Receipt Tracking App

Constantin Ana-Maria, Gălățianu Mihai,
Munteanu Denisa-Maria-Ionela, Țiplea Matei

March 31, 2025

Introduction

In this report, we present the architecture and implementation of a distributed receipt tracking application built using a hybrid approach: combining an on-premise system with multiple services from the Google Cloud Platform (GCP). The application processes receipts uploaded by users, extracts relevant data using AI services, and presents this data in a real-time web interface.

1 Overview of the Architecture

Our architecture, illustrated in Figure 1, consists of three main components:

- **Frontend Web Application**: React-based user interface hosted on-premise, providing upload functionality and real-time updates. Real-time receipt status messages are delivered via a local WebSocket server.
- **WebSocket Server**: Hosted on-premise, this server listens to Google Cloud Pub/Sub and relays receipt processing updates to connected frontend clients in real-time.
- **Backend API**: A FastAPI service that handles file uploads, communicates with cloud services, and exposes RESTful endpoints for frontend consumption.
- **Cloud Infrastructure**: GCP services including Cloud Storage, Firestore, Firebase Authentication, Cloud Functions, Pub/Sub, Cloud Vision API, and Gemini API (LLM).

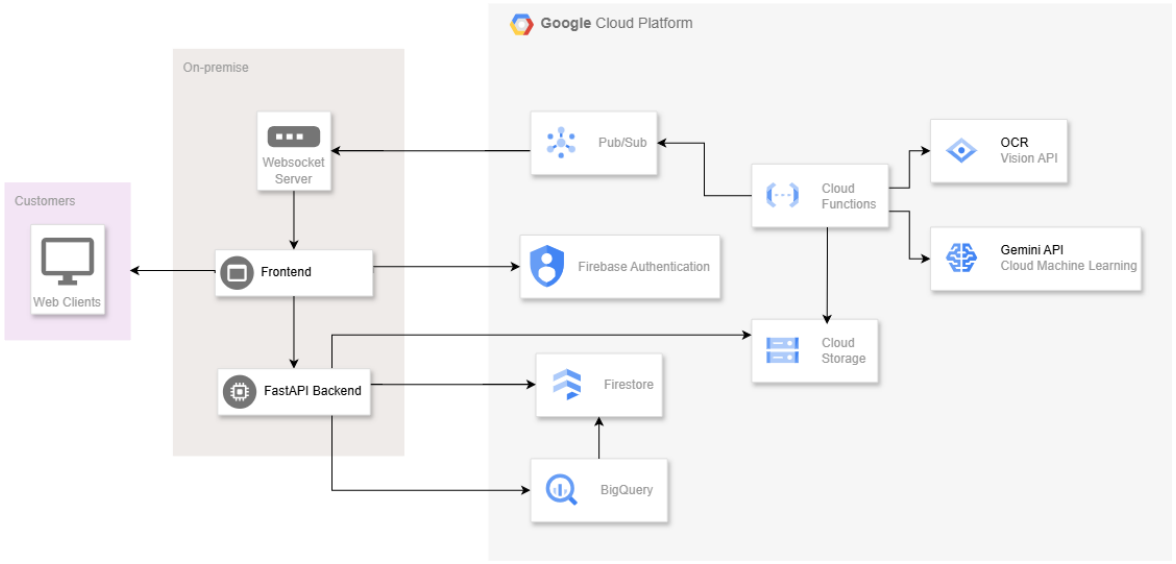

Figure 1: Architectural Diagram

2 Cloud Services Used and Their Roles

This section provides an in-depth explanation of the cloud-native services integrated into our system and highlights how each contributes to the receipt tracking pipeline.

2.1 Firebase Authentication

Firebase Authentication[1] is used to handle user sign-in on the frontend. The system is configured to support authentication exclusively through Google accounts. Upon successful login, the frontend retrieves the user's unique identifier (UID), which is stored locally and used throughout the system to:

- Organize uploaded files in user-specific folders in Cloud Storage.
- Filter receipt data queried from Firestore or BigQuery to return only the relevant records.
- Match WebSocket real-time updates from Pub/Sub based on the user_uid field in each message.

This service abstracts away the complexity of account management while ensuring secure and verified access for each user.

