



DALHOUSIE
UNIVERSITY

CSCI 5410
SERVERLESS DATA PROCESSING

“Quick Data Processor”

Sprint 3 Report

SDP-Team8

Submitted By:

- Rushil Borad [rs519505@dal.ca]
- Mansi Kalathiya [mn410013@dal.ca]
- Bhavya Dave [bh392017@dal.ca]
- Avadh Rakholiya [AV786964@DAL.ca]

GitLab Repository: https://git.cs.dal.ca/rakholiya/serverless_project

Contents

| | |
|---|----|
| Project Planning Overview | 3 |
| System Architecture Design | 5 |
| Module Implementations Module 2: Virtual Assistant Module | 6 |
| Module 3: Message Passing Module | 9 |
| Pseudo Code | 11 |
| Module 4: Notifications | 15 |
| Module 5: Data Processing | 17 |
| Data Processing 1: JSON to CSV | 17 |
| Data Processing 2: Named Entities Extraction | 22 |
| Data Processing 3: Word Cloud using Looker Studio | 24 |
| Process Flow | 24 |
| Future Considerations | 24 |
| Pseudocode:Frontend-file-handling-function..... | 27 |
| Pseudocode: Word-count -function pseudocode | 28 |
| Module 6: Data Analysis | 29 |
| Implementation Details: | 29 |
| Testing and Validation:..... | 29 |
| Pseudo-code | 33 |
| Module 7: Web Application Building and Deployment | 35 |
| Deployed URL: | 36 |
| Gantt Chart | 37 |
| Conclusion..... | 38 |
| References | 39 |

Project Planning Overview

In Sprint 3, our focus was to continue developing and integrating other core modules, including the completion of necessary backend functionality and enhancement of the frontend interface. This sprint has been devoted to the assurance of smooth user interaction, enhancement in real-time data processing, and laying the ground for further modules like data analysis and messaging. By addressing key components of the system strategically, we tried to solidify the core infrastructure of the system to make the integration of more complex modules easier in future sprints.

1. Modules Planned for Sprint 3:

- a. Module 2: Virtual Assistant (remaining part completed)
- b. Module 3: Message Passing
- c. Module 5: Data Processing
- d. Module 6: Data Analysis
- e. Module 7: Web Application Frontend

2. Rationale for Module Selection in Sprint 3:

- Selection of the above modules was based on the need for the completion of the critical functionality of the system. These modules are essential for:
- Message Passing: This will enable the customers to communicate with QDP agents to resolve issues in real time.
- Data Processing: Automate data transformation and processing tasks, which is necessary for processing user-submitted files and enhancing the overall system.
- Data Analysis: Incorporating sentiment analysis to provide actionable insights from customer feedback for improving customer experience and informed operational decisions.
- Web Application Front-end: Ensure the front end is functional with user-facing capabilities such as the display of feedback, authentication, and all the necessary integrations to backend systems.

3. Solution Overview:

- Message Passing Module: In the module, real-time messaging and integrating databases like DynamoDB and Firestore were tricky tasks that involved seamless communication and logging. The mitigation for such challenges has been done through schema refinements and implementation of error handling strategies.

- Data Processing Module: This module has integrated AWS Glue, Lambda functions, and DynamoDB into the architecture, where major challenges involved the orchestration of data flows or any delay in the input processing.
- Data Module: The integration of real-time feedback analysis with the frontend required careful synchronization with real-time data updates and the Google Natural Language API.
- Frontend Integration: Completion of the frontend setup included integrating multiple modules and making it user-friendly, especially for dynamic data visualization.

Through Sprint 3, we will develop a well-structured serverless project by utilizing the services provided by AWS and GCP to ensure the system is robust, scalable, and efficient. This project showcases how cloud technologies can be leveraged for enhanced operational efficiency.

System Architecture Design

Architecture Overview

Our system architecture integrates AWS and GCP services to build a hybrid serverless application. This architecture is designed to maximize scalability, optimize costs, and enhance security while leveraging the specialized capabilities of each platform.

1. AWS Services:

- a. **Cognito** for multi-factor authentication in User Management.
- b. **Lambda** for handling third-factor math-based authentication and processing tasks.
- c. **SNS** for sending notifications (email) post user registration and login.
- d. **DynamoDB** as the primary storage solution for user information, chosen for its high availability and serverless model.
- e. AWS Glue: Converts uploaded JSON files into CSV format. This process is logged in dynamodb **for tracking and reporting purposes.**

2. GCP Services:

- a. **Dialogflow** for the Virtual Assistant module, allowing automated user assistance and navigation.
- b. **Cloud Functions** for backend support in the Virtual Assistant and Message Passing functionalities.
- c. **Firestore** for real-time data storage and retrieval to support messaging and user queries.

3. Application Frontend:

- a. **React** was chosen for building the frontend due to its component-based structure, which provides an interactive and user-friendly interface. It also allows easy integration with backend APIs.
- b. **GCP Cloud Run** is planned for frontend deployment, allowing serverless hosting and seamless scalability.

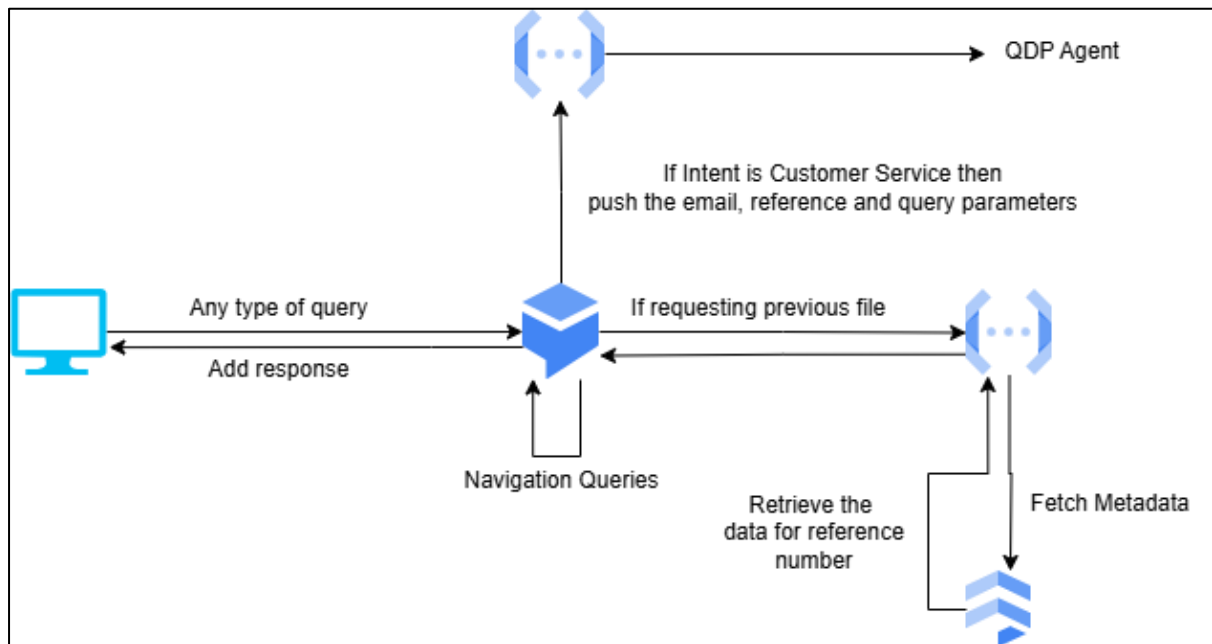
4. Integration and Communication:

- a. AWS SNS is connected to the user registration workflow to trigger email notifications.
- b. GCP Pub/Sub will be used for messaging once the Message Passing module .

Module Implementations

Module 2: Virtual Assistant Module

Figure 1 Virtual Assistant architecture



The Virtual Assistant Module is designed to elevate user experience by providing efficient and intuitive support. The module leverages Google Cloud Platform technologies, including Dialogflow, Firestore, and Cloud Functions, to deliver robust and scalable virtual assistance.

Key Features and Functionalities:

- **Online Virtual Assistance:** The module offers real-time assistance to users navigating the application site. It can answer common queries such as "how to register?" and provide guidance on various tasks.
- **Result Retrieval:** Users can easily obtain info of previously processed files by simply providing a processing reference code.
- **Customer Support:** The module efficiently captures customer concerns and seamlessly forwards them to QDP Agents for further investigation and resolution.

Future Considerations:

To further enhance the module's capabilities, we plan to explore the following:

- **Advanced Natural Language Processing:** Integrating more sophisticated NLP techniques to improve the accuracy and relevance of responses.
- **Proactive Assistance:** Utilizing machine learning to anticipate user needs and provide timely suggestions.
- **Multilingual Support:** Expanding the module's language capabilities to cater to a diverse user base

