

ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І.І.МЕЧНИКОВА

(повне найменування вищого навчального закладу)

Факультет математики, фізики та інформаційних технологій

(повне найменування інституту, назва факультету (відділення))

Кафедра математичного забезпечення комп'ютерних систем

(повна назва кафедри (предметної, циклової комісії))

Кваліфікаційна робота

на здобуття рівня вищої освіти «магістр»

(рівень вищої освіти)

на тему Дослідження і розробка архітектур систем агрегації зображень

Research and development of the image aggregation systems architectures

Виконав: студент денної форми навчання

спеціальності 123 – Комп'ютерна інженерія.

(шифр і назва напрямку підготовки, спеціальності)

Освітня програма «Комп'ютерна інженерія»

(назва освітньої програми)

Колесник Олексій Олексійович

(прізвище, ім'я, по-батькові)

Керівник

к.ф.-м.н. Антоненко О. С.

(науковий ступінь, вчене звання, прізвище та ініціали, підпис)

Рецензент

к.т.н., доц. Пенко В.Г.

(науковий ступінь, вчене звання, прізвище та ініціали)

Рецензент

к.т.н., проф. Рувінська В.М.

(науковий ступінь, вчене звання, прізвище та ініціали)

Рекомендовано до захисту:

Протокол засідання кафедри

№ від « » 2023 р.

Завідувач кафедри

Євгеній МАЛАХОВ

(підпис)

(ім'я, прізвище)

Захищено на засіданні ЕК №

протокол № від « » 2023 р.

Оцінка / /

(за національною шкалою, шкалою ECTS, бали)

Голова ЕК

Алла КОБОЗЄВА

(підпис)

(ім'я, прізвище)

АНОТАЦІЯ

Мета роботи – проектування, розробка та аналіз архітектур систем для агрегації зображень. Розглядається два основних підходи – ETL та ELT. Також окрім двох класичних підходів пропонується третій – змішаний підхід, який поєднає в собі два класичних підходи. Такий підхід дозволяє побудувати універсальну систему, яка може обробляти як структуровані, так і не структуровані дані. Реалізація виконана з використанням мови програмування Python, та фреймворків PySpark, dbt.

Статичні дані – зображення – система отримує з декількох різнотипних джерел.

Результатом роботи є дві робочі системи агрегації зображень, CI/CD для розгортання. Частина елементів інфраструктури була розгорнута на власному сервері, частина – у хмарних сервісах.

Система за рахунок своєї гнучкої архітектури легко масштабується та має потенціал для розширення. Розширення може бути як за рахунок збільшення кількості джерел, так і за рахунок оптимізації кожного окремого елементу системи.

ABSTRACT

The goal of the work is the design, development, and analysis of systems for image aggregation. Two main approaches are considered – ETL and ELT. In addition to the two classical approaches, a third one is proposed – a mixed approach that combines the two classical methods. This approach allows for the construction of a universal system that can process both structured and unstructured data. The implementation is done using the Python programming language, PySpark, and dbt.

Static data – images – are obtained by the system from several sources. The result of the work is two operational image aggregation systems, CI/CD for deployment. Part of the infrastructure elements were deployed on a dedicated server, while another part was deployed in cloud services.

Due to its flexible architecture, the system easily scales and has the potential for expansion. Expansion can occur both by increasing the number of sources and by optimizing each individual element of the system.

ЗМІСТ

ВСТУП.....	5
1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ.....	7
1.1 ETL.....	7
1.2 ELT.....	8
1.3 Огляд існуючих рішень систем для агрегації даних	8
1.4 Вимоги до системи, що розробляється	9
2. ПРОЕКТУВАННЯ АРХІТЕКТУРИ.....	11
3. ПРОГРАМНА РЕАЛІЗАЦІЯ Extract ДОДАТКУ.....	16
4.ІНТЕГРАЦІЯ CI/CD У СИСТЕМУ.....	23
5. РОЗРОБКА Transform у ETL/ELT.....	27
5.1 Transform у ETL.....	27
5.2 Load у ELT	37
5.3 Transform у ELT.....	38
6. ПОКРАЩЕННЯ СИСТЕМИ.....	50
7. ПОРІВНЯЛЬНИЙ АНАЛІЗ РЕАЛІЗАЦІЙ АРХІТЕКТУР.....	56
ВИСНОВКИ.....	59
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	61
ДОДАТОК А КОД ETL.....	63
ДОДАТОК В КОД ELT PIPELINE.....	83
ДОДАТОК С YML ФАЙЛИ GITHUB ACTIONS.....	91

ВСТУП

Системи агрегації даних – це програмний комплекс, який об'єднує, збирає та обробляє дані з різних джерел. На сьогодні будь-яке веб-застосування, призначені для великої кількості користувачів, такі як YouTube, Instagram, Spotify, включає в себе такі системи. Їх використовують для аналітики, статистики та розробки моделей машинного навчання. Інформація в сучасному світі постійно зростає, тому системи, які класифікували б, агрегували та трансформували великий обсяг даних, є актуальними, як ніколи.

В сучасному бізнес-середовищі, де важливість точних аналітичних даних та оперативності прийняття рішень важко переоцінити, системи агрегації даних стають ключовим елементом успішної діяльності підприємства. У результаті роботи таких систем отримується великий агрегований комплекс інформації, яка використовується для аналітики та навіть прогнозування, що є надзвичайно важливим у сучасному конкурентному середовищі.

З розвитком технологій та зростанням популярності візуального контенту вони стали неодмінною складовою багатьох підприємств та інтернет-платформ. Це особливо актуально для соціальних мереж, стрімінгових сервісів, торгових платформ та багатьох інших сучасних додатків, які активно використовують зображення для залучення користувачів та покращення користувацького досвіду. Саме тому у роботі у якості інформації, яку слід обробити, виступають зображення.

Дві найбільш поширені архітектури систем агрегації даних базуються на різних технологіях та підходах до обробки даних. Перший підхід використовує обробку даних за допомогою різних фреймворків мов програмування, таких як Python, Java, Scala і т.д., наприклад, Apache Spark[1]. Цей підхід відомий як ETL[2]. Друга архітектура використовує обробку даних за допомогою аналітичних баз даних, таких як Snowflake, Redshift і т.д. Цей підхід відомий як ELT[3].

Обираючи між ETL та ELT, враховується специфіку завдань, обсяг даних, технічні можливості та вимоги до продуктивності. Обидва підходи мають свої переваги та обмеження, і вибір залежить від конкретних потреб та умов використання в конкретному проекті. Саме тому у даній роботі буде порівняно дві архітектури у контексті агрегації саме зображень та їх метаданих.

Мета роботи – проектування, розробка та аналіз архітектур систем для агрегації зображень. Об’єкт – методи агрегації даних (зображень). Предмет – процеси агрегації даних. Для виконання роботи необхідно вирішити наступні завдання:

- 1) провести аналіз предметної області створення архітектур систем агрегації зображень;
- 2) реалізувати архітектуру ETL для системи агрегації зображень;
- 3) реалізувати архітектуру ELT для системи агрегації зображень;
- 4) зробити порівняльний аналіз реалізацій архітектур ELT та ETL для задачі агрегації зображень, виділити їх переваги та недоліки для цієї задачі;
- 5) зробити висновки щодо того, яка з архітектур краще підходить для агрегації зображень.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 ETL

У рамках даної роботи було визначено, що об'єкти, які будуть агреговані у системі, будуть вивантажуватись з різноманітних зовнішніх джерел. З цього виникають такі питання: як саме вивантажувати об'єкти? Які дані про об'єкти зберігати? У якому форматі? Чи потрібно їх якось фільтрувати, та якщо потрібно, то як саме? Всі ці питання вирішуються в рамках технології ETL Pipeline.

Extract, Transform, Load (ETL) або Витяг, Перетворення та Завантаження — процес, який використовується в базах даних та, особливо, у сховищах даних та у засобах Business Intelligence для забезпечення їх роботи для підтримки прийняття рішень.

Він охоплює наступні етапи обробки даних:

- 1) виймання даних із зовнішніх джерел (етап Extract);
- 2) перетворення даних, для зберігання даних у відповідній структурі або форматі, з метою подальшого аналізу (етап Transform);
- 3) завантаження даних у кінцеву базу даних. Більш точно, це може бути вітрина даних або сховище даних (етап Load).

Поняття ETL може стосуватися процесу завантаження будь-якої бази даних. Оскільки виймання даних займає багато часу, то для скорочення загального часу обробки, поширеним є одночасна робота всіх трьох етапів ETL[4]. Поки дані виймаються, процес перетворення отримує інші дані і готує їх для завантаження, щоб уникнути очікування виконання попередніх етапів. Основні технології, які використовуються для реалізації архітектури ETL, включають мову програмування Python, різноманітні фреймворки для неї, такі як Apache Spark або Pandas[5], та розподілені файлові системи[6].

1.2 ELT

ELT (Extract, Load, Transform) – це архітектурний підхід до обробки даних, де основний акцент робиться на завантаженні сирової або необробленої інформації в систему для подальшої обробки та трансформації. Порівнюючи концептуальний рівень даного підходу із ETL, можна відзначити загальний етап Extract. Він дійсно ідентичний для обох технологій і є програмним забезпеченням, яке вивантажує дані з великої кількості джерел і зберігає їх у розподілену файловою системою. Відмінність полягає в тому, що після цього в архітектурі ELT дані зберігаються в аналітичній базі даних, і всі операції трансформації даних вже виконуються там. Основні технології, які використовуються для реалізації ELT – це мова програмування Python, розподілені файлові системи, аналітичні бази даних, такі як Snowflake, мова SQL та технологія dbt[7].

1.3 Огляд існуючих рішень систем для агрегації даних

Для реалізації системи для агрегації даних треба провести огляд схожої вже реалізованої системи. Найбільш схожа система з системою, яку потрібно реалізувати – це Facebook[8]. В Facebook була реалізована велика аналітична платформа, до якої також входять системи ETL для агрегації зображень. У Facebook в якості мов програмування використовуються Python, Scala, Java. Основним фреймворком для роботи з даними є Apache Spark. Раніше вони використовували створену ними самими аналітичну базу даних – Hive, але від неї відмовилися через складність системи. У системі використовувалася мова програмування HiveQL. Зараз використовується Spark як один з елементів платформи Hadoop. Платформа Hadoop – це набір фреймворків для обробки даних.

Ще одним елементом Hadoop, який використовується в Facebook, є HDFS – розподілена файлова система. Використання HDFS є основною

відмінністю системи, яка розробляється у цій роботі, від системи у Facebook. Система HDFS має більш глибоке налаштування, але при цьому важче підтримувати. Система S3, яка використовується у цій роботі, є одним із сервісів AWS, вона легка в налаштуванні та має прозоре API. Другою відмінністю цієї системи від системи Facebook є те, що більшість модулів для роботи з даними у їхній системі написані на мові програмування Scala. З одного боку, це зручно, оскільки більшість рішень екосистеми Hadoop написані на Scala, але, з іншого боку, мова Scala є складною мовою програмування з великим порогом входження, що збільшує загальну швидкість розробки. У цій системі використовується мова програмування Python. Вона має простий і зрозумілий синтаксис, а крім того, є повністю об'єктно-орієнтованою, що дозволяє створити більш масштабовану програму.

Щодо додатків, які реалізують підхід ETL, компанія Amazon використовує саме такий архітектурний підхід для свого однойменного магазину. Схоже, це також стосується багатьох відомих проєктів електронної комерції, таких як Walmart, Fozzy та інші. Це ефективно, оскільки вони в основному працюють зі статистикою, яку легко відобразити у вигляді аналітичних таблиць. Відмінністю системи Amazon від системи ETL, представленої в цій роботі, є те, що Amazon використовує Redshift – аналітичну базу даних, розроблену самою компанією. У представленій роботі використовується база даних Snowflake. Суттєвих відмінностей між двома аналітичними базами даних немає, вони відрізняються лише політикою монетизації. Redshift оплачується за зберігання даних, тоді як Snowflake – за обчислювальні потужності.

1.4 Вимоги до системи, що розробляється

Загальною функціональністю системи є:

- 1) збір статичних об'єктів (зображень) з багатьох різнотипних джерел;
- 2) збереження зображень та метаданих у DFS;

3) агрегація даних за допомогою python/sql.

Не функціональні вимоги до інфраструктури:

- 1) масштабованість – система повинна коректно працювати при будь якій кількості даних;
- 2) гнучкість – у разі зміни джерела, чи додавання іншого джерела, чи іншої очікуваної зміни, система повинна коректно відпрацьовувати.

2. ПРОЕКТУВАННЯ АРХІТЕКТУРИ

Для того, щоб приступити до проектування архітектури системи агрегації даних, треба сформулювати її складові частини. Перелік складових частин інфраструктури:

- 1) extract додаток;
- 2) розподілене файлове сховище dfs;
- 3) аналітична база даних;
- 4) transform додаток;
- 5) середовище для розгортання баз даних, бекенда, набору скриптів.

Для реалізації extract додатку використовується мова програмування Python, бібліотека click для консольного трігера додатку та технологія cron для запланованого запуску додатку.

Реалізація інтерфейсу для extract приложения может быть разной. Самыми распространенными из них являются веб-интерфейс на REST архитектуре или запуск их из консоли.

REST (Representational State Transfer) – це архітектурний стиль, який використовується для розробки веб-сервісів. Основні принципи REST були введені Роем Філдінгом у його докторській дисертації 2000 року і визначають підходи до взаємодії між розподіленими системами в Інтернеті.

Переваги REST архітектури у порівнянні з CLI:

- 1) REST використовує стандартні HTTP-методи (GET, POST, PUT, DELETE), що робить його універсальним для взаємодії з різними системами і платформами;
- 2) REST ідеально підходить для інтеграції з веб-службами та API;
- 3) REST використовує стандарти, такі як URI і HTTP, що сприяє стандартизації взаємодії між системами.

Недоліки REST архітектури у порівнянні з CLI:

- 1) REST-запити можуть призводити до збільшення трафіку мережі, особливо при великому обсязі даних;

- 2) request-response система погано працює у випадку великонавантажених систем;
- 3) REST використовує стандарти, такі як URI і HTTP, що сприяє стандартизації взаємодії між системами.

Оскільки у Extract частині є багато довгих за часом операцій (вивантаження та завантаження зображень) та робити саме веб-додаток у даному контексті не принципово була обрана саме CLI.

У якості розподіленого файлового сховища був обраний S3, через гарантію надійності, інтеграцію у Amazon та багаті налаштування приватності.

У якості фреймворка для трансформації даних була обрана Python варіація Spark – PySpark, тому що він має зрозумілу sdk, та зрозумілу логіку роботи для такої ситуації.

У якості аналітичної бази даних був обран Snowflake через простоту налаштування та найменшу ціну в порівнянні з конкурентами

У якості інструмента трансформації даних у ELT був обран dbt, тому що цей стандарт індустрії не має конкурентів. Dbt – це поєднання шаблонізатору Jinja2 та мови sql. Він є досить важким для розуміння, але з його допомогою можна дуже добре налаштувати процеси трансформції.

Розділення ETL на окремі додатки, які вивантажують та трансформують дані дозволяє нам досягти гнучкості у питаннях обробки даних. Кожен з цих додатків, згідно з концепцією такого розділення, працює окремо та ніяк не взаємодіє один з одним. Вони взаємодіють лише з даними, які залишає попередній.

