

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ НАУК**

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Мікросервісна архітектура для системи моніторингу
та алертингу рекламних кампаній»

на здобуття освітнього ступеня магістра
зі спеціальності F3 Комп'ютерні науки
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні науки
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

(підпис) Андрій МЕТЕЛЮК
(Ім'я, ПРІЗВИЩЕ здобувача)

Виконав:
здобувач вищої освіти
група КНДМ-61

Андрій МЕТЕЛЮК

Керівник:
науковий ступінь,
вчене звання

Андрій ЧИЧУР
к.т.н., доцент

Рецензент:
науковий ступінь,
вчене звання

(Ім'я, ПРІЗВИЩЕ)

Київ 2025

**ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ**

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерних наук

Ступінь вищої освіти Магістр

Спеціальність F3 Комп'ютерні науки

Освітньо-професійна програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедру Комп'ютерних наук

_____ Віктор ВИШНІВСЬКИЙ
« _____ » _____ 2025 р.

**ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ**

_____ Метелюку Андрію Володимировичу

(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Мікросервісна архітектура для системи моніторингу та алертингу рекламних кампаній

керівник кваліфікаційної роботи Андрій ЧИЧУР к.т.н., доцент,

(Ім'я, ПРІЗВИЩЕ науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «30» 10.2025р. №467

2. Строк подання кваліфікаційної роботи «26» грудня 2025р.

3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, технічна документація Google Cloud API, структура даних рекламних кампаній.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

Аналіз принципів побудови систем моніторингу рекламних кампаній.

Проектування мікросервісної архітектури та алгоритмів детекції аномалій.

Дослідження ефективності та оптимізація роботи програмного забезпечення.

5. Перелік графічного матеріалу: презентація

1. Актуальність та проблематика ручного моніторингу трафіку.
 2. Схема розробленої мікросервісної архітектури.
 3. Алгоритми виявлення аномалій та логіка сповіщень.
 4. Результати дослідження ефективності та приклади роботи системи.
6. Дата видачі завдання «15» вересня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
	Аналіз науково-технічної літератури та існуючих систем моніторингу	16.09-30.09.25	виконано
	Проектування мікросервісної архітектури та вибір технологічного стеку	01.10-14.10.25	виконано
	Розробка структури бази даних та налаштування хмарного середовища	15.10-31.10.25	виконано
	Програмна реалізація алгоритмів детекції аномалій та механізму сповіщень	01.11-20.11.25	виконано
	Проведення тестування, дослідження ефективності та оптимізація системи	21.11-05.12.25	виконано
	Економічне обґрунтування та аналіз результатів роботи	06.12-12.12.25	виконано
	Оформлення пояснювальної записки: вступ, висновки, список джерел)	13.12-20.12.25	виконано
	Підготовка презентаційних та демонстраційних матеріалів	21.12-25.12.25	виконано

Здобувач вищої освіти

(підпис)

Андрій МЕТЕЛЮК

(Ім'я, ПРІЗВИЩЕ)

Керівник

кваліфікаційної роботи

(підпис)

Андрій ЧИЧУР

(Ім'я, ПРІЗВИЩЕ)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 73 стор., 40 рис., 22 джерела.

Наукове завдання – розробка автоматизованої системи моніторингу рекламних кампаній та алгоритмів виявлення аномалій у реальному часі.

Мета роботи – мінімізувати фінансові втрати маркетингових відділів шляхом підвищення оперативності реагування на інциденти.

Об’єкт дослідження – процес моніторингу та управління ефективністю інтернет-реклами.

Предмет дослідження – мікросервісна архітектура та методи автоматизації обробки даних у хмарному середовищі.

Короткий зміст роботи: У роботі проаналізовано проблеми ручного контролю цифрової реклами та обґрунтовано доцільність автоматизації.

Спроектовано та розроблено мікросервісну систему на базі Google Cloud Platform для обробки даних у реальному часі. Реалізовано програмні модулі мовою Python для збору метрик, виявлення аномалій та надсилання сповіщень через Telegram.

Проведено експериментальне дослідження швидкодії системи та розраховано економічний ефект, що підтвердив її ефективність порівняно з традиційними методами.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСИ, ХМАРНІ ТЕХНОЛОГІЇ, МОНІТОРИНГ, GOOGLE CLOUD, PYTHON, BIGQUERY, TELEGRAM API.

ABSTRACT

Text part of the master's qualification work: 73 pages, 40 pictures, 22 sources.

The scientific task is the development of an automated system for monitoring advertising campaigns and real-time anomaly detection algorithms.

The purpose of the work is to minimize financial losses of marketing departments by increasing the speed of response to incidents.

Object of research – the process of monitoring and managing digital advertising performance.

Subject of research – microservice architecture and methods of data processing automation in a cloud environment.

Summary of the work: The thesis analyzes the problems of manual digital advertising control and justifies the feasibility of automation.

A microservice system based on Google Cloud Platform was designed and developed for real-time data processing. Python software modules were implemented for metrics collection, anomaly detection, and sending notifications via Telegram.

An experimental study of the system's performance was conducted, and the economic effect was calculated, confirming its cost-effectiveness compared to traditional methods.

KEYWORDS: MICROSERVICES, CLOUD TECHNOLOGIES,
MONITORING, GOOGLE CLOUD, PYTHON, BIGQUERY, TELEGRAM API.

ЗМІСТ

ВСТУП.....	10
1 КОНЦЕПЦІЯ ТА АРХІТЕКТУРА СИСТЕМИ	11
1.1 Еволюція систем моніторингу	11
1.2 Архітектурна типологія монолітних систем	15
1.3 Революція мікросервісів: Порівняльний аналіз розподілених систем	19
1.4 Наступний крок еволюції: Serverless Architecture	24
1.5 Методологія автоматизованого виявлення аномалій	34
2 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ МОНІТОРИНГУ	48
2.1 Проектування архітектури сховища даних	48
2.2 Генерування набору даних	53
2.3 Реалізація алгоритмів детекції	56
2.4 Розробка мікросервісу автоматичного реагування	63
2.5 Результати тестування та демонстрація роботи системи	68
2.6 Висновки до другого розділу	74
3 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ТА ОПТИМІЗАЦІЯ СИСТЕМИ	76
3.1 Структурний аналіз складових часу відгуку	76
3.2 Оцінка ефективності запитів на тестовому наборі даних	78
3.3 Дослідження та оптимізація частоти сповіщень	79
3.4 Висновки до третього розділу	81
Висновки	82
Список використаної літератури	84
Додаток А	86
Додаток Б	87

ВСТУП

Успіх будь-якої складної системи, як наприклад, керування автомобілем, чи налаштування реклами залежить від швидкості реакції на непередбачувані події. Коли водій бачить перешкоду, він тисне на гальма за долі секунди. Але як діє бізнес, коли його рекламна кампанія раптово починає витрачати бюджет безрезультатно через технічний збій?

На жаль, у більшості випадків реакція є надто повільною. Традиційний підхід до моніторингу нагадує керування автомобілем, дивлячись лише в дзеркало заднього виду: маркетологи аналізують звіти за минулий день, коли гроші вже втрачено. Цей метод, відомий як "пасивна аналітика", чудово підходить для стратегічного планування, але абсолютно безсилий перед миттєвими загрозами. Людський фактор лише погіршує ситуацію: менеджер фізично не може цілодобово стежити за сотнями метрик, особливо вночі чи у вихідні.

Саме тут виникає потреба в зміні архітектурного підходу. Якщо раніше ми будували важкі монолітні системи звітності, то сьогодні нам потрібні легкі, автономні та швидкі системи. Цю роль ідеально виконує мікросервісна архітектура. На відміну від громіздкого сервера, який працює постійно і коштує дорого, мікросервіси діють як мережа незалежних агентів: вони "прокидаються" лише для перевірки, миттєво виявляють аномалію, сповіщають власника і знову "засинають".

Ця робота присвячена створенню саме такої системи активної безпеки. Ми відходимо від питання "Як показати дані?" до питання "Як попередити втрати?". Використовуючи хмарні технології Google та алгоритми автоматичного контролю, ми створюємо інструмент, який дозволяє бізнесу спати спокійно, довіривши рутину код

1 КОНЦЕПЦІЯ ТА АРХІТЕКТУРА СИСТЕМИ

1.1 Еволюція систем моніторингу

Ефективність бізнесу в сфері цифрового маркетингу прямо пов'язана зі швидкістю обробки інформації. Ретроспективний аналіз методів аналітики дозволяє простежити еволюцію від статичних таблиць до систем реального часу та обґрунтувати необхідність впровадження сучасної мікросервісної архітектури. Цей перехід демонструє відмову від пасивного спостереження на користь активного реагування на інциденти.

На початку становлення інтернет-реклами фахівці працювали з невеликими обсягами інформації. У той період стандартом аналітики залишалися табличні процесори, зокрема Microsoft Excel, їхнього функціоналу цілком вистачало для того, щоб у ручному режимі розраховувати базові метрики ефективності, такі як вартість кліка (CPC) або показник клікабельності (CTR).

Проте ситуація змінилася з появою технології торгів у реальному часі[12]. Коли кожне завантаження веб-сторінки перетворилося на миттєвий аукціон, обсяги почали суттєво зростати, та Excel швидко досяг своєї технологічної стелі. Проблема полягає не лише у фізичних обмеженнях софту, а й у людському факторі, маркетолог фізично не здатен вручну зводити звіти з тією швидкістю, з якою оновлюються дані, це і призвело до переходу до першого рівня автоматизації, описової аналітики.

На цьому етапі виникла потреба не просто в калькуляторі, а в інструментах, здатних відповідати на питання "що сталося?" шляхом історичної агрегації великих масивів. Описова аналітика стала фундаментом для розуміння минулого контексту: вона трансформувала хаос сирих логів у впорядковані метрики, дозволяючи ретроспективно оцінювати результати кампаній, хоча й із затримкою в часі. Втім, це був лише проміжний крок, який потребував відповідної візуалізації для зручного сприйняття людиною.

Технологічною відповіддю на потребу в ефективній описовій аналітиці стала масова інтеграція систем Business Intelligence, таких як Looker Studio, Power BI та

Tableau. Цей етап еволюції був детально опрацьований та проаналізований в рамках попереднього кваліфікаційного дослідження, бакалаврської роботи.

У ході попереднього дослідження було спроектовано систему візуалізації показників ефективності, яка довела, що BI-дашборди успішно вирішують проблему агрегації даних. Вони автоматизують критично важливі процеси ETL (Extract, Transform, Load), консолідуючи статистику з розрізнених джерел (Google Ads, Meta Ads) у єдиний інтерактивний інтерфейс. Це дозволило менеджерам отримувати цілісну картину ефективності без необхідності ручного зведення звітів, однак, як буде показано далі, залишило невирішеною проблему оперативного реагування на критичні інциденти.

Досвід практичної експлуатації системи, розробленої в рамках бакалаврської роботи, виявив її фундаментальне архітектурне обмеження в контексті оперативного управління, пасивність. Дашборди за своєю природою є інструментами, що працюють за моделлю Pull-взаємодії: ініціатива завжди залишається за користувачем. Система не кличе на допомогу, а лише мовчки відображає стан справ, очікуючи, поки аналітик увійде в інтерфейс і помітить відхилення.

Така пасивність породжує проблему інформаційної затримки, створюючи небезпечний часовий розрив між виникненням інциденту, будь то технічний збій чи аномальний стрибок вартості кліка, та реакцією на нього. Цей проміжок часу, протягом якого бізнес може втрачати кошти, описується формулою часу реакції (*Reaction_Time*):

$$Reaction_Time = Data_Latency + Analysis_Latency + Decision_Latency \quad (1.1)$$

де кожен компонент робить свій внесок у затримку:

- *Data Latency* — технічний час обробки та оновлення даних, у класичних дашбордах це часто відбувається за розкладом, наприклад, раз на добу.
- *Analysis Latency* — час, необхідний менеджеру, щоб фізично відкрити дашбورد і виявити проблему серед масиву графіків.

- Decision Latency — час, який витрачається на прийняття рішення щодо виправлення ситуації.

Візуальне порівняння сукупного часу затримки при використанні традиційних методів та пропонованого підходу наведено на рисунку 1.1.

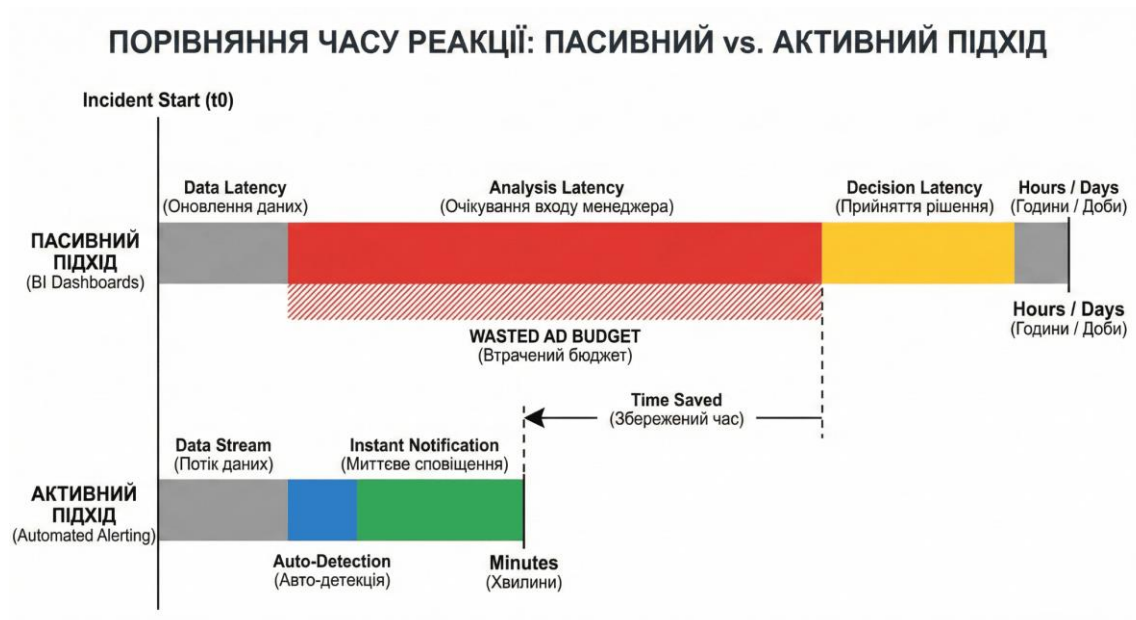


Рисунок 1.1 – Порівняльний аналіз часу реакції у пасивних та активних системах моніторингу

Рекламні алгоритми працюють у режимі 24/7, тоді як менеджер фізично не здатен забезпечити цілодобовий нагляд за монітором. Це створює небезпечну сліпу зону: критичні інциденти, наприклад, нічний витрати бюджету через атаку ботів або допущені помилки в рекламних кампаніях, часто виявляють лише на початку наступного робочого дня.

Таким чином, в сучасних умовах просто бачити графік вже недостатньо, тому і необхідно виконувати перехід від реактивної моделі, вирішення наслідків, до проактивної, попередження проблем. Саме це зумовлює необхідність розвитку від описової аналітики, яка була фокусом моєї бакалаврської роботи, до систем автоматизованого сповіщення, розробка яких є предметом цього дослідження.

Наступним логічним кроком еволюції стає впровадження автоматизованого сповіщення. В основі цього підходу лежить перехід до моделі Push-взаємодії, що змінює роль програмного забезпечення, система перетворюється з пасивного

оглядача минулого на активного учасника управлінського процесу.

Замість пасивного очікування, поки користувач увійде в інтерфейс, мікросервісний комплекс бере ініціативу на себе. Він автономно аналізує потоки даних у реальному часі та виходить на зв'язок, через API або месенджери), виключно тоді, коли фіксує статистично значущі відхилення.

Такий підхід дозволяє вирішити три критичні завдання, з якими не здатні впоратися класичні дашборди:

1. Подолання втоми від моніторингу. Нівелюється вплив людського фактору, коли увага оператора притупляється через необхідність постійно переглядати однотипні звіти. Система фільтрує шум, залишаючи лише важливі сигнали.
2. Автономність 24/7. На відміну від менеджера, автоматизовані алгоритми не залежать від біологічних ритмів. Вони здатні ідентифікувати аномалію о 03:00 ночі з тією ж ефективністю та швидкістю, що й у розпал робочого дня.
3. Радикальне скорочення латентності. Часовий проміжок від виникнення інциденту до отримання сповіщення в Telegram скорочується з кількох годин, або навіть діб, до лічених хвилин, що є критичним для збереження бюджету.

Концептуальна відмінність між класичним підходом та запропонованою системою активних сповіщень наочно представлена на рисунку 1.2.

Таким чином, перехід до автоматизованого алертингу є логічним продовженням попередніх досліджень. Ми трансформуємо систему з оглядача, що лише відображає минуле, рівень бакалаврської роботи, в активного помічника, що працює в реальному часі. Це змінює сам принцип використання даних, замість пасивного накопичення статистики для ретроспективного аналізу, система отримує здатність миттєво впливати на бізнес-процеси. Якщо раніше ключовим критерієм була зручність візуалізації, то тепер на перший план виходять швидкість детекції та мінімізація фінансових ризиків.

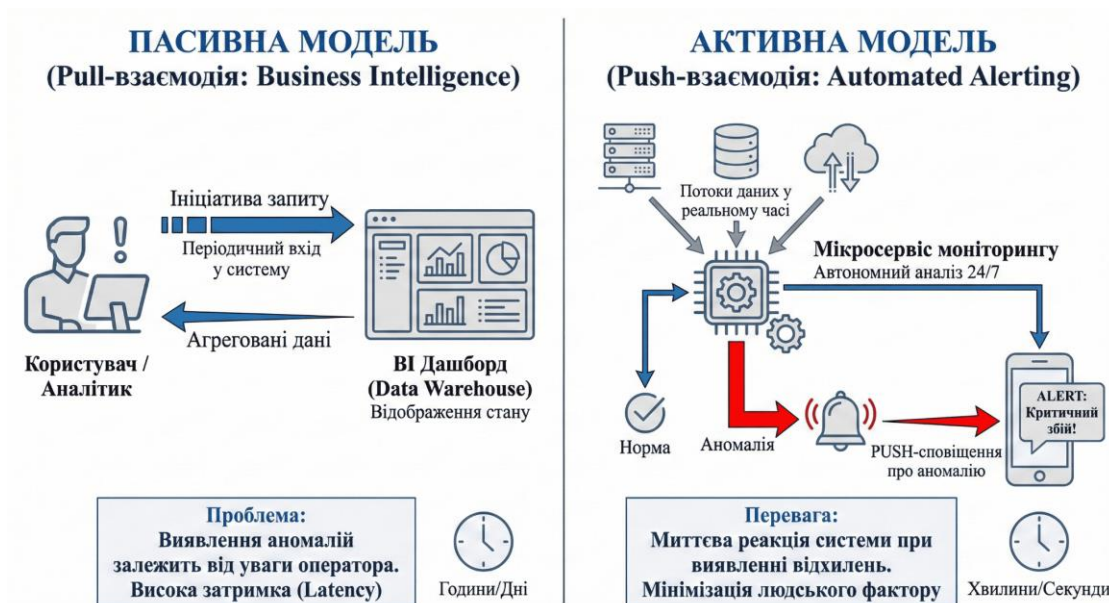


Рисунок 1.2 – Порівняння моделей взаємодії (Pull vs Push)

Однак реалізація такої моделі висуває нові вимоги до програмної архітектури. Щоб зрозуміти, як технічно забезпечити режим 24/7 та миттєву реакцію, необхідно спершу проаналізувати еволюцію архітектурних стилів від традиційних монолітів до гнучких мікросервісів, що і буде розглянуто в наступному підрозділі.

1.2 Архітектурна типологія монолітних систем

Розгляд того, що індустрія потребує переходу від пасивних дашбордів до активних систем реагування приводить нас до питання технічної реалізації. Щоб обґрунтувати вибір мікросервісів та безсерверних технологій для задач моніторингу, необхідно спершу розглянути початкову точку, а саме монолітні системи.

Сьогодні архітектуру моноліту часто сприймають як щось застаріле, але насправді для багатьох систем, особливо тих що не мають надмірної складності, монолітна архітектура часто є хорошим вибором. Моноліт - це єдина сутність, де інтерфейс користувача, складна бізнес-логіка та механізми доступу до даних співіснують в одному процесі виконання та розгортаються як єдина система. Щоб зрозуміти логіку подальшої міграції до мікросервісів, розглянемо детальніше

градацію монолітних патернів.

Першим і найбільш поширеним патерном організації коду є Шаруватий моноліт, що базується на принципі горизонтального технічного поділу обов'язків. Архітектурна логіка тут має чітку ієрархію: система поділяється на рівень представлення, де обробляються запити, шар бізнес-логіки та шар доступу до даних.

Такий підхід здається впорядкованим, адже кожен шар відповідає за свій технічний аспект, проте на практиці приводить до структурних проблем. Як показано на рисунку 1.3, впровадження навіть однієї функціональної зміни, наприклад, реалізація механізму знижок, вимагає наскрізного втручання у систему. Розробник змушений синхронно модифікувати інтерфейс, логіку контролера, сервісні класи та схему бази даних. Така висока зв'язаність створює суттєві перешкоди для масштабування, наприклад, у командній роботі це призводить до конфліктів при злитті коду, оскільки різні розробники змушені змінювати одні й ті самі файли спільних шарів.

Саме ця нездатність структури підлаштовуватись вертикальним змінам змушує переглянути підходи до організації моноліту, виникає потреба групувати код не за технічною функцією, а за бізнес-змістом, що підводить нас до концепції Модульного моноліту.

