Course Project

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/cot5930p3

Introduction

This project builds upon the work from earlier milestones:

- Project 1: Deployed a simple Flask web application to Google Cloud Run, using Google Cloud Storage (GCS) for file storage.

- Project 2: Enhanced the application with Gemini (PaLM) AI integration to generate automated titles and descriptions for uploaded images.

- Project 3: Introduced blue/green deployments via Cloud Build (using cloudbuild_blue.yaml and cloudbuild_green.yaml) to enable safe, parallel release strategies with minimal downtime.

This Course project finalizes the sequence by converging traffic to a single revision (100% on either blue or green) and wrapping up our project requirements with a consolidated report. This ensures the application runs at a stable final version without splitting traffic.

Project Requirements

Automated Builds & Deployments

- Used Cloud Build to automatically build Docker images and deploy to Cloud Run.
    - No manual CLI deployments; all deployments triggered by either the cloudbuild_blue.yaml or cloudbuild_green.yaml config.

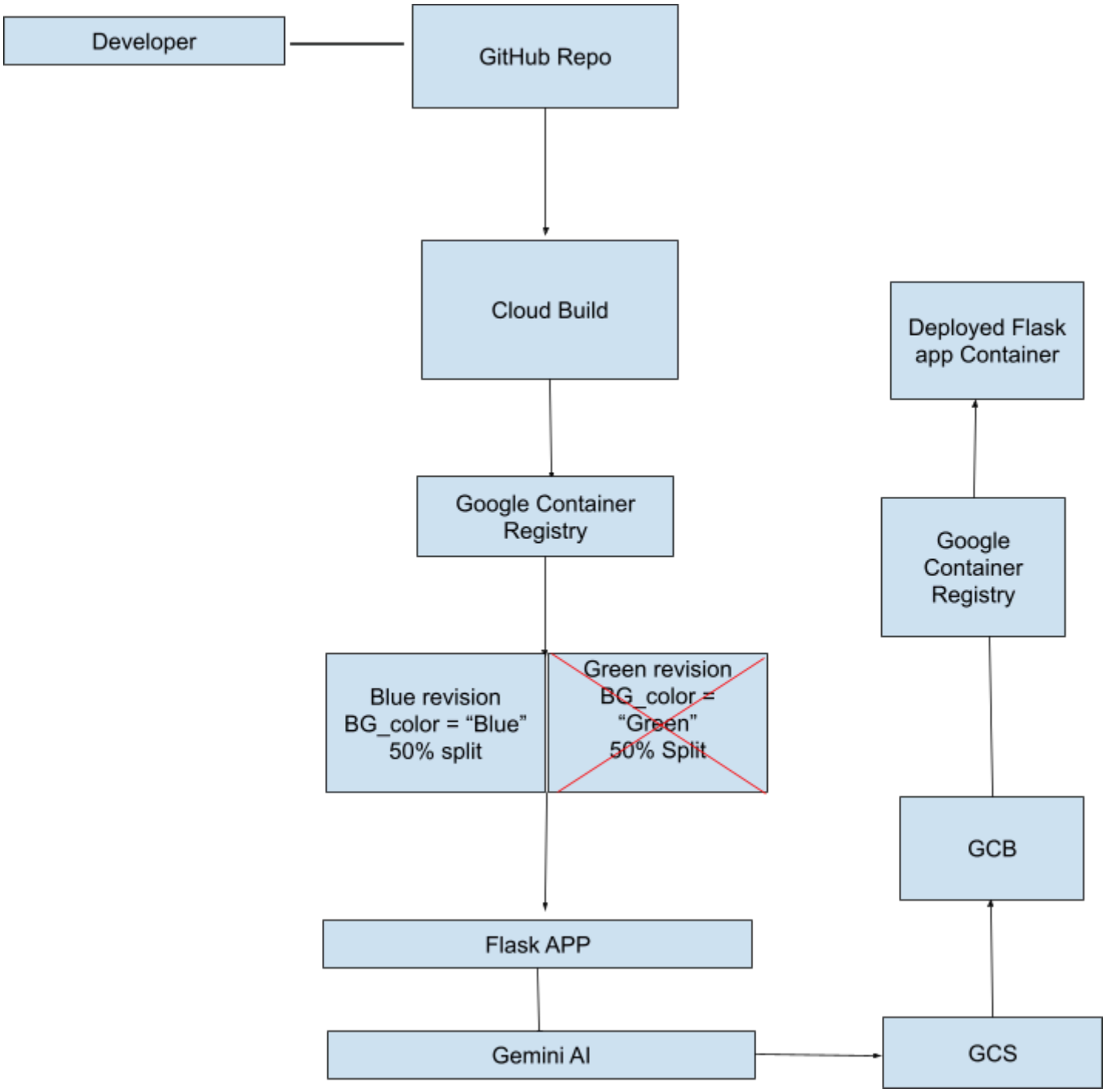
- Blue/Green Release Strategy
    - Maintained two parallel versions (blue and green) receiving 50/50 traffic during the testing phase.
    - Validated that separate revisions can be updated independently, ensuring minimal downtime.

Final Step: Revert to Single Revision

- As part of the final hand-off, traffic was shifted to 100% on a single revision in Cloud Run.

- This consolidates the application to the final, stable version

Architecture



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 container code, relying on environment variables instead
3. **Blue/Green**
    a. Blue: Sets BG_color=blue
    b. Green: sets BG_color= Green
    c. Deployment Approach: Originally split traffic 50/50 between two revisions to test or update safely
    d. Final Stage: All traffic (100%) is now routed to one revision (either Blue or Green) to conclude the release.
4. **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
5. **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
6. **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
7. **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
8. **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

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.
        v. GET / for listing images and providing upload form
        vi. POST /upload for receiving a JPEG, calling Gemini AI, storing results
        vii. GET / view/<filename> to display an image from GCS plus its AI description
        viii. AI calls with function generate_title_and_description(), which sends the base64-encoded image to the Gemini endpoint, expecting JSON.
    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. **cloudbuild_blue.yaml, cloudbuild_green.yaml**:
    a. Each runs a build pipeline that:
    b. Installs dependencies + does a placeholder test step.
    c. Builds/pushes Docker image to Container Registry.
    d. Deploys to Cloud Run with BG_COLOR=blue or BG_COLOR=green.
4. Blue/Green Deployment
    a. Originally service revisions were set to 50/50 (Blue Vs green)
    b. Final step 100% of traffic is routed to one revision blue
5. **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

6. **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
7. **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
8. **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. **Automated Builds with Cloud Build**
    a. In the earlier phase of this project, rather than manually running gcloud builds submit, we created two Cloud Build YAMLs (cloudbuild_blue.yaml and cloudbuild_green.yaml).
    b. Each YAML sets a different environment variable (BG_COLOR=blue or BG_COLOR=green).
    c. These files are connected to two Cloud Build triggers, so whenever I push changes to my GitHub repo, Cloud Build automatically packages my code into a Docker container and deploys it to Cloud Run.
2. **Setting Enviorment Variables**
    a. I no longer have to pass GENAI_API_KEY or