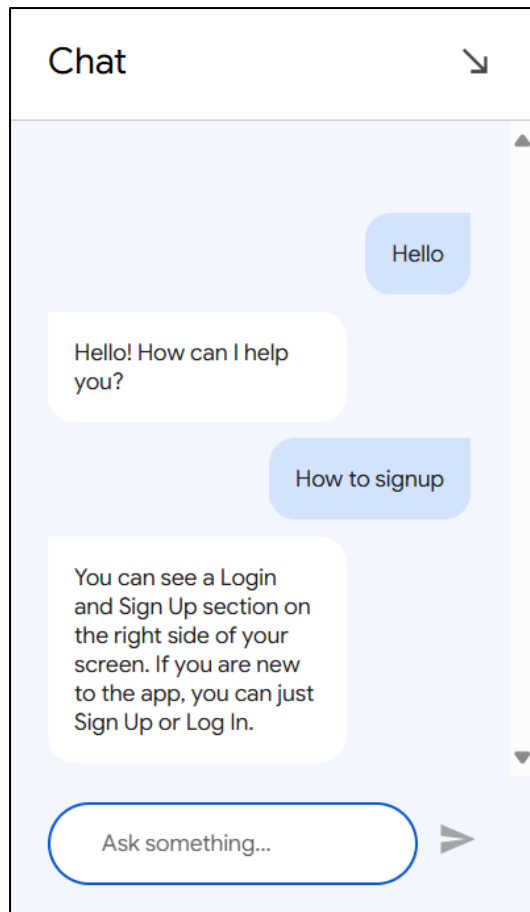


Figure 3 Chatbot navigation

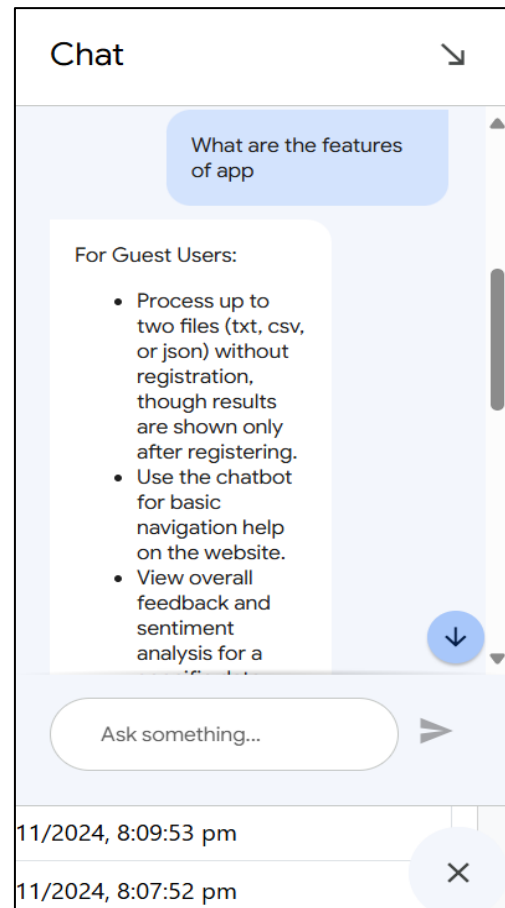


Figure 2 Chatbot highlighting features

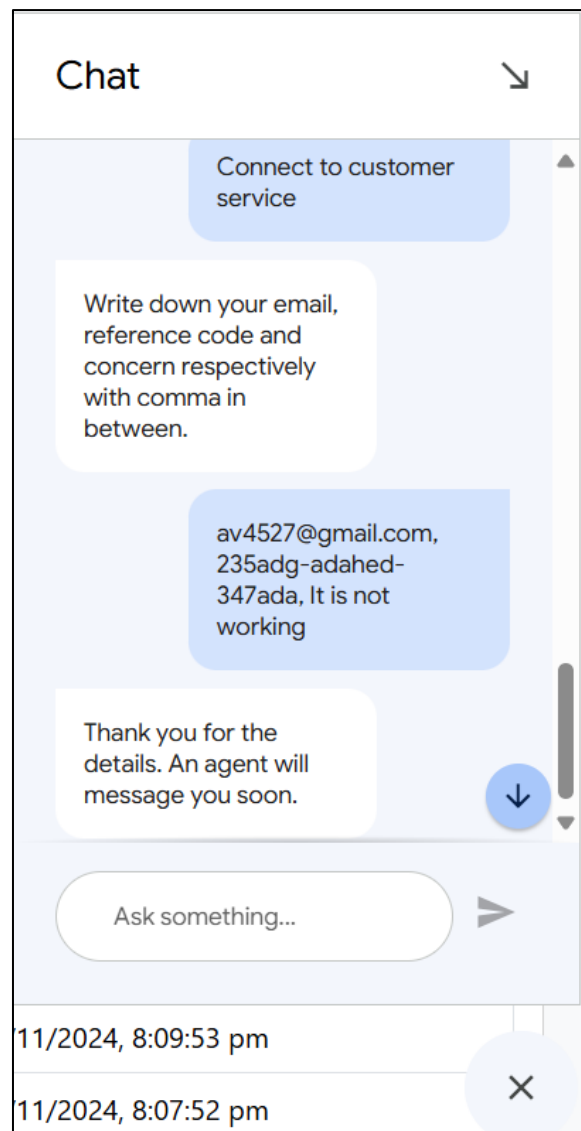


Figure 4 Chatbot redirecting to QDP agents

Module 3: Message Passing Module

Objective

The Message Passing Module is designed to provide a means for efficient, scalable, and reliable communication between the different components or services of a distributed system. It provides an easy way of sending messages with minimum latency, ensuring reliability in message delivery and the proper handling of asynchronous tasks. This module, through the use of cloud technologies like AWS Lambda[4], Google Cloud Pub/Sub[5], and GCP Pub/Sub and Cloud Functions[6], ensures responsiveness and scalability of the system while supporting event-driven architectures.

Some of the key objectives include:

Efficient Communication: This technique aims at achieving timely, trustworthy delivery of a message, in both synchronous and asynchronous contexts.

Scalable: It is designed with support for processing large quantities of messages and, when under high load conditions, has scalability with elasticity .

Event-Driven Architecture Support: Allow for seamless integration with event-driven systems, enabling components to act upon and trigger actions concerning specific events or conditions.

Architecture Diagram:

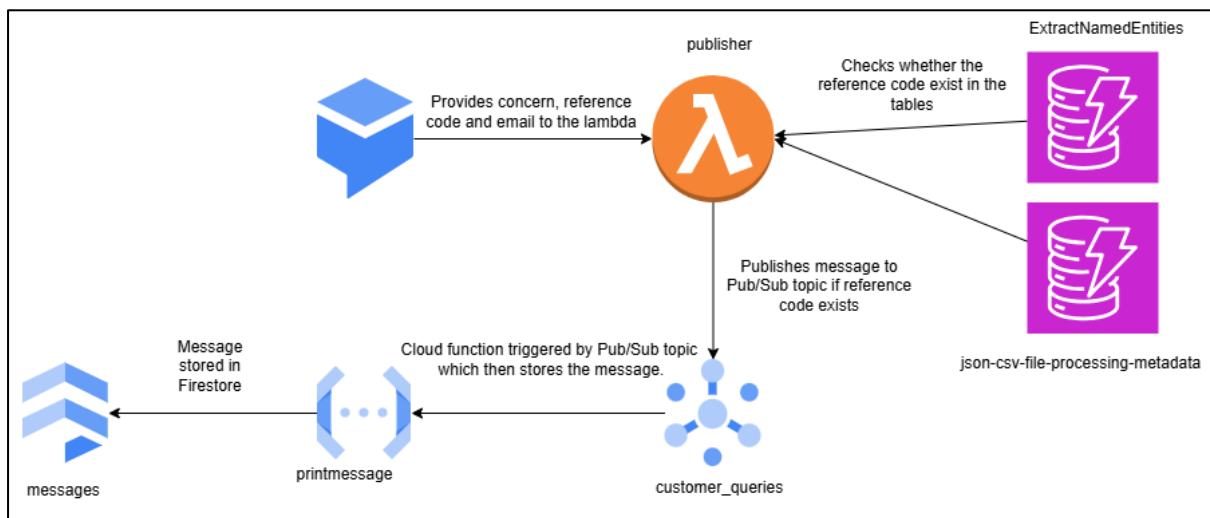


Figure 5 Architecture Diagram

Implementation Details

GCP Dialogflow- A custom payload is set that when it detects @sys.any, @sys.any and @sys.email, it sends the payload to the AWS Lambda through the http endpoint API Gateway. So, these payloads are set as trigger for the lambda.

AWS Lambda (Publisher)- The lambda checks whether the reference_code that has come in the payload is present in the two tables or not. If the reference_code is present, then the request with payload that contains the reference_code, email and the concern is then published to the pub/sub named customer_queries[4].

AWS DynamoDB- It is used for checking whether the reference_code contains the reference_code or not[3].

GCP Pub/Sub- Once the reference_code is confirmed by the lambda, the message is then published to the Pub/Sub. Also, this Pub/Sub is a trigger to a Cloud function named printmessage.[5]

GCP CloudFunction- This is the CloudFunction which retrieves the published message from Pub/Sub and then stores the message in the Firestore. It then assigns an agent to the user and then stores the assigned agent, the reference code, the email of the user and the concern in the Firestore collection[6].

GCP Firestore- It is used to store the assigned agent, the reference code, the email of the user and the concern[8].

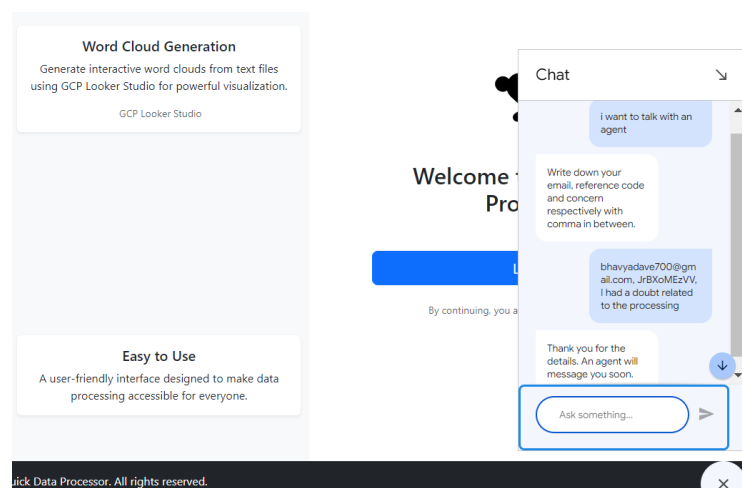


Figure 6: Sending concern, email and reference code

Chat

Query: I had a doubt related to the processing

Response:

Figure 7 Asking for Query

Answer the Queries

Query from av4527@gmail.com
It is not working

Response
Enter your response here
Submit

Query from bhavyadave700@gmail.com
I had a doubt related to the processing

Response
Enter your response here
Submit

Figure 8 Answer the Queries

Pseudo Code

Publisher Lambda function

- **Initialize GCP Pub/Sub client** with project credentials
- **Initialize AWS DynamoDB client** with region settings
- **On handler trigger:**
- Parse the incoming event body
- Extract required parameters: email, reference_code, concern
- **If any required parameter is missing:**
 - Return error response (400 Bad Request)
- Create a **Pub/Sub message** with extracted data (reference_code, concern, email)
- **Check if Pub/Sub topic exists:**
 - If topic does not exist, return error response (404 Not Found)
- **Publish message to Pub/Sub topic:**
 - If message is published successfully, return success response (200 OK)
 - If message publishing fails, return error response (500 Internal Server Error)