Схему архітектури інфраструктури ETL знаходиться на рис. 2.1

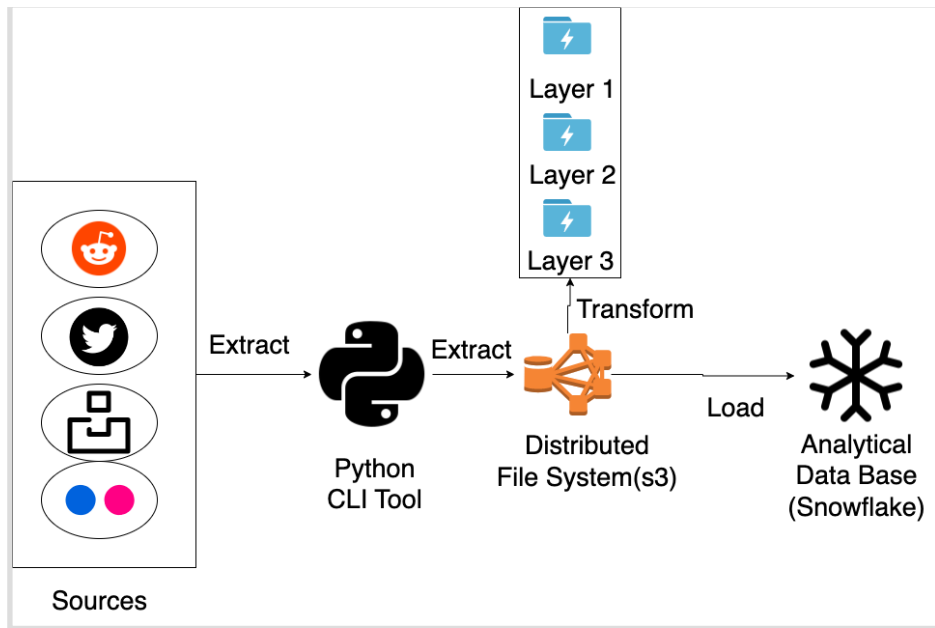


Рисунок 2.1 — Схема архітектури ETL

Схема архітектури ELT знаходиться на рис. 2.2

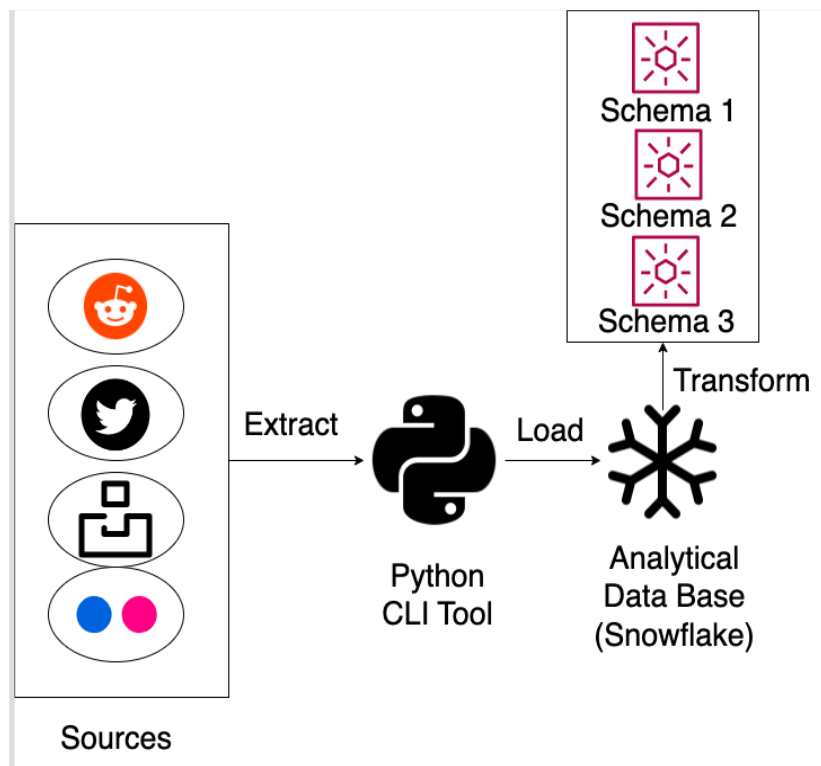


Рисунок 2.2 — Схема архітектури ELT

Для тегування та фільтрації зображень було використано один з сервісів AWS – AWS Recognition.

Теги до зображення відносяться до метаданих зображень. У системах обробляються такі метадані як:

- 1) заголовок зображення;
- 2) опис зображення;
- 3) ключові слова;
- 4) категорії;
- 5) дата публікації;
- 6) оцінки та відгуки;
- 7) розмір та роздільна здатність зображення;
- 8) автор зображення;
- 9) місцезнаходження зображення (геотеги);
- 10) інформація про колір зображення та інші фізичні метадані;
- 11) статистика переглядів.

Опис, оцінки, автор, геотеги та статистика переглядів – це опціональні метадані, це залежить від джерела, з якого беруться зображення. Опис, ключові слова та категорії отримуються завдяки сервісу AWS Recognition. Інформація про колір зображення отримується завдяки технології ExifTool. Також ця технологія дозволяє отримати такі фізичні дані про об'єкт зображення, як:

- 1) ImageSize – розмір зображення;
- 2) FileType – тип зображення jpeg/png;
- 3) BitsPerSample – вказує на кількість бітів, які використовуються для представлення інтенсивності кольору або яскравості кожного пікселя. Більше бітів дозволяють кодувати більше кольорових відтінків, що покращує якість зображення;
- 4) ICC – International Color Consortium. Ця структура містить інформацію про кольоровий профіль, який визначає простір кольору та інші характеристики;

5) `RenderingIntent` – це атрибут, що визначає, як система кольорового управління повинна взаємодіяти з кольоровими даними під час їхнього перетворення з одного кольорового простору в інший. Цей інтент визначає, як система кольорового управління повинна вирішувати потенційні розходження між кольорами в різних просторах кольору.

Окреме питання стосовно систем – це файли у яких треба тримати усі метадані. У таких системах існують два основних типу файлів – `.avro` та `.parquet`.

Формат файлу `.avro` використовується для зберігання даних у структурованому бінарному вигляді. Формат файлу `.parquet` – це колонковий формат зберігання даних, розроблений для ефективної обробки та зберігання табличних даних. Не зважаючи на те, що у сфері великих даних обидва формати є популярними (`.avro` наприклад використовується у стрімінгових технологіях, наприклад в `Apache Kafka`), був обраний саме `.parquet` файл через те, що він має колонковий формат даних.

3. ПРОГРАМНА РЕАЛІЗАЦІЯ Extract ДОДАТКУ

Бекенд додаток написано мовою програмування Python.

Python – це об'єктно-орієнтована, інтерпретована мова програмування високого рівня. Python має строгу динамічну типізацію.

Був обраний завдяки тому, що він є об'єктно-орієнтованим, має простий синтаксис та потужні бібліотеки.

Додаток представляє собою класичну реалізацію MVC архітектурою.

MVC – Це архітектурна схема, яка складається з трьох компонентів Model, View та Controller, що ефективно відокремлює Business Logic від користувальницького інтерфейсу програми.

Основні дані, якими оперує Extract – це зображення, та метадані.

Бекенд додаток складається з таких файлів:

- 1) requirements.txt – список завантажених пакетів для розгортання;
- 2) main.py – у контексті MVC це view, тут додаток отримує параметри;
- 3) Dockerfile – для розгортання додатку у докері;
- 4) config.py – файл для роботи зі змінними середовища;
- 5) .env – файл у якому саме прописуються усі змінні середовища;
- 6) unsplash_controller/unsplash.py – у контексті MVC – controller. Файл для роботи з unsplash api;
- 7) reddit_controller/reddit.py – у контексті MVC - controller. Файл для роботи з reddit api;
- 8) reddit_controller/reddit_client.py – файл з ініціалізацією reddit api, імплементація патерну Repository;
- 9) flickr_controller/flickr.py – у контексті MVC – controller. Файл для роботи з flickr api;
- 10) flickr_controller/flickr_client.py – файл з ініціалізацією flickr api, імплементація патерну Repository;

- 11) `image/models/image_model.py` – програмна модель зображень у вигляді класу. У контексті MVC – модель;
- 12) `image/amazon/storage/s3.py` – файл для роботи с розподіленим файловим середовищем s3;
- 13) `image/amazon/storage/client.py` – файл з ініціалізацією amazon sdk, імплементація патерну Repository.

У файлі `image_model.py` реалізується модель зображення, яке отримується з кожного джерела (див. лістинг 3.1).

```
class ImageModel:
    source: str
    sub_source: str # Subreddit in case reddit, channel in
case Telegram, etc
    url: str
    caption: str
    file_name: str

    def to_json(self):
        return {"source": self.source,
                "sub_source": self.sub_source,
                "url": self.url,
                "caption": self.caption,
                "file_name": self.file_name}

    @staticmethod
    def photo_prefix():
        return uuid.uuid4().hex

    @staticmethod
    def get_name(source, sub_source, photo_prefix):
        return f"{source}-{sub_source}-{photo_prefix}.png"
```

Лістинг 3.1 – Модель зображень

Контролери, що працюють з джерелами є у вигляді двох файлів – клієнта, де виконується ініціалізація sdk чи api, та контролера з логікою вивантаження зображення з джерела та завантаження до dfs. Ініціалізація з

sdk, використовуючи такі дані, як, наприклад, пароль, досить захищено, тому що усі дані знаходяться у змінних середовища (див. лістинг 3.2).

```
class RedditClient:
    def __init__(self):
        self.reddit = praw.Reddit(
            client_id=CLIENT_ID_REDDIT,
            client_secret=CLIENT_SECRET_REDDIT,
            user_agent=USER_AGENT_REDDIT,
            username=USERNAME_REDDIT,
            password=PASSWORD_REDDIT,
        )
```

Лістинг 3.2 – клієнт Reddit

Сам контроллер має композиційне відношення до класу S3. Кожен з таких класів має два метода – отримання та завантаження зображень до dfs (див. лістинг 3.3).

```
class RedditController:
    def __init__(self, reddit_model: RedditModel):
        self.reddit_client = RedditClient()
        self.reddit_model = reddit_model
        self.s3 = S3()

    def download_pictures(self) -> bool:
        list_of_images = self._get_list_of_pictures()
        self.s3.upload_memes(list_of_images)
        self.s3.upload_metadata(list_of_images)
        return True

    def _get_list_of_pictures(self) -> List[ImageModel]:
        result_list = []
        line = self.reddit_model.list_of_subreddits
        sub = line.strip()
        subreddit = self.reddit_client.reddit.subreddit(sub)
        for submission in subreddit.new(limit=self.reddit_model.count_of_images):
            if "jpg" in submission.url.lower() or "png" in
                submission.url.lower():
```

```

        file_name = ImageModel.get_name(source=REDDIT_SOURCE,
sub_source=line, photo_prefix=ImageModel.photo_prefix())
        result_list.append(
            ImageModel(
                url=submission.url.lower(),
                source=REDDIT_SOURCE,
                sub_source=line,
                caption=submission.title,
                file_name=file_name
            )
        )
    return result_list

```

Лістинг 3.3 – Код RedditController

Схожим чином працює клас `UnsplashController` – ініціалізація та два методи – для отримання та завантаження зображень у `dfs` (див. лістинг 3.4).

```

class UnsplashController:
    def __init__(self, unsplash_model: UnsplashModel):
        self.unsplash_model = unsplash_model
        self.s3 = S3()

    def download_pictures(self) -> bool:
        list_of_images = self._get_list_of_pictures()
        self.s3.upload_memes(list_of_images)
        self.s3.upload_metadata(list_of_images)
        return True

    def _get_list_of_pictures(self) -> List[ImageModel]:
        response = requests.get(UNSPLASH_URL, params={
            'query': self.unsplash_model.query,
            'page': 1,
            'per_page': self.unsplash_model.count_of_images,
            'client_id': UNSPLASH_ACCESS_KEY
        })

        data = json.loads(response.text)
        list_of_images = []
        for image in data['results']:
            image_url = image['urls']['raw']
            file_name = ImageModel.get_name(source=UNSPLASH_SOURCE,
sub_source=self.unsplash_model.query,

```

```

photo_prefix=ImageModel.photo_prefix())
    list_of_images.append(
        ImageModel(
            url=image_url,
            source=UNSPLASH_SOURCE,
            sub_source=self.unsplash_model.query,
            caption='mock',
            file_name=file_name
        )
    )
return list_of_images

```

Лістинг 3.4 – Код класу UnsplashController

Схожим чином працює й FlickrController – два класи – клас клієнт (див. лістинг 3.5) та клас контролер з тим ж функціоналом (див. лістинг 3.6).

```

class FlickrClient:
    def __init__(self):
        self.flickr = flickrapi.FlickrAPI(FLICKR_KEY, FLICKR_SECRET,
cache=True)

```

Лістинг 3.5 – Код flickr_client.py

```

class FlickrController:
    def __init__(self, flickr_model: FlickrModel):
        self.flickr_model = flickr_model
        self.flickr_client = FlickrClient()
        self.s3 = S3()

    def download_pictures(self) -> bool:
        list_of_images = self._get_list_of_pictures()
        print(f'here is list_of_images = {list_of_images}')
        self.s3.upload_memes(list_of_images)
        self.s3.upload_metadata(list_of_images)
        return True

    def _get_list_of_pictures(self) -> List[ImageModel]:
        photos =
self.flickr_client.flickr.walk(text=self.flickr_model.tag,
tags=self.flickr_model.tag, extras='url_c',
per_page=self.flickr_model.count_of_images, sort='relevance',

```

```

)

list_of_images = []
c = 0
for photo in photos:
    image_url = photo.get('url_c')
    file_name = ImageModel.get_name(source=FLICKR_SOURCE,
sub_source=self.flickr_model.tag,
photo_prefix=ImageModel.photo_prefix())
    list_of_images.append(
        ImageModel(
            url=image_url,
            source=FLICKR_SOURCE,
            sub_source=self.flickr_model.tag,
            caption='mock',
            file_name=file_name
        )
    )
    c += 1
    if c == self.flickr_model.count_of_images:
        break
return list_of_images

```

Лістинг 3.6 – Код flickr контролеру

Файл `main.py` приймає параметри та оркеструє усі контролери. Усі контролери приймають однакові параметри – кількість зображень, які треба завантажити та тег, по якому їх буде знайдено. Приклад роботи `main` на прикладі отримання команди `get_reddit`, що означає – завантажити зображення саме з редіту (див. лістинг 3.7).

```

@click.group()
def cli():
    pass

@cli.command(name="get_reddit")
@click.argument("count_of_image", required=True, type=click.INT)
@click.argument("subreddits", required=True, type=click.STRING)
def get_reddit(count_of_image: int, subreddits: str):
    reddit_model = RedditModel(
        count_of_images=count_of_image,
        list_of_subreddits=subreddits,
    )

```

```

reddit_controller = RedditController(reddit_model)
if reddit_controller.download_pictures():
    print("Everything is ok")

return True
session = boto3.Session(
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY
)

```

Лістинг 3.7 – Код view

```

CLIENT_ID=
CLIENT_SECRET=
USERNAME=
PASSWORD=
USER_AGENT=

AAA_ACCESS_KEY=
AAA_SECRET_ACCESS_KEY=

UNSPLASH_CLIENT_ID=

FLICKR_KEY=
FLICKR_SECRET=

```

Лістинг 3.8 – Структура .env файлу

Docker – це платформа для розробки, доставки та виконання програмного забезпечення у контейнерах. Контейнери – це легковагомі та портативні одиниці виконання, які включають усі необхідні компоненти для програми, включаючи код, середовище виконання, бібліотеки та залежності.

```

FROM python:3.8-slim
RUN useradd --create-home --shell /bin/bash app_user
WORKDIR /home/app_user
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
USER app_user
COPY . .
CMD ["bash"]

```

Лістинг 3.9 – Код у Dockerfile

4. ІНТЕГРАЦІЯ CI/CD У СИСТЕМУ

Розглядаючи інфраструктуру для Extract потрібно торкнутися двох тем – це сервер, де буде зберігатися додаток, і, безпосередньо, CI/CD.

CI (Continuous Integration) — безперервна інтеграція. Під час написання коду розробники постійно вносять зміни, що підвантажуються до репозиторію. Для автоматичного тестування та перевірки використовують спеціалізовані сервіси (наприклад, Github), що створюють скрипти.

CD (Continuous Delivery) — безперервна доставка. Відповідає за автоматичне розгортання збірки у будь-якому оточенні: продакшн, середовище тестування чи розробки. Наприклад, після редагування коду він автоматично вміщується в область тестування.

Платформа для розгортання програми – це власний сервер, на якому стоїть операційна система Linux, дистрибутив Ubuntu.

У якості CI/CD був обраний GitHub Actions.

Actions – це готові набори команд, якими описується порядок дій при спрацьовуванні певного тригера: комміта у гілку, створеного завдання, публікації нового релізу тощо. Також сценарії можна запускати за часом, як це робиться за допомогою утиліти cron.

Actions можуть взаємодіяти з кодом програми та історією його змін, з робочим оточенням репозиторію (завдання, проекти, документація, релізи), зі сторонніми сервісами для розгортання програми або надсилання якихось запитів із повідомленнями. Це потужний інструмент, який працюватиме з вами над проектом.

Вони були обрані через безкоштовність, багату документацію та пряму інтеграцію з GitHub, в якому зберігається проект.

Етапи CI/CD поділяються на два етапи – безперервної інтеграції та безперервної доставки. У GitHub Actions конфігураційні файли для CI/CD пишуться у декларативному стилі у yaml файлах. Виходить два yaml файли для безперервної доставки та безперервної інтеграції. Безперервна інтеграція

працює створення запиту на push у гілку master, тобто на pull request. Саме перед тим як злити гілку користувача та основну гілку повинні проходити тести та створюватися тестові складання програми, щоб протестувати всю програму з новим кодом. На рис 4.1 можна бачити, як виглядає запит на «merge». Користувач, який перевіряє код, який потрібно додати, праворуч.

