PROOF
OF
CONCEPT
DATA PIPELINE

BY

PRAJJWAL PANDEY

TO

KAVAYA.AI

April, 2025

Introduction

The purpose of this POC is to design and implement a multi-stage data pipeline, encompassing data extraction(and cleaning), enrichment via AI analysis, and publishing. This task simulates a real-world scenario involving gathering information from substack publications (like https://robertreich.substack.com/about and https://www.publicnotice.co/), processing it, and making it available for downstream systems through Kafka topic.

Objectives

1. Monitors a defined set of Substack publications(2).
2. Poll at certain interval to get the data(Every Midnight).
3. Extracts key information and content from these new posts.
4. Analyzes the extracted article content using an AI model (LLM) to identify key topics and associated sentiment per topic(Mocked Sentiment Analysis).
5. Publishes the combined extracted and analyzed data to a streaming platform (Kafka topic).

Substack Publications Monitored

Chose on the basis of popularity and daily updates.

1. **Robert Reich** RSS Feed → https://robertreich.substack.com/feed
2. **Public Notice** RSS Feed → https://www.publicnotice.co/feed

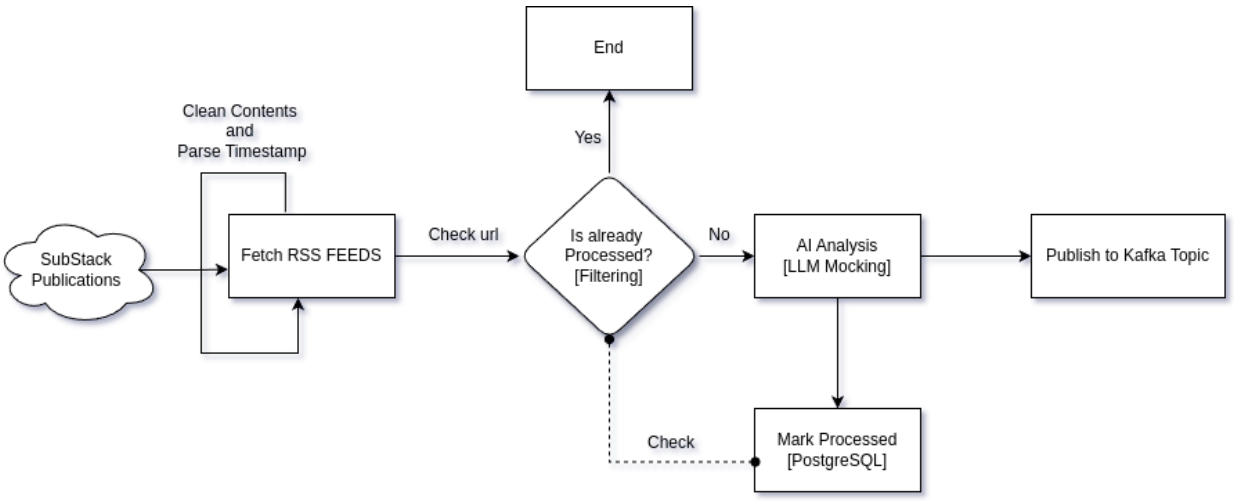
Tools

- Python : Program Source Code
- Plombery : Orchestration Tool
- Kafka : Publishing Message to Kafka Topic
- Docker : Kafka Containerization
- Feedparser, BeautifulSoup, html, re, emoji : Clean Text Parising
- PostgreSQL : Persistent Storage(Marks and Checks Already Processed)

File Structuring

```
substack/
    utils/ # Utility folder for modules
            __init__.py # Package initialization
            db.py # Database utility
            kafka_producer.py # Kafka producer utility
            llm_analyzer.py # LLM analyzer utility
            rss_fetcher.py # RSS fetcher utility

    docker-compose.yml # Docker Compose configuration file
    main.py # Main entry point of the application
    pipeline.py # Defines the pipeline logic
    requirements.txt # Python dependencies
```



Pipeline Implementation

1. **Detection Strategy**
   Every time we fetch RSS feeds, we parse each post's url.Compare the post url against url stored in a PostgreSQL database. Only new posts (newer than the last stored) are sent for processing. Thus, duplication is avoided and incremental updates are ensured.

