 1})

    return {
    'statusCode': 200, 'body': json.dumps({
    'count': response['Count'],
    'msg': 'Success',
    'top': counter.most_common(10), 'items': response['Items']
    }, ensure_ascii=False)
    }
```

Лістинг А.9 – get\lambda_function.py

```
    import collections import json

    import boto3

    dynamodb = boto3.resource('dynamodb')


    def lambda_handler(event, context):
    user_id = event['requestContext']['authorizer']['jwt']['claims']['sub']

    filters = boto3.dynamodb.conditions.Attr('user_id').eq(user_id)

    response = None try:
    n_table = dynamodb.Table('Notifications') response = n_table.scan(
    Limit=200, FilterExpression=filters
    )
    except Exception as e:
    print('Failed to get rules from DynamoDB: ' + str(e)) return {
```

```
    'statusCode': 200, 'body': json.dumps({
    'count': 0,
    'msg': 'Failed to get notifications'
    })
    }

    return {
    'statusCode': 200, 'body': json.dumps({
    'count': response['Count'],
    'msg': 'Success', 'items': response['Items']
    }, ensure_ascii=False)
    }
```

Лістинг А.10 – scrapper\lambda_function.py

```
    import ast import uuid

    import boto3 import requests
    from bs4 import BeautifulSoup dynamodb = boto3.resource('dynamodb')

def lambda_handler(event, context): for record in event['Records']:
    # Get payload
    payload = ast.literal_eval(record["body"])

    # Parse page
    page = requests.get(payload['url'])
    soup = BeautifulSoup(page.content, 'html.parser') container =
    soup.select(payload['container_selector']) notif_list = []
for item in container: notif_text = ""
    for rule in payload['items']:
    rule_res = item.select_one(rule).string if rule_res is not None:
            notif_text += (rule_res + ' ') notif_list.append(notif_text.strip())

    # Get only new items
    diff = [x for x in notif_list if x not in payload['previous']] if len(diff) == 0:
    print('No new items') return {
    'statusCode': 200
    }

    # Send new items
    try:
    results_table = dynamodb.Table('Notifications') for notif in diff:
results_table.put_item(Item={ 'id': str(uuid.uuid4()),
    'service': payload['url'], 'user_id': payload['user_id'], 'text': notif
    })
```

```
    except Exception as e:
    print('Failed to add new notification: ' + str(e))

    # Update previous list
    try:
    rules_table = dynamodb.Table('ScrapeRules') rules_table.update_item(
    Key={'rule_id': payload['rule_id']}, UpdateExpression='set previous = :r',
    ExpressionAttributeValues={
    ':r': notif_list,
    }
    )
    except Exception as e:
    print('Failed to update previous list: ' + str(e))

    return {
    'statusCode': 200
    }
```

Лістинг А.11 – scrapper_targets\lambda_function.py

```
    import boto3

    dynamodb = boto3.resource('dynamodb') sqs
    = boto3.resource('sqs')


    def lambda_handler(event, context):
    # Get all rules rules
    = None try:
    rules_table = dynamodb.Table('ScrapeRules') response =
    rules_table.scan()

if response['Count'] == 0: print('No rules
    in table') return

    rules = response['Items']

    except Exception as e:
    print('Failed to get rules from DynamoDB: ' + str(e))

    # Send items to SQS
    try:
            queue = sqs.get_queue_by_name(QueueName='notification-collector-
    targets')

    max_batch_size = 10
            chunks = [rules[x:x + max_batch_size] for x in range(0, len(rules),
    max_batch_size)]
for chunk in chunks: entries = []
for x in chunk: entry = {
    'Id': x['rule_id'], 'MessageBody': str(x)
    }
    entries.append(entry)
    resp = queue.send_messages(Entries=entries)

if len(resp.get('Failed', [])) != 0: print('Failed to send some
    rules to SQS!')
```

```
except Exception as e:
print('Failed to send rules to SQS: ' + str(e))
```

Лістинг А.12 – webhooks\lambda_function.py

```
import json import boto3 import uuid

dynamodb = boto3.resource('dynamodb')


def lambda_handler(event, context):
# Parameters service = None user_id = None

# Get params
try:
service = event['queryStringParameters']['service'] user_id =
event['queryStringParameters']['user_id']
except KeyError as e:
print('Failed to get input parameters: ' + str(e)) return {
'statusCode': 403
}

# Check user_id
try:
users = dynamodb.Table('Users') query_res = users.query(

KeyConditionExpression=boto3.dynamodb.conditions.Key('sub').eq(user_id)
)

if query_res['Count'] != 1:
print('Failed to get user with sub: ' + user_id) return {
'statusCode': 403
}

if service not in query_res['Items'][0]['webhooks']:
print('Service ' + service + ' is not in user webhooks list: ' +
user_id)
            return {
            'statusCode': 403
}

except Exception as e:
print('Failed to query users: ' + str(e)) return {
'statusCode': 500
}

# Get fields and actions for service
notification_text = None try:
triggers = dynamodb.Table('Triggers')

service_trigger = triggers.get_item(Key={'Service': service}) fields =
service_trigger['Item']['fields']
actions = service_trigger['Item']['actions'
```

```
for k, v in fields.items():
fields[k] = eval(v, {}, {'input': json.loads(event['body'])})

    notification_text = eval(actions, {}, fields) except Exception as e:
print('Failed to process notification: ' + str(e)) return {
'statusCode': 500
}

# Create notification
try:
results = dynamodb.Table('Notifications') results.put_item(Item={
'id': str(uuid.uuid4()), 'service': service, 'user_id': user_id, 'text':
notification_text
})
except Exception as e:
print('Failed to add new notification: ' + str(e)) return {
'statusCode': 500
}

return {
'statusCode': 200
```

Додаток Б

Диск з роботою