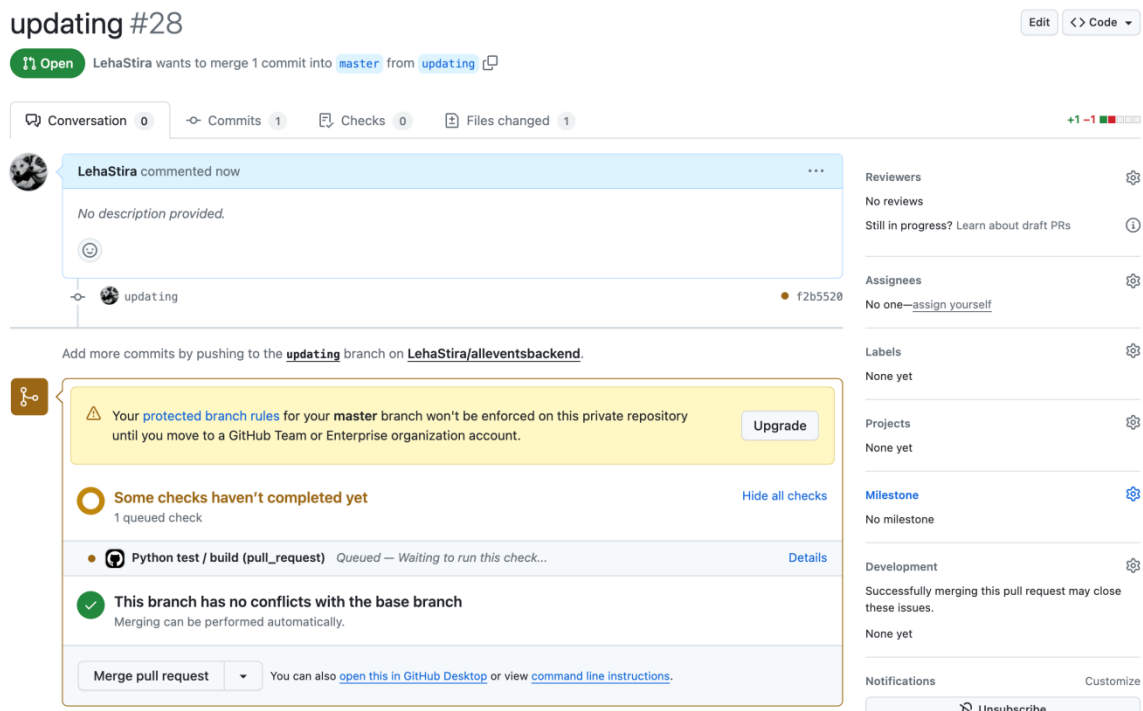


Рисунок 4.1 – PULL REQUEST

Знизу – перевірки, це і є CI частина. Більш детально її можна побачити на рис 4.2.

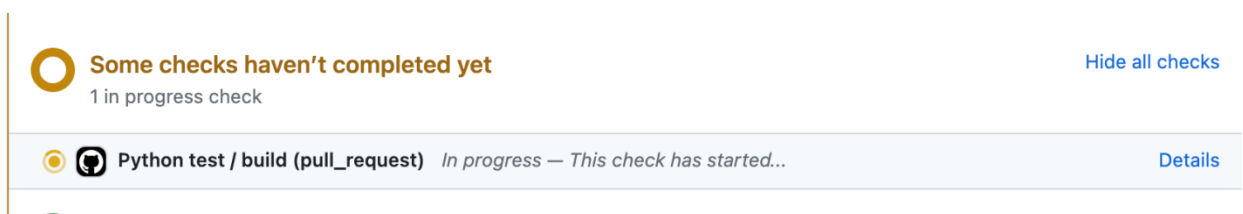


Рисунок 4.2 – Перевірки CI

У якості теста ми запускаємо проект на двох версіях Python, 3.8 і 3.9.

Після того, як перевірки пройдені успішно, можна зливати гілки, це видно на рис 4.3.

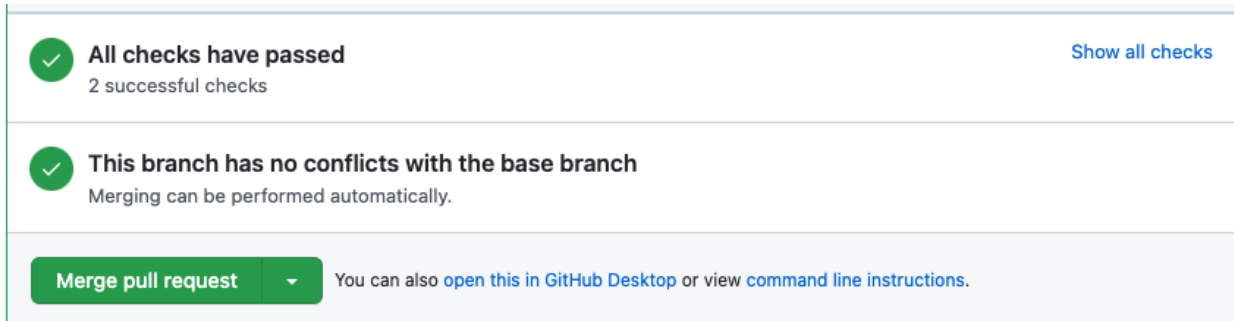


Рисунок 4.3 — Після успішного проходження СІ можна зливати

Після того як зіллється гілка програміста з основною гілкою, почнеться етап CD, який включає кілька кроків:

- 1) з'єднання з сервером по ssh;
- 2) преривання поточної версії додатку;
- 3) клонування оновленої версії додатку;
- 4) створення .env файлу;
- 5) створення усіх змінних середовища;
- 6) запуск докеру.

Весь процес CD відбувається автоматично після того, як новий код опинився у гілці майстер.

Процес виконання CD видно на рис. 4.4, рис.4.5, рис.4.6, рис.4.7, рис.4.8.

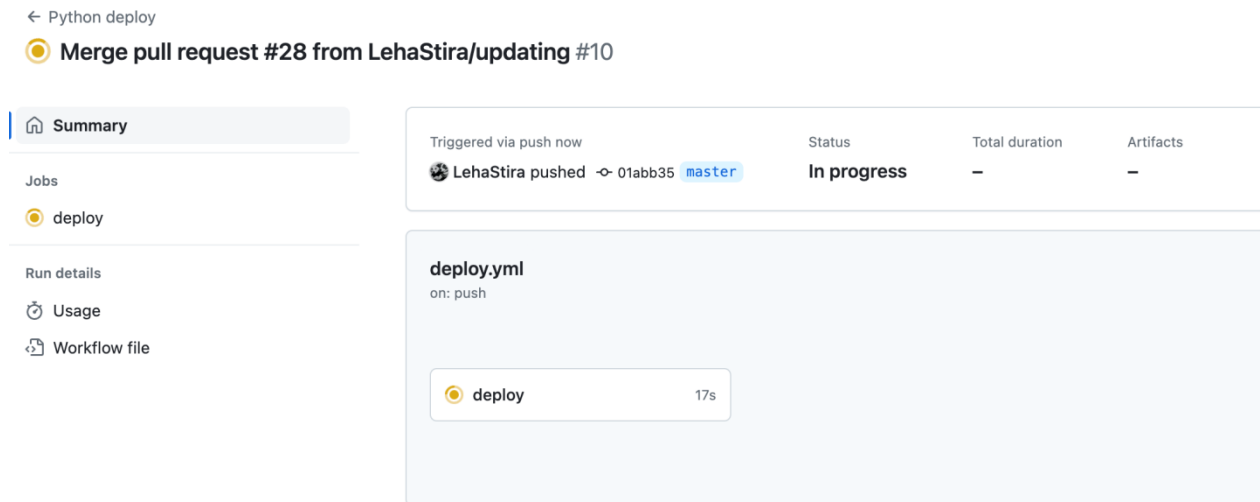


Рисунок 4.4 — Запущений процес CD

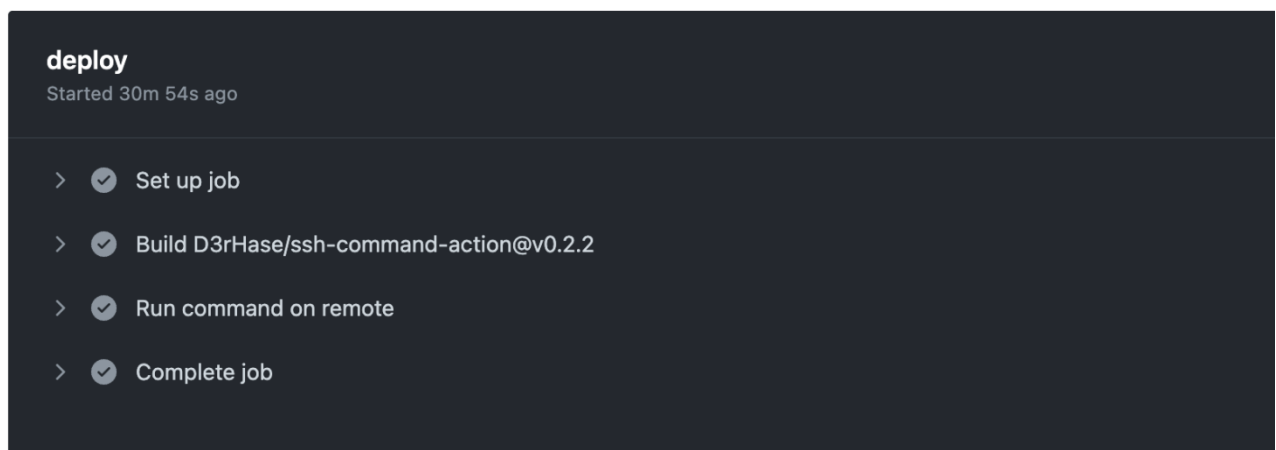


Рисунок 4.5 — Вдало завершена частина CD

Код усіх yml файлів вказано у Додатку С.

5. РОЗРОБКА Transform у ETL/ELT

5.1 Transform у ETL

Додаток transform складається з наступних файлів:

- 1) requirements.txt – список завантажених пакетів для розгортання;
- 2) transform/transform.py – головний файл, де описана основна логіка;
- 3) Dockerfile – для розгортання додатку у докері;
- 4) config.py – файл для роботи зі змінними середовища;
- 5) .env – файл у якому саме прописуються усі змінні середовища;
- 6) transform/aws/s3/extract_metadata.py – файл, який відповідає за усю логіку з AWS Recognition – тегування, фільтрація і тд.

У Додаток Transform був зтворений у функціональному стилі. В Transform-додатках виділяють кілька етапів обробки даних. У різній літературі ці логічні кроки можуть мати різні назви, а в даній роботі вони будуть позначатися:

- 1) bronze – це повністю сирі дані, щойно вивантажені з джерел;
- 2) silver – це дані, які пройшли очистку та фільтрацію, були видалені пошкоджені файли, файли без імені або інших важливих метаданих, файли, метадані яких не завантажилися через несправність і так далі;
- 3) gold – це дані із новими колонками, перетворені та агреговані згідно з технічною необхідністю.

Transform також є CLI-застосунком, який легко запускається за допомогою cron або bash-скрипта. У кожній пайтон програмі є своя точка входу (див. лістинг 5.1)

```
@click.group()
def cli():
    pass

@cli.command(name="transform")
def transform_data():
    s3 = boto3.client(
```

```

    "s3",
    aws_access_key_id=AWS_ACCESS_KEY,
    aws_secret_access_key=AWS_SECRET_KEY,
)
list_files = get_s3_files(s3)
print(list_files)
for file in list_files:
    process_file(file, s3)

def main():
    cli()

```

Лістинг 5.1 – Точка входу у Transform

Функція `transform_data` є основною, і тут видно всі трансформації, які відбуваються. Програма завантажує дані з S3 dfs, а потім трансформує кожен з них. До того, як файли з метаданими потрапляють на рівень трансформації, вони знаходяться у форматі `.json`. Результатом є метадані з великою кількістю полів/стовпців, які знаходяться у файлі `parquet`. Трансформація даних відбувається за допомогою технології датафрейму. Датафрейм (`DataFrame`) – це структура даних, яка представляє собою табличну форму організації даних у вигляді двовимірного масиву. У багатьох програмних бібліотеках, таких як `pandas` у Python або `Spark` у Scala/Java/Python, використовується термін "датафрейм" для опису такої структури. Вони мають табличну структуру, дуже гнучкі у роботі, мають підтримку різних типів даних. Взагалом вони є стандартом у агрегації даних. У коді вони позначаються змінною `df` (див. лістинг 5.2).

```

def process_file(filename: str, s3):
    df = get_json_from_s3(s3=s3, key=filename)

    print(df)
    df = get_tags_df(df)

    print(df)
    df = filter_df(df)
    df = drop_none_tag(df)

```

```
df = get_metadata_df(df)

print(df)
save_file(df)
```

Лістинг 5.2 – Функція process_file

Розглянемо функцію `get_json_from_s3`. Вона складається з двох логічних етапів. Перший – це завантаження файлів з `dfs` в оперативну пам'ять комп'ютера. Другий – це перетворення їх у датафрейм (див. лістинг 5.3).

```
def get_json_from_s3(s3, key):
    data = get_data_from_s3(s3=s3, key=key)
    print(f"data = {data}")
    string_data = StringIO(data)
    print(f"string_data = {string_data}")
    df = pd.read_json(string_data)
    return df
```

Лістинг 5.3 – Функція process_file

Перевагою датафреймів є те, що ця структура даних сумісна з великою кількістю інших типів даних, включаючи JSON. Код функції `get_data_from_s3`

```
def get_data_from_s3(s3, key):
    response = s3.get_object(Bucket=BUCKET_NAME_METADATA, Key=key)
    return response["Body"].read().decode("utf-8")
```

Лістинг 5.4 – Код функції get_data_from_s3

Далі відбувається фільтрація даних. Розглянемо конкретну логіку фільтрації. В результаті роботи `Extract` скрипта в одному бакеті `S3` зберігаються зображення, а в іншому – метадані. Припустимо, що файл з метаданими завантажився, але зображення – ні. Для цього є функція `check_s3_object_exists` (див. лістинг 5.5).

```

def check_s3_object_exists(bucket_name, key):
    s3 = boto3.client('s3')
    try:
        s3.head_object(Bucket=bucket_name, Key=key)
        return True
    except NoCredentialsError:
        print("Credentials not available")
        return False
    except Exception as e:
        print(e)
        return False

```

Лістинг 5.5 – Код функції check_s3_object_exists

Тепер нам потрібна функція, яка видалить з фрейму всі рядки, в яких немає цього зображення. Спочатку ми створюємо новий булевий стовпчик, який є True, якщо зображення є, і False, якщо немає. Після цього ми видаляємо ті рядки, де значення стовпчика є False. Після цього ми також видаляємо зайвий стовпчик. Ці операції з даними є витратними за ресурсами.

```

def filter_notexisting(df, bucket_name):
    df['exists_in_s3'] = df.apply(lambda row:
    check_s3_object_exists(bucket_name, row['file_name']), axis=1)
    df_filtered = df[df['exists_in_s3']] # Вибрати тільки рядки,
де exists_in_s3 = True
    df_filtered = df_filtered.drop(columns=['exists_in_s3'])
    # Видалити створений стовпець
    return df_filtered

```

Лістинг 5.6 – Код функції filter_notexisting

Тепер розглянемо ситуацію додавання нових стовпців. У межах програми додаються теги та стовпці з фізичною інформацією про статичний об'єкт зображення. Функція get_tags_df додає новий стовпець tags, додаючи туди застосовану до датафрейму функцію detected_labels, параметром якої є ім'я файлу.

```
def get_tags_df(df):
    df["tags"] = df["file_name"].apply(detect_labels)
    return df
```

Лістинг 5.7 – Код функції get_tags_df

Розглянемо функцію `detected_labels`. Вона використовує SDK сервісу AWS – Recognition. Amazon Rekognition – це сервіс від Amazon Web Services (AWS), який надає можливості розпізнавання та аналізу зображень і відео. Основні можливості Amazon Rekognition включають розпізнавання обличчя, фільтрацію невалідного контенту, ідентифікацію об'єктів, аналіз зображень та відео, розпізнавання тексту та виявлення емоцій.

