Banking Assistant Conversational Agent - Cloud Function Documentation

1. Introduction

The Banking Assistant Conversational Agent is a cloud-native application designed to provide secure and scalable access to customer data stored in CSV format on **Google Cloud Storage (GCS) and in a spanner database**. The application is implemented as a **Google Cloud Function** and exposes an HTTP endpoint for querying customer records by their unique identifier.

This solution is ideal for lightweight banking systems, data lookups, and integration with conversational agents or front-end applications requiring quick access to customer records.

---

2. System Architecture

The application operates on the following architecture:

- **Client Request**: A client sends a POST request with a `customer_id` in the request body.
- **Cloud Function**: The HTTP-triggered function processes the request, retrieves the CSV from GCS or a table from spanner , and searches for the corresponding record.
- **Data Access**: The function reads the data using python functions, filters it, and returns the result as a structured JSON response.
- **Storage**: The CSV file (`credit_data.csv`) is maintained in a GCS bucket (`csv_bank`) and also in table format at spanner, enabling centralized data access and updates.

---

3. Functional Description

3.1. credit_data.ipynb – Customer Data Lookup

The main functionality of this module is to search for a customer's data based on the provided `customer_id`. The Cloud Function performs the following steps:

- Validates the presence of `customer_id` in the incoming HTTP request.
- Connects to Google Cloud Storage (GCS) and downloads the `credit_data.csv` file.
- Loads the file into a Pandas DataFrame.
- Filters the records to find rows matching the given `customer_id`.
- Returns a structured JSON response with customer details or a descriptive error if no match is found.

3.2. Address_change.ipynb – Customer Address Update

This module handles the process of updating a customer's address securely. The Cloud Function follows these steps:

- Parses the incoming request and validates that both `customer_id` and `new_address` are provided.

- Invokes an external helper function (`update_customer_address`) from a backend module (e.g., `spanner_connector1.py`).
- Performs the update in the cloud database (such as Google Cloud Spanner or Firestore).
- Returns a JSON message indicating whether the update was successful or if an error occurred (e.g., customer not found).

3.3. check_balance.ipynb – Account Balance Inquiry

This module is responsible for verifying a customer's identity and fetching their current account balance. The Cloud Function executes the following logic:

- Accepts a request containing `customer_id` and `pin`.
- Validates the input fields and checks the credentials using an authentication function (`authenticate_customer`).
- If authentication is successful, retrieves detailed customer information including account balance.
- Responds with a JSON payload containing the customer's financial data or an authentication failure message.

---

4. API Endpoint

4.1. credit_data.ipynb – Customer Data Lookup

- **Method**: `POST`
- **URL**: `https://us-central1-<project-id>.cloudfunctions.net/customerLookup`
- **Request Header**:
  `Content-Type: application/json`
- **Request Body**:

```
{
  "customer_id": "C001"
}
```

- **Success Response** (`200 OK`):

```
{
  "status": "success",
  "customer_id": "C001",
  "records": [
    {
      "CustomerID": "C001",
      "Name": "John Doe",
      "CreditScore": 750,
      "LoanAmount": 20000,
      "BalanceDue": 5000,
      "Status": "Active"
    }
  ]
}
```

- **Error Responses**:
  - `400 Bad Request`: Missing `customer_id`
  - `404 Not Found`: No matching record
  - `500 Internal Server Error`: Any unhandled error

4.2. Address_change.ipynb – Customer Address Update

- **Method**: `POST`
- **URL**: `https://us-central1-<project-id>.cloudfunctions.net/updateAddress`
- **Request Header**:
  `Content-Type: application/json`
- **Request Body**:

```
{
  "action": "update_address",
  "customer_id": "C001",
  "new_address": "123 Elm Street, Springfield"
}
```

- **Success Response** (`200 OK`):

```
{
  "success": true,
  "message": "Address updated successfully for customer C001."
}
```

- **Error Responses**:
  - `400 Bad Request`: Missing fields
  - `500 Internal Server Error`: Update failed due to backend error

4.3. check_balance.ipynb – Account Balance Inquiry

- **Method**: `POST`
- **URL**: `https://us-central1-<project-id>.cloudfunctions.net/checkBalance`
- **Request Header**:
  `Content-Type: application/json`
- **Request Body**:

```
{
  "action": "authenticate",
  "customer_id": "C001",
  "pin": "1234"}
```

- **Success Response** (`200 OK`):

```
{
  "authenticated": true,
  "customer": {
    "CustomerID": "C001",
    "Name": "John Doe",
    "CreditScore": 750,
    "BalanceDue": 5000,
    "LoanAmount": 20000,
    "Status": "Active"
  }
}
```

- **Failure Response** (`200 OK`):

```
{
  "authenticated": false,
  "message": "Invalid customer ID or PIN"
}
```

- **Error Responses**:
  - `400 Bad Request`: Missing authentication fields
  - `500 Internal Server Error`: System or logic error

---

5. Deployment

Each notebook corresponds to a Google Cloud Function and can be deployed using the `gcloud` CLI. Ensure you have the required files (`main.py`, `requirements.txt`, and any helper modules) in place before running the deployment command.

