Final Report: Cloud Pipe Line Engineering – Amazon Web Services

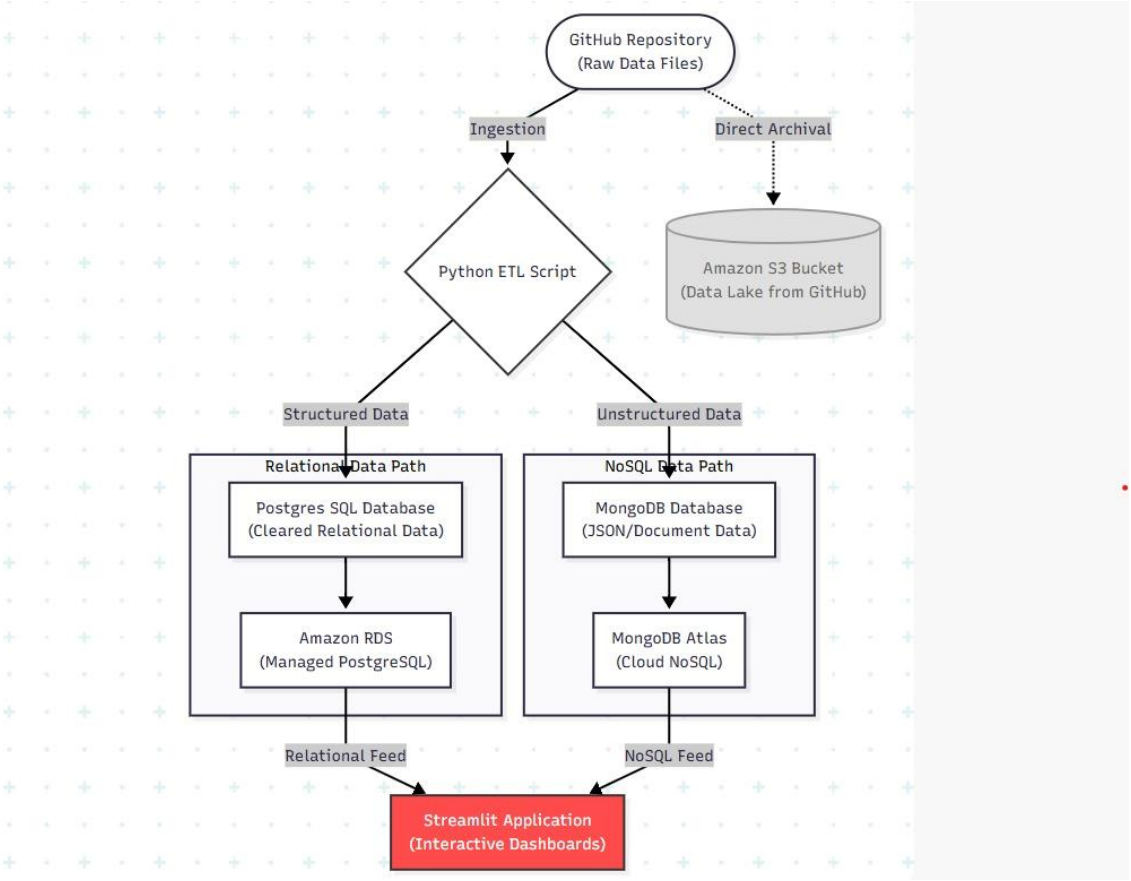
Subject: Data Engineering, Modeling, and Visualization

Submission Date: February 20, 2026

1. Executive Summary

This project presents a robust, end-to-end data engineering pipeline designed to process, model, and visualize public city council meeting data using the MeetingBank dataset. The objective was to transform disparate raw JSON metadata and transcript files into a highly queryable, dual-database architecture. By leveraging an Agile methodology, our three-member team successfully implemented a pipeline that extracts actionable intelligence—such as city-wise participation trends and speaker metrics—and serves it through an interactive Streamlit application.

2. System Architecture



Our data architecture follows a structured pipeline from raw ingestion to end-user visualization, ensuring data is routed to the most appropriate storage paradigm (Relational vs. NoSQL).