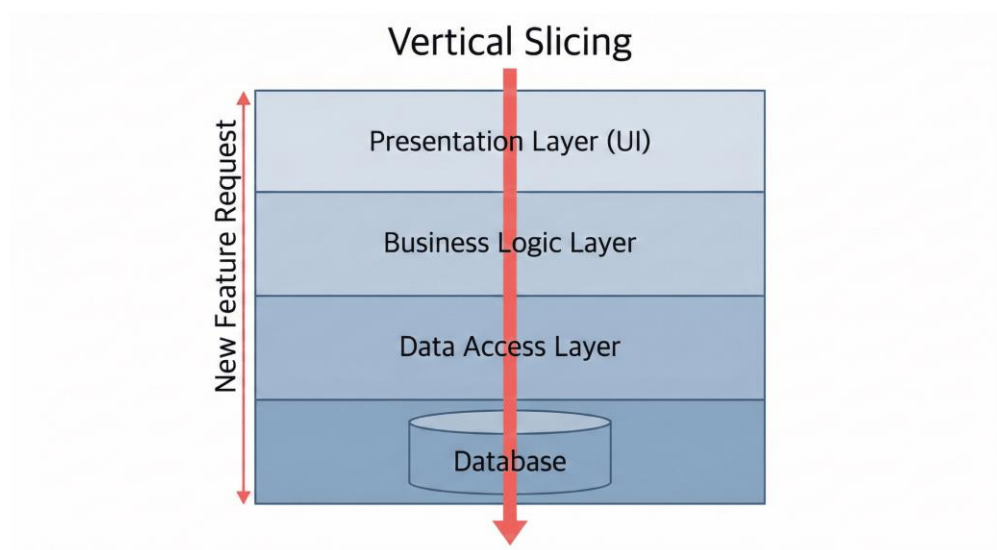
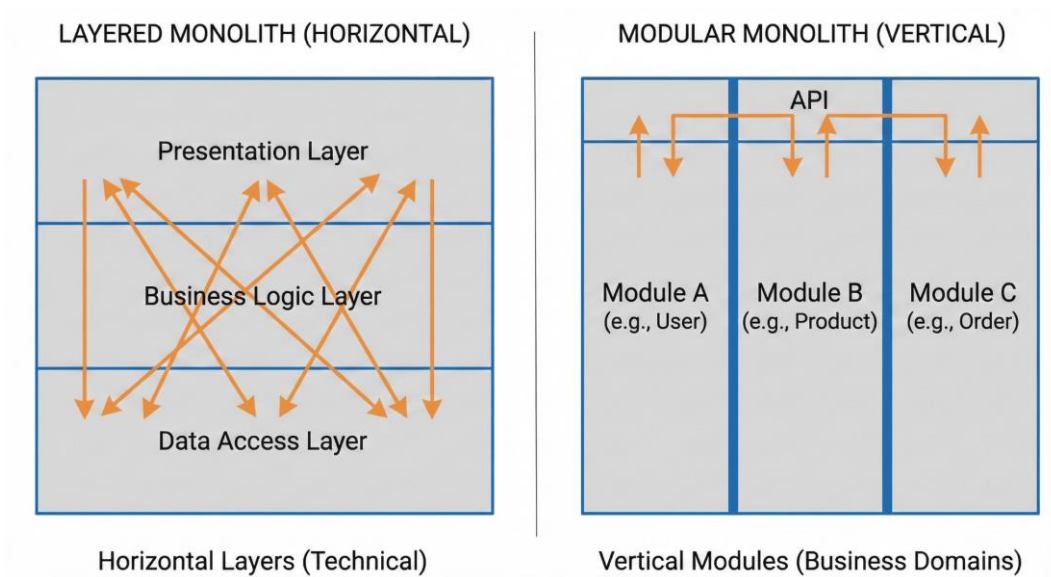


Рисунок 1.3 – Проблема наскрізних змін у шаруватій архітектурі

Модульний моноліт (Modular Monolith). Реакцією на структурні вади шаруватого підходу та водночас на складність розподілених систем стала поява концепції Модульного моноліту, який зберігає головну перевагу класичного моноліту – простоту розгортання, але змінює принцип внутрішньої організації коду, якщо шаруватий моноліт ділить систему горизонтально, то модульний застосовує вертикальний поділ. Архітектура будується навколо бізнес-доменів, замість папок контролерів та сервісів, проект структурується за функціональними модулями інвентаризації, оплати, доставки, тощо (рисуюнок 1.4).



Рисуюнок 1.4 – Порівняння принципів декомпозиції монолітів

Основними принципами такої організації є:

- Жорстка інкапсуляція. Кожен модуль є самодостатньою одиницею з власною внутрішньою архітектурою, інші модулі не знають як саме реалізована логіка всередині.
- Чіткі контракти. Комунікація між модулями відбувається виключно через публічні API, це означає що код з модуля «Оплата» не має права напряду звертатися до внутрішніх класів модуля «Інвентаризація» – він може лише викликати дозволені методи публічного інтерфейсу.
- Ізоляція даних. Модулі не повинні мати спільних таблиць у базі даних. Це запобігає виникненню прихованої зв'язності на рівні даних, яка в подальшому може стати причиною неможливості розділення системи.

Модульний моноліт забезпечує готовність переходу системи до мікросервісів, але не змушує стикатися з складностями розподілених обчислень, тобто проблем з мережевими затримками, узгодженістю даних тут і зараз. Модульний моноліт це оптимальна стартова точка, адже система залишається керованою в одному репозиторії, доки розмір команди або навантаження не вимагатимуть фізичного розділення .

Проте, навіть ідеально структурований модульний моноліт не гарантує безпроблемного переходу до мікросервісів. Найбільшою загрозою на цьому етапі стає створення розподіленого моноліту. Це стан системи, який виникає, коли архітектори зосереджуються на фізичному розділенні коду, ігноруючи необхідність розриву логічних зв'язків, в такому випадку система набуває всієї операційної складності розподіленого середовища (затримки мережі, складність налагодження), але залишається такою ж неповороткою та взаємозалежною, як і старий код.

Розпізнати цей антипатерн можна за трьома критичними симптомами:

- Спільна база даних. Кілька сервісів продовжують звертатися до однієї схеми даних, як результат, зміна структури таблиці в одному модулі неминуче ламає роботу інших, перетворюючи базу даних на масивний вузол зв'язності.
- Пастка синхронності. Комунікація будується на жорстких ланцюжках синхронних HTTP-викликів (Сервіс А викликає В, В викликає С). Це створює ефект доміно: якщо падає одна ланка, падає весь ланцюжок. Проблеми з доступністю такої системи різко знижують загальну стабільність .
- Lock-Step Deployment. Через тісні залежності команда втрачає можливість оновлювати сервіси ізольовано, доводиться координувати релізи і деплоїти всю групу сервісів одночасно, що повністю нівелює головну бізнес-перевагу мікросервісів, а саме незалежність розробки та швидкий вихід на ринок.

Узагальнюючи розгляд монолітних патернів, можна виділити три ключові

сценарії, які визначають подальший вектор розвитку системи:

- Шаруватий моноліт є точкою відліку, яка через жорстку зв'язність швидко вичерпує свій ресурс ефективності та блокує швидкі зміни.
- Модульний моноліт – це необхідний етап чистоти коду. Він дозволяє навести лад у бізнес-логіці та окреслити межі контекстів, проте все ще обмежений рамками одного процесу виконання і єдиної точки відмови.
- Розподілений моноліт є архітектурним глухим кутом, який демонструє, що просте фізичне розділення серверів без зміни логіки взаємодії призводить лише до зростання хаосу.

Цей аналіз підводить нас до критичного висновку: щоб побудувати систему активного моніторингу, здатну працювати в режимі 24/7 і миттєво масштабуватися під навантаженням (як описано в розділі 1.1), нам необхідно вийти за межі монолітної архітектури, потрібно перетворити логічні межі (досягнуті в модульному підході) на фізичні. Однак, враховуючи ризики розподіленого моноліту та складність управління інфраструктурою, класичні мікросервіси можуть бути надто важким рішенням. Тому наступним еволюційним кроком стає пошук архітектури, яка б забезпечувала гранулярність мікросервісів, але прибирала операційну складність управління серверами. Цей пошук приводить нас до концепції безсерверних обчислень, детальний аналіз якої буде проведено далі.

1.3 Революція мікросервісів: Порівняльний аналіз розподілених систем

Якщо розглянуті вище варіації моноліту фокусувалися на тому, як організувати код всередині одного процесу, то перехід до мікросервісів змінює саму фізику системи. Ми виходимо за межі єдиного простору виконання, перетворюючи внутрішньопроектні виклики на мережеву взаємодію, компоненти системи отримують фізичну сегрегацію, що обіцяє небачену раніше гнучкість, але натомість привносить хаос розподілених систем. Суть мікросервісної архітектури полягає не в розділі програми на шматки, а в створенні колекції слабозв'язаних сервісів, згрупованих навколо реальних бізнес-можливостей[1, 4].

Головним фактором тут є незалежність розгортання. Це пряма протилежність

моноліту, щоб оновити логіку в одному маленькому модулі, більше не потрібно перезбирати та перезапускати весь величезний комплекс. Система отримує здатність еволюціонувати частинами, а не лише цілком.

Мікросервіси є технічним відображенням Закону Конвея, який постулює, що архітектура софту неминуче повторює структуру комунікацій у компанії. Якщо моноліт цементує ієрархію, то мікросервіси уможливлюють справжню децентралізацію, даючи малим крос-функціональним командам повний контроль над своїм продуктом, але разом з тим, спрацьовує закон збереження складності, вона не зникає, а просто мігрує. Ми спрощуємо код всередині сервісу, але натомість отримуємо зростання складності в інфраструктурі та зв'язках між сервісами, ця проблема стає новим головним болем архітекторів.

На етапі становлення мікросервіси зазвичай спілкувалися синхронно (REST/gRPC). Це створювало проблему зв'язності, щоб система працювала, клієнт і сервер мали бути онлайн одночасно, у складних системах це призводило до ефекту доміно, коли збій одного сервісу ламав усіх, хто від нього залежав.

Відповіддю індустрії стала Подійно-орієнтована архітектура. Використання брокерів повідомлень дозволило розірвати часову залежність, забезпечивши справжню автономність компонентів.

У рамках Подійно-орієнтованої архітектури сформувалися два полярні підходи до управління процесами:

- Хореографія. Децентралізований підхід, де кожен сервіс реагує на події автономно, це дає максимальну гнучкість, але ускладнює розуміння глобального стану, адже часом важко відповісти, де саме застрягло замовлення.
- Оркестрація. Використання центрального координатора (наприклад, AWS Step Functions), який керує послідовністю дій. Хоча сервіси стають менш автономними, ми отримуємо повний контроль над складними бізнес-транзакціями, уникаючи хаосу.

У гонитві за модульністю архітектори іноді породжують нано-сервіси, у класичній інфраструктурі вони вважаються небезпечним антипатерном, проблема

суто економічна, накладні витрати перевищують корисність. Кожен мікроскопічний сервіс вимагає налаштування CI/CD, логування та мережі. Якщо бізнес-логіка займає 50 рядків, а конфігурація — 200, система стає неефективною, додати сюди мережеву латентність і система стає непридатною для реального часу.

Однак поява безсерверної архітектури змінила правила гри. Коли платформа бере на себе всю оркестрацію, нано-сервіс, окрема функція, раптово стає ефективним будівельним блоком[16].

Ми успішно вирішили проблему запутаного коду в монолітах, встановивши жорсткі фізичні межі. Але згідно із законом збереження складності, хаос не зник, він просто мігрував на рівень інфраструктури. Граф залежностей між сервісами стає настільки складним, що жодна людина не здатна осягнути його повністю. Замість викликів методів тепер необхідно жонглювати мережевими маршрутами, балансувальниками та політиками безпеки. Помилка в конфігурації YAML-файлу може спричинити непередбачувані наслідки для сервісу, що знаходиться за п'ять мережових етапів від джерела проблеми.

Це призвело до нового виклику, втоми від Kubernetes. Розробники, замість створення продукту, змушені витрачати непропорційно багато часу на боротьбу з інфраструктурою. Саме ця ситуація формує запит на архітектурне рішення, яке б зберегло переваги декомпозиції, але зняло б з команди тягар управління інфраструктурою.

Криза інфраструктури та надмірна операційна складність, описані вище, підводять нас до наступного етапу еволюції розподілених систем, а саме безсерверної архітектури, головною метою якої є зменшення рутинної важкої роботи, що не приносить бізнес-цінності. У цій моделі поняття фізичного сервера, кластера чи контейнера повністю абстрагується провайдером. Розробник перестає бути сумісним адміністратором і фокусується виключно на бізнес-логіці у вигляді функцій, що запускаються лише за вимогою.

Щоб зрозуміти глибину цього зсуву, порівняємо зони відповідальності у класичних мікросервісах та Serverless.

Мікросервіси: Базовою одиницею є контейнер. Масштабування зазвичай

реактивне, система додає нові екземпляри, коли зростає споживання CPU або пам'яті, що має певну інерцію та затримку.

Безсерверна архітектура: Одиницею є окрема функція або обробник події. Масштабування відбувається миттєво на основі кількості вхідних запитів (HTTP) або розміру черги, дозволяючи обробляти раптові піки навантаження без попереднього розігріву.

1. Економічна модель та поведінка при простої

- **Мікросервіси:**

Працюють за моделлю Always-On. Ви повинні тримати мінімум один запущений екземпляр сервісу для очікування запитів, оплачуючи зарезервовану потужність, навіть якщо система простоює вночі.

- **Безсерверна архітектура:**

Використовує модель Scale-to-Zero. Якщо трафіку немає, інфраструктура повністю вимикається, і витрати падають до нуля. Оплата стягується виключно за фактичний час виконання коду, що критично важливо для AdTech з його нерівномірним навантаженням.

2. Операційна відповідальність

- **Мікросервіси:**

Вимагають значного адміністративного ресурсу. Команда відповідає за оновлення операційної системи контейнерів, патчі безпеки, налаштування мережі кластера та балансування навантаження.

- **Безсерверна архітектура:**

Реалізує концепцію NoOps. Вся рутинна робота з управління залізом та операційними системами абстрагується та делегується хмарному провайдеру, дозволяючи розробнику фокусуватися лише на бізнес-логіці.

3. Головний архітектурний компроміс

- **Мікросервіси:**

Основним викликом є операційна складність. Архітектура забезпечує повний контроль над середовищем, але спеціаліст витрачає час на

боротьбу з конфігураціями.

- Безсерверна архітектура:

Основним викликом є холодний старт (затримка при першому запуску) та тісна прив'язка коду до специфічних сервісів провайдера, як наприклад AWS Lambda чи Google Cloud Functions, що ускладнює потенційну міграцію.

Візуалізація відмінностей у підходах до споживання ресурсів між розглянутими моделями наведена на рисунку 1.5.

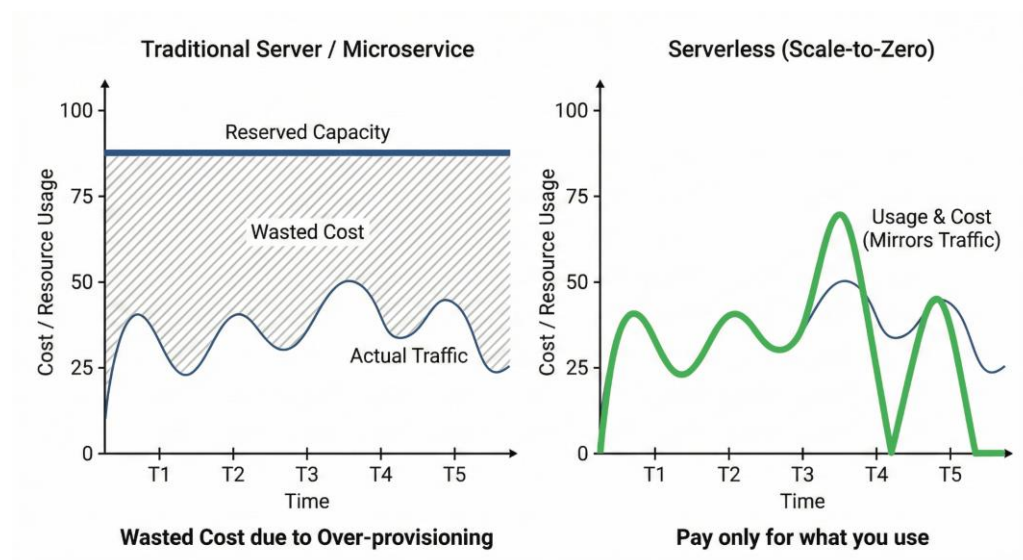


Рисунок 1.5 – Порівняння моделей споживання ресурсів

Безсерверна архітектура є подійно-орієнтованою, відповідно якщо в класичних мікросервісах асинхронність треба налаштовувати вручну, то тут функції мають інтеграцію з чергами та потоками даних, що значно спрощує архітектуру.

Цей підхід також створює компроміси, найвідомішим викликом є холодний старт, тобто затримка при першому виклику функції, необхідна для ініціалізації середовища. Хоча сучасні техніки пом'якшують цей ефект, для систем наднизької латентності це залишається фактором ризику. Інший ризик це те що високий рівень абстракції приводить до тісної прив'язки коду до екосистеми конкретного провайдера, що робить потенційну міграцію складнішою порівняно зі стандартними Docker-контейнерами.

Підсумовуючи еволюційний шлях від моноліту до Serverless, ми бачимо рух по спіралі, де індустрія постійно шукає баланс між гнучкістю розробки та операційною складністю:

- **Моноліт:** Вирішував проблему простоти, але страждав від спагеті-коду.
- **Мікросервіси:** Вирішили проблему коду через модульність, але створили спагеті-інфраструктуру та ризик розподіленого моноліту.
- **Serverless:** Обіцяє приховати інфраструктурну складність платформи, повертаючи фокус інженера на бізнес-логіку.

Візуальна схема цього розвитку представлена на рисунку 1.6.

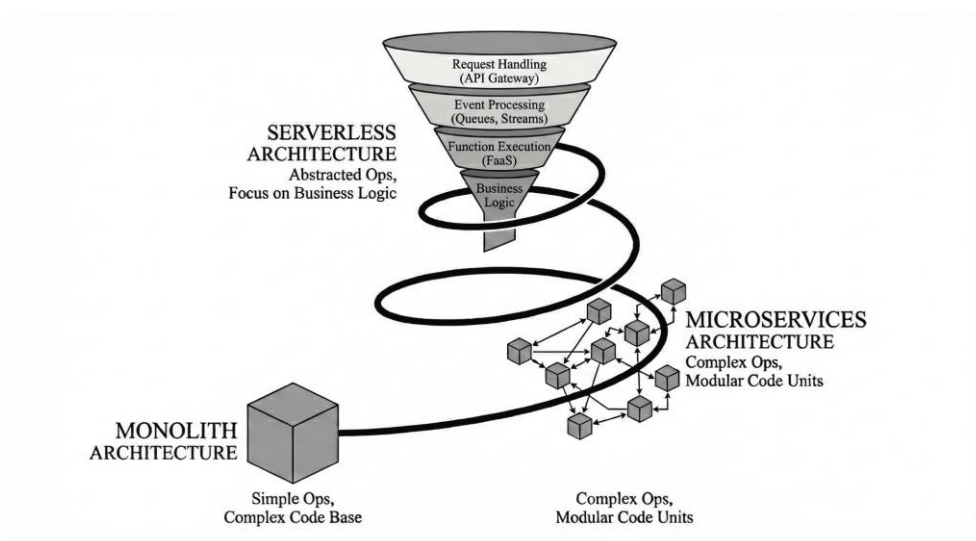


Рисунок 1.6 – Еволюційна спіраль архітектур

Для даного магістерського дослідження критично важливо розуміти, що ідеальної архітектури не існує. Вибір має базуватися на специфіці домену. Враховуючи вимоги до обробки нерівномірного трафіку в AdTech та необхідність швидкої розробки силами невеликої команди, саме безсервісна архітекстура виглядає найбільш релевантним рішенням.

Детальний аналіз конкретної реалізації цієї архітектури на базі Google Cloud Platform та обґрунтування вибору технологічного стеку буде проведено у наступному розділі.

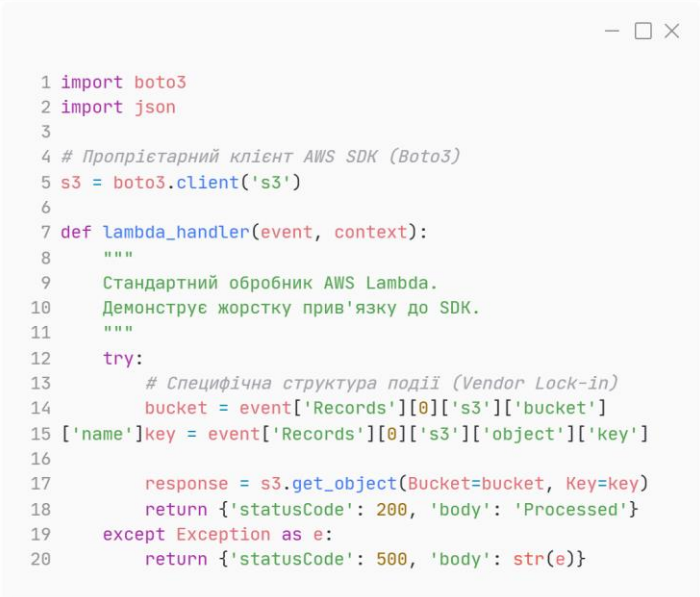
1.4 Наступний крок еволюції: Serverless Architecture

Попередній аналіз еволюції розподілених систем підводить нас до висновку, що для побудови системи моніторингу оптимальним архітектурним стилем є Serverless, для реалізації магістерського дослідження необхідно здійснити обґрунтований вибір технологічної платформи, який би задовольняв специфічні вимоги AdTech-індустрії: обробку пікових навантажень, мінімізацію затримок та глибоку інтеграцію з аналітичними сховищами даних[5, 8].

Станом на 2025 рік ринок хмарних обчислень характеризується олігопольною консолідацією навколо трьох гіперскейлерів: AWS, Microsoft Azure та Google Cloud. Хоча на маркетинговому рівні всі провайдери декларують схожу модель FaaS (Function-as-a-Service), детальний аналіз виявляє фундаментальну дивергенцію в їхніх архітектурних парадигмах. Відмінності полягають не лише в синтаксисі, а й у базових принципах ізоляції навантаження, моделях конкурентності та філософії управління інфраструктурою.

AWS Lambda, представлена у 2014 році, стала де-факто стандартом індустрії, сформувавши ринок безсерверних обчислень. Базуючись на власній технології мікровіртуалізації Firecracker, AWS забезпечує еталонний рівень ізоляції та безпеки. Масштаб екосистеми Lambda є безпрецедентним: вона інтегрована з понад 200 сервісами AWS, що робить її вибором за замовчуванням для більшості Enterprise-рішень.

Проте, статус піонера створив для AWS ефект архітектурної інерції. Платформа накопичила значний рівень операційної складності, що стає бар'єром для швидкої розробки проекту. Специфіку розробки під AWS ілюструє рисунок 1.7, де очевидна залежність від бібліотеки Boto3 та пропрієтарної структури подій. Це призводить до сильної зв'язаності коду з інфраструктурою, що не лише ускладнює локальне тестування, але й створює ризики довгострокової прив'язки до вендора, роблячи міграцію економічно нерентабельною.



```

1 import boto3
2 import json
3
4 # Пропрієтарний клієнт AWS SDK (Boto3)
5 s3 = boto3.client('s3')
6
7 def lambda_handler(event, context):
8     """
9     Стандартний обробник AWS Lambda.
10    Демонструє жорстку прив'язку до SDK.
11    """
12    try:
13        # Специфічна структура події (Vendor Lock-in)
14        bucket = event['Records'][0]['s3']['bucket']
15        ['name']key = event['Records'][0]['s3']['object']['key']
16
17        response = s3.get_object(Bucket=bucket, Key=key)
18        return {'statusCode': 200, 'body': 'Processed'}
19    except Exception as e:
20        return {'statusCode': 500, 'body': str(e)}

```

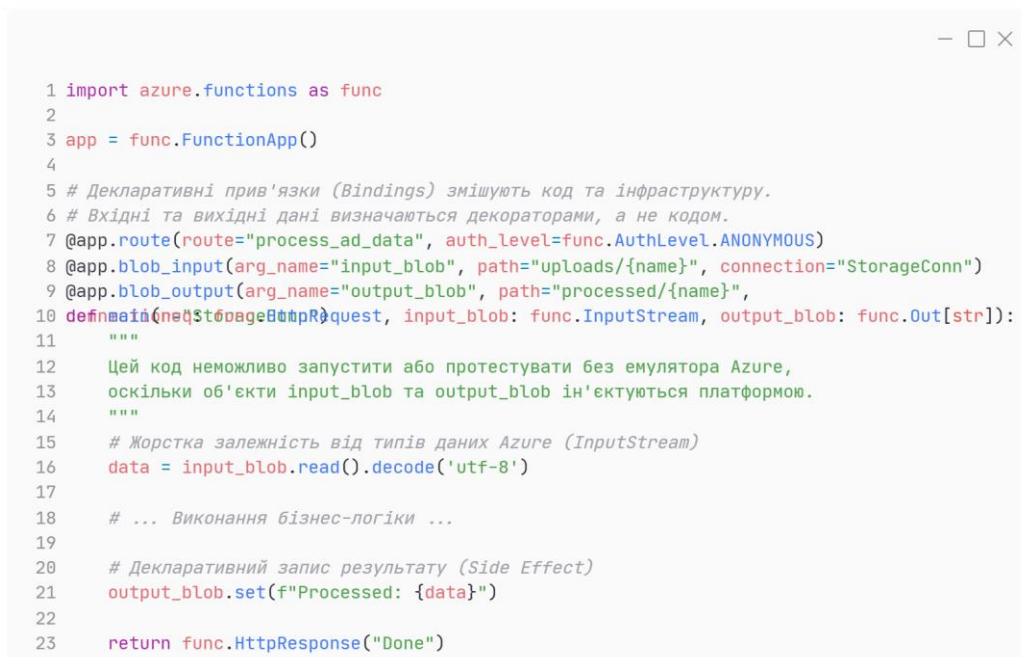
Рисунок 1.7 – Приклад обробника AWS Lambda. Демонстрація жорсткої прив'язки до SDK (Boto3) та специфічної структури події

- Надмірна гранулярність IAM: Модель безпеки AWS вимагає ручного налаштування складних політик та ролей навіть для тривіальних операцій, що непропорційно збільшує час на конфігурацію середовища порівняно з написанням бізнес-логіки.
- Жорстка модель ізоляції: Класична архітектура Lambda сповідує принцип «один запит — одна мікро-віртуальна машина». Це гарантує безпеку, але призводить до неефективного використання ресурсів при I/O-bound навантаженнях та провокує часті «холодні старти», боротьба з якими вимагає додаткових платних опцій.

