Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра мережних технологій факультету інформатики

Кваліфікаційна робота

освітній ступінь — магістр

на тему: «**ПОБУДОВА МЕРЕЖЕВОГО ЗАСТОСУВАННЯ З ВИСОКОЮ ДОСТУПНІСТЮ НА ХМАРНІЙ ПЛАТФОРМІ**»

Виконав: студент 2-го року
освітньої програми «Інженерія програмного забезпечення»
спеціальності 121 «Інженерія програмного забезпечення»
Кириченко Євгеній Борисович
Керівник: Черкасов Д.І., кандидат тех.наук, ст.викладач

Рецензент_____

Кваліфікаційна робота захищена з оцінкою _____

Секретар ЕК _____

«__»_____2025 р.

Київ 2025

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

«КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра мережних технологій факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри мережних технологій,

проф, доктор фіз.-мат.наук

_____ Малашонок Г.І.

(підпис)

„____”_____2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту 2 року навчання магістерської програми “Інженерія програмного забезпечення”

Кириченку Євгенію Борисовичу

на тему: «**Побудова мережевого застосування з високою доступністю на хмарній платформі**»

Зміст текстової частини до кваліфікаційної роботи:

Зміст

Анотація

Вступ

1. Огляд існуючих рішень
2. Проектування структури власного рішення
3. Реалізація компонентів системи

Висновки

Список використаних джерел

Додатки

Дата видачі: “___”_____2022 р.

Керівник: к.т.н. Черкасов Д.І. _____(підпис)

Завдання отримав: Кириченко Є.Б _____(підпис)

Тема: Побудова мережевого застосування з високою доступністю на хмарній платформі.

Календарний план виконання роботи:

п/п	Назва етапу роботи	Термін виконання етапу	Примітка
1.	Отримання теми кваліфікаційної роботи	12.12.2024	
2.	Пошук і ознайомлення з джерелами	15.03.2025	
3.	Написання вступу	22.03.2025	
4.	Написання першого розділу	24.03.2025	
5.	Написання другого розділу	04.04.2025	
6.	Написання третього розділу	14.04.2025	
7.	Написання висновку	21.04.2025	
8.	Захист кваліфікаційної роботи	10.06.2025	

Студент _____

Керівник _____

“____” _____

Зміст

Анотація

Метою роботи є розробка рішення для цифрового бюро знахідок, яке мало би високу доступність. Додаток даватиме людям можливість знаходити загублені речі. Для реалізації задуму проаналізовано аналогічні українські рішення, здійснено огляд основних компонентів архітектури веб-додатків, розглянуто інструменти хмарної платформи AWS. Було створено оптимальне структурне рішення, описано компоненти та їх взаємодію, пояснено типові сценарії взаємодії користувача з системою. Детально розроблено модулі пошуку схожих речей по фотографії, нотифікації та очистки бази даних.

Вступ

У повсякденному житті люди щодня гублять особисті речі - ключі, гаманці, документи та безліч інших. Проте не завжди вдається їх знайти, адже ніша цифрових бюро знахідок, тобто додатків для комунікації між тими, хто загубив та знайшов річ досить порожня в Україні. Звісно, існують подібні додатки, проте їх функціонал обмежений, а користувацький досвід не завжди приємний. Попри цифровізацію багатьох речей в Україні - створити універсальне цифрове бюро знахідок так і не вдалось. Платформа мала б надавати користувачам можливість швидко розміщувати оголошення про втрати та знахідки, мати хороший автоматизований пошук та надавати користувачам інструменти взаємодії один з одним.

Метою даної дипломної роботи є створення архітектури веб-застосунку «Цифрове бюро знахідок», який дозволить користувачам розміщувати оголошення, здійснювати пошук, вести комунікацію, і реалізація окремих компонентів системи.

Для досягнення поставленої мети необхідно зробити наступне:

- Провести аналіз аналогічних існуючих рішень та визначити їхні переваги й недоліки.
- Розглянути важливі компоненти для побудови веб-застосунків.
- Розглянути обчислювальні технології та інструменти платформи AWS
- Сформувати технічне завдання для цифрового бюро знахідок.
- Розробити масштабовану високодоступну архітектуру веб-застосунку.
- Забезпечити автентифікацію та авторизацію.
- Забезпечити моніторинг.
- Детально реалізувати окремі компоненти системи

Архітектура сервісу базуватиметься на хмарній інфраструктурі Amazon Web Services (AWS), що забезпечить масштабованість, високу продуктивність застосунку та доступність. Використання хмарних сервісів, таких як AWS, має ряд переваг у порівнянні з використанням власної, виділеної інфраструктури:

- Економія у випадку розумного використання. Не потрібно залучати велику кількість людей та витрачати велику кількість матеріальних ресурсів на побудову, розгортання та підтримку інфраструктури на власних серверах.
- Гнучкість, тобто можливість динамічно збільшувати чи зменшувати кількість виділених ресурсів.
- Плата тільки за ті ресурси, які використовуються. Якщо у випадку виділеної інфраструктури надлишково придбані сервери будуть просто простоювати, то при використанні хмарної надлишкові ресурси можна видалити, після чого зменшиться вартість її підтримки.
- Наявність готових компонентів, які можна використати для:

- побудови архітектури застосунку: віртуальні машини, контейнери, лямбди, балансувальники навантаження, CDN, сховища даних, тощо.

- моніторингу, візуалізації.

- CI/CD.

- Хмарні сервіси мають високу надійність, досягти подібного власними силами - дуже важко.

До вищесказаного необхідно додати, що у зв’язку з війною на території України - мати виділену інфраструктуру стало більш ризиковано, адже ймовірність її знищення та появи необхідності перевезення зросла.

Проект має як соціальне значення — допомога людям у пошуку речей, так і технічну цінність — демонстрація сучасного підходу до розробки, проектування та розгортання веб-застосунку в хмарному середовищі.

1. Аналіз предметної області

В цьому розділі проаналізовано аналоги цифрових бюро знахідок, визначено їх слабкі місця та можливості для покращення. Після цього розглянуто веб архітектури та основні елементи багатошарових архітектур. Наступним кроком розглянуто способи забезпечення високої доступності. Далі здійснено огляд інструментарію AWS: основні обчислювальні технології та хмарні компоненти багатошарової архітектури.

1.1 Огляд аналогів цифрових бюро знахідок

Система об’єднує тих, хто знайшов, і тих, хто втратив речі. Якщо вони обидва є на платформі, то ймовірно, що річ буде знайдена та повернена власнику. Шукачі та ті, що знайшли створюють пости, після чого можуть знаходити один одного та мати спосіб комунікації. Також важливо, щоб пошук речей був ефективним та надавав користувачу релевантні відповіді.

Розглянемо тепер які функції надають аналоги. Було підібрано 3 сайти для аналізу:

- luckfind, посилання на цей сайт є на офіційному сайті Київської міської адміністрації
- poisk ua, сайт що найвище ранжується браузерами
- verni.com.ua, теж високо ранжований сайт з подібними функціями

Скріншоти та посилання на сайти розміщено у додатку А.

1.1.1 Luckfind

На головній сторінці можна додати знахідку та втрату. На десктопі на ній також присутня карта, яка довго та незрозуміло вантажиться. Є можливість реєстрації. Можна переглянути “останні оголошення”. Проте немає відображення всіх оголошень, відсутні фільтри за категоріями, є лише пошукова стрічка. Кількість результатів пошуку не відображається правильно, це завжди 0. Пошук ведеться як по назві, так і по опису, що є перевагою. При введенні у пошукову

стрічку присутній дивний suggest, який містить та відображає дуже багато пропозицій, які виходять далеко за межі іншого контенту сторінки (див. Додаток А). Присутня дивна пагінація, відображення якої внизу сторінки є поганим користувацьким досвідом. Є 3 локалізації - ru, en, ua. При додаванні знахідки чи втрати теж доступна мапа, щоб вказати точну локацію. Можна обрати категорію та ввести додаткову інформацію. Обов'язково треба вказати пошту та номер телефону. Інтерфейс не є інтуїтивним. Так, при додавання знахідки неясно які поля необхідно заповнити.

Важливим плюсом є безкоштовне додавання оголошень, з можливістю надання винагороди при знахідці. Навіть при створені знахідки без контрольного питання, його введення є обов'язковим. При ідентифікації втрати чи знахідки можна зв'язатись з автором публікації вказавши свою пошту та телефон. Присутній модуль для коментарів, який навіть за української локалізації містить весь текст російською, тому констатуємо проблеми з локалізацією. Російська та англійська локалізація насправді просто не працюють, проте наявні в інтерфейсі. Загалом, додаток вже має хороший базовий функціонал, проте є що покращувати: додати фільтри за категоріями, пошук по всім знахідкам, виправити баги з картою, зробити локалізацію або прибрати її з інтерфейсу, виправити баг пропозиції в пошуковій стрічці, пагінацію, підвищити зрозумілість інтерфейсу.

1.1.2 Verni

Одразу кидається в очі не найприємніший інтерфейс, на який не було витрачено багато сил. З позитивних моментів в інтерфейсі - є панель категорій. Функціональність пошуку була перевірена декілька разів в різні дні, і можна зробити висновок, що вона не працює. Є "останні знахідки" на головній сторінці. На сайті присутня велика кількість низькосортної реклами, що погіршує користувацький досвід. Як зв'язатись з автором оголошень неясно, принаймні без реєстрації. Оголошення про знахідку безкоштовне, але про втрату коштує 50-100 грн, що є значним недоліком даної платформи. Щоб створити навіть оголошення про знахідку необхідно авторизуватись. Загалом, додаток має багато недоліків, та

не може називатись робочим.

1.1.3 Poisk

На головній сторінці є актуальні знахідки, їх можна фільтрувати за категорією. Проте там у всіх категоріях, крім “Всі категорії” знахідки яким 4 роки, тому очевидно, що проблема у функціоналі, а не в тому, що останні 4 роки не було публікацій. Є пошук по всім знахідкам з фільтрами категорії, місця втрати, та дати. Попри те, що інтерфейс приємний візуально, є моменти де користувацький досвід не є продуманим, наприклад вибір міста серед сотень міст у випадаючому списку. Також не вистачає пошуку за назвою чи описом, як у luckfind. Зараз при виставленні дати доводиться шукати свою знахідку серед усіх втрат починаючи від вказаної дати, а це можуть бути тисячі знахідок. Присутні 2 локалізації - ru та ua. Обидві дійсно працюють. Публікація про втрату коштує 50 грн. Крім цього є функція автоматичного пошуку за 50 грн на місяць. Так як на аналіз аналогів не закладено до бюджету, неможливо перевірити чи робоча функція публікації, але можна припустити, що вона працює коректно. Публікація про знахідку безкоштовна. Треба вказати контактні дані, місто, фото (обов’язково). Вибрати точну локацію на мапі не можна. Також не можна додати контрольне запитання. Мав би бути огляд модератором, але судячи з усього його немає. Загалом, додаток є найкращим серед трьох, які були розглянуті, і має хороший функціонал. Проте є що покращити: виправити “актуальні знахідки”, додати пошук за назвою чи описом, додати контрольне запитання, подумати про додавання функції вибору локації на мапі.

1.1.4 Результати огляду аналогів

Отже, було розглянуто 3 аналоги цифрових бюро знахідок, та з’ясовано їх основні переваги та недоліки. Ціллю є спроектувати додаток який імплементуватиме наступний функціонал:

- пошук по всім знахідкам з фільтрами по категорії, місцю, даті, назві та опису (є простір для використання ШІ), та з пагінацією.

- актуальні оголошення (створені в останній тиждень).

- безкоштовне додавання оголошень, з контрольним питанням, та контактами для зв’язку.

- функція вибору локації на мапі.

- можливість прикріпити кілька фото до втрати і знахідки

- опціональна можливість залишити нагороду

Крім очевидних функцій можна додати наступні покращення:

- пошук за фото. Користувач вантажить фото втраченої речі, і отримує список схожих публікацій.

- інтерактивна мапа загублених речей.

- пуш-повідомлення при появі схожої знахідки.

- пошук речей за фото.

- гейміфікація: нагороди та бейджики за добрі справи, соціальний рейтинг.

Для проектування додатку будуть використанні переваги хмарної платформи AWS.

1.2 Огляд веб-архітектур

В підрозділі розглянуто класичну трирівневу та багатошарову архітектури, концепції мікросервісів та моноліту.

1.2.1 Трирівнева веб-архітектура

Типово архітектура містить три основні комоненти:

- веб-браузер.

- веб-сервер.

- базу даних.

До появи трирівневої архітектури була дворівнева, яка мала інтерфейс та базу даних. За такого підходу бізнес-логіка була частиною фронтенду або сервера бази даних. Зрозуміло, що бізнес-логіку не можна було змінювати та масштабувати незалежно. В тришаровій архітектурі кожен компонент виконує свою функцію, і може розроблятись та масштабуватись незалежно. Трирівнева

архітектура має рівень клієнта, рівень бізнес логіки та рівень даних. Запити надходять на сервер, який керує бізнес логікою, здійснює запити в базу даних, і повертає клієнту відповідь.

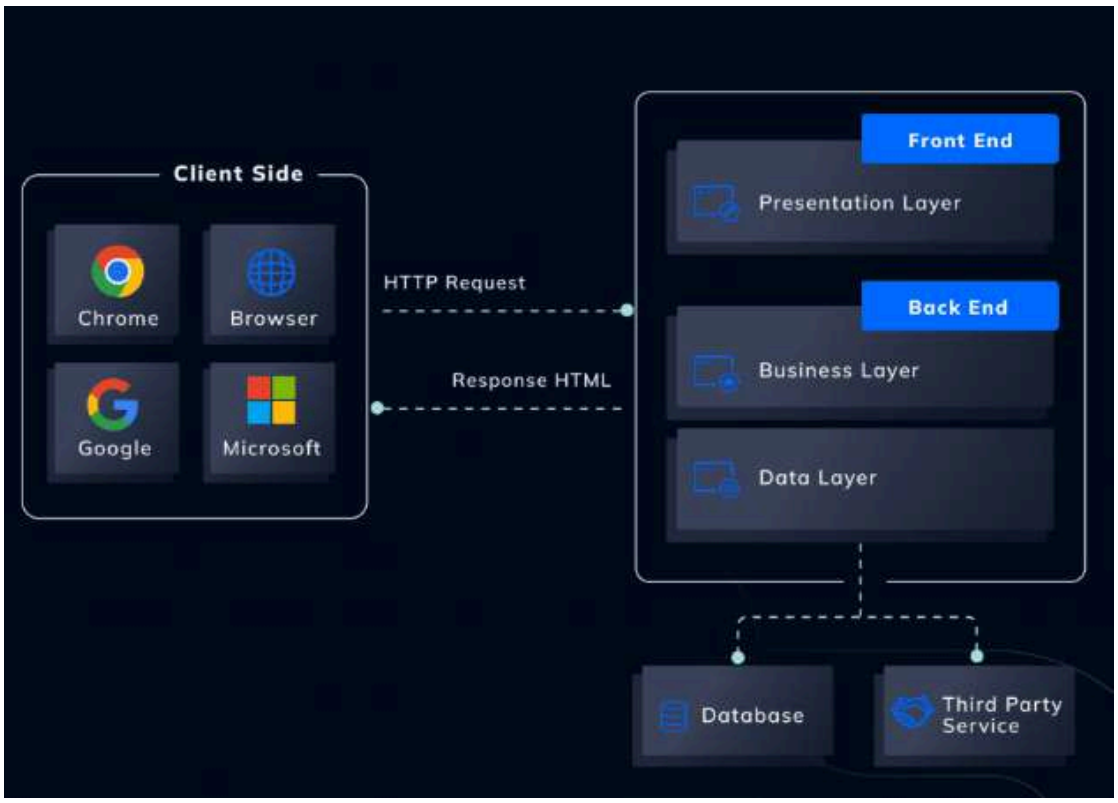


Рисунок 1.1. Тришарова архітектура.

1.2.2 Багаторівнева веб-архітектура

У класичній тришаровій архітектурі чітко розділені функції інтерфейсу, логіки обробки та зберігання даних. Але треба розуміти, що сучасні веб-додатки зазвичай мають більше шарів, тобто реалізуються як багаторівневі системи.

Багаторівнева архітектура передбачає додаткові рівні, кожен з яких виконує певну функцію [1]:

- DNS для зіставлення доменного імені та IP адреси сервера.
- CDN для швидкої доставки контенту.
- Load Balancer для розподілу трафіку між екземплярами серверів.
- API Gateway як єдина точка входження на сервер.
- рівень аутентифікації та авторизації.
- рівень кешування бази даних для швидкого повернення відповідей

(MemCached, Redis).

- рівень черги повідомлень для розділення логіки сервісів (Kafka, Rabbit MQ).

- рівень логування для відслідковування неполадок.

- рівень резервного копіювання для відновлення даних при неполадках.

- фаєрвол для захисту.

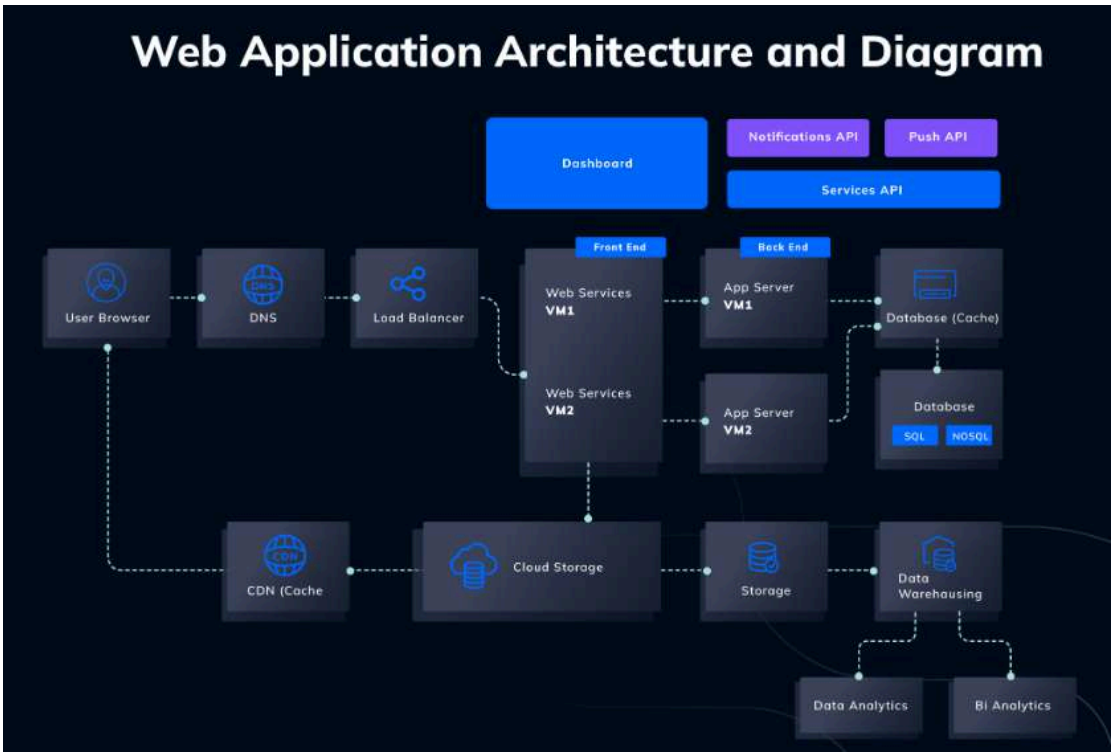


Рисунок 1.2. Багатошарова архітектура.

Цей підхід підвищує масштабованість, відмовостійкість, гнучкість розгортання та безпеку системи. Саме тому в дипломній роботі реалізовано багаторівневу архітектуру.

1.2.3 Моноліт проти мікросервісу

Можна виділити два архітектурні підходи до розробки додатків: моноліт та мікросервіси. Моноліт - традиційний спосіб, згідно з яким кожен шар будується як неподільна одиниця. У випадку тришарової архітектури вся бізнес-логіка буде в одному місці. При потребі зміни будь-якого функціоналу потрібно вносити зміну в одну й ту ж кодову базу.

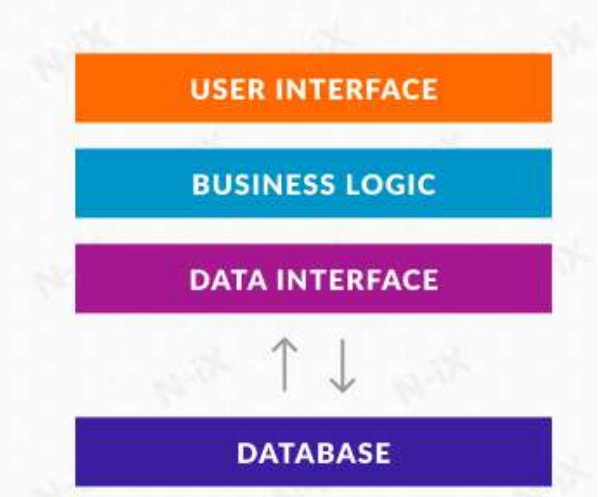


Рисунок 1.3. Монолітна архітектура.

Мікросервісна архітектура передбачає, що додаток складатиметься з декількох сервісів, які можуть спілкуватись за допомогою АПІ. Кожен сервіс має свою логіку, та може мати свою або спільну з іншими базу даних залежно від функціональних вимог.

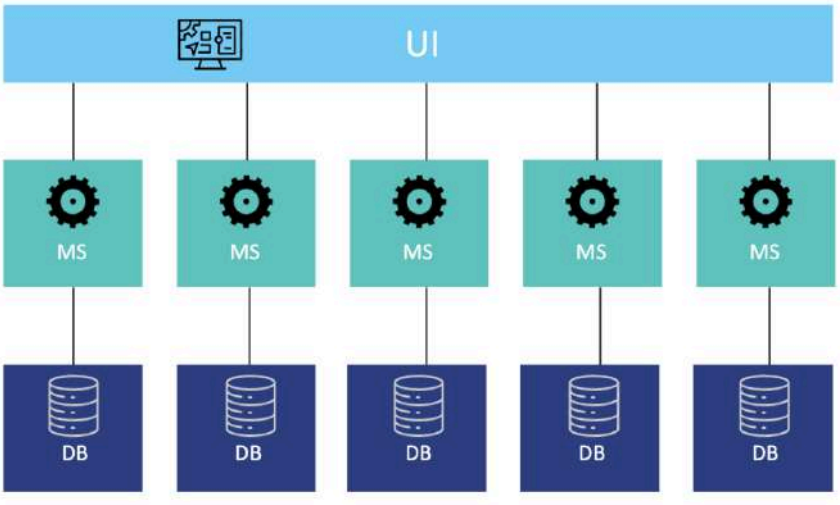


Рисунок 1.4. Мікросервісна архітектура.

Розглянемо переваги та недоліки кожної з архітектур [2]

Переваги монолітної архітектури:

- легко дебажити та тестувати взаємодію між елементами.

- мало cross-cutting concerns: легко виявляти помилки, налаштувати логування, небагато безпекових проблем порівняно з мікросервісами.

- легко деплоїти, не треба створювати окремий пайплайн для кожної одиниці.

- архітектура знайома всім, не потрібно витрачати час та гроші на навчання розробників.

Недоліки монолітної архітектури:

- багато коду в одному місці, відповідно важче додавати нові функції.

- неможливість окремого масштабування більш використовуваного функціоналу бізнес-логіки.

- висока зв’язність: зміни в одному компоненті викликають потребу у змінах в іншому.

Переваги мікросервісної архітектури:

- компоненти незалежні, відповідно їх можна розробляти, масштабувати, тестувати окремо від інших.

- легше вносити зміни в код порівняно з монолітом, адже кодова база менша.

- можливість використання різних засобів розробки для різних сервісів.

- якщо один компонент недоступний, інші все ще можуть бути доступними.

Недоліки мікросервісної архітектури:

- cross-cutting concerns.

- необхідність комунікації між сервісами.

- складність тестування, дебагу, відслідковування помилок.

- необхідність створювати CI/CD пайплайн для кожного сервісу .

Загалом, якщо додаток відносно простий, створюється однією людиною, його потрібно швидко розробити та запустити, то доречно використати монолітну архітектуру. Саме тому архітектура цифрового бюро знахідок буде монолітною.