Функція ініціалізує сервіс Recognition у SDK AWS. Вона відправляє за ім'ям файлу з бакета з зображеннями на `detected_labels`, отримує весь можливий відповідь і повертає результат у формі списку (Python list).

```
def detect_labels(photo: str) -> List or str:
    #try:
        result_list = []
        client = boto3.client(
            "rekognition",
            aws_access_key_id=AWS_ACCESS_KEY,
            aws_secret_access_key=AWS_SECRET_KEY,
            region_name='us-east-1'
        )
        response = client.detect_labels(
            Image={"S3Object": {"Bucket": BUCKET_NAME_IMAGE, "Name":
photo}}
        )
        print()
        print(response)
        labels = response.get("Labels")
        if not labels:
            return "NONE"

        for label in labels:
            result_list.append(label["Name"])
            print("Label: " + label["Name"])
            print("Confidence: " + str(label["Confidence"]))
```

```

print("Instances:")
instances = label.get("Instances")
if instances:

    for instance in instances:
        print(" Bounding box")
        print("           Top:           "           +
str(instance["BoundingBox"]["Top"]))
        print("           Left:          "           +
str(instance["BoundingBox"]["Left"]))
        print("           Width:         "           +
str(instance["BoundingBox"]["Width"]))
        print("           Height:        "           +
str(instance["BoundingBox"]["Height"]))
        print("           Confidence:     "           +
str(instance["Confidence"]))
        print()

print("Parents:")
parents = label.get("Parents")
#if not parents:
#    return "NONE"
if parents:
    for parent in parents:
        print(" " + parent["Name"])
        result_list.append(parent["Name"])
aliases = label.get("Aliases")
if aliases:
    for alias in label.get("Aliases"):
        print(" " + alias["Name"])
        result_list.append(alias["Name"])

    print("Categories:")
categories = label.get("Categories")
if categories:
    for category in categories:
        print(" " + category["Name"])
        result_list.append(category["Name"])
        print("-----")
        print()

if "ImageProperties" in str(response):
    print("Background:")
    print(response["ImageProperties"]["Background"])
    print()
    print("Foreground:")

```

```

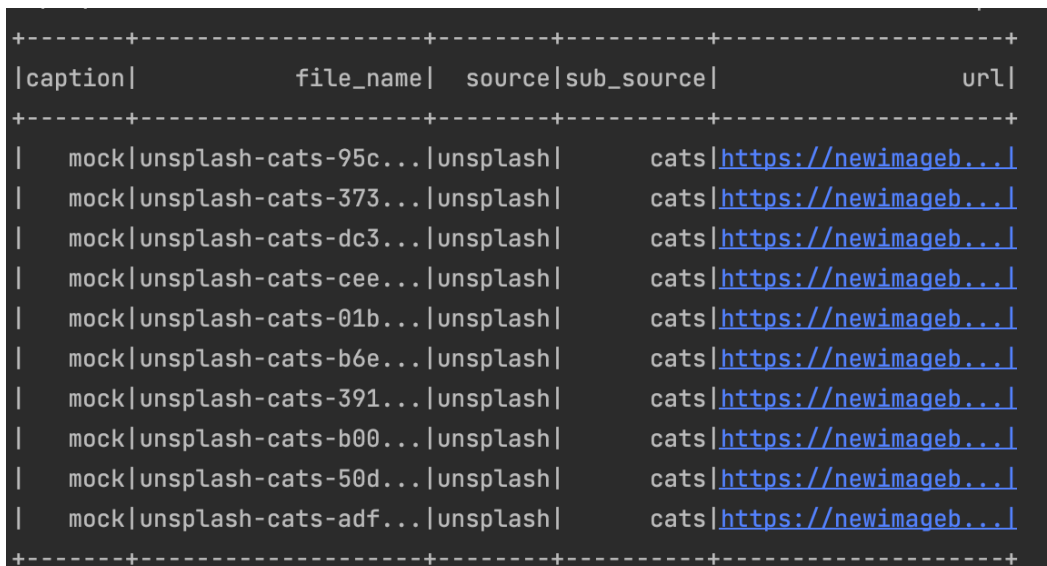
print(response["ImageProperties"]["Foreground"])
print()
print("Quality:")
print(response["ImageProperties"]["Quality"])
print()

# return len(response["Labels"])
return result_list

```

Лістинг 5.8 – Код функції detected_labels

Таким чином, у датафреймі отримуються теги. На рис. 5.1 видно датафрейм без тегів.



caption	file_name	source	sub_source	url
mock unsplash-cats-95c...	unsplash	cats		https://newimageb...
mock unsplash-cats-373...	unsplash	cats		https://newimageb...
mock unsplash-cats-dc3...	unsplash	cats		https://newimageb...
mock unsplash-cats-cee...	unsplash	cats		https://newimageb...
mock unsplash-cats-01b...	unsplash	cats		https://newimageb...
mock unsplash-cats-b6e...	unsplash	cats		https://newimageb...
mock unsplash-cats-391...	unsplash	cats		https://newimageb...
mock unsplash-cats-b00...	unsplash	cats		https://newimageb...
mock unsplash-cats-50d...	unsplash	cats		https://newimageb...
mock unsplash-cats-adf...	unsplash	cats		https://newimageb...

Рисунок 5.1 — Датафрейм без тегів

В той час, як на рис. 5.2 видно, що в датафреймі з'являється новий стовпець «tags».

```

+-----+-----+-----+-----+-----+-----+
|caption|      file_name|  source|sub_source|      url|      tags|
+-----+-----+-----+-----+-----+-----+
|  mock|unsplash-cats-95c...|unsplash|      cats|https://newimageb...|[Animal, Animals ...|
|  mock|unsplash-cats-373...|unsplash|      cats|https://newimageb...|[Blanket, Home an...|
|  mock|unsplash-cats-dc3...|unsplash|      cats|https://newimageb...|[Basket, Everyday...|
|  mock|unsplash-cats-cee...|unsplash|      cats|https://newimageb...|[Hardwood, Wood, ...|
|  mock|unsplash-cats-01b...|unsplash|      cats|https://newimageb...|[Abyssinian, Anim...|
|  mock|unsplash-cats-b6e...|unsplash|      cats|https://newimageb...|[Wood, Materials,...|
|  mock|unsplash-cats-391...|unsplash|      cats|https://newimageb...|      null|
|  mock|unsplash-cats-b00...|unsplash|      cats|https://newimageb...|[Animal, Animals ...|
|  mock|unsplash-cats-50d...|unsplash|      cats|https://newimageb...|[Animal, Animals ...|
|  mock|unsplash-cats-adf...|unsplash|      cats|https://newimageb...|[Animal, Animals ...|
+-----+-----+-----+-----+-----+-----+

```

Рисунок 5.2 — Датафрейм з тегами

Тепер розглянемо додавання метаданих за допомогою інструменту Exiftools. Додаються нові стовпці за тією ж логікою (див. лістинг 5.9).

```

def get_metadata_df(df):
    try:
        df[
            [
                "ImageSize",
                "FileType",
                "MIMEType",
                "BitsPerSample",
                "ProfileClass",
                "RenderingIntent",
                "CreateDate",
            ]
        ] = df["file_name"].apply(lambda x:
pd.Series(extract_metadata_s3(x))
        except Exception as err:
            print(err)
    return df

```

Лістинг 5.9 – Додавання метаданих з Exiftools

Давайте розглянемо функцію `extract_metadata_s3`, завдяки якій з'являються нові стовпці в датафреймі. Логіка тут така ж, як і з навішуванням тегів. Спочатку зображення вивантажується з S3, потім обробляється за допомогою технології Exiftool, і отриманий результат повертається функцією. Код функції:

```
s3 = boto3.client('s3',
                  aws_access_key_id=AWS_ACCESS_KEY,
                  aws_secret_access_key=AWS_SECRET_KEY,
                  )

# Create a temporary file to save the downloaded image
with tempfile.NamedTemporaryFile() as temp_file:
    print(f'here is photo_name = {photo_name}')
    print(f'here is temp_file.name = {temp_file.name}')
    # Download the image from S3 to the temporary file
    s3.download_file(BUCKET_NAME_IMAGE, photo_name,
                    temp_file.name)

    # Use pyexiftool to extract metadata from the downloaded
    image file
    with exiftool.ExifToolHelper() as et:
        metadata = et.get_metadata(temp_file.name)
        metadata = metadata[0]

result_data = {"ImageSize": metadata.get("Composite:ImageSize"),
               "FileType": metadata.get("File:FileType"),
               "MIMETYPE": metadata.get("File:MIMETYPE"),
               "BitsPerSample": metadata.get("File:BitsPerSample"),
               "ProfileClass": metadata.get("ICC_Profile:ProfileClass"),
               "RenderingIntent": metadata.get("ICC_Profile:RenderingIntent"),
               "CreateDate": metadata.get("XMP:CreateDate")\
               }
}
```

Лістинг 5.10 – Код функції `extract_metadata_s3`

Після цього дані оброблені і готові до збереження. Для цього їх необхідно зберегти на сервері у форматі файлу `.parquet`, після чого вивантажити. Є багато варіантів, куди саме вивантажувати кінцевий результат – зазвичай остаточні дані вивантажують або в аналітичну базу, або

знову в `s3 dfs`. Розглянемо функцію `save_file`. Вона зберігає файл у форматі `.parquet` (датафрейми дозволяють це зробити нативно), відправляє його на `S3` і видаляє з фізичного носія сервера.

```
def save_file(df):
    parquet_name = _write_parquet_file(df)
    send_s3_parquet(parquet_name)
    os.remove(parquet_name)
```

Лістинг 5.11 – Код функції `save_file`

Для видалення файлу використовується стандартний пакет Python `os`, який дозволяє взаємодіяти з операційною системою.

```
def _write_parquet_file(df) -> str:
    table = pa.Table.from_pandas(df)
    time_now = datetime.now().strftime("%Y-%m-%d-%H-%M-%S")

    parquet_name = f"data{time_now}.parquet"
    pq.write_table(table, parquet_name)
    return parquet_name
```

Лістинг 5.12 – Код експорту даних в `.parquet` на жорсткий диск

Далі дані відправляються в розподілену файлову систему за допомогою `SDK AWS`(див. лістинг 5.13).

```
def send_s3_parquet(parquet_name: str):
    s3 = boto3.resource("s3", aws_access_key_id=AWS_ACCESS_KEY,
                        aws_secret_access_key=AWS_SECRET_KEY)

    s3.Bucket(BUCKET_NAME_METADATA).upload_file(parquet_name,
        f"gold/{parquet_name}")
```

Лістинг 5.13 – Код завантаження даних до `DFS`

5.2 Load у ELT

Щоб працювати з файлами в аналітичній базі даних, їх необхідно туди завантажити. Для цього знову потрібен сценарій на Python.

Аналітичні бази даних дуже тривалий час виконують запит на вставку (insert), і тому тактика читання файлу та побудовування запитів на построчну вставку буде неефективною. Замість цього слід використовувати конструкцію COPY INTO, яка дозволяє вставляти файл в базу даних, де він буде представлений як таблиця. Скрипт починається ініціалізацією всіх змінних середовища, необхідних для взаємодії з aws та snowflake (див. лістинг 5.14)

```
snowflake_account = os.getenv("snowflake_account")
snowflake_user = os.getenv("snowflake_user")
snowflake_password = os.getenv("snowflake_password")
snowflake_warehouse = os.getenv("snowflake_warehouse")
snowflake_database = os.getenv("snowflake_database")
snowflake_schema = os.getenv("snowflake_schema")
snowflake_table = os.getenv("snowflake_table")

s3_bucket = os.getenv('s3_bucket')
s3_prefix = os.getenv("s3_prefix")
```

Лістинг 5.14 – Ініціалізація змінних середовища

Далі отримуємо файли з s3, на цей раз не за допомогою boto3, а за допомогою вбудованого aws cli. Це зроблено тому, що встановлювати великий пакет boto3 для такої невеликої задачі є занадто витратно (див. лістинг 5.15).

```
aws_s3_list_files_command = f"aws s3 ls
s3://{s3_bucket}/{s3_prefix} --recursive --no-sign-request --
region your_s3_region"
return os.popen(aws_s3_list_files_command).read().strip().split('\n')
```

Лістинг 5.15 – Код функції get_s3_files

Після цього ми встановлюємо з'єднання з Snowflake, встановлюємо курсор і копіюємо всі отримані файли з s3 (див. лістинг 5.16).

```
def save_to_db(s3_files):
    conn = snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    )

    cur = conn.cursor()

    for s3_file in s3_files:
        s3_file = s3_file.split()[-1]
        copy_into_command = f"COPY INTO '{snowflake_table}'
FROM s3://{s3_bucket}/{s3_file} " \
        f"FILE_FORMAT = (TYPE = 'JSON'
STRIP_OUTER_ARRAY = TRUE) PURGE = TRUE"
        cur.execute(copy_into_command)
        print(f"File {s3_file} loaded into Snowflake.")

    cur.close()
    conn.close()
```

Лістинг 5.16 – Код функції save_to_db

Дані було завантажено в таблицю метаданих. Тепер усе буде відбуватися в сполученні Snowflake + dbt.

5.3 Transform у ELT

Snowflake – це хмарна база даних, призначена для зберігання та обробки даних в хмарному середовищі. Основною перевагою Snowflake є те, що вона надає керовану, масштабовану та високопродуктивну аналітичну хмарну базу даних як сервіс (DBaaS). dbt (data build tool) – це інструмент для аналізу даних, який дозволяє аналітикам та науковцям здійснювати розвідку даних та

створювати звіти. dbt спрощує та автоматизує процеси підготовки та обробки даних, забезпечуючи ефективну роботу з базою даних.

dbt (data build tool) використовує ряд концепцій та термінів, які допомагають організувати та автоматизувати процес аналізу даних.

Модель – це SQL-файл, який містить логіку аналізу та трансформації даних. Моделі визначаються за допомогою SQL-запитів та організовані в директорії models в структурі проекту dbt[9]. Моделі можуть використовувати дані з інших моделей та виражати залежності між ними.

Макроси – це перевикористовувані блоки SQL-коду, які можуть бути викликані з інших моделей або макросів. Вони дозволяють абстрагувати логіку або фрагменти коду, що часто використовуються в різних місцях проекту dbt. Це полегшує управління та підтримку коду.

В dbt термін моделі використовується для позначення логічних або віртуальних представлень даних, які визначаються за допомогою SQL-запитів. Моделі можна розглядати як аналітичні view (віртуальна таблиця або набір даних, який визначається за допомогою SQL-запит.), оскільки вони представляють собою оброблені та трансформовані дані, готові для використання в аналітичних запитах та звітах.

Основна ідея моделей в dbt полягає в тому, щоб організувати SQL-код, який визначає аналітичний запит, в окремі файли, які називаються моделями. Кожна модель може використовувати дані з інших моделей та будувати логіку аналізу на основі джерел даних.

Напишемо першу модель, яка просто візьме всі дані з таблиці (див. лістинг 5.17).

```
-- models/metadata_model.sql

WITH metadata_cte AS (
  SELECT
    caption,
    file_name,
    source,
    sub_source,
```

```

        url
    FROM {{ref('metadata')}}
)

SELECT
    caption,
    file_name,
    source,
    sub_source,
    url
FROM metadata_cte;

```

Лістинг 5.17 – Код моделі metadata_model.sql

`{{ref('metadata')}}` – це dbt-функція, яка використовується для посилання на іншу dbt-модель або джерело даних в рамках вашого dbt проекту. Вона у двійних дужках, тому що це частина шаблонізатора Jinja2.

Шаблонізатор (Template Engine) – це інструмент або програмне забезпечення, яке використовується для генерації тексту або коду на основі певного шаблону та набору даних. Основна ідея полягає в тому, щоб розділити структуру шаблону від конкретних значень, які потрібно вставити в цей шаблон. Jinja2 – це шаблонізатор для мови програмування Python. Він дозволяє вбудовувати логіку Python в шаблони та генерувати тексти, код, HTML або інші файли з використанням даних та логіки, що передаються в шаблон (див. лістинг 5.18).

```

from jinja2 import Template

template_string = "Hello, {{ name }}!"
template = Template(template_string)

result = template.render(name="John")
print(result)

```

Лістинг 5.18 – Приклад роботи шаблонізатору

Таким чином при рендері цей механізм дозволяє нам передавати певні дані. Таким чином можна, наприклад реалізувати перехід даних у фронт-енд

в різних бекенд фреймворках. В такому випадку, приймаючи до уваги всі концепції, нам дуже легко видалити з таблиці ті колонки, у яких немає назви, створивши ще одну модель (див. лістинг 5.19).

```
-- models/filtered_metadata_model_name.sql

WITH filtered_data AS (
  SELECT
    *
    FROM {{ ref('metadata_model') }} -- Використовуємо dbt-
функцію ref для посилання на іншу модель
    WHERE file_name IS NOT NULL AND file_name != '' AND
file_name != 'NONE' -- Фільтруємо за умовою, що file_name не
порожнє і не NULL
)

SELECT
  *
FROM filtered_data;
```

Лістинг 5.19 – Код моделі фільтрації метаданих

Отже, ми отримуємо великі можливості для різних агрегацій та статистичних обчислень. Однак робота з зображеннями зазвичай включає в себе інші завдання, такі як накладання тегів або отримання фізичної інформації про зображення. Схожі завдання неможливо вирішити чистим SQL навіть з використанням dbt. Саме тому запропонована система агрегації даних у "класичному" вигляді ELT стає менш якісною порівняно з ETL. Дійсно, SQL ніколи не задумувався як мова програмування, завдяки якій можна обробляти зображення, аудіо чи інші статичні об'єкти. Таким чином, система втрачає великий функціональний обсяг.

SQL також не передбачає роботи зі сторонніми сервісами через API, він призначений для інших завдань. Отже, в чистому вигляді ELT уступає ETL в завданні агрегації зображень. Однак цю проблему можна вирішити іншим чином за допомогою макросів в dbt. Макроси можуть викликати будь-яке програмне забезпечення ззовні. Отже, можна покращити систему наступним

чином: додати скрипт на будь-якій мові програмування, який виконував би ті завдання, які не може виконати SQL в класичному вигляді, наприклад, відсилати зображення через API на сервіс AWS Rekognition, отримувати додаткові дані про фізичні характеристики зображень. З точки зору SQL-коду ці макроси будуть викликатися як функції в класичних імперативних мовах програмування.