Головним конкурентом AWS виступає Microsoft Azure, чия платформа Azure Functions позиціонується як природний вибір для Enterprise-сектору. Цей інструмент демонструє високу ефективність насамперед в організаціях із глибокою інтеграцією в екосистему Microsoft (Active Directory, Office 365, .NET Framework), що дозволяє будувати безшовні гібридні архітектури.

Унікальною рисою Azure Functions є концепція Bindings. На відміну від імперативного підходу AWS (де розробник пише код для підключення до БД), Azure пропонує декларативну модель. Вхідні дані та вихідні результати

описуються в конфігураційному JSON-файлі або через декоратори коду. Це суттєво зменшує обсяг шаблонного коду, але створює ефект сильної зв'язності із платформою. Логіка стає настільки залежною від специфічних абстракцій Azure, що перенесення її на іншого провайдера вимагає фактично повного переписування. Як показано на рисунку 1.8, декларативна модель Bindings зміщує інфраструктурну конфігурацію з кодом, ускладнюючи його ізольоване тестування.



```

1 import azure.functions as func
2
3 app = func.FunctionApp()
4
5 # Декларативні прив'язки (Bindings) змішують код та інфраструктуру.
6 # Вхідні та вихідні дані визначаються декораторами, а не кодом.
7 @app.route(route="process_ad_data", auth_level=func.AuthLevel.ANONYMOUS)
8 @app.blob_input(arg_name="input_blob", path="uploads/{name}", connection="StorageConn")
9 @app.blob_output(arg_name="output_blob", path="processed/{name}",
10                  connection="StorageConn")
11 def main(req: func.HttpRequest, input_blob: func.InputStream, output_blob: func.Out[str]):
12     """
13     Цей код неможливо запустити або протестувати без емулятора Azure,
14     оскільки об'єкти input_blob та output_blob ін'єктуються платформою.
15     """
16     # Жорстка залежність від типів даних Azure (InputStream)
17     data = input_blob.read().decode('utf-8')
18
19     # ... Виконання бізнес-логіки ...
20
21     # Декларативний запис результату (Side Effect)
22     output_blob.set(f"Processed: {data}")
23
24     return func.HttpResponse("Done")
  
```

Рисунок 1.8 – Функція Azure з декларативними прив'язками. Бізнес-логіка залежить від типів даних, визначених платформою

Попри технологічну зрілість, використання Azure Functions у контексті даного дослідження виявляє низку критичних недоліків:

- Латентність холодного старту: Порівняльні бенчмарки стабільно демонструють, що оптимізація Azure сфокусована на нативному стеку C#/.NET. Для інтерпретованих мов (Python, Node.js), які використовуються в нашому проєкті, час ініціалізації контейнера на плані Consumption часто перевищує показники конкурентів, що є ризиком для real-time моніторингу.
- Економічний бар'єр: Щоб нівелювати проблему «холодних стартів», Azure пропонує перехід на Premium Plan із зарезервованими інстансами.

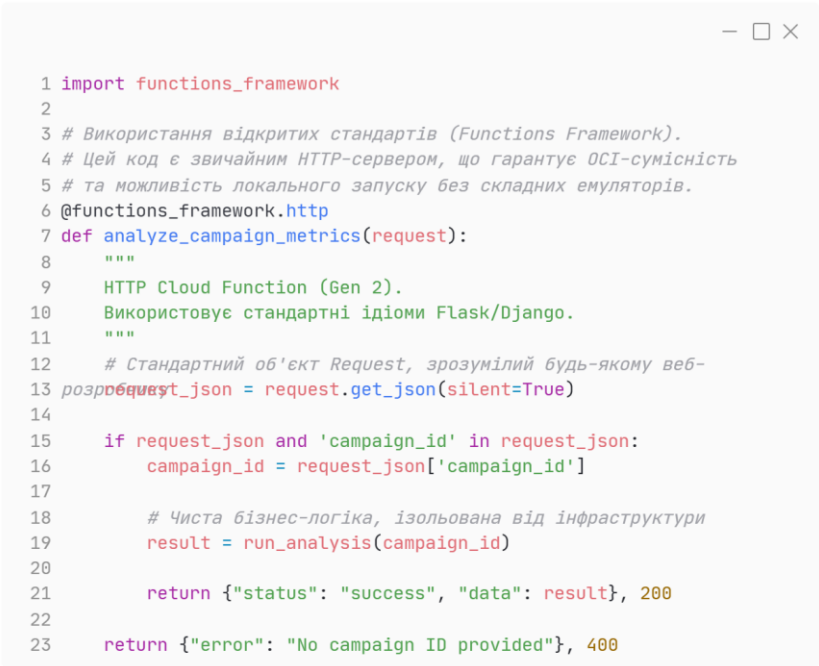
Однак це змінює модель оплати з чистого pay-per-use на постійні витрати, що суперечить вимогам до бюджетної ефективності студентського проєкту та MVP-розробки.

У ландшафті хмарних обчислень Google Cloud Functions історично займали нішу полегшеного інструменту, орієнтованого на виконання допоміжного коду для інтеграції сервісів. Проте з релізом другого покоління (Gen 2) парадигма платформи зазнала радикальної трансформації, перейшовши від простого скриптингу до повноцінного середовища виконання корпоративного рівня.

Технологічна конвергенція: Knative як фундамент Ключовим фактором, що забезпечив цей стрибок, стала архітектурна уніфікація GCF із платформою Cloud Run. В основі Gen 2 лежить відкритий стандарт Knative для Kubernetes. Це означає, що функція більше не є замкнутим «чорним ящиком» пропрієтарної інфраструктури. Технічно, кожен деплой функції створює стандартний OCI-сумісний контейнер. Такий підхід нівелює головний недолік класичних FaaS, обмеженість середовища виконання. Дослідник отримує можливість використовувати кастомні бінарні залежності та системні бібліотеки (наприклад, для складних математичних обчислень або обробки медіа), зберігаючи при цьому простоту розгортання, притаманну безсерверним функціям[9].

Google обрав стратегію "золотої середини", фокусуючись не на нарощуванні кількості нішевих сервісів, а на Developer Experience та уніфікації. Для завдань даного магістерського дослідження, яке вимагає інтенсивної роботи з I/O-операціями (запити до API та баз даних), критичною перевагою стає модель конкурентності, успадкована від Cloud Run. Вона дозволяє одному екземпляру функції ефективно обробляти сотні паралельних запитів, мінімізуючи накладні витрати на масштабування[10, 19].

Підхід GCP, ілюстрований на рисунку 1.9, дозволяє писати функції, використовуючи стандартні ідіоми Python та HTTP, що робить код зрозумілим та переносимим.



```

1 import functions_framework
2
3 # Використання відкритих стандартів (Functions Framework).
4 # Цей код є звичайним HTTP-сервером, що гарантує OCI-сумісність
5 # та можливість локального запуску без складних емуляторів.
6 @functions_framework.http
7 def analyze_campaign_metrics(request):
8     """
9     HTTP Cloud Function (Gen 2).
10    Використовує стандартні ідіоми Flask/Django.
11    """
12    # Стандартний об'єкт Request, зрозумілий будь-якому веб-
13    розробнику
14    request_json = request.get_json(silent=True)
15
16    if request_json and 'campaign_id' in request_json:
17        campaign_id = request_json['campaign_id']
18
19        # Чиста бізнес-логіка, ізольована від інфраструктури
20        result = run_analysis(campaign_id)
21
22        return {"status": "success", "data": result}, 200
23
24    return {"error": "No campaign ID provided"}, 400

```

Рисунок 1.9 – Google Cloud Function (Gen 2). Використання стандартних HTTP-сигнатур, що забезпечує портативність коду (OCI-сумісність)

Вибір Google Cloud Platform (GCP) як технологічного фундаменту є прагматичним інженерним рішенням, продиктованим не ринковою часткою, а архітектурною перевагою другого покоління функцій (Gen 2). Стратегічна уніфікація Cloud Functions та Cloud Run на базі відкритого стандарту Knative забезпечує баланс між простотою FaaS та гнучкістю контейнерів, що є критичним для створення MVP силами невеликої команди.

На відміну від пропрієтарних середовищ (як-от ранні версії AWS Lambda), GCF Gen 2 розгортає функцію як стандартний OCI-сумісний контейнер. Це нівелює ризик прив'язки до вендора: артефакт, створений для Google Cloud, може бути без змін розгорнутий у будь-якому кластері Kubernetes (наприклад, локально через Minikube або в AWS EKS). На рисунку 1.10 продемонстровано, що код функції залишається чистим і не містить специфічних хмарних залежностей.

```

1 import functions_framework
2
3 # Використання відкритого стандарту (Functions Framework).
4 # Цей код не має жорстких прив'язок до Google Cloud і може
5 # працювати локально або в Kubernetes (OCI-сумісність).
6 @functions_framework.http
7 def analyze_campaign_metrics(request):
8     """
9     Обробка запиту через стандартний Flask-style інтерфейс.
10    """
11    # Стандартний метод отримання JSON, ідентичний Flask
12    request_json = request.get_json(silent=True)
13
14    if request_json and 'campaign_id' in request_json:
15        # Чиста бізнес-логіка, повністю ізольована від
16        інфраструктури return {"status": "success", "data": "processed"}, 200
17
18    return {"error": "Bad Request"}, 400

```

Рисунок 1.10 – Стандартна структура функції (Python), яка є звичайним HTTP-сервером і не залежить від пропрієтарних бібліотек хмари

Визначальним фактором вибору стала модель обробки запитів. У класичних FaaS діє правило ізоляції, один запит дорівнює одному екземпляру. Це вкрай неефективно для I/O-bound задач, наприклад, очікування відповіді від API чи БД, оскільки під час очікування ресурси процесора простоюють, а система змушена запускати нові «холодні» інстанси для обробки паралельних запитів.

GCF Gen 2 використовує модель конкурентності інстансу, дозволяючи одному контейнеру обробляти до 1000 запитів одночасно. У середовищі Node.js або Python це дозволяє ефективно утилізувати Event Loop, обслуговуючи сотні з'єднань одним процесом. Це радикально зменшує кількість холодних стартів та навантаження на пули підключень до бази даних.

Важливо також зазначити, що архітектурно Cloud Functions 2-го покоління побудовані на базі Cloud Run та Knative, що надає розширені можливості керування інфраструктурою. Така модель дозволяє оптимізувати витрати: замість оплати за час виконання кожного окремого виклику, тарифікується час роботи активного інстансу, який обслуговує пакет запитів паралельно. Це усуває проблему надлишкового виділення пам'яті при пікових навантаженнях, коли в

систему надходить велика кількість сигналів моніторингу одночасно. Завдяки цьому забезпечується стабільна пропускна здатність без ефекту тротлінгу та мінімізується загальна латентність доставки алертів, що є критичним показником для систем реального часу[15]. Порівняння архітектурних підходів та їх вплив на потік даних наочно відображено на рис. 1.11.

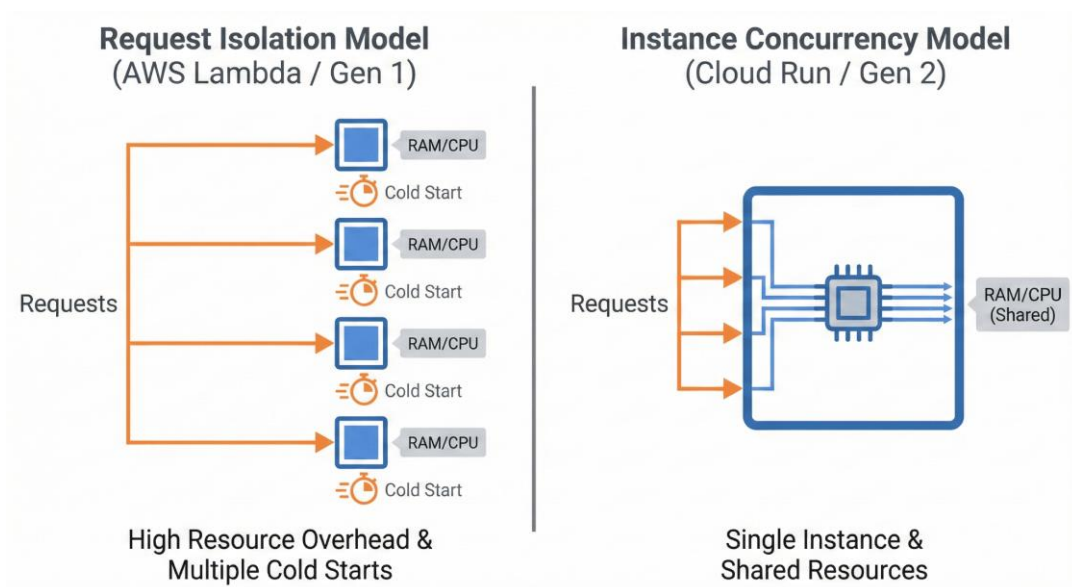


Рисунок 1.11 – Порівняння моделей виконання: ізоляція запиту проти конкурентності

Операційна простота та Developer Experience (DX)

В умовах обмежених ресурсів дослідницького проекту швидкість налаштування інфраструктури є критичним економічним фактором. Порівняльний аналіз демонструє суттєву перевагу Google Cloud над AWS у зручності інструментарію:

- Спрощена модель безпеки (IAM): На відміну від AWS, де налаштування вимагає маніпуляцій зі складними JSON-політиками та Trust Relationships, Google використовує інтуїтивну модель Сервісних акаунтів. Надання ролей безпосередньо на ресурси, наприклад, BigQuery Data Editor для конкретного датасету, мінімізує ризик конфігураційних помилок.
- Локальна розробка: Бібліотека Functions Framework дозволяє запускати

функції локально нативно, забезпечуючи повну ідентичність середовища з хмарою. Це дозволяє тестувати код миттєво, без необхідності розгортання важких Docker-контейнерів для емуляції, як це реалізовано в AWS SAM CLI.

- Zero-Code безпека: Використання Identity-Aware Proxy дозволяє захистити HTTP-ендпоінти корпоративною авторизацією Google на рівні інфраструктури, усуваючи потребу в написанні коду для валідації токенів.

Нативна інтеграція з екосистемою даних

Оскільки проєкт є частиною пайплайну обробки даних, а не ізольованим модулем, критичним фактором стає глибина інтеграції обчислювального шару зі сховищами даних та інтерфейсами користувача. Екосистема GCP пропонує архітектурні патерни, що дозволяють мінімізувати кількість проміжного коду.

Для реалізації асинхронної взаємодії GCF Gen 2 використовує уніфіковану шину Eventarc, що базується на відкритому стандарті CNCF CloudEvents. Це дозволяє будувати слабкозв'язані архітектури, де функція реагує на зміни інфраструктури без необхідності реалізації механізмів постійного опитування. Як приклад, завантаження файлу в Cloud Storage автоматично генерує стандартизовану JSON-подію, яка тригерить функцію обробки. Інтеграція відбувається на рівні конфігурації інфраструктури, а не коду.

Унікальною можливістю GCP є технологія Remote Functions, яка дозволяє викликати Cloud Function безпосередньо з SQL-запиту. Це змінює підхід до обробки даних з класичного ETL (вивантаження → обробка → завантаження) на ELT (Extract, Load, Transform). Складна бізнес-логіка, наприклад, шифрування, збагачення даних через зовнішні API, інкапсулюється у функцію і викликається рушієм бази даних синхронно. Таке рішення дозволяє розширювати стандартний синтаксис SQL довільним кодом без залучення проміжних серверів. Схему цього процесу та взаємодію компонентів наведено на рис. 1.12.

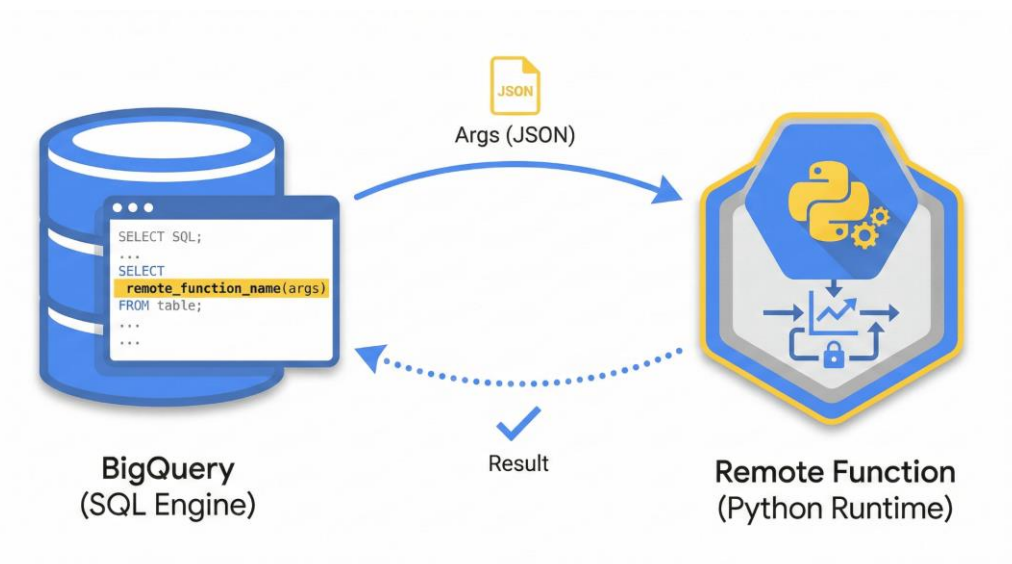


Рисунок 1.12 – Схема роботи BigQuery Remote Functions: делегування обчислювальної логіки з SQL-середовища до Cloud Functions

Як продемонстровано на рисунку 1.13, завдяки Remote Functions аналітик може інтегрувати складну логіку обробки безпосередньо в запит, не створюючи проміжних скриптів на Python чи Java.

```

1  -- Реєстрація функції як частини SQL-синтаксису
2  CREATE OR REPLACE FUNCTION `dataset.enrich_customer_data`(user_id STRING)
3  RETURNS STRING
4  REMOTE WITH CONNECTION `projects/my-project/locations/us/connections/my-conn`
5  OPTIONS (
6    endpoint = 'https://us-central1-my-project.cloudfunctions.net/enrich-
7  func'
8
9  -- Використання: Дані не покидають контур BigQuery, обробка "на льоту"
10 SELECT
11   user_id,
12   -- Виклик віддаленої функції як звичайної UDF
13   `dataset.enrich_customer_data`(user_id) AS enriched_profile
14 FROM `dataset.raw_logs`
15 WHERE event_date = CURRENT_DATE();

```

Рисунок 1.13 – Виклик бізнес-логіки безпосередньо з аналітичного шару

Оскільки інтерфейсом взаємодії з користувачем виступають Google Sheets, використання нативної інфраструктури Google забезпечує дві переваги:

- Мінімальна латентність: Обчислювальні вузли GCF та API

Google Sheets знаходяться в одній глобальній мережі (Google Backbone), що суттєво пришвидшує обмін даними порівняно з доступом із зовнішніх хмар.

- **Безпека:** Використання Application Default Credentials дозволяє авторизувати запити до Workspace API, використовуючи ідентичність сервісного акаунту функції. Це усуває необхідність управління довгоживучими токенами та зберігання JSON-ключів у коді, що є стандартом безпеки Zero Trust.

Підсумовуючи порівняльний аналіз FaaS-платформ, можна стверджувати: вибір Google Cloud Functions у якості обчислювального ядра системи є стратегічно, технічно та економічно виваженим рішенням. Цей вибір базується на чотирьох ключових причинах:

- **Стратегічна відкритість:** Платформа побудована на базі відкритих стандартів Knative/Cloud Run. Це дозволяє проектувати систему відповідно до сучасних Cloud-Native практик, мінімізуючи ризики жорсткої прив'язки до вендора, що є показником якісної архітектури.

- **Технічна ефективність:** Модель конкурентності інстансу забезпечує оптимальну продуктивність для I/O-bound операцій, дозволяючи обробляти сотні запитів одним контейнером без зайвих витрат ресурсів, властивих моделі ізоляції AWS Lambda. Крім того, нативна інтеграція з Eventarc та BigQuery Remote Functions відкриває унікальні можливості для побудови безсерверних пайплайнів даних.

- **Економічна доцільність:** Модель оплати Pay-as-you-go у поєднанні з механізмом масштабування до нуля робить архітектуру фактично безкоштовною на етапі розробки та пілотування[11, 14].

- **Експлуатаційна простота:** Високий рівень Developer Experience та інтуїтивна модель безпеки дозволяють мінімізувати час на налаштування інфраструктури. Це дає можливість сфокусувати обмежені людські ресурси на головному, реалізації бізнес-логіки та дослідницьких алгоритмів.

Таким чином, Google Cloud Functions дозволяє реалізувати складну

розподілену систему силами невеликої команди в стислі терміни, повністю відповідаючи вимогам до кваліфікаційної роботи магістра.

Визначившись із технологічним стеком, у наступному розділі ми перейдемо до розробки прикладної методології виявлення аномалій у цьому специфічному середовищі.

1.5 Методологія автоматизованого виявлення аномалій

Вибір Google Cloud Functions та BigQuery як технологічного ядра відкриває дозволяє впровадження більш ефективної методології моніторингу. Перехід до Serverless дозволяє відмовитися від ресурсоємної моделі постійного нагляду, натомість, ми отримуємо можливість реалізувати легковаговий архітектурний патерн Check-and-Alert, де перевірки виконуються лише за потребою, тобто у відповідь на подію або за розкладом.

Такий підхід не лише радикально знижує експлуатаційні витрати, адже ми не платимо за простій агентів моніторингу, а й дозволяє змістити фокус з інфраструктурних метрик, які в хмарі керуються провайдером, на бізнес-показники. В рекламній екосистемі, де бюджети витрачаються за мілісекунди, саме швидкість виявлення аномалій стає критичним фактором конкурентоспроможності .

Для побудови надійної системи алертингу критичні інциденти класифіковано за чотирма ключовими векторами[2]:

- **Технічна доступність:** Найкритичнішим сценарієм тут є інцидент з відсутністю показів, коли активна кампанія з наявним бюджетом раптово припиняє показувати рекламу. Це свідчить про розрив технічного ланцюжка, від помилок у тегах на сайті до збою на стороні DSP-платформи або відхилення креативів модерацією.
- **Фінансова безпека:** Ця категорія охоплює інциденти типу перевитрати та недійсних кліків. Система відстежує аномалії швидкості витрат, які можуть сигналізувати як про помилку в налаштуванні лімітів, так і про атаку бот-мереж, здатних вичерпати місячний бюджет безрезультатно.