1.3 Компоненти багатошарової веб-архітектури

Щоб спроектувати багатошарову архітектуру варто детальніше розібрати її

класичні компоненти, що й зроблено в даному підрозділі.

1.3.1 DNS

DNS(Domain name system) - ієрархічна система, що відповідає за перетворення доменних імен у IP адреси, які необхідні для маршрутизації запитів у інтернеті.

Процес перетворення працює наступним чином [3]:

1. Користувач вводить адресу сайту в адресний рядок браузера.
2. Браузер надсилає локальному DNS-клієнту запит з метою дізнатись IP адресу, яка відповідає доменному імені.
3. Якщо адреса не знайдена локально, запит передається до рекурсивного розпізнавача DNS. Той отримує запит, та шукає відповідні дані у себе.
4. Якщо він не знаходить їх в кеші DNS, то передає запит DNS серверу, що стоїть вище в ієрархії, починаючи з кореневого сервера, потім - до серверів доменів верхнього рівня (.org), і закінчуючи авторитетним сервером, який має відповідь.
5. Результат запиту, тобто потрібна IP адреса, проходить шлях по ланцюгу серверів назад до браузера. Проміжні сервери кешують результати запитів, щоб одразу відповісти користувачу наступний раз без необхідності здійснювати рекурсивні звернення до ієрархічно вищих DNS-серверів.

Весь процес схематично можна зобразити так:

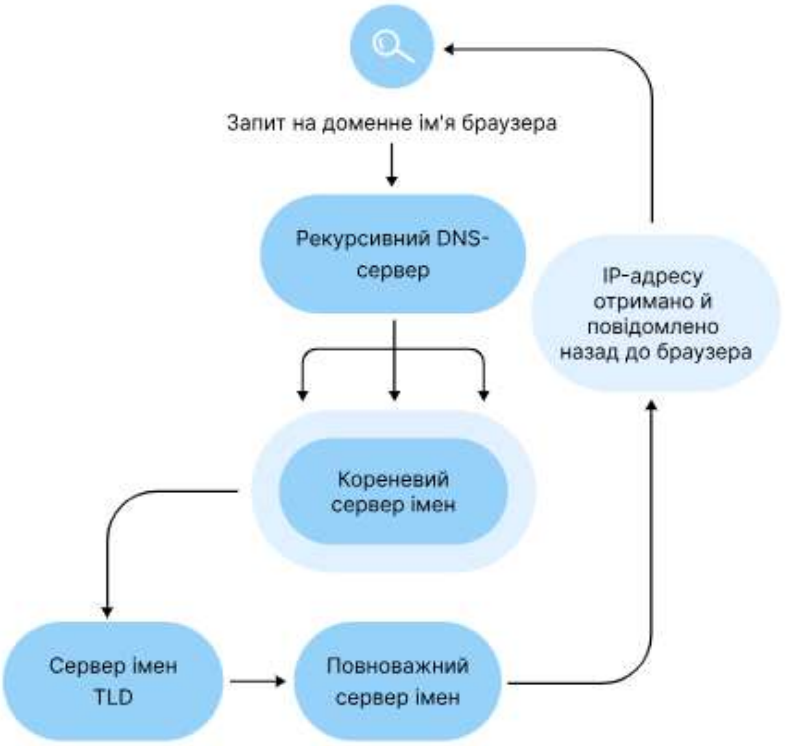


Рисунок 1.5. Архітектура DNS.

Цей механізм є важливою частиною інфраструктури інтернету. У сучасних архітектурах часто використовують спеціалізовані керовані служби DNS, які дозволяють реалізувати розподілене навантаження, моніторинг доступності, геолокаційні відповіді та автоматичне перемикання трафіку.

1.3.2 CDN

CDN (Content Delivery Network), або мережа доставки контенту, це мережа взаємопов’язаних серверів, яка пришвидшує процес завантаження веб-сторінки. Якщо користувач знаходиться далеко від веб-сервера, то буде велика затримка отримання відповіді при завантаженні великих файлів. CDN дозволяє зробити так, щоб користувач звертався до географічно найближчого CDN серверу, таким чином завантаження сторінки відбувається швидше. CDN фактично підвищує швидкість завантаження завдяки додаванню проміжних серверів між клієнтами та серверами. Крім цього CDN зменшують мережевий трафік на веб-серверах, збільшують пропускну здатність, підвищують доступність контенту, допомагають витримати DDoS атаку, та покращують користувацький досвід.

CDN складається з граничних серверів, які знаходяться в різних географічних локаціях, і працює на трьох принципах [4]:

- кешування, тобто процес зберігання декількох копій даних для швидшого доступу до них. Статичний контент веб-сайтів зберігається на багатьох серверах в мережі.

Кешування працює наступним чином:

1. Віддалений користувач робить запит статичного контенту.
2. Запит доходить до сервера. Сервер надсилає відповідь користувачу, і копію відповіді CDN серверу, який знаходиться найближче до користувача.
3. CDN сервер зберігає копію.
4. Коли будь-який користувач з цієї ж географічної локації звернеться за цими даними, то отримає їх від CDN сервера.

- динамічне прискорення, тобто зменшення часу відповіді від сервера при запиті динамічного контенту завдяки проміжним CDN серверам. Кешування не можна ефективно застосувати до динамічного контенту, адже він постійно змінюється. При отриманні кожного динамічного запиту, CDN сервер має робити запит до головного веб-сервера, проте такий запит швидше, ніж прямий запит від користувача до головного сервера, тому що з’єднання між CDN та головним серверами є оптимізованим.

- обчислення на граничних серверах, тобто можливість виконувати на граничному CDN сервері такі операції:

1. Змінювати поведінку кешування.
2. Валідувати запити та обробляти некоректні.
3. Модифікувати та оптимізовувати контент перед відправкою відповіді.

Розподіл логіки між головним сервером та граничним допомагає знизити навантаження на головний сервер, чим покращує продуктивність сайту.

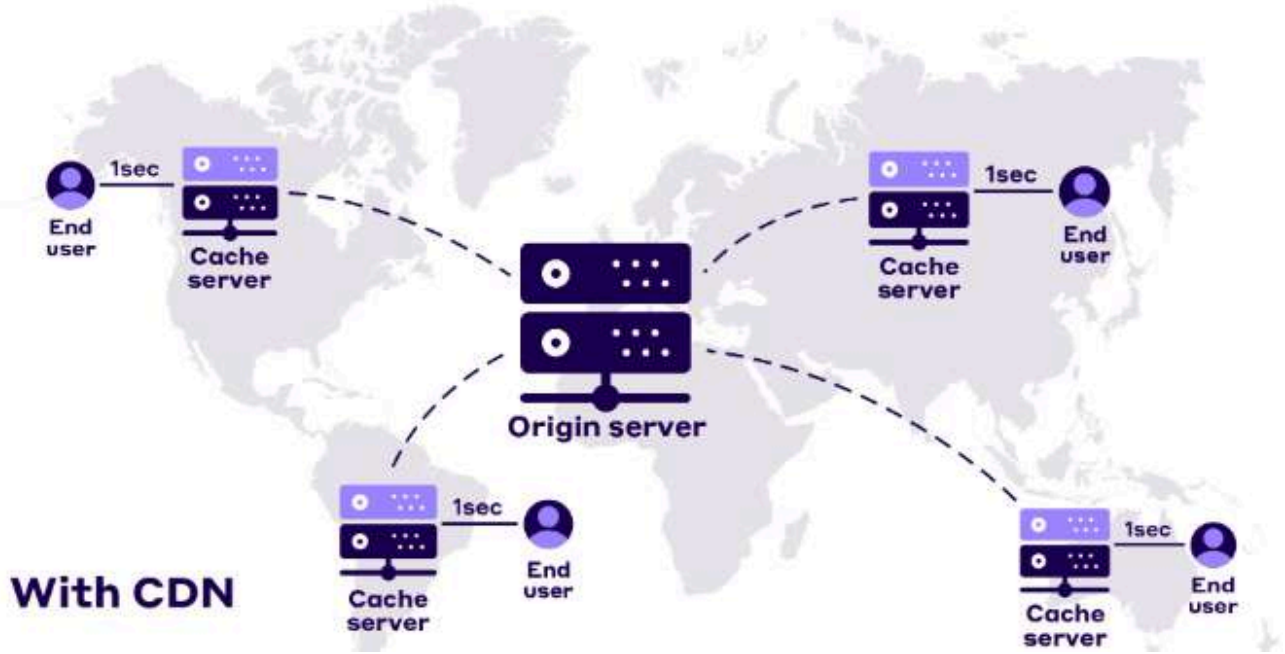


Рисунок 1.6. Архітектура CDN.

1.3.3 Load Balancer

Балансування навантаження - це розподіл обчислень між кількома серверами. У веб-архітектурі балансувальник навантаження розподіляє інтернет трафік. Це дає змогу покращити продуктивність додатку та знизити затримку.

Для розподілу запитів між серверами можна використовувати два основні типи алгоритмів [5]:

- статичні, тобто ті, які не враховують стан системи. Вони направляють запити чітко за планом, незалежно від того, чи сервер перевантажений. При використанні даного типу балансування є ймовірність, що деякі сервери будуть перевантажені, в той час як інші використовуватимуться недостатньо. Прикладом статичного алгоритму розподілу трафіку є round robin.

- динамічні, враховують доступність та навантаження кожного сервера. Здатні розподілити трафік таким чином, що зберігається рівномірний розподіл навантаження. Динамічні балансувальники моніторять стан та швидкість відповіді серверів. Якщо сервер відповідає повільно, то балансувальник направляє на нього менше трафіку. Якщо сервер виходить з ладу, то трафік перенаправляється на інші сервери. Динамічне балансування важче налаштувати, адже треба врахувати

розмір завдань, потужність та стан кожного сервера. Прикладом динамічних алгоритмів розподілу трафіку є least connection, weighted least connection, resource-based, та geolocation-based load balancing.

Балансувальник навантаження необхідний при високому навантаженні на систему .

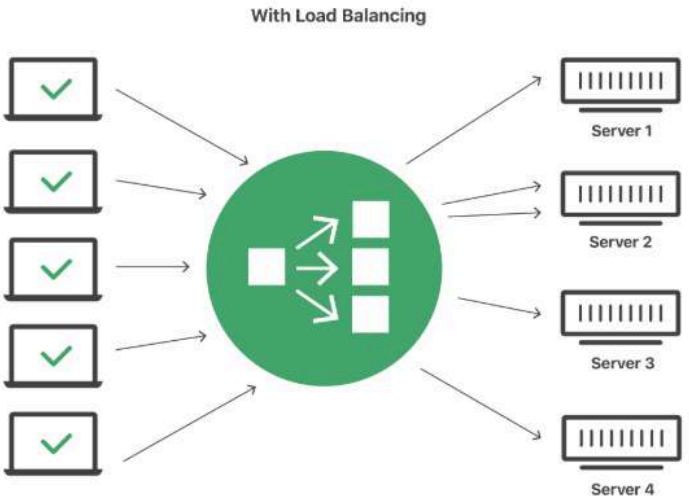


Рисунок 1.7. Архітектура балансувальника навантаження.

1.3.4 API Gateway

API Gateway є інструментом керування API, який знаходиться між клієнтом та бекенд сервісами і є єдиною точку входу для викликів API [6].

API Gateway виконує наступні функції:

- маршрутизація запитів у потрібний сервіс.

- аутентифікація та авторизація, тобто перевірка прав доступу користувача перед передачею запиту.

- обмеження частоти запитів, що запобігає перевантаженню бекенда.

- агрегація відповідей з кількох сервісів.

- кешування, що зменшує кількість звернень до сервера.

- моніторинг та логування.

До переваг API Gateway можна віднести:

- централізація керування безпекою та логікою доступу.

- зменшення кількості залежностей на боці клієнта.

- можливість поступового оновлення сервісів без зупинки всієї системи. Спершу на одному інстансі запускається нова версія сервісу. API Gateway перенаправить частину трафіку на інстанс з оновленою версією. Якщо все працює стабільно - оновлюються всі інстанси, інакше оновлення зупиняється або відкочується назад.

- зменшується зв'язність фронтенду та бекенду, що робить систему більш гнучкою.

До недоліків можна віднести складність впровадження, а саме:

- додаткова точка відмови, треба масштабувати.

- велика кількість сервісів ускладнює конфігурацію.

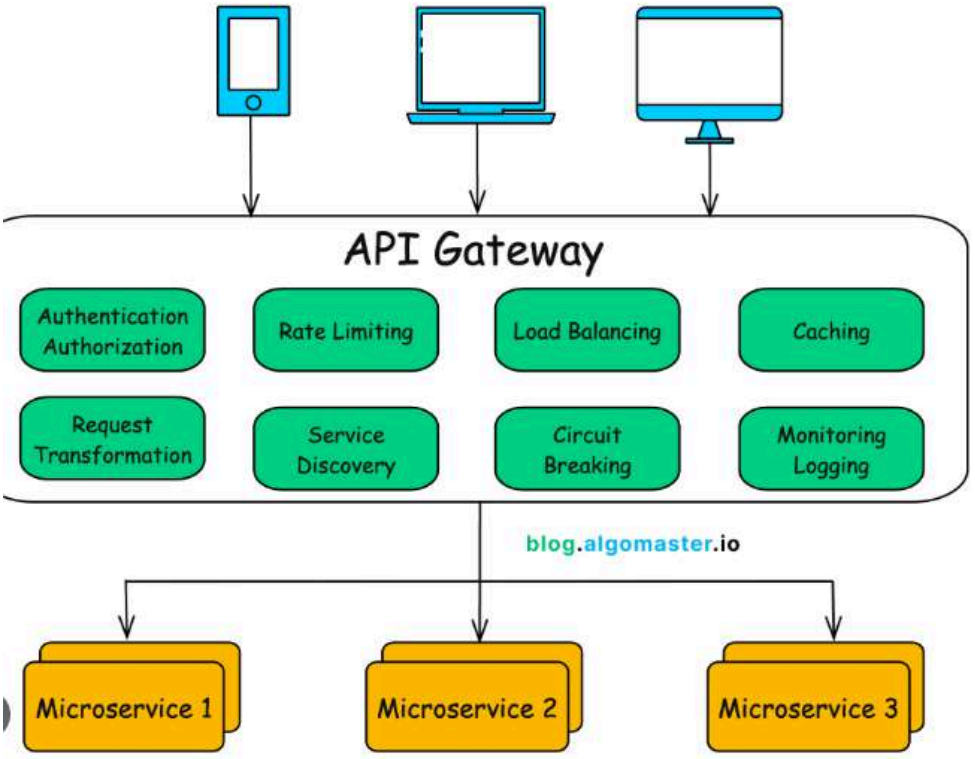


Рисунок 1.8. Функції API Gateway.

Загалом, на рівні API Gateway можна реалізувати багато корисного функціоналу, який здатний підвищити продуктивність додатку та покращити користувацький досвід як розробників, які використовують API, так і відвідувачів

сайту.

1.3.5 Кеш Бази Даних

Кешування бази даних - це техніка збереження даних, які часто зчитуються, у оперативній пам’яті. У типових додатках більшість запитів - це читання даних, які змінюються нечасто. Наприклад, інформація про користувача, популярні товари, результати пошуку. Такі дані доцільно кешувати, адже це зменшує кількість звернень до бази даних.

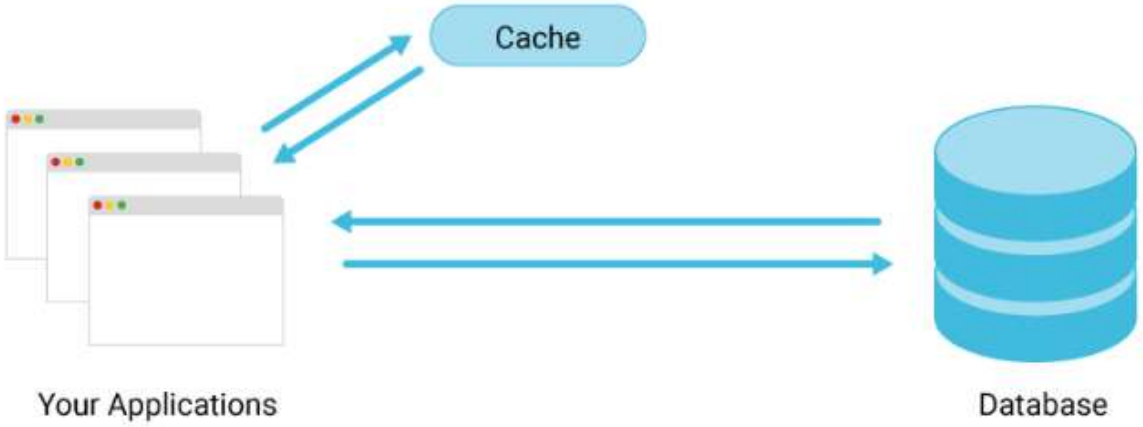


Рисунок 1.9. Кеш база даних.

Можна виділити наступні переваги кеш баз даних [7]:

- продуктивність, адже доступ до оперативної пам’яті значно швидший, ніж до дискової.

- зменшення навантаження на базу даних, адже кеш база даних сама повертає відповідь на деякі запити.

- краще масштабування, при великому навантаженні легше масштабувати кеш базу даних, ніж основну.

- зниження витрат, адже при використанні ефективного кешування зменшується потреба у використанні потужних баз даних.

Загалом, кеш база даних дозволяє підвищити продуктивність та зробити систему більш стійкою до пікових навантажень.

1.3.6 Черги повідомлень

Черги повідомлень - форма асинхронної комунікації між сервісами. Повідомлення зберігаються в черзі, поки не будуть оброблені та видалені. Кожне повідомлення обробляється тільки раз одним сервісом споживачем. Черга має ендпоінти, які дозволяють сервісам додавати та забирати повідомлення. Повідомленнями можуть виступати запити, відповіді, повідомлення про помилки, і будь-яка потрібна інформація.

Чергу повідомлень доцільно використати для [8]:

- зменшення зв’язності компонентів, а саме відправника та отримувача повідомлень. Компоненти стають більш незалежними, не звертаються один до одного напряму.

- балансування навантаження, тобто розподіл повідомлень між різними отримувачами.

- підвищення надійності, адже черга забезпечує, що повідомлення не загубляться навіть якщо сервіс отримувач тимчасово недоступний.

- асинхронної обробки, це може покращити користувацький досвід, адже сервер може швидко відповідати користувачам, тоді як завдання, які вимагають багато часу, відправляти в чергу, таким чином делегуючи іншому обробнику.

- масштабованості, зі зростанням навантаження на систему можна додати ще отримувачів не вносячи значні зміни в архітектуру.

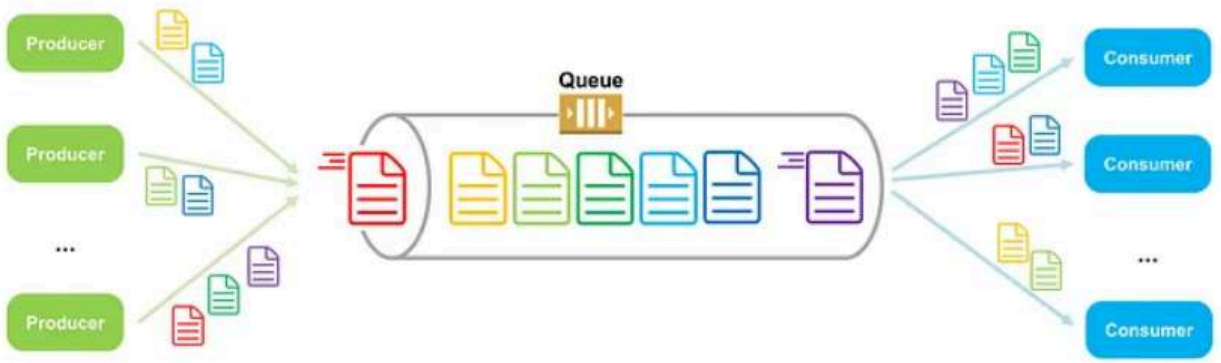


Рисунок 1.10. Архітектура черги.

Отже, алгоритм роботи черги наступний:

1) Сервіс відправник додає повідомлення в чергу.

2) Черга зберігає повідомлення.

3) Споживач читає повідомлення і обробляє. Їх може бути багато, і вони оброблятимуть повідомлення паралельно.

4) Після обробки повідомлення, споживач підтверджує успішне виконання завдання або ставить завдання в чергу повторно.

Загалом, черга повідомлень може підвищити надійність архітектури додатку та покращити користувацький досвід. Чергу можна використати для відкладених завдань, агрегації логів різних сервісів, комунікації мікросервісів, збору даних з різних джерел для збереження та подальшої обробки.

1.3.7 Моніторинг та логування

Моніторинг та логування важливі компоненти архітектури, які забезпечують спостережуваність системи. Вони дозволяють вчасно діагностувати проблеми та оптимізувати роботу додатку на основі отриманих даних.

[9] Моніторинг полягає у зборі, аналізі, та візуалізації важливих метрик роботи системи. Наприклад, корисно знати такі технічні метрики: використання пам’яті та процесора, час відповіді на запити, кількість помилок. Метрики можуть бути і бізнесовими: кількість успішних покупок, середній чек, кількість реєстрацій. Моніторинг дозволяє зрозуміти стан системи та виявити аномалії, тобто відхилення від норми.

[10] Логування фіксує події, які відбуваються в системі: запити, помилки, виклики функції, сповіщення. За допомогою логів можна аналізувати активність користувачів, активність системних процесів, нетипову активність та помилки системи. Важливо, щоб логи містили достатньо інформації, яка дає змогу зрозуміти проблему, але щоб вони не містили приватних та персональних даних користувачів.

Системи моніторингу та логування дозволяють аналізувати історію подій, передбачувати необхідність виділення додаткової інфраструктури, швидко виявляти проблеми та підвищувати надійність, доступність, і продуктивність

додатку.

1.3.8 Фаєрвол

Фаєрвол - це компонент безпеки мережі, який контролює вхідний та вихідний трафіки. Завдання фаєрволу - дозволяти доступ у мережу тільки санкціонованим джерелам. Фаєрволи перевіряють пакети даних, що приходять через мережу, і на основі визначених правил дозволяють або блокують доступ. Правила можуть ґрунтуватись на основі порту, протоколу, напрямку трафіку, часу, типу запиту, IP-адреси. Існують два типи фаєрволів [11]:

- мережеві, тобто ті, які розміщуються між мережами (наприклад приватними та локальними). Вони забезпечують базовий рівень захисту.

- прикладні, тобто ті, які аналізують трафік на рівні додатків. Виявляють такі типи загроз як SQL-ін’єкції та XSS атаки.

Фаєрволи можуть бути програмними, апаратними, або поєднувати два підходи. У хмарних середовищах використовують віртуальні фаєрволи, які інтегруються з сервісами безпеки. Використання фаєрволів є важливою частиною створення захищеної інфраструктури, адже знижують ризик витоку даних, зламу системи.

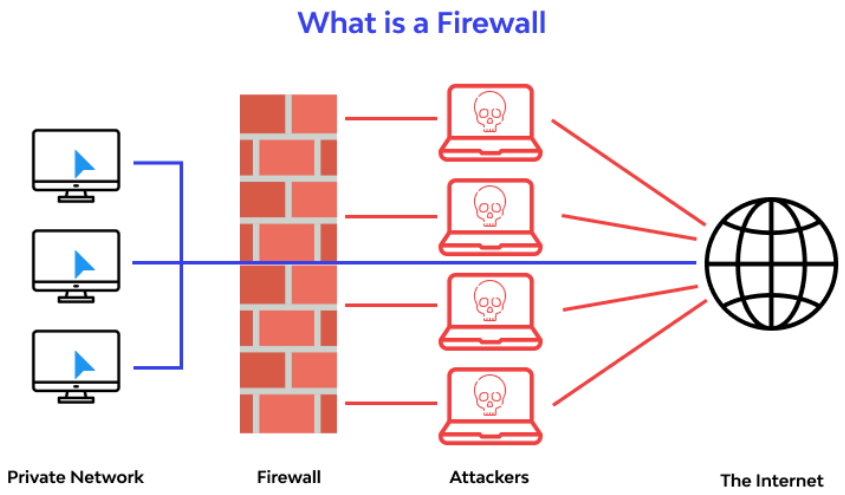


Рисунок 1.11. Архітектура Firewall.