2. **Extraction Approach**

   - **Source**
     RSS feeds from two Substack sites:
     - Robert Reich's Substack (https://robertreich.substack.com/feed)
     - Public Notice (https://www.publicnotice.co/feed)
   - **Library Used**:
     - feedparser for parsing RSS XML
     - BeautifulSoup for HTML content extraction
     - re, html, emoji for cleaning
   - **Parsing Logic**
     Extract fields:
     - post_title
     - author_name
     - publication_name
     - timestamp

- post_url
- post_content (special cleaned extraction from ¡content:encoded¿)

- **Cleaning Content**:
  - Remove HTML tags, unwanted promotional messages, images, buttons, emojis.
  - Decode HTML entities.
  - Strip non-ASCII characters.
  - Normalize whitespace.

  Thus, only pure readable text remains for AI analysis.

3. **AI Analysis Method**

   For each cleaned post content, we call an LLM(mocked). Send the post content with a prompt like:

```
"Analyze the following article and return:

1. "The list of main topics discussed (as keywords)."

2. "The overall sentiment (Positive, Negative, Neutral).""
```

   The LLM is expected to return structured data:

```
{
  "topics": ["economy", "inflation", "policy"],
  "sentiment": "Negative"
}
```

   In this case, the AI analysis function can simply mock the output based on random assignment.

4. **Kafka Message Schema**

   Each post after enrichment (content + AI analysis) is published as a JSON message to Kafka, in a topic like processed_substack_articles.

```
{
  "post_title": "Post Title Here",
  "author_name": "Author Name Here",
  "publication_name": "Publication Name Here",
  "timestamp": "2025-04-25T12:00:00+00:00",
  "post_url": "https://example.substack.com/p/post-title",
  "post_content": "Cleaned main text here...",
  "topics": ["topic1", "topic2", "topic3"],
  "sentiment": {
    "topic1": "positive",
    "topic2": "neutral"
  }
}
```

5. **Update Database**

   Update database with latest processed timestamp and URL for tracking.

Pipeline finishes cleanly and waits for the next scheduled run.

Setup Instructions

1. **Execution Steps**

   (a) Clone/download the project repository.

   (b) Install dependencies.

   (c) Start Kafka (docker compose up -d) and create a kafka topic.

   (d) Run the Python script main.py: "uvicorn main:app –reload"

2. **Dependencies Installation**

```
pip install -r requirements.txt
OR
manually install all the above mentioned tools.
```

3. **Kafka Setup**

   docker-compose.yml file as below and run "docker compose up -d"

```
services:
  # Zookeeper Service
  zookeeper:
    image: confluentinc/cp-zookeeper:7.5.0
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
    networks:
      - default

  # Kafka Broker Service
  kafka:
    image: confluentinc/cp-kafka:7.5.0 # Kafka broker image version 7.5.0
        from Confluent
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
    networks:
      - default

networks:
  default:
```

Create the topic

```
docker exec -it <kafka_container_id> /bin/bash
kafka-topics.sh --create --topic processed_substack_articles --bootstrap-
    server localhost:9092 --partitions 1 --replication-factor 1
```

4. **Database Configuration**

   Replace the following credentials with your own from utils/db.py file.

```
dbname='database_name',
user='username',
password='0000',
host='localhost',
port='5432'
```

Assumptions Made

1. Both RSS entries contain either (content:encoded) or similar (content).
2. Polling everymidnight to get the batch of data.
3. A dummy notification system through plombery for catching and seeing notifications easily of pipline. [config file in root directory]
4. AI model(mocked but if real) responds within a reasonable time in correct formatting as requested.
5. PostgreSQL database (for tracking latest processed post timestamps) is available if using full production mode.

Design Considerations

1. **Separation of Concerns (Decoupling Stages)**

   By structuring the pipeline so that the message is pushed to Kafka after AI analysis, we maintain clear separation between different stages of the pipeline. Each stage is focused on a specific task, which makes the system modular and easier to maintain. This approach allows us to:

   Easily Replace or Update Components: For example, we can change the AI model or the RSS feed parsing logic without impacting other stages (e.g., the Kafka publishing logic).

   Isolate Failures: If the AI analysis fails, it won't directly impact the Kafka stream or any downstream consumers. This makes it easier to identify and fix issues in individual components.

   This modular design follows the principle of single responsibility in system design, making the system easier to scale, extend, and debug.

2. **Better Control Over Data Flow**

   When we use Kafka Streams directly from the beginning, every post has to be published to Kafka immediately after it's detected and extracted. This means that:

   We would be pushing raw data (possibly unclean or incomplete) into Kafka, which may require additional post-processing in Kafka consumers.