Print-message Cloud Function

- **Initialize Firestore client** using Google Cloud Firestore
- **Define Cloud Event handler** (`log_pubsub_message`):
- **Decode the Pub/Sub message data** (base64 encoded)
- **Log the decoded message**
- **Parse the message data** from JSON
- **Extract and validate** required fields (`reference_code`, `concern`, `email`):
 - If any of the fields are missing, **raise an error** and return a failure response
- **Prepare message structure** for storing:
 - Create a new query object with the concern and an empty response
- **Hardcode assigned agent's email**
- **Fetch all agents from Firestore:**
 - Get the list of agents from the `agents` collection
- **Check if agents exist:**
 - If no agents are found, log a message and exit
- **Check if an existing message document for the email exists:**
 - If yes, **append the new query** to the existing document in Firestore
 - If no, **create a new document** in Firestore with the message details and assigned agent
- **Handle exceptions:**
 - If an error occurs during processing, **log the error** and terminate

Retrieve Chats

Define Cloud Function handler (`retrieve_chats`):

- **Check if the request is a preflight OPTIONS request** (for CORS):
 - If yes, **return a 204 status** with CORS headers allowing POST, GET, and OPTIONS methods
- **Check if the request method is POST:**
 - If not, **return 405 status** with an error message and allowed methods
- **Parse JSON payload** from the request:
 - If the payload is missing or doesn't contain the required `email` field, **return a 400 error** with a message
- **Initialize Firestore client** to interact with the Firestore database
- **Query Firestore collection** (`messages`) for documents where `email` matches the provided email:
 - If no documents are found, **return a 404 error** with a message
- **Prepare the response** with chat messages:
 - Extract each message's query and response from Firestore documents

- **Return a JSON response** containing the chat messages with a 200 status
- **Handle exceptions:**
 - If an error occurs during the process, **return a 500 error** with the error message

Unanswered_queries CloudFunction

Define Cloud Function handler (get_unanswered_queries):

- **Initialize Firestore client** to interact with the Firestore database
- **Reference Firestore collection** (messages)
- **Fetch all documents** from the messages collection
- **Prepare the response** by initializing an empty list for unanswered queries
- **Iterate over each document:**
 - Extract email and messages from the document data
 - For each message, **check if the response is empty:**
 - If yes, append the query and email to the unanswered queries list
- **If no unanswered queries are found:**
 - Return a 200 response with a message: "No unanswered queries found."
- **If unanswered queries are found:**
 - Return a 200 response with the list of unanswered queries
- **Handle exceptions:**
 - If an error occurs during processing, **return a 500 error** with the error message

Agent_response Cloud Function

Define Cloud Function handler (update_response_in_firestore):

- **Set common CORS headers** for response
- **Handle preflight OPTIONS request** (for CORS):
 - Return a 204 status with CORS headers
- **Parse the JSON payload** from the request:
 - If the payload is invalid (missing email or response), return a 400 error
- **Extract email and response** from the payload
- **Initialize Firestore client** to interact with Firestore database
- **Reference the Firestore collection** (messages)
- **Query Firestore** to find the document with the specified email:
 - If no document is found, return a 404 error with an appropriate message
- **Check for message with empty response** in the document:
 - If found, update the response field with the new value

- If no message with an empty response is found, return a 404 error
- **Update the Firestore document** with the modified messages array
- **Return success response** indicating that the update was successful
- **Handle exceptions:**
 - If any error occurs during the process, return a 500 error with the exception message

Module 4: Notifications

Objective

The goal of the Notification Module is to offer a dependable and scalable way of notifying users or systems in real time. This module targets increasing user engagement and responsiveness through timely updates, alerts, and reminders on email.

Architecture Diagram

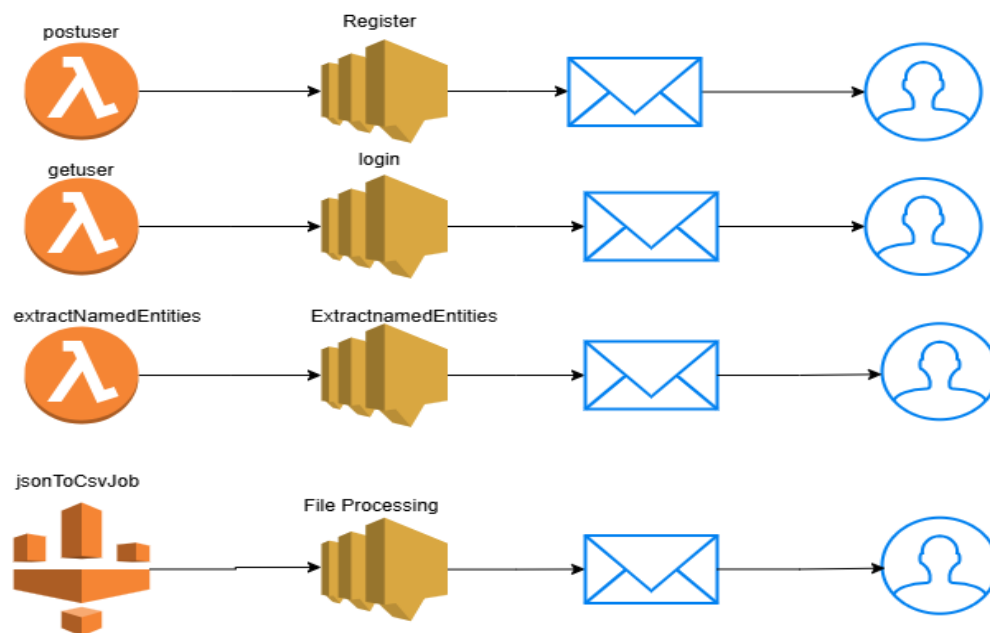


Figure 9 Flow Diagram


Implementation







The implementation of the module is through the creation of 4 SNS topics which are assigned to the different lambdas. Messages are published to these topics whenever the implementation of the respective lambda is successful[4].

Email is the endpoint of subscription of these SNS topics. Through this, the user retrieves the message of success when the message is published do the SNS topic as the endpoint is the email of the user.

AWS Notification Message

 AWS Notifications <no-reply@sns.amazonaws.com>

 Monday, December 02, 2024 10:56:54 PM

 Deliverability  Reply  Forward  Print  Delete 

JSON to CSV file processing has been done.
File: PWmq0AILI1/part-00000-9fbbfbd7-477d-4bc2-a9ba-3f5fc2801fbb-c000.csv
Reference Code: PWmq0AILI1

--

If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
<https://sns.us-east-1.amazonaws.com/unsubscribe.html?SubscriptionArn=arn:aws:sns:us-east-1:668916080027:JSONtoCSV:283e7759-d08f-4708-94da-5184404e0a24&Endpoint=rushil@yopmail.com>

Please do not reply directly to this email. If you have any questions or comments regarding this email, please contact us at <https://aws.amazon.com/support>

Module 5: Data Processing

Data Processing 1: JSON to CSV

Objective:

DP1 automates the process of converting JSON files to CSV format while handling complex, deeply nested structures. The module ensures users can upload JSON files, monitor processing, and access the converted results effectively.

Architecture Diagram:

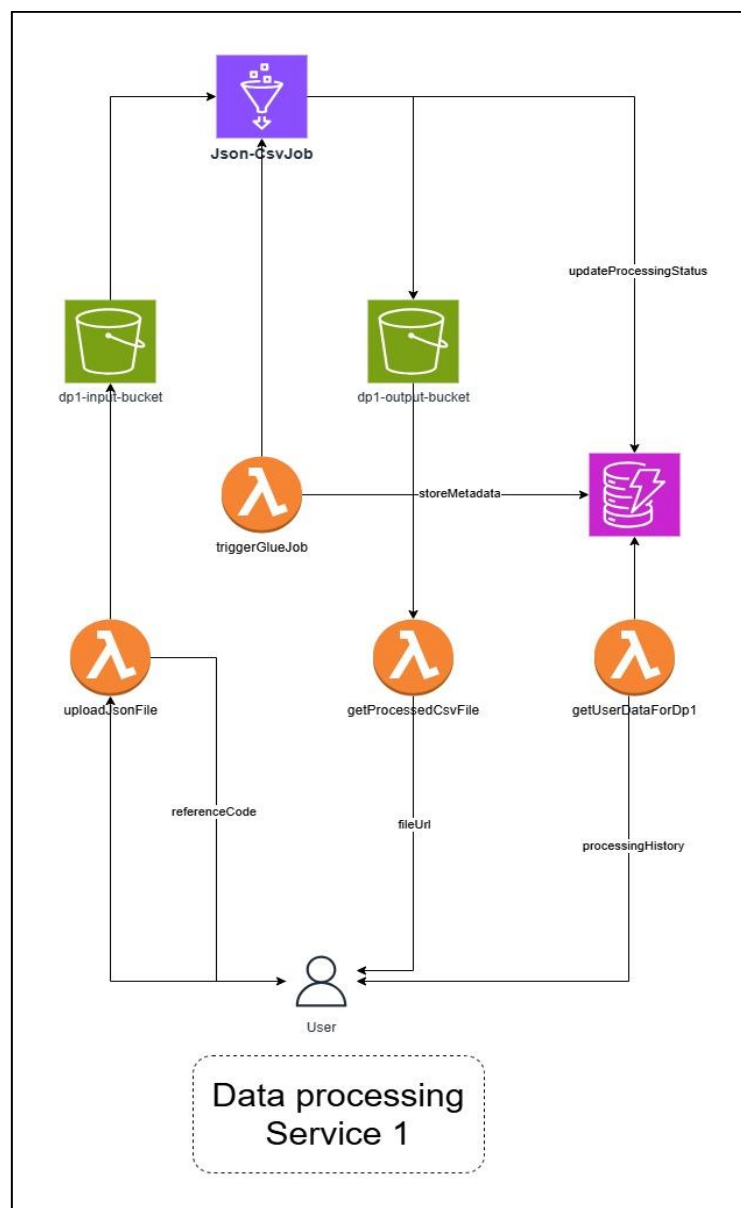


Figure 100 Data Processing Service 1 Architecture Diagram

Implementation Details:

File Upload (uploadJsonFile Lambda):

Users initiate the process by uploading a JSON file through the frontend. The Lambda function encodes the file in Base64, validates the format, and stores metadata like the filename and *userEmail* in DynamoDB. This setup allows seamless tracking of user-specific tasks.