Для цього потрібен скрипт на Python, який буде викликатися з макросу. Це рішення ускладнює читання всього проекту, і подібний проект стає важко підтримувати.

Отже, перевіряється наявність зображення на S3, і якщо його немає, рядок у таблиці відсівається. У папці macros створюємо файл check.py.

```
check.py
import boto3
from botocore.exceptions import NoCredentialsError

BUCKET_NAME_IMAGE = "newimagebuckettest"

def check_existing(image_key):
    s3 = boto3.client('s3')
    try:
        s3.head_object(Bucket=BUCKET_NAME_IMAGE,
Key=image_key)
        return True
    except NoCredentialsError:
        print("Credentials not available")
        return False
    except Exception as e:
        print(e)
        return False
    return True
```

Лістинг 5.20 – Код check.py

Після цього створюється макрос (див. лістинг 5.21)

```
-- macros/check_s3_object_exists.sql

{% macro check_s3_object_exists(image_key) %}
    {{ return(run_python_file("check.py", "check_existing",
[image_key])) }}
{% endmacro %}
```

Лістинг 5.21 – Код макросу check_s3_object_exists.sql

І вже після цього створюється нова модель (див. лістинг 5.22)

```
WITH filtered_data AS (
    SELECT
        *
    FROM {{ ref('filtered_metadata_model_name') }}
    WHERE {{ check_s3_object_exists('file_name') }}
)

SELECT
    *
FROM filtered_data;
```

Лістинг 5.22 – Код моделі filtered_data

Отже, отримуємо новий вигляд, який в Snowflake є таблицею, а в термінології dbt вважається моделлю.

Для порівняння проведемо інші операції, які ми виконали в ETL. Вирішення задачі тегування також вимагає викликів функцій Python. Слід зауважити, що існують альтернативні способи вирішення цієї задачі – в dbt існує велика кількість пакетів, включаючи ті, що створені іншими користувачами. Серед них є пакети, які дозволяють виконувати HTTP-запити прямо з мови SQL. Зазначимо, що ідея реалізації імперативного алгоритму у декларативному стилі призводить до погіршення читабельності всього проекту та збільшення часу, витраченого на розробку проекту (див. лістинг 5.23).

```
-- macros/get_image_tags.sql

{% macro get_image_tags(image_name) %}
  {{ run_python_script(
    script="macros/get_image_tags.py",
    image_name=image_name
  ) }}
{% endmacro %}
```

Лістинг 5.23 – Код макросу get_image_tags.sql

Макрос посилається на пайтон скрипт (див. лістинг 5.24).

```
import os

import boto3
from typing import List

from macros.check import BUCKET_NAME_IMAGE

AWS_ACCESS_KEY = os.getenv("AWS_ACCESS_KEY")
AWS_SECRET_KEY = os.getenv("AWS_SECRET_KEY")

def detect_labels(photo: str) -> List or str:
    #try:
        result_list = []
        # session = boto3.Session(profile_name='profile-name')
        # client = session.client('rekognition')
        client = boto3.client(
            "rekognition",
            aws_access_key_id=AWS_ACCESS_KEY,
            aws_secret_access_key=AWS_SECRET_KEY,
            region_name='us-east-1'
        )
        response = client.detect_labels(
            Image={"S3Object": {"Bucket": BUCKET_NAME_IMAGE,
"Name": photo}}
            # MaxLabels=10,
            # Uncomment to use image properties and filtration
settings
            # Features=["GENERAL_LABELS", "IMAGE_PROPERTIES"],
            # Settings={"GeneralLabels":
{"LabelInclusionFilters":["Cat"]},
```

```

# "ImageProperties": {"MaxDominantColors":10}}
)
print()
print(response)
labels = response.get("Labels")
if not labels:
    return "NONE"

for label in labels:
    result_list.append(label["Name"])
    print("Label: " + label["Name"])
    print("Confidence: " + str(label["Confidence"]))
    print("Instances:")
    instances = label.get("Instances")
    #if not instances:
    #    return "NONE"
    if instances:

        for instance in instances:
            print(" Bounding box")
            print("         Top:           " +
str(instance["BoundingBox"]["Top"]))
            print("         Left:           " +
str(instance["BoundingBox"]["Left"]))
            print("         Width:          " +
str(instance["BoundingBox"]["Width"]))
            print("         Height:          " +
str(instance["BoundingBox"]["Height"]))
            print("         Confidence:       " +
str(instance["Confidence"]))
            print()

        print("Parents:")
        parents = label.get("Parents")
        #if not parents:
        #    return "NONE"
        if parents:
            for parent in parents:
                print(" " + parent["Name"])
                result_list.append(parent["Name"])

    print("Aliases:")
    aliases = label.get("Aliases")
    if aliases:
        for alias in label.get("Aliases"):
            print(" " + alias["Name"])

```

```

        result_list.append(alias["Name"])

        print("Categories:")
        categories = label.get("Categories")
        #if not categories:
        #    return "NONE"
        if categories:
            for category in categories:
                print(" " + category["Name"])
                result_list.append(category["Name"])
                print("-----")
                print()

    if "ImageProperties" in str(response):
        print("Background:")
        print(response["ImageProperties"]["Background"])
        print()
        print("Foreground:")
        print(response["ImageProperties"]["Foreground"])
        print()
        print("Quality:")
        print(response["ImageProperties"]["Quality"])
        print()

    # return len(response["Labels"])
    return result_list

```

Лістинг 5.24 – Код файлу get_image_tags.py

Після цього створюється нова модель (див. лістинг 5.25).

```

WITH image_tags AS (
    SELECT
        *,
        {{ get_image_tags('caption') }} AS image_tags
    FROM {{ ref('second_filtered') }}
)
SELECT
    *
FROM image_tags;

```

Лістинг 5.25 – Код моделі image_tags

Отже, була створена нова колонка тегів у новій таблиці. Додамо також метадані щодо фізичних характеристик зображення за допомогою Python та бібліотеки `exiftool`. Спочатку створюється макрос (див. лістинг 5.26).

```
{% macro process_image_exif(image_name) %}
  {{ run_python_script(
    script="macros/process_image_exif.py",
    image_name=image_name
  ) }}
{% endmacro %}
```

Лістинг 5.26 – Код макросу `process_image_exif`

Після макросу створюється `.py` файл, який буде визиватися з макросу (див. лістинг 5.27).

```
import os

import boto3

from macros.check import BUCKET_NAME_IMAGE
import exiftool
import tempfile

AWS_ACCESS_KEY = os.getenv("AWS_ACCESS_KEY")
AWS_SECRET_KEY = os.getenv("AWS_SECRET_KEY")

def extract_metadata_s3(photo_name):
    s3 = boto3.client('s3',
                      aws_access_key_id=AWS_ACCESS_KEY,
                      aws_secret_access_key=AWS_SECRET_KEY,
                      )

    # Create a temporary file to save the downloaded image
    with tempfile.NamedTemporaryFile() as temp_file:
        print(f'here is photo_name = {photo_name}')
```

```

        print(f'here is temp_file.name = {temp_file.name}')
        # Download the image from S3 to the temporary file
        s3.download_file(BUCKET_NAME_IMAGE, photo_name,
temp_file.name)

        # Use pyexiftool to extract metadata from the
downloaded image file
        with exiftool.ExifToolHelper() as et:
            metadata = et.get_metadata(temp_file.name)
            metadata = metadata[0]
        result_data = {«ImageSize»:
metadata.get("Composite:ImageSize"),
                    "FileType":
metadata.get("File:FileType"),
                    "MIMEType":
metadata.get("File:MIMEType"),
                    "BitsPerSample":
metadata.get("File:BitsPerSample"),
                    "ProfileClass":
metadata.get("ICC_Profile:ProfileClass"),
                    "RenderingIntent":
metadata.get("ICC_Profile:RenderingIntent"),
                    "CreateDate":
metadata.get("XMP:CreateDate") \
                    }
        return result_data

```

Лістинг 5.27 – Код скрипту process_image_exif.py

Останнім кроком є реалізація моделі, яка буде використовувати макрос (див. лістинг 5.28).

```

WITH image_metadata AS (
    SELECT
        *, {{ process_image_exif('file_name') }} AS exif_metadata
    FROM {{ ref('metadata_with_tags') }}
)
SELECT
    *
FROM image_metadata;

```

Лістинг 5.28 – Код моделі image_metadata

Внаслідок, саме ця модель є повною копією файлу .parquet, який ми отримали в результаті обробки ETL.

Які висновки можна зробити щодо архітектури ELT у контексті завдання агрегації зображень? У чистому вигляді класична реалізація ELT, така як аналітична база даних Snowflake + dbt для подібних завдань, не має сенсу, оскільки така реалізація була б менш якісною порівняно з іншими архітектурами, наприклад, ETL. Вона стає менш якісною через втрату функціональності.

Отримання нових даних щодо зображень за допомогою сторонніх сервісів і різних інструментів є важливим кроком у агрегації даних, оскільки це дає більше уявлення про об'єкт. Ці дані можна використовувати в подальшому аналізі, при створенні моделей машинного навчання і т.д. Зробити це в чистому ELT без використання сторонніх мов програмування неможливо.

Однак, якщо систему поліпшити і додати макроси, які викликали б код на Python, який працював би з неструктурованими даними – зображеннями, то така функціональність в архітектурі ELT з'явиться. Проте подібна реалізація повна недоліків. Система втрачає свою швидкість, оскільки викликається інтерпретована повільна мова програмування Python. Система втрачає в читабельності, і, як наслідок, в швидкості розробки. Більше того, таким чином виходить, що дана система просто повністю повторює систему ETL, лише замість розподіленої файлової системи все відбувається в аналітичній базі даних. З цього можна зробити висновок, що архітектура ELT погано підходить для реалізації агрегації зображень.

6. ПОКРАЩЕННЯ СИСТЕМИ

Отже, існують два зручних інструменти для роботи з обробкою даних[10]. З переліку технологій, необхідних для реалізації кожного з них, впливають свої переваги та недоліки. Декларативний SQL важко підходить для обробки зображень та аналізу зображень. Під аналізом зображень тут розуміється випробування висновків не лише на основі метаданих, але саме на основі фізичного зображення. Наприклад, отримання тегів, оцінка глибини резкості зображення чи передачі кольору. Саме для таких маніпуляцій мова програмування Python та інші фреймворки ідеально підходять. Крім того, існує велика кількість бібліотек для аналізу зображень, від зазначеного у цій роботі Exiftool до OpenCV, Pillow та інших. Крім того, оскільки мова йде не про високонавантажену веб-додаток чи якусь іншу систему, де був би явний користувач, питання оптимізації не так актуальне. Можна сказати, що нам не дуже важливо, наскільки швидко обробиться та чи інша фотографія (звісно, це також залежить від постановки завдання), головне для нас — отримати певний результат, який ми, як користувачі системи, могли б побачити і взаємодіяти з ним. Наприклад, подібний обсяг даних дуже легко поділити на тестовий та навчальний і використовувати для навчання моделі машинного навчання. У цьому випадку важлива саме швидкість розробки, читабельність і можливість масштабування. З іншого боку, ELT, реалізоване за допомогою більш складних технологій, таких як dbt, добре працює з більш формалізованими даними. Зазвичай всі дані розділяють на структуровані та неструктуровані (structured and unstructured data). Структуровані дані – це зображення, аудіофайли, відеофайли і т.д. Неструктуровані дані, як правило, представляють собою дані у вигляді колонок, тексту, статистики. Логічно було б припустити, що неструктуровані дані зручніше обробляти за допомогою підходу ETL, а структуровані – за допомогою ELT.

Але структуровані дані в строгому сенсі не завжди означають різні статистичні та аналітичні завдання, з якими ELT впорається ідеально. Часто

виходить так, що навіть структуровані дані залежать від неструктурованих, і тоді виходить, що їх неможливо вирішити одним конкретним підходом. Особливо, якщо додати більше полів у метадані – статистику переглядів умовних «користувачів», наприклад, як у більшості соціальних мереж, або іншу інформацію, пов'язану з відгуками користувачів. У такому випадку найкраще вирішення – це змішаний підхід.

В розв'язанні поставленої задачі були використані окремі скрипти, які були розгорнуті на окремих серверах – віртуальних машинах. Це хороший і поширений підхід, основна перевага якого – це гнучкість. Проте також стають очевидними недоліки подібного підходу – відсутність певного "центрального" елемента, де, наприклад, можуть зберігатися всі конфігурації в одному місці. Основним недоліком використання підходу, описаного у роботі, є саме порушення принципу DRY – don't repeat yourself. В різних місцях було використано та ініціалізовано одні й ті ж речі. Крім того, саме такий "центральный" елемент стане необхідним в разі реалізації запропонованого вище "змішаного" підходу. Інакше система буде виглядати занадто змітано і втратить в читабельності та можливості масштабування.

Якщо використовувати змішаний підхід, то необхідний оркестратор. Оркестратор в контексті обробки даних – це програмний засіб або система, яка відповідає за керування та координацію виконання різних етапів або компонентів обробки даних. Це може включати в себе планування завдань, взаємодію з іншими службами та системами, відстеження прогресу та обробку помилок.

В контексті великих даних та обробки даних оркестратор може керувати виконанням різних операцій, таких як завантаження даних, їхню обробку, моделювання, агрегацію та виведення результатів. Оркестратор допомагає забезпечити, щоб всі етапи обробки даних працювали вірно та синхронізовано.

Деякі інструменти, такі як Apache Airflow, Luigi, Apache NiFi та інші, використовуються як оркестратори для створення та управління конвеєрами

обробки даних. Оркестратори є важливою частиною інфраструктури для ефективної та надійної обробки даних в розподілених системах.

Переваги використання оркестратора очевидні – ми зможемо зберігати всю необхідну конфігурацію в одному місці. Також варто відзначити, що можна зберігати всю необхідну конфігурацію проекту в одному місці і без застосування оркестратора. Достатньо додати до архітектури ще одну систему – Ansible. Ansible – це відкрите програмне забезпечення для автоматизації задач у сфері інформаційних технологій. Він надає інфраструктурний код та інструменти для конфігурації та управління системами, а також для оркестрації та автоматизації різноманітних процесів. Ansible використовує декларативний стиль опису потрібного стану системи, а не імперативний.

Таке рішення було б дорічним у випадку реалізації мікросервісної архітектури, але в даному випадку воно лише ще більше ускладнить систему, оскільки воно не буде центром всієї системи – а буде лише її частиною, відповідальною за зберігання конфігурації.

Тому в разі реалізації змішаного підходу ETL/ELT для обробки даних найбільш поширеним рішенням буде Apache Airflow. Apache Airflow – це відкрите програмне забезпечення для планування та оркестрації робочих процесів (workflows). Він дозволяє автоматизувати, планувати та моніторити складні послідовності робіт (задач), такі як великі обчислення даних, обробка даних, ETL-процеси, великомасштабні обчислення та інші завдання. На рис. 6.1- змішана архітектура з оркестратором.

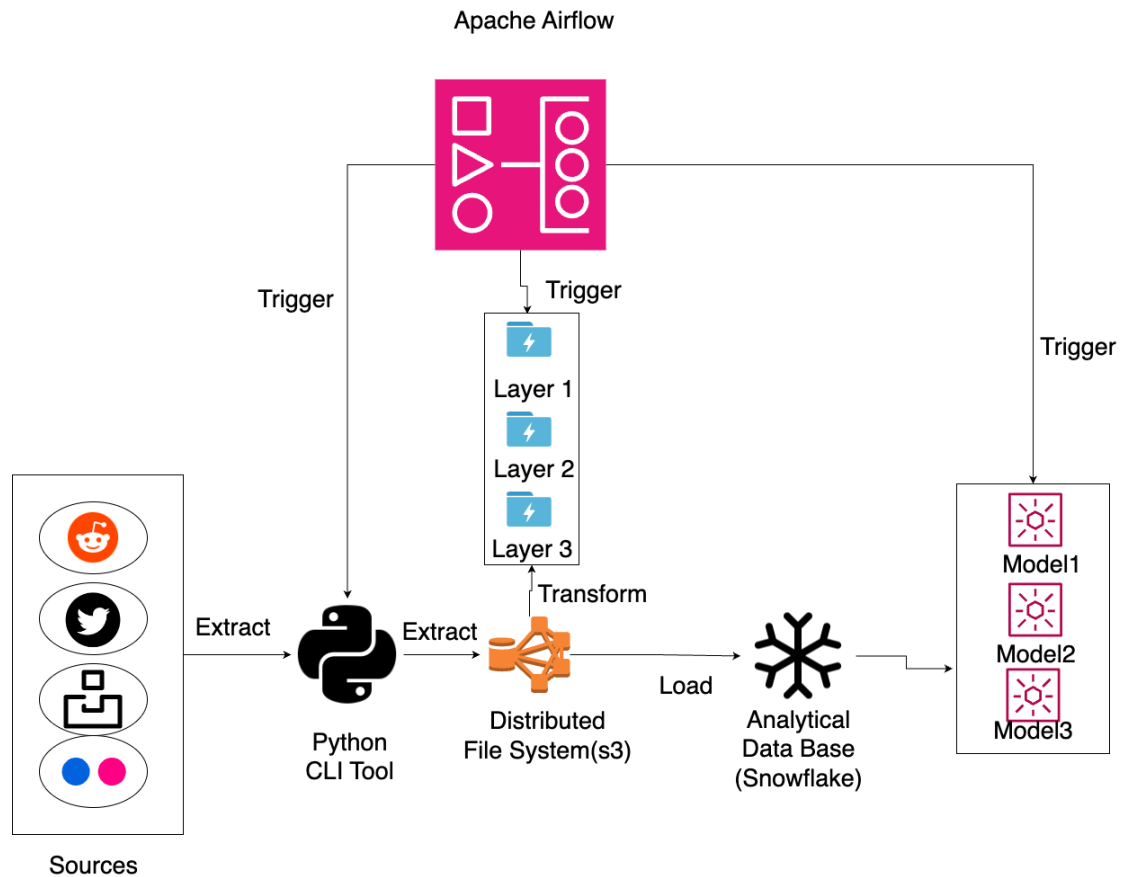


Рисунок 6.1 ETL + ELT архітектура

Apache Airflow використовує систему DAGs (directed acyclic graphs), тобто направлені ациклічні графи, де кожен вузол представляє окремий крок обробки даних.