Architecture Components:

1. Source & Archival: Raw JSON and CSV data is sourced from GitHub/Zenodo repositories to a local staging environment. A direct archival path to an Amazon S3 Bucket acts as our Data Lake, ensuring raw files are preserved immutably.
2. Processing Engine: A custom Python ETL script serves as the central processing node, reading downloaded data, performing transformations, and splitting the data stream.
3. Relational Path (Structured Data): Aggregated metrics and metadata flow into a PostgreSQL database hosted on Amazon RDS.
4. NoSQL Path (Unstructured Data): Heavy text data (full transcripts and summaries) flows into a MongoDB Atlas cloud cluster.
5. Presentation Layer: A Streamlit application acts as the data consumer, simultaneously querying both AWS RDS and MongoDB Atlas to provide a unified dashboard.

3. Data Ingestion and Cleaning

The MeetingBank dataset contains complex nested JSON structures. Our ingestion strategy required processing a representative subset of the data, focusing on handling anomalies and extracting meaningful derived attributes.

3.1 Metadata Parsing and Normalization

We utilized Python's json and pandas libraries to traverse the dataset. Key cleaning steps included:

- **City Extraction:** Split from the meeting_id string**.**
- **Robust Date Handling:** Meeting IDs contained dates, but formats occasionally varied. We implemented Regular Expressions (r'(\d{8})') paired with try/except blocks to safely parse strings into datetime.date objects**.**

3.2 Deriving Attributes and Fallback Logic

Raw data is rarely perfect. We encountered missing VideoDuration fields and negative integer errors. To fix this, we built a fallback mechanism that reads the actual transcript segment timestamps if the metadata duration is invalid.

Furthermore, we derived two new operational attributes by iterating through the transcript segments:

- **Speaker Count:** Calculated by adding unique speakers to a Python set().
- **Word Count:** Calculated by traversing the nbest nested arrays and counting individual word tokens.

```python
# Extracting derived attributes from nested JSON
speaker_set = set()
word_count = 0

if "segments" in tdata:
    for seg in tdata["segments"]:
        speaker = seg.get("speaker")
        if speaker is not None:
            speaker_set.add(speaker)
        for alt in seg.get("nbest", []):
            word_count += len(alt.get("words", []))

speaker_count = len(speaker_set)
```

4. Data Transformation and Modeling

To support both analytical queries (e.g., "What is the average meeting length in Seattle?") and operational queries (e.g., "Fetch the transcript for meeting X"), we implemented two distinct schemas.

4.1 Relational Schema (PostgreSQL)

Designed for fast aggregations and filtering.

Column Name	Data Type	Constraint	Description
meeting_id	**VARCHAR**	**PRIMARY KEY**	**Unique identifier for the meeting.**
city	**VARCHAR**	**NOT NULL**	**City where the meeting occurred.**
meeting_date	**DATE**	**NULL**	**Date of the meeting.**

Column Name	Data Type	Constraint	Description
duration	**FLOAT**	**NULL**	**Length of the meeting in seconds.**
agenda_type	**VARCHAR**	**NULL**	**Categorization of the meeting topic.**
speaker_count	**INT**	**NOT NULL**	**Derived total unique speakers.**
word_count	**INT**	**NOT NULL**	**Derived total words spoken.**

4.2 Document Schema (MongoDB)

Designed for high-throughput reads of massive text strings.

Field Name	Data Type	Description
meeting_id	String	Indexed field. Matches the PK in PostgreSQL.
transcript	String	Full text of the meeting.
summary	String	AI-generated summary of the meeting.
agenda_items	Array	List of specific agenda points (if available).

5. Data Storage and Indexing Strategy

- **Amazon RDS (PostgreSQL**): We utilized SQLAlchemy to establish a connection to our managed AWS instance. We chose PostgreSQL for its robust adherence to ACID properties, ensuring our metadata is strictly typed and consistent.
- **MongoDB Atlas:** We utilized pymongo to insert records into a cloud cluster. MongoDB's BSON structure is ideal for the varying lengths of meeting transcripts, which would cause severe bloat in a relational table.

- **Indexing Strategy:** To ensure the Streamlit app remains highly responsive, we applied a B-Tree index on the meeting_id column in PostgreSQL and a Single Field Index on the meeting_id field in MongoDB. This allows rapid lookup times when a user clicks a meeting in the dashboard**.**

6. Querying and Analysis

Our Streamlit application acts as the analytics engine. It relies on specific querying patterns to generate insights.