2.2 Cloud Storage

Google Cloud Storage (GCS)[2] serves as the initial ingestion point for user-uploaded receipt images. When a user uploads an image through the frontend, the FastAPI backend stores it in a Cloud Storage bucket under a folder named after the user's UID. This structured approach helps in associating uploaded images with individual users. Furthermore, GCS is configured to trigger a Cloud Function upon each successful image upload, initiating the serverless processing pipeline. GCS also serves as a static storage backend, providing persistent access to original receipt images via public URLs.

2.3 Cloud Functions

Google Cloud Functions[3] acts as the core of our event-driven architecture. A single function is triggered automatically whenever a new receipt image is added to Cloud Storage. The function orchestrates the following steps:

- OCR Invocation: Sends the image to Google Cloud Vision API to extract raw text from the receipt.
- Text Processing: Passes the OCR output to Google's Gemini 2.0 Flash large language model with a structured prompt to parse relevant fields such as store name, total amount, and categories.
- Database Insertion: Saves the extracted structured data into Firestore using a predefined schema.
- Event Publishing: After each significant step (OCR start, OCR finish, Gemini start, Gemini finish, success, or failure), the function publishes an update message to the Pub/Sub topic receipt-updates. These updates are critical for real-time client-side feedback.

This approach guarantees scalability, fault tolerance, and minimal operational overhead, all while leveraging a purely serverless paradigm.

2.4 Cloud Vision API

The Cloud Vision API provides Optical Character Recognition (OCR)[4] capabilities that transform image content into text. When the Cloud Function is triggered, it forwards the image to this API, which returns the full receipt text and associated metadata, including confidence scores and number of detected blocks. This API plays a vital role in transforming unstructured image data into machine-readable text that can later be interpreted and classified.

2.5 Gemini API (Large Language Model)

Gemini 2.0 Flash[5] is a foundation model provided by Google Cloud's AI suite. Once the OCR text is retrieved, the Cloud Function sends it to Gemini using a structured prompt. This prompt instructs the model to extract:

- Store name and address
- Date and time of purchase
- Total amount spent
- One or more categories chosen from a predefined list of 15 domain-specific expense categories

The model's output is a clean JSON object containing the desired fields. This enables the system to automatically categorize and structure receipt information with a high level of semantic understanding, avoiding the need for manually designed rule-based systems.

2.6 Firestore

Cloud Firestore[6] serves as the primary document database for storing receipt information. Each receipt is saved as a document in the receipts collection, with fields such as store_name, total_amount, date, user_uid, raw_text, and categories. Firestore's real-time syncing and fast querying capabilities make it ideal for a responsive application architecture. Data stored here is queried both by the backend (for analytics and reporting) and the cloud function (to populate the receipt table).

2.7 Pub/Sub

Pub/Sub[7] acts as the asynchronous messaging backbone of the system. It decouples the event producer (Cloud Function) from event consumers (e.g., the on-premise WebSocket server). Only one topic, receipt-updates, is used throughout the application. The Cloud Function publishes structured update messages as receipts progress through the pipeline. These messages include:

- Type of update (e.g., `receipt_update`)
- Current processing status (e.g., `processing`, `success`, `failed`)
- Descriptive message (e.g., `"Gemini processing started"`)
- Identifiers like receipt filename and user UID

On-premise, a dedicated Python WebSocket server subscribes to this topic and relays these updates to the frontend in real-time, enabling a rich, interactive user experience.

2.8 BigQuery

BigQuery[8] is used as the analytics backend of the system. Upon request from the frontend, the backend can query BigQuery to:

- Generate downloadable Excel reports for users containing receipt history (store, total, address, etc.)
- Create aggregated statistics, such as most-frequented stores or total spending by category

This allows users to gain insight into their purchasing patterns over time. The integration of BigQuery transforms the application from a simple receipt tracker into a meaningful financial visualization tool.

3 On-Premise Components

While the system is heavily integrated with cloud-native services, several critical components are hosted and executed on-premise. These components provide real-time communication, user interaction, and direct integration with the Google Cloud Platform via API calls and Pub/Sub messages.