Незважаючи на те, що цей фреймворк написаний на Python, його можна назвати, в певному сенсі, декларативним. Кожен DAG містить в собі певну кількість операторів, які є вузлами графа. В Apache Airflow термін "оператор" вказує на об'єкт, який виконує конкретну дію або обчислення у рамках DAG. Оператори визначаються та конфігуруються як частини DAG та відповідають за виконання певних завдань.

Серед найпоширеніших операторів можна виділити PythonOperator, який викликає Python-код, тобто виконує певну джобу. В контексті Apache Airflow та обробки даних "джоба" (job) використовується для позначення конкретної одиниці виконання задачі або операції у фреймворку. Коли DAG запускається, він генерує виконавчий процес, який виконує конкретний джоб.

DockerOperator тригерує джобу, використовуючи клонування Docker-імеджа з Docker Hub. У бібліотеці Apache Airflow існує багато операторів, і, більше того, система дозволяє реалізовувати власні оператори.

Враховуючи, що всі елементи архітектури ETL є CLI-додатками, у цій ситуації ідеально підходить BashOperator, який просто за допомогою команди Bash може запускати практично будь-яку систему.

Dag створюється як об'єкт у мові програмування Python (див. лістинг 6.1)

```
dag = DAG(
    'e_dag',
    description='Dag with ETL steps',
    schedule_interval=None,
    start_date=datetime(2023, 1, 1),
    catchup=False
)
```

Лістинг 6.1 – Приклад ініціалізації дага

Далі ми створюємо вузли – послідовно. Спочатку потрібно встановити всі бібліотеки, що відносяться до Extract, з requirements.txt. Потім – кілька разів тригерити Extract етап. Потім – встановити бібліотеки, що відносяться до Transform застосунку, також з requirements.txt. Потім – тригерити Transform застосунок. А потім розставити їх послідовно (див. лістинг 6.2)

```
install_deps = BashOperator(
    task_id='install_dependencies',
    bash_command=f'pip install -r
{dag_directory}/extract_test_cli/requirements.txt',
    dag=dag
)

# Определение второго оператора PythonOperator
run_reddit = BashOperator(
    task_id='run_reddit',
    bash_command=f"python3
{dag_directory}/extract_test_cli/main.py get_reddit 10 cats ",
    dag=dag
)
```

```

run_flickr = BashOperator(
    task_id='run_flickr',
    bash_command=f"python3
{dag_directory}/extract_test_cli/main.py get_flickr 10 dogs ",
    dag=dag
)

run_unsplash = BashOperator(
    task_id='run_unsplash',
    bash_command=f"python3
{dag_directory}/extract_test_cli/main.py get_unsplash 10 ducks ",
    dag=dag
)

install_second_deps = BashOperator(
    task_id='install_second_deps',
    bash_command=f'pip          install          -r
{dag_directory}/transform/requirements.txt',
    dag=dag
)

run_transform = BashOperator(
    task_id='run_transform',
    bash_command=f"python3
{dag_directory}/transform/transform.py transform ",
    dag=dag
)

# Установка зависимостей между задачами
run_reddit    >>    run_flickr    >>    run_unsplash    >>
install_second_deps >> run_transform

```

Лістинг 6.2 – Декларативний виклик ETL елементів

7. ПОРІВНЯЛЬНИЙ АНАЛІЗ РЕАЛІЗАЦІЙ АРХІТЕКТУР

Оцінка результатів архітектур буде здійснюватися за декількома метриками. Оцінці підлягатимуть наступні архітектури: ETL, чистий ELT без макросів, ELT з макросами та змішана система. На таблиці 7.1 можна побачити порівняння функціональності систем (Functional Comparison).

Таблиця 7.1 Functional Comparison

Functionality	ETL	ELT Clean	ELT + Python	Mixed
Extract data	True	True	True	True
Filtering data	True	True	True	True
Attribute Addition	True	False	True	True
Saving results	True	True	True	True

Також важливою метрикою є час роботи систем. У цій роботі час роботи кожної системи представлений для 1, 10 і 100 гігабайт зображень. Час вимірюється у секундах. Порівняльна характеристика за часом виконання наведена в таблиці 7.2.

Таблиця 7.2 Порівняння часу виконання

	1 gb	10 gb	100 gb
ETL	223	1933	15785
ELT Clean	111	250	454
ELT + Python	200	1850	16487
Mixed	180	1993	15532

Візуалізація отриманих результатів на рисунку 7.1.

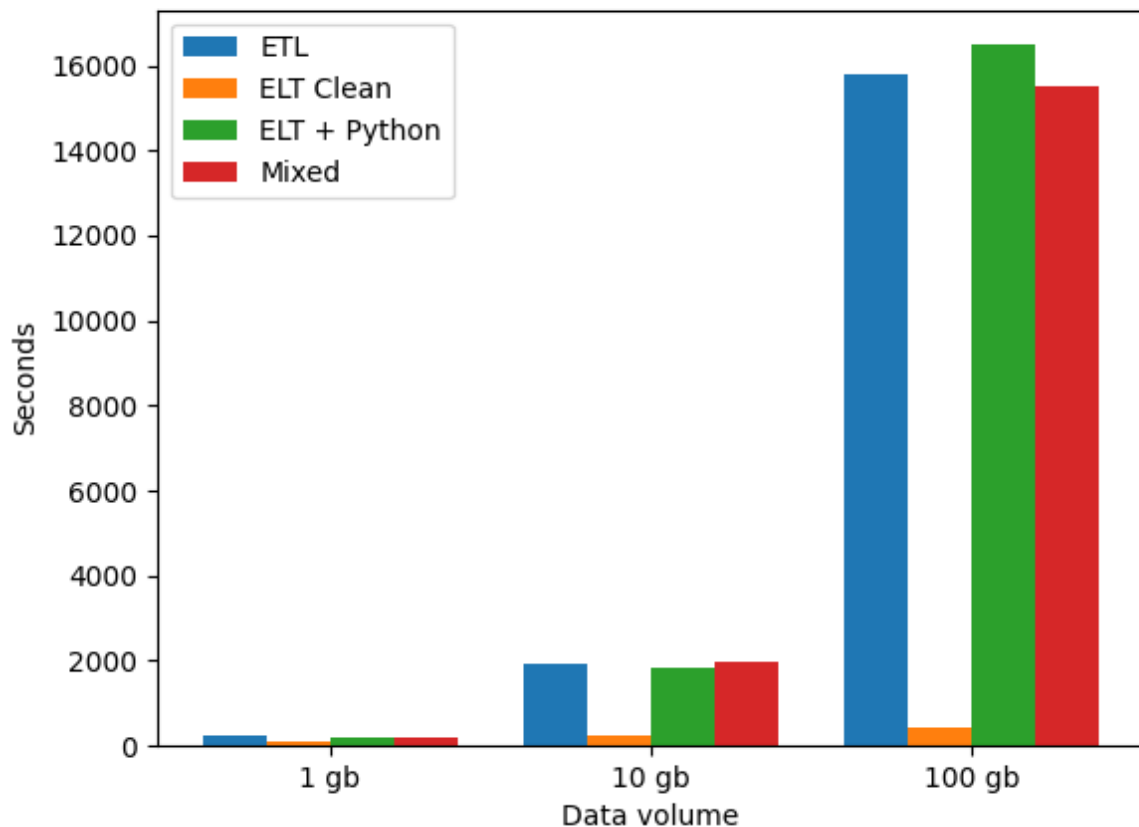


Рисунок 7.1 Графік порівняння часу виконання

З отриманих результатів видно, що архітектура ELT вимагає менше часу для виконання. Більша частина часу виконання в ELT витрачається на етап Extract, який є однаковим для всіх розглянутих архітектур у даній роботі. Крім того, швидкість роботи зменшується через особливості компіляції dbt. Окрім цього, в чистому ELT відсутня частина функціональності, така як додавання нових атрибутів. Це робить систему, побудовану на архітектурі ELT, менш якісною.

Найбільше часу виконання на 100 гб даних в ELT + Python зумовлено повільною швидкістю роботи мови програмування Python. Взагалі враховуючи архітектури системи, підхід ETL та підхід ELT + Python з точки зору структури практично однакові. Обидва на етапі Transform поділяються на два рівні – рівень роботи з даними в таблиці (або датафрейми) та рівень роботи з

зовнішніми API чи бібліотеками. Просто результат роботи рівня, який працює з зовнішніми API в системі ELT + Python, зберігається в таблицях і використовується при виконанні sql-запиту, а в системі ETL він зберігається в датафреймах. Підхід ELT + Python є ненатуральним для такої архітектури, оскільки виконує завдання, по суті, непридатні для неї. Такий підхід втрачає в читабельності, важко налаштовується та конфігурується (за рахунок виклику python-коду в макросах), і при цьому він виконується стільки ж часу, скільки і ETL. Виходячи з цього можна зробити висновок, що підхід ELT, незважаючи на кращу продуктивність, гірший для обробки неструктурних даних, таких як зображення.

Змішана архітектура, з іншого боку, займає приблизно стільки ж часу, скільки і ETL, оскільки ETL є частиною змішаної архітектури. Змішана архітектура, на відміну від ELT, містить в собі весь функціонал системи, оскільки там виконуються два паралельні процеси – ELT швидко обробляє структуровані метадані зображень, ETL обробляє неструктуровані дані – самі зображення. Змішана архітектура складна в реалізації через конфігурацію центрального елемента – Apache Airflow, але при цьому містить в собі весь функціонал і використовує всі технології за призначенням.

Результат роботи – це структуровані, відфільтровані дані, які можна використовувати для аналітики чи для тренування моделей машинного навчання. Отримані моделі машинного навчання можна також додати до цієї системи, розгорнувши їх у іншому сервісі AWS – AWS Sagemaker, або на власному сервері[11].

Запропоновану систему можна вдосконалити шляхом додавання більше джерел, більше оброблюваних метаданих та оптимізації коду на Python за допомогою впровадження багатопотокових обчислень.

ВИСНОВКИ

В рамках дипломної роботи були виконані наступні завдання:

- 1) був проведений аналіз предметної області створення архітектур агрегації зображень;
- 2) реалізована архітектура ETL для агрегації зображень;
- 3) реалізована архітектура ELT для агрегації зображень;
- 4) був зроблений порівняльний аналіз реалізацій архітектур ELT та ETL для задачі агрегації зображень, виділені їх переваги та недоліки для цієї задачі;
- 5) були зроблені висновки та запропонована покращена – змішана система з комбінацією підходів ELT та ETL.

У рамках роботи були реалізовані системи ETL і ELT для завдання агрегації зображень та їх метаданих. Для конкретної логіки аналізу метаданих більше підходить архітектура ETL, зокрема враховуючи той факт, що у структурі даних можна використовувати SQL-код для трансформації, що є ще одним аргументом на користь цієї структури даних. ELT-підхід варто використовувати там, де більше числових даних, в ситуаціях, де їх потрібно якось змінити для аналітики. Статистика – ідеальна область застосування ELT-підходу. Він більш складний та важкий у розробці, у таких інструментах, як dbt, великий поріг входу, і часто можуть виникати труднощі з конфігурацією. Тим не менш, якщо є задача статистично щось розрахувати на великому об'ємі даних, то ELT ідеально підходить. Спроби використовувати його для аналізу зображень все одно призводять до використання Python, але в якості макросів до SQL-запиту. Така реалізація виглядає неконсистентно, і взагалі вирішення подібних завдань за допомогою SQL виглядає неорганічно.

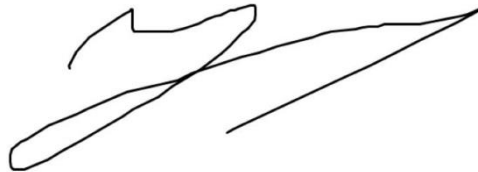
ETL ідеально підходить для неструктурованих даних, таких як зображення, аудіо та відеофайли. Така система є більш простою у розробці, конфігурації та розгортанні на сервері, завдяки таким технологіям, як Docker.

У випадку взаємодії виключно з неструктурованим типом даних і даними, які описують ці дані, використання чогось іншого, крім підходу ETL, є зайвим.

З точки зору ресурсів та інфраструктури, вони є однаковими. Навіть при тому, що для розгортання ETL потрібен власний налаштований сервер, а ELT виконується переважно в пропрієтарному хмарному середовищі аналітичної бази даних, яку використовують. dbt має велику кількість конфігураційних файлів, тому його налаштування також є складним.

У разі, якщо дані більші, ніж у представленій роботі, і якщо потрібна більш глибока статистика, логічно використовувати змішаний підхід – ELT для статистики, ETL – для зображень, керуючи усіма цими процесами за допомогою Apache Airflow.

За результатом роботи опубліковано тези на всеукраїнській конференції студентів і молодих науковців[12], а також тези доповіді міжнародної наукової конференції [11].

A handwritten signature in black ink, consisting of several overlapping loops and a long horizontal stroke extending to the right.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What is Apache Spark? // Spark – Режим доступу: <https://spark.apache.org/>
2. What is ETL Pipeline? // Snowflake – Режим доступу: <https://www.snowflake.com/guides/etl-pipeline>
3. What is ETL Pipeline? // Stitchdata – Режим доступу: <https://ua.stitchdata.com/resources/what-is-elt/>
4. Comparative Analysis of ETL Tools in Big Data Analytics // Muhammad Umer Farooq, Nazia Abrar, Syed Muhammad Nabeel Mustafa, Asma Qaiser – Режим доступу: https://www.researchgate.net/publication/369094822_Comparative_Analysis_of_ETL_Tools_in_Big_Data_Analytics
5. What is Pandas? // Nvidia – Режим доступу: <https://www.nvidia.com/en-us/glossary/data-science/pandas-python/>
6. A Comparative Study on Distributed File Systems // Suman De, Megha Panjwani – Режим доступу: https://www.researchgate.net/publication/351116448_A_Comparative_Study_on_Distributed_File_Systems
7. Understanding DBT (Data Build Tool) // Shashank Mishra – Режим доступу: <https://m.mage.ai/understanding-dbt-data-build-tool-an-introduction-751112595dc7>
8. Apache Spark @Scale: A 60 TB+ production use case // Meta – Режим доступу: <https://engineering.fb.com/2016/08/31/core-infra/apache-spark-scale-a-60-tb-production-use-case/>
9. dbt in Production / Cameron Cyr, Dustin Dorsey // Unlocking dbt. September 2023 – С. 321-149
10. A Comparative Study between ETL (Extract-Transform-Load) and E-LT (ExtractLoad-Transform) approach for loading data into a Data Warehouse // Vikas Ranjan – Режим доступу: <http://www.ecst.csuchico.edu/~bjuliano/csci693/Presentations/2009w/Materials/Ranjan/Ranjan.pdf>
11. Koliesnyk O. O. Comparison of technologies for full cycle development of machine learning models / O. O. Koliesnyk, O. S. Antonenko // Нейромережні технології та їх застосування НМТЗ-2023: збірник наукових праць XXII Міжнародної наукової конференції «Нейромережні технології та їх застосування НМТЗ2023». – Краматорськ: ДДМА, 2023. – С. 27-34.

12. Колесник О. О. Дослідження і розробка архітектур систем агрегації зображень / О.О. Колесник, О.С. Антоненко // Інформатика, інформаційні системи та технології: тези доповідей двадцятої всеукраїнської конференції студентів і молодих науковців. Одеса, 28 квітня 2023 р. – Одеса, 2023. – С. 148-152.