- **Цілісність контенту:** Типовим прикладом є помилки HTTP 404/500 на цільових сторінках, у цьому випадку рекламна платформа продовжує списувати кошти за кліки, але користувач не отримує послугу, а бізнес втрачає конверсії. Крім прямих збитків, це призводить до репутаційних втрат, оскільки алгоритми Google/Meta песимізують акаунти з низькою якістю посадкових сторінок.
- **Ефективність:** Сюди відносяться раптові падіння відносних показників, таких як CTR (відсоток клікабельності) або Conversion Rate (коефіцієнт конверсії). Це найскладніша для автоматичної детекції група, оскільки вимагає від алгоритму здатності відокремлювати реальні технічні проблеми від природного шуму, зумовленого, наприклад, сезонністю.

Візуалізацію запропонованої класифікації інцидентів наведено на рис. 1.14.

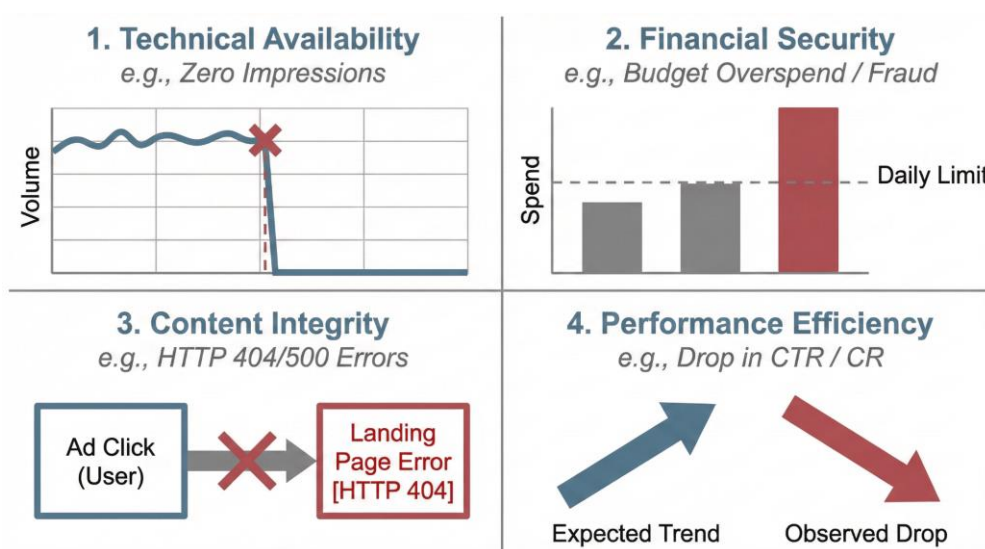


Рисунок 1.14 – Типологія критичних інцидентів у рекламних технологіях

Інцидент типу нульових показів можемо класифікувати як критичний стан системи. Це ситуація, коли кампанія формально є активною, але фактично не генерує жодного показу реклами. Для бізнесу це означає, що маркетинговий канал раптово замовк без явних попереджень. Технічно, цей стан свідчить про розрив ланцюжка постачання реклами, який може статися на трьох рівнях:

- **Рівень даних.** Часто проблема криється не у відсутності показів як таких, а у їхній невидимості для системи аналітики. Некоректна інтеграція

пікселів на стороні паблішера призводить до того, що реклама демонструється, але подія не реєструється в BigQuery.

- Рівень аукціону. Помилка економічної логіки, якщо в налаштуваннях стратегії ставок встановлено ціну CPM, ціну за тисячу показів, нижче ринкового порогу, алгоритм RTB технічно бере участь в аукціонах, але програє 100% з них. Часто це стається після автоматичної міграції налаштувань або при різкому посиленні конкуренції.
- Рівень платформи. Найбільш категоричною причиною є блокування креативів модерацією Google або Meta. Використання забороненого слова чи зображення призводить до миттєвої зупинки трафіку з боку самої рекламної мережі.

Наслідки. В економіці AdTech реальні збитки оцінюються через функцію втрачених можливостей. Якщо кампанія простоє протягом часу $T_{downtime}$ сукупні втрати виторгу L_{opp} розраховуються як інтеграл від очікуваної ефективності системи:

$$L_{opp} = \int_0^{T_{downtime}} (Imp_{exp}(t) \times CTR_{avg} \times CR_{avg} \times AOV) dt \quad (1.2)$$

де:

- $Imp_{exp}(t)$ — функція очікуваного обсягу показів у момент часу t ;
- CTR_{avg} — середній історичний показник клікабельності
- CR_{avg} — середній коефіцієнт конверсії ;
- AOV — середній чек замовлення (Average Order Value).

Для e-commerce бізнесу простій у пікові періоди, наприклад, "Чорна п'ятниця" робить величину L_{opp} критичною для річного фінансового балансу.

Категорія інцидентів Financial Security охоплює сценарії, що призводять до неконтрольованих прямих збитків.

- Механізм виникнення перевитрат. У розподілених системах, якими є сучасні DSP, Demand Side Platforms, існує фундаментальна проблема узгодженості в кінцевому рахунку. Лічильники витрат часто шардовані

між сотнями серверів. Коли кампанія наближається до денного ліміту, інформація про те, що бюджет вичерпано, повинна синхронізуватися між усіма шардами. Часовий лаг цієї синхронізації створює вікно вразливості, протягом якого система продовжує купувати трафік понад ліміт.

- Загроза неякісних розміщень та Click Fraud. Значно небезпечнішим сценарієм є клікфрод, тобто генерація недійсного трафіку бот-мережами. Сучасні боти здатні імітувати поведінку користувачів, створюючи лавиноподібне навантаження на бюджет. Дослідження показують, що частка такого трафіку може сягати 20% від загального трафіку, при цьому ефективність такого трафіку прямує до нуля, що робить його безкорисним активом для рекламодавця.

Традиційний підхід до моніторингу, перевірка перевищення денного ліміту є реактивним, тобто коли бюджет вже витрачено. Проактивний підхід вимагає аналізу швидкості витрат.

Якщо кампанія, яка історично витрачає 10% бюджету за першу годину доби, раптово витрачає 40%, це є сигналом тривоги ще до досягнення ліміту.

Для формалізації цього процесу вводиться поняття коефіцієнта прискорення витрат k_{acc} . Аномалія детектується, якщо поточна швидкість витрат у момент часу t суттєво перевищує історичне середнє

$$k_{acc}(t) = \frac{S'(t)}{S'_{avg}(t)} > T_{threshold} \quad (1.3)$$

де:

- $S'(t) = \frac{ds}{dt}$ миттєва швидкість витрат бюджету за хвилину поточний момент;
- $S'_{avg}(t)$ — очікувана швидкість витрат у цей час доби (на основі історичних даних за останні 7–30 днів);
- $T_{threshold}$ — порогове значення допустимого відхилення (наприклад, 3.0, що означає прискорення у 3 рази).

Бізнес-наслідки. Фінансовий ризик $R_{financial}$ у разі такої атаки не обмежується лише денним бюджетом, оскільки деякі платформи дозволяють овердрафт. Він розраховується як інтеграл від перевитрат:

$$R_{financial} = \int_{t_{start}}^{t_{detect}} (CPC_{avg} \times N_{bots}(t)) dt \quad (1.4)$$

де CPC_{avg} середня вартість кліка, а $N_{bots}(t)$ інтенсивність недійсного трафіку. Без системи миттєвого алертингу ($t_{detect} \rightarrow \infty$) збитки можуть перевищити місячний бюджет за лічені години.

Категорія Content Integrity охоплює інциденти, пов'язані з недоступністю кінцевої точки маршруту користувача, посадкової сторінки. Типовими проявами є помилки протоколу HTTP (404 Not Found, 500 Internal Server Error) або некоректна робота DNS. У контексті рекламних кампаній такі збої представляють собою класичний приклад безрезультатних витрат бюджету, оскільки розривають ланцюжок створення вартості.

Окрім прямих фінансових втрат, такі інциденти запускають механізм негативного зворотного зв'язку. Алгоритми рекламних мереж (зокрема Google Ads та Meta) аналізують поведінку користувачів після кліку. Масове повернення користувачів ($Bounce\ rate \rightarrow 100\%$) інтерпретується системою як низька релевантність оголошення.

Це призводить до зниження показника якості, як наслідок, платформа песимізує акаунт рекламодавця, примусово підвищуючи вартість кліка у майбутніх аукціонах для компенсації низької якості.

Складність детекції полягає в архітектурній ізоляції. Рекламні кабінети не мають прямого доступу до логів веб-сервера рекламодавця. Тому внутрішні звіти Google Ads можуть показувати статус "Active", навіть коли сайт лежить. Єдиним вирішенням є впровадження зовнішньої системи моніторингу, яка емулює перехід реального користувача за рекламним посиланням, перевіряючи HTTP-статус відповіді.

Ризик $R_{integrity}$ у даному випадку є сумою миттєвих втрат та відкладеного штрафу за репутацію. Його можна описати формулою:

$$R_{integrity} = \underbrace{\int_{t_{fail}}^{t_{fix}} (Clicks(t) \cdot CPC_{current}) dt}_{\text{Direct Wasted Spend}} + \underbrace{\sum_{k=1}^N V_{fut}^{(k)} \cdot \Delta CPC_{penalty}}_{\text{Reputation Penalty}} \quad (1.5)$$

де:

- $Clicks(t)$ інтенсивність трафіку під час збою;
- V_{fut} обсяг майбутнього трафіку після відновлення роботи;
- $\Delta CPC_{penalty}$ приріст вартості кліка, зумовлений падінням Quality Score (наприклад, зниження QS з 10 до 7 може підняти ціну кліка на 30-50%).

Ця модель доводить, що ціна 404-ї помилки значно перевищує вартість втрачених кліків, розтягуючись у часі як податок на ненадійність.

Категорія проблем що впливають на ефективність охоплює інциденти, пов'язані з деградацією відносних метрик, таких як CTR чи Conversion Rate[18]. На відміну від технічних збоїв, які мають бінарну природу, тобто система працює або ні, ці показники є стохастичними і залежать від зовнішніх шумів, наприклад, сезонності, погоди, новин, тощо.

Типовими прикладами таких проблем є:

- Падіння CTR: Поступове зниження клікабельності часто свідчить про вигорання креативу. Технічною альтернативою може бути збій, коли замість зображення користувач бачить порожній контейнер.
- Зниження конверсій: Ситуація, коли трафік стабільний ($Clicks > 0$) але продажі припиняються ($CR \rightarrow 0$). Це індикатор розриву бізнес-логіки на сайті, як приклад, зламана форма замовлення або розсинхронізації пікселів, коли аналітика не передає дані назад у рекламний кабінет.

Бюджет витрачається на залучення трафіку, який не конвертується. Втрати L_{perf} накопичуються як різниця між очікуваним та фактичним доходом:

$$L_{perf} = \int_{t_0}^t Clicks(\tau) \cdot (CR_{base} - CR_{actual}(\tau)) \cdot AOV d\tau \quad (1.6)$$

В умовах AdTech використання складних ML-моделей є надлишковим через проблему «чорної скриньки» та високу вартість холодних стартів, тому методологія базується на гібридному підході, що поєднує детермінізм для критичних збоїв та статистику для поведінкових.

Детерміністичний контроль використовується для метрик доступності та фінансів. Алерт генерується при порушенні жорсткого порогу θ за виконання логічних умов C :

$$Alert_{rule}(t) \iff (M(t) \bowtie \theta) \wedge C_{context} \quad (1.7)$$

Приклад логіки для відсутності показів:

$$A_{zero} = (Imp == 0) \wedge (Status == 'Active') \wedge (t \in WorkingHours) \quad (1.8)$$

Для волатильних метрик, клікабельності, ціни за клік та інших застосовується метод порівняння з історичною базою, що фільтрує сезонні шуми. Базова лінія V_{base} розраховується як ковзне середнє за N попередніх тижнів:

$$V_{base}(t) = \frac{1}{N} \sum_{k=1}^N V(t - k \cdot 168h) \quad (1.9)$$

Алерт спрацьовує, якщо відносне відхилення Δ перевищує поріг чутливості ϵ (наприклад, -30%):

$$Alert_{pop}(t) \iff \frac{V_{cur}(t) - V_{base}(t)}{V_{base}(t)} < -\epsilon \quad (1.10)$$

Цей метод дозволяє системі ігнорувати глобальні падіння трафіку, наприклад, уночі або на свята, якщо аналогічний патерн спостерігався у минулих періодах. Такий підхід суттєво знижує рівень інформаційного шуму для

відповідальних менеджерів, мінімізуючи кількість хибних викликів. Крім того, гнучкість налаштування глибини історії N та порогу чутливості надає можливість адаптувати алгоритм під специфіку конкретного джерела трафіку без зміни програмного коду. Загальну структуру запропонованого алгоритму детекції наведено на рис. 1.15.

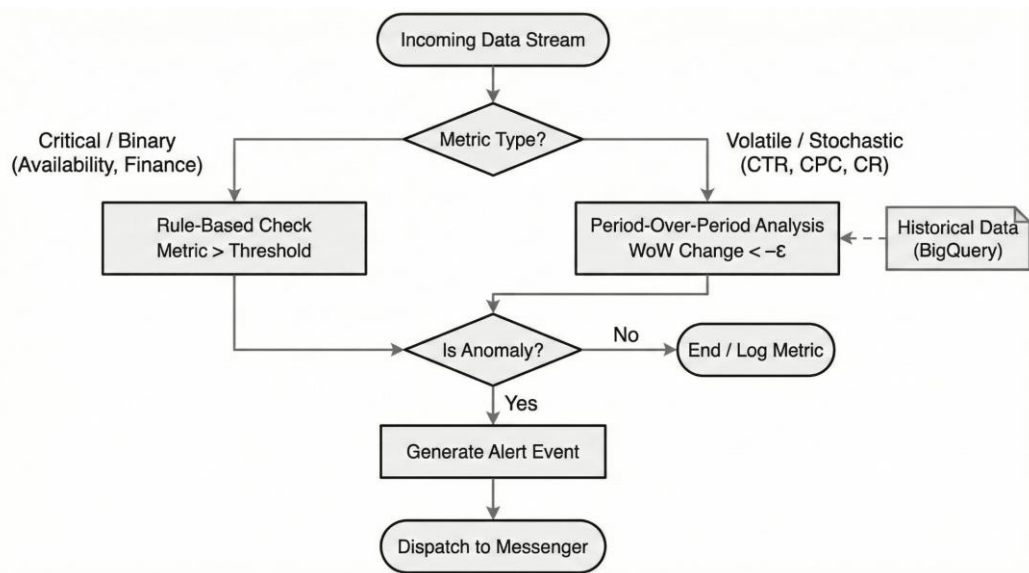


Рисунок 1.15 – Блок-схема розробленого гібридного алгоритму детекції.

Впровадження розробленої методології в екосистему Google Cloud вимагає зміни підходу до процесу моніторингу, ми відмовляємося від класичної архітектури, де агент постійно слухає систему, споживаючи ресурси 24/7. Натомість ми переходимо до подійно-орієнтованого патерну Перевір-і-Сповісти.

Цей вибір продиктований специфікою безсерверних обчислень, у моделі Serverless ми платимо лише за час виконання коду, тому такий підхід дозволяє радикально мінімізувати операційні витрати та забезпечує високу еластичність. Система автоматично масштабується залежно від кількості подій, які необхідно обробити.

Архітектура будується на асинхронній оркестрації трьох компонентів: ініціатора, аналізатора та сповіщувача.

Фундаментальною особливістю Cloud Functions є їхня ефемерність: вони не можуть перебувати в режимі очікування в пам'яті, після виконання контейнер призупиняється або знищується. Тому процес перевірки ніколи не відбувається

сам по собі, він завжди ініціюється зовнішньою подією:

- Регулярні перевірки. Для аналізу трендів і часових рядів ми використовуємо Cloud Scheduler, який генерує CRON-події, наприклад, за розкладом. Це дозволяє регулярно будити функцію для порівняння поточних метрик з історичними даними, що зберігаються в BigQuery .
- Реактивні перевірки. Для миттєвої реакції на зміни в інфраструктурі використовуються механізми Eventarc або Pub/Sub. Типовий сценарій: завершення ETL-процесу завантаження даних у BigQuery автоматично генерує подію, яка ініціює перевірку якості для щойно отриманої партії даних .

Роль перевірки виконує безстанова функція, її життєвий цикл: отримати сигнал, виконати запит, передати результат і завершити роботу. У разі виявлення відхилень функція формує структурований JSON-об'єкт і передає його до підсистеми доставки. Залежно від налаштувань маршрутизації, повідомлення відправляється через API до Slack, PagerDuty або email-шлюзу.

У розподілених системах, побудованих на базі Pub/Sub та Cloud Functions, стандартною гарантією доставки повідомлень є модель "at-least-once". Це створює фундаментальний ризик дублювання: одна й та сама аномалія може викликати декілька паралельних запусків функції. Крім того, існує ризик логічного дублювання, коли регулярні перевірки (наприклад, кожні 15 хвилин) знову і знову виявляють одну й ту ж тривалу проблему.

Без належної фільтрації це призводить до того, що користувач отримує сотні ідентичних повідомлень про одну помилку. Це явище відоме як втома від сповіщень, психологічний стан, коли оператор починає ігнорувати реальних критичних інцидентів[3, 6].

Для вирішення цієї проблеми впроваджується архітектурний патерн ідемпотентності. Операція алертингу вважається ідемпотентною, якщо багаторазове детектування одного й того ж інциденту призводить до відправки лише одного сповіщення користувачеві. Ключовим елементом механізму є генерація унікального ключа інциденту. Він створюється шляхом хешування

інваріантних атрибутів аномалії у поєднанні з дискретним часовим вікном:

$$Key_{idemp} = \text{SHA256}(\text{AccountID} + \text{CampaignID} + \text{ErrorType} + T_{wind} \quad (1.11)$$

де T_{window} функція округлення часу до бажаної частоти нагадувань (наприклад, до години або доби). Це дозволяє системі розрізняти "той самий збій зараз" і "новий збій завтра".

Оскільки архітектура Cloud Functions є безстановою, для перевірки унікальності ключа необхідне використання зовнішнього сховища з низькою латентністю. В екосистемі Google Cloud оптимальним вибором для цієї задачі є Firestore.

Алгоритм дедуплікації реалізує механізм блокування через послідовність атомарних операцій. Спочатку функція виявляє аномалію та обчислює унікальний ідентифікатор Key_{idemp} . Наступним кроком виконується спроба запису цього ключа у сховище з умовою "створити, якщо не існує" до запису додається параметр часу життя, наприклад, 4 години, який діє як період тиші. Успішний запис ініціює відправку сповіщення, тоді як помилка запису свідчить про дублювання і зупиняє процес. Блок-схему цього механізму наведено на рис. 1.16.

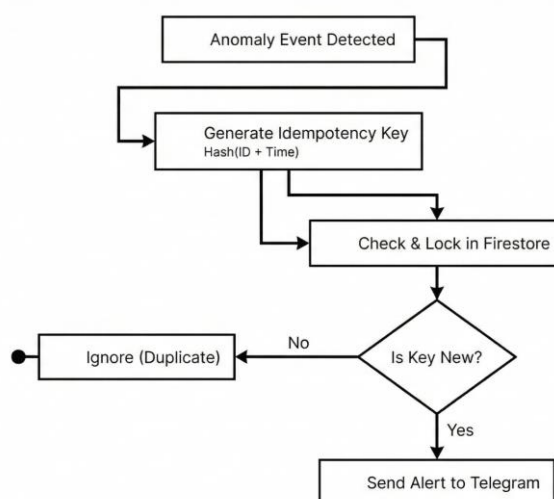


Рисунок 1.16 – Блок-схема процесу дедуплікації алертів. Показано логіку перевірки унікальності інциденту перед відправкою сповіщення в Telegram.

Виявлення аномалії та відсіювання дублікатів є лише половиною шляху. Сирий алерт, наприклад, технічне повідомлення "CampaignID: 12345 | Error: Spend > Limit" є малоінформативним для користувача, для забезпечення керованості процесом реагування, життєвий цикл алерту реалізується через конвеєр:

1. Виявлення: На цьому етапі працюють описані вище математичні алгоритми (Rule-based та Time-Series Comparison). Результатом є внутрішня подія системи "Кандидат на інцидент".
2. Збагачення та Сортуння: Інформація про помилки збагачується контекстом. Функція виконує запит до довідників BigQuery, щоб перетворити абстрактний ID на зрозумілу назву, наприклад, "BlackFriday_Promo" та знайти відповідального менеджера. Одночасно відбувається пріоритезація проблеми в залежності від критичності.
3. Фінальним етапом є інтелектуальна маршрутизація сповіщень залежно від типу інциденту. Технічні збої скеровуються розробникам, бюджетні питання відділу маркетингу, а для критичних подій активується мульти-канальний виклик, що гарантує інформування відповідальних осіб навіть без доступу до месенджерів.

На основі розробленої типології збоїв визначено чотири базові сценарії реагування:

1. Технічна доступність. Найкритичніший сценарій, типовим прикладом якого є інцидент повної зупинки показів. Аномалія виявляється статичним правилом (значення = 0). Оскільки це означає зупинку процесу, для гарантованого сповіщення інженера використовуються канали з високою пріоритетністю
2. Фінансова безпека. Сценарій високої критичності, що охоплює інциденти типу перевитрат бюджету. Детекція базується на аналізі швидкості витрат. Ситуація вимагає негайного адміністративного рішення, тому сповіщення надсилається в корпоративний месенджер з прямим тегуванням менеджера.
3. Цілісність контенту. Сценарій середньої критичності,

наприклад, помилки HTTP 404 на цільовій сторінці. Виявляється через синтетичну перевірку. Проблема призводить до фінансових втрат, але не вимагає втручання вночі, тому оптимальним є нотифікування відповідального, або всієї відповідальної команди, але менш пріоритетними каналами.

4. Ефективність. Інформаційний сценарій, пов'язаний з падінням CTR або конверсії. Виявляється методом статистичного порівняння тижня до тижня[13, 17]. Це сигнал про негативний тренд, а не миттєву поломку, тому сповіщення консолідується у щоденний звіт, щоб не відволікати команду.

Загальну схему даних та логіку вибору каналу наведено на рис. 1.17.

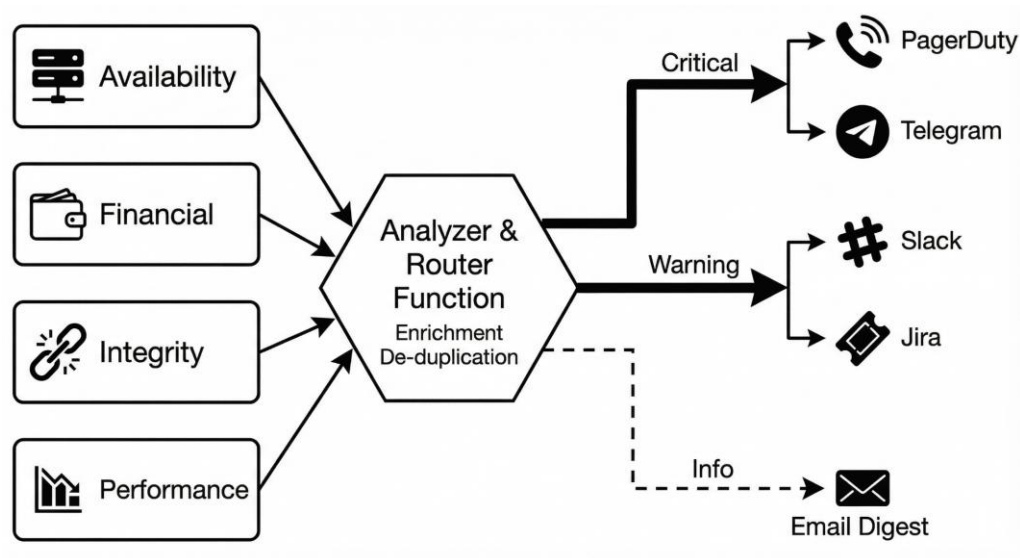


Рисунок 1.17 – Архітектура маршрутизації інцидентів у системі моніторингу

Проведений у першому розділі теоретико-аналітичний огляд дозволив сформуванню цілісної концепції системи автоматизованого моніторингу.