Triggering Glue Job (triggerGlueJob Lambda):

When a new file is uploaded to the input S3 bucket, an S3 event triggers the *triggerGlueJob* Lambda function.

- The Lambda initiates an **AWS Glue Job (JsonToCsvJob)** to handle the conversion process.
- Glue's Python-based script flattens deeply nested JSON structures, such as arrays or hierarchical objects, ensuring the CSV output is clean and usable.
- The processed CSV is stored in the **output S3 bucket**, and its metadata is updated in DynamoDB.

Fetching Processed Files (getProcessedCSVFile Lambda):

This function retrieves the processed file's metadata and the download link from DynamoDB.

- Metadata includes the **file name**, **processing status**, and **timestamp**, giving users a clear overview of their file processing.

User History (getUserDetailsForDp1 Lambda):

To enhance the user experience, this Lambda provides access to a user's **processing history**, including previously uploaded files and their corresponding results.

Pseudocode: JSON to CSV File Processing Workflow

Upload JSON File

- **Input:** User uploads JSON file (*file_name*, *file_data*) along with *user_email*.
- **Validate Input:**
 - Ensure *user_email*, *file_name*, and *file_data* are provided.
- **Decode File:**
 - Decode *file_data* from Base64 format.
- **Generate Reference Code:**
 - Create a unique 10-character alphanumeric reference code.

- **Upload to S3:**
 - Save the file to the `input_bucket` with the path format: `input/{reference_code}/{file_name}`.
- **Save Metadata to DynamoDB:**
 - Save details (ReferenceCode, userEmail, FileName, S3InputPath, ProcessingStatus, etc.) with ProcessingStatus set to **Pending**.

Trigger Glue Job

- **Input:** S3 event triggered when a file is uploaded.
- **Parse Event:**
 - Extract `bucket_name`, `object_key`, and `reference_code` from the S3 event.
- **Trigger Glue Job:**
 - Start a Glue job (JsonToCsvJob) with the following arguments:
 - `input_bucket`, `input_key`, `output_bucket`, `reference_code`, and `dynamodb_table`.
- **Update DynamoDB:**
 - Update ProcessingStatus to **In Progress** and save the Glue JobRunId.

Glue Job: JSON to CSV Conversion

- **Input:** File from S3 (`input_bucket` and `input_key`).
- **Read JSON File:**
 - Load the JSON file into a DataFrame.
- **Flatten Nested JSON:**
 - Recursively flatten all deeply nested structures (e.g., arrays and structs).
- **Write CSV File:**
 - Save the flattened DataFrame as a CSV file to the `output_bucket` with the path: `{reference_code}/`.
- **Update DynamoDB:**
 - Locate the processed CSV file in the S3 output directory.
 - Construct the public download URL.
 - Update ProcessingStatus to **Succeeded**, along with the OutputLocation and PublicDownloadURL.
- **Notify User:**
 - Retrieve userEmail from DynamoDB.
 - Subscribe the user to an SNS topic (JSONtoCSV).
 - Publish a notification with the `reference_code` and file details.

Get Processed CSV File Metadata

- 1. **Input:** reference_code provided by the user.
- 2. **Fetch Metadata:**
 - a. Query DynamoDB using the reference_code.
- 3. **Return Response:**
 - a. Include file_name, processing_status, and public_download_url.
 - b. Return error if the reference_code does not exist.

Screenshots:

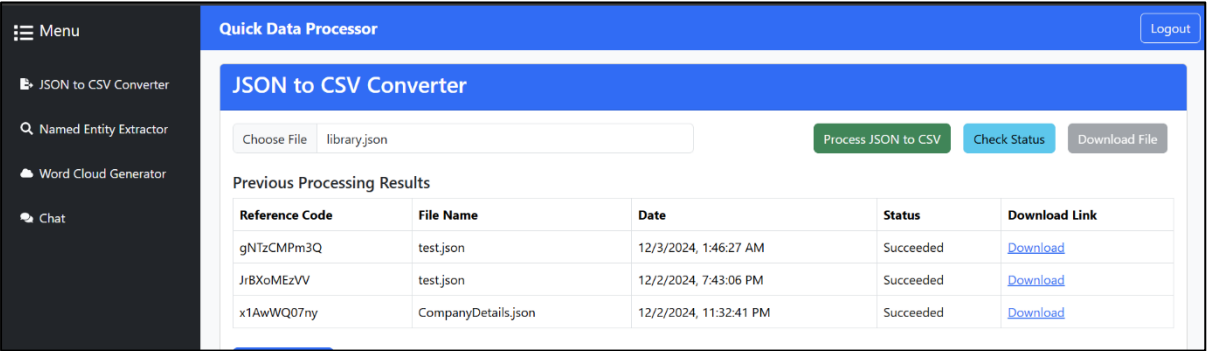


Figure 11 Data Processing 1

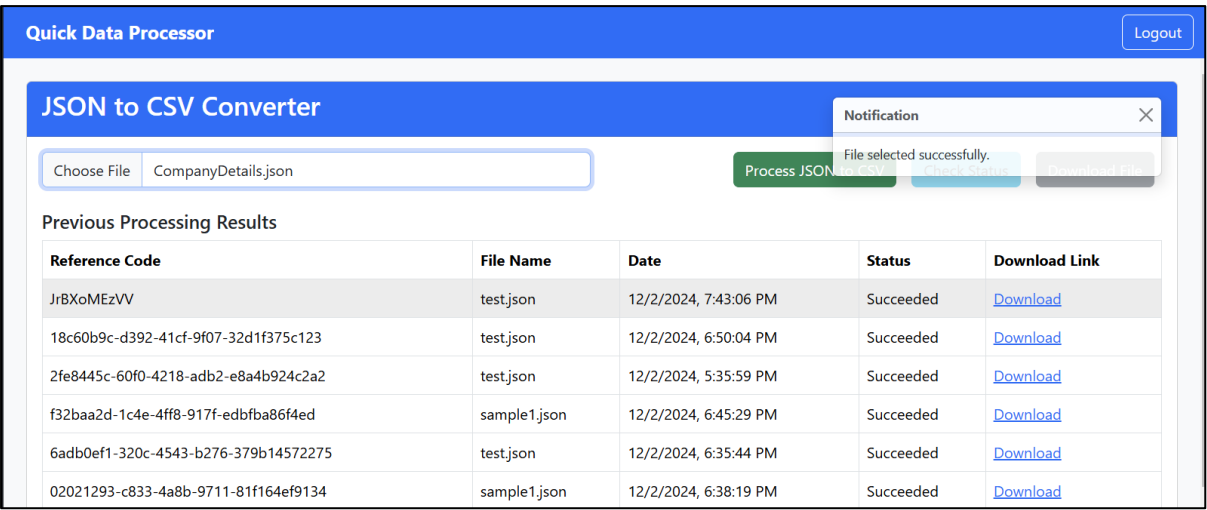


Figure 12 Data Processing 1-File Selected successfully

JSON to CSV Converter

Choose File

test.json

Process JSON to CSV

Check Status

Download File

Notification

File is still processing. Please check again later.

Previous Processing Results

| Reference Code | File Name | Date | Status | Download Link |
|--------------------------------------|---------------------|------------------------|-----------|--------------------------|
| JrBXoMEzVV | test.json | 12/2/2024, 7:43:06 PM | Succeeded | Download |
| 18c60b9c-d392-41cf-9f07-32d1f375c123 | test.json | 12/2/2024, 6:50:04 PM | Succeeded | Download |
| x1AwWQ07ny | CompanyDetails.json | 12/2/2024, 11:32:41 PM | Succeeded | Download |
| 2fe8445c-60f0-4218-adb2-e8a4b924c2a2 | test.json | 12/2/2024, 5:35:59 PM | Succeeded | Download |

Figure 13 Data Processing 1-File is still processing

JSON to CSV Converter

Choose File

library.json

Process JSON to CSV

Check Status

Download File

Notification

File processing completed. Ready for download.

Previous Processing Results

| Reference Code | File Name | Date | Status | Download Link |
|----------------|---------------------|------------------------|-----------|--------------------------|
| xZcNi1mVVd | library.json | 12/3/2024, 1:57:23 AM | Succeeded | Download |
| gNTzCMPm3Q | test.json | 12/3/2024, 1:46:27 AM | Succeeded | Download |
| JrBXoMEzVV | test.json | 12/2/2024, 7:43:06 PM | Succeeded | Download |
| x1AwWQ07ny | CompanyDetails.json | 12/2/2024, 11:32:41 PM | Succeeded | Download |

Give Feedback

Customer Feedbacks

Figure 14 Data Processing 1-File processing completed

Data Processing 2: Named Entities Extraction

Objective:

DP2 processes text files to extract named entities like people, organizations, and locations. It showcases the power of Natural Language Processing (NLP) using **SpaCy** in AWS Lambda for real-time processing [7].