ДОДАТОК А КОД ETL

```

main.py:
import click

from flickr_controller.flickr import FlickrController
from models.flickr_model import FlickrModel
from models.reddit_model import RedditModel
from models.unsplash_model import UnsplashModel
from reddit_controller.reddit import RedditController
from unsplash_controller.unsplash import UnsplashController

@click.group()
def cli():
    pass

    @cli.command(name="get_reddit")
    @click.argument("count_of_image",                      required=True,
type=click.INT)
    @click.argument("subreddits",                          required=True,
type=click.STRING)
    def get_reddit(count_of_image: int, subreddits: str):
        reddit_model = RedditModel(
            count_of_images=count_of_image,
            list_of_subreddits=subreddits,
        )
        print(f"Here is reddit_model = {reddit_model}")
        print(f"here          is          count_of_image          =
{reddit_model.count_of_images}")
        print(f'here          is          subreddits          =
{reddit_model.list_of_subreddits}')
        reddit_controller = RedditController(reddit_model)
        if reddit_controller.download_pictures():
            print("Everything is ok")

    return True

    @cli.command(name="get_unsplash")
    @click.argument("count_of_image",                      required=True,
type=click.INT)
    @click.argument("tag", required=True, type=click.STRING)
    def get_unsplash(count_of_image: int, tag: str):
        unsplash_model = UnsplashModel(
            count_of_images=count_of_image,

```

```

        query=tag,
    )
    unsplash_controller = UnsplashController(unsplash_model)
    if unsplash_controller.download_pictures():
        print("Everything is ok")

    return True

@click.command(name="get_flickr")
@click.argument("count_of_image", type=click.INT, required=True)
@click.argument("tag", required=True, type=click.STRING)
def get_flickr(count_of_image: int, tag: str):
    flickr_model = FlickrModel(count_of_images=count_of_image,
                               tag=tag)
    print(f'here is flickr_model = {flickr_model}')
    flickr_controller = FlickrController(flickr_model)
    if flickr_controller.download_pictures():
        print(f"Images for tag {tag} in count of
{count_of_image} was downloaded to s3")

def main():
    cli()

if __name__ == '__main__':
    main()

Dockerfile:
FROM python:3.8-slim
RUN useradd --create-home --shell /bin/bash app_user
WORKDIR /home/app_user
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt
USER app_user
COPY . .
CMD ["bash"]

config.py
import os

UNSPLASH_SOURCE = "UNSPLASH"

```

```

TMP_FILE_NAME = "tmp_storage" # folder to tmp saving image

UNSPLASH_CLIENT_ID = os.getenv("UNSPLASH_CLIENT_ID")

unsplash_controller/unsplash.py
from typing import List

import requests
import json

from allmemes.amazon.storage.s3 import S3
from allmemes.models.image_model import ImageModel
from models.unsplash_model import UnsplashModel
from unsplash_controller import UNSPLASH_ACCESS_KEY,
UNSPLASH_SOURCE, UNSPLASH_URL

class UnsplashController:
    def __init__(self, unsplash_model: UnsplashModel):
        self.unsplash_model = unsplash_model
        self.s3 = S3()

    def download_pictures(self) -> bool:
        list_of_images = self._get_list_of_pictures()
        self.s3.upload_memes(list_of_images)
        self.s3.upload_metadata(list_of_images)
        return True

    def _get_list_of_pictures(self) -> List[ImageModel]:

        # получаем данные из Unsplash API
        response = requests.get(UNSPLASH_URL, params={
            'query': self.unsplash_model.query,
            'page': 1,
            'per_page': self.unsplash_model.count_of_images,
            'client_id': UNSPLASH_ACCESS_KEY
        })

        # преобразуем ответ в формат json
        data = json.loads(response.text)
        print(f'here is data = {data}')
        # проходим по всем изображениям и скачиваем их на
компьютер

        list_of_images = []
        for image in data['results']:

```

```

        print(f'here is image = {image}')
        image_url = image['urls']['raw']
        file_name
ImageModel.get_name(source=UNSPLASH_SOURCE,
sub_source=self.unsplash_model.query,

photo_prefix=ImageModel.photo_prefix())
    list_of_images.append(
        ImageModel(
            url=image_url,
            source=UNSPLASH_SOURCE,
            sub_source=self.unsplash_model.query,
            caption='mock',
            file_name=file_name
        )
    )
    return list_of_images

```

unsplash_controller/__init__.py

```

import os
from dotenv import load_dotenv
load_dotenv()

```

```

UNSPLASH_ACCESS_KEY = os.getenv("UNSPLASH_CLIENT_ID")
UNSPLASH_SOURCE = "unsplash"
UNSPLASH_URL = "https://api.unsplash.com/search/photos"

```

reddit_controller/reddit_client.py

```

import praw

```

```

from reddit_controller import CLIENT_ID_REDDIT,
CLIENT_SECRET_REDDIT, USER_AGENT_REDDIT, USERNAME_REDDIT, \
    PASSWORD_REDDIT

```

```

class RedditClient:
    def __init__(self):
        self.reddit = praw.Reddit(
            client_id=CLIENT_ID_REDDIT,
            client_secret=CLIENT_SECRET_REDDIT,
            user_agent=USER_AGENT_REDDIT,
            username=USERNAME_REDDIT,
            password=PASSWORD_REDDIT,

```

```

    )
reddit_controller/reddit.py

from typing import List

from allmemes.amazon.storage.s3 import S3
from allmemes.models.image_model import ImageModel
from models.reddit_model import RedditModel
from reddit_controller import REDDIT_SOURCE
from reddit_controller.reddit_client import RedditClient

class RedditController:
    def __init__(self, reddit_model: RedditModel):
        self.reddit_client = RedditClient()
        self.reddit_model = reddit_model
        self.s3 = S3()

    def download_pictures(self) -> bool:
        list_of_images = self._get_list_of_pictures()
        self.s3.upload_memes(list_of_images)
        self.s3.upload_metadata(list_of_images)
        return True

    def _get_list_of_pictures(self) -> List[ImageModel]:
        result_list = []
        line = self.reddit_model.list_of_subreddits
        print(f'here is line = {line}')
        sub = line.strip()
        print(f"HERE IS sub = {sub}")
        subreddit = self.reddit_client.reddit.subreddit(sub)
        print(f"Now we working with {subreddit} subreddit")
        for submission in
subreddit.new(limit=self.reddit_model.count_of_images):
            if "jpg" in submission.url.lower() or "png" in
submission.url.lower():
                file_name =
ImageModel.get_name(source=REDDIT_SOURCE, sub_source=line,
photo_prefix=ImageModel.photo_prefix())
                result_list.append(
                    ImageModel(
                        url=submission.url.lower(),
                        source=REDDIT_SOURCE,
                        sub_source=line,
                        caption=submission.title,
                        file_name=file_name

```

```

        )
    )
    return result_list
models/flickr_model.py
from dataclasses import dataclass

@dataclass
class FlickrModel:
    count_of_images: int
    tag: str
models/reddit_model.py

from dataclasses import dataclass
from typing import List

#@dataclass
#class RedditModel:
#    count_of_images: int
#    list_of_subreddits: List[str]

@dataclass
class RedditModel:
    count_of_images: int
    list_of_subreddits: str

models/flickr_model.py
from dataclasses import dataclass

@dataclass
class FlickrModel:
    count_of_images: int
    tag: str
flickr_controller/flickr_client.py
import flickrapi

from flickr_controller import FLICKR_KEY, FLICKR_SECRET

class FlickrClient:
    def __init__(self):
        self.flickr = flickrapi.FlickrAPI(FLICKR_KEY,
FLICKR_SECRET, cache=True)

```

```

flickr_controller/flickr.py
from typing import List

from allmemes.amazon.storage.s3 import S3
from allmemes.models.image_model import ImageModel
from flickr_controller.flickr_client import FlickrClient
from models import FLICKR_SOURCE
from models.flickr_model import FlickrModel

class FlickrController:
    def __init__(self, flickr_model: FlickrModel):
        self.flickr_model = flickr_model
        self.flickr_client = FlickrClient()
        self.s3 = S3()

    def download_pictures(self) -> bool:
        list_of_images = self._get_list_of_pictures()
        print(f'here is list_of_images = {list_of_images}')
        self.s3.upload_memes(list_of_images)
        self.s3.upload_metadata(list_of_images)
        return True

    def _get_list_of_pictures(self) -> List[ImageModel]:
        photos = self.flickr_client.flickr.walk(text=self.flickr_model.tag,
        tag_mode='all',
        tags=self.flickr_model.tag,
        extras='url_c',
        per_page=self.flickr_model.count_of_images,
        sort='relevance',
        list_of_images = []
        c = 0
        for photo in photos:
            image_url = photo.get('url_c')
            file_name = ImageModel.get_name(source=FLICKR_SOURCE,
            sub_source=self.flickr_model.tag,
            photo_prefix=ImageModel.photo_prefix())

```

```

        list_of_images.append(
            ImageModel(
                url=image_url,
                source=FLICKR_SOURCE,
                sub_source=self.flickr_model.tag,
                caption='mock',
                file_name=file_name
            )
        )
        c += 1
        if c == self.flickr_model.count_of_images:
            break
    return list_of_images

images/amazon/storage/client.py
import boto3

from images.amazon import AWS_ACCESS_KEY, AWS_SECRET_KEY

class S3Client:
    def __init__(self):
        self.client = boto3.client(
            "s3",
            aws_access_key_id=AWS_ACCESS_KEY,
            aws_secret_access_key=AWS_SECRET_KEY,
        )
images/amazon/storage/s3.py

import json
from datetime import datetime
from io import BytesIO
from typing import List

# import cv2
import fastavro
import requests

from images.amazon.storage import IMAGE_BUCKET_NAME,
METADATA_BUCKET_NAME
from images.amazon.storage.client import S3Client
from images.models.image_model import ImageModel

DEBUG = False

```

```

class S3:
    def __init__(self):
        self.repo_s3 = S3Client()

    def upload_meme(self, image: ImageModel):
        try:
            print(f'here is image_url = {image.url}')
            response = requests.get(image.url)
            if response.status_code == 200:
                bytesIO = BytesIO(bytes(response.content))
                with bytesIO as data:
                    self.repo_s3.client.upload_fileobj(data,
IMAGE_BUCKET_NAME, image.file_name)
            except Exception as err:
                print(err)

    def upload_memes(self, memes: List[ImageModel]):
        print(f"here is memes = {memes}")
        for meme in memes:
            print(f'here is meme = {meme}')
            self.upload_meme(meme)

    def get_s3_link_by_name(self, photo_name):
        response =
self.repo_s3.client.generate_presigned_url("get_object",
Params={'Bucket': IMAGE_BUCKET_NAME,
'Key': photo_name})
        return response
        #response =
s3_client.generate_presigned_url('get_object', Params={'Bucket':
bucket_name, 'Key': object_name},
#
ExpiresIn=expiration)

    def upload_metadata(self, memes: List[ImageModel]):
        json_list = [meme.to_json() for meme in memes]
        for json_ in json_list:
            json_["url"] =
self.get_s3_link_by_name(json_["file_name"])
            print(f'Here is json_list = {json_list}')
            json_file = json.dumps(json_list)
            print(f'here is json_file = {json_file}')
            source = memes[0].source
            print(f'here is source = {source}')

```

```

        current_date = datetime.today().strftime('%Y-%m-%d-
%H')

        print(f'here is current_date = {current_date}')
        file_name = f"{source}-{current_date}.json"

self.repo_s3.client.put_object(Bucket=METADATA_BUCKET_NAME,
Key=file_name, Body=json_file)

    """
    @staticmethod
    def get_avro_schema():
        return {
            "type": "record",
            "name": "image",
            "namespace": "image.avro",
            "fields": [
                {"name": "source", "type": "string"},
                {"name": "sub_source", "type": "string"},
                {"name": "url", "type": "string"},
                {"name": "caption", "type": "string"}
            ]
        }

    def upload_metadata(self, memes: List[ImageModel]):
        json_list = [meme.to_json() for meme in memes]
        for json_object in json_list:
            print(f'here is json_object = {json_object}')
            for key in json_object.keys():
                print(f'here is key = {key}')
                print(f'here is value = {json_object[key]}')
                if isinstance(json_object.get(key), str):
                    print('HERE IS ERROR INTO AVRO WRITING')
                    json_object[key] = "MOCK FILE ERROR"
            print(f'Here is json_list = {json_list}')
            source = memes[0].source
            print(f'here is source = {source}')
            current_date = datetime.today().strftime('%Y-%m-%d-
%H')

            print(f'here is current_date = {current_date}')
            file_name = f"{source}-{current_date}.avro"
            print(f'here is file_name = {file_name}')
            bytes_io = BytesIO()
            bytes_io.seek(0)
            fastavro.writer(bytes_io, self.get_avro_schema(),
json_list)
            bytes_io.seek(0)

```

```

        self.repo_s3.client.upload_fileobj(bytes_io,
METADATA_BUCKET_NAME, file_name)
    """

images/models/image_model.py

import uuid
from dataclasses import dataclass

@dataclass(frozen=True)
class ImageModel:
    source: str
    sub_source: str # Subreddit in case reddit, channel in
case Telegram, etc
    url: str
    caption: str
    file_name: str

    # file_name: Optional[str] = None

    def to_json(self):
        return {"source": self.source,
                "sub_source": self.sub_source,
                "url": self.url,
                "caption": self.caption,
                "file_name": self.file_name}

    @staticmethod
    def photo_prefix():
        return uuid.uuid4().hex

    @staticmethod
    def get_name(source, sub_source, photo_prefix):
        return f"{source}-{sub_source}-{photo_prefix}.png"

```

Лістинг А.1 – Extract крок у ETL

```

transform/transform.py
import os
from datetime import datetime
from typing import List

import boto3

```

```

import click
import pandas as pd
import pyarrow as pa
import pyarrow.parquet as pq
from botocore.exceptions import NoCredentialsError
from dotenv import load_dotenv
from io import StringIO

from aws.labels_metadata import detect_labels
from aws.s3.extract_metadata import extract_metadata_s3

load_dotenv()

AWS_ACCESS_KEY = os.getenv("AAA_ACCESS_KEY")
AWS_SECRET_KEY = os.getenv("AAA_SECRET_ACCESS_KEY")

BUCKET_NAME_METADATA = "metadatabucketttt"
BUCKET_NAME_IMAGE = "newimagebuckettest"

def get_s3_files(s3) -> List:
    response
    s3.list_objects_v2(Bucket=BUCKET_NAME_METADATA)

    # Проверка наличия файлов в ведре
    if "Contents" in response:
        # Извлечение списка файлов
        objects = response["Contents"]

        # Печать имен файлов
        data = [obj["Key"] for obj in objects if ".json" in
obj["Key"]]
        return data
    else:
        print("Ведро S3 пустое.")

def get_data_from_s3(s3, key):
    print(f"BUCKET_NAME_METADATA = {BUCKET_NAME_METADATA}")
    print(f"key = {key}")

    # Загрузка JSON-файла из S3
    response = s3.get_object(Bucket=BUCKET_NAME_METADATA,
Key=key)
    return response["Body"].read().decode("utf-8")

```

```

def get_json_from_s3(s3, key):
    data = get_data_from_s3(s3=s3, key=key)
    print(f"data = {data}")
    string_data = StringIO(data)
    print(f"string_data = {string_data}")
    print()
    print()
    print()
    df = pd.read_json(string_data)
    return df

def get_tags_df(df):
    df["tags"] = df["file_name"].apply(detect_labels)
    return df

def drop_none_tag(df):
    return df[~(df["tags"] == "NONE")]

def get_metadata_df(df):
    try:
        df[
            [
                "ImageSize",
                "FileType",
                "MIMEType",
                "BitsPerSample",
                "ProfileClass",
                "RenderingIntent",
                "CreateDate",
            ]
        ] = df["file_name"].apply(lambda x:
pd.Series(extract_metadata_s3(x)))
    except Exception as err:
        print(err)
    return df

def _write_parquet_file(df) -> str:
    table = pa.Table.from_pandas(df)
    time_now = datetime.now().strftime("%Y-%m-%d-%H-%M-%S")

    parquet_name = f"data{time_now}.parquet"

```

```

pq.write_table(table, parquet_name)
return parquet_name

def send_s3_parquet(parquet_name: str):
    # Підключаємося к S3
    s3 = boto3.resource("s3",
aws_access_key_id=AWS_ACCESS_KEY,
aws_secret_access_key=AWS_SECRET_KEY)

    # Загружаємо файл на S3

s3.Bucket(BUCKET_NAME_METADATA).upload_file(parquet_name,
f"gold/{parquet_name}")

def save_file(df):
    parquet_name = _write_parquet_file(df)
    send_s3_parquet(parquet_name)
    os.remove(parquet_name)

def check_s3_object_exists(bucket_name, key):
    s3 = boto3.client('s3')
    try:
        s3.head_object(Bucket=bucket_name, Key=key)
        return True
    except NoCredentialsError:
        print("Credentials not available")
        return False
    except Exception as e:
        print(e)
        return False

def filter_metadata_in_s3(df, bucket_name):
    df['exists_in_s3'] = df.apply(lambda row:
check_s3_object_exists(bucket_name, row['image_key']), axis=1)
    df_filtered = df[df['exists_in_s3']] # Вибрати тільки
рядки, де exists_in_s3 = True
    df_filtered = df_filtered.drop(columns=['exists_in_s3'])
# Видалити створений стовпець
    return df_filtered

def filter_df(df):