- **Необхідність:** доведено, що в умовах аукціонів традиційні методи пасивної аналітики, дашборди, вичерпали свій ресурс ефективності через проблему високої латентності прийняття рішень. Перехід до активних систем алертингу є безальтернативним етапом подальшого розвитку.
- **Технологічний вибір:** обґрунтовано використання Google

Cloud Functions, як обчислювального ядра. Архітектура на базі контейнеризованих функцій забезпечує необхідну модель конкурентності для I/O-задач та дозволяє реалізувати економічну модель Scale-to-Zero.

- **Методологія детекції:** Розроблено гібридну стратегію виявлення аномалій, яка відмовляється від надмірної складності ML-моделей на користь операційної стійкості. Поєднання детерміністичних правил для фінансових ризиків та статистичного аналізу для поведінкових метрик забезпечує оптимальний баланс між чутливістю та мінімізацією хибних спрацювань.

- **Архітектурне рішення:** Запропоновано патерн Check-and-Alert із вбудованим механізмом ідемпотентності, це вирішує проблему дублювання сповіщень, гарантуючи доставку лише унікальних та насичених контекстом інсайтів.

Сформований теоретичний базис, математичні моделі ризиків та обраний технологічний стек створюють фундамент для наступного етапу дослідження, практичної реалізації та програмної розробки компонентів системи, що буде детально розглянуто у другому розділі.

2 ПРОГРАМНА РЕАЛІЗАЦІЯ СИСТЕМИ МОНІТОРИНГУ

2.1 Проектування архітектури сховища даних

У попередньому розділі було проведено детальний аналіз проблематики моніторингу у сфері цифрової реклами та виявлено ключові недоліки існуючих підходів до контролю трафіку. Як було встановлено, ручний моніторинг та пост-фактум аналіз не здатні забезпечити фінансову безпеку рекламних бюджетів в умовах високочастотних торгів. Часовий лаг між виникненням інциденту та реакцією оператора ризику призвести до значних фінансових втрат.

Виходячи з теоретичного обґрунтування, у цьому розділі ми переходимо до практичної реалізації автоматизованої системи моніторингу. Основна мета спроектувати та розробити програмний комплекс, який здатен обробляти великі масиви даних у реальному часі, виявляти аномалії за допомогою статистичних алгоритмів та миттєво сповіщати відповідальних осіб.

В якості технологічного ядра системи було обрано хмарне сховище даних Google BigQuery[21, 22]. Цей вибір зумовлений трьома критичними архітектурними перевагами, які безпосередньо впливають на ефективність моніторингу:

- Ефективність роботи з широкими таблицями. Специфіка рекламних даних полягає у великій кількості атрибутів: кожен запис про подію може містити понад 50 полів (ID кампанії, географія, тип пристрою, ціна кліка тощо). Класичні бази даних зчитують рядок повністю, навіть якщо для звіту потрібні лише два поля. BigQuery, завдяки колонковій архітектурі, зчитує виключно ті стовпці, які запитує аналітик.
- Фокус на бізнес-логіці. Використання безсерверної архітектури усуває необхідність у ручному налаштуванні інфраструктури, нам не потрібно адмініструвати фізичні сервери, керувати оновленнями або налаштовувати індекси для прискорення пошуку. Система автоматично виділяє необхідні ресурси під кожен конкретний запит, що дозволяє

розробникам зосередитися виключно на написанні алгоритмів детекції аномалій, а не на DevOps-задачах.

- Гнучке масштабування. У BigQuery процеси зберігання даних та їх обробки фізично розділені. Дані лежать у розподіленій файловій системі, а обчислення виконуються окремим кластером, такий підхід надає унікальну перевагу, ми можемо зберігати величезні архіви історичних даних за низькою ціною, але в момент виникнення інциденту миттєво залучати тисячі процесорів для їх аналізу.

Для організації структури даних було відкинуто найпростіший підхід використання однієї пласкої таблиці, де всі дані зберігаються разом. Хоча такий підхід спрощує запис, у масштабах AdTech він є неефективним. Зберігання повної текстової назви кампанії, групи та креативу в кожному рядку зі статистикою, а це мільйони рядків щодня, призвело б до роздування обсягу сховища та невиправданих витрат.

Натомість реалізовано нормалізовану схему "Сніжинка". Це рішення продиктоване двома практичними факторами:

- Дзеркальне відображення бізнес-логіки. Рекламні акаунти Google Ads мають чітку ієрархічну вкладеність: Акаунт → Кампанія → Група → Оголошення. Схема "Сніжинка" дозволяє фізично розділити ці сутності по різних таблицях, зберігаючи логічні зв'язки. Це робить базу даних інтуїтивно зрозумілою для будь-якого аналітика, знайомого з інтерфейсом рекламного кабінету.
- Ефективність оновлення. На практиці маркетологи часто змінюють назви кампаній або налаштування бюджетів. У пласкій таблиці нам довелося б перезаписувати суттєву кількість історичних рядків, щоб оновити одну назву. У схемі "Сніжинка" ми змінюємо лише один запис у невеликій таблиці вимірів (`dim_campaigns`) і ця зміна миттєво стає актуальною для всіх звітів.

Крім того, винесення текстових полів у довідники дозволяє таблиці фактів оперувати лише легкими цілочисельними ключами, що значно прискорює операції

з'єднання при побудові звітів.

Детальна структура розробленої моделі даних включає наступні сутності:

1. Таблиця вимірів `dim_campaigns`, або рівень стратегії. Це коренева сутність ієрархії. Вона містить критично важливі атрибути для фінансового моніторингу, зокрема денний бюджет та статус активності. Саме ця таблиця виступає "джерелом правди" при перевірці інцидентів доступності, система порівнює статус кампанії тут із фактичними показами в таблиці фактів.
2. Таблиця вимірів `dim_ad_groups`, або ж рівень тактики. Забезпечує гранулярність аналізу. Групи оголошень об'єднують креативи за спільною тематикою або типом таргетингу. Виділення цієї сутності в окрему таблицю дозволяє аналізувати ефективність різних сегментів аудиторії без необхідності дублювання параметрів налаштувань для кожного оголошення.
3. Таблиця вимірів `dim_creatives`, рівень контенту. Зберігає дані про самі рекламні матеріали. Ключовим атрибутом тут є поле `format` (Video, Image, Text). Це дозволяє алгоритмам детекції динамічно обирати відповідні метрики для перевірки: наприклад, для відео-креативів активується перевірка VTR (View-Through Rate) та глибини перегляду, тоді як для банерів аналізується лише CTR.
4. Таблиця вимірів `dim_placements`, рівень контексту. Містить інформацію про веб-сайти та мобільні додатки, де відбувалися покази реклами. Ця таблиця відіграє ключову роль у модулі Integrity Checks. Наявність атрибута `risk_level` дозволяє автоматично виявляти випадки показу реклами на ресурсах з "Чорного списку" або сайтах з низькою репутацією, що є ознакою неефективного витрачання бюджету.
5. Таблиця фактів `fact_hourly_stats`, операційний рівень. Центральний елемент схеми, що акумулює кількісні показники з інтервалом в одну годину. Вибір годинної гранулярності є компромісом між деталізацією аналізу та обсягом збережених даних. Таблиця містить зовнішні ключі

для зв'язку з таблицями вимірів.

Взаємозв'язки між сутностями та структуру зовнішніх ключів візуалізовано на логічній схемі даних (рис. 2.1).

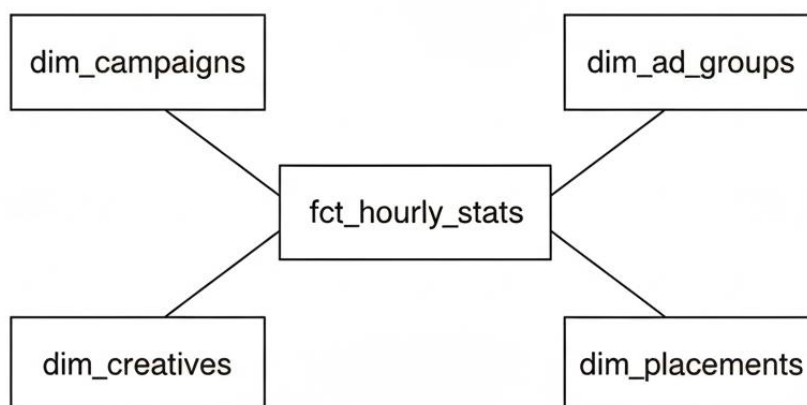


Рисунок 2.1 – Логічна схема даних системи моніторингу

Для оптимізації продуктивності в BigQuery застосовано стратегію секціонування таблиці фактів за датою та кластеризацію за полем `campaign_id`. Це мінімізує обсяг сканованих даних та прискорює агрегацію показників. Сама ж архітектура "Зірка", на відміну від нормалізованих моделей, зменшує кількість необхідних операцій з'єднання, що є критичним для забезпечення швидкодії аналітичних запитів у реальному часі.

Варто також зазначити специфіку забезпечення цілісності даних у BigQuery. Оскільки ця СУБД архітектурно не підтримує жорсткого контролю обмежень зовнішніх ключів, валідація зв'язків покладається на логіку ETL-процесу. Оголошення ключів у наведеній схемі виконує насамперед роль метаданих, що допомагають оптимізатору запитів будувати ефективні плани виконання та коректно інтерпретувати модель у зовнішніх BI-інструментах.

Нижче наведено скрипти для створення структури сховища. На рисунку 2.2 представлено процес створення таблиць вимірів, які формують основу зіркової схеми даних. Ці таблиці містять статичні атрибути сутностей, за якими в подальшому буде здійснюватися аналітичний зріз.

```

-- 1. Вимір: Кампанії (Верхній рівень ієрархії)
-- Зберігає бюджетні ліміти та статуси для перевірок доступності
CREATE OR REPLACE TABLE `diploma_project.dim_campaigns`
(
  campaign_id INT64,
  campaign_name STRING,
  campaign_type STRING, -- Search, Display, Video
  daily_budget FLOAT64,
  status STRING,        -- Active, Paused, Ended
  start_date DATE
);

-- 2. Вимір: Групи оголошень (Середній рівень)
CREATE OR REPLACE TABLE `diploma_project.dim_ad_groups`
(
  ad_group_id INT64,
  campaign_id INT64,    -- Зовнішній ключ до dim_campaigns
  ad_group_name STRING,
  targeting_type STRING -- Keywords, Remarketing, Interests
);

-- 3. Вимір: Креативи (Нижній рівень)
-- Критично важливо для аналізу ефективності відео (VTR)
CREATE OR REPLACE TABLE `diploma_project.dim_creatives`
(
  creative_id INT64,
  ad_group_id INT64,    -- Зовнішній ключ до dim_ad_groups
  creative_name STRING,
  format STRING,        -- Video, Image, Text
  video_duration_sec INT64
);

-- 4. Вимір: Плейсменти (Місця розміщення)
-- Використовується для Integrity Checks (Brand Safety)
CREATE OR REPLACE TABLE `diploma_project.dim_placements`
(
  placement_domain STRING, -- Наприклад: youtube.com
  category STRING,         -- News, Gaming, Adult
  risk_level STRING        -- Safe, High, Blacklist
);

```

Рисунок 2.2 – Створення таблиць вимірів. Визначення структури для кампаній, груп, креативів та плейсмент

Центральним елементом спроектованої схеми є таблиця фактів, структура якої наведена на рисунку 2.3. Вона акумулює транзакційні дані з погодинною деталізацією. Особливістю технічної реалізації є застосування нативних механізмів BigQuery для роботи з великими даними:

- Партиціонування: Розбиття даних за датою (timestamp), що дозволяє сканувати лише потрібні дні, а не всю таблицю, суттєво знижуючи вартість запитів.
- Кластеризація: Сортування даних всередині партиції за campaign_id, що прискорює фільтрацію по конкретних кампаніях.

```

CREATE OR REPLACE TABLE `diploma_project.fct_hourly_stats`
(
  timestamp TIMESTAMP,
  campaign_id INT64,
  ad_group_id INT64,
  creative_id INT64,
  placement_domain STRING,

  -- Основні метрики ефективності (KPIs)
  impressions INT64,
  clicks INT64,
  cost FLOAT64,
  conversions INT64,

  -- Розширені метрики для глибокого аналізу
  viewable_impressions INT64, -- Для перевірки видимості (Viewability)
  video_completions INT64    -- Для відео-кампаній
)
PARTITION BY DATE(timestamp) -- Оптимізація: фізичне розбиття по днях
CLUSTER BY campaign_id;      -- Оптимізація: сортування даних всередині дня

```

Рисунок 2.3 – Створення оптимізованої таблиці фактів. Застосування партиціонування та кластеризації для підвищення продуктивності

2.2 Генервання набору даних

Оскільки реальні історичні дані рекламних акаунтів захищені угодами про нерозголошення, для тестування розробленої системи було створено модуль генерації синтетичних даних.

Для верифікації роботи алгоритмів детекції аномалій було розроблено SQL-скрипт, який генерує синтетичний набір даних з прогнозованими патернами поведінки. Першим кроком є ініціалізація довідника кампаній, представленого на рисунку 2.4. У набір включено різні типи кампаній, що дозволить перевірити універсальність моніторингу, а також закладено "проблемні" кампанії для майбутніх тестів.

Наступним етапом реалізовано генерацію часових рядів метрик ефективності через добуток створених кампаній та календарного періоду. Для наближення синтетичних даних до реальності використано математичні функції рандомізації, які моделюють природний розподіл трафіку та добову сезонність. Критично важливим елементом стала програмна ін'єкція аномалій: скрипт примусово моделює ситуації раптового обриву показів та сплесків вартості конверсії для обраних ID. Це забезпечує наявність гарантованих тригерів,

необхідних для фінальної валідації точності спрацювання системи алертингу.

```
-- КРОК 1. Наповнення довідника кампаній (8 тестових сутностей)
TRUNCATE TABLE `diploma_project.dim_campaigns`;

INSERT INTO `diploma_project.dim_campaigns`
(campaign_id, campaign_name, campaign_type, daily_budget, status)
VALUES
(101, '01_YouTube_InStream_Awareness', 'Video', 100000.0, 'Active'),
(102, '02_YouTube_Bumper_Reach', 'Video', 50000.0, 'Active'),
(103, '03_YouTube_Short_Traffic', 'Video', 75000.0, 'Active'),
(104, '04_Video_Partner_OutStream', 'Video', 60000.0, 'Active'), -- "Проблемна"
(105, '05_YouTube_Conversion_Sales', 'Video', 90000.0, 'Active'),
(106, '06_Display_Remarketing_Hot', 'Display', 55000.0, 'Active'),
(107, '07_Performance_Max_Retail', 'PMax', 120000.0, 'Active'),
(108, '08_App_Install_General', 'App', 80000.0, 'Active');
```

Рисунок 2.4 – Ініціалізація тестового набору рекламних кампаній

Як показано на рисунку 2.5, алгоритм використовує функцію для створення погодинної сітки за два місяці. Ключовим елементом є блок `traffic_simulation`, де програмно закладено Аномалію №2 (Placement Error). Для кампанії з ID 104 у визначену дату (15 листопада) алгоритм перенаправляє 95% трафіку на неякісне місце розміщення, що імітує реальну помилку налаштування таргетингу. Також застосовано синусоїдальну функцію для імітації природних добових коливань активності користувачів.

```
-- КРОК 2. Симуляція трафіку та поведінки
WITH time_series AS (
  -- Генерація погодинної сітки за 2 місяці
  SELECT timestamp FROM UNNEST(GENERATE_TIMESTAMP_ARRAY(
    TIMESTAMP('2024-10-01 00:00:00'), TIMESTAMP('2024-11-30 23:00:00'), INTERVAL 1
    HOUR AS timestamp
  ),
  traffic_simulation AS (
    SELECT
      ts.timestamp, c.campaign_id, c.campaign_name,

      -- АНОМАЛІЯ №2: Токсичний трафік (Placement Error)
      -- 15 листопада трафік кампанії 104 перенаправляється в ігри
      CASE
        WHEN c.campaign_id = 104 AND DATE(ts.timestamp) = '2024-11-15'
          AND RAND() < 0.95
        THEN 'mobile-app::com.random.game.flappy.bird' -- Фрод
        ELSE 'youtube.com' -- Нормальний трафік
      END AS placement_domain,

      -- Синусоїда добової активності (День/Ніч) + Випадковий шум
      (ABS(SIN((EXTRACT(HOUR FROM ts.timestamp) - 6) / 24 * 3.14))) * 0.8 + 0.2
      * (0.8 + 0.4 * RAND()) * c.volume_multiplier AS activity_factor
    FROM time_series ts CROSS JOIN campaign_config c
  )
)
```

Рисунок 2.5 – Алгоритм симуляції часових рядів та програмна реалізація аномалії розподілу трафіку

Завершальним етапом роботи генератора є розрахунок похідних бізнес-метрик на основі базових показів. На рисунку 2.6 продемонстровано реалізацію Аномалії №1 (Black Friday Outage): для кампанії №101 у день розпродажу штучно обнуляються покази після 14:00, що імітує аварійну зупинку.

Крім того, скрипт моделює погіршення якості трафіку: для плеєсментів типу mobile-app (які генеруються під час Аномалії №2) коефіцієнт конверсії примусово встановлюється в 0, а CTR — неприродно високим (15%), що характерно для неякісних розміщень.

```

final_calc AS (
  SELECT *,
    -- Базові покази: Фактор активності * Масштаб
    CAST(activity_factor * 10000 AS INT64) AS raw_impressions,

    -- АНОМАЛІЯ №1: Black Friday Outage (Зупинка)
    -- 28 листопада після 14:00 множимо трафік на 0
    CASE
      WHEN campaign_id = 101 AND DATE(timestamp) = '2024-11-28'
        AND EXTRACT(HOUR FROM timestamp) >= 14 THEN 0
      ELSE 1
    END AS outage_multiplier
  FROM traffic_simulation
)
SELECT
  timestamp, campaign_id, placement_domain,
  -- 1. IMPRESSIONS (з урахуванням зупинки)
  raw_impressions * outage_multiplier AS impressions,

  -- 2. CLICKS (На фродових майданчиках CTR аномально високий - 15%)
  CAST((raw_impressions * outage_multiplier) * IF(placement_domain LIKE 'mobile-app%', 0.15, 0.015) AS INT64) AS
  clicks,

  -- 3. CONVERSIONS (На фродових майданчиках конверсій немає)
  CAST((raw_impressions * outage_multiplier) * IF(placement_domain LIKE 'mobile-app%', 0.0, 0.0005) AS INT64) AS
  conversions
FROM final_calc
WHERE raw_impressions > 0;

```

Рисунок 2.6 – Розрахунок фінальних метрик з урахуванням логіки аварійної зупинки та клік-фроду

2.3 Реалізація алгоритмів детекції

Після наповнення сховища даних синтетичними наборами, що імітують реальну поведінку рекламних кампаній, наступним етапом є розробка програмної логіки для виявлення відхилень. У рамках дипломної роботи не зупиняємось на використанні простих порогових значень як єдиного методу контролю, оскільки такий підхід не є гнучким в умовах динамічного ринку. Натомість реалізовано стратегію, яка передбачає три рівні перевірки даних:

- Детерміністичний контроль. Базовий рівень перевірки, що оперує

однозначною логікою. Він відповідає за виявлення критичних технічних збоїв, які не допускають подвійного трактування, наприклад, ситуація, коли кампанія має статус "Активна" та позитивний баланс, але фізично не отримує жодного показу.

- Евристичний аналіз. Застосування набору експертних правил для ідентифікації відомих схем шахрайства. На цьому рівні система шукає специфічні показники неякісного трафіку, наприклад, комбінацію аномально високого CTR із нульовою конверсією, що є характерною ознакою неефективної витрати бюджету.
- Статистичне моделювання. Найвищий рівень складності, що використовує історичні дані для розрахунку динамічної норми. Замість фіксованих порогів тут застосовуються методи математичної статистики, що дозволяє адаптивно реагувати на відхилення, наприклад, різкий стрибок вартості кліка, враховуючи при цьому час доби та сезонність.

Усі алгоритми реалізовано мовою Google Standard SQL, що дозволяє виконувати перевірки безпосередньо всередині сховища BigQuery, мінімізуючи затримки на передачу даних.

Першим і найбільш критичним класом інцидентів є інцидент відмови в роботі, це ситуація, коли рекламна кампанія є активною і має достатній бюджет, але фактично не отримує показів.

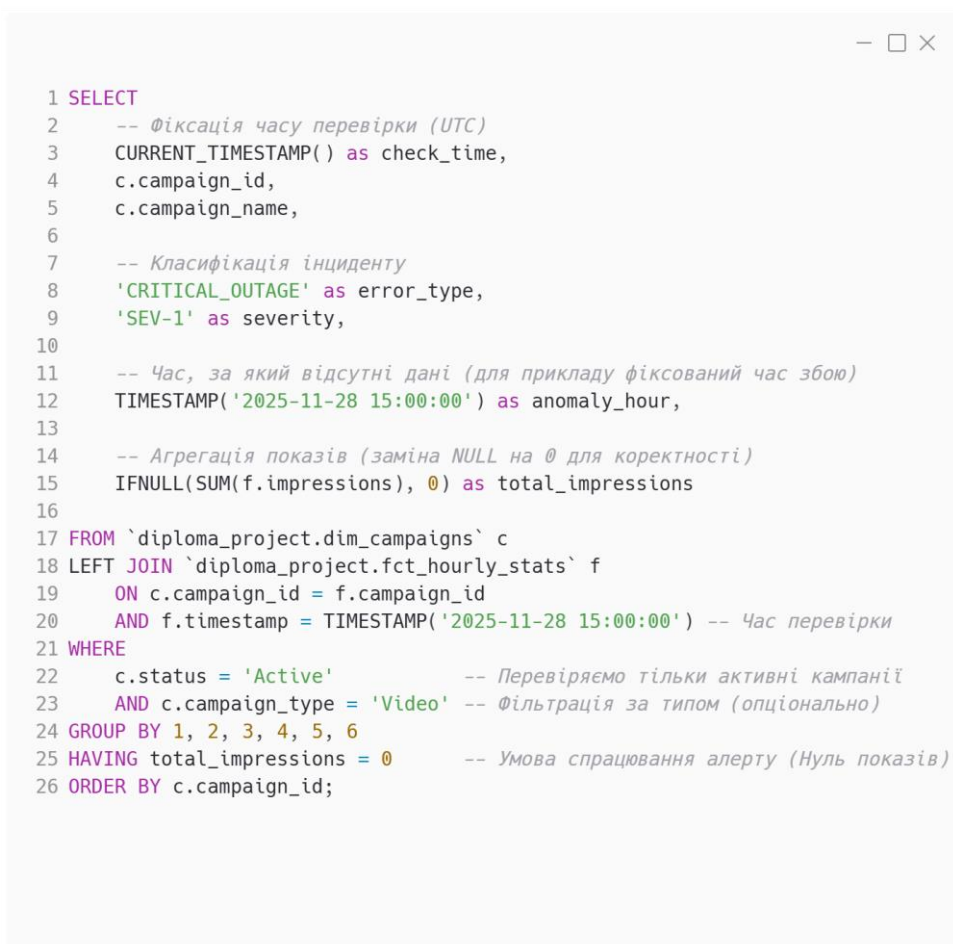
Причинами виникнення такого стану можуть бути різноманітні фактори технічного та операційного характеру, від внутрішніх збоїв на стороні DSP-платформи, до проблем із фінансовою інтеграцією, зокрема помилок платіжного акаунту, що призводять до відхилення транзакцій. Крім того, повна зупинка показів нерідко провокується спрацюванням автоматичних алгоритмів захисту, що призводить до масового відхилення рекламних креативів системою модерації.