1.4 Стратегії досягнення високої доступності

В сучасному світі будь-який даунтайм (час, коли система не працює) може вдарити по бізнесу, призвівши до великих фінансових та репутаційних втрат. Тому важливо знати як будувати розподілені системи з високою доступністю. Висока доступність означає, що система працює навіть тоді, коли відбуваються певні збої. Для забезпечення цього потрібно мінімізувати даунтайм, прибрати єдині точки відмови, імплементувати надлишковість. В розподілених системах кожен індивідуальний компонент може зазнати збоїв. Це в свою чергу може призвести до збоїв у всій системі. В цьому підрозділі буде розглянуто важливі принципи дизайну систем для забезпечення високої доступності.

1.4.1 Масштабованість та надійність

Є два ключових принципи в дизайні систем: масштабованість та надійність.

Висока масштабованість дозволяє системі впоратись зі зростаючим навантаженням без зниження продуктивності додатку. Дизайн системи має бути таким, щоб його можна було легко масштабувати для використання більшою кількістю користувачів.

Є два типи масштабованості:

- горизонтальна, тобто додавання серверів чи нод.

- вертикальна, тобто виділення більшої кількості ресурсів для ноди чи сервера.

Щоб система була масштабована важливо розглянути такі речі, як балансування навантаження, партиціонування, кешування, та оптимізацію бази даних.

Висока надійність орієнтована на зменшення ймовірності виникнення збоїв та на мінімізацію їх впливу на роботу системи. Важливо імплементувати механізми обробки помилок, спостерігати за компонентами, щоб передбачити збої, та організувати відновлення компонентів після збоїв. Механізми обробки помилок включають логування помилок, гнучке відновлення після помилок -

обробку помилок без збоїв та втрат даних, резервні механізми - забезпечення альтернативних ресурсів у випадку збоїв. Важливо щоб архітектура мінімізувала вплив збоїв у компонентах на продуктивність системи. Таку архітектуру називають відмовостійкою. Їй притаманне використання надлишковості, коли розгортають декілька екземплярів важливих компонентів, щоб у випадку помилок в одному, система працювала далі. Надлишковість досягається за допомогою кластеризації, реплікації, та механізмів відмовостійкості.

Масштабованість та надійність - фундаментальні принципи дизайну систем, які грають ключову роль у досягнення високої доступності. Система, яка адаптується до високого навантаження, імплементує механізми обробки помилок та має відмовостійку архітектуру здатна забезпечити високу доступність та продуктивність.

1.4.2 Ключові стратегії досягнення високої доступності

Після розгляду фундаментальних принципів дизайну систем можна перейти до ключових стратегій досягнення високої доступності [12]:

1) Надлишковість та реплікація.

Стратегія передбачає дублювання критично важливих компонентів чи систем. Якщо система падає, то надлишкова система перебирає її функції на себе, відповідно сервіс доступний користувачу без перебоїв.

2) Балансування навантаження.

Стратегія передбачає розподіл навантаження між кількома серверами, і забезпечує що жоден сервер не перевантажений. Детально вже було розглянуто в підрозділі балансувальник навантаження.

3) Кластер відновлення

Передбачає створення кластеру серверів, які працюють разом. Якщо один сервер виходить з ладу, інший перебирає його функції. Стратегія поширена в системах керування базами даних.

4) Розподілене сховище даних

Передбачає зберігання даних в декількох дата-центрах у різних локаціях,

щоб знизити ризик втрати даних. Розподілені сховища реплікують дані в різні географічні локації, таким чином підвищуючи їх доступність. Навіть, якщо в одному регіоні станеться природна катастрофа, і дані будуть втрачені, вони все ще будуть доступні з інших регіонів.

5) Моніторинг стану систем та оповіщення

Впровадження моніторингу систем дає можливість усунути проблеми до того, як вони вплинуть на доступність системи. Моніторинг дозволяє швидко та вчасно реагувати на проблеми. Оповіщення з'являються, коли стаються граничні події, наприклад використання CPU 90%, або використання пам'яті 85%. Тоді можуть бути здійснені певні дії, щоб сервіси не вийшли з ладу і залишились доступними для користувачів.

6) Регулярне обслуговування та оновлення системи

Оновлення системи найновішими патчами, покращення безпеки, виправлення багів знижує ризик збоїв та вразливостей, які можуть вплинути на доступність системи.

7) Географічний розподіл

Стратегія передбачає розгортання компонентів в різних географічних локаціях чи дата-центрах. Якщо компоненти в одній локації не доступні, то компоненти в інших все ще надають користувачу сервіс.

Надлишковість, реплікацію, партиціонування, кешування, балансування навантаження, механізми обробки помилок, моніторинг систем, географічне розподілення компонентів та даних - це ті стратегії, які потрібно комбінувати, щоб досягти високої доступності та забезпечити приємний користувацький досвід.

1.4.3 Вибір стратегій досягнення високої доступності

Кожна система унікальна, і вибір стратегій залежить від наступних чинників:

- вимог масштабованості. Обрана стратегія має впоратись зі зростанням навантаження.

- вимог продуктивності. Є стратегія активне-пасивне відновлення, в якій

відновлення компонента передбачає невелику затримку. В той же час є стратегія активне-активне відновлення, яке відбувається майже миттєво, і мінімально впливає на продуктивність.

- критичності системи. Наприклад, у сфері медицини вона надвисока. У банківській сфері недоступність системи може принести великі фінансові втрати. Відповідно у цих царинах система має бути максимально надійною та доступною, і забезпечити якнайшвидше відновлення після збоїв, навіть якщо це дорого. В той же час є менш критичні системи, в яких можна думати про співвідношення ціна-доступність.

- бюджету. Наприклад кластер відновлення може бути дорогим. Важливо оцінити чи виправдовують себе фінансові вкладення, адже може бути ситуація, коли вартість підвищення доступності системи вище за вартість даунтаймів.

У кожної медалі є дві сторони. Впровадження стратегій високої доступності збільшує складність та вартість системи, та іноді може негативно впливати на продуктивність системи. Наприклад, реплікація додатково навантажує мережу. Тому важливо тестувати систему та оптимізовувати її продуктивність.

Загалом, вибір стратегій високої доступності вимагає аналізу системних вимог, потенційних проблем та рішень. Вибравши правильні стратегії для системи можна гарантувати потрібну доступність за розумний бюджет.

1.5 Огляд AWS інструментарію

У цьому підрозділі розглянуто обчислювальні сервіси AWS та те, як різні хмарні компоненти інтегруються в архітектуру високодоступного веб-застосунку

1.5.1 Обчислювальні сервіси

В даному підрозділі спершу будуть оглянуті моделі хмарних обчислень. Після цього буде детально розглянуто та порівняно їх аналоги на платформі AWS.

1.5.1.1 Моделі хмарних обчислень

Перед тим як заглиблюватись у специфіку AWS сервісів, важливо розібрати три фундаментальні моделі хмарних обчислень [13]:

- інфраструктура як сервіс (IaaS), є найбільш гнучкою моделлю, яка пропонує віртуалізовані обчислювальні ресурси в інтернеті. Користувачі IaaS керують операційною системою, застосунками і даними, тоді як AWS відповідальний за фізичну інфраструктуру. Дана модель значно дешевша та масштабованіша порівняно з традиційними фізичними серверами.

- контейнери як сервіс (CaaS), забезпечує ефективне і масштабоване керування контейнерами. CaaS має вищий рівень абстракції, ніж IaaS, і дозволяє розгортати, масштабувати та керувати контейнерами без необхідності власноруч налаштовувати віртуальні машини чи кластерну інфраструктуру. Контейнери забезпечують легкий та ефективний метод пакування додатку з усіма залежностями та бібліотеками, і можуть потім бути запущені в будь-якому середовищі, яке підтримує контейнери. Модель спрощує розгортання, масштабування та управління додатками.

- функції як сервіс (FaaS), мають найвищий рівень абстракції, абстрагують все, окрім коду. Користувач завантажує код, все інше робить AWS: забезпечення, масштабування та обслуговування серверів. FaaS надає розробникам можливість запускати код у відповідь на події без необхідності управління інфраструктурою. Події, які тригерять функцію можуть бути різними: HTTP-запити, додавання в чергу, зміни в БД чи S3 сховищі. Модель пропонує найвищу зручність та ефективність для сервісів, керованих подіями.

Тепер можна перейти до глибшого розгляду обчислювальних сервісів AWS.

1.5.1.2 EC2 - IaaS в AWS

IaaS в AWS представлений сервісом EC2, який вважається кістяком обчислювальних AWS сервісів. EC2 надає користувачам широкий вибір віртуальних серверів, які підходять для різних видів навантажень. Екземпляри

EC2 можна конфігурувати, обравши потрібні параметри пам’яті, процесора та мережі [14].

Існує декілька типів екземплярів EC2, зокрема:

- загального призначення (серії t, m).

- оптимізовані для обчислень (серія c).

- з оптимізованою пам’яттю для роботи з великими обсягами даних (серія r).

- оптимізовані для зберігання для інтенсивного дискового вводу/виводу (серія i).

- прискореного обчислення з GPU для ML та графіки (серії p, inf).

Також можна обрати AMI (Amazon Machine Images) для EC2, які є шаблонами для екземпляра і містять операційну систему та додаткове програмне забезпечення[16].

Можна підключити EBS (Elastic Block Store) volumes - постійне сховище даних. Екземпляри зберігають тимчасові дані, які видаляються, коли екземпляр переходить в сплячий режим, зупиняється або видаляється.

EC2 дозволяє конфігурувати Security groups, які є віртуальним фаєрволом, в якому можна вказати протоколи, порти, діапазони IP для вхідних та вихідних з’єднань.

EC2 інтегрується з наступними сервісами, які можуть стати у нагоді [15]:

- EC2 Auto Scaling. автоматичне масштабування, що дозволяє динамічно змінювати кількість активних екземплярів відповідно до навантаження. Це допомагає підтримувати доступність та економити ресурси.

- Backup, створення резервних копій EC2 та EBS volume, прикріпленого до нього.

- CloudWatch для моніторингу

- ELB (Elastic Load Balancer) для розподілення навантаження між екземплярами.

- GuardDuty, визначає неавторизоване та шкідливе використання EC2

- EC2 Image Builder, автоматизує створення, керування, розгортання

імеджів серверів.

Екземпляри EC2 також можуть запускатись при використанні таких сервісів, як Elastic Container Service (ECS) та Elastic Kubernetes Service (EKS), які є представниками CaaS в AWS.

Є різні способи нарахування оплати за екземпляри EC2:

- безкоштовний період, оплата не нараховується.

- на вимогу, тобто оплата за час використання.

- резерв, довгострокова оренда на рік чи три. Дає змогу заощадити до 72% коштів.

- спот, використання вільних машин AWS, дає змогу значно заощадити кошти (до 90%), проте екземпляр може бути зупинений AWS у будь-який момент, коли зросте попит на EC2.

- план збереження, користувач бере зобов’язання постійно використовувати EC2 на певну суму доларів на годину на термін від року до трьох.

- виділені хости, використання фізичного сервера повністю виділеного тільки користувачеві, який може використати власні серверні ліцензії на програмне забезпечення та отримати допомогу у виконанні нормативних вимог.

1.5.1.3 ECS, EKS, Fargate - CaaS в AWS

[16] Amazon ECS - це повністю керований сервіс для запуску докер контейнерів . Підтримує два типи запуску контейнерів:

- EC2, коли користувач самостійно керує масштабуванням, вибором типу EC2, оновленнями операційної системи, безпекою екземплярів. AWS надає ECS-оптимізовані AMI (Amazon Machine Image), які включають Docker та ECS агент.

- Fargate, коли AWS керує інфраструктурою, забезпечуючи безсерверний підхід.

ECS має планувальник завдань, в якому можна визначити коли і як запускати контейнери, враховуючи навантаження та доступні ресурси. Таким чином ECS надає можливість швидкого масштабування за запитом.

[17] Amazon EKS - це керований сервіс Kubernetes, який дає змогу розгортати контейнери з використанням стандартного Kubernetes API. Є ідеальним рішенням для команд, які вже мають досвід із Kubernetes. AWS відповідає за управління та оновлення компонентів Kubernetes. EKS сумісний зі стандартним Kubernetes, тож застосунок, розгорнутий у локальному Kubernetes кластері можна легко перенести на AWS. EKS як і ECS надає гнучкість у виборі технологій для запуску контейнерів: можна запускати робочі ноди як на EC2, так і безсерверно за допомогою Fargate. EKS підтримує автоматичне масштабування як подів (HPA), так і нод (Cluster Autoscaler).

Варто відмітити, що найкраще використовувати EKS для застосунків з мікросервісною архітектурою та для команд, які вже мають налаштовані Kubernetes CI/CD пайплайни.

AWS Fargate - це безсерверна платформа для запуску контейнерів. Вона може використовуватись як бекенд для EKS та ECS. Використання Fargate дає користувачеві змогу зосередитись на логіці застосунку, не переймаючись керуванням серверів, виділенням ресурсів, масштабуванням. Таким чином процес перетворення ідеї додатку в готове хмарне рішення стає значно швидшим. Користувач сплачує тільки за ресурси, які використовуються під час виконання контейнера. Fargate підходить для архітектур, керованих подіями та завдань, які швидко виконуються.

Fargate доречно використати в таких сценаріях [18]:

- завдання за розкладом.
- завдання - реакції на події, як обробка HTTP-запитів.
- попередня обробка даних.
- автоматичне масштабування в пікові навантаження.

Після розгляду ECS, EKS та можливості безсерверного запуску контейнерів Fargate доречно їх порівняти. Як вже зазначено ECS та EKS можуть запускати контейнери на EC2, або за допомогою Fargate. Тож порівняймо ці підходи:

Таблиця 1.1. Порівняння ECS та EKS з типом запуску контейнерів EC2 та Fargate.

Сервіс	Керування серверами	Гнучкість	Складність використання	Ціна
ECS + EC2	Так	Висока	Середня	Низька
EKS + EC2	Так	Дуже висока	Висока	Середня
ECS/EKS + Fargate	Ні	Низька/Трохи вища	Низька/Трохи вища	Вища ніж з EC2

Загалом, Fargate найпростіший у налаштуванні, адже потребує мінімум конфігурації. В той же час при використанні його для довготривалих сервісів, вартість буде найвища. Економічно Fargate доцільно обирати для невеликих або нестабільних навантажень. ECS з EC2 найдешевший варіант, адже AWS не бере додаткову плату за ECS як сервіс, тому користувач платить тільки за екземпляри EC2. ECS та EKS з EC2 потребують управління кластером екземплярів EC2, масштабування та оновлення. Крім того EKS повністю сумісний з Kubernetes, і його конфігурація ще більш гнучка, ніж у ECS, і відповідно важча. Щодо вартості, EKS стягує 10 центів за годину за кластер, незалежно від кількості нод, що в сумі дає 72 долари на місяць. Крім того, як і у випадку з ECS потрібно платити за використання екземплярів EC2.

З цих опцій для дипломного проекту найкраще підходить ECS+EC2/ECS+Fargate. Немає сенсу розглядати Kubernetes, адже EKS це досить дорого, потребує серйозних навичок та не має сенсу для монолітного застосування. Крім того, ECS створений повністю всередині екосистеми AWS, тому значно легше інтегрується з AWS сервісами. Fargate зручніший і простіший у використанні, проте практично не конфігурується. З урахуванням безкоштовного періоду на AWS та невеликого навантаження на початкових етапах, опція EC2 + Fargate найкраще CaaS рішення, яке дає змогу швидше розгорнути інфраструктуру. Якщо дивитись у розрізі підтримки інфраструктури роками та великої кількості клієнтів, то значно вигідніше обрати ECS + EC2.

1.5.1.4 Lambda - FaaS в AWS

AWS Lambda - це безсерверне рішення для хмарних обчислень, яке є представником FaaS в AWS. Кожна лямбда функція запускається в окремому контейнері. Коли функція створюється, лямбда запаковує її в новий контейнер [19]. Перед запуском кожному контейнеру виділяється необхідна кількість пам'яті та потужності процесора. Після закінчення функції, використана пам'ять множиться на час виконання, і стягується відповідна плата.

Лямбду доцільно обирати у таких сценаріях [20]:

- завдання запускається на короткий період часу.

- кожне завдання є самодостатнім.

- є велика різниця між найнижчим та піковим навантаженням.

Найпопулярнішими сценаріями використання AWS Lambda є:

- масштабовані API. Тоді одне виконання лямбда функції відповідає одному HTTP запиту. Різні запити направляються до різних лямбда функцій за допомогою сервісу API Gateway. Лямбда автоматично масштабує окремі функції залежно від навантаження. Таким чином різні частини API масштабуються незалежно, що дозволяє зробити API гнучким та економічно вигідним.

- обробка даних: лямбда легко інтегрується зі сховищами даних, такими як DynamoDB, і дозволяє робити певні операції у відповідь на додавання нового екземпляру чи оновлення наявного. Таким чином лямбда може бути інструментом для аналітики, підрахунку, сповіщень.

Лямбда дозволяє автоматизувати завдання, які не потребують постійно активного сервера. Наприклад, крон джоба, яка виконується з певним інтервалом часу, чи обробка даних з форм сайту.

Для всіх мов, що лямбда підтримує (Java, Node JS, Python, Ruby, Go, C#) створені SDK для полегшення написання лямбда функцій, та їх інтеграції з іншими сервісами AWS.

Лямбда має кілька переваг:

- оплата тільки за час обчислень.

- відсутність необхідності керування інфраструктурою, що дає змогу

зекономити на оновленні операційної системи та керуванні мережею, та гарантує автоматичне масштабування.

Проте варто зазначити і недоліки лямбди:

- холодний старт: між подією, яка викликає лямбду, і її виконанням може бути певна затримка. Якщо функція не використовувалась 15 хвилин, затримка може досягати 10 секунд. Тоді треба, наприклад, підігрівати функцію, викликаючи частіше, або резервувати мінімум одну машину на постійній основі, що буде дорожче.

- обмеження функцій. Лямбда не може виконуватись довше 15 хвилин, не може використовувати більше 3008 мб оперативної пам’яті. Обсяг зіп архівованого пакету не має перевищувати 50мб, а розархівованого - 250 мб. По дефолту з одного акаунту не можна одночасно виконувати більше 1000 лямбда функцій, хоча це можна змінити через підтримку користувачів AWS. Якщо використовувати API Gateway для тригеру лямбди у відповідь на HTTP запит, корисне навантаження не може перевищувати 10 мб.

1.5.1.5 ECS vs EC2 vs Lambda

В попередніх підрозділах було розглянуто різні моделі хмарних обчислень на AWS, та сервіси, які відповідають різним моделям. IaaS представлений EC2; CaaS - ECS, EKS, Fargate; FaaS - Lambda. В цьому підрозділі ми підіб’ємо підсумки та зробимо порівняння сервісів. CaaS буде представлений ECS, адже визначено, що це найкраще рішення для дипломного застосунку, пов’язане з контейнерами.

EC2 надає користувачеві максимальну гнучкість у виборі конфігурації віртуальної машини, операційної системи, безпеки, масштабування, мережевої інфраструктури. Конфігурація у свою чергу вимагає багато часу та дій від користувача. Рішення добре підходить для монолітних систем та у випадках, коли необхідний повний контроль над середовищем виконання.

ECS часто використовується для мікросервісної архітектури. При запуску ECS завдання можуть виконуватись на EC2 або безсерверно за допомогою Fargate.

У випадку запуску контейнерів на EC2, контейнери керуються AWS, але користувач має налаштувати EC2. У випадку використання Fargate зникає потреба в управлінні екземплярами EC2. Загалом, ECS пропонує баланс між гнучкістю та зручністю [22].

Lambda найкраще підходить для коротких сценаріїв, орієнтованих на події. AWS повністю відповідає за інфраструктуру, а розробники повністю зосереджуються на написанні коду. Лямбда найзручніша у використанні і найекономніша в сценаріях, коли немає необхідності тримати постійно активний сервер.

Таблиця 1.2. Порівняння EC2, ECS з EC2 та Fargate, Lambda

Характеристика	EC2	ECS + EC2/Fargate)	Lambda
Модель обчислень	IaaS	CaaS	FaaS
Гнучкість	Найбільша	Досить висока/ Досить низька	Найменша
Масштабування	Налаштовується користувачем	Дворівневе 2 рівні налаштовуються користувачем / тільки рівень контейнерів налаштовується користувачем	Автоматичне без участі користувача

Оплата	За час роботи	За EC2/ виділені процесор і пам’ять	За кількість викликів, використання процесора і пам’яті
Сценарії використання	Моноліти, великі сервіси	Мікросервіси/ орієнтовані на події сервіси	Орієнтовані на події сервіси, рідко виконувані, короткі завдання
Інтеграція з AWS сервісами	Налаштовується вручну, потребує конфігурації	Краще за EC2 на рівні контейнерів	Ідеальна інтеграція
Тривалість виконання	Необмежена	Необмежена/ залежить від завдання, може бути короткою	До 15 хвилин на одне виконання функції
Холодний старт	Відсутній	Відсутній	Є, після паузи 15 хвилин
Ціна	Дешево, якщо завантаження постійне. Дорого, якщо сервер простоює	Аналогічно EC2 / дорожче, плата за зручність	Дешево при низьких нечастих навантаженнях, дорого при високих і постійних

Важливо розуміти різницю між використанням EC2 та ECS на базі EC2. В першому випадку, якщо потрібно розгорнути контейнер, то потрібно поставити

докер та запустити контейнери вручну або скриптами. EC2 надає контроль над операційною системою, тоді як ECS бере ці функції на себе, а також забезпечує двигун для контейнерів. ECS допомагає керувати контейнерами: автоматично розподіляє їх по EC2, стежить за станом, може перезапустити, масштабувати. EC2 - це сервер, ECS - диспетчер контейнерів. ECS надає напівавтоматичне налаштування балансування навантаження та реєструє контейнери в цільову групу балансувальника. Тоді як при використанні EC2 треба вручну створювати балансувальник та цільову групу і додавати екземпляри до цільової групи балансувальника. Також для EC2 треба писати скрипти оновлення, щоб слідкувати за зупинкою старої версії та запуском нової, тоді як ECS керує цим процесом автоматично: запускає нові контейнери, реєструє в цільовій групі, чекає перевірку працездатності (health check), і лише потім зупиняє старі контейнери. Масштабування у EC2 потребує налаштування Auto Scaling Group, тоді як в ECS є два рівні масштабування: один на рівні EC2 і один на рівні контейнерів, який визначає їх запущену кількість. ECS краще інтегрується з багатьма сервісами AWS на рівні контейнерів: автоматична інтеграція з CloudWatch, більш автоматизоване налаштування балансування при масштабуванні, нативна підтримка підключення секретів до контейнерів.

Загалом, під час аналізу різниці між сервісами стає зрозуміло, що Lambda не підходить для основної частини API, адже це буде дуже дорого. Для Rest API варто обрати чистий EC2 або ECS залежно від вимог, потрібного рівня контролю та гнучкості та того чи використовують контейнери як основну одиницю деплою застосунку. Для одного застосунку можна використати декілька різних обчислювальних сервісів: так в дипломній роботі Lambda може використовуватись для очищення постів з речами, які вже були знайдені, або речами, які не знайшли протягом 3 років. Таким чином зменшиться навантаження на основний додаток, розгорнутий за допомогою EC2 чи ECS.

ECS з Fargate порівняно з ECS на основі EC2 дещо дорожчий, надає менше контролю, проте зручніший, адже вимагає менше налаштувань та надає змогу швидше розгорнути додаток. Дорожчий поняття відносне, адже при використанні

Fargate відпадає необхідність в адмініструванні EC2, що зменшує час, потрібний для підтримки інфраструктури, відповідно знижує і кошторис.

1.5.2 Компоненти багатошарової архітектури в AWS

В одному з попередніх підрозділів було розібрано важливі компоненти багатошарової архітектури. В цьому будуть розглянуті їх відповідники в AWS.

