Project 2

Ibrahim Luke Barhoumeh
ibarhoumeh2022@fau.edu

GCP Project Name: ibarhoumeh-cot5930P1

GCP Project ID: ibarhoumeh-cot5930p1

GCP Project Console: PROJECT LINK

Cloud Run Service name: imaging-app

App URL:
https://imaging-app-856239458079.us-east1.run.app

GitHub Repository Link:lukebarhoumeh/COT5930-Project2: Project 2 for Cloud Native Development Class using GCP

Introduction

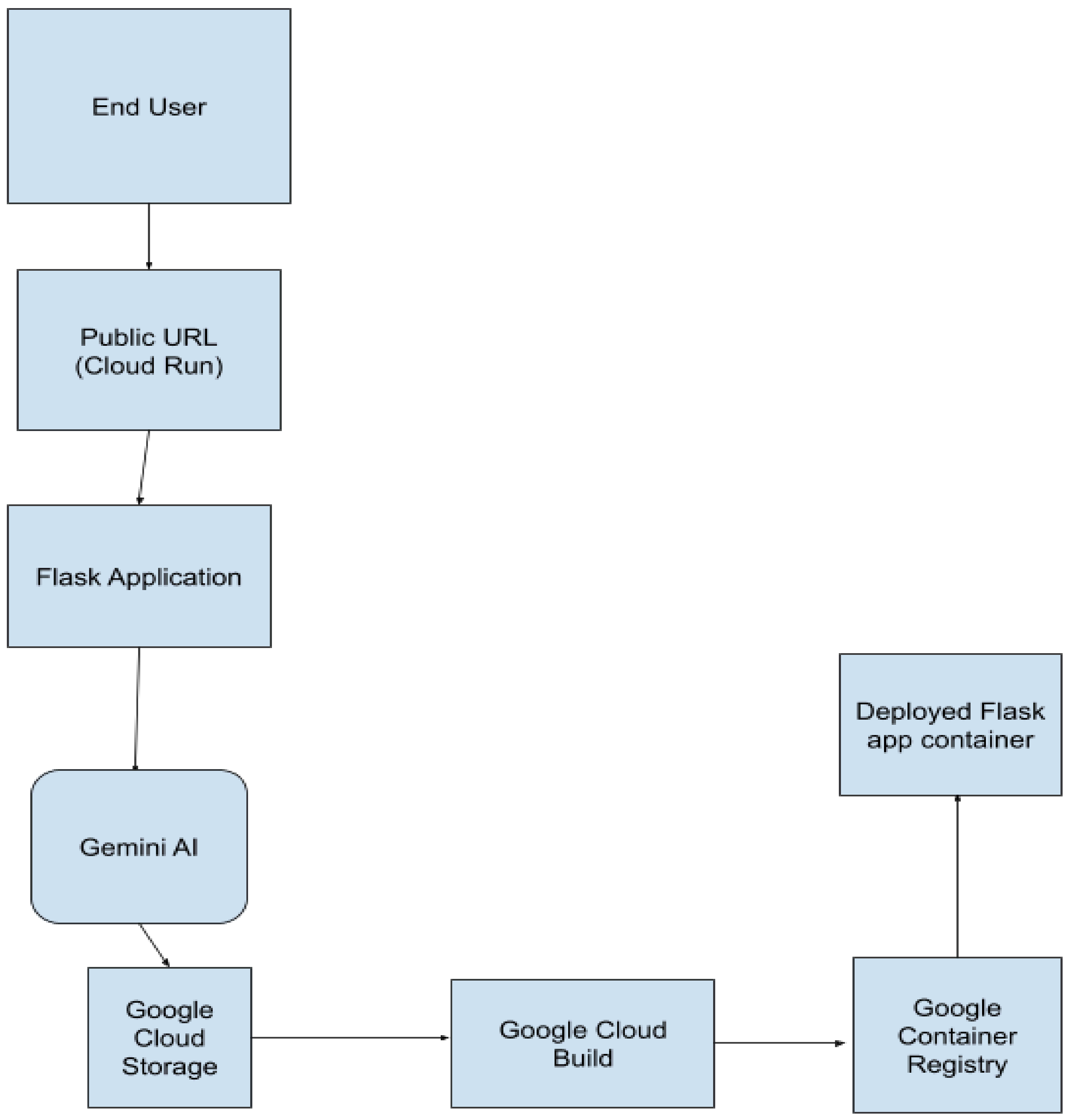
This project builds on the foundations of our first assignment—using Flask, Cloud Run, and Google Cloud Storage—to add an exciting new feature: image captioning with Gemini AI. Now, each time a user uploads a JPEG to our web app, we call Google's multimodal LLM (Gemini) for a brief description of the image. We store the resulting text in a JSON file alongside the image, ensuring it's easy to retrieve both the file and its automated caption from the same bucket. By keeping our bucket private and funneling all operations through the app, we preserve security and avoid exposing direct storage links. Meanwhile, using environment variables for the API key prevents hardcoding secrets into the code. The result is a serverless, scalable application that demonstrates how a lightweight Python/Flask service can tap into advanced AI capabilities—all deployed seamlessly on Cloud Run.