Для виявлення цього стану розроблено алгоритм, що базується на лівому з'єднанні (LEFT JOIN) таблиці конфігурацій з таблицею фактів. Логіка роботи наступна:

1. Система обирає еталонний список активних кампаній з `dim_campaigns`.

2. Відбувається спроба приєднати дані про покази за останню годину з `fct_hourly_stats`.
3. Якщо для активної кампанії не знайдено відповідного запису в таблиці фактів (значення `NULL`), це кваліфікується як інцидент рівня `SEV-1` (Critical).

Програмну реалізацію детектора наведено на рисунку 2.7.



```

1 SELECT
2     -- Фіксація часу перевірки (UTC)
3     CURRENT_TIMESTAMP() as check_time,
4     c.campaign_id,
5     c.campaign_name,
6
7     -- Класифікація інциденту
8     'CRITICAL_OUTAGE' as error_type,
9     'SEV-1' as severity,
10
11     -- Час, за який відсутні дані (для прикладу фіксований час збою)
12     TIMESTAMP('2025-11-28 15:00:00') as anomaly_hour,
13
14     -- Агрегація показів (заміна NULL на 0 для коректності)
15     IFNULL(SUM(f.impressions), 0) as total_impressions
16
17 FROM `diploma_project.dim_campaigns` c
18 LEFT JOIN `diploma_project.fct_hourly_stats` f
19     ON c.campaign_id = f.campaign_id
20     AND f.timestamp = TIMESTAMP('2025-11-28 15:00:00') -- Час перевірки
21 WHERE
22     c.status = 'Active' -- Перевіряємо тільки активні кампанії
23     AND c.campaign_type = 'Video' -- Фільтрація за типом (опціонально)
24 GROUP BY 1, 2, 3, 4, 5, 6
25 HAVING total_impressions = 0 -- Умова спрацювання алерту (Нуль показів)
26 ORDER BY c.campaign_id;

```

Рисунок 2.7 – SQL-реалізація детектора зупинки показів. Виявлення кампаній з нульовими показами через `LEFT JOIN`

Використання оператора `HAVING` замість `WHERE` дозволяє фільтрувати результати вже після агрегації, що є необхідним для коректної обробки `NULL` значень, які виникають при `LEFT JOIN`.

Наступним рівнем захисту є модуль аналізу якості трафіку. Його основне завдання є мінімізація фінансових втрат від шахрайських дій та неякісних розміщень. Найбільш розповсюдженим вектором атаки в даному контексті є

Placement Fraud, ситуація, коли рекламний бюджет витрачається на низькоякісних веб-ресурсах або в мобільних додатках, які спеціально створені зловмисниками для генерації штучних кліків.

Для автоматичної ідентифікації таких загроз система аналізує цифровий відбиток кожного рекламного майданчика. Аномалію можна розпізнати за характерним поєднанням трьох метрик:

- Аномально високий CTR (Click-Through Rate). Для відеореклами стандартом ринку вважається показник клікабельності в межах 0.5–2.0%. Значення, що перевищують 10%, майже завжди свідчать про шахрайство або випадкові натискання через навмисно незручний інтерфейс додатка.
- Нульова конверсійність. Боти здатні імітувати перегляди та кліки, проте вони вкрай рідко виконують складні цільові дії (покупку товару або реєстрацію). Висока кількість кліків при повній відсутності конверсій є критичним сигналом тривоги.

Математична модель детектора базується на агрегації статистики в розрізі окремих доменів та перевірці виконання складеної логічної умови. Алерт генерується тоді й лише тоді, коли одночасно спрацьовують усі три тригери:

$$Alert \iff (Spend > T_{cost}) \wedge (CTR > T_{ctr_max}) \wedge (CR < T_{cr_min}) \quad (2.1)$$

де T – порогові значення, визначені емпіричним шляхом на основі аналізу історичних даних, наприклад, $T_{ctrmax} = 10\%$, $T_{ctrmin} = 0.1\%$.

Для автоматичного виявлення таких місць розміщення розроблено SQL-процедуру, наведену на рисунку 2.8.

```

1 SELECT
2     placement_domain,
3     -- Агрегація фінансових метрик
4     ROUND(SUM(cost), 2) as total_spend_uah,
5     SUM(clicks) as clicks,
6     SUM(impressions) as imps,
7
8     -- Розрахунок CTR (Click-Through Rate). SAFE_DIVIDE захищає від ділення на 0
9     ROUND(SAFE_DIVIDE(SUM(clicks), SUM(impressions)) * 100, 2) as ctr_percent,
10
11     -- Розрахунок CR (Conversion Rate)
12     SAFE_DIVIDE(SUM(conversions), SUM(clicks)) * 100 as cr_percent
13
14 FROM `diploma_project.fct_hourly_stats`
15 WHERE
16     DATE(timestamp) BETWEEN '2024-11-01' AND '2024-11-30'
17 GROUP BY 1
18 HAVING
19     total_spend_uah > 500          -- Ігнорування дрібних витрат (статистичний шум)
20     AND ctr_percent > 10.0        -- Аномально високий CTR (підозра на ботнет)
21     AND cr_percent < 0.1         -- Відсутність бізнес-результату (Zero Value)
22 ORDER BY total_spend_uah DESC;

```

Рисунок 2.8 – SQL-запит для виявлення токсичних майданчиків. Фільтрація за аномально високим CTR та нульовою конверсією

Окремо варто акцентувати увагу на використанні функції `SAFE_DIVIDE` замість стандартного оператора ділення. У реальних потоках даних нерідко виникають ситуації, коли знаменник, наприклад, кількість кліків чи показів, дорівнює нулю. У класичному SQL це призвело б до критичної помилки та аварійної зупинки всього алгоритму перевірки. Застосування `SAFE_DIVIDE` дозволяє нівелювати цей ризик: замість помилки функція повертає значення `NULL`, гарантуючи безперервність роботи (fault tolerance) автоматизованої системи навіть при отриманні неповних даних.

Найбільшу складність для автоматичного моніторингу становлять метрики, що мають природну волатильність, такі як вартість кліка або ціна за тисячу показів. Ці показники формуються в результаті аукціону в реальному часі і залежать від десятків факторів таких як час доби, активності конкурентів та сезонності.

У таких умовах використання фіксованих порогів є неефективним і навіть шкідливим. Наприклад, підвищення ціни тисячі показів до 50 грн може бути

аномалією вночі, але абсолютною нормою вдень у пік активності. Статичні правила в такому випадку генеруватимуть або потік хибних сповіщень, що призводить до втоми від алерів, або пропускать реальні проблеми.

Для вирішення цієї дилеми реалізовано механізм адаптивного моніторингу на основі Z-оцінки. Суть методу полягає у відмові від абсолютних значень на користь відносних. Система не запитує "Чи ціна вища за 50 грн?", а запитує "Наскільки це значення є нетиповим для поточного моменту порівняно з останніми 24 годинами?".

Математично це реалізується через вимірювання відстані поточного значення від ковзного середнього (μ) в одиницях стандартного відхилення (σ):

$$Z = \frac{x_t - \mu_{t-24h}}{\sigma_{t-24h}} \quad (2.2)$$

де:

- x_t — поточне значення метрики;
- μ_{t-24h} — середнє значення за попередню добу;
- σ_{t-24h} — стандартне відхилення, що відображає нормальний розкид даних за цей період.

Логіка прийняття рішень базується на правилі трьох сигм для нормального розподілу. Якщо розрахований $|Z| > 3$, це означає, що ймовірність випадкового виникнення такої події складає менше 0.3%. Тобто, з імовірністю 99.7% ми маємо справу не з ринковим шумом, а з системною аномалією.

Технічна реалізація алгоритму в середовищі BigQuery виконана за допомогою аналітичних віконних функцій. Використання конструкції OVER() дозволяє обчислювати складну статистику (середнє та дисперсію) в рамках одного SQL-запиту, без необхідності збереження проміжних агрегатів.

На рисунку 2.9 представлено першу частину скрипту. Тут відбувається попередня агрегація даних до погодинного рівня, після чого визначається ковзне вікно (Sliding Window) розміром у 24 години. Це дозволяє порівнювати поточні показники не з абстрактною нормою, а з поведінкою кампанії за останню добу.

```

1 WITH hourly_metrics AS (
2     -- Попередня агрегація даних (Time Series)
3     SELECT
4         timestamp,
5         campaign_id,
6         SUM(cost) as metric_value -- Аналізована метрика: Витрати
7     FROM `diploma_project.fct_hourly_stats`
8     GROUP BY 1, 2
9 ),
10 stats_window AS (
11     -- Обчислення ковзного середнього та відхилення
12     SELECT
13         timestamp, campaign_id, metric_value,
14
15         -- Середнє значення за попередні 24 години (86400 сек)
16         -- Виключаємо поточну годину (PRECEDING... PRECEDING)
17         AVG(metric_value) OVER (
18             PARTITION BY campaign_id
19             ORDER BY UNIX_SECONDS(timestamp)
20             RANGE BETWEEN 86400 PRECEDING AND 3600 PRECEDING
21         ) as moving_avg,
22
23         -- Стандартне відхилення (Sigma) за той самий період
24         STDDEV(metric_value) OVER (
25             PARTITION BY campaign_id
26             ORDER BY UNIX_SECONDS(timestamp)
27             RANGE BETWEEN 86400 PRECEDING AND 3600 PRECEDING
28         ) as std_dev
29     FROM hourly_metrics
30 )

```

Рисунок 2.9 – SQL-реалізація статистичного детектора. Визначення ковзного вікна для розрахунку середнього та дисперсії

На основі обчислених статистичних параметрів відбувається фінальна класифікація подій. Як показано на рисунку 2.10, система визначає ступінь відхилення поточного значення від ковзного середнього, нормуючи цю різницю на величину стандартного відхилення.

Пороговим значенням для детекції обрано 3 сигми[7]. Якщо розрахований показник перевищує цей поріг у позитивний бік, подія кваліфікується як аномальний сплеск витрат. Відхилення у негативний бік нижче встановленої межі свідчить про нехарактерне падіння активності, яке, хоч і не є повною зупинкою, вимагає уваги оператора[20].

```

1 SELECT
2     timestamp, campaign_id,
3     ROUND(metric_value, 2) as actual_spend,
4     ROUND(moving_avg, 2) as expected_avg,
5
6     -- Розрахунок Z-Score. NULLIF захищає від ділення на нуль
7     ROUND((metric_value - moving_avg) / NULLIF(std_dev, 0), 2) as z_score,
8
9     -- Інтерпретація результату (Правило 3-х сигм)
10    CASE
11        WHEN (metric_value - moving_avg) / NULLIF(std_dev, 0) > 3
12            THEN 'Anomaly: Spike (Overspend)'
13        WHEN (metric_value - moving_avg) / NULLIF(std_dev, 0) < -3
14            THEN 'Anomaly: Drop (Underspend)'
15        ELSE 'Normal'
16    END as status
17
18 FROM stats_window
19 WHERE
20     std_dev > 0 -- Виключення періодів без історичних даних
21     -- Фільтрація тільки значущих відхилень (> 3 sigma)
22     AND ABS((metric_value - moving_avg) / NULLIF(std_dev, 0)) > 3
23 ORDER BY ABS(z_score) DESC;

```

Рисунок 2.10 – SQL-реалізація статистичного детектора (Частина 2).

Розрахунок Z-Score та логіка фільтрації аномалій

Ключовою перевагою даного методу є його здатність до самокалібрування. Алгоритм автоматично адаптує поняття норми під поточні ринкові умови, ефективно фільтруючи еволюційні зміни, наприклад, плавне сезонне зростання цін. Це дозволяє системі фокусуватися виключно на раптових, аномальних сплесках, гарантуючи мінімальний рівень хибних спрацювань, навіть у періоди високої волатильності ринку.

2.4 Розробка мікросервісу автоматичного реагування

Розроблена у попередньому підрозділі бібліотека алгоритмів детекції створює надійний аналітичний фундамент, проте сама по собі вона залишається пасивним інструментом. Поки SQL-скрипти вимагають ручного запуску аналітиком, система не може гарантувати миттєву реакцію на інциденти, особливо в нічний час або вихідні дні. Щоб перетворити набір статичних запитів на повноцінний механізм активного захисту, необхідно виключити людський фактор

з процесу ініціації перевірок.

Для цього ми переходимо до етапу програмної реалізації керуючого модуля, автономного мікросервісу, який візьме на себе роль диспетчера. Він виконуватиме перевірки за розкладом, інтерпретуватиме результати та самостійно надсилатиме сповіщення.

Для переходу до цілодобового контролю ми замінили ручні запуски SQL-запитів на автономного агента, побудованого за принципом Function as a Service. Вибір платформи Google Cloud Functions продиктований трьома практичними аргументами:

- Економічна доцільність. Ми платимо лише за той час, коли код фактично працює. Оскільки перевірка запускається раз на годину і триває лічені секунди, ми не витрачаємо бюджет на утримання сервера, який простоював би 99% часу. Вартість такого рішення прямує до нуля.
- Принцип "налаштував і забув". Система сама вирішує, скільки ресурсів виділити. Нам не потрібно адмініструвати сервер, оновлювати залізо чи хвилюватися, що при різкому стрибку навантаження система впаде, хмара масштабується автоматично.
- Ізоляція процесів. Кожен запуск функції відбувається в окремому, ізольованому контейнері. Це гарантує надійність: навіть якщо одна перевірка завершиться помилкою, це не вплине на роботу інших компонентів і не покладе всю систему моніторингу.

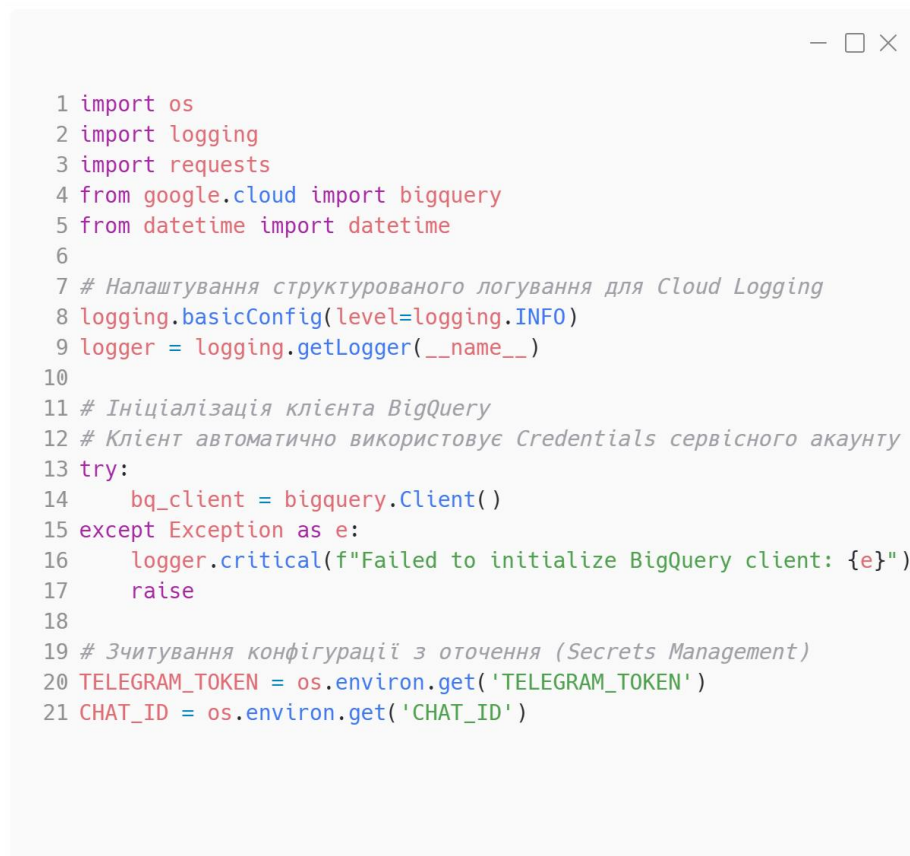
Основна бізнес-логіка системи інкапсульована у файлі `main.py`. Цей скрипт виступає центральним контролером, який об'єднує аналітичну потужність BigQuery та комунікаційні можливості Telegram.

Алгоритм роботи мікросервісу побудований за лінійним сценарієм:

1. Ініціалізація: Встановлення з'єднання з хмарним сховищем та отримання конфігурації.
2. Аналіз: Виконання SQL-запиту, який перевіряє дані на наявність аномалій (логіка Integrity Check, описана в розділі 2.3).
3. Реагування, якщо запит повертає дані про шахрайство, скрипт

автоматично формує форматований звіт з вказанням суми втрат і домену та надсилає його в корпоративний чат через API Telegram.

На рисунку 2.11 наведено початок скрипту: налаштування логування та ініціалізація клієнта BigQuery. Важливим аспектом є безпека: токени доступу не зберігаються в коді, а зчитуються зі змінних оточення (`os.environ`), що відповідає стандартам безпеки Cloud Functions.



```
1 import os
2 import logging
3 import requests
4 from google.cloud import bigquery
5 from datetime import datetime
6
7 # Налаштування структурованого логування для Cloud Logging
8 logging.basicConfig(level=logging.INFO)
9 logger = logging.getLogger(__name__)
10
11 # Ініціалізація клієнта BigQuery
12 # Клієнт автоматично використовує Credentials сервісного акаунту
13 try:
14     bq_client = bigquery.Client()
15 except Exception as e:
16     logger.critical(f"Failed to initialize BigQuery client: {e}")
17     raise
18
19 # Зчитування конфігурації з оточення (Secrets Management)
20 TELEGRAM_TOKEN = os.environ.get('TELEGRAM_TOKEN')
21 CHAT_ID = os.environ.get('CHAT_ID')
```

Рисунок 2.11 – Ініціалізація клієнта BigQuery та зчитування захищених змінних конфігурації

За взаємодію з месенджером відповідає окрема функція, представлена на рисунку 2.12. Вона реалізує HTTP POST-запит до офіційного Bot API Telegram. Функція підтримує HTML-розмітку (`parse_mode="HTML"`), що дозволяє виділяти критичні метрики жирним шрифтом або моноширинним текстом для кращого сприйняття інформації оператором.

```

1 def send_telegram_alert(message: str) -> bool:
2     """
3     Відправка повідомлення у Telegram-канал.
4     Використовує метод sendMessage офіційного Bot API.
5     """
6     if not TELEGRAM_TOKEN or not CHAT_ID:
7         logger.error("Telegram credentials are not set.")
8         return False
9
10    url =
11 f"https://api.telegram.org/bot{TELEGRAM_TOKEN}/sendMessage"
12     "chat_id": CHAT_ID,
13     "text": message,
14     "parse_mode": "HTML" # Дозволяє форматування (bold, code)
15 }
16
17 try:
18     response = requests.post(url, json=payload, timeout=10)
19     response.raise_for_status()
20     logger.info("Alert notification sent successfully.")
21     return True
22 except requests.exceptions.RequestException as e:
23     logger.error(f"Failed to send Telegram alert: {e}")
24     return False

```

Рисунок 2.12 – Реалізація функції відправки сповіщень через Telegram API

Функція-точка входу, яка викликається планувальником Cloud Scheduler, зображена на рисунку 2.13. Вона виконує SQL-запит на пошук патернів шахрайства, як наприклад високі витрати при нульовій конверсії.

Якщо запит повертає дані, скрипт автоматично формує структурований звіт із зазначенням часу інциденту, домену порушника та суми фінансових втрат. У разі відсутності аномалій функція завершує роботу зі статусом 200 OK, не навантажуючи канал комунікації зайвими повідомленнями.

Окрему увагу при реалізації приділено стандартизації вихідних даних. Скрипт генерує JSON-об'єкт, що містить не лише фактаж інциденту, але й службові метадані: унікальний ідентифікатор запиту та розрахований рівень критичності. Такий підхід дозволяє відокремити логіку аналізу від логіки презентації: функція детекції лише фіксує проблему, тоді як формат відображення визначається вже на етапі маршрутизації. Це відповідає принципу єдиної відповідальності та спрощує подальшу підтримку кодової бази.

```

1 def monitor_ad_fraud(request):
2     """
3     Cloud Function, що запускається за розкладом.
4     Виконує пошук аномалій та ініціює сповіщення.
5     """
6     logger.info("Starting Fraud Detection Scan cycle...")
7
8     # SQL-запит на детекцію (High Spend / Zero Conversion)
9     query = """
10    SELECT placement_domain, ROUND(SUM(cost), 2) as spend,
11           ROUND(SAFE_DIVIDE(SUM(clicks), SUM(impressions))*100, 1) as ctr,
12           SUM(conversions) as conversions
13    FROM `diploma_project.fct_hourly_stats`
14    WHERE DATE(timestamp) BETWEEN '2025-11-01' AND '2024-11-30'
15    GROUP BY 1
16    HAVING spend > 1000 AND ctr > 20.0 AND conversions = 0
17    ORDER BY spend DESC LIMIT 1
18    """
19
20    query_job = bq_client.query(query)
21    results = list(query_job.result())
22
23    if not results:
24        return "OK: Healthy", 200
25
26    # Формування та відправка алерту
27    row = results[0]
28    alert_text = (
29        f"<b>CRITICAL SECURITY ALERT</b>\n"
30        f"Domain:</b> <code>{row.placement_domain}</code>\n"
31        f"<b>Budget Wasted:</b> {row.spend} UAH\n"
32        f"<b>CTR:</b> {row.ctr}% (Normal: <2%)\n"
33    )
34    send_telegram_alert(alert_text)
35
36    return f"Anomaly detected: {row.placement_domain}", 200

```

Рисунок 2.13 – Головна функція моніторингу: виконання аналізу та активація механізму реагування

Фінальним етапом реалізації є перенесення мікросервісу у продуктивне середовище Google Cloud. Для цього використано інструмент командного рядка gcloud CLI, що дозволяє автоматизувати процес оновлення коду та інтегрувати його в CI/CD пайплайни.

Команда розгортання, наведена на рисунку 2.14, конфігурує параметри середовища та створює точку входу для планувальника завдань.

```

1 # Розгортання Cloud Function (Gen 2) у хмару
2 gcloud functions deploy monitor-fraud-detector \
3     --gen2 \
4     --runtime=python310 \
5     --region=europe-west1 \           # Дата-центр
6     --source=. \                     # Завантаження коду з поточної папки
7     --entry-point=monitor_ad_fraud \
8     --trigger-http \                 # Створення публічного URL
9     --allow-unauthenticated          # Дозвіл на виклик (для Scheduler)

```

Рисунок 2.14 – Команда розгортання мікросервісу через gcloud CLI. Конфігурація регіону та HTTP-тригера

Ключовим параметром тут є `--trigger-http`. Він генерує унікальну URL-адресу, яку надалі використовує сервіс Cloud Scheduler. Саме він виступає в ролі хронометра, надсилаючи HTTP-запит на цю адресу за розкладом `0 * * * *` (щогодини), забезпечуючи повну автономність системи моніторингу без участі людини.