1.5.2.1 CloudFront - CDN в AWS

CloudFront є реалізацією CDN в AWS і забезпечує низьку затримку доставки контенту, автоматичне масштабування, інтеграцію з іншими сервісами AWS. CloudFront використовується для [23]:

- доставки даних через більш ніж 450 граничних серверів по всьому світу за допомогою автоматизованого зіставлення мережі та інтелектуальної маршрутизації.

- покращення безпеки за допомогою шифрування трафіку та контролю доступу і використання AWS Shield Standard для запобігання DDoS атакам. CloudFront підтримує HTTPS, SSL сертифікати, інтегрується з Amazon WAF (Web Application Firewall).

- запуску невеликих обчислень на граничних серверах за допомогою Lambda@Edge

- гнучкого керування TTL (Time to Live), тобто часу, протягом якого інформація є актуальною; кешування за шляхами; кукі; хедерами. Також сервіс надає можливість інвалідувати кеш, після чого він буде оновлений.

Cloudfront може працювати зі статичним та динамічним контентом і його можна інтегрувати з S3, EC2, Load Balancer, API Gateway.

1.5.2.2 Route 53 - DNS в AWS

[24] Route 53 це масштабований та високодоступний DNS сервіс, який дозволяє здійснювати реєстрацію домену, DNS роутинг, та перевірку доступності

ресурсів .

Route 53 може маршрутизувати трафік наступним чином:

- simple routing, звичайне перенаправлення на ресурс

- weighted routing, балансуванням між кількома ресурсами

- latency-based routing, направляє до регіону з найменшою затримкою.

- failover routing, автоматичне перенаправлення трафіку, якщо ресурс недоступний.

- geolocation routing, направлення запиту у найближчий регіон до користувача.

В мікросистемі AWS Route 53 дозволяє розподіляти трафік між кількома регіонами; підключати домен до S3, EC2, API Gateway, балансувальника; підвищувати доступність за допомогою failover routing; показувати в різних регіонах локалізовані версії сайтів за допомогою geolocation routing. Route 53 часто використовують з CloudFront. Route 53 забезпечує керування доменним ім’ям та направляє трафік на граничні сервера CloudFront з найменшою затримкою для користувача.

1.5.2.3 ELB - Load Balancers в AWS

В AWS функції балансування навантаження виконує Elastic Load Balancer (ELB). ELB перевіряє стан цільових ресурсів і розподіляє вхідний трафік тільки по доступним [25]. ELB масштабується автоматично залежно від кількості трафіку. В AWS є поняття цільової групи. Коли користувач надсилає запит на балансувальник, він перенаправляється до одного з ресурсів у цільовій групі на основі визначених правил маршрутизації. Є два важливі типи ELB, які забезпечують високу доступність у хмарній архітектурі: Application Load Balancer (ALB) та Network Load Balancer (NLB).

ALB має такі особливості [26]:

- працює на 7-ому рівні моделі OSI (Open Systems Interconnection), яка описує 7 шарів комунікації комп’ютерних пристроїв через мережу. Даному рівню відповідають протоколи HTTP та HTTPS.

- підтримує маршрутизацію за шляхом, хостом, заголовками, методами.

- інтегрується з EC2, ECS, EKS, Lambda, Cognito для автентифікації, WAF.

NLB у свою чергу такі [27]:

- працює на 4-ому рівні моделі OSI, тобто застосовний при використанні протоколів TCP та UDP.

- дуже висока продуктивність, вище ніж у ALB. Здатний масштабуватись до виконання мільйонів запитів на секунду.

- підтримує IP-based routing, тобто перенаправлення запиту на конкретну IP адресу.

Також, варто відмітити третій тип ELB, який називається Gateway Load Balancer. [28] Він працює на 3-ому рівні моделі OSI і використовується для інспекції вхідного трафіку перед доступом до внутрішніх сервісів, розгортання комерційних фаєрволів, фільтрування між зоною доступу і внутрішньою мережею. Для дипломного застосунку не буде розгортатись комерційний фаєрвол, тому даний тип ELB точно не буде використаний.

Загалом, ALB підходить для REST API, мікросервісів, вебсокетів, тоді як NLB для маршрутизації TCP-з'єднань з низькою затримкою. Також, ALB може направляти навантаження на лямбду, чого не може NLB. ALB трохи дорожчий, але це не критично, адже він чудово підходить для бекенду дипломного застосунку.

1.5.2.4 WAF - фаєрвол в AWS

WAF (Web Application Firewall) дозволяє перевіряти HTTP та HTTPS запити, які надходять до застосунку. Дає змогу захистити Cloudfront, API Gateway, ALB, пул користувачів Amazon Cognito, Amplify, App Runner, Verified Access instance. WAF дає змогу контролювати доступ до контенту залежно від IP адреси відправника запиту, стрічки запиту.

AWS WAF надає таку базову поведінку [29]:

- дозволити всі запити, крім вказаних.

- блокувати всі запити, крім вказаних.

- підраховувати запити, які відповідають певному критерію, корисно для

моніторингу та перевірки нових правил обробки запитів.

- запускати капчу чи перевірки для запитів, які відповідають певному критерію, дає змогу знизити трафік ботів.

WAF надає такі переваги:

- додатковий захист від веб-атак за такими критеріями як: IP адреса відправника запиту, країна відправника, значення хедерів, довжина запиту, наявність SQL коду (SQL-ін’єкції) , наявність скрипта (XSS-атаки).

- наявність базових правил поведінки.

- перевикористання правил для різних застосунків.

- наявність готових правил для захисту від найпоширеніших атак.

- можливість обмежити кількість запитів від клієнта за проміжок часу, ефективно проти ботів і DDoS-атак.

Можна поставити WAF перед ALB і фільтрувати запити до того, як вони потраплять на цільовий ресурс, EC2 або безсерверне рішення.

Оплата здійснюється за кількість створених правил та кількість оброблених запитів.

Загалом, WAF є потужним і гнучким інструментом безпеки, який надає захист від DDoS-атак, XSS-атак, SQL-ін’єкцій.

1.5.2.5 API Gateway в AWS

Amazon API Gateway - AWS сервіс для створення, публікації, моніторингу, захисту та підтримки API. API Gateway є високомасштабованим сервісом, і може обробляти сотні тисяч викликів API одночасно, відповідаючи за керування трафіком, авторизацію, контроль доступу, моніторинг, версіонування API [30]. API Gateway є вхідною точкою до бекенд сервісів побудованих на EC2 або на безсерверних рішеннях.

API Gateway може забезпечити наступні специфічні для AWS функції:

- автентифікація з лямбда авторизаторами, пулами користувачів Cognito.

- можливість використати CloudFormation шаблони для створення API.

- інтеграція з WAF, CloudFront, CloudWatch, Lambda.

- можливість створювати приватні API, доступні лише у VPC.
- має вбудований AWS Shield для захисту від DDoS-атак

За допомогою API Gateway можна будувати безсерверні API на лямбді, або робити єдину точку входу для різних сервісів. Також API Gateway може бути використаний як проксі до сторонніх API. В дипломному додатку API Gateway може направляти запити на EC2 та тригерити лямбди. Крім цього API Gateway буде звертатись до Cognito або лямбда авторизаторів для авторизації та автентифікації користувачів перед направленням запиту на бекенд, таким чином знизивши на нього навантаження.

1.5.2.6 Elasticache - Кеш БД в AWS

Elasticache - сервіс, який спрощує налаштування, керування і масштабування розподіленого сховища даних у пам’яті або кеш-середовища у хмарі [31]. Сервіс можна використовувати у двох форматах: створити власний кластер кешу, або використовувати безсерверний кеш. Створення власного кластеру надає гнучкість у виборі типу нод, кількості, розташування в потрібних зонах доступності.

Можна працювати з такими рушіями:

- Redis, потужний, має розширену функціональність у вигляді черг, підписок, публікацій, TTL, скриптів, дозволяє зберігати знімки даних.
- Memcached, простіший, не має постійного сховища.
- Valkey, форк Redis з відкритим джерелом коду. Створений, бо Redis тепер має обмежене використання у хмарних провайдерах. Можливо, буде використовуватись замість Redis. Valkey поки не є офіційно доступним в Elasticache.

Кеш база даних пришвидшує доступ до даних та розвантажує основні бази даних застосунку, кешуючи часті запити. Порівняно з RDS доступ стає швидшим у десятки разів. Так як Elasticache - керований сервіс, AWS бере на себе функції оновлення, моніторингу та автоматичного відновлення. Сервіс інтегрується з EC2, Lambda, API Gateway, RDS. Є можливість горизонтального масштабування за

допомогою шардингу. Може бути вигідним у сценаріях, коли викликається лямбда для отримання даних, тоді кеш зменшує кількість викликів лямбда функції.

1.5.2.7 Повідомлення та події в AWS: SQS, SNS, EventBridge, Step Functions

Сучасні архітектури вимагають гнучкої та масштабованої взаємодії між компонентами. В AWS є набір керованих сервісів, які допомагають організувати обробку подій, повідомлень, ланцюжків завдань. В даному підрозділі буде здійснено короткий огляд Amazon SQS, SNS, EventBridge та Step Functions.

Amazon SQS [32] - черга повідомлень для асинхронного обміну даними між різними сервісами чи частинами системи. В чергу додаються повідомлення, після чого вони очікують, доки споживач забере їх з черги.

SQS підтримує два основні типи черг:

- Standard Queue, яка здатна забезпечити високу пропускну здатність, але не гарантує порядок повідомлень та відсутність дублікатів.

- FIFO Queue, гарантує порядок повідомлень та відсутність дублікатів. Доцільно використати для серйозних фінансових застосунків чи в галузі охорони здоров’я.

SQS підходить для задач з великою кількістю подій, які потребують зберігання на певний проміжок часу для асинхронної обробки та фонових задач.

Amazon SNS, або Simple Notification Service, дозволяє розсилати повідомлення багатьом отримувачам за допомогою моделі публікації та підписки. Коли повідомлення публікується, воно автоматично пересилається всім підписникам.

Отримувачами повідомлень від SNS можуть бути такі сервіси [33]:

- HTTP/HTTPS ендпоінти, наприклад сервіс може надіслати POST запит з корисним навантаженням.

- Email, популярним сценарієм є надсилання листа з певним вмістом.

- SMS, повідомлення на телефон.
- Лямбда функції.
- SQS-черга для асинхронної обробки.

При публікації в SNS, всі підписники одразу одночасно отримують повідомлення, таку систему комунікації називають push-based. В черзі повідомлення очікують на обробку, а систему комунікації називають pull-based. SNS не зберігає повідомлення, якщо підписник не доступний (винятком є SQS: якщо SQS є ціллю, SNS зберігає повідомлення). SNS підходить для розсилання сповіщень та транслювання події багатьом підписникам.

[34] Amazon EventBridge - асинхронний сервіс для маршрутизації подій, який дозволяє приймати події з різних сервісів, платформ, додатків та направляти їх відповідно до заданих правил. Event Bridge дозволяє гнучко фільтрувати події за структурою, значенням, типом, джерелом, на відміну від SNS, який має лише фільтрацію по темі. Таким чином можна налаштувати умовну маршрутизацію, коли одна подія може спричиняти різну поведінку залежно від її змісту. Event Bridge може направляти події в такі сервіси як Lambda, SQS, Step Functions, ECS, EC2. Події в сервісі можуть зберігатись до 24 годин. На одну подію можуть реагувати кілька сервісів. Event Bridge як і SNS використовує модель публікації-підписки. Сервіс підтримує структуровані схеми подій, у той час як події в SNS передаються у будь-якому форматі. Event Bridge інтегрується з деякими SaaS платформами, на приклад Shopify, Zendesk. Загалом, якщо SNS це простий механізм надсилання повідомлень підписникам, то EventBridge може маршрутизувати, гнучко фільтрувати та трансформувати події, керувати ними, а також має глибшу інтеграцію з іншими AWS сервісами.

[35] Step Functions - сервіс побудови сценаріїв виконання задач. Він дозволяє описати логіку, яка визначає послідовність викликів сервісів AWS або мікросервісів. Кроки виконання можуть включати умови, обробки помилок, затримки, паралельні виклики. Прикладом використання Step Functions може бути

побудова багатокрокового бізнес-процесу: перевірка замовлення, списання коштів, оновлення бази даних, відправлення повідомлення про завершення обробки замовлення користувачу. Step Functions дає повне уявлення про послідовність та логіку виконання подій.

Кожен з сервісів вирішує специфічні задачі, а разом вони утворюють потужний інструментарій для побудови асинхронних, масштабованих, орієнтованих на події систем.

1.5.2.8 CloudWatch і CloudTrail - моніторинг в AWS

У багатошаровій архітектурі хмарних застосунків важливо забезпечити спостережуваність та аудит дій системи та користувачів. Ключовими інструментами для забезпечення моніторингу на платформі AWS є CloudWatch та CloudTrail.

[36] Amazon CloudWatch - повністю керований сервіс для збору та аналізу метрик, логів, подій у реальному часі. Основними функціями CloudWatch є:

- моніторинг ресурсів, на приклад метрики CPU, пам’яті, дискового-вводу виводу EC2, RDS, ECS.

- збір та аналіз, пошук, візуалізація, логів.

- налаштування алертів при граничних подіях, наприклад CPU EC2 працює на 90%.

- візуалізація метрик за допомогою дашбордів.

- оброблення подій, інтеграція з EventBridge для реакцій на стан ресурсів. Таким чином можна забезпечити AutoScaling.

CloudWatch є важливим інструментом, адже дозволяє аналізувати тренди, виявляти слабкі місця системи та автоматично реагувати на небезпечні для системи події, такі як перевантаження CPU.

[37] AWS CloudTrail - це сервіс для логування та аудиту, який записує усі API-виклики та операції над ресурсами в AWS-акаунті. CloudTrail забезпечує такі функції:

- логування дій користувачів та сервісів, записує запити від AWS SDK, CLI, Management Console, внутрішніх сервісів AWS.

- трасування, запис подій з інформацією про джерело, час, IP, id користувача, роль, запити і відповіді від API AWS.

- зберігання логів в S3, пошук та аналіз при інтеграції з CloudWatch Logs.

- аудит безпеки та відповідність стандартам, можливість відстежувати аномалії, оглядати політики доступу та перевіряти відповідність стандартам, таким як GDPR.

- мультирегіональна підтримка, можливість повноцінного аудиту ресурсів розгорнутих в будь-якій географічній локації.

CloudTrail є важливим інструментом для збору історії дій та їх подальшого аналізу, аудиту безпеки, забезпечення відповідності стандартам.

CloudWatch фокусується на моніторингу показників продуктивності, а CloudTrail на аудиті безпеки і трасуванні подій. Разом вони дозволяють вчасно виявляти технічні та безпекові інциденти, автоматизувати реагування на них та проводити аналіз при розслідуванні подій.

1.5.3 Швидке розгортання в AWS

Іноді виникає необхідність дуже швидко розгорнути інфраструктуру. AWS пропонує деякі інструменти для цього, в цьому підрозділі коротко розглянемо наступні: AWS Amplify та AWS SAM.

[38] Amplify забезпечує робочий процес на основі гіта для хостингу та безперервного розгортання безсерверних веб-застосунків. Amplify розгортає застосунок за допомогою глобальної CDN AWS. Сервіс підтримує багато відомих SPA та SSR фреймворків, серед них Next, Nuxt, React, Angular, Vue. Amplify автоматизує CI/CD та надає готові модулі для автентифікації, зберігання файлів, push-повідомлень, API. Розгортання відбувається за допомогою декількох команд в AWS CLI, або через гіт інтеграцію: при кожному пуші в репозиторій оновлена версія застосунку потрапляє в продакшн. Amplify є високорівневим, треба вказати тільки хостинг, API, конфігурацію автентифікації, все інше забезпечує Amplify.

[39] SAM (Serverless Application Model) - фреймворк з відкритим кодом для побудови безсерверних застосунків за допомогою визначення інфраструктури як коду. За допомогою SAM можна в одному шаблоні описати лямбда функції, API Gateway, DynamoDB таблиці, IAM політики, та інші ресурси. Навіть складні безсерверні архітектури можна розгорнути за декілька хвилин, написавши мінімум конфігураційних файлів.

Загалом, Amplify використовується для фронтенду та бекенду мобільних та SPA додатків, в SAM для безсерверного бекенду. Завдяки Amplify та SAM розробники можуть зосередитись на написанні коду, замість ручного налаштування CI/CD та інфраструктури, і це в свою чергу пришвидшує розгортання рішення.

2. Проектування власного рішення

В цьому розділі буде визначено технічне завдання, після чого буде спроектовано рішення для реалізації визначеного функціоналу. Також буде описано компоненти системи та їх взаємодію, обґрунтовано вибір технологій для різних задач на основі теоретичної бази отриманої під час аналізу предметної області.

2.1 Технічне завдання

В системі початково буде два типи користувачів: адміністратори та звичайні користувачі. Пізніше планується додавання нової ролі для персоналу технічної підтримки.

Цифрове бюро знахідок має надавати наступний функціонал для всіх користувачів:

- можливість реєстрації.
- створення оголошення про знахідку з контрольним питанням, та контактами для зв’язку.
- створення оголошення про втрату.
- функція вибору локації на мапі при створенні оголошення.
- можливість прикріпити кілька фото до втрати і знахідки.
- пошук по всім знахідкам з фільтрами по категорії, місцю, даті, назві, опису та пагінацією.
- перегляд актуальних оголошень (створених в останній тиждень).
- написання коментарів під постами.
- можливість отримати номер автора посту, заповнивши свої контактні дані.
- видалення свого поста.
- позначення поста відміткою “ЗНАЙДЕНО”.

Для адміністраторів мають бути доступні такі функції:

- видалення будь-якого поста.
- видалення будь-якого коментаря.

- блокування користувачів.

- винесення попереджень користувачам.

- створення запиту на редагування поста.

Для користувачів також можуть бути додані наступні функції:

- надсилання скарги на учасників, пост, коментар.

- транзакції вдячності людям, які знайшли втрачену річ.

- отримання повідомлення про схожі знахідки

Для зареєстрованих користувачів буде додана можливість здійснювати пошук знахідок по фото до десяти разів на місяць.

Для того, щоб не зберігати застарілі дані буде додана автоматична очистка знайдених речей та знахідок/втрат, які не знайшли протягом трьох років. Очистку планується проводити раз на тиждень, але частоту можна конфігурувати відповідно до потреб.

Серед нефункціональних вимог можна відмітити наступні:

- відмовостійкість та надійність, мінімізація кількості помилок, забезпечення механізмів відновлення при збоях.

- швидкодія, швидкі відповіді на запити користувачів.

- масштабованість, адаптація до зростання навантаження.

- безпека, забезпечення конфіденційності персональних даних користувачів.

- сумісність, підтримка різних пристроїв, браузерів, операційних систем.

- адаптивний та інтуїтивний дизайн.

2.2 Вибір сервісів

Важливо підібрати сервіси так, щоб вони могли виконувати поставлені перед ними задачі. Це й буде зроблено в даному підрозділі.

Як вже було визначено в теоретичному розділі, для API не підходить FaaS, тобто Lambda. Серед контейнерних рішень ECS на базі Fargate було визначено найкращим рішенням. Такий сервіс дешевший за EKS та простіше конфігурується. В порівнянні з ECS на базі EC2 - менше конфігурації та швидше

розгортання, хоча й дорожча вартість і менша гнучкість. Через свою зручність Fargate був визначений пріоритетним на перших етапах запуску додатку. Тому фінальні опції вибору обчислювальної платформи - EC2 та ECS на базі Fargate. EC2 надає більшу гнучкість у налаштуванні, вигідніший економічно за постійного навантаження та чудово підходить для монолітів, в той час як ECS гарне рішення для мікросервісної архітектури. Крім того, якщо контейнери не використовуються як основна одиниця деплою, то не має сенсу і в ECS. У випадку з дипломним додатком використати EC2 простіше та економніше.

Для повідомлення користувачів про те, що схожу річ було знайдено доцільно використати SNS. При створенні оголошення про втрату EventBridge буде передавати подію Lambda, яка додаватиме користувача до SNS топіку для розсилки. Потім при створенні нової знахідки в EC2 EventBridge буде передавати подію в іншу Lambda, яка в свою чергу додаватиме в потрібний топік SNS корисне навантаження. Для кожного міста та знахідки буде свій топік, наприклад kyiv-documents буде нотифікувати користувачів, які загубили документи в Києві. Підписниками SNS будуть емейли та телефони користувачів, таким чином буде забезпечено автоматичну відправку сповіщень без використання додаткових мікросервісів. Такий орієнтований на події підхід дозволяє уникнути постійного опитування бази даних для фіксування змін.

Для очистки постів, яким більше трьох років, та знайдених речей, яка відбувається раз на тиждень доцільно використати Event Bridge і Lambda, адже це навантаження, яке відбувається рідко. Event Bridge дозволяє тригерити події за графіком, а використання Lambda зменшує навантаження на основний сервер.

Пошук по фото доступний лише для зареєстрованих користувачів 10 разів на місяць. Це непостійне і невелике навантаження, для якого доцільно використати Lambda. Для реалізації функціональності пошуку за зображенням буде використана модель CLIP, яка генеруватиме описові ознаки фотографій, а подальший пошук потрібно виконати по базі даних на основі одного з алгоритмів подібності. Для зберігання описових ознак доцільно використати окрему від основної базу даних. Варто розглянути три варіанти: Aurora Serverless, OpenSearch

з kNN плагіном, Pinecone. Aurora Serverless є реляційною базою даних, вона не оптимізована для векторного пошуку, а реалізація подібності вимагає написання логіки порівняння векторів. Розробка вимагає часу, а її складність досить висока. OpenSearch з k-NN плагіном дозволяє зберігати вектори та виконувати запити на пошук схожості, проте потребує налаштування кластерів EC2, індексів, масштабування. OpenSearch не є серверлес, тому конфігурація вимагає навичок та часу. Pinecone - векторна база даних як сервіс, не входить до мікросистеми AWS, проте легко інтегрується з нею. Pinecone є безсерверним рішенням, тому не потребує налаштувань інфраструктури. Сервіс дуже просто налаштовується, автоматично масштабується, створений спеціально для роботи з векторами та має безкоштовний пробний період. З урахуванням особливостей трьох розглянутих сервісів найкращим рішенням буде Pinecone, який чудово підходить для векторного пошуку, потребує мінімум налаштувань та дозволить швидко реалізувати функціонал.

Для авторизації та аутентифікації користувачів буде використаний AWS Cognito. [40] Сервіс надає готову надійну та безпечну реалізацію реєстрації і дозволяє керувати користувачами.

Фронтенд застосунку зберігатиметься в сервісі S3 як статичний вебсайт. [41] S3 - сховище об’єктів, яке зберігає дані щонайменше в трьох зонах доступності в межах регіону. Це спростить хостинг інтерфейсу, забезпечить високу доступність та масштабованість. Використання S3 для інтерфейсу економить ресурси розробників, адже відпадає необхідність керувати кластером контейнерів ECS або конфігурувати EC2. Також в S3 будуть зберігатись фотографії знахідок.

Для рівня даних буде використана реляційна база даних. Для реалізації реляційної бази даних було розглянуто три варіанти: Amazon RDS, Amazon Aurora та Aurora Serverless 2. Всі три сервіси - керовані рішення від AWS, які знімають з розробника необхідність ручного налаштування серверів баз даних. [42] RDS підтримує популярні рушії, зокрема MySQL, PostgreSQL, MariaDB. RDS надає змогу обрати тип інстансу та зону доступності. Масштабування необхідно

налаштовувати вручну. Порівняно з Aurora, RDS легший у налаштуванні та дешевший, тому RDS хороший вибір для невеликих проектів з передбачуваним навантаженням. [43] Aurora в свою чергу високопродуктивна база, сумісна лише з PostgreSQL та MySQL. Aurora створена спеціально для AWS і забезпечує продуктивність у декілька разів вищу за RDS, при цьому зберігаючи дані у трьох зонах доступності. Aurora теж масштабується вручну, проте пропонує вищу доступність та відмовостійкість, саме тому є дорожчою. [44] Aurora Serverless v2 - безсерверна версія Aurora. Основна її перевага це автоматичне масштабування залежно від навантаження. Коли запити не надходять Serverless база даних засинає, і може бути невеликий холодний старт при запитах після перерви. Aurora Serverless v2 підходить для нерівномірного трафіку або при розробці MVP (Minimum Value Product) для швидкого виходу на ринок, проте при постійному навантаженні її вартість буде високою. Якщо припустити стабільне навантаження на сервер та взяти до уваги простоту налаштування і економічну доцільність рішення, краще обрати Amazon RDS. Для забезпечення доступності варто створити репліку RDS принаймні в одній зоні доступності.