3.1 FastAPI Backend

The backend is built with FastAPI[9], a modern Python web framework designed for high-performance APIs. It serves as the intermediary between the frontend and Google Cloud services. Its responsibilities include:

- Handling file uploads and routing them to Google Cloud Storage under the correct user folder.
- Exposing RESTful API endpoints that the frontend uses to retrieve the list of processed receipts from Firestore, filtered by user UID.
- Triggering analytical queries to BigQuery and generating downloadable Excel reports with receipt history and statistics.
- Serving as a local interface for testing and debugging the receipt processing workflow before it reaches the cloud.

By keeping this API layer on-premise, we retain development flexibility while maintaining secure interactions with cloud components through credentials configured via Google Application Default Credentials (ADC).

3.2 WebSocket Server

A custom-built WebSocket server runs continuously on-premise and serves as a real-time communication bridge between the Google Cloud Pub/Sub system and the React frontend. Its responsibilities are:

- Subscribing to the receipt-updates Pub/Sub topic using the Google Pub/Sub Python client.
- Listening for updates published by the Cloud Function regarding the status of receipt processing (e.g., "OCR started", "No text extracted from image, Gemini processing skipped", or "Receipt processed successfully").
- Broadcasting these updates to all connected frontend clients over WebSocket in JSON format.
- Filtering messages to ensure each user only receives relevant updates based on their UID.

This component enables the system to achieve low-latency, server-push updates to the UI, fulfilling the project's requirement for a real-time web technology. Its modular design allows it to be deployed independently and scaled or containerized as needed.

3.3 React Frontend

The user interface is a single-page application (SPA) built with React[10]. It provides users with the following capabilities:

- Upload receipt images using a user-friendly UI.
- View a dynamic, paginated table of all past receipts, including store details, date, total amount, and a link to view the original image.
- Receive real-time notifications about the receipt processing status using a WebSocket connection and Ant Design's notification component.
- Download Excel reports of receipt history and view summarized statistics.

Upon login (via Firebase Authentication), the user's credentials are stored in local storage, and their UID is used to:

- Route the uploaded images to the correct folder in Cloud Storage.
- Filter the receipts displayed in the frontend.
- Match Pub/Sub real-time messages to the correct session.

The React app provides a modern and intuitive experience that abstracts the complexity of the backend logic and cloud integration, allowing users to interact with the system effortlessly.

4 Performance Evaluation

To validate the reliability and responsiveness of our distributed system, we utilized Google Cloud Monitoring[11] to track performance metrics across key components. This section summarizes our findings based on metrics related to execution time, resource usage, and user activity.

4.1 Cloud Function Execution Times

Figure 2 shows the average execution time of the Cloud Function throughout the day. In most cases, it remained below 6 seconds, with only a few spikes likely caused by cold starts or external API latency. The results indicate reliable performance even during periods of higher activity.

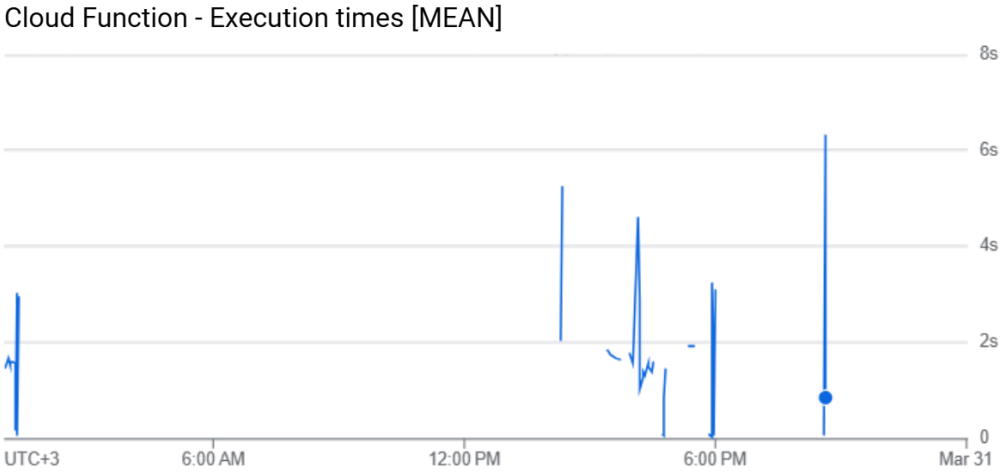

Figure 2: Cloud Function - Execution Time (Mean) - Sample 1

4.2 Pub/Sub Message Publishing

The latency in publishing messages to the Pub/Sub topic (Figure 3) remained consistently low, indicating effective message propagation to the WebSocket server. This helped preserve the system's real-time characteristics.