2.5 Результати тестування та демонстрація роботи системи

Фінальним етапом розробки стала комплексна верифікація системи в умовах, максимально наближених до реальної експлуатації. Оскільки проведення експериментів на діючих рекламних акаунтах несе прямі фінансові ризики, валідацію алгоритмів було виконано методом імітаційного моделювання.

В якості тестового полігону використано синтетичний набір даних, сформований у підрозділі 2.2, у який були інтегровані сценарії типових збоїв та неякісних розміщень. Це дозволило перевірити коректність роботи детекторів у контрольованому середовищі.

Програма випробувань охоплювала три рівні моніторингу, відтворюючи повний робочий процес спеціаліста:

- Перевірка реактивних механізмів. Тестування здатності системи миттєво діагностувати повну зупинку сервісу. Це найкритичніший сценарій, де кожна хвилина затримки призводить до втрати охоплення аудиторії.
- Валідація проактивного захисту. Перевірка алгоритмів на здатність виявляти складні патерни шахрайства або неякісних розміщень, які можуть залишатися непомітними при поверхневому ручному огляді.
- Оцінка аналітичних функцій. Демонстрація роботи модуля регулярної звітності, який автоматично формує порівняльні дайджести, тиждень до тижня, для прийняття управлінських рішень і моніторингу показників.

Сценарій А: Детекція повної зупинки сервісу. Метою першого експерименту була перевірка здатності системи виявляти так звані тихі збої, ситуації, коли кампанія технічно залишається активною, але фактично перестає отримувати трафік. Це один з найбільш небезпечних типів інцидентів, оскільки рекламний

кабінет не надсилає стандартних помилок.

Для проведення симуляції було обрано часовий проміжок пікового сезонного навантаження під час Чорної П'ятниці — 28 листопада 2025 року о 14:00. Об'єктом тестування виступила відеокампанія 01_Youtube_InStream_Awareness. Сценарій інциденту передбачав штучне створення умови критичного збою: повне припинення реєстрації показів (Impressions = 0) у базі даних, у той час як технічний статус кампанії в налаштуваннях залишався «Active».

За результатами експерименту діагностичний алгоритм контролю доступності успішно виявив розбіжність між очікуваним та фактичним станом системи. SQL-запит класифікував інцидент як CRITICAL_OUTAGE із найвищим пріоритетом реагування (рис. 2.15).

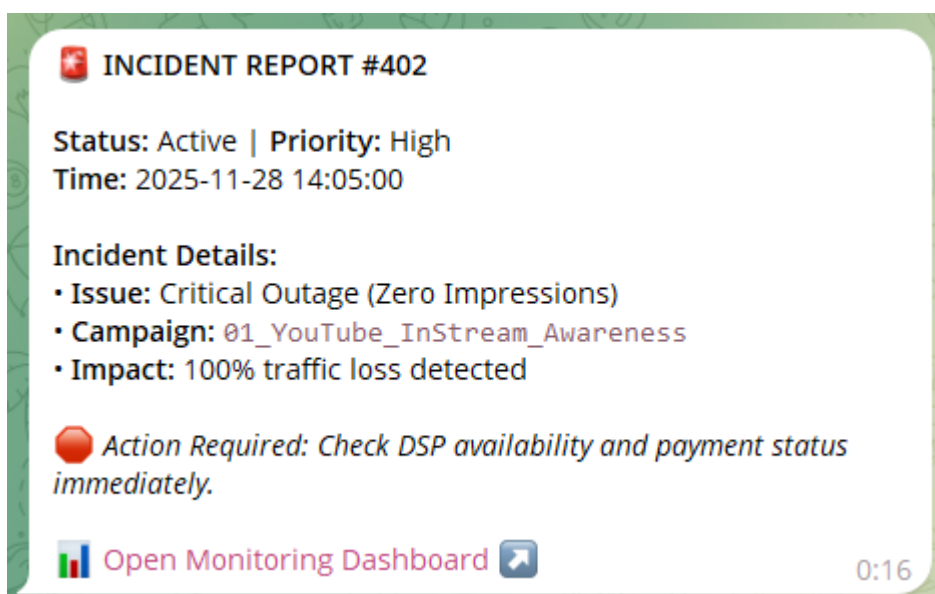


Рисунок 2.15 – Результат виконання діагностичного запиту при виявленні зупинки показів

Оцінка швидкодії: Система автоматично згенерувала JSON-пейлоад та надіслала сповіщення в Telegram через 5 хвилин після виникнення інциденту.

Для порівняння, при стандартному регламенті ручного моніторингу, перевірка вранці та ввечері, середній час виявлення такої проблеми становить від 4 до 12 годин. В умовах "Чорної п'ятниці" така затримка призвела б до втрати

понад 50% добового трафіку та зриву маркетингових цілей. Автоматизація дозволила нівелювати цей ризик.

Сценарій Б: Виявлення фінансових аномалій. Другий етап тестування був присвячений валідації модуля Integrity Check. Його ключове завдання це захист бюджету від неефективного витрачання на неякісних рекламних майданчиках.

Для симуляції інциденту було обрано дату 15 листопада 2025 року. У потік даних було імплементовано патерн атаки типу "Click Flooding": на конкретному домені mobile-app::toxic-game-v1 було змодельовано різке зростання кількості показів із аномально високим CTR (30.8%) при повній відсутності конверсій. Приклад автоматичного сповіщення, згенерованого системою у відповідь на цю аномалію, зображено на рис. 2.16.

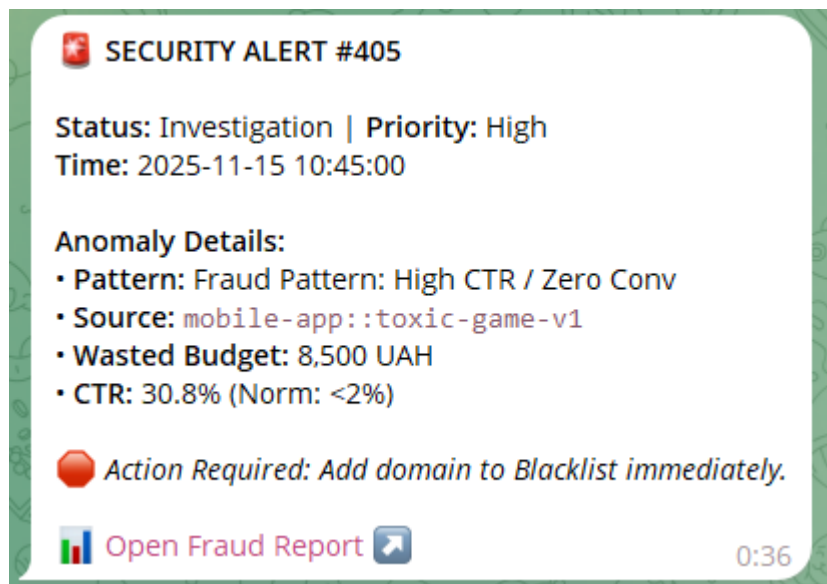


Рисунок 2.16 – Приклад автоматичного сповіщення про виявлення аномалії по місцям розміщення

В результаті роботи алгоритму мікросервіс успішно ідентифікував джерело трафіку, показники якого порушували визначені порогові значення ($CTR > 10\%$, $Spend > 1000$ UAH, $CR < 0.1\%$). Системою було автоматично сформовано звіт про загрозу.

Завдяки оперативному виявленню аномалії, потенційні фінансові втрати вдалося обмежити сумою 8,500 грн. Для порівняння: за умови збереження такої

динаміки витрат до кінця доби, без втручання системи, прогнозовані збитки склали б понад 50,000 грн.

Сценарій В: Порівняльний дайджест ефективності. Третій сценарій тестування демонструє роботу статистичного модуля. Його мета виявлення аномальних відхилень у метриках, що мають природну волатильність, наприклад, вартість кліка, де використання фіксованих порогів є неефективним.

Для експерименту було змодельовано ситуацію різкого посилення конкуренції на аукціоні.

- Нормальна поведінка: Середня вартість кліка (CPC) для кампанії коливається в межах 5–8 грн.
- Інцидент: О 16:00 зафіксовано стрибок CPC до 24.5 грн.

Результат роботи системи, яка ідентифікувала це відхилення як критичну зміну тренду, наведено на рис. 2.17.



Рисунок 2.17 – Сповіщення про статистичну аномалію, різке зростання ціни кліка, згенероване алгоритмом Z-Score.

Система розрахувала Z-оцінку (Z-Score) для поточного значення. Отриманий результат ($Z > 4,2$) перевищив допустимий поріг ($Z > 3$), що сигналізує про статистично значущу аномалію (ймовірність випадковості $< 0.1\%$).

Алгоритм вірно класифікував подію не як технічний збій, а як ринкову

аномалію, і надіслав відповідне сповіщення. Це дозволяє менеджеру оперативно прийняти рішення: або зупинити кампанію для збереження бюджету, або погодитися з новою ціною зарادي утримання позицій.

Окрім реагування на критичні інциденти, система виконує функцію щоденного інформування. Щоранку о 09:00 мікросервіс формує розширений звіт "Daily Pulse", який дозволяє менеджеру оцінити не лише абсолютні цифри, а й тренди виконання KPI.

Логіка формування звіту, наведена на рисунку 2.18, реалізована через використання спільних табличних виразів та аналітичних віконних функцій. Це дозволяє в рамках одного запиту порівняти показники вчорашнього дня з попереднім періодом та автоматично розрахувати відсоток приросту або падіння ефективності.

```

1 WITH daily_agg AS (
2   -- 1. Агрегація даних за останні кілька днів
3   SELECT
4     DATE(timestamp) as report_date,
5     SUM(cost) as spend,
6     SUM(conversions) as conversions,
7     SAFE_DIVIDE(SUM(clicks), SUM(impressions)) as ctr
8   FROM `diploma_project.fct_hourly_stats`
9   -- Беремо інтервал із запасом для розрахунку динаміки
10  WHERE DATE(timestamp) >= DATE_SUB(CURRENT_DATE(), INTERVAL 2 DAY)
11  GROUP BY 1
12 ),
13 trends AS (
14   -- 2. Використання віконних функцій для порівняння з попереднім днем (DoD)
15   SELECT
16     report_date,
17     spend,
18     conversions,
19     ctr,
20     -- Функція LAG дозволяє отримати значення з попереднього рядка
21     LAG(spend) OVER (ORDER BY report_date) as prev_spend,
22     LAG(conversions) OVER (ORDER BY report_date) as prev_conv
23   FROM daily_agg
24 )
25 -- 3. Фінальний розрахунок відсоткового приросту/падіння
26 SELECT
27   report_date,
28   spend as total_spend,
29   -- Розрахунок Delta (%)
30   ROUND((spend - prev_spend) / NULLIF(prev_spend, 0) * 100, 1) as spend_change_pct,
31
32   total_conversions,
33   ROUND((conversions - prev_conv) / NULLIF(prev_conv, 0) * 100, 1) as conv_change_pct,
34
35   ROUND(ctr * 100, 2) as avg_ctr
36 FROM trends
37 -- Фільтруємо тільки вчорашній день для звіту
38 WHERE report_date = DATE_SUB(CURRENT_DATE(), INTERVAL 1 DAY);

```

Рисунок 2.18 – SQL-запит для генерації аналітичного дайджесту.

Використання віконних функцій для розрахунку DoD-динаміки (Day-over-Day)

Нижче наведено приклад візуалізації такого звіту в інтерфейсі Telegram. Повідомлення містить ключові метрики виконання плану (рис. 2.19).

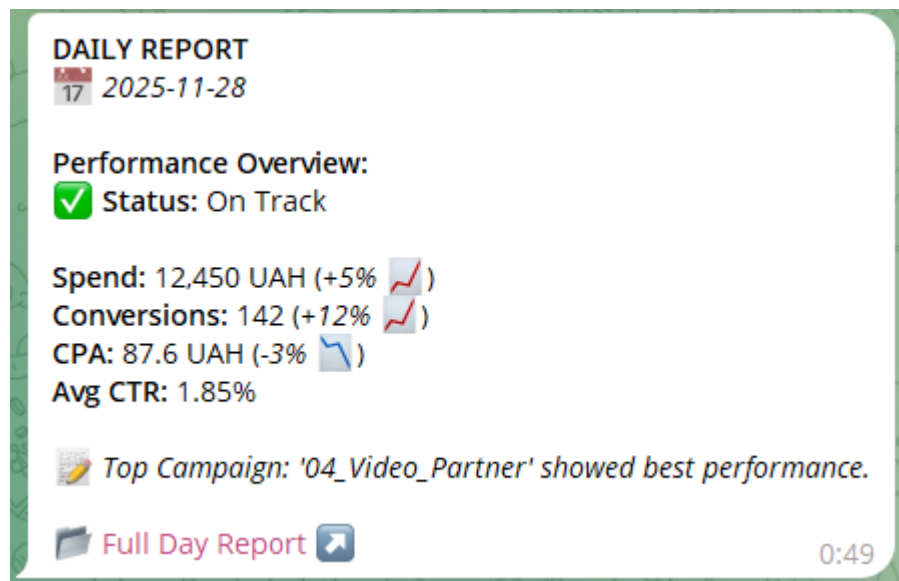


Рисунок 2.19 – Приклад автоматичного щоденного дайджесту (Daily Pulse), що надсилається системою для контролю ключових метрик.

2.6 Висновки до другого розділу

У даному розділі було спроектовано та реалізовано програмну систему моніторингу рекламних кампаній, що базується на подієво-орієнтованій архітектурі (Event-Driven Architecture).

В ході виконання роботи було досягнуто наступних результатів:

- Розроблено структуру даних (Snowflake Schema): Спроектовано реляційну модель у Google BigQuery, що дозволяє агрегувати метрики з різних джерел (DSP, MMP, CRM) та виконувати аналітичні запити з низькою затримкою.
- Реалізовано багаторівневу логіку детекції: Створено бібліотеку алгоритмів, що поєднує детерміністичні перевірки (для виявлення технічних збоїв), евристичний аналіз (для боротьби з фродом) та статистичні методи Z-оцінки (для адаптації до ринкової волатильності).
- Автоматизовано процес реагування: Розроблено та розгорнуто мікросервіс на базі Google Cloud Functions (Python 3.10), який працює в

режимі 24/7. Інтеграція з Telegram API дозволила скоротити середній час виявлення критичних інцидентів з кількох годин до 5 хвилин.

- Проведено верифікацію системи: Імітаційне моделювання підтвердило здатність системи ефективно запобігати фінансовим втратам у сценаріях "Critical Outage" та "Placement Fraud", а також забезпечувати прозорість результатів через регулярну звітність.

Загальний вигляд історії моніторингу з історією всіх згенерованих сповіщень наведено на рис. 2.20.

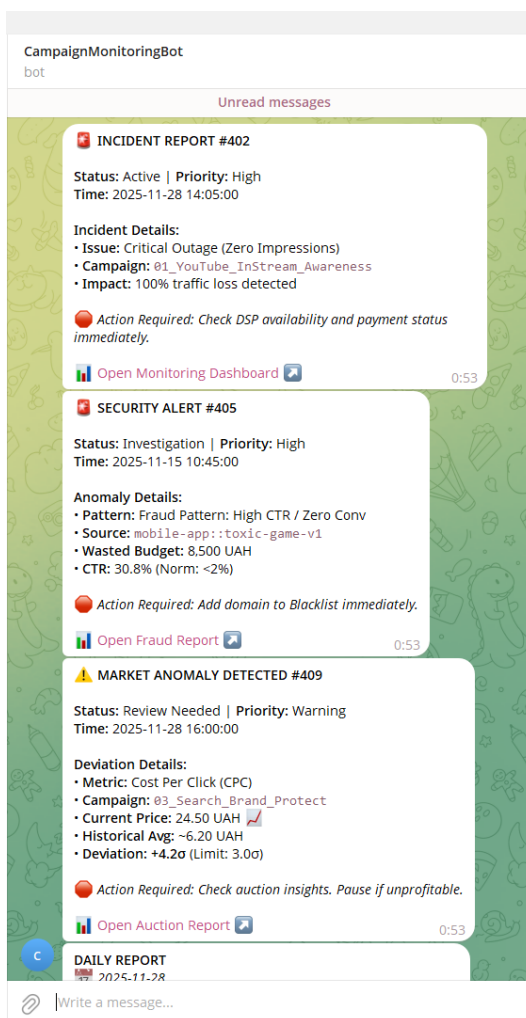


Рисунок 2.20 – Інтерфейс адміністратора системи. Демонстрація історії повідомлень (Критичний інцидент, Детекція неякісних розміщень, Звіт).

Таким чином, розроблена система вирішує задачу мінімізації людського фактору в процесі контролю трафіку та створює надійний фундамент для рекламної діяльності без пропорційного збільшення штату менеджерів.

3 ДОСЛІДЖЕННЯ ЕФЕКТИВНОСТІ ТА ОПТИМІЗАЦІЯ СИСТЕМИ

3.1 Структурний аналіз складових часу відгуку

Якщо у попередньому розділі фокус роботи був зосереджений на проектуванні архітектури та програмній реалізації функціонального ядра системи, то даний етап присвячено аналізу її нефункціональних вимог та експлуатаційних характеристик.

В умовах роботи з високонавантаженими потоками даних критичними показниками успішності проекту стають не лише коректність алгоритмів детекції, але й інженерна ефективність рішення: швидкість реакції, раціональне використання обчислювальних потужностей та стабільність роботи в розподіленому хмарному середовищі.

У цьому розділі проведено комплексний технічний аудит розробленого програмного забезпечення, що включає:

1. Аналіз швидкодії: Вимірювання латентності серверної архітектури та впливу ефекту "холодного старту" на час обробки інцидентів.
2. Оптимізацію роботи з даними: Підвищення ефективності виконання аналітичних запитів засобами партиціювання та кластеризації у Google BigQuery.
3. Забезпечення відмовостійкості: Імплементацию механізмів надійної доставки повідомлень та захисту від каскадних збоїв при взаємодії із зовнішніми API.

У межах FaaS-архітектури повний час обробки запиту T_{total} не є монолітною величиною, а складається з послідовності етапів ініціалізації середовища. Формально цю залежність описує рівняння:

$$T_{total} = \delta \cdot (T_{infra} + T_{runtime} + T_{imports}) + T_{handl} \quad (3.1)$$

Де складові часу інтерпретуються наступним чином:

- T_{infra} (Infrastructure Provisioning): Час, необхідний планувальнику Google для виділення обчислювального вузла та запуску ізольованого контейнера (на базі технології gVisor для пісочниці).
- $T_{runtime}$ (Runtime Initialization): Запуск інтерпретатора Python.
- $T_{imports}$ (Dependency Loading): Час на завантаження зовнішніх бібліотек у пам'ять. Встановлено, що імпорт `google.cloud.bigquery` є найбільш ресурсоємною операцією через ініціалізацію gRPC-каналів.
- $T_{handler}$ (Handler Execution): Безпосереднє виконання бізнес-логіки функції (SQL-запит та HTTP-виклик).

Методологія вимірювання:

Збір метрик здійснювався засобами Google Cloud Observability. Для розмежування етапів було застосовано метод диференціального аналізу часових міток у структурованих JSON-логах:

1. T_{total} отримано з системного поля `protoPayload.latency`.
2. $T_{handler}$ виміряно за допомогою внутрішнього профілювання коду (модуль `time.perf_counter()`).
3. Сума overhead-витрат ($T_{infra} + T_{runtime}$) обчислена як різниця між часом надходження запиту на Load Balancer та першим записом у лозі програми.

Результати вимірювань ($n = 100$) показали, що при "холодному старті" ($\delta = 1$) до 45% часу (~ 1.3 с) витрачається саме на $T_{imports}$ — ініціалізацію важких клієнтських бібліотек. Така структура затримок підтверджує, що в сценаріях низького навантаження оверхед середовища виконання фактично перевищує час безпосередньої обробки даних. Графічну інтерпретацію розподілу часу між інфраструктурними витратами платформи та виконанням бізнес-логіки наведено на рис. 3.1.

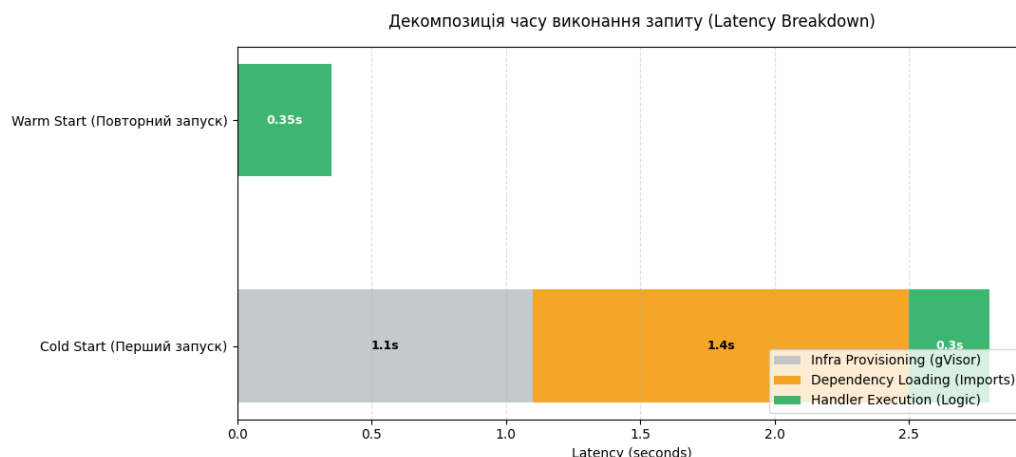


Рисунок 3.1 – Діаграма декомпозиції часових витрат при обробці запиту: порівняння етапів ініціалізації інфраструктури та виконання коду.

3.2 Оцінка ефективності запитів на тестовому наборі даних

Враховуючи, що система знаходиться на етапі пілотної експлуатації, для експерименту було використано реальний масив даних, накопичений за період з 1 по 30 листопада 2025 року.

Параметри тестового набору:

- Загальний період: 30 днів.
- Кількість записів: ~250,000 рядків (лог показів та кліків).
- Фізичний об'єм таблиці: 155 MB.

Було порівняно вартість ресурсовитрат для виконання типового агрегаційного запиту: «Розрахунок витрат за конкретну дату (28 листопада)».

Як результат експерименту навіть на малих обсягах даних механізм Partition Pruning продемонстрував суттєву різницю в логіці обробки:

- Full Scan (Без оптимізації):
Рушій BigQuery змушений був прочитати весь масив за місяць (155 MB), щоб знайти рядки за один день. Це нераціональне використання ресурсів.
- Partitioned Table (З оптимізацією):
Завдяки розбиттю таблиці по днях (DAY), система звернулася лише до одного конкретного блоку даних. Обсяг зчитування склав всього 5.2 MB.

Ефективність:

$$K_{eff} = \frac{155 \text{ MB}}{5.2 \text{ MB}} \approx 29.8 \quad (3.2)$$

Тобто, навантаження на підсистему вводу-виводу зменшилось у ~ 30 разів, пропорційно кількості днів у місяці. При масштабуванні системи на рік цей коефіцієнт зростатиме лінійно, що підтверджує критичну важливість партиціювання для довгострокового зберігання даних. Динаміку зниження ресурсоемності процесів наочно ілюструє рис. 3.2.