Для зменшення навантаження на базу даних та пришвидшення доступу до частозапитуваних даних буде використано кеш базу даних Elasticache на основі Redis. Redis потужніший за Memcached, дозволяє відновити дані після падіння, підтримує TTL та складні типи даних, тоді як Memcached - просте key-value сховище без збереження стану. Redis гнучкіший та має більше сценаріїв використання.

Тепер після вибору сервісів та технологій для різних задач у додатку можна переходити до створення архітектурної схеми застосунку.

2.3 Архітектура застосунку

На початку підрозділу показана базова архітектура. Поетапно будуть додаватись зображення різних додаткових флоу в додатку, які включають Lambda, EventBridge, SNS, сторонній API, S3, RDS, Pinecone. В кінці показано повну архітектуру додатку.

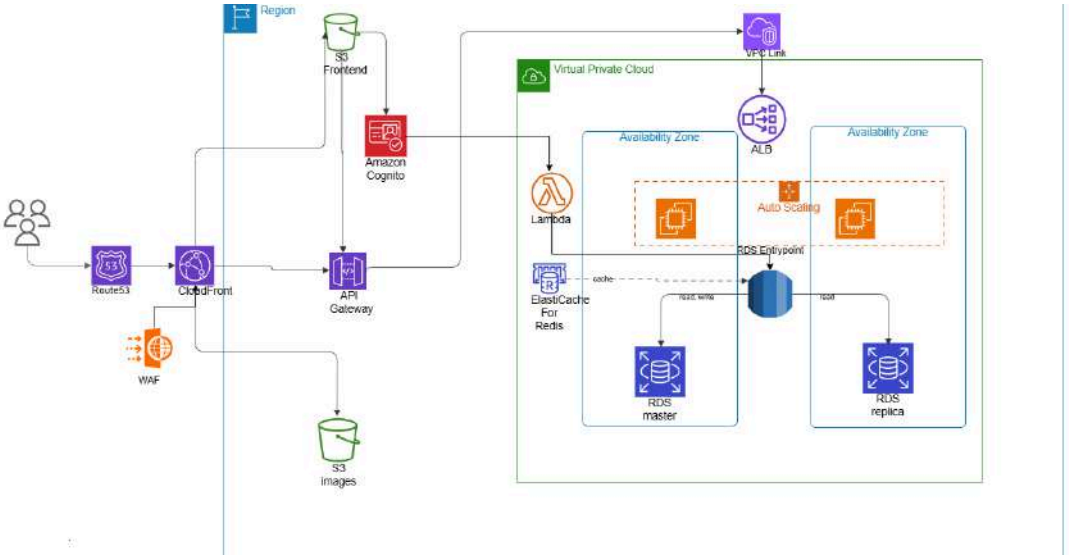


Рисунок 2.1. Базова архітектура застосунку

Користувач звертається до Route 53, який керує DNS-записами та маршрутизує трафік до CloudFront. WAF інтегровано з CloudFront на глобальному рівні для фільтрації трафіку ще до того, як він потрапляє на фронтенд. За необхідності можна також додати WAF перед API Gateway чи ALB, тоді він буде інтегрований на регіональному рівні та на рівні VPC відповідно. Варто розуміти, що WAF насправді не є фізичним компонентом, а є логічним рівнем, який прив'язується до компонента, у випадку даної архітектури до CloudFront. Розмістивши WAF таким чином, ми і захищаємо статичні ресурси в S3, і фільтруємо запити до бекенду. CloudFront та Route 53 глобальні сервіси, тому не знаходяться в межах регіону. CloudFront можна налаштувати на кешування даних з API Gateway, S3 фронтенд та S3 з зображеннями знахідок. Якщо в кеші є інформація потрібна користувачу, то він одразу отримує відповідь, інакше запит передається далі у відповідний сервіс. Коли CloudFront отримує відповідь від сервіса, він відправляє її користувачеві та зберігає в кеш.

Клієнтський код, розміщений на S3, звертається до AWS Cognito для автентифікації. Користувач заходить в систему або реєструється і отримує JWT токен, який потім використовується для розпізнавання користувача при наступних запитах до API Gateway. При створенні нового користувача викликається Lambda,

яка додає його в базу даних.

Клієнтський код при взаємодії з користувачем відправляє запити на бекенд через API Gateway. CRUD запити, пов’язані з постами про знахідки йдуть у VPC (Virtual Private Cloud). VPC - ізольована від публічної частина мережі, використовується, щоб до компонентів у ній був доступ тільки з цієї ж мережі, а зовнішній доступ потрібно було додатково налаштовувати. Так, API Gateway з’єднаний з VPC за допомогою VPC Link. VPC Link вказує на ALB, тобто балансувальник навантаження рівня застосунку. Балансувальник розподіляє навантаження між екземплярами EC2. Для масштабування EC2 налаштовано AutoScaling Group. Буде мінімум два екземпляри в різних зонах доступності. Таким чином рішення стає надійнішим, адже якщо сервера в одній зоні доступності вийдуть з ладу, залишаться сервера в іншій. При значному збільшенні навантаження в AutoScaling Group будуть створені додаткові екземпляри EC2, які візьмуть на себе частину навантаження. Нові EC2 потрібно додати до цільової групи балансувальника. EC2 звертаються до рівня даних для отримання та зміни інформації. Звернення до бази даних йдуть через Entrypoint - логічний маршрутизатор, який розподіляє навантаження між мастер нодою та реплікою. Мастер нода використовується для запису та читання важливої інформації, тоді як репліка автоматично синхронізується з нею і використовується тільки для читання. Такий підхід дозволяє розділити операції запису та читання, підвищуючи продуктивність системи. Якщо мастер виходить з ладу, то його функції перебирає на себе репліка. Щоб уникнути холодного старту при відмові мастера, обидві бази повинні бути постійно активними. Такий підхід називають hot standby, його використання дає змогу уникнути простоїв, які неодмінно з’являються, якщо репліка запускається лише після падіння мастера. Додатково використовується ElastiCache на базі Redis для кешування часто запитуваних даних. Таким чином бекенд спершу звертається до кешу, якщо там немає потрібних даних, тоді вже йде звернення до бази даних. Отримані дані записуються у Redis, після чого відправляються користувачу. Під час наступного запиту на ті ж дані, вони миттєво повернуться користувачеві з Redis без звернення до RDS, що прискорить

відповідь.

При пошуку клієнтом схожих фото запит через API Gateway йде на Lambda. Та в свою чергу звертається до стороннього API і з використанням моделі CLIP генерує вектор ознак фотографії. Вектор порівнюється з тими, що вже наявні в БД для пошуку найбільш схожих. Зрештою Lambda повертає користувачеві відповідь - найбільш схожі візуально знахідки. При створенні будь-якого поста про знахідку в S3 з зображеннями буде додаватись нове фото. Додавання тригеритиме Lambda, яка генеруватиме вектор ознак фотографії та зберігатиме його в Pinecone для майбутнього пошуку.

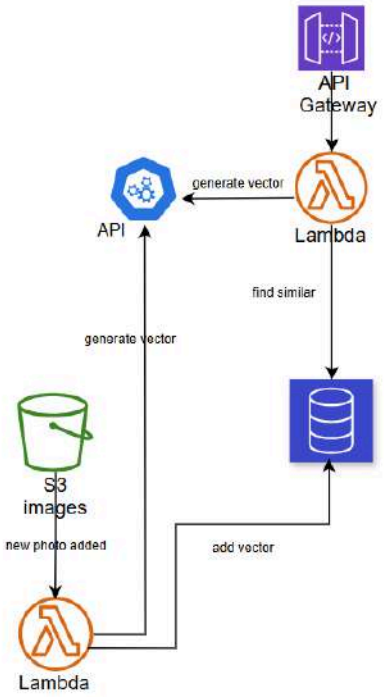


Рисунок 2.2. Архітектура пошуку за зображенням

Два флоу містять EventBridge та Lambda. Перший з них - нотифікація користувачів. Користувач створює оголошення про втрату і в цей момент підписується на SNS топік. Потім при створенні нових оголошень про знахідку, EventBridge передає подію Lambda. Lambda додає у відповідний SNS топік повідомлення, яке відправляється підписникам топіка - телефонам та поштам користувачів.

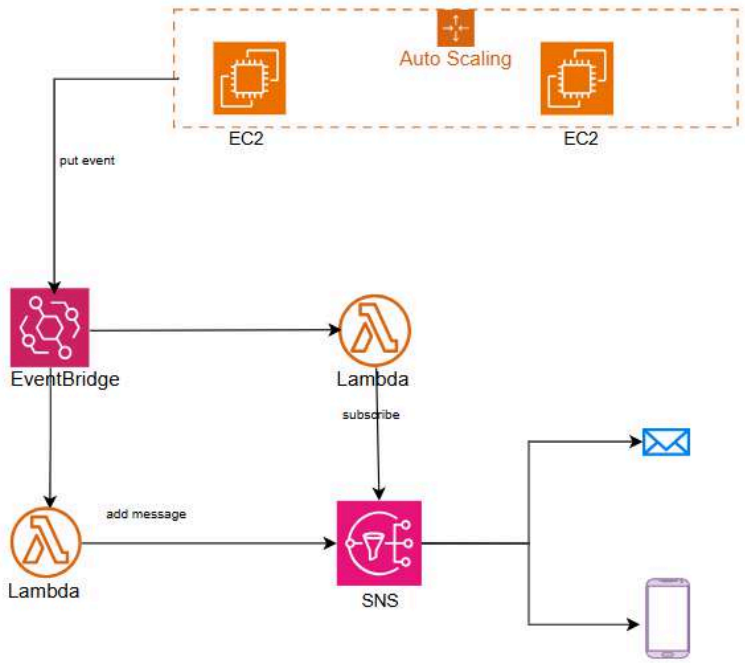


Рисунок 2.3. Архітектура нотифікації про схожу знахідку

Другий флоу - очистка застарілих та знайдених постів. В EventBridge налаштовується cron, тобто регулярність з якою подія буде передаватись отримувачу. Лямбда отримує сигнал та робить очистку.

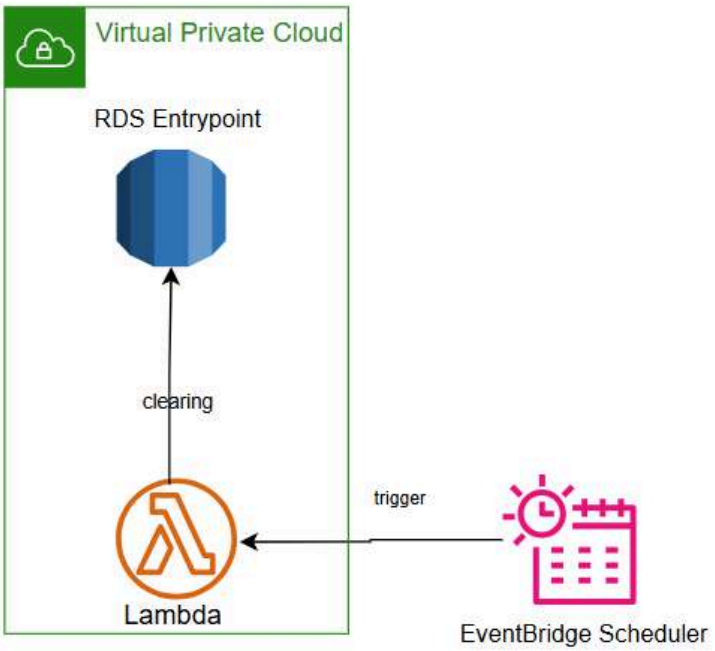


Рисунок 2.4. Архітектура очищення застарілих постів

EventBridge працює на рівні регіону, тому не знаходиться у VPC. При цьому компоненти всередині VPC можуть викликатись за допомогою EventBridge, так як викликається Lambda для очистки постів.

Lambda функції, які працюють з базою даних RDS, знаходяться у VPC. Таких Lambda дві: одна лямбда додає користувачів при реєстрації, друга робить очистку постів. Lambda, які не взаємодіють з RDS розташовані на рівні регіону. Таких Lambda чотири: дві для реалізації пошуку схожих зображень і дві для нотифікації користувачів.

Підсумуємо сценарії роботи системи.

Таблиця 2.1. Сценарії роботи системи.

Операції	Сервіси
CRUD для постів про знахідки	EC2
Очистка знайдених та застарілих постів	Event Bridge + Lambda
Пошук схожих фотографій	S3 trigger + Lambda + Pinecone
Нотифікація користувачів про знахідку схожої речі	Event Bridge + Lambda + SNS

Повну архітектуру додатку наведено нижче.

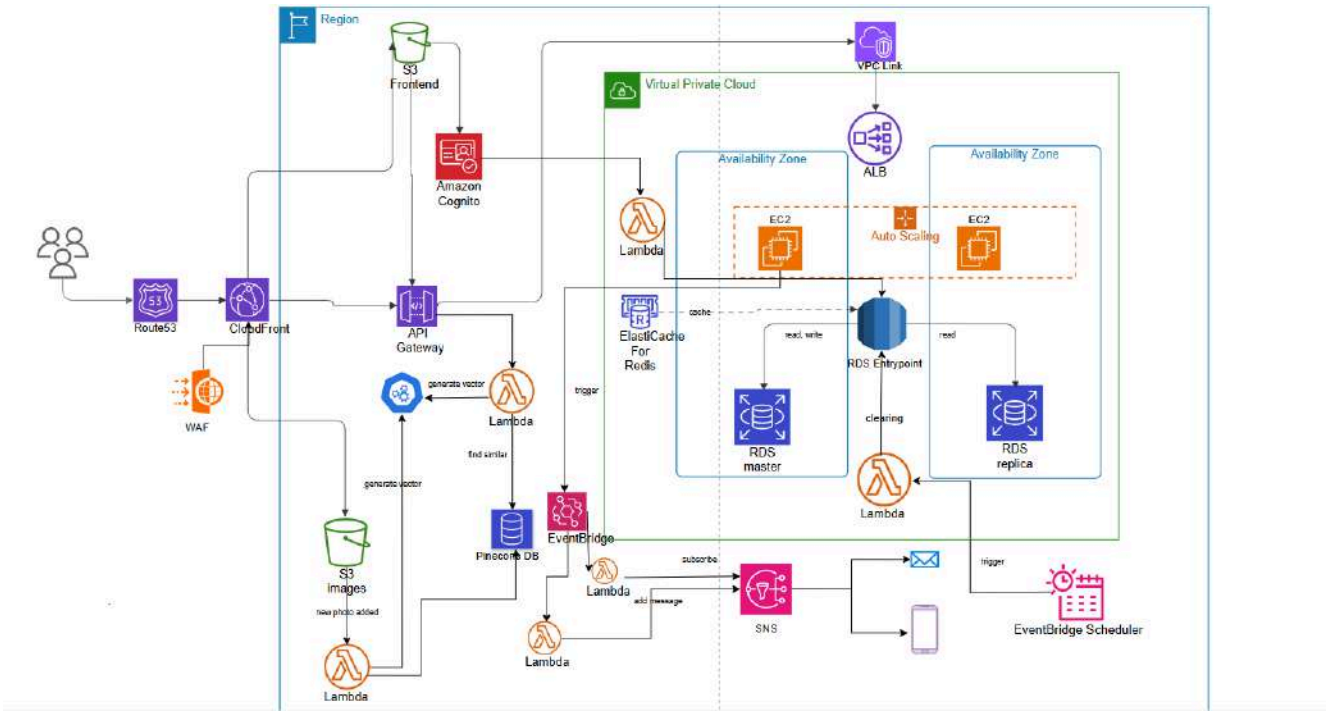


Рисунок 2.5. Повна архітектура застосунку

CloudWatch автоматично інтегрується з більшістю сервісів AWS і забезпечує логування і моніторинг в системі.

Безсерверні та глобальні компоненти гарантовано мають високу доступність. Компоненти, які потребують дій розробника для забезпечення доступності в архітектурі додатку - EC2 та RDS. Екземпляри EC2 розміщені в AutoScaling Group в двох різних зонах доступності. Екземпляри RDS так само розміщені в двох зонах доступності, при відмові мастера його роль на себе бере репліка. Репліка постійно активна, що збільшує доступність, адже зменшується час, коли база даних не доступна. Крім того, дані можна зчитати з кеш бази Elasticache на основі Redis. Доступність додатково підвищує кешування на рівні CloudFront, яке дозволяє повертати збережені дані навіть коли сервери недоступні. Route 53 в свою чергу перевіряє доступність граничних серверів CloudFront перед направленням запиту на них. З урахуванням вищесказаного, архітектура є надійною і високодоступною.

3. Реалізація компонентів системи

В розділі реалізації не буде реалізовано API і каркас застосунку, який спільний для багатьох архітектур на AWS і який реалізований вже безліч разів. Мова йде про налаштування EC2 з AutoScaling Group та ALB, RDS, VPC, Cognito авторизацію, CloudFront та Route 53. Натомість реалізація зосереджена на тих компонентах, які специфічні для конкретної системи: пошук за зображенням; очистка знайдених постів; відправка повідомлень про те, що схожа річ знайдена. В даному розділі детально показано розгортання компонентів архітектури та їх взаємодію. Код компонентів написаний на Node JS з використанням AWS SDK.

3.1 Реалізація пошуку за зображенням

Архітектура цієї частини системи зображена на рисунку 2.6. Вибір компонентів загалом пояснений в другому розділі. Події пов’язані з пошуком тригеряться доволі рідко. Події додавання постів теж відбуваються відносно не часто, тому використана лямбда. Лямбда звертається до стороннього API . Однією з причин такої архітектури є потреба у швидкій реалізації. В майбутньому можна розгорнути модель на власних потужностях в декілька різних способів: контейнеризувати з лямбдою; створити періодичне EC2/ECS завдання, яке запускатиметься раз у певний час і оброблятиме нові зображення, які з’явились в S3. При зростанні системи, якщо створення постів відбуватиметься дуже часто, можна буде подумати про зміну архітектурного рішення. Для пошуку в будь-якому разі краще використати лямбду, ніж тримати окремий сервер.

Реалізацію флоу розпочато зі створення S3 для зображень.

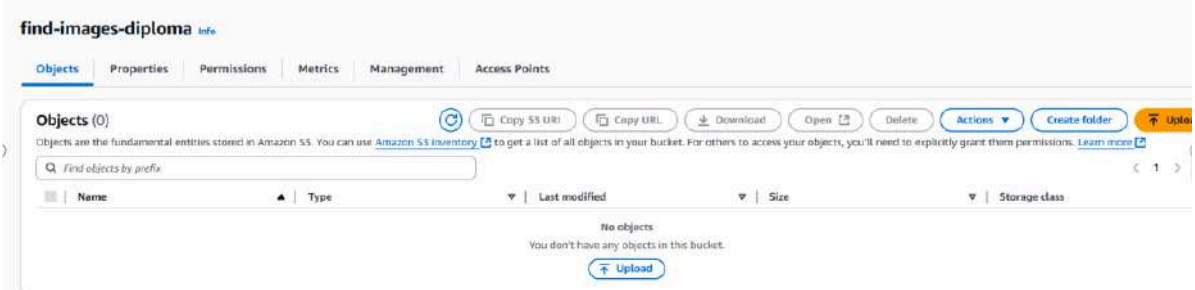


Рисунок 3.1. S3 для зображень

Створено лямбда функцію для перетворення зображень на вектор та додавання його в Pinecone.

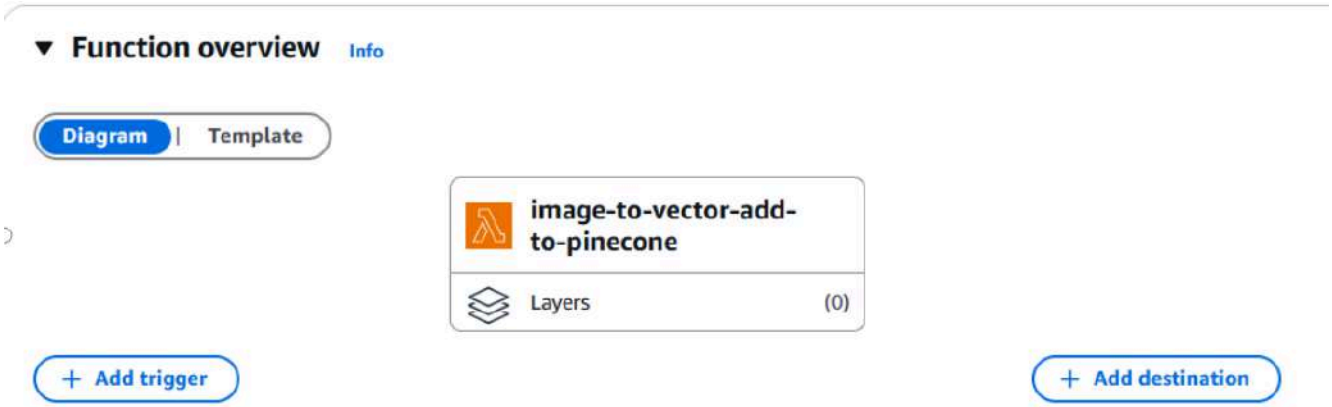


Рисунок 3.2. Лямбда для перетворення зображення на вектор та додавання його в базу даних.

Створено тригер в S3 на подію додавання зображення, який буде викликати Лямбда функцію.

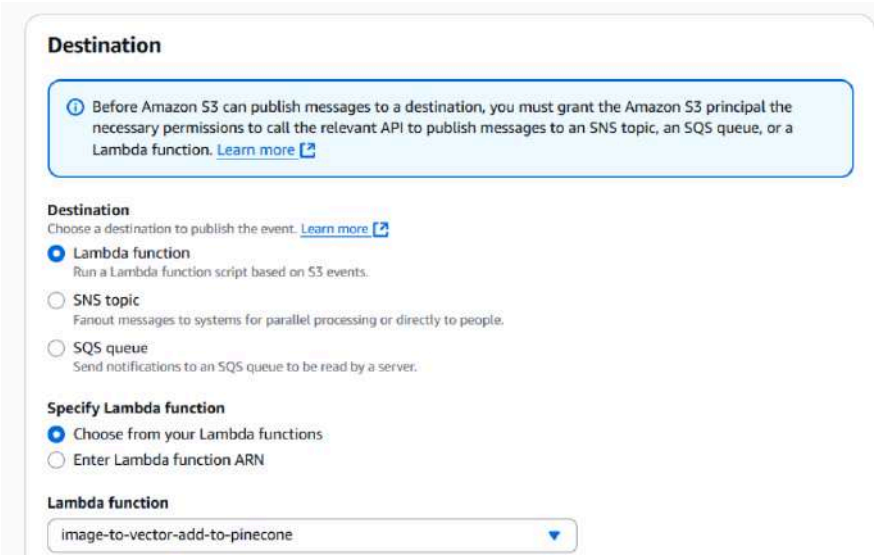


Рисунок 3.3. Створення тригеру S3 для виклику лямбди

Необхідно налаштувати дозволи для лямбди, щоб вона могла отримувати зображення з S3. Для лямбди треба створити роль з такими правами:

- читати об'єкти з S3 bucket.

- робити HTTPS-запити до стороннього API з моделлю та Pinecone.

До ролі лямбда додано політику читання з S3.