Architecture Diagram:

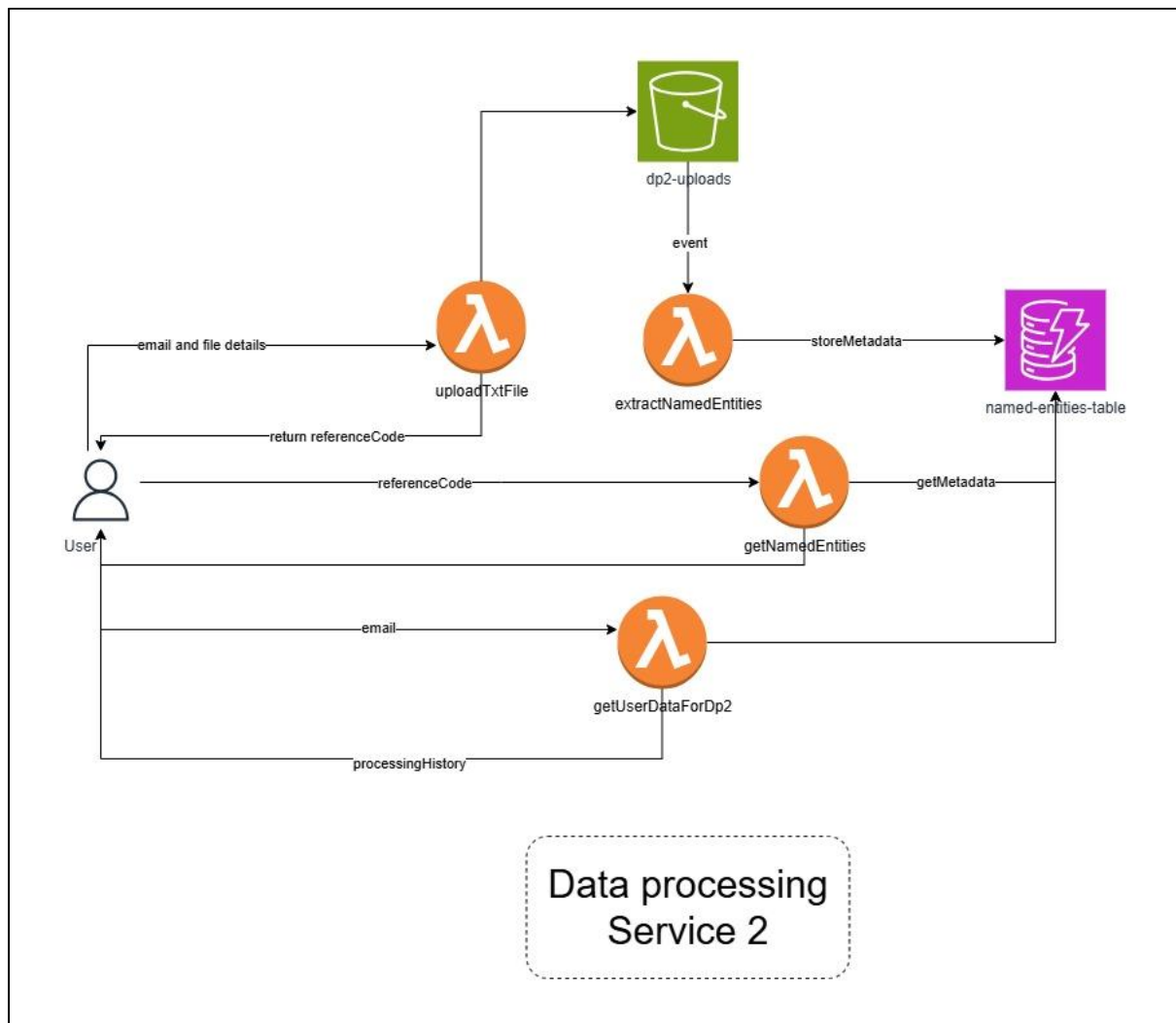


Figure 15 Data Processing Service 2 Architecture Diagram

Implementation:

File Upload (uploadTxFile Lambda):

- Similar to DP1, users upload text files through the frontend. The Lambda encodes the file in **Base64**, validates its format, stores the file in S3 bucket and also stores metadata in DynamoDB for tracking.

Entity Extraction (extractNamedEntities Lambda):

An S3 event triggers this Lambda when a text file is uploaded to the input bucket.

- The function uses the **SpaCy NLP model** (added as a Lambda layer) to analyze the text and extract named entities.
- Extracted entities, along with metadata like entity type (e.g., Person, Organization), are stored in DynamoDB for further use.

Fetching Results (getNamedEntities Lambda):

Users can retrieve the processed results, including the extracted entities, their types, and associated metadata.

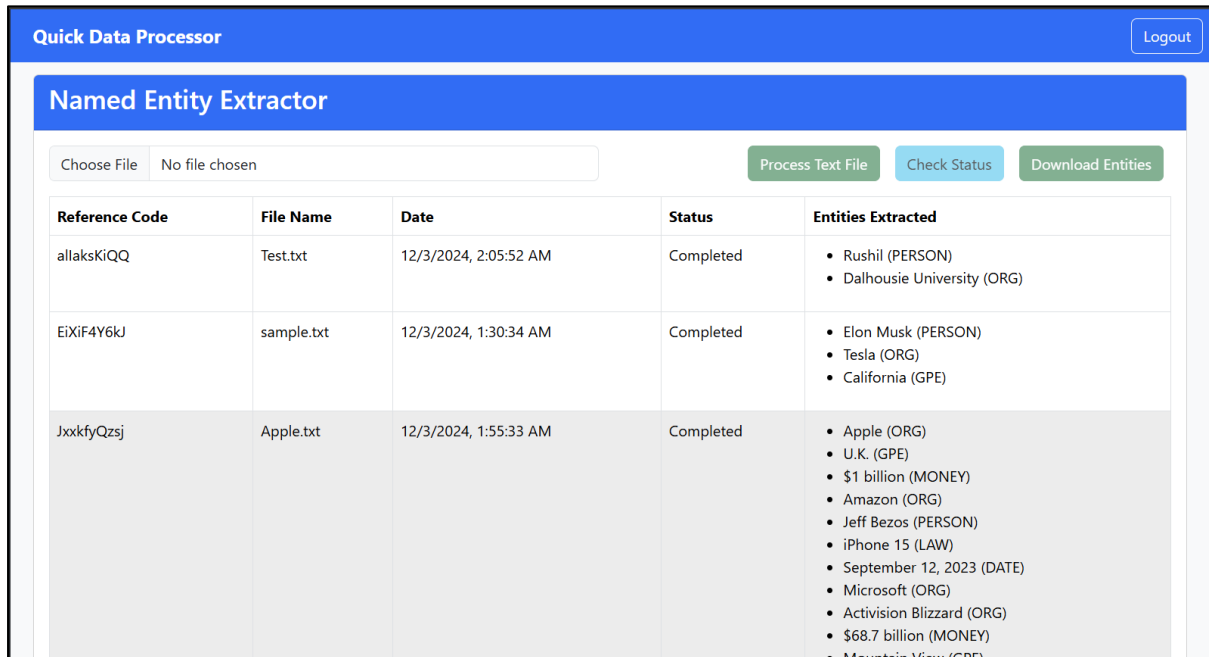
User History (getUserDetailsForDp2 Lambda):

This Lambda provides users with a history of their processed text files, including extraction results, for easy reference in the frontend.

Testing and Validation:

Various text files were tested to evaluate the accuracy of named entity extraction. Files with diverse structures and languages were used to ensure SpaCy's capabilities were leveraged effectively.

Screenshots:



| Reference Code | File Name | Date | Status | Entities Extracted |
|----------------|------------|-----------------------|-----------|--|
| allaksKiQQ | Test.txt | 12/3/2024, 2:05:52 AM | Completed | <ul style="list-style-type: none">• Rushil (PERSON)• Dalhousie University (ORG) |
| EiXiF4Y6kJ | sample.txt | 12/3/2024, 1:30:34 AM | Completed | <ul style="list-style-type: none">• Elon Musk (PERSON)• Tesla (ORG)• California (GPE) |
| JxxkfyQzsj | Apple.txt | 12/3/2024, 1:55:33 AM | Completed | <ul style="list-style-type: none">• Apple (ORG)• U.K. (GPE)• \$1 billion (MONEY)• Amazon (ORG)• Jeff Bezos (PERSON)• iPhone 15 (LAW)• September 12, 2023 (DATE)• Microsoft (ORG)• Activision Blizzard (ORG)• \$68.7 billion (MONEY)• Mountain View (GPE) |

Figure 16

Data Processing 3: Word Cloud using Looker Studio

This module streamlines the process of generating word clouds from uploaded text files, leveraging Google Cloud Platform (GCP) services for efficient automation and visualisation.

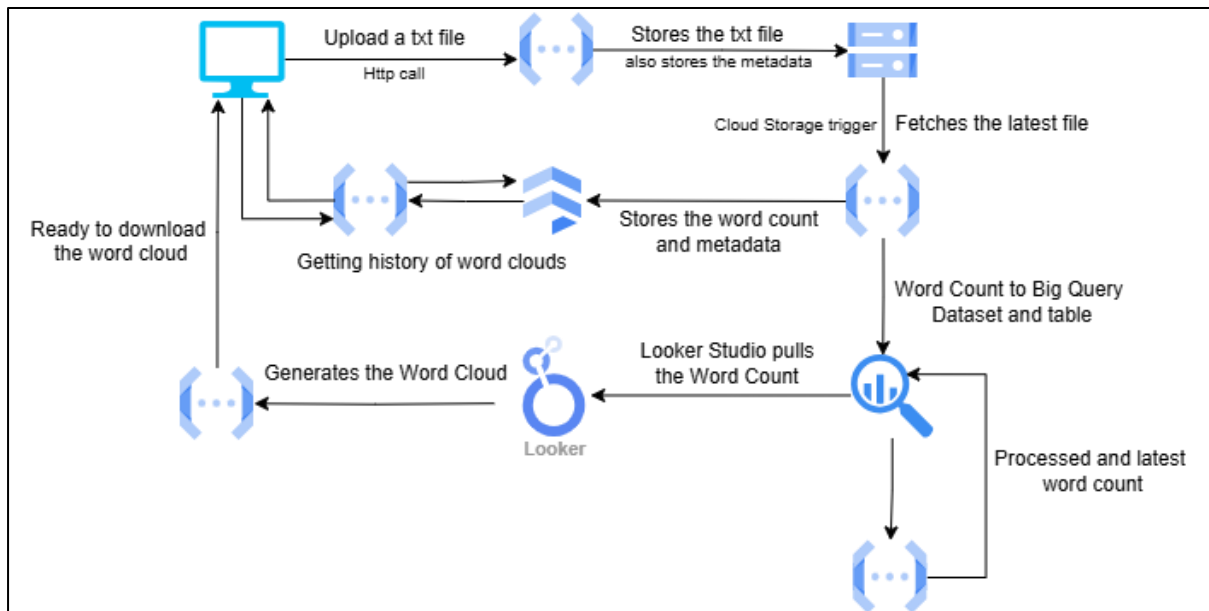


Figure 17 Data Processing Service 3 Architecture Diagram

Process Flow

- File Upload: Users upload a .txt file to the system.
- File Storage: The uploaded file, along with its metadata, is securely stored in Cloud Storage.
- Cloud Storage Trigger: A trigger is set to monitor for new file uploads. When a new file is detected, the system initiates the word count function.
- Word Frequency count function: The word frequencies are used to generate frequency data for each word.
- Word Count to BigQuery: The word count data is transferred to a BigQuery dataset and table for further analysis and reporting.
- Looker Studio Integration: Looker Studio is configured to pull the word count data from BigQuery.
- Word Cloud Display: The generated word cloud is displayed from Looker Studio, providing an interactive visualization of the text content.