Project Requirements

1. **Leverage Gemini AI API (Multimodal LLM)**
   a. Must call googles gemini model to generate caption/description of image
2. **Invoke API at upload time only**
   a. Gemini call happens exactly once when the uder uploads new JPEG
3. **Stored Response in a .json file matching the image**
   a. For each myimage.jpeg must create and upload myimage.json into the same bucket location
4. **Security and privacy measures**
   a. Bucket must not be public
   b. App must not provide direct bucket URLS to end users
   c. No service account keys or other secret should appear in repo or codebase
5. **Deployment**
   a. Must be deployed via cloud run and be fully functional
   b. Provide the url for TA/Instructor
6. **No hardcoded secrets or keys**
   a. Gemini API key must not be stored in main.py
   b. Environment variables or secret manager approach should be used
7. **Code Repo and reporting**
   a. All code must be hosted in a shored or private repo
   b. Instructor and TA must have Viewer access to repo
   c. Final report should include
      i. The cloud console with viewer access granted
      ii. Cloud run service URL public
      iii. And GitHub REPO
8. **Submission guidelines**
   a. Any missing links, access, or info results in a automatic 0/10
   b. No late submissions

Architecture

The architecture for this project was set to leverage all of google cloud services and modern cloud native principals to ensure scalability and user accessibility.



Explanation of Each component

1. **End User**
    a. Interacts with a web interface to upload JPEG images
    b. Accesses the application through a public URL hosted on Cloud run
2. **Cloud Run**
    a. Hosts our containerized Flask App
    b. Automatically scales based on incoming traffic
    c. Ensures that no direct secrets (Ex. API keys) are stored in the containers code, relying on environment variables instead
3. **Flask APP**
    a. Receives the uploaded file from user via a POST request to the /upload route
    b. Calls Gemini AI endpoint with base64-encoded image to generate a short description
    c. Saves both image and JSON to GCS
4. **Gemini AI**
    a. Uses GenerateContent endpoint to handle images plus a short prompt
    b. The app sends a payload containing base64 image data and a short intro
    c. Gemini returns a textual description, which the app parses and saves in a .json file
5. **Google Cloud Storage (GCS)**
    a. Stores all uploaded images under the files/ prefix in a private bucket
    b. Saves .json files named after the image, each containing the title and description fields
6. **Deployment Pipeline**
    a. Dockerfile:defines how to build python/flask app into a container
    b. Requirements.txt: lists dependencies(flask,request, GCS, etc..)
    c. Cloud build: automate building and pushing docker image to Google Container Registry or Artifact registry
    d. Cloud Run: Deploys the built image container image, providing a public URL and Managing Scalability
7. **Environment Variables**
    a. Pass Gemini API key into Cloud Run as GENAI_API_KEY
    b. Prevents storing secrets in main.py
    c. Flaks code reads os.environ["GENAI_API_KEY"] at runtime, constructing the AI request endpoint securely

By using all of these components, the system provides a severless, secure, and scalable way to generate AI based captions for user uploaded images, without exposing bucket or Gemini API credentials.