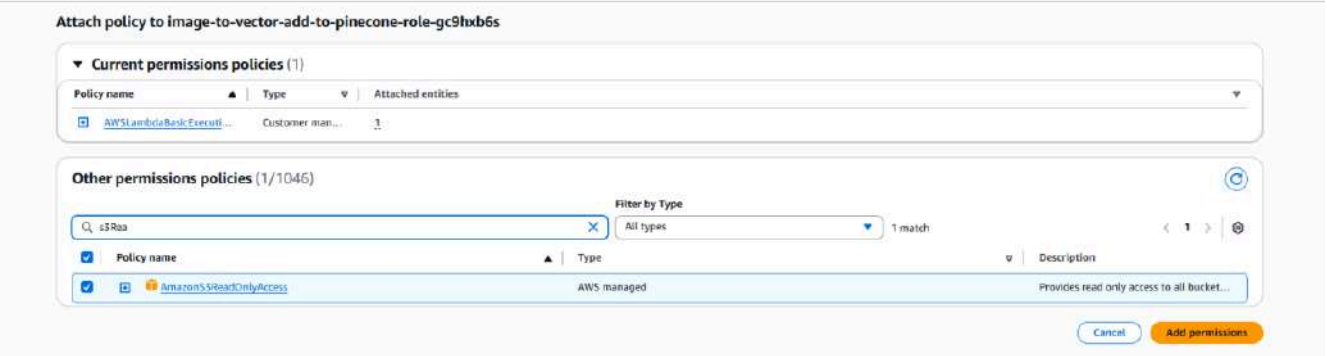


Рисунок 3.4 Політика читання S3 для лямбди

Для доступу до HTTPS-ендпоінтів додаткових IAM-прав не потрібно: достатньо, щоб у Lambda був вихід в інтернет (через NAT Gateway або щоб функція знаходилась поза VPC). Лямбда знаходиться поза VPC у межах регіону, відповідно додаткових політик доступу та конфігурації не потребує.

Перейдімо до реалізації лямбди. Її задача наступна: відправити зображення з S3 до стороннього API, отримати у відповідь вектор ознак, записати його у векторну базу даних Pinecone. Для звернення до Pinecone API необхідно зареєструватись на сайті. Pinecone генерує дефолтний ключ при реєстрації.

Створимо базу даних в Pinecone. При створенні вказуємо, що зберігатимемо 512-розмірні вектори, які отримуватимемо від CLIP, та використовуватимемо косинусну метрику подібності для пошуку.

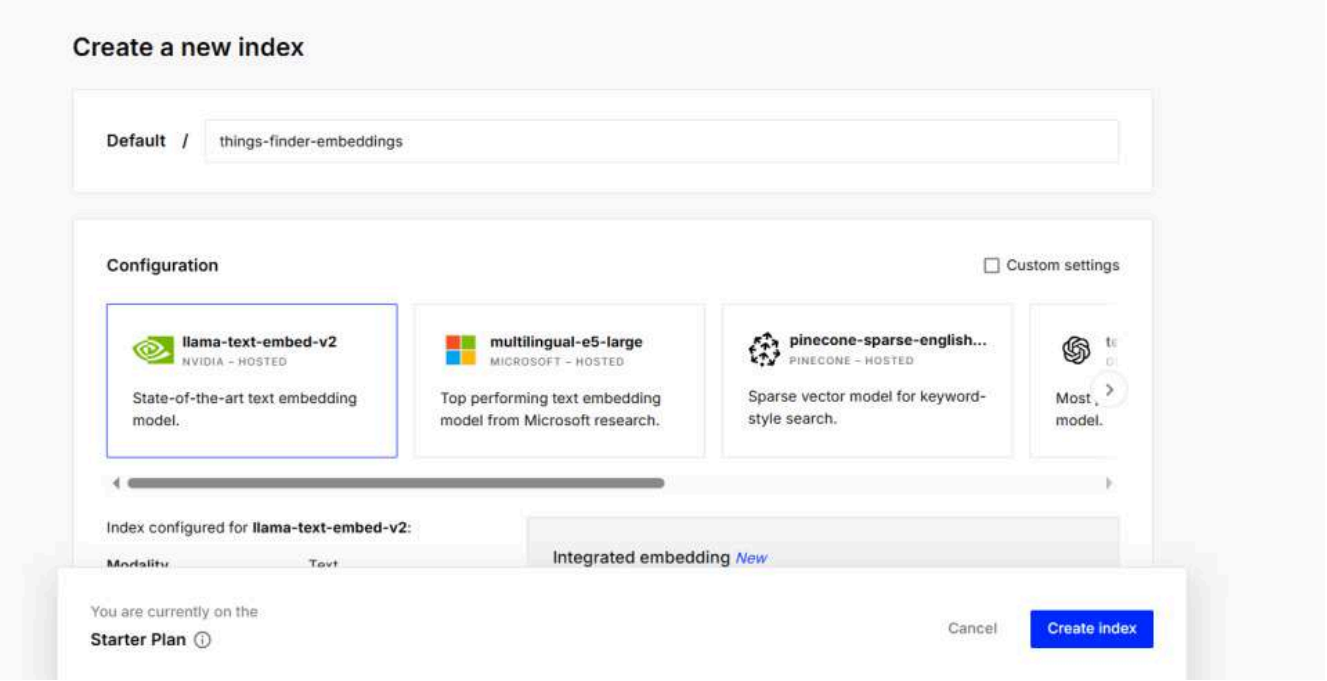


Рисунок 3.5. Розгортання бази даних Pinecone

Далі налаштуємо змінні оточення лямбди - додамо API ключ для підключення до бази даних Pinecone.

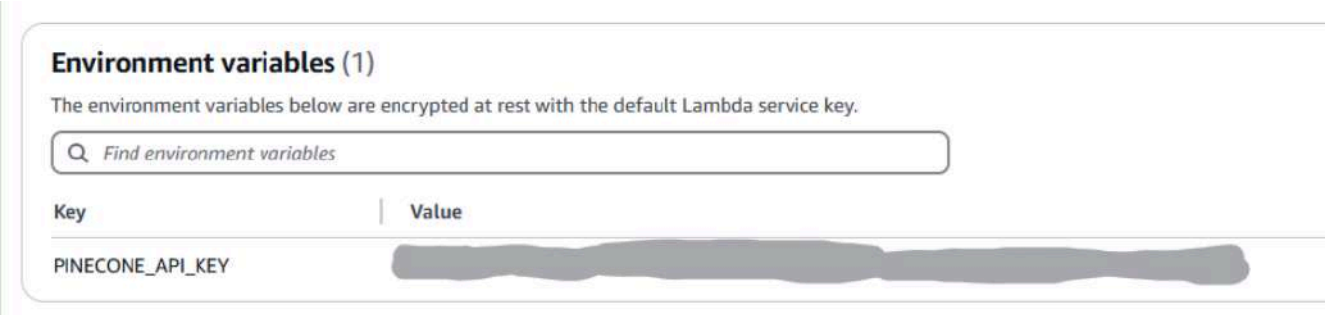


Рисунок 3.6. Додавання змінних оточення в лямбду

В лямбді не можна напряму імпортувати залежності. Потрібно або запаковувати файли коду з модулями в zip, або контейнеризувати лямбду, вказуючи всі потрібні пакети. При контейнеризації образ спершу необхідно завантажити в Amazon ECR, реєстр контейнерів, а потім в Lambda обрати тип виконання Container Image. Для лямбди є і третя опція, яка дозволяє відділити бібліотеки від коду в окремий шар - Lambda Layers. Щоб додати Lambda Layer було створено та заархівовано проект з усіма потрібними залежностями. Далі створено шар, в який завантажено архів. Шар додано до Lambda функції.

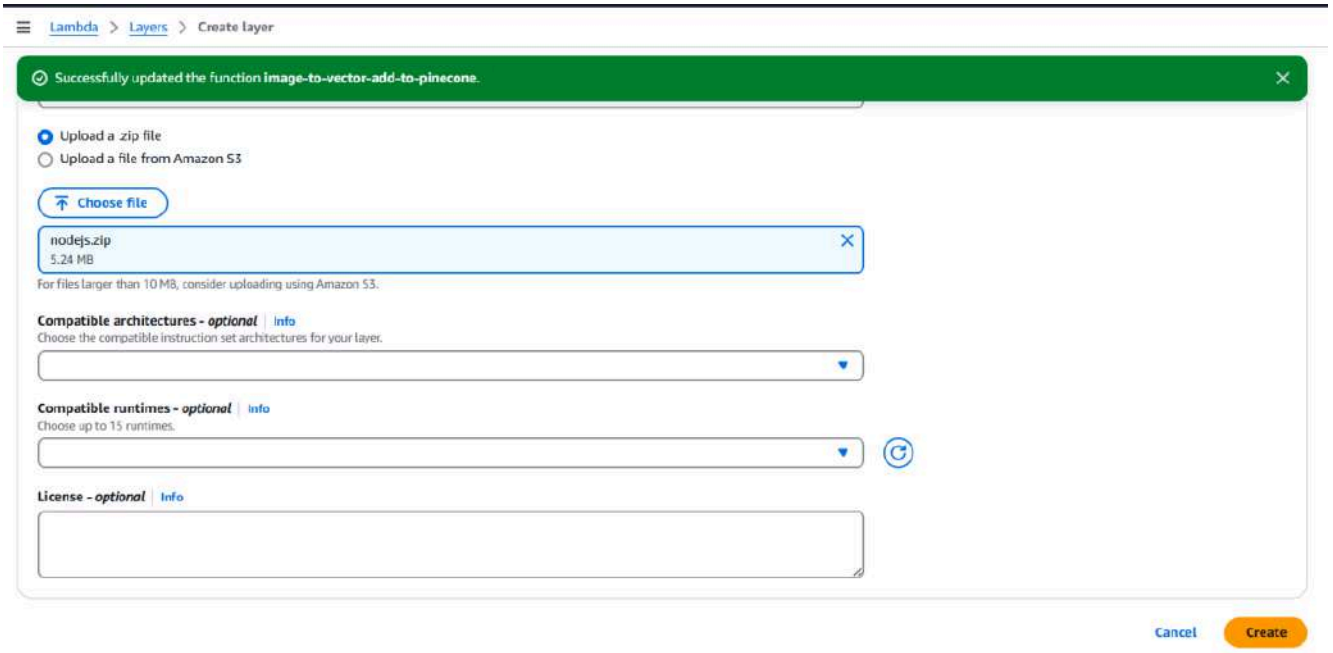


Рисунок 3.7. Процес створення Lambda Layer

Сторонній сервіс, до якого звертається Lambda для векторизації не буде реалізований в рамках розгортання цього флоу. Сервісом може виступати будь-який сервер, на якому є потрібна модель, або навіть інша лямбда функція, яка контейнеризована, мінімізована, і містить модель в залежностях. Замість звернення до API буде згенеровано випадковий вектор розмірністю 512.

В лямбда функцію передається подія, яка містить назву бакета та назву зображення. Назва зображення буде виступати ключем в базі даних.

```
const record = event.Records[0].s3;
const bucket = record.bucket.name;
const key = decodeURIComponent(record.object.key.replace(/\+/g, ' '));

const getObj = await s3.send(new GetObjectCommand({ Bucket: bucket, Key: key }))
```

Вектор разом з метаданими додається до бази даних за допомогою наступного коду.

```
await index.upsert([{
    id: key,
    values: vector,
    metadata: metadata
}]);
```

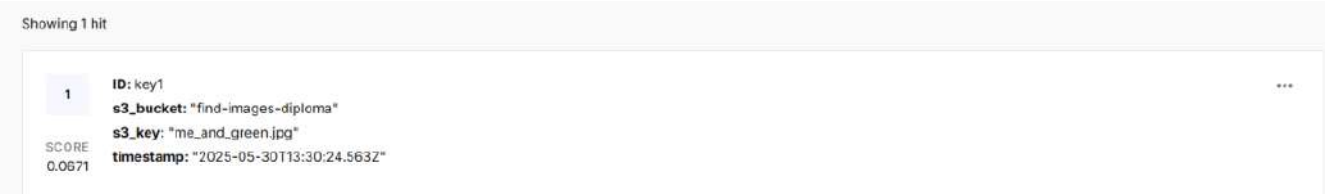


Рисунок 3.8. Вектор в Pinecone

На цьому флоу обробки додавання нових зображень завершений.

Тепер можна перейти до флоу пошуку схожих зображень. Для початку створено Lambda, яка буде перетворювати зображення у вектор та шукати схожі вектори у базі даних. Процес аналогічний до створення попередньої лямбди, буде підключений той же Lambda Layer, проте цю лямбду не треба під'єднувати до S3 через тригери.

Код для обробки зображення виглядає так:

```
let bodyBuffer;
if (event.isBase64Encoded) {
   bodyBuffer = Buffer.from(event.body, "base64");
} else {
bodyBuffer = Buffer.from(event.body, "binary");
}
```

Код для пошуку схожих векторів так:

```
const resp = await index.query({
    vector: queryVector,
    topK: 3,
    includeValues: false,
    includeMetadata: true,
})
```

На виході - ключі трьох найбільш схожих векторів за метрикою косинусної

подібності, які є назвами файлів в S3, і в основній базі RDS.

Щоб Lambda виконувалась потрібно маршрутизувати їй запити. Задачу перенаправлення виконуватиме API Gateway. Для створення API обрано його тип - REST API у випадку дипломного додатку.

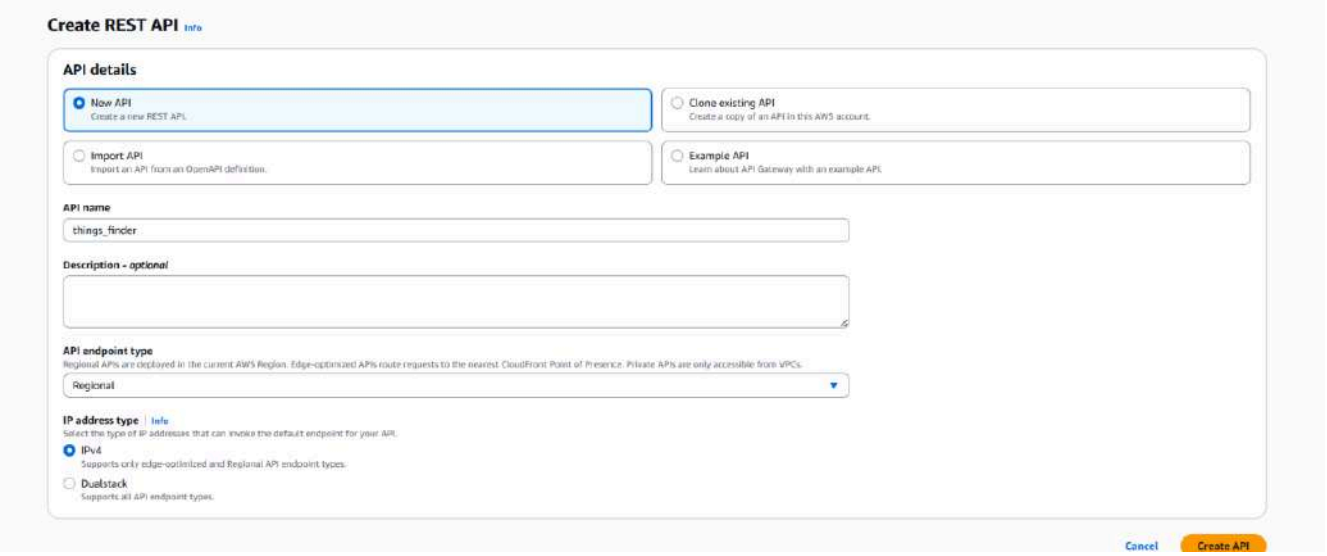


Рисунок 3.9. Створення REST API в API Gateway

Визначено path, за яким користувач буде звертатись до ресурсу. Створено POST метод на цей ресурс, який перенаправляє запит на Lambda.

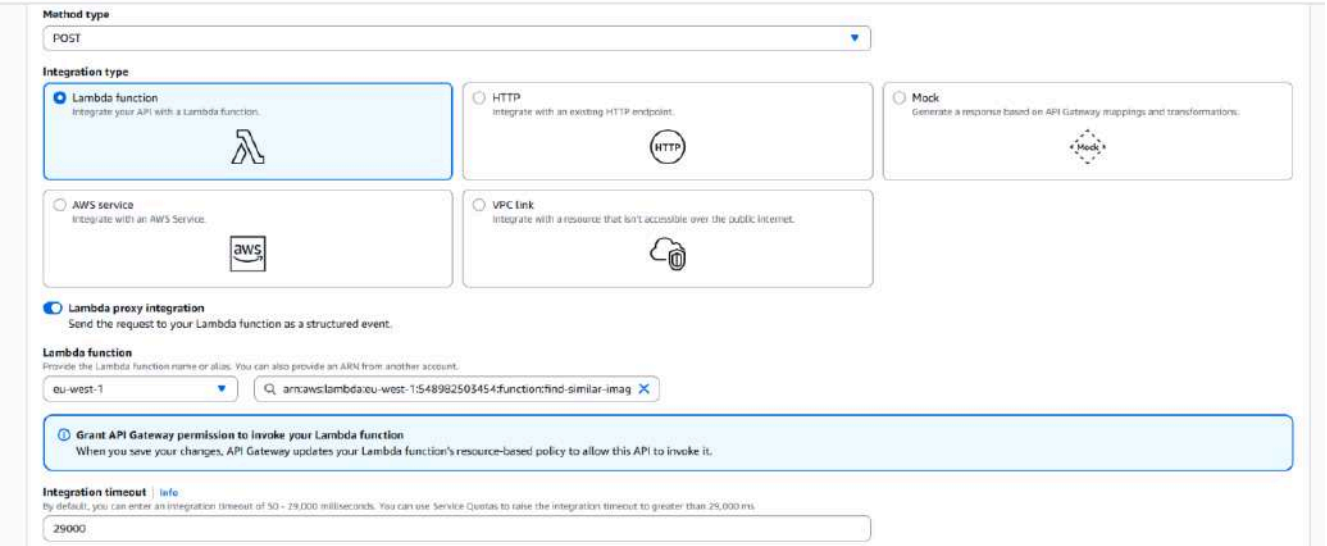


Рисунок 3.10. Створення POST ендпоінту для пошуку схожих зображень

API було розгорнуто, можна перевірити ендпоінт в Postman. Зображення передано у вигляді бінарного файлу.

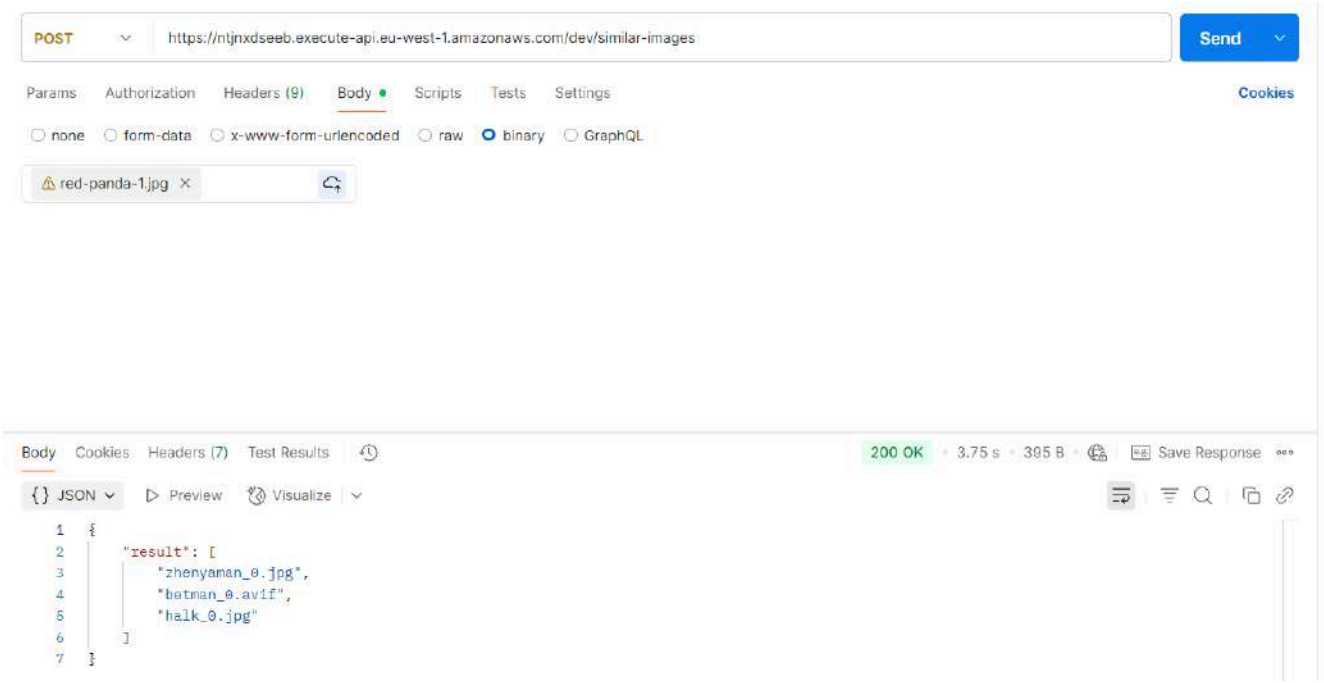


Рисунок 3.11. Результат запиту на POST ендпоінт для пошуку схожих зображень

Флоу отримання схожих зображень теж реалізовано, отже реалізацію даної частини системи завершено.

3.2 Нотифікація користувачів

Вибір сервісів для нотифікації описаний в другому підрозділі. Так як EC2 не розгортатимуться в межах даного флоу, додавати подію в EventBridge буде Lambda, яка симулює додавання з EC2. Діаграму даної частини застосунку можна побачити на рисунку 2.3. Видозмінена для реалізації діаграма розміщена нижче.

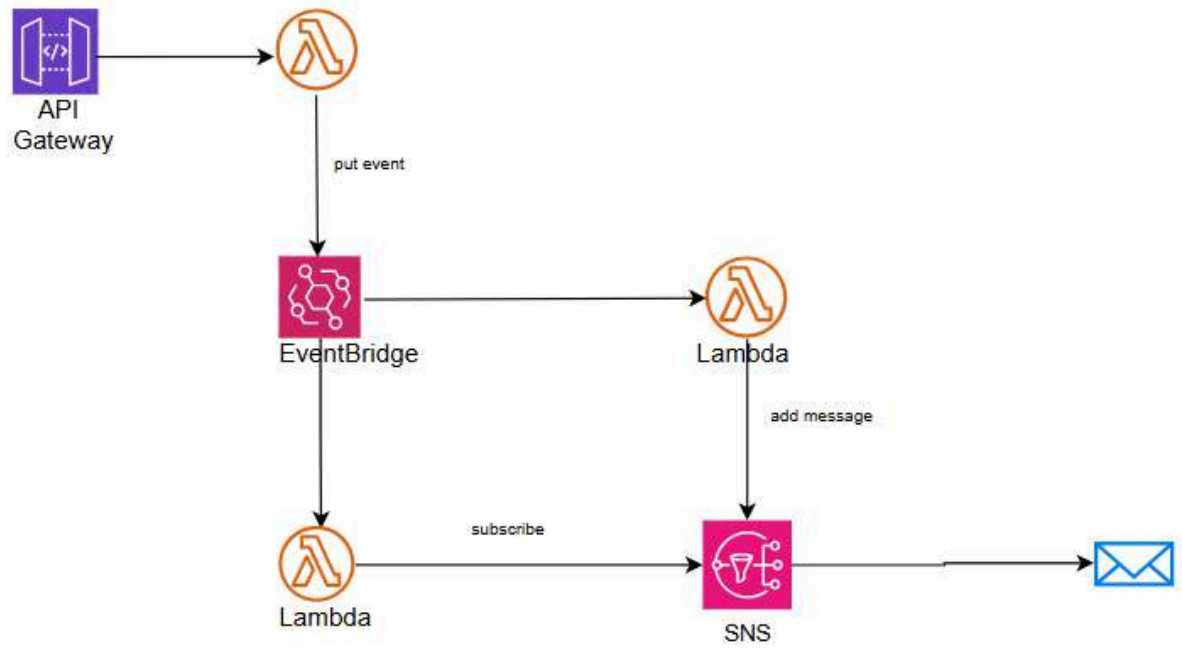


Рисунок 3.12. Змінена для реалізації архітектура нотифікації

В API Gateway буде створено ендпоінт, який направлятиме запити на лямбду. Лямбда виконуватиме функцію EC2 в цьому флоу, тобто передаватиме подію з корисним навантаженням до EventBridge.

На діаграмі присутні дві лямбди, які взаємодіють з SNS. Одна має підписувати користувача на SNS топік, друга надсилати в топік повідомлення. Всі підписники топіка отримають повідомлення на пошту.

Першим кроком створено лямбду, яка додаватиме подію, та ендпоінт в API Gateway.