6.1 Analytical SQL Queries

To generate the metrics for our dashboard, we utilize SQL aggregations.

```sql
-- Example Analytical Query: Aggregating meeting statistics by city
SELECT
    city,
    COUNT(meeting_id) as total_meetings,
    SUM(speaker_count) as total_speakers,
    AVG(word_count) as average_words_per_meeting
FROM meetings_postgres_ready_final_v5
GROUP BY city
ORDER BY total_meetings DESC;
```

6.2 NoSQL Retrieval Queries

When a user selects a specific meeting from the Streamlit dropdown, a targeted NoSQL query fetches the unstructured data instantly without scanning the whole database.

```python
# MongoDB precise document retrieval
doc = mongo_collection.find_one({"meeting_id": selected_meeting})
transcript_text = doc.get("transcript", "No transcript available")
```

7. Presentation and Visualization

The frontend pipeline was built using Streamlit, matplotlib, and seaborn. The interactive dashboard allows users to:

- **Filter Context:** Select a specific city from the sidebar, which dynamically updates the SQL query parameters.
- **View High-Level Metrics:** Total meetings, words, and speakers are calculated in real-time.

- **Analyze Trends:** A time-series line plot shows word_count trends over the meeting_date, helping identify periods of heavy civic activity.
- **Operational Retrieval:** A dedicated text area dynamically loads the massive transcript payload from MongoDB only when requested, keeping the app lightweight.

8. Project Management and Team Reflection

8.1 Agile Methodology and Work Allocation

Our 3-member team adopted an Agile framework, utilizing a Kanban board to track progress across three specific sprints

Team Member	Core Responsibility	Sprint Focus & Technical Execution
Shesha Shayana Panyala	**ETL & Data Architect**	**Sprint 1:** Built the Python extraction scripts. Handled all RegEx cleaning, JSON parsing, and calculation of derived attributes (word/speaker counts).
Venu Kamatam	**Cloud & DB Engineer**	**Sprint 2:** Provisioned AWS RDS and MongoDB Atlas. Designed the schemas and managed the data ingestion via sqlalchemy and pymongo.
Dinesh Karthik	**Analytics & Frontend**	**Sprint 3:** Developed the Streamlit app. Implemented data visualizations and authored the comprehensive technical report.

8.2 Technical Lessons Learned

The primary technical challenge was establishing a "Hybrid Join" at the application layer. Because structured and unstructured data lived on completely different clouds (AWS vs. Atlas), we could not use standard SQL JOINs. We solved this by using pandas to hold the relational metadata in memory, and executing targeted MongoDB lookups based on user interaction, optimizing memory usage.

8.3 Challenges, Mistakes, and Rectifications

Throughout our Agile sprints, the team encountered several technical hurdles that temporarily broke our data flow.

- **Mistake 1: Data Type Mismatches in PostgreSQL**

    - **The Issue:** Inconsistent formats and unexpected null values in the raw JSON metadata caused sqlalchemy to throw errors when inserting records into PostgreSQL, violating our schema constraints.
    - **The Rectification:** We implemented strict data validation and explicit type casting in Python before the loading phase. We introduced a try/except block to guarantee meeting_id strings were properly parsed into datetime.date objects.
- **Mistake 2: Assuming Schema Consistency (Missing Durations)**
    - **The Issue:** We initially assumed all meeting records would uniformly possess a valid VideoDuration attribute. We soon discovered missing fields and negative integers, which would have skewed our analytical dashboard metrics.
    - **The Rectification:** We engineered a dynamic fallback mechanism. If the script detects invalid durations, it overrides the metadata by diving into the unstructured segments array to calculate the duration directly from the transcript timestamps.

9. Critical and Ethical Reflection

9.1 Ethical Aspects of Public Meeting Data

Working with the MeetingBank dataset presents unique ethical considerations. While city council meetings are public record, massive aggregation and algorithmic processing of this data change the nature of its accessibility.

- **Privacy of Citizens:** Constituents speaking at open mics may not anticipate their words being permanently indexed, analyzed, and distributed globally in datasets. We must ensure our pipeline does not inadvertently highlight or dox vulnerable individuals.
- **Transparency:** To ensure ethical use, our pipeline clearly documents its data sources. Furthermore, any analytical biases—such as inaccurate transcription software misinterpreting certain dialects—must be acknowledged, as these transcripts form the basis of our word_count metrics.

9.2 Conclusion

This project successfully demonstrates a modern, scalable data engineering workflow. By decoupling storage based on data structure (SQL for analytics, NoSQL for documents), we achieved a highly performant system. The resulting dashboard provides a transparent, user-friendly lens into local civic processes, fully satisfying the operational and analytical goals of the assignment.