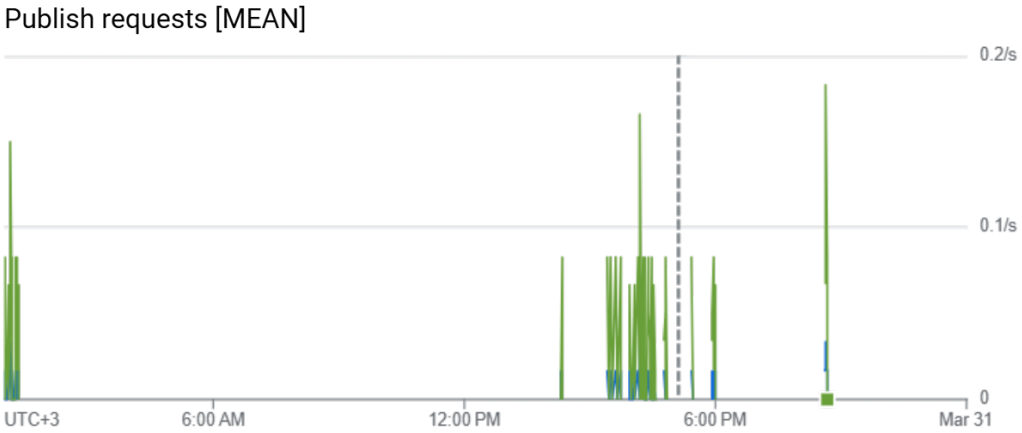

Figure 3: Pub/Sub - Publish Requests (Mean)

4.3 Database Performance

Firestore database request latencies remained below 0.06 seconds at the 50th percentile (Figure 4). This confirms that our backend and Cloud Function interactions with the database were efficient, even under moderate load.

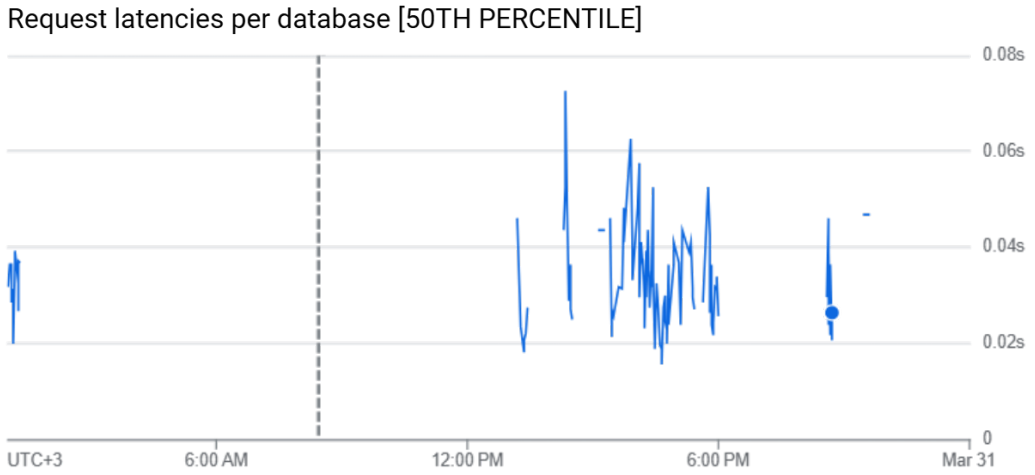

Figure 4: Firestore - Request Latencies (50th Percentile)

4.4 Authentication Metrics

Figure 5 shows the volume of service account authentication events, reflecting successful and secure access to GCP resources, including Cloud Functions and Firestore, using the Firebase identity tokens received during frontend sign-in.