Implementation /Code Structure

1. **main.py**
    a. This is the core Flask application that defines the routes:
        i. /: Displays a simple upload form and lists existing images in GCS.
        ii. /upload: Handles file uploads, calls Gemini AI for a short description, then stores both the image and the generated JSON in Google Cloud Storage.
        iii. /view/<filename>: Retrieves a specific image and its corresponding JSON description, showing them in a simple webpage.
        iv. /files/<filename>: Serves an individual file from local disk or downloads it from GCS if not present locally.
    b. Within main.py, the key function generate_image_description() reads the uploaded file from disk, base64-encodes it, and sends it to the Gemini API. The API's response is then parsed to extract a short description. That description is saved in a .json file that mirrors the image's name.
2. **storage.py**
    a. Contains helper functions to interact with Google Cloud Storage:
        i. Upload a file to the files/ folder in the bucket.
        ii. Download a file from the bucket to local storage.
    b. This keeps the code for GCS I/O separate and more maintainable.
3. **Requirements.txt**
    a. Lists all Python dependencies needed by the app:
        i. Flask for web server
        ii. Requests for making HTTP calls to the Gemini endpoint
        iii. Google-cloud-storage for GCS interactions
4. **Dockerfile**
    a. Describes how to containerize the application for Cloud-Run
        i. Start from python 3:10 slim base image
        ii. Installs the packages from requirements.txt
        iii. Copy all code into container
        iv. Expose port 8080 and run flask app with CMD
5. **Gemini Integration**
    a. Request Payload
        i. When a user uploads an image, generate_image_description() reads the file in binary, then encoded with Base64. Includes it in the JSON payload
    b. Response Parsing:
        i. Gemini typically returns array called "candidates", retvireved the first candidate and look for "content", which in turn has "parts" containing the generated text. This text is assembled into the final description that the app

stores

   c. **Storing “title” and “description”**
      i. Once text is retrieved the code writes out .json file
      ii. That Json file is uploaded to the same files/ prefix in GCS as the Original image, ensuring the two remain paired

6. **Security**
   a. Instead of embedding the Gemini API key directly into main.py, the app reads it from an environment variable called GENAI_API_KEY
   b. PRevents secrets from being checked into version control. Cloud Run is configured so that any new revisions pass this key at deploy time, keeping secret free

Deployment and Usage instructions:

1. **Steps to deploy on cloud run**
   a. **Build docker image**
      i. Navigate to project folder container dockerfile and run command gcloud build submit
      ii. Uploads to cloud build, which produces a container image stored in google container
2. **Set environment variable**
   a. Rather than hard coding Gemini API, pass it at deploy time using –update-env-vars GENAI_API_KEY (api key)
3. **Deploy Container to cloud run**
   a. Gcloud run deploy
      – image gcr
      – region
      –unauthenticated
      – update env vars

Running and Testing

1. **Upload an Image**
   a. Once you open cloud run URL, you'll see an upload form where you can select and upload a JPEG file
2. **View JSON output**
   a. After upload, the app calls Gemini AI to generate a short description
   b. Both the image and a json file are saved in GCS bucket
   c. Index page (/) lists any uploaded images by name, clicking on one shows the image and the generated caption

Pros and Cons

PROS:

1. **Serverless and Auto Scaling**
    a. Deploying on Cloud Run means the app scales automatically in response to traffic, removing the need for manual server management.
    b. If thousands of users upload images simultaneously, Cloud Run can spin up more container instances to handle the load.
2. **Secure Data Storage**
    a. Storing files in a private GCS bucket ensures only the application can read or write files, preventing unauthorized access.
    b. Moreover, environment variables are used to pass secrets (like the Gemini API key), avoiding hard coded credentials in the code.
3. **Simple integration of AI**
    a. The application's design uses a straightforward POST request to the Gemini API.
    b. This modular approach allows easy updates to AI endpoints or logic without major architectural changes.
4. **Easily Extensible**
    a. Because the logic is separated into functions for GCS interactions, AI calls, and Flask routes, adding new features—such as advanced prompts or multi-image uploads—is straightforward.