5.1. credit_data.ipynb – Customer Data Lookup

Function Name: `customerLookup`
Entry Point: `main`
Region: `us-central1`
Memory: `512MB`

Deployment Command:

```
gcloud functions deploy customerLookup \
  --runtime python310 \
  --trigger-http \
  --entry-point main \
  --allow-unauthenticated \
  --memory 512MB \
  --region us-central1
```

5.2. Address_change.ipynb – Address Update Handler

Function Name: `updateAddress`
Entry Point: `bank_agent_webhook1`
Region: `us-central1`
Memory: `512MB`

Deployment Command:

```
gcloud functions deploy updateAddress \
  --runtime python310 \
  --trigger-http \
  --entry-point bank_agent_webhook1 \
  --allow-unauthenticated \
  --memory 512MB \
  --region us-central1
```

5.3. check_balance.ipynb – Balance Authentication and Lookup

Function Name: `checkBalance`
Entry Point: `bank_agent_webhook1`
Region: `us-central1`
Memory: `512MB`

Deployment Command:

```
gcloud functions deploy checkBalance \
  --runtime python310 \
  --trigger-http \
  --entry-point bank_agent_webhook1 \
  --allow-unauthenticated \
  --memory 512MB \
  --region us-central1
```

> *Note*: If you're using the same `bank_agent_webhook1` function for both address update and balance check, make sure the `action` field in the request determines the appropriate flow.

---

6. Dependencies

Each Cloud Function requires a `requirements.txt` file to define its Python dependencies. Below are the libraries used across all three functions.

6.1. Shared Core Dependencies

These packages are common to all three functions:

```
pandas
google-cloud-storage
```

6.2. Additional Dependencies (for Address Update and Balance Check)

If you're using custom authentication or address update logic that interacts with a database like Cloud Spanner, Firestore, or similar, you may need:

```
google-cloud-spanner
```

If you're importing a custom module like `spanner_connector1.py`, ensure it is placed in the root of your Cloud Function folder during deployment.

6.3. Example requirements.txt

For `customerLookup` (credit_data.ipynb):

```
pandas
google-cloud-storage
```

For `updateAddress` and `checkBalance` (Address_change.ipynb & check_balance.ipynb):

```
pandas
google-cloud-storage
google-cloud-spanner
```

---

7. Error Handling

The application includes robust error handling to ensure graceful failure in the event of:

- Missing input fields
- Malformed JSON
- Missing or inaccessible GCS files
- CSV parsing issues
- Unhandled internal exceptions

---

8. Conversational Agent Integration Using Datastore-OpenAPI and Playbooks

8.1 Overview

The banking APIs have been integrated into a Conversational Agent Platform (e.g., Google Agent Builder or Dialogflow CX), using OpenAPI specifications to define intents and fulfillment behavior and instructions given in playbooks. This approach allows for dynamic user interaction while leveraging robust backend services.

8.2 Purpose of Integration

- To allow natural language queries (e.g., “What is my balance?”, “Update my address”) to trigger secure backend functions.
- To automate customer service and reduce the need for manual banking operations.
- To utilize Cloud storage and databases for real-time session management, storing authentication states, and caching interactions.

8.3 Architecture Highlights

- User Intent Recognition: The agent matches user inputs to predefined intents such as `check_balance`, `update_address`, or `lookup_customer`.

- OpenAPI Fulfillment: Each intent is connected to an OpenAPI-defined endpoint, allowing the agent to directly call Google Cloud Functions.
- Response Handling: The Cloud Functions return structured JSON responses, which are then parsed and transformed into human-like responses by the agent.
- Datastore Integration: Used to:
    - Store user session information.
    - Log conversation context (e.g., last authenticated customer ID).
    - Maintain persistent variables across sessions.

8.4 Advantages

- Separation of Concerns: Business logic remains in Cloud Functions, while conversational logic stays in the agent.
- Modularity: Each Cloud Function can evolve independently of the agent's design.
- Scalability: Both Cloud Functions and Datastore scale with user demand.
- Maintainability: Updating APIs through the OpenAPI spec automatically reflects in the agent integration.
- Security: Datastore can store session tokens or partial authentication results securely.

8.5 Flow Example

User: "What is my balance?"
Agent:

1. Recognizes the `check_balance` intent.
2. Calls the OpenAPI-bound `checkBalance` function with `customer_id` and `pin`.
3. Receives JSON from the Cloud Function.
4. Parses and formats the response into:
   "Your current balance is $5,000. Your loan amount is $20,000."

---

9. Conclusion

This application suite demonstrates the effective use of **Google Cloud Functions and Database** to implement serverless, scalable banking operations, providing seamless integration with **cloud-based data storage** and **external APIs**. The three modular components—**Customer Data Lookup**, **Address Update**, and **Balance Inquiry**—each fulfill a distinct purpose while sharing a common architecture that emphasizes simplicity, security, and performance.

By leveraging **HTTP-triggered Cloud Functions**, **Google Cloud Storage**,**Spanner** and tools like **Pandas**, the solution avoids traditional server overhead while maintaining flexibility in how customer data is accessed, authenticated, and updated. This microservices-based design allows for easy expansion of additional services (e.g., transaction history, loan eligibility) as needed.

The application is cloud-native, easy to deploy using `gcloud`, and well-suited for chatbot integration, mobile banking apps, or customer service automation.