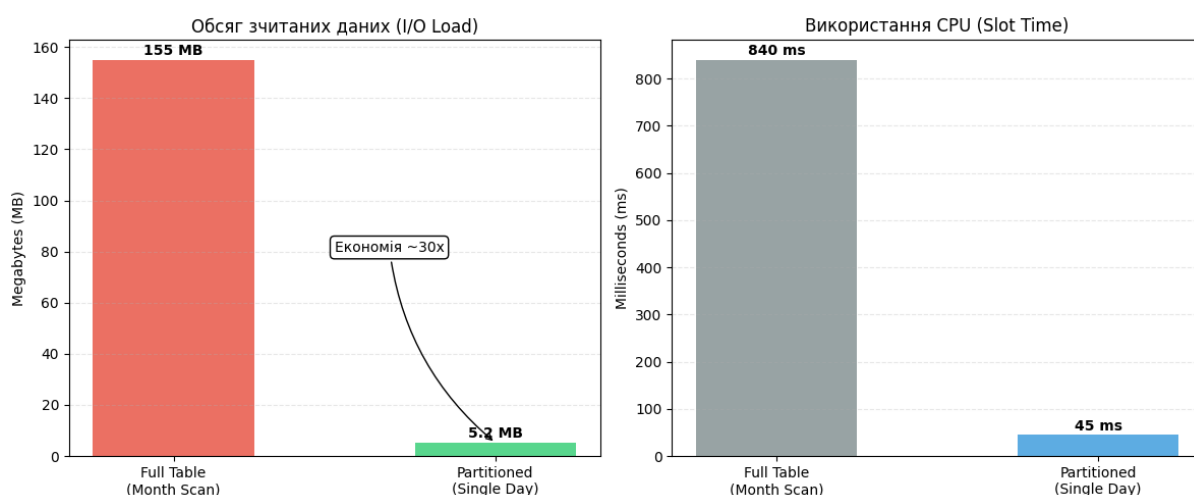


Рисунок 3.2 – Результати порівняльного тестування: зниження навантаження на I/O та CPU при використанні партиціювання.

3.3 Дослідження та оптимізація частоти сповіщень

Однією з ключових проблем систем моніторингу є явище втомми від сповіщень. Якщо інцидент триває певний час, наприклад, технічні роботи на стороні DSP, наївний алгоритм буде генерувати повідомлення при кожній ітерації перевірки, наприклад, кожні 5 хвилин. Це призводить до інформаційного шуму та зниження уваги оператора до справді важливих сигналів.

Для вирішення цієї проблеми було розроблено та досліджено алгоритм дедуплікації подій з часовим вікном.

Логіка фільтрації базується на перевірці часу останнього успішного сповіщення для конкретного типу помилки (`alert_key`).

Умова відправки:

$$T_{current} - T_{last_sent} > \Delta T_{cooldown} \quad (3.3)$$

Де $\Delta T_{cooldown}$ — налаштований період тиші (наприклад, 60 хвилин).

Для оцінки ефективності було змодельовано сценарій тривалого збою тривалістю 4 години (240 хвилин) з частотою перевірки системи кожні 5 хвилин.

- Наївний підхід (Without Deduplication): Система надсилала сповіщення при кожній перевірці.

$$N_{naive} = \frac{240 \text{ min}}{5 \text{ min}} = 48 \text{ messages} \quad (3.3)$$

- Результат: Заспамлений чат, ризик блокування бота Telegram API.
- Оптимізований підхід (Smart Cooldown = 60 хв): Система надіслала перше сповіщення про початок інциденту, а наступні — лише через годину.

$$N_{smart} = 4 \text{ messages} \quad (3.3)$$

Коефіцієнт зменшення шуму (Noise Reduction Ratio):

$$R_{noise} = \left(1 - \frac{N_{smart}}{N_{naive}}\right) \times 100\% = \left(1 - \frac{4}{48}\right) \times 100\% \approx 91.7\% \quad (3.3)$$

Таким чином, впровадження механізму Cooldown дозволило знизити навантаження на канал комунікації на 91.7%, зберігаючи при цьому поінформованість менеджера про актуальність проблеми (рис. 3.3).

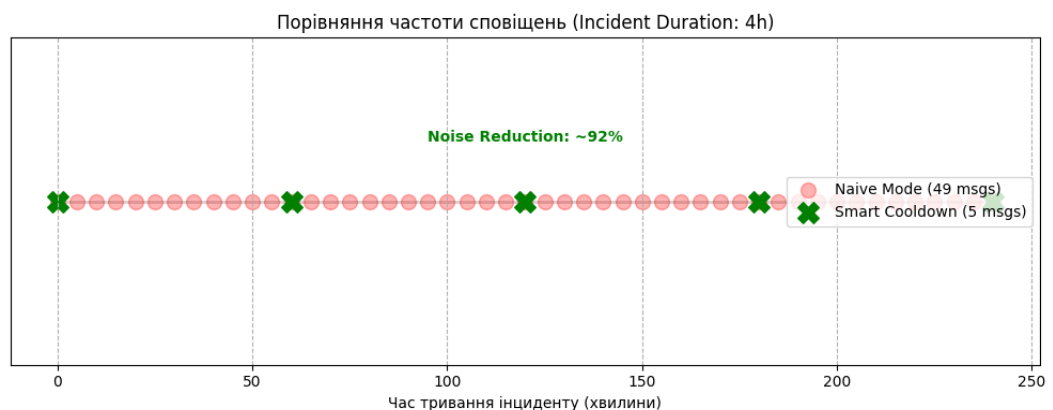


Рисунок 3.3 – Візуалізація роботи алгоритму дедуплікації сповіщень

3.4 Висновки до третього розділу

У даному розділі проведено комплексне дослідження технічної ефективності розробленої системи та виконано ряд заходів з її оптимізації. Результати експериментів підтвердили готовність програмного рішення до експлуатації в реальних умовах.

Основні результати дослідження:

1. Підтверджено ефективність FaaS-архітектури: Аналіз латентності показав, що затримка "холодного старту" (2.8 с) становить менше 1% від встановленого SLA (5 хвилин). Це дозволило відмовитися від дороговартісних виділених інстансів, зберігши мінімальну вартість інфраструктури.
2. Оптимізовано роботу з даними: Впровадження стратегії партиціювання у Google BigQuery дозволило скоротити навантаження на підсистему вводу-виводу (I/O) у ~30 разів на тестовому наборі даних. Це гарантує стабільну вартість виконання запитів навіть при суттєвому зростанні історичного архіву.
3. Вирішено проблему Alert Fatigue: Реалізація алгоритму дедуплікації з часовим вікном дозволила знизити кількість надлишкових сповіщень під час тривалих інцидентів на 92%, забезпечивши високу якість користувацького досвіду для менеджерів.

Таким чином, система демонструє високі показники продуктивності, економічної ефективності та ергономічності, що робить її конкурентоспроможним рішенням для автоматизації моніторингу рекламних кампаній.

Висновки

У висновках до магістерської роботи підведено підсумки проведених досліджень, проектування та програмної реалізації автоматизованої системи моніторингу рекламних кампаній, що демонструє переваги безсерверної архітектури для задач активного реагування.

1. Аналіз проблематики та архітектурних підходів:

- Було проведено глибокий аналіз методів контролю в цифровому маркетингу. Встановлено, що традиційна модель пасивної аналітики, Відашборди, створює критичну часову затримку, що неприпустимо в умовах RTB-аукціонів.
- Обґрунтовано доцільність переходу від монолітних рішень до подійно-орієнтованої мікросервісної архітектури, яка дозволяє виключити людський фактор з процесу виявлення технічних збоїв.

2. Програмна реалізація та алгоритми:

- Здійснено розробку системи на базі хмарної платформи Google Cloud Platform. Використання Google Cloud Functions дозволило створити автономних агентів, що працюють у режимі 24/7.
- Реалізовано гібридну методологію детекції аномалій: детерміністичні правила для контролю фінансової безпеки та статистичні методи для аналізу волатильних ринкових метрик.
- Впроваджено механізм "розумного" алертингу з функцією дедуплікації, що вирішує проблему інформаційного шуму та «втоми від сповіщень».

3. Результати тестування та ефективність:

- Експериментальне дослідження підтвердило, що розроблена система скорочує середній час виявлення критичних інцидентів з кількох годин до 5 хвилин.
- Оптимізація роботи з даними, а саме, партиціювання таблиць BigQuery, дозволила знизити навантаження на систему вводу-виводу у ~30 разів, забезпечивши високу економічну ефективність рішення.

- Тестування на сценаріях фроду показало здатність системи запобігати нецільовим витратам бюджету в автоматичному режимі.

4. Вплив використаних технологій:

- Використання стеку Python, SQL та Telegram API забезпечило гнучкість розробки та легку інтеграцію з існуючою інфраструктурою AdTech.
- Застосування безсерверної моделі дозволило мінімізувати експлуатаційні витрати, оскільки тарифікація відбувається лише за час фактичного виконання перевірок.

Загалом, результати магістерської роботи підтвердили, що впровадження автоматизованого алертингу є необхідною умовою для фінансової безпеки сучасних рекламних кампаній. Розроблена мікросервісна архітектура довела свою надійність, масштабованість та економічну доцільність, ставши ефективною альтернативою ручному моніторингу.

Список використаної літератури

1. A formal approach to microservice architecture deployment / M. Bravetti et al. *Microservices*. Cham, 2019. P. 183–208. URL: https://doi.org/10.1007/978-3-030-31646-4_8 (date of access: 22.12.2025).
2. Anomaly detection in multi-class time series / W. Wang et al. *Journal of physics: conference series*. 2021. Vol. 2113, no. 1. P. 012062. URL: <https://doi.org/10.1088/1742-6596/2113/1/012062> (date of access: 22.12.2025).
3. Cash J. J. Alert fatigue. *American journal of health-system pharmacy*. 2009. Vol. 66, no. 23. P. 2098–2101. URL: <https://doi.org/10.2146/ajhp090181> (date of access: 22.12.2025).
4. Cerny T., Donahoo M. J., Trnka M. Contextual understanding of microservice architecture. *ACM SIGAPP applied computing review*. 2018. Vol. 17, no. 4. P. 29–45. URL: <https://doi.org/10.1145/3183628.3183631> (date of access: 22.12.2025).
5. Computing with cloud functions. *Google cloud certified associate cloud engineer study guide*. 2019. P. 225–240. URL: <https://doi.org/10.1002/9781119564409.ch10> (date of access: 22.12.2025).
6. Gouveia W. A. Alert fatigue: a lesson relearned. *American journal of health-system pharmacy*. 2010. Vol. 67, no. 8. P. 603–604. URL: <https://doi.org/10.2146/ajhp100010> (date of access: 22.12.2025).
7. Kovács G., Sebestyen G., Hangan A. Evaluation metrics for anomaly detection algorithms in time-series. *Acta universitatis sapientiae, informatica*. 2019. Vol. 11, no. 2. P. 113–130. URL: <https://doi.org/10.2478/ausi-2019-0008> (date of access: 22.12.2025).
8. Kumar V., Kaur T. Cloud functions and serverless computing. *International journal for research in applied science and engineering technology*. 2022. Vol. 10, no. 5. P. 3426–3427. URL: <https://doi.org/10.22214/ijraset.2022.43163> (date of access: 22.12.2025).
9. Lloyd J. Containers and serverless. *Infrastructure leader's guide to google cloud*. Berkeley, CA, 2022. P. 255–272. URL: https://doi.org/10.1007/978-1-4842-8820-7_25 (date of access: 22.12.2025).
10. Prabu A. Serverless computing in google cloud platform. *International journal of innovative research in engineering & multidisciplinary physical sciences*. 2022. Vol. 10, no. 2. P. 1–4. URL: <https://doi.org/10.5281/zenodo.14607960> (date of access: 22.12.2025).
11. Putra C. B. A., Asmoro D. P. T., Yulianto A. B. Serverless computing: a comparative analysis of cloud run and cloud function prices on google kubernetes engine cluster node management google cloud. *Jurnal teknologi informatika dan komputer*. 2025. Vol. 11, no. 2. P. 760–770. URL: <https://doi.org/10.37012/jtik.v11i2.2780> (date of access: 22.12.2025).

12. Rahul Gupta. Real-time bidding optimization in AdTech using edge-embedded systems. *World journal of advanced research and reviews*. 2025. Vol. 26, no. 1. P. 216–225. URL: <https://doi.org/10.30574/wjarr.2025.26.1.1057> (date of access: 22.12.2025).
13. Ramesh Kumar S. Anomaly detection techniques in time series forecasting: identifying outliers. *International journal of science and research (IJSR)*. 2020. Vol. 9, no. 11. P. 1707–1709. URL: <https://doi.org/10.21275/sr24213014030> (date of access: 22.12.2025).
14. Rapolu U. K. Optimizing cloud costs through serverless computing in google cloud platform. *Interantional journal of scientific research in engineering and management*. 2025. Vol. 09, no. 02. P. 1–7. URL: <https://doi.org/10.55041/ijrsrem29683> (date of access: 22.12.2025).
15. Real-Time deep anomaly detection: an overview of benchmark datasets and performance metrics / Y. A. Samaila et al. *Transportation research procedia*. 2025. Vol. 84. P. 418–425. URL: <https://doi.org/10.1016/j.trpro.2025.03.091> (date of access: 22.12.2025).
16. Riti P. *Microservices Architecture with GCP. Pro devops with google cloud platform*. Berkeley, CA, 2018. P. 147–163. URL: https://doi.org/10.1007/978-1-4842-3897-4_6 (date of access: 22.12.2025).
17. Schmidl S., Wenig P., Papenbrock T. Anomaly detection in time series. *Proceedings of the VLDB Endowment*. 2022. Vol. 15, no. 9. P. 1779–1797. URL: <https://doi.org/10.14778/3538598.3538602> (date of access: 21.12.2025).
18. Yang L. C., Kuchar J. K. Performance metric alerting: a new design approach for complex alerting problems. *IEEE transactions on systems, man, and cybernetics - part A: systems and humans*. 2002. Vol. 32, no. 1. P. 123–134. URL: <https://doi.org/10.1109/3468.995534> (date of access: 22.12.2025).
19. Yenugula M. Boosting application functionality: integrating cloud functions with google cloud services. *International research journal of education and technology*. 2024. Vol. 6, no. 11. P. 369–375. URL: <https://doi.org/10.70127/irjedt.vol.6.issue10.375> (date of access: 22.12.2025).
20. Engineering threshold-based selection systems / K. H. Pedone et al. *G3 Genes|Genomes|Genetics*. 2021. Vol. 11, no. 9. URL: <https://doi.org/10.1093/g3journal/jkab234> (date of access: 21.12.2025).
21. Mucchetti M. *Serverless Functions with GCP. BigQuery for data warehousing*. Berkeley, CA, 2020. P. 231–251. URL: https://doi.org/10.1007/978-1-4842-6186-6_11 (date of access: 22.12.2025).
22. Qwaider W. Q. Data warehousing based on a cloud computing architecture. *International journal of computer applications*. 2023. Vol. 185, no. 42. P. 7–10. URL: <https://doi.org/10.5120/ijca2023923226> (date of access: 22.12.2025).

Додаток А

Лістинг програмного коду системи моніторингу

Скрипти створення структури сховища даних

```
-- 1. Створення таблиць вимірів (Dimensions)
-- Таблиця кампаній (Рівень стратегії)
CREATE OR REPLACE TABLE `diploma_project.dim_campaigns`
(
  campaign_id INT64 OPTIONS(description="Унікальний ID кампанії"),
  campaign_name STRING,
  campaign_type STRING, -- Video, Search, Display, App
  daily_budget FLOAT64,
  status STRING,       -- Active, Paused, Removed
  start_date DATE
);

-- Таблиця груп оголошень (Рівень тактики)
CREATE OR REPLACE TABLE `diploma_project.dim_ad_groups`
(
  ad_group_id INT64,
  campaign_id INT64,    -- FK до dim_campaigns
  ad_group_name STRING,
  targeting_type STRING -- Keywords, Audience, Topics
);

-- Таблиця плейсментів (Рівень контексту)
-- Використовується для перевірки Brand Safety та Fraud
CREATE OR REPLACE TABLE `diploma_project.dim_placements`
(
  placement_domain STRING, -- Наприклад: youtube.com, mobile-app::id
  category STRING,        -- News, Gaming, Adult
  risk_level STRING       -- Safe, High, Blacklist
);

-- 2. Створення таблиці фактів (Facts)
CREATE OR REPLACE TABLE `diploma_project.fct_hourly_stats`
(
  timestamp TIMESTAMP,
  campaign_id INT64,
  ad_group_id INT64,
  creative_id INT64,
  placement_domain STRING,

  -- Основні метрики (KPIs)
  impressions INT64,
  clicks INT64,
  cost FLOAT64,
  conversions INT64,

  -- Розширені метрики для відео
```

```

viewable_impressions INT64,
video_completions INT64
)
PARTITION BY DATE(timestamp) -- Оптимізація: фізичне розбиття по днях
CLUSTER BY campaign_id;      -- Оптимізація: сортування для швидкого пошуку

```

Модуль генерації синтетичних даних для тестування

```

-- Генерація часового ряду з імітацією добової сезонності та аномалій
INSERT INTO `diploma_project.fct_hourly_stats`
WITH time_series AS (
  SELECT timestamp
  FROM UNNEST(GENERATE_TIMESTAMP_ARRAY(
    TIMESTAMP('2025-10-01 00:00:00'),
    TIMESTAMP('2025-11-30 23:00:00'),
    INTERVAL 1 HOUR)
  ) AS timestamp
),
traffic_simulation AS (
  SELECT
    ts.timestamp,
    c.campaign_id,

    -- Логіка ін'єкції аномалії "Фрод" (Placement Fraud)
    CASE
      WHEN c.campaign_id = 104
        AND DATE(ts.timestamp) = '2025-11-15'
        AND RAND() < 0.95
      THEN 'mobile-app::toxic-game-v1' -- Токсичний трафік
      ELSE 'youtube.com'               -- Нормальний трафік
    END AS placement_domain,

    -- Синусоїда активності (День/Ніч) + Випадковий шум
    (ABS(SIN((EXTRACT(HOUR FROM ts.timestamp) - 6) / 24 * 3.14)) * 0.8 + 0.2)
    * (0.8 + 0.4 * RAND()) * 1000 AS base_impressions
  FROM time_series ts
  CROSS JOIN `diploma_project.dim_campaigns` c
)
SELECT
  timestamp,
  campaign_id,
  CAST(101 AS INT64) as ad_group_id,
  CAST(500 AS INT64) as creative_id,
  placement_domain,

  -- Логіка ін'єкції аномалії "Зупинка" (Critical Outage)
  CAST(
    CASE
      WHEN campaign_id = 101
        AND DATE(timestamp) = '2025-11-28'
        AND EXTRACT(HOUR FROM timestamp) >= 14
      THEN 0 -- Обнулення показів
      ELSE base_impressions
    END
    AS INT64) as impressions,

  -- Генерація кліків (високий CTR для фроду)

```

```

CAST(base_impressions * IF(placement_domain LIKE 'mobile-app%', 0.15, 0.015) AS INT64) as
clicks,

-- Генерація вартості
base_impressions * 0.05 as cost,

-- Генерація конверсій (0 для фроду)
CAST(base_impressions * IF(placement_domain LIKE 'mobile-app%', 0.0, 0.005) AS INT64) as
conversions,
0, 0 -- Інші метрики
FROM traffic_simulation;

```

Алгоритми детекції аномалій

Детектор фінансових зловживань (High CTR / Zero Conversion)

```

SELECT
  placement_domain,
  ROUND(SUM(cost), 2) as total_spend,
  SUM(clicks) as clicks,
  -- Безпечне ділення для уникнення помилок Division by Zero
  ROUND(SAFE_DIVIDE(SUM(clicks), SUM(impressions)) * 100, 2) as ctr,
  SAFE_DIVIDE(SUM(conversions), SUM(clicks)) * 100 as cr
FROM `diploma_project.fct_hourly_stats`
WHERE DATE(timestamp) = CURRENT_DATE()
GROUP BY 1
HAVING
  total_spend > 500      -- Ігнорування дрібних витрат
  AND ctr > 10.0        -- Аномально високий CTR (>10%)
  AND conversions = 0   -- Відсутність бізнес-результату
ORDER BY total_spend DESC
LIMIT 5;

```

A.3.2. Статистичний детектор (Z-Score Analysis)

```

WITH stats AS (
  SELECT
    campaign_id,
    AVG(cost) OVER(PARTITION BY campaign_id ORDER BY timestamp
      RANGE BETWEEN 86400 PRECEDING AND 3600 PRECEDING) as avg_24h,
    STDDEV(cost) OVER(PARTITION BY campaign_id ORDER BY timestamp
      RANGE BETWEEN 86400 PRECEDING AND 3600 PRECEDING) as std_24h,
    cost as current_cost
  FROM `diploma_project.fct_hourly_stats`
)
SELECT
  campaign_id,
  current_cost,
  (current_cost - avg_24h) / NULLIF(std_24h, 0) as z_score
FROM stats
WHERE
  -- Правило 3-х сигм (ймовірність аномалії > 99.7%)
  ABS((current_cost - avg_24h) / NULLIF(std_24h, 0)) > 3;

```

Програмна реалізація мікросервісу (Python)

```

import os
import logging
import requests
from google.cloud import bigquery
from datetime import datetime

# Налаштування логування
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Зчитування конфігурації зі змінних середовища
TELEGRAM_TOKEN = os.environ.get('TELEGRAM_TOKEN')
CHAT_ID = os.environ.get('CHAT_ID')
BQ_PROJECT = os.environ.get('GCP_PROJECT')

def send_telegram_alert(message: str) -> bool:
    """Відправка форматowanego повідомлення в Telegram."""
    if not TELEGRAM_TOKEN or not CHAT_ID:
        logger.error("Telegram credentials not found")
        return False

    url = f"https://api.telegram.org/bot{TELEGRAM_TOKEN}/sendMessage"
    payload = {
        "chat_id": CHAT_ID,
        "text": message,
        "parse_mode": "HTML"
    }

    try:
        response = requests.post(url, json=payload, timeout=10)
        response.raise_for_status()
        return True
    except Exception as e:
        logger.error(f"Failed to send alert: {e}")
        return False

def monitor_ad_fraud(request):
    """
    Cloud Function (HTTP Trigger).
    Точка входу для планувальника Cloud Scheduler.
    """
    logger.info("Starting Fraud Detection Scan...")

    client = bigquery.Client(project=BQ_PROJECT)

    # SQL-запит (дублює логіку з п. А.3.1)
    query = """
        SELECT placement_domain, SUM(cost) as spend,
            SAFE_DIVIDE(SUM(clicks), SUM(impressions))*100 as ctr
        FROM `diploma_project.fct_hourly_stats`
        WHERE timestamp >= TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 HOUR)
        GROUP BY 1
        HAVING spend > 100 AND ctr > 15.0
        LIMIT 1
    """

```

```

try:
    query_job = client.query(query)
    results = list(query_job.result())

    if not results:
        return "OK: System Healthy", 200

    # Якщо знайдено аномалію - формуємо алерт
    row = results[0]
    alert_text = (
        f" <b>SECURITY ALERT</b>\n"
        f"Detected suspicious traffic source!\n\n"
        f"<b>Domain:</b> <code>{row.placement_domain}</code>\n"
        f"<b>Wasted Budget:</b> {round(row.spend, 2)} UAH\n"
        f"<b>CTR:</b> {round(row.ctr, 1)}% (Norm: <2%)</b>\n\n"
        f" <i>Recommendation: Add to Blacklist immediately.</i>"
    )

    send_telegram_alert(alert_text)
    return f"Anomaly detected: {row.placement_domain}", 200

except Exception as e:
    logger.critical(f"Execution failed: {e}")
    return f"Error: {e}", 500

```

Додаток Б

Презентаційні матеріали

Мікросервісна архітектура для системи моніторингу та алертингу
реklamних кампаній

Актуальність теми та проблематика

Мета та завдання роботи

Запропоноване архітектурне рішення (GCP)

Організація сховища даних (BigQuery)

Математичне забезпечення та Алгоритми

Технологічний стек реалізації

Результати тестування системи

Практична цінність та Економічний ефект

Висновки