Future Considerations

- Advanced Text Analysis: Incorporate advanced text analysis techniques like sentiment analysis or topic modelling to extract deeper insights from the text data.

- Machine Learning Integration: Utilize machine learning models to automatically categorize and classify text content, enhancing the word cloud generation process.

Word Cloud Generator

Choose File

example.txt

Generate Word Cloud

Check Status

Download

Processing History

| File Name | Upload Date | Reference Code | Status |
|----------------------|-------------|----------------|--------|
| No history available | | | |

Figure 118 Word Cloud frontend

Word Cloud Generator

Choose File

No file chosen

Generate Word Cloud

Check Status

Download

Processing History

| File Name | Upload Date | Reference Code | Status |
|----------------------|-------------|----------------|--------|
| No history available | | | |

Figure 129 Word cloud file upload

Pseudocode:Frontend-file-handling-function

- Use CORS middleware to handle the request.
- Check if the request is a preflight OPTIONS request:
 - If yes, set CORS headers to allow methods (GET, POST, OPTIONS) and respond with a 204 status.
 - Validate the request method:
 - If it is not POST, return a 405 status with an error message.
 - Check for Authorization header in the request:
 - If missing or invalid, return a 401 status with an error message.
 - Extract and decode JWT token from the Authorization header:
 - If decoding fails, return a 401 status with an error message.
 - Extract the user ID from the decoded token.
 - Validate the file data in the request body:
 - If missing or incomplete, return a 400 status with an error message.
 - Generate a unique file name using the user ID and timestamp.
 - Convert the file content to a buffer (UTF-8 encoded).
 - Access the storage bucket and prepare to upload the file:
 - Save the file with metadata, including the user ID.
 - Log successful upload with the file name.
 - Set CORS headers to allow all origins.
 - Respond with a 200 status and a success message.
 - Handle exceptions:
 - Log any errors encountered and respond with a 500 status and error message.

Pseudocode: Word-count -function pseudocode

Define Cloud Function handler (processFile):

- Log the start of file processing with the file name and bucket name from the event.

- Retrieve file and metadata from the source bucket:

 - If the metadata does not contain a user ID, log an error and exit.

- Download file content:

Parse content and compute word counts by splitting text and counting occurrences of each word.

- Prepare BigQuery table for the user:

 - Generate a sanitized table name based on the user ID.

- Check if the table exists in the BigQuery dataset:

 - If the table does not exist, create it with the required schema.

- Insert word count data into the BigQuery table:

 - Create rows containing the reference code, word, and count.

 - Insert rows into the user's table.

- Save processed data to the destination bucket:

Save the word counts as a JSON file in the destination bucket with a modified file name.

Log completion of file processing.

- Handle exceptions:

 - Log any errors encountered during the process.

Module 6: Data Analysis

This module is designed to provide user feedback visualization, sentiment analysis, and administrative insights. The module will serve various user roles, including QDP Agents, Guests, and Customers, each with different functionalities. The module integrates several storage solutions and utilizes the Natural Language API of Google Cloud for automated sentiment analysis[1].

Implementation Details:

User Features:

- **QDP Agents:**
 - Access an admin page with login statistics and total user counts, presented through a Looker Studio dashboard embedded directly into the frontend[2].
 - Gain insights into user activity to improve operational decision-making.
- **Guests, Customers, and QDP Agents:**
 - View customer feedback related to the data processing experience in a structured tabular format within the frontend interface.
 - Feedback is dynamically retrieved from a backend data store, which stored in DynamoDB ensuring flexibility and scalability[3].
- **Feedback Analysis:**
 - Feedback collected from users is automatically analyzed using Google Natural Language API to determine sentiment (e.g., positive, neutral, negative)[1].
 - Sentiment results are displayed on the frontend for all user types, offering real-time insights into customer experiences.

Testing and Validation:

- **Functional Testing:** Tested the retrieval and display of feedback from multiple backend storage solutions.
- **Validation Testing:** Ensured that feedback sentiment is accurately classified by Google Natural Language API across diverse test cases (e.g., positive, negative, and ambiguous feedback)[1].

Screenshots:

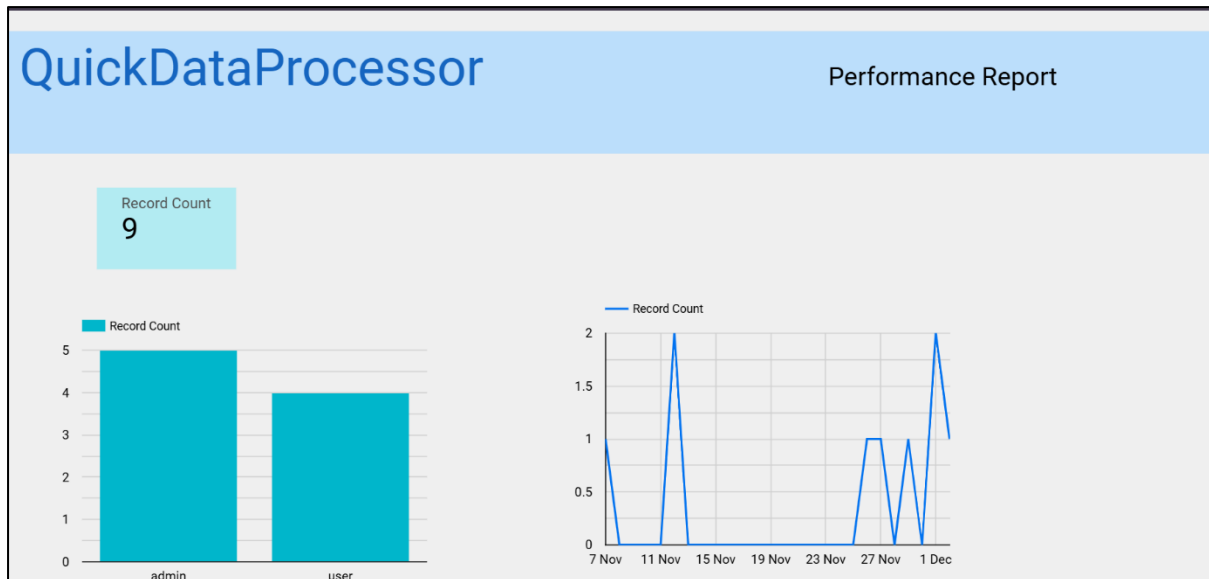


Figure 21 Dashboard(1)

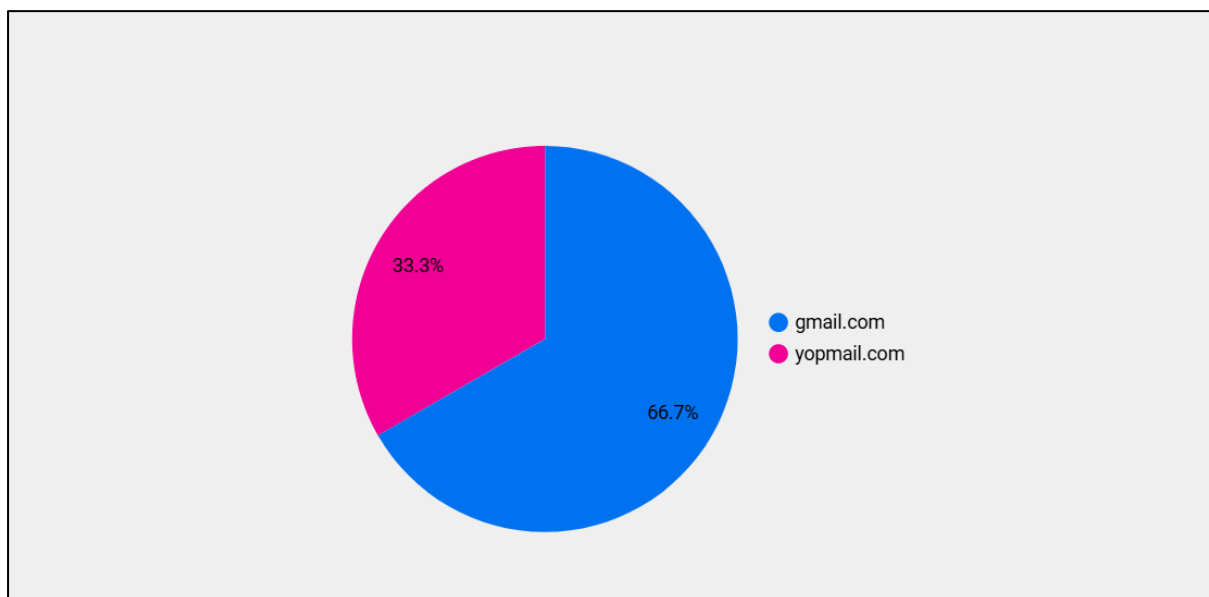


Figure 22 Dashboard(2)

Give Feedback

Customer Feedbacks

| User Email | Feedback | Polarity | Sentiment | Timestamp |
|-------------------------------|-------------------|----------|-----------|------------------------|
| mansikalathiya@gmail.com | hi | Positive | 0.30 | 11/30/2024, 8:32:30 PM |
| rushil@yopmail.com | Excellent service | Positive | 0.90 | 12/1/2024, 10:35:59 PM |
| mansikalathiya.1712@gmail.com | hey | Positive | 0.20 | 11/30/2024, 8:40:19 PM |

Figure 23 Give Feedback

Feedback

Write your feedback here...