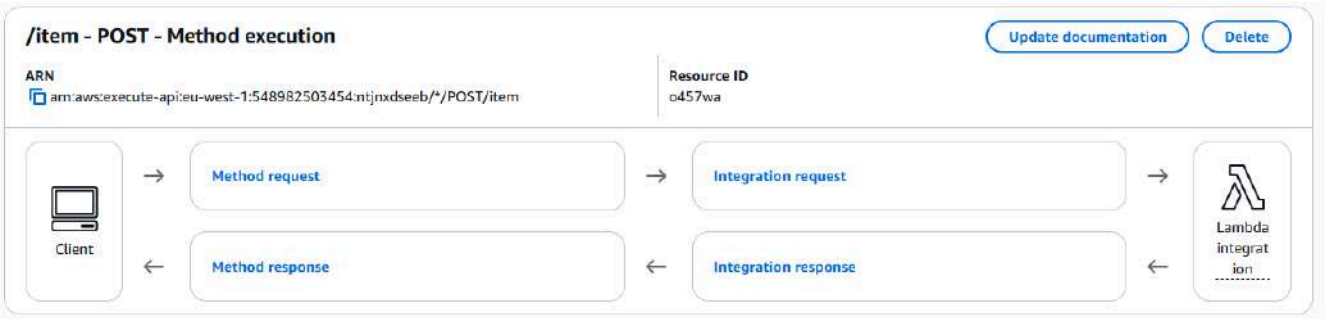


Рисунок 3.13. Направлення на лямбду через ендпоінт

Далі створено EventBridge bus - шина, де сервіс зберігає події для передачі іншим сервісам.

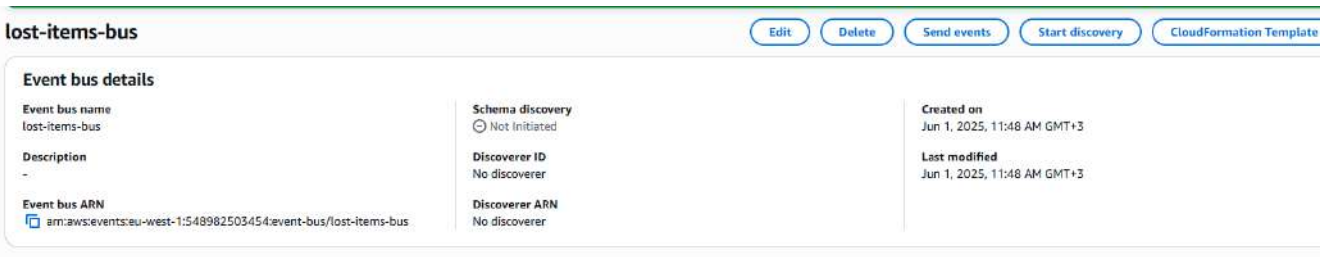


Рисунок 3.14. EventBridge bus

Лямбді надано дозвіл на запис в lost-items-bus, для цього створено і додано до її ролі наступну політику:

```
"Effect": "Allow",
"Action": "events:PutEvents",
"Resource": "arn:aws:events:eu-west-1:548982503454:event-bus/lost-items-bus"
```

Для лямбди також створено і додано Layer з необхідними залежностями.

Лямбда приймає корисне навантаження та передає в EventBridge за допомогою наведеного нижче коду.

```
const params = {
  Entries: [
    {
      Source: "myapp.lost-and-found",
      DetailType: "LostItemCreated",
      Detail: JSON.stringify(item),
      EventBusName: BUS_NAME,
    },
  ],
};

const result = await ebClient.send(new PutEventsCommand(params));
```

EventBridge отримує таке body:

```
const { category, city, authorEmail, userId, found }
```

Залежно від значення змінної found EventBridge направлятиме повідомлення на потрібну лямбду. Якщо found true, значить річ було знайдено, і повідомлення передається лямбді розсилки, інакше - лямбді підписки.

Далі створено дві вищезгадані лямбди. При створенні правил маршрутизації запитів ролі з дозволами тригерити лямбда функції створюються автоматично. Правила задати можна у форматі JSON. Нижче наведено одне з двох правил.

```
"detail": {
  "found": ["true"]
}
```

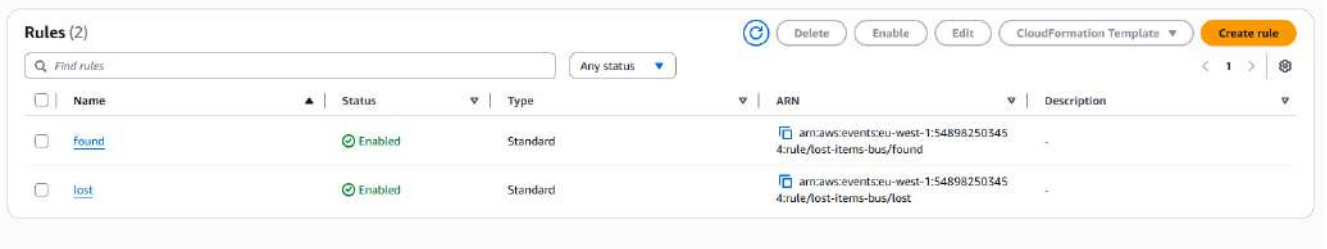


Рисунок 3.15. EventBridge правила для маршрутизації запитів на потрібну лямбду

Для кожної лямбди додано відповідний Layer з потрібними залежностями.

В лямбді підписки назва топіку конструюється з переданих в неї полів категорії та міста. Якщо топіку ще не існує, він створюється. Після цього пошта користувача підписується на топік. Це відбувається за допомогою наведеного нижче коду.

```
const { TopicArn } = await sns.send(
  new CreateTopicCommand({ Name: topicName })
);

await sns.send(
  new SubscribeCommand({
    TopicArn,
    Protocol: "email",
    Endpoint: authorEmail
  })
);
```

Лямбда публікації отримує дані, які містять категорію і місто, де була знайдена річ, та пошту автора. Після чого динамічно формує назву топіка, і додає в нього повідомлення про публікацію нової знахідки. Повідомлення містить пошту автора - спосіб для зв’язку з ним. Регіон та id акаунта визначені в змінних оточення лямбди. Код лямбди наведений нижче.

```
const body = event.detail;
const { category, city, authorEmail } = body;

const topicName = `lost-${category}-${city}`;
const topicArn = `arn:aws:sns:${process.env.AWS_REGION}:${process.env.AWS_ACCOUNT_ID}:${topicName}`;

const message = `Someone may have found your lost item in ${city}. Contact: ${authorEmail}`;

await sns.send(new PublishCommand({
  TopicArn: topicArn,
  Message: message,
  Subject: "Possible Match Found"
}));
```

Для лямбди підписки додано можливість створювати топіки та підписуватись на них за допомогою наведеної нижче політики.

```
{
    "Sid": "Statement2",
    "Effect": "Allow",
     "Action": [
        "sns:CreateTopic",
        "sns:Subscribe"
        ],
    "Resource": "*"
}
```

В свою чергу для лямбди публікації додано можливість публікувати повідомлення в SNS топік.

Тепер можна перевірити працездатність флоу. Через Postman відправимо запит на створення поста про загублену річ.

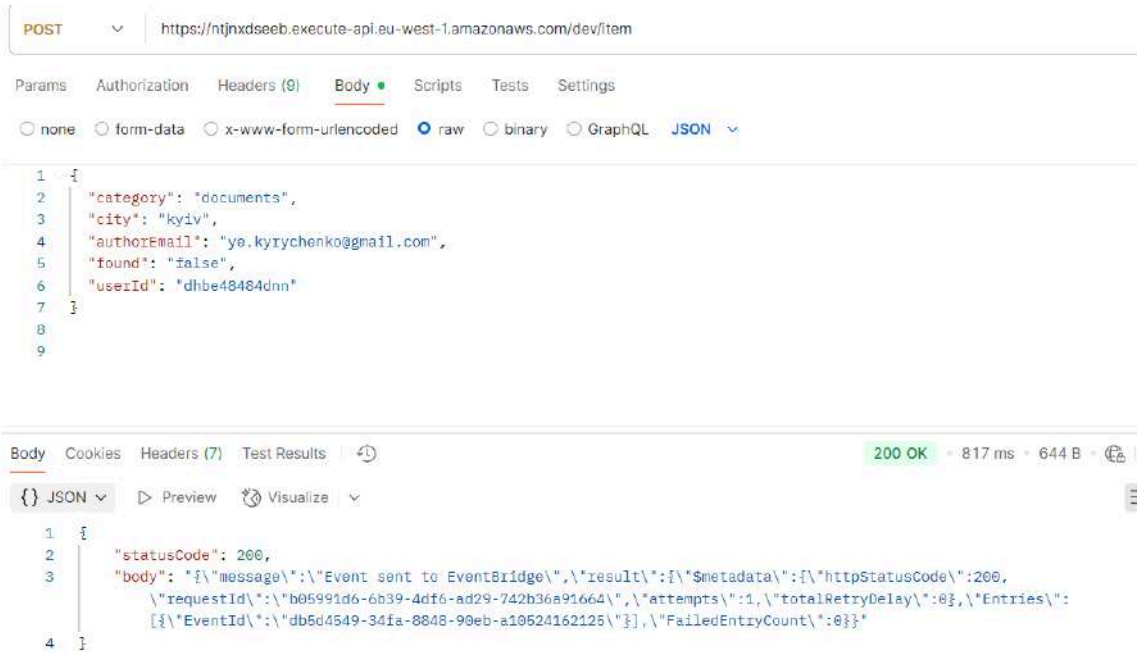


Рисунок 3.16. Запит на створення поста про втрату

Після створення поста AWS надсилає на пошту автору лист, де можна підтвердити свою підписку.

Тепер відправимо запит на створення знахідки документа в Києві.

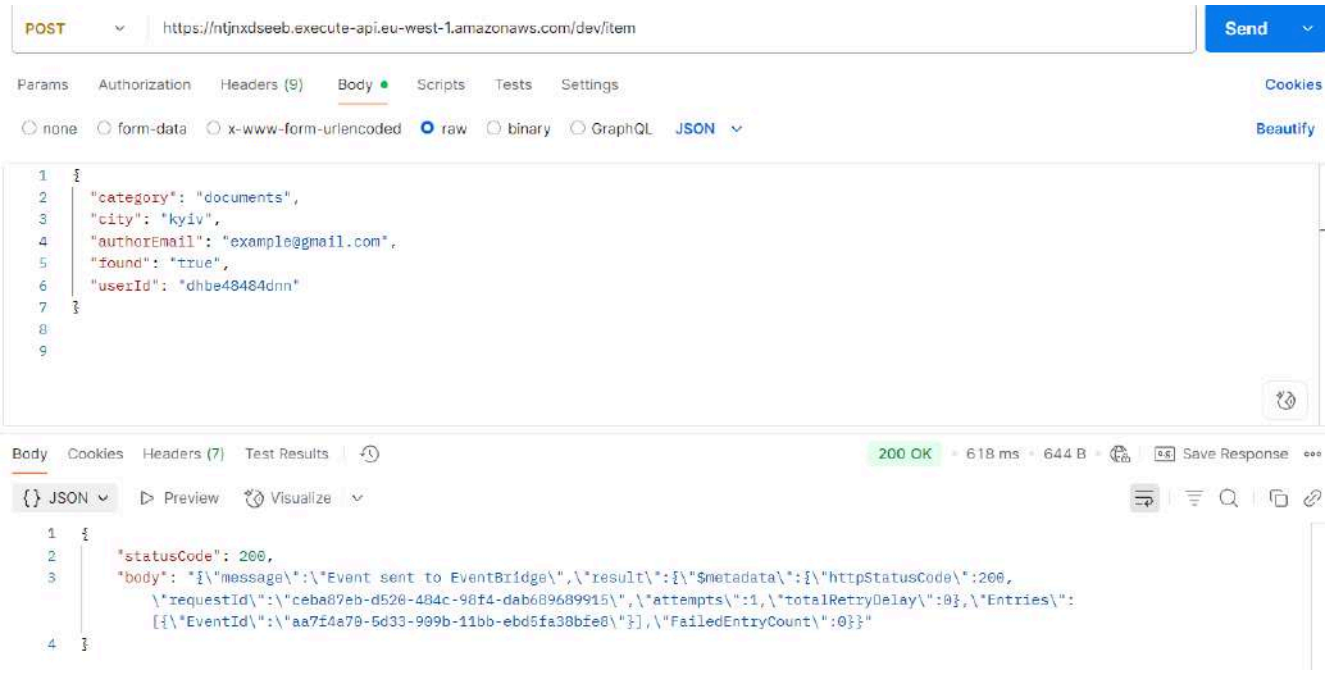


Рисунок 3.17. Запит на створення поста про знахідку

На пошту користувача, який загубив річ, приходить повідомлення про схожу знахідку. Від підписки можна відписатись у будь-який момент.

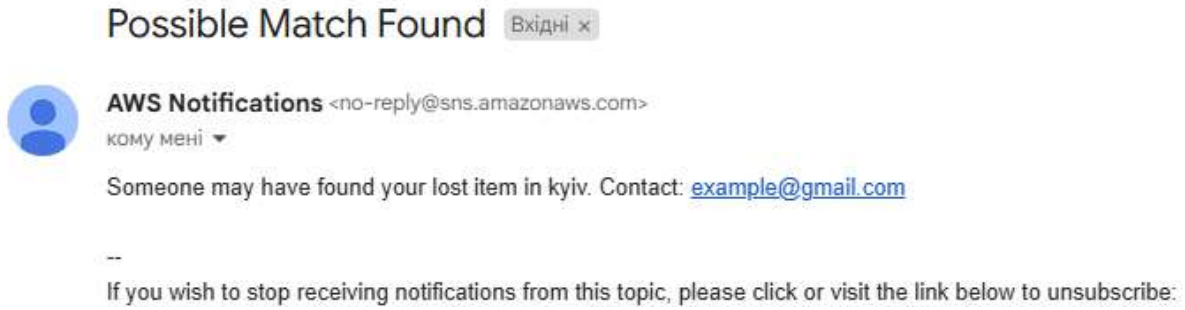


Рисунок 3.18. Нотифікація користувача про появу схожої знахідки

Флоу нотифікації користувачів закінчено, у процесі його реалізації було вручну створено 3 лямбди та Event bus, і програмно SNS топік.

3.3 Очистка БД за розкладом

Архітектура даної частини системи зображена на рисунку 2.4. У рамках

демонстрації RDS матиме публічний доступ, який можна обмежити за допомогою Security Group. Lambda в свою чергу знаходитиметься не у VPC, а на рівні регіону. Це спростить конфігурацію і дасть зробити акцент на основному - тригер лямбди очистки з визначеною періодичністю. Для реалізації флоу використано базу даних Postgres. Креди для підключення до бази можна зберігати в змінних оточення лямбди, проте кращим підходом є винесення їх в Secrets Manager. Secrets Manager дозволяє безпечно зберігати секрети, здійснювати їх автоматичну ротацію та керувати доступами до них через IAM ролі. Спираючись на вищесказане, схема очистки за розкладом фактично виглядатиме так:

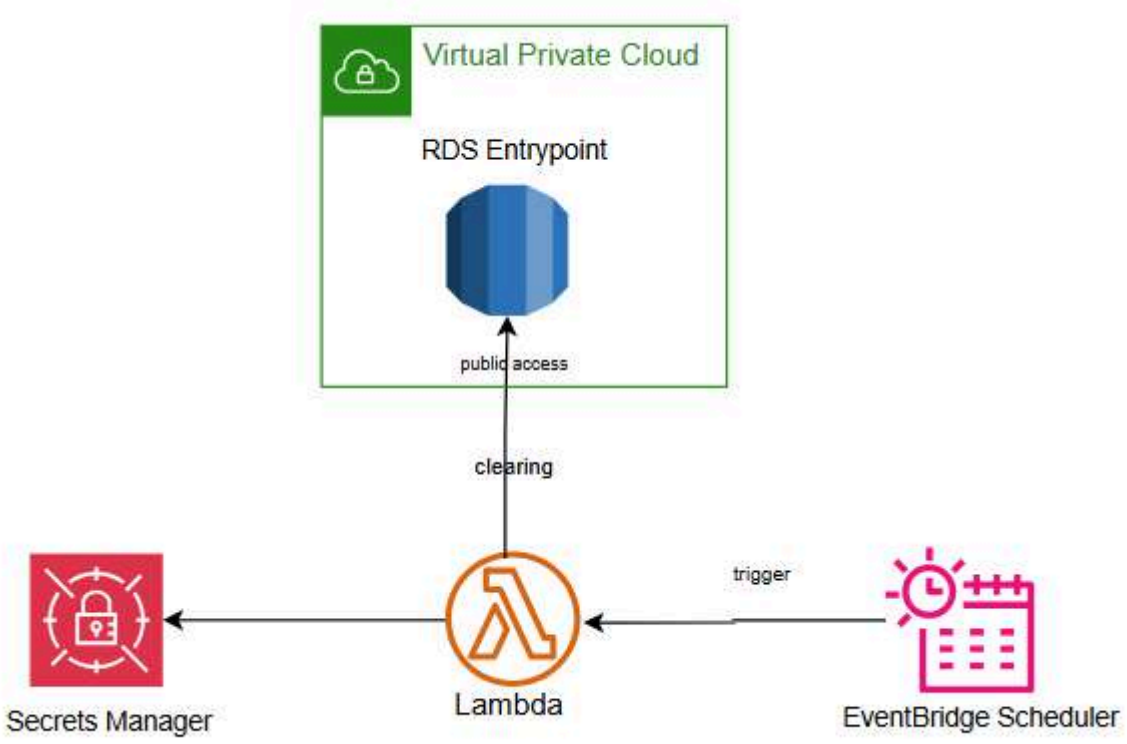


Рисунок 3.19. Змінена для реалізації архітектура очистки

Скрипт очистки видалятиме пости, створені більше 3-ох років тому, та ті, які вже були закриті.

Для реалізації створено Lambda з Layer, який містить залежності для роботи з Postgres та SceretsManager.

В EventBridge Scheduler створено розклад події, яка буде тригеритись кожні

7 днів. Далі обрано цільову функцію - Lambda очистки. Дозвіл на виклик лямбди додається автоматично при створенні розкладу.

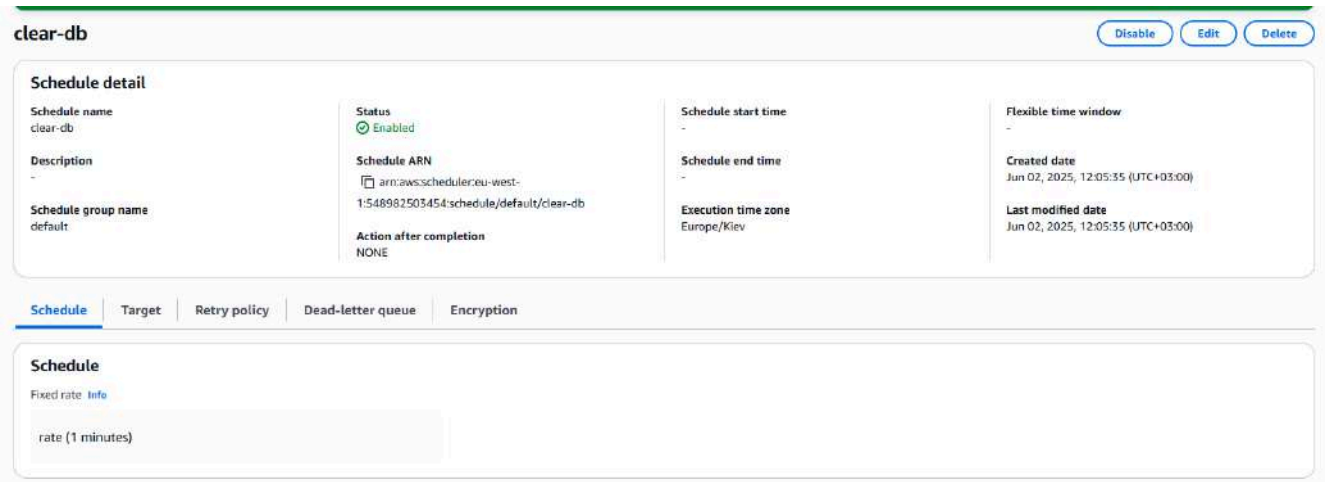


Рисунок 3.19. Створений розклад в EventBridge Sheduler

Наступний крок - створення бази даних Postgres в RDS. При її розгортанні було встановлено публічний доступ та обрано тип екземпляру db.t3.micro. Для зберігання секретів було обрано Secrets Manager.

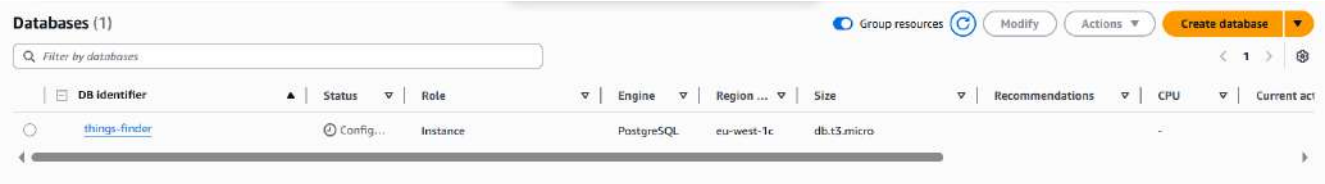


Рисунок 3.20. Створена база даних Postgres в RDS

Лямбді надано дозвіл на читання секретів, який наведено нижче.

```
"Effect": "Allow",
"Action": [
    "secretsmanager:GetSecretValue"
],
"Resource": "arn:aws:secretsmanager:eu-west-1:548982503454:secret:rds!db-b54e5e70-098a-4fc4-ad5e-89eeecc46b24-hECzWA-*"
```

Для створення таблиці та заповнення бази даних створено скрипт. В цілях демонстрації створюється 1 таблиця для знахідок та втрат, в яку додається 100 сутностей. Таблиця включатиме поля, за якими буде здійснена очистка: час створення та булева змінна, яка визначає чи закритий пост. Нижче наведено схему таблиці.

```
id SERIAL PRIMARY KEY,
title TEXT NOT NULL,
created_at TIMESTAMP NOT NULL DEFAULT NOW(),
close BOOLEAN NOT NULL DEFAULT false
```

Для демонстрації флоу для бази даних змінено Security group, а саме відкрито трафік з будь-якого джерела, та знято SSL захист. Щоб зняти захист треба було створити свою групу параметрів, вказати в ній rds.force_ssl 0 і прив’язати її до бази даних.

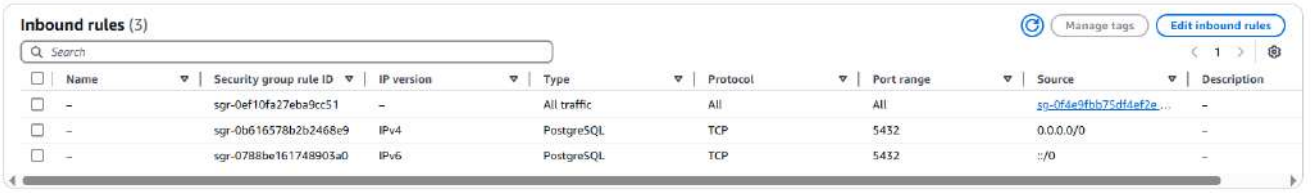


Рисунок 3.21. Налаштування Security group

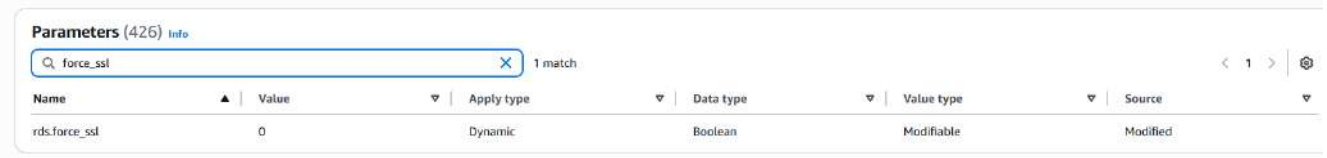


Рисунок 3.22. Створення групи параметрів

Зрештою, за допомогою скрипта вдалось заповнити базу даних. Для тестування 5 постів мають дату створення 4 роки тому і ще 5 закриті, виходячи з цього скрипт очистки має видалити 10 постів.

	id [PK] integer	title text	created_at timestamp without time zone	close boolean
1	1	Post #1	2021-06-02 11:57:38.169212	false
2	2	Post #2	2021-06-02 11:57:38.189114	false
3	3	Post #3	2021-06-02 11:57:38.208955	false
4	4	Post #4	2021-06-02 11:57:38.210961	false

Рисунок 3.23. Пости в базі даних в pgAdmin

Лямбда під’єднується до бази даних за допомогою pg Client і очищає пости за допомогою запиту на видалення. Код наведено нижче.

```
const posts = await appClient.query(`
  DELETE FROM posts
  WHERE created_at < NOW() - INTERVAL '4 years'
    OR close = true;
`);
```

Можна протестувати флоу тимчасово виставивши в Scheduler хвилинний інтервал виклику лямбди. За допомогою логів CloudWatch можна переконатись, що лямбда викликалась.

```
2025-06-02T13:40:12.862Z        5d683da9-b483-4d56-9ed3-67c70cec5807    INFO    Result {
  command: 'DELETE',
  rowCount: 10,
```

Рисунок 3.24. Логи CloudWatch з інформацією про виклик лямбди

В базі даних тепер 90 постів.