CONS

1. **Cost with High Volume**
    a. Generating AI captions for "millions of users" can become expensive, as each request to Gemini may incur costs. Monitoring usage and setting quotas might be necessary to control spending.
2. **Single Region Limitation**
    a. Currently, Gemini is only available in us-central1, while our Cloud Run and bucket might be in us-east1. This can lead to cross-region traffic.
3. **No Advanced User Authentication**
    a. Although the bucket is private, anyone with the public Cloud Run URL can upload images. If the application were to operate at large scale, we might need user accounts, rate limits, or a login system to prevent abuse.
4. **Performance Bottlenecks**
    a. While Cloud Run scales horizontally, at extremely high concurrency there might be performance considerations around GCS writes or AI response times.
    b. Queue-based or asynchronous patterns might be necessary to handle peak loads

Where to improve / Future enhancements

1. **Multi-Region or CDN**
    a. If the application grows to serve a global user base, hosting resources closer to end users can reduce latency. Replicating the bucket or using a Content Delivery Network (CDN) can help images load faster and avoid cross-region data egress costs.
2. **Authentication**
    a. Currently, anyone with the Cloud Run URL can upload images. Introducing user accounts or an OAuth-based system would help manage permissions, track usage, and prevent abuse. Authentication also opens the door to features like per-user galleries or access controls.
3. **Advanced Error Handling & Logging**
    a. For a large-scale production environment, more robust error handling (e.g., retry logic for transient AI errors, structured logs for debugging) is essential. Integrating Google Cloud Logging/Monitoring provides better insight into failures and performance bottlenecks.
4. **Caching or Asynchronous Processing**
    a. If high upload volumes lead to bottlenecks (e.g., many simultaneous calls to Gemini), an asynchronous queue-based architecture could ensure uploads are accepted quickly while AI captioning happens in the background. Caching repeated images or frequently requested content might also help, although repeated uploads of the exact same image may be uncommon.

Challenges / Troubleshooting

1. **Container Startup Errors**
   a. **Issue**: initially, the container would fail to start on Cloud run, resulting in an error message indicating it could not bind to port 8080
   b. **Troubleshooting**: updated our Dockerfile's CMD to explicitly use "python3" instead of "python", ensuring our Python 3 installation was recognized. Additionally, verified that our Flask app correctly bound to 0.0.0.0:8080 by reading the PORT environment variable.
2. **Hardcoded Secrets**
   a. **Issue:** The Gemini API key was accidentally placed directly in main.py, violating best practices and the assignment's requirements.
   b. **Troubleshooting:** switched to passing the key via an environment variable (GENAI_API_KEY) at deploy time. This removed sensitive information from the code base and ensured compliance with project guidelines.
3. **Bucket Permissions & Privacy**
   a. **Issue:** Bucket was originally set to "Public to internet," which contradicts the mandate for a private GCS bucket.
   b. **Troubleshooting:** Adjusted the bucket's IAM policies to remove "allUsers" or "allAuthenticatedUsers," ensuring only the application could access the contents. Maintaining secure storage while still allowing the app to read and write.
4. **Gemini API Payload Format**
   a. **Issue:** Early attempts to call the Gemini endpoint returned "Invalid JSON payload" errors because I used outdated or incorrect fields (e.g., instances vs. contents).
   b. **Troubleshooting:** Carefully reviewed the latest Gemini documentation, updated generate_image_description() function to send inline_data for the image and a text part for the prompt, and parsed the "candidates" array for the AI's response. This ensured the correct request/response flow.
5. **Cross-Region Latency**
   a. **Issue:** Our Cloud Run service and bucket are in us-east1, while Gemini is only available in us-central1, sometimes leading to extra latency.
   b. **Troubleshooting:** Currently, we accept this slight increase in response time. For future optimization, we could move the entire setup or replicate resources in the same region as Gemini.

Conclusion

In this project, I successfully integrated Gemini AI into my existing Cloud Run + Flask application to generate concise textual descriptions for each uploaded image. By storing both the original file and a corresponding JSON metadata file in a private Google Cloud Storage bucket, I ensured secure and organized handling of user uploads. This design meets the key requirements of preventing public bucket access, avoiding hard coded secrets, and maintaining a straightforward, containerized deployment via Cloud Run.

Through this integration, I gained insights into balancing AI calls with serverless scaling, protecting access with environment variables, and systematically handling file operations in GCS. I also learned how to troubleshoot container startup issues, properly manage AI payload formats, and keep costs and performance in mind for high-volume usage. Overall, this project demonstrates a robust and extensible foundation for embedding multimodal AI into a cloud-native application, opening the door to additional improvements such as advanced user authentication, asynchronous processing, and multi-region support in the future.