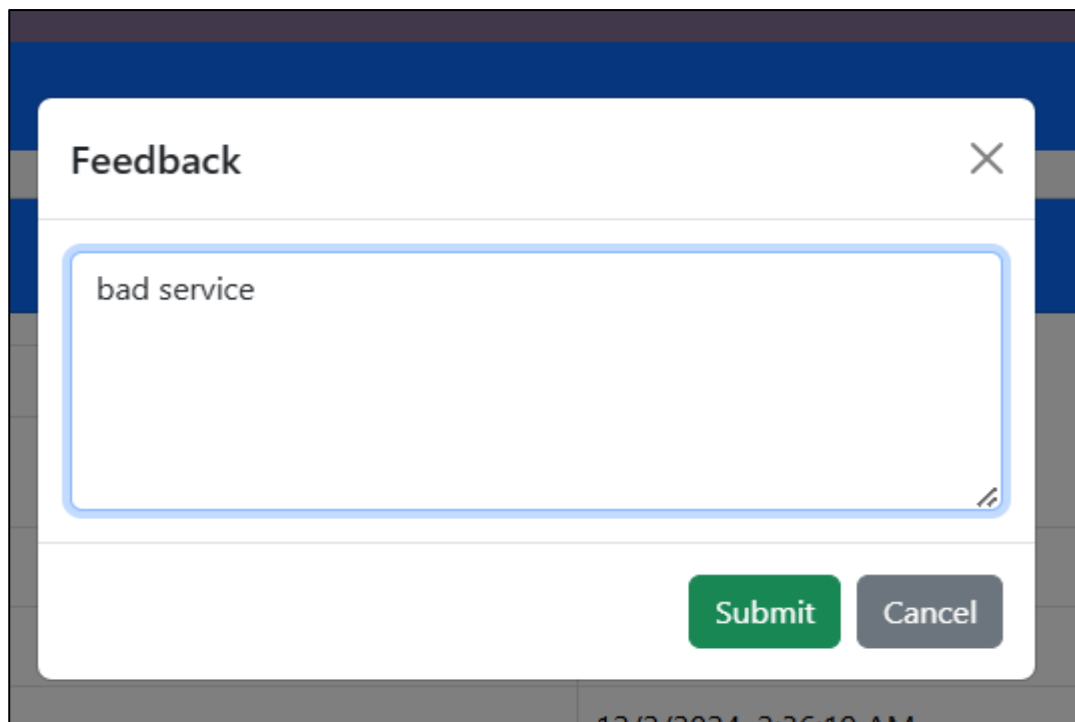
Submit

Cancel

12/2/2024, 2:36:19 AM

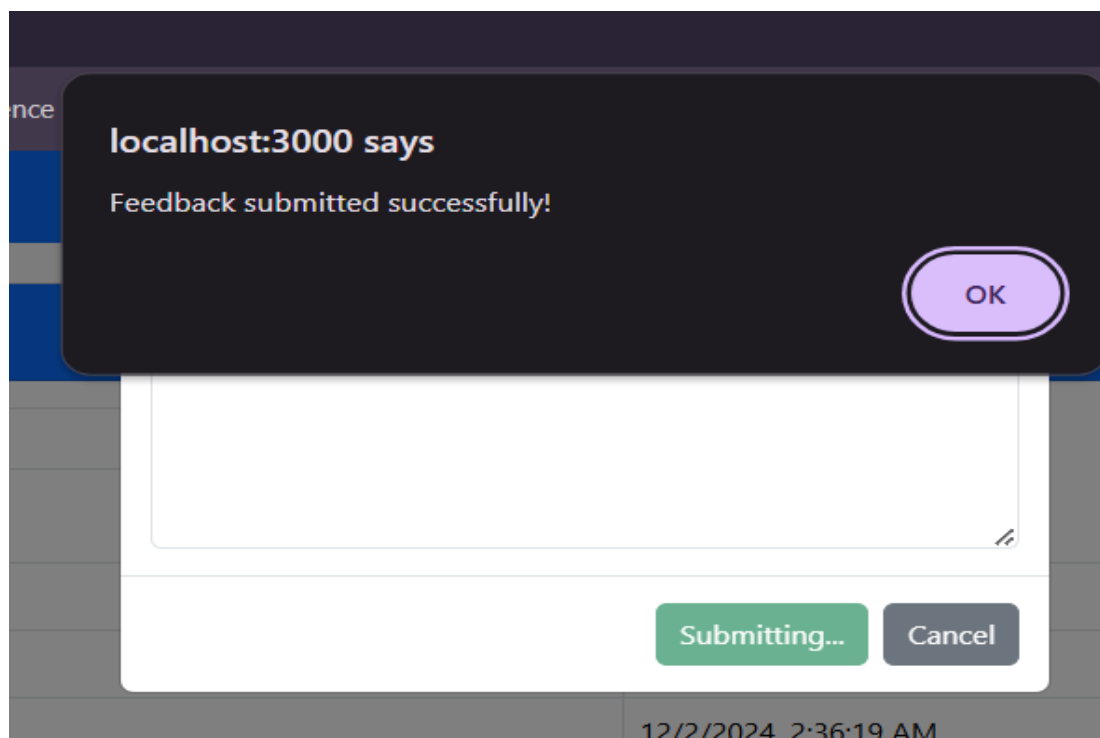
12/2/2024, 2:49:49 AM

Figure 24 Feedback Form(1)



A screenshot of a web application interface showing a 'Feedback' dialog box. The dialog box has a title bar with the word 'Feedback' and a close button (X). Inside the dialog, there is a text input field containing the text 'bad service'. Below the input field, there are two buttons: a green 'Submit' button and a grey 'Cancel' button. The background of the application is dark blue with a grey sidebar.

Figure 25: Feedback Form(2)



A screenshot of the same web application interface showing a confirmation dialog box. The dialog box is dark grey with a title bar that says 'localhost:3000 says'. The main text inside the dialog reads 'Feedback submitted successfully!'. There is a purple 'OK' button in the bottom right corner. Below the dialog box, the 'Feedback' form is visible, but the 'Submit' button is now disabled and labeled 'Submitting...'. The background of the application is dark blue with a grey sidebar.

Figure 26: Feedback submitted

Give Feedback

Customer Feedbacks

| User Email | Feedback | Polarity | Sentiment | Timestamp |
|-------------------------------|-------------------|----------|-----------|------------------------|
| mansikalathiya@gmail.com | hi | Positive | 0.30 | 11/30/2024, 8:32:30 PM |
| rushil@yopmail.com | Excellent service | Positive | 0.90 | 12/1/2024, 10:35:59 PM |
| mansikalathiya.1712@gmail.com | hey | Positive | 0.20 | 11/30/2024, 8:40:19 PM |
| mansikalathiya.1712@gmail.com | bad service | Negative | -0.90 | 12/2/2024, 10:25:55 AM |

Figure 27: Feedback table

Pseudo-code

1) getFeedback Function

- **Define Function fetchFeedback:**
 - **Input:** event
- **Initialize DynamoDB Resource:**
 - Create a DynamoDB resource object.
 - Reference the FeedbackTable.
- **Log Incoming Event:**
 - Print the incoming event for debugging purposes.
- **Fetch All Feedback Records from DynamoDB:**
 - Call the scan() operation on the FeedbackTable.
 - Store the result in response.
- **Extract Feedback Items:**
 - Assign feedback_items = response['Items'] to retrieve all feedback records.
- **Check for Errors in Response:**
 - If the scan() operation fails, catch the exception.
 - Log the error message for debugging.
 - Return a failure response with an appropriate error message.
- **Return Success Response:**
 - If feedback_items exist:
 - Return a response with statusCode: 200 and the feedback_items in the response body.
- **Return Failure Response (if applicable):**
 - If an exception is raised, return statusCode: 500 and an "Internal Server Error" message.

2) postFeedback Function

- Define Function `analyzeAndStoreFeedback`:
 - Input: `event`
- **Initialize DynamoDB Client:**
 - Create a `DocumentClient` instance for AWS DynamoDB.
- **Prepare Google Cloud Client:**
 - Check if the Google Cloud Language client is already initialized:
 - If not, initialize a `LanguageServiceClient`.
- **Set Google Service Account Environment:**
 - Define the temporary service account file path (`/tmp/service-account-file.json`).
 - Copy the service account file from the handler's directory to the temporary path.
- **Handle Errors in Service Account Setup:**
 - If copying the file fails:
 - Log the error and return a `500 Internal Server Error` response.

3) `getAllUsers` Function

- **Define Function `fetchAllUsers`**
 - Input: `event`, `context`
- **Setup Decimal Serialization Helper:**
 - Define a function `decimal_serializer` to handle `Decimal` objects:
 - Convert to `int` if the value has no fractional part.
 - Convert to `float` otherwise.
 - Raise `TypeError` for unsupported types.
- **Initialize DynamoDB Resource:**
 - Use `boto3.resource('dynamodb')` to initialize DynamoDB.
 - Reference the user table using `dynamodb.Table('user')`.
- **Scan the DynamoDB Table:**
 - Call `table.scan()` to retrieve all items.
 - Extract items from the response using `response.get('Items', [])`.
- **Handle Empty or Missing Data:**
 - Ensure data defaults to an empty list if no items are returned.
- **Return Successful Response:**
 - Return a `200 OK` status code.
 - Serialize data using `json.dumps`, with `decimal_serializer` for `Decimal` types.
- **Catch Exceptions:**
 - Wrap the scan operation in a try-except block.
 - If an error occurs:
 - Log the error.
 - Return a `500 Internal Server Error` response with the error message.