```

```
filter_metadata_in_s3(df, BUCKET_NAME_IMAGE)

def process_file(filename: str, s3):
    df = get_json_from_s3(s3=s3, key=filename)

    print(df)
    df = get_tags_df(df)

    print(df)
    df = filter_df(df)
    df = drop_none_tag(df)

    df = get_metadata_df(df)

    print(df)
    save_file(df)

@click.group()
def cli():
    pass

@cli.command(name="transform")
def transform_data():
    s3 = boto3.client(
        "s3",
        aws_access_key_id=AWS_ACCESS_KEY,
        aws_secret_access_key=AWS_SECRET_KEY,
    )
    list_files = get_s3_files(s3)
    print(list_files)

    for file in list_files:
        process_file(file, s3)

def main():
    cli()

if __name__ == '__main__':
    main()
```

```

transform/config.py
import os

from dotenv import load_dotenv

load_dotenv()

AWS_ACCESS_KEY = os.getenv("AAA_ACCESS_KEY")
AWS_SECRET_KEY = os.getenv("AAA_SECRET_ACCESS_KEY")

BUCKET_NAME_METADATA = "metadatabucketttt"
BUCKET_NAME_IMAGE = «newimagebuckettest"

transform/aws/labels_metadata.py

import os
from typing import List

import boto3
from dotenv import load_dotenv

load_dotenv()

AWS_ACCESS_KEY = os.getenv("AAA_ACCESS_KEY")
AWS_SECRET_KEY = os.getenv("AAA_SECRET_ACCESS_KEY")

BUCKET_NAME_METADATA = "metadatabucketttt"
BUCKET_NAME_IMAGE = "newimagebuckettest"

def detect_labels(photo: str) -> List or str:
    #try:
        result_list = []
        # session = boto3.Session(profile_name='profile-name')
        # client = session.client('rekognition')
        client = boto3.client(
            "rekognition",
            aws_access_key_id=AWS_ACCESS_KEY,
            aws_secret_access_key=AWS_SECRET_KEY,
            region_name='us-east-1'
        )
        response = client.detect_labels(
            Image={"S3Object": {"Bucket": BUCKET_NAME_IMAGE,
"Name": photo}}
            # MaxLabels=10,

```

```

# Uncomment to use image properties and filtration
settings
# Features=["GENERAL_LABELS", "IMAGE_PROPERTIES"],
#           Settings={"GeneralLabels":
{"LabelInclusionFilters":["Cat"]},
# "ImageProperties": {"MaxDominantColors":10}}
)
print()
print(response)
labels = response.get("Labels")
if not labels:
    return "NONE"

for label in labels:
    result_list.append(label["Name"])
    print("Label: " + label["Name"])
    print("Confidence: " + str(label["Confidence"]))
    print("Instances:")
    instances = label.get("Instances")
    #if not instances:
    #    return "NONE"
    if instances:
        for instance in instances:
            print(" Bounding box")
            print("         Top:         " +
str(instance["BoundingBox"]["Top"]))
            print("         Left:        " +
str(instance["BoundingBox"]["Left"]))
            print("         Width:       " +
str(instance["BoundingBox"]["Width"]))
            print("         Height:      " +
str(instance["BoundingBox"]["Height"]))
            print("         Confidence:   " +
str(instance["Confidence"]))
            print()

        print("Parents:")
        parents = label.get("Parents")
        #if not parents:
        #    return "NONE"
        if parents:
            for parent in parents:
                print(" " + parent["Name"])
                result_list.append(parent["Name"])

```

```

print("Aliases:")
aliases = label.get("Aliases")
if aliases:
    for alias in label.get("Aliases"):
        print(" " + alias["Name"])
        result_list.append(alias["Name"])

        print("Categories:")
categories = label.get("Categories")
#if not categories:
#    return "NONE"
if categories:
    for category in categories:
        print(" " + category["Name"])
        result_list.append(category["Name"])
        print("-----")
        print()

if "ImageProperties" in str(response):
    print("Background:")
    print(response["ImageProperties"]["Background"])
    print()
    print("Foreground:")
    print(response["ImageProperties"]["Foreground"])
    print()
    print("Quality:")
    print(response["ImageProperties"]["Quality"])
    print()

# return len(response["Labels"])
    return result_list
#except Exception as err:
#    print(f"here is error: ")
#    print(err)
#    print(f'error in image = {photo}')
#    return "NONE"

transform/aws/s3/extract_metadata.py
import boto3
import exiftool
import tempfile
from pprint import pprint

import pandas as pd
import os

```

```

AWS_ACCESS_KEY = os.getenv("AAA_ACCESS_KEY")
AWS_SECRET_KEY = os.getenv("AAA_SECRET_ACCESS_KEY")

BUCKET_NAME_METADATA = "metadatabucketttt"
BUCKET_NAME_IMAGE = "newimagebuckettest"

def extract_metadata_s3(photo_name):
    s3 = boto3.client('s3',
                      aws_access_key_id=AWS_ACCESS_KEY,
                      aws_secret_access_key=AWS_SECRET_KEY,
                      )

    # Create a temporary file to save the downloaded image
    with tempfile.NamedTemporaryFile() as temp_file:
        print(f'here is photo_name = {photo_name}')
        print(f'here is temp_file.name = {temp_file.name}')
        # Download the image from S3 to the temporary file
        s3.download_file(BUCKET_NAME_IMAGE, photo_name,
temp_file.name)

        # Use pyexiftool to extract metadata from the
downloaded image file
        with exiftool.ExifToolHelper() as et:
            metadata = et.get_metadata(temp_file.name)
            metadata = metadata[0]

            result_data = {"ImageSize":
metadata.get("Composite:ImageSize"),
                        "FileType":
metadata.get("File:FileType"),
                        "MIMEType":
metadata.get("File:MIMEType"),
                        "BitsPerSample":
metadata.get("File:BitsPerSample"),
                        "ProfileClass":
metadata.get("ICC_Profile:ProfileClass"),
                        "RenderingIntent":
metadata.get("ICC_Profile:RenderingIntent"),
                        "CreateDate":
metadata.get("XMP:CreateDate") \
}

    """

```

```

        result_data
[metadata.get("Composite:ImageSize"),
        metadata.get("File:FileType"),
        metadata.get("File:MIMEType"),

metadata.get("File:BitsPerSample"),

metadata.get("ICC_Profile:ProfileClass"),

metadata.get("ICC_Profile:RenderingIntent"),
        metadata.get("XMP:CreateDate")
]
"""

```

```

"""
Composite:ImageSize
File:FileType
File:MIMEType
File:BitsPerSample
ICC_Profile:ProfileClass
ICC_Profile:RenderingIntent
XMP:CreateDate
"""
#return metadata
return result_data

```

```
# Example usage
```

```

"""

def main():
    photo_name = "FLICKR-dogs-
0fe3350239cd46e99dca35f5ecda5973.png"
    result = extract_metadata_s3(photo_name=photo_name)
    pprint(result)

if __name__ == '__main__':
    main()

```

Лістинг А.2 – Transform крок у ETL

ДОДАТОК В КОД ELT PIPELINE

```

import snowflake.connector
import os

# Параметры подключения к Snowflake
snowflake_account = os.getenv("snowflake_account")
snowflake_user = os.getenv("snowflake_user")
snowflake_password = os.getenv("snowflake_password")
snowflake_warehouse = os.getenv("snowflake_warehouse")
snowflake_database = os.getenv("snowflake_database")
snowflake_schema = os.getenv("snowflake_schema")
snowflake_table = os.getenv("snowflake_table")

s3_bucket = os.getenv('s3_bucket')
s3_prefix = os.getenv("s3_prefix")

def save_to_db(s3_files):
    conn = snowflake.connector.connect(
        user=snowflake_user,
        password=snowflake_password,
        account=snowflake_account,
        warehouse=snowflake_warehouse,
        database=snowflake_database,
        schema=snowflake_schema
    )

    cur = conn.cursor()

    for s3_file in s3_files:
        s3_file = s3_file.split()[-1]
        copy_into_command = f"COPY INTO '{snowflake_table}'
FROM s3://{s3_bucket}/{s3_file} " \
        f"FILE_FORMAT = (TYPE = 'JSON'
STRIP_OUTER_ARRAY = TRUE) PURGE = TRUE"
        cur.execute(copy_into_command)
        print(f"File {s3_file} loaded into Snowflake.")

    cur.close()
    conn.close()

def get_s3_files():

```

```

        aws_s3_list_files_command = f"aws s3 ls
s3://{s3_bucket}/{s3_prefix} --recursive --no-sign-request --
region your_s3_region"
        return
os.popen(aws_s3_list_files_command).read().strip().split('\n')

```

```

def main():
    s3_files = get_s3_files()
    save_to_db(s3_files)

```

```

if __name__ == '__main__':
    main()

```

Лістинг В.1 – Load крок у ELT

```

macros/check.py
# check.py
import boto3
from botocore.exceptions import NoCredentialsError

BUCKET_NAME_IMAGE = "newimagebuckettest"

def check_existing(image_key):
    s3 = boto3.client('s3')
    try:
        s3.head_object(Bucket=BUCKET_NAME_IMAGE,
Key=image_key)
        return True
    except NoCredentialsError:
        print("Credentials not available")
        return False
    except Exception as e:
        print(e)
        return False
    return True

check_s3_object_exists.sql

-- macros/check_s3_object_exists.sql

{% macro check_s3_object_exists(image_key) %}

```

```

        {{ return(run_python_file("check.py", "check_existing",
[image_key])) }}
    {% endmacro %}

```

```

macros/get_image_tags.py

```

```

import os

```

```

import boto3

```

```

from typing import List

```

```

from macros.check import BUCKET_NAME_IMAGE

```

```

AWS_ACCESS_KEY = os.getenv("AWS_ACCESS_KEY")

```

```

AWS_SECRET_KEY = os.getenv("AWS_SECRET_KEY")

```

```

def detect_labels(photo: str) -> List or str:

```

```

    #try:

```

```

        result_list = []

```

```

        # session = boto3.Session(profile_name='profile-name')

```

```

        # client = session.client('rekognition')

```

```

        client = boto3.client(

```

```

            "rekognition",

```

```

            aws_access_key_id=AWS_ACCESS_KEY,

```

```

            aws_secret_access_key=AWS_SECRET_KEY,

```

```

            region_name='us-east-1'

```

```

        )

```

```

        response = client.detect_labels(

```

```

            Image={"S3Object": {"Bucket": BUCKET_NAME_IMAGE,

```

```

            "Name": photo}}

```

```

            # MaxLabels=10,

```

```

            # Uncomment to use image properties and filtration

```

```

settings

```

```

            # Features=["GENERAL_LABELS", "IMAGE_PROPERTIES"],

```

```

            # Settings={"GeneralLabels":

```

```

{"LabelInclusionFilters":["Cat"]},

```

```

            # "ImageProperties": {"MaxDominantColors":10}}

```

```

        )

```

```

        print()

```

```

        print(response)

```

```

        labels = response.get("Labels")

```

```

        if not labels:

```

```

            return "NONE"

```

```

        for label in labels:

```

```

result_list.append(label["Name"])
print("Label: " + label["Name"])
print("Confidence: " + str(label["Confidence"]))
print("Instances:")
instances = label.get("Instances")
#if not instances:
#    return "NONE"
if instances:

    for instance in instances:
        print(" Bounding box")
        print(" Top: " +
str(instance["BoundingBox"]["Top"]))
        print(" Left: " +
str(instance["BoundingBox"]["Left"]))
        print(" Width: " +
str(instance["BoundingBox"]["Width"]))
        print(" Height: " +
str(instance["BoundingBox"]["Height"]))
        print(" Confidence: " +
str(instance["Confidence"]))
        print()

print("Parents:")
parents = label.get("Parents")
#if not parents:
#    return "NONE"
if parents:
    for parent in parents:
        print(" " + parent["Name"])
        result_list.append(parent["Name"])

print("Aliases:")
aliases = label.get("Aliases")
if aliases:
    for alias in label.get("Aliases"):
        print(" " + alias["Name"])
        result_list.append(alias["Name"])

        print("Categories:")
categories = label.get("Categories")
#if not categories:
#    return "NONE"
if categories:
    for category in categories:
        print(" " + category["Name"])

```



```

)

# Create a temporary file to save the downloaded image
with tempfile.NamedTemporaryFile() as temp_file:
    print(f'here is photo_name = {photo_name}')
    print(f'here is temp_file.name = {temp_file.name}')
    # Download the image from S3 to the temporary file
    s3.download_file(BUCKET_NAME_IMAGE, photo_name,
temp_file.name)

# Use pyexiftool to extract metadata from the
downloaded image file
with exiftool.ExifToolHelper() as et:
    metadata = et.get_metadata(temp_file.name)
    metadata = metadata[0]

    result_data = {"ImageSize":
metadata.get("Composite:ImageSize"),
                    "FileType":
metadata.get("File:FileType"),
                    "MIMEType":
metadata.get("File:MIMEType"),
                    "BitsPerSample":
metadata.get("File:BitsPerSample"),
                    "ProfileClass":
metadata.get("ICC_Profile:ProfileClass"),
                    "RenderingIntent":
metadata.get("ICC_Profile:RenderingIntent"),
                    "CreateDate":
metadata.get("XMP:CreateDate")\
                    }

#return metadata
return result_data
macros/process_image_exif.sql
-- macros/process_image_exif.sql

{% macro process_image_exif(image_name) %}
  {{ run_python_script(
    script="macros/process_image_exif.py",
    image_name=image_name
  ) }}
{% endmacro %}

models/exiftool_model.sql
WITH image_metadata AS (

```

```

SELECT
    *,
    {{ process_image_exif('file_name') }} AS exif_metadata
FROM {{ ref('metadata_with_tags') }}
)

SELECT
    *
FROM image_metadata;

models/filtered_metadata_model_name.sql

-- models/filtered_metadata_model_name.sql

WITH filtered_data AS (
    SELECT
        *
        FROM {{ ref('metadata_model') }} -- Використовуємо dbt-
        функцію ref для посилання на іншу модель
        WHERE file_name IS NOT NULL AND file_name != '' AND
        file_name != 'NONE' -- Фільтруємо за умовою, що file_name не
        порожнє і не NULL
    )

SELECT
    *
FROM filtered_data;
models/metadata_model.sql
-- models/metadata_model.sql

WITH metadata_cte AS (
    SELECT
        caption,
        file_name,
        source,
        sub_source,
        url
    FROM {{ ref('metadata') }}
)

SELECT
    caption,
    file_name,
    source,
    sub_source,

```

```
        url
FROM metadata_cte;
models/metadata_with_tags.sql

WITH image_tags AS (
    SELECT
        *,
        {{ get_image_tags('file_name') }} AS image_tags
    FROM {{ ref('second_filtered') }}
)

SELECT
    *
FROM image_tags;

models/second_filtered.sql
-- models/second_filtered.sql
WITH filtered_data AS (
    SELECT
        *
        FROM {{ ref('filtered_metadata_model_name') }}
        WHERE {{ check_s3_object_exists('file_name') }}
)
SELECT
    *
FROM filtered_data;
```

Лістинг В.2 – Transform крок у ELT

ДОДАТОК С

YML ФАЙЛИ GITHUB ACTIONS

```

name: CI

on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]

jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      max-parallel: 4
      matrix:
        python-version: [3.8, 3.9]

    steps:
      - uses: actions/checkout@v3
      - name: Set up Python ${ matrix.python-version }
        uses: actions/setup-python@v3
        with:
          python-version: ${ matrix.python-version }
      - name: Install Dependencies
        run: |
          python3 -m pip install --upgrade pip
          pip install -r requirements.txt

```

ЛІСТИНГ С.1 – Test.yml

```

name: Python deploy
on:
  push:
    branches: [ "master" ]
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Run command on remote
        uses: D3rHase/ssh-command-action@v0.2.2
        with:
          host: ${ secrets.HOST }

```

```
port: ${secrets.PORT}
user: ${secrets.USER}
private_key: ${secrets.GIT_SECRET_KEY}
command: |
  cd /home/nginx;
  docker-compose down;
  cd ..;
  rm -fr project;
  git clone git@github.com:project;
  cd project;
  touch .env;
  echo
'Var=${Var}\SecondVar=${secrets.SecondVar}' > .env;
  docker-compose up --build -d;
```

-e

Лістинг C.2 – Deploy.yml