   If there's a failure in one of the earlier steps (e.g., content extraction or AI analysis), we'd have to rely on Kafka Streams' fault tolerance and replay mechanisms, which could become more complex.
   By pushing data after AI processing, we gain full control over:

   Content Validation: We can ensure that only valid, complete, and well-analyzed data is published to Kafka, reducing the need for error-prone post-processing.

   Error Handling: Failures in any stage (e.g., AI analysis) can be handled locally (e.g., retrying or routing the message to a dead-letter queue), rather than relying on Kafka Streams' built-in error handling and retries, which can be more difficult to debug in complex pipelines.

3. **Simplicity and Maintainability**

   While Kafka Streams provides powerful real-time stream processing capabilities, it can add unnecessary complexity for simple tasks like RSS feed monitoring and basic AI analysis.

4. **Scalability and Future Flexibility**

   While Kafka Streams is excellent for real-time stream processing, this design offers more flexibility as the system scales:

   Future Integration with Kafka Streams: If, in the future, real-time processing of posts becomes a requirement (e.g., for analyzing streams of posts in real-time rather than in batches), we can integrate Kafka Streams at a later stage without disrupting the entire pipeline.

   Scalability: The design allows for easy scaling. For example, we can scale the extraction and AI analysis components independently of Kafka. If we need to scale the

AI analysis (perhaps using a distributed model), we can focus on those components without worrying about Kafka Streams' limitations.

5. **Error Handling**

   - Logging processes.
   - Retry Mechanism Through Orchestration tool(Plombery).
   - Runtime Error Notification(Mocked) mechanism.

6. **Source Interaction**
   Polling Frequency: Respectful polling (e.g., daily midnight), No aggressive scraping, Configurable polling intervals for each source to avoid overloading sites.
   Caching:We remember the latest fetched post timestamps (persisted in POstgres DB) to avoid re-fetching or re-processing old posts.

7. **Scalability**
   Parallelization: Can be Extracted and processed posts in parallel using lightweight threading or async operations. Multiple feeds can be polled independently (multi-threaded or using an event loop).
   Distributed Setup: Split components (fetcher, AI analyzer, Kafka publisher) into independent microservices later.Deploy multiple instances of fetchers and analyzers as needed (scale horizontally).
   Batch Processing:Instead of sending posts one by one to AI and Kafka, we can batch posts: Bulk send to LLM (if API supports) and Bulk publish to Kafka to reduce overhead. This is because substack posts are usually posted once a day in any publication.

8. **Data Schema Management**

   Define a clear JSON schema upfront for the message structure.

```
{
  "title": "Post Title",
  "author": "Author Name",
  "publication": "Publication Name",
  "timestamp": "ISO 8601 timestamp",
  "url": "Post URL",
  "content": "Cleaned Post Content",
  "topics": ["topic1", "topic2"],
  "sentiment": {
    "topic1": "positive",
    "topic2": "neutral"
  }
}
```

Known Limitations Potential Future Enhancements

- Couldn't find how to implement retries in case of failure in plombery(From Official Website Documentation). ChatGPT Suggestion(Didn't Work on Implementing):

```
from plombery import flow, step

@step(retries=5, retry_delay=10)
def fetch_data():
    # Simulate risky operation
    raise ValueError("Failed to fetch data!")

@step
def process_data():
    print("Processing data...")

@flow
def my_pipeline():
    fetch_data()
    process_data()
```

  In future, with correct corrections retires can be implemented.

- Only Applicable to Subsatck Publications i.e. 2. Further scaled to larger publication.

- The LLM implementaiton is mocked, which can be replaced easily with actual implementation.

- Kafka Broker Dependency, the system assumes a working Kafka cluster and if Kafka is unavailable, message publishing will fail and no backup queue exists yet.
  Solution : Kafka Publishing Retry and Dead Letter Queue (DLQ)

- Url based Duplication check.
  Future Enhancement: Get the publication time and url of last feed post and compare with lastest stored post in database to know if it has to be published.
  Limitaion: Couldn't Get the implementation correct due to project time constraints.

- AI based Enhanced Feed Parsing instead of current "Simple RSS Feed Parsing".

- No vault integration((e.g., AWS Secrets Manager, HashiCorp Vault) yet.) for Secrets Management.