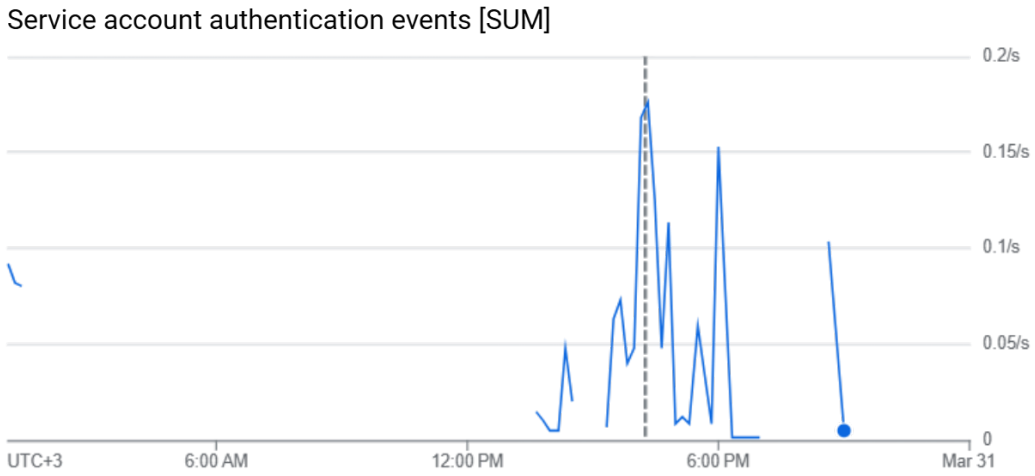

Figure 5: Service Account Authentication Events

4.5 BigQuery Log Activity

BigQuery logs (Figure 6) indicate active usage from the backend when generating downloadable reports. These logs validate that data queries and exports are correctly triggered via the backend on user request.

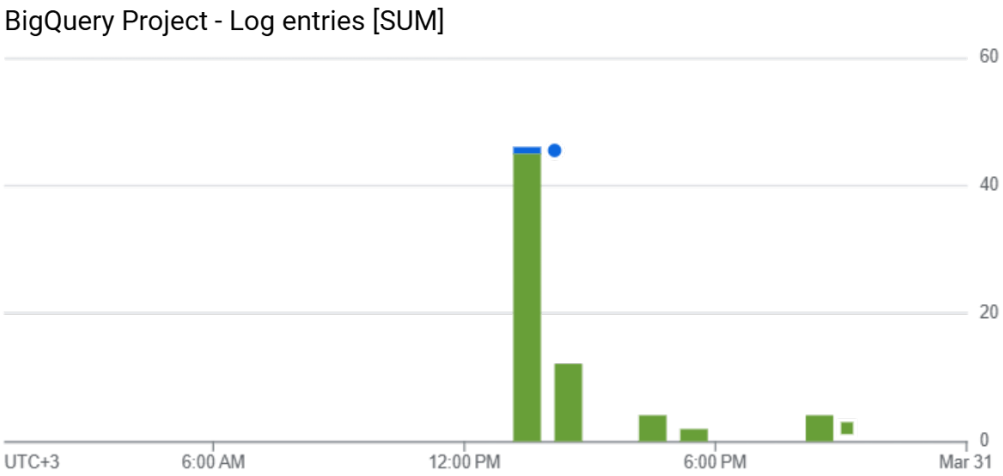

Figure 6: BigQuery - Log Entries (Sum)

4.6 Resource Utilization

To ensure the application operates efficiently under deployment, we tracked CPU and memory utilization of the backend containers:

- Figure 7 shows CPU utilization percentiles, which mostly stayed below 20%, suggesting available headroom for scaling.
- Figure 8 displays memory usage remaining under 80% during peak activity, indicating no critical memory pressure.