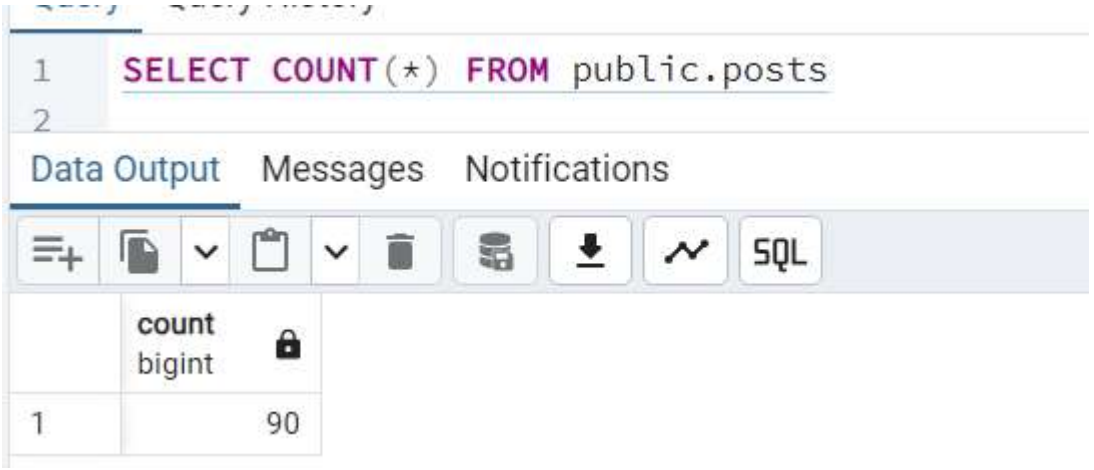


Рисунок 3.25. Підрахунок постів після очистки

Флоу очистки за графіком реалізовано. У процесі було створено Postgres базу даних в RDS, EventBridge Shedule і Lambda.

Висновки

В першому розділі здійснено аналіз аналогів та теоретичний огляд. В результаті огляду аналогів було визначено потрібні функції та покращення, які можна внести до наявних систем, зокрема функції нотифікації та пошуку за схожими фото.

Було оглянуто веб архітектури та вирішено створити застосунок на основі монолітної багатошарової архітектури. Далі розглянуто ключові компоненти багатошарової архітектури, а саме: DNS, CDN, балансувальник навантаження, API Gateway, кеш бази даних, черги повідомлень, фаєрвол, компоненти моніторингу та логування.

Наступним етапом теоретичного огляду став розбір та порівняння обчислювальних технологій платформи AWS. Було порівняно EC2, ECS та Lambda, і визначено, що для основного сервера застосунку буде обрано EC2 чи ECS, а Lambda буде використовуватись для навантажень, які відбуваються відносно рідко. Також було окремо порівняно контейнерні рішення ECS та EKS з використанням Fargate та EC2. ECS на базі Fargate визначено пріоритетним рішенням у випадку вибору CaaS.

Під час огляду інструментарію AWS було розглянуто аналоги компонентів багатошарової архітектури на платформі: Route 53, Cloudfront, ELB, API Gateway, Elasticache, SQS, EventBridge, StepFunctions, SNS, WAF, CloudWatch, CloudTrail.

Після детального аналізу теорії було сформовано технічне завдання та спроектовано схему, яка задовольняє вимогам високої доступності та масштабованості. В схемі присутні такі компоненти: Route 53, Cloudfront, ALB, API Gateway, Elasticache, WAF, EventBridge, EventBridge Scheduler, SNS, Firewall, CloudWatch, EC2 AutoScaling Group, RDS, Lambda функції, сторонній сервіс векторних баз даних Pinecone, S3, Cognito та стороннє API. Було детально обґрунтовано вибір сервісів для різних задач у застосунку. Після цього описано взаємодію компонентів архітектури та сценарії роботи системи.

В розділі реалізації детально розібрано та розгорнуто три частини системи:

пошук за зображенням, нотифікації про схожі знахідки та очищення застарілих даних.

При створенні пошуку за зображенням було продемонстровано створення API за допомогою API Gateway, доєднання до ендпоінту Lamda, роботу з S3 та налаштування виконання Lambda при додаванні в S3, додавання векторів та пошук схожих в Pinecone.

В сценарії нотифікації показано процес додавання подій в шину EventBridge за допомогою Lambda, подальшу передачу події в одну з декількох Lambda за допомогою EventBridge Rules та роботу з SNS топіками для створення розсилок користувачам на пошту.

При розробці компоненту для очищення застарілих даних було продемонстровано розгортання в RDS Postgres бази даних, створення EventBridge Schedule, визначення Lambda як цільової функції для Scheduler. Також показано взаємодію Lambda з Secrets Manager для отримання секретів бази даних та підключення до неї.

При розробці всіх трьох частин показано налаштування IAM ролей для визначення дозволів на комунікацію між компонентами. В сценарії з базою даних також показано налаштування Security Group. Загалом, практичний доробок включає роботу з наступними сервісами: API Gateway, Lambda, EventBridge, Pinecone, RDS, EventBridge Scheduler, S3, SNS, Secrets Manager та IAM.

Список використаних джерел

1. Web Application Architecture: The Latest Guide for 2025. ClickIT. URL: https://www.clickittech.com/software-development/web-application-architecture/ (дата звернення: 04.06.2025).

2. Beznos M. Microservices vs monolithic architecture: How each of them can impact your business?. N-iX. URL: https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/
(дата звернення: 04.06.2025).

3. Božena Voitkevič. Що таке система DNS і які її функції. *NordVPN*. URL: https://nordvpn.com/uk/blog/shcho-take-dns/ (дата звернення: 04.06.2025).

4. What is a CDN? - Content Delivery Network Explained - AWS. *Amazon Web Services, Inc.* URL: https://aws.amazon.com//what-is/cdn/ (дата звернення: 04.06.2025).

5. What is load balancing? | How load balancers work?. CloudFlare. URL: https://www.cloudflare.com/learning/performance/what-is-load-balancing/ (дата звернення: 04.06.2025).

6. Jackson G., Goodwin M. What Is an API Gateway? | IBM. *IBM - United States*. URL: https://www.ibm.com/think/topics/api-gateway (дата звернення: 04.06.2025).

7. Database-caching. Amazon Web Services, Inc. URL: https://aws.amazon.com/caching/database-caching/ (дата звернення: 04.06.2025).

8. What is a Message Queue?. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/message-queue (дата звернення: 04.06.2025).

9. Amazon Cloudwatch. *Amazon Web Services, Inc.* URL: https://docs.aws.amazon.com/cloudwatch (дата звернення: 04.06.2025).

10. Євгеній Сафонов. Дзен логування. Як полюбити свої логи та почати жити. Dou. URL: https://dou.ua/lenta/columns/about-logging/ (дата звернення: 04.06.2025).

11. Firewalls. IBM. URL: https://www.ibm.com/docs/en/i/7.5.0?topic=ssw_ibm_i_75/rzaj4/rzaj4fwfirewallconcept.html (дата звернення: 04.06.2025).

12. Arslan Ahmad. System Design Basics: Strategies for achieving high availability in distributed systems. *Design Gurus*. URL: https://www.designgurus.io/blog/high-availability-system-design-basics (дата звернення: 04.06.2025).

13. Davis N. Comparing ECS, EC2, and Lambda: A Guide to AWS Compute Services. *Medium*. URL: https://neal-davis.medium.com/comparing-ecs-ec2-and-lambda-a-guide-to-aws-compute-services-3cd62c0b3c48 (дата звернення: 04.06.2025).

14. Amazon EC2. *Amazon Web Services, Inc.* URL: https://aws.amazon.com/ec2/ (дата звернення: 04.06.2025).

15. What is Amazon EC2?. *Amazon Web Services, Inc. URL:* https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html (дата звернення: 04.06.2025).

16. Amazon Elastic Container Service. *Amazon Web Services, Inc. URL:* https://aws.amazon.com/ecs/ (дата звернення: 04.06.2025).

17. Amazon Elastic Kubernetes Service. *Amazon Web Services, Inc. URL:* https://aws.amazon.com/eks/ (дата звернення: 04.06.2025).

18. AWS Fargate. *Amazon Web Services, Inc. URL:* https://aws.amazon.com/fargate/ (дата звернення: 04.06.2025).

19. AWS Lambda. *Amazon Web Services, Inc. URL:*https://aws.amazon.com/lambda/ (дата звернення: 04.06.2025).

20. AWS Lambda The Ultimate Guide. *Serverless*. URL: https://www.serverless.com/aws-lambda/ (дата звернення: 04.06.2025).

21. ECS vs EC2 vs Lambda. *Digital Cloud*. URL: https://digitalcloud.training/ecs-vs-ec2-vs-lambda/ (дата звернення: 04.06.2025).

22. AWS Compute Services: Lambda vs EC2 vs ECS — Choosing the Most Cost-Effective and Highly Available Solution. *Medium*. URL:

https://medium.com/@moradiyabhavik/aws-compute-services-lambda-vs-ec2-vs-ecs-choosing-the-most-cost-effective-and-highly-available-1cfb410aff1f (дата звернення: 04.06.2025).

23. What is a CDN (Content Delivery Network)?. *Amazon Web Services, Inc. URL:* https://aws.amazon.com/what-is/cdn/ (дата звернення: 04.06.2025).

24. What is Amazon Route 53?. *Amazon Web Services, Inc.* URL: https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html (дата звернення: 04.06.2025).

25. What is Elastic Load Balancing?. *Amazon Web Services, Inc.* URL: https://docs.aws.amazon.com/elasticloadbalancing/latest/userguide/what-is-load-balancing.html (дата звернення: 04.06.2025).

26. What is an Application Load Balancer? *Amazon Web Services, Inc. URL:* https://docs.aws.amazon.com/elasticloadbalancing/latest/application/introduction.html (дата звернення: 04.06.2025).

27. What is a Network Load Balancer? *Amazon Web Services, Inc. URL:* https://docs.aws.amazon.com/elasticloadbalancing/latest/network/introduction.html (дата звернення: 04.06.2025).

28. What is a Gateway Load Balancer? *Amazon Web Services, Inc. URL:* https://docs.aws.amazon.com/elasticloadbalancing/latest/gateway/introduction.html (дата звернення: 04.06.2025).

29. What are AWS WAF, AWS Shield Advanced, and AWS Firewall Manager? *Amazon Web Services, Inc.*

URL: https://docs.aws.amazon.com/waf/latest/developerguide/what-is-aws-waf.html (дата звернення: 04.06.2025).

30. What is Amazon API Gateway?. *Amazon Web Services, Inc. URL:* https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html (дата звернення: 04.06.2025).

31. What is Amazon ElastiCache?. *Amazon Web Services, Inc. URL:* https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/WhatIs.html (дата звернення: 04.06.2025).

32. What is Amazon Simple Queue Service?. *Amazon Web Services, Inc. URL:* https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/welcome.html (дата звернення: 04.06.2025).

33. What is Amazon SNS?. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/sns/latest/dg/welcome.html (дата звернення: 04.06.2025).

34. What Is Amazon EventBridge? Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-what-is.html (дата звернення: 04.06.2025).

35. What is Step Functions. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html (дата звернення: 04.06.2025).

36. What Is Amazon CloudWatch? Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html (дата звернення: 04.06.2025).

37. What Is AWS CloudTrail?. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/awscloudtrail/latest/userguide/cloudtrail-user-guide.html (дата звернення: 04.06.2025).

38. Welcome to AWS Amplify Hosting. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/amplify/latest/userguide/welcome.html (дата звернення: 04.06.2025).

39. What is the AWS Serverless Application Model (AWS SAM)?. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/what-is-sam.html (дата звернення: 04.06.2025).

40. What Is Amazon Cognito? Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/cognito/latest/developerguide/what-is-amazon-cognito.html (дата звернення: 04.06.2025).

41. What is Amazon S3?. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html (дата звернення: 04.06.2025).

42. What is Amazon Relational Database Service (Amazon RDS)?. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html (дата звернення: 04.06.2025).

43. What is Amazon Aurora?. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/CHAP_AuroraOverview.html (дата звернення: 04.06.2025).

44. Using Aurora Serverless v2. Amazon Web Services, Inc. URL: https://docs.aws.amazon.com/AmazonRDS/latest/AuroraUserGuide/aurora-serverless-v2.html (дата звернення: 04.06.2025).

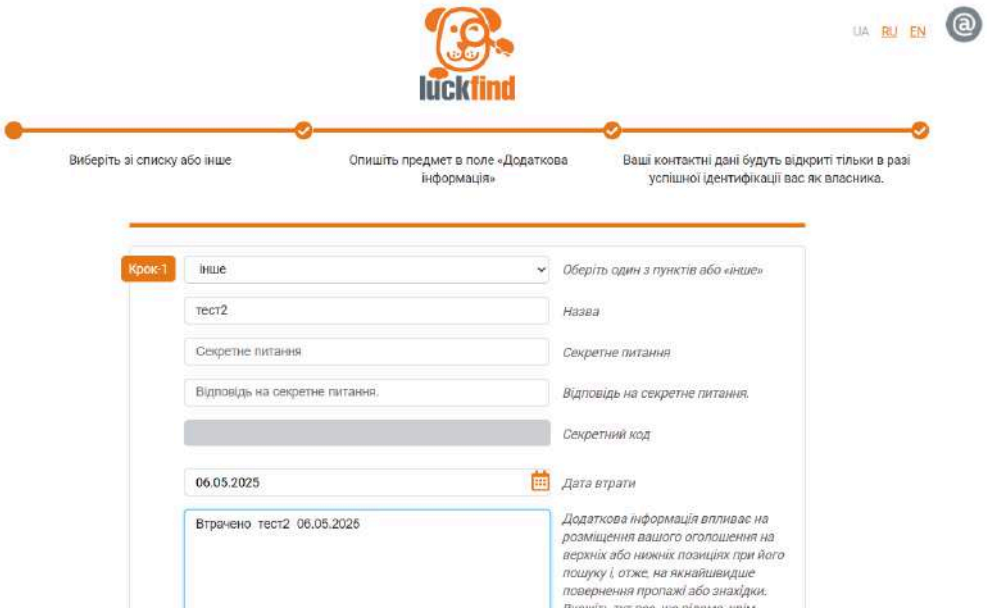
Додатки

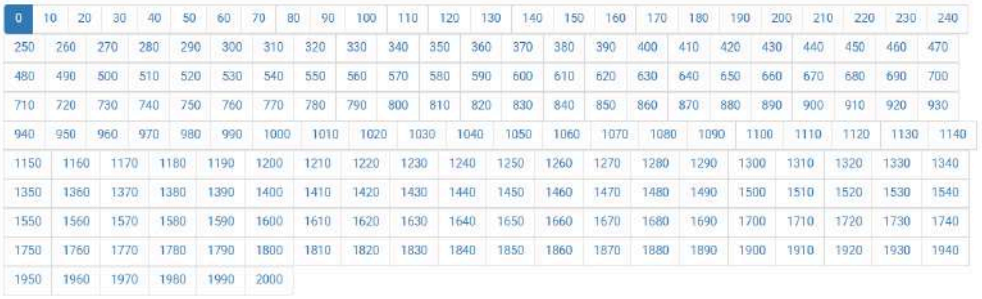
Додаток А

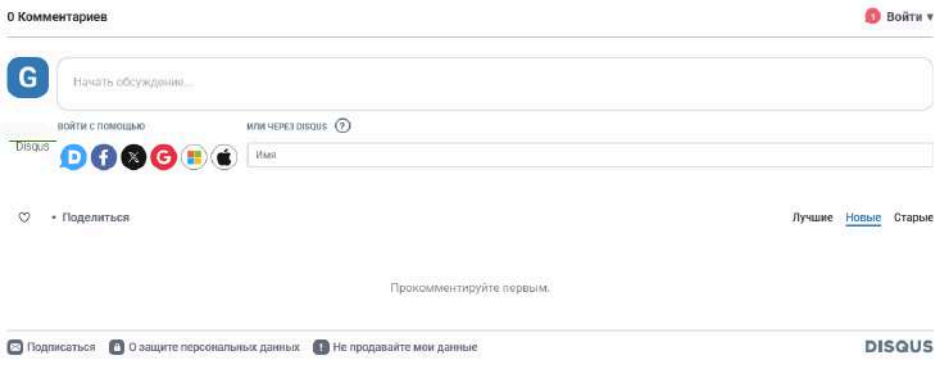
Сайти аналоги

1. Luckfind

Посилання: https://www.luckfind.me

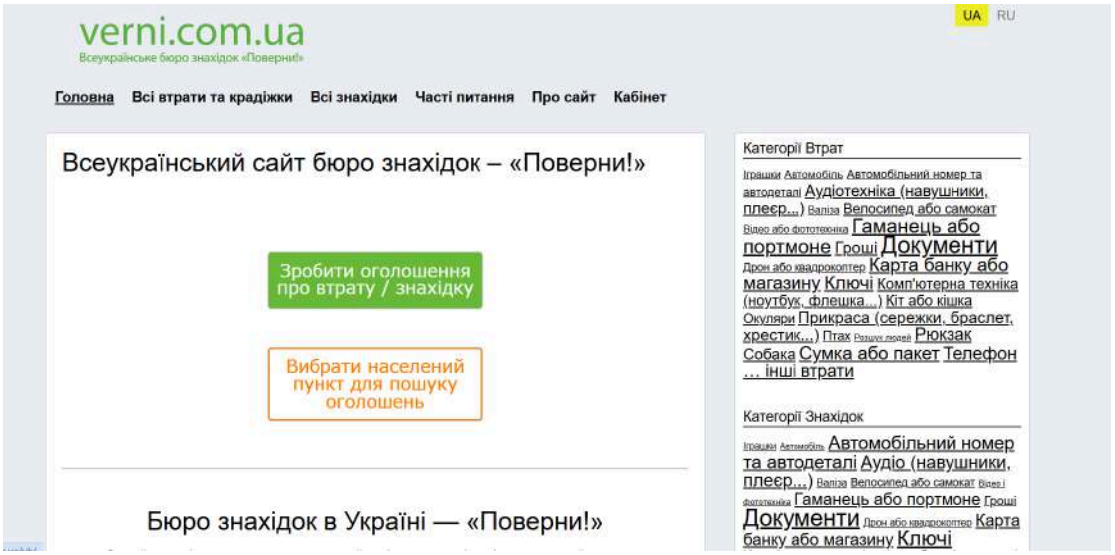






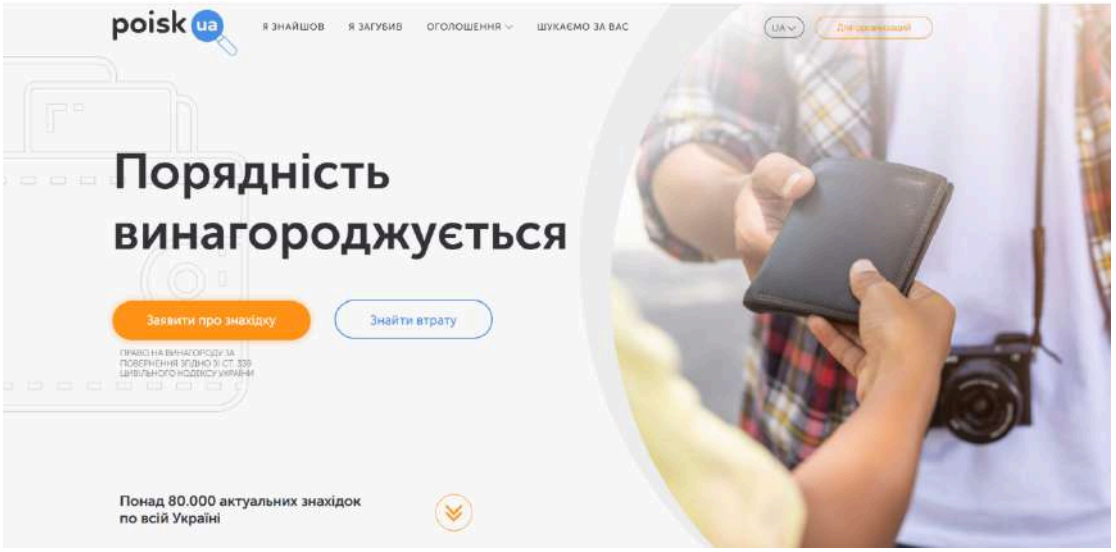
2. Verni

Посилання: https://verni.com.ua

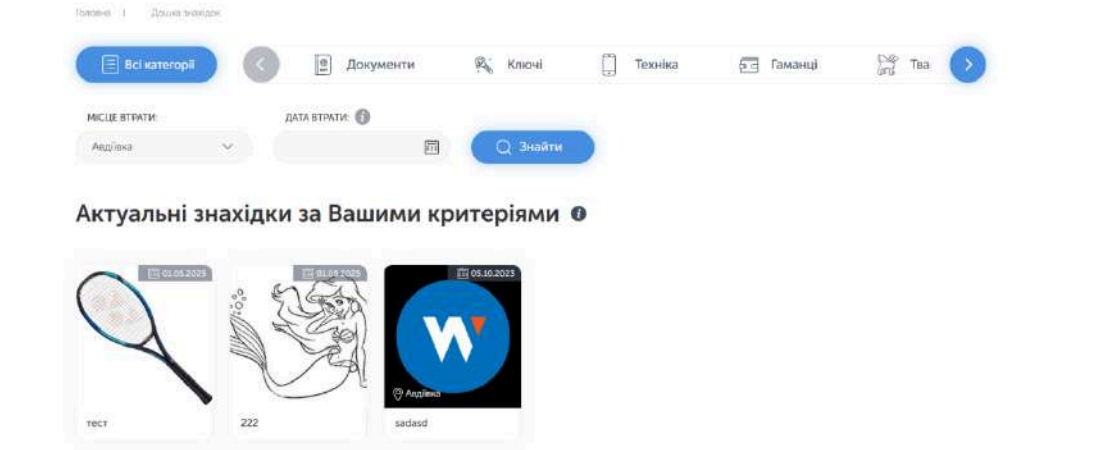


3. Poisk

Посилання: https://poisk.ua/


poisk ua
Я ЗНАЙШОВ
Я ЗАГУБИВ
ОГОЛОШЕННЯ
ШУКАЄМО ЗА ВАС
UA
Порядність винагороджується
Заявити про знахідку
Знайти втрату
ПРАВО НА ВИНАГОРОДУ ЗА ПОВЕРНЕННЯ ЗГІДНО ЗІ СТ. 339 ЦИВІЛЬНОГО КОДЕКСУ УКРАЇНИ
Понад 80.000 актуальних знахідок по всій Україні


НАЙБІЛЬША МЕРЕЖА ЗАГУБЛЕНИХ РЕЧЕЙ


Всі категорії
Документи
Ключі
Техніка
Гаманці
Тва
МІСЦЕ ВТРАТИ:
Авдіївка
ДАТА ВТРАТИ:
Знайти
Актуальні знахідки за Вашими критеріями
тест
222
sadasd