Module 7: Web Application Building and Deployment

Objective:

In this module we created a front-end application using **React** that connects to all backend services from cloud and then deployed the frontend application using **GCP Cloud Run**[10] with a CI/CD pipeline to automate the process using the **GCP Cloud Build**[11].

Frontend Development:

- Started by building a React application as the user interface. Each service, like JSON to CSV Converter or Word Cloud Generator, has its own route for better organization and navigation.
- Used **React Router** for managing navigation and built reusable components for consistency. The design ensures users can easily interact with backend services, maintaining simplicity and clarity in the interface.

Containerization:

To make the app deployment ready, wrote a **Dockerfile** that containerized the React application[9].

Used a multi-stage build to minimize the final image size. This optimization was crucial for better performance in the Cloud Run environment.

CI/CD Pipeline with Cloud Build:

The next step was automating the deployment. We connected the **GitLab repository** (mirrored to GitHub) to GCP Cloud Build and set up a **push trigger**[11].

- The **cloudbuild.yaml** configuration file defines the build steps, including creating a new container image on every push and fetching and deploying it to Cloud Run[10].
- This automation means that every push to the repository triggers the build and deploys the latest version of the app.

Hosting on GCP Cloud Run:

Once the pipeline was in place, the app was deployed to **Cloud Run**. This fully managed service scales automatically based on traffic, so there's no need to worry about provisioning or maintaining infrastructure.

Cloud Run also provides HTTPS by default, which added an extra layer of security without additional configuration.

Validation and Testing:

After deployment, the app was tested to ensure that all features worked as expected. This included validating the user interface, confirming the backend service integrations, and checking the CI/CD pipeline for smooth updates. Any bugs I found were resolved iteratively during testing.

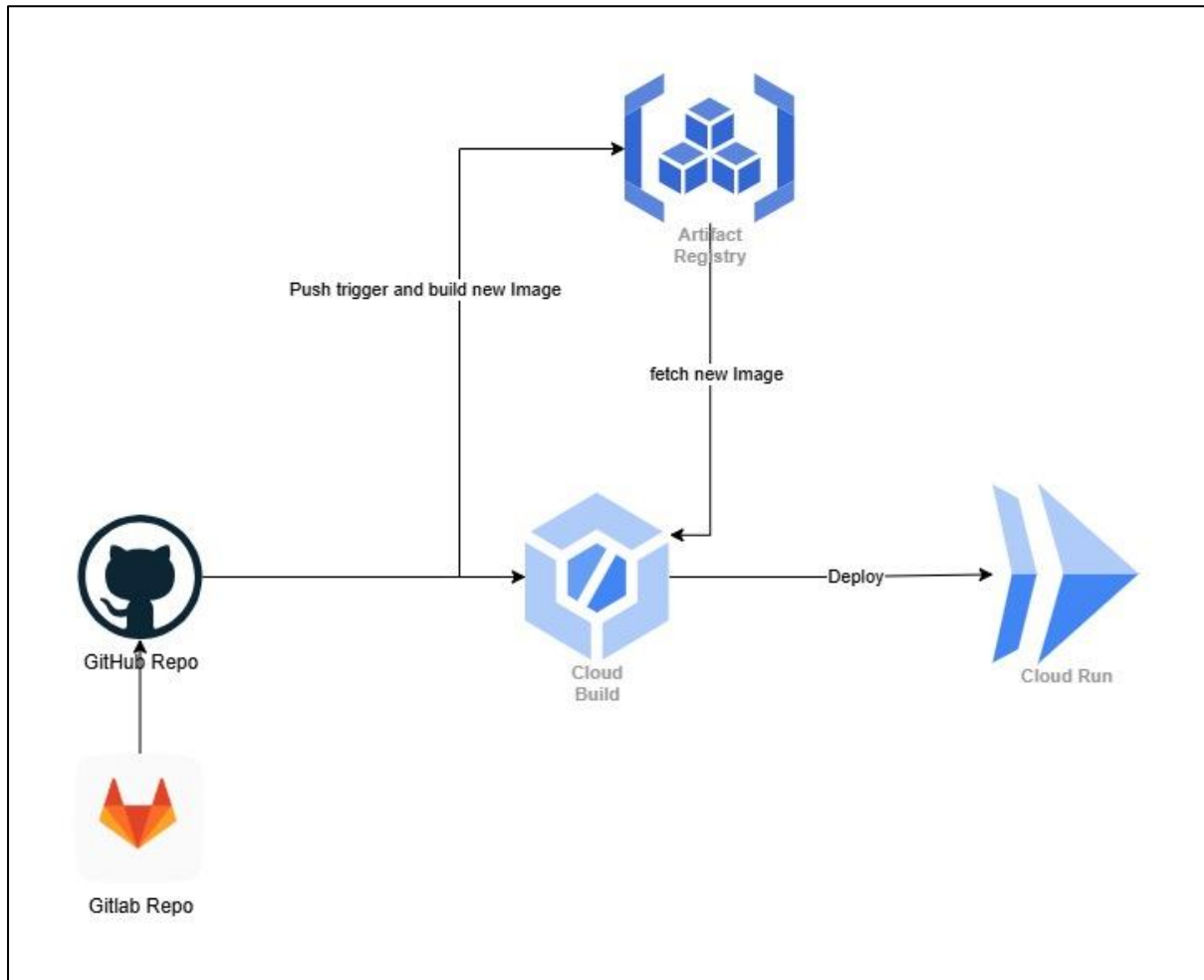


Figure 29 CI/CD Workflow for Cloud Run Deployment

Deployed URL:

<https://frontend-service-349910712067.us-central1.run.app/>

Gantt Chart

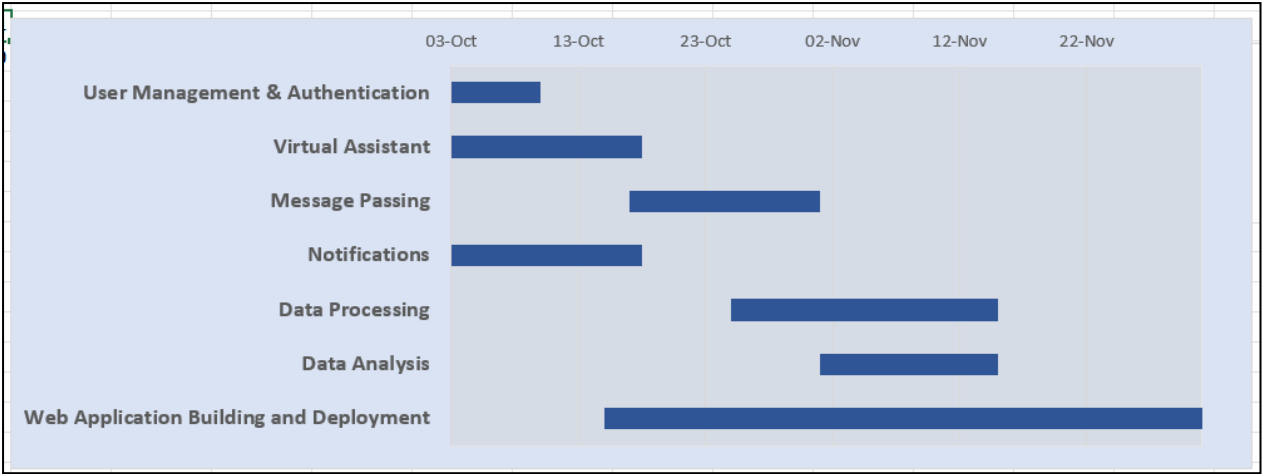


Figure 30 Gantt Chart

Conclusion

The successful project represents a very fruitful implementation of the serverless architecture using both AWS and GCP services. The developed system is robust, efficient, and scalable by using AWS Glue for processing data, the Google Natural Language API for sentiment analysis, and dynamic dashboards using GCP Looker Studio. The project ensures that the multi-factor authentication is secure, automates data processing tasks, and provides real-time notifications and valuable insights. Though challenges like real-time data synchronization and schema refinements were present, collaborative efforts led to an end-to-end solution meeting functional requirement, ensuring scalability and ease of maintenance, enhancing user experience. This project shows the power of cloud technologies in building innovative, serverless solutions that unlock real-time insights and drive operational efficiency.

References

- [1] Google Cloud, "Natural Language API," [Online]. Available: <https://cloud.google.com/natural-language>. [Accessed: 02-Dec-2024].
- [2] Google, "Looker Studio," [Online]. Available: <https://lookerstudio.google.com>. [Accessed: 02-Dec-2024].
- [3] Amazon.com, "NoSQL Database Service," [Online]. Available: <https://aws.amazon.com/dynamodb/>. [Accessed: 02-Dec-2024].
- [4] Amazon Web Services, "AWS Lambda," [Online]. Available: <https://aws.amazon.com/lambda/>. [Accessed: 02-Dec-2024].
- [5] Google Cloud, "Cloud Pub/Sub," [Online]. Available: <https://cloud.google.com/pubsub>. [Accessed: 02-Dec-2024].
- [6] Google Cloud, "Cloud Functions," [Online]. Available: <https://cloud.google.com/functions>. [Accessed: 02-Dec-2024].
- [7] M. Honnibal and I. Montani, "spaCy 3: Industrial-strength Natural Language Processing in Python," Explosion, 2020. [Online]. Available: <https://spacy.io>. [Accessed: Nov. 29, 2024].
- [8] Google Cloud, "Firestore," [Online]. Available: <https://cloud.google.com/firestore>. [Accessed: 02-Dec-2024].
- [9] Docker, "Docker Documentation," [Online]. Available: <https://docs.docker.com>. [Accessed: 02-Dec-2024].
- [10] Google Cloud, "Cloud Run," [Online]. Available: <https://cloud.google.com/run>. [Accessed: 02-Dec-2024].
- [11] Google Cloud, "Cloud Build," [Online]. Available: <https://cloud.google.com/build>. [Accessed: 02-Dec-2024].
- [13] GitLab, "GitLab Documentation," [Online]. Available: <https://docs.gitlab.com>. [Accessed: 02-Dec-2024].