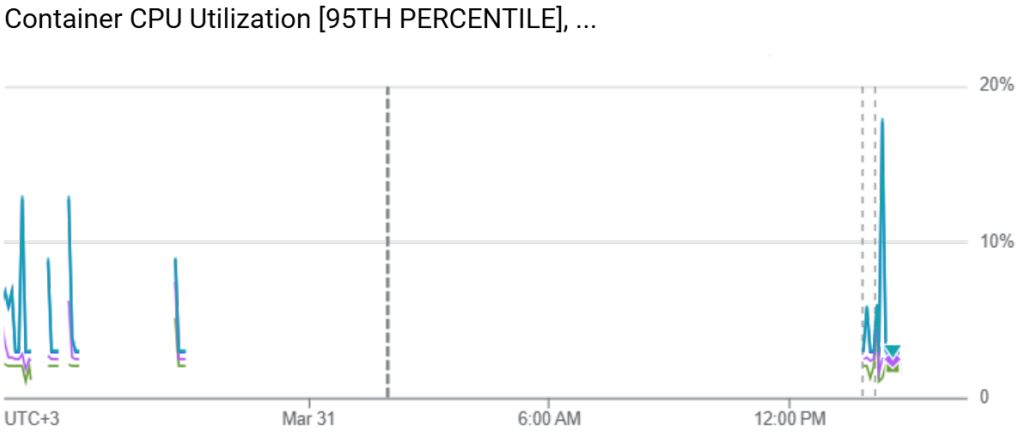

Figure 7: Container CPU Utilization (95th Percentile)

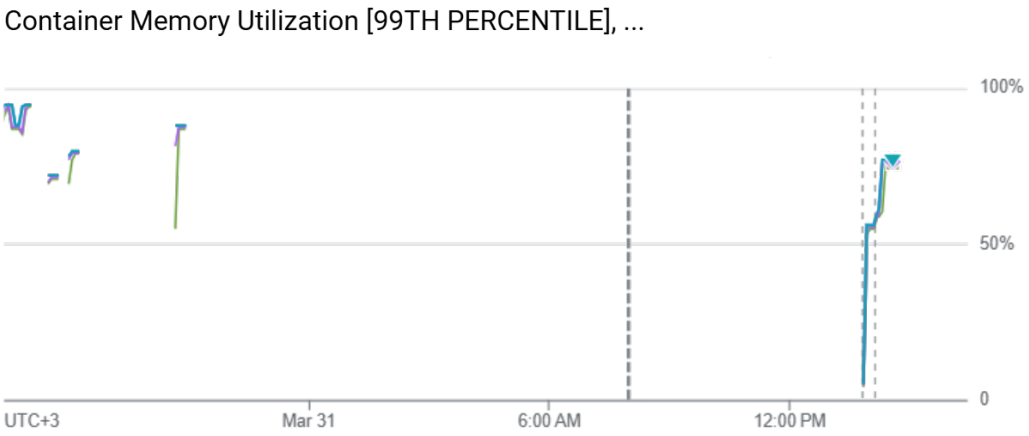

Figure 8: Container Memory Utilization (99th Percentile)

4.7 Summary

The metrics presented here confirm that our architecture is capable of handling concurrent processing and real-time interactions. With low-latency database and messaging operations, minimal execution times, and stable resource utilization, our system meets the performance expectations for a modern distributed cloud-native application.

Conclusion

This project demonstrates how a real-time, cloud-integrated distributed application can be built using Google Cloud Platform services. By combining cloud-native functions with on-premise components, the system achieves both flexibility and performance, reflecting architectural patterns seen in modern large-scale applications.

References

[1] *Firebase Authentication*. URL: https://firebase.google.com/docs/auth.

[2] *Google Cloud Storage*. URL: https://cloud.google.com/storage?hl=en.

[3] *Cloud Run Functions*. URL: https://cloud.google.com/functions?hl=en.

[4] *OCR (Optical Character Recognition) with world-class Google Cloud AI*. URL: https://cloud.google.com/use-cases/ocr?hl=en.

[5] *Start building with Gemini 2.0 Flash and Flash-Lite*. URL: https://developers.googleblog.com/en/start-building-with-the-gemini-2-0-flash-family/.

[6] *Cloud Firestore*. URL: https://firebase.google.com/products/firestore.

[7] *Pub/Sub*. URL: https://cloud.google.com/pubsub?hl=en.

[8] *From data warehouse to a unified, AI-ready data platform*. URL: https://cloud.google.com/bigquery?hl=en.

[9] *FastAPI*. URL: https://fastapi.tiangolo.com/.

[10] *React: The library for web and native user interfaces*. URL: https://react.dev/.

[11] *Cloud Monitoring*. URL: https://cloud.google.com/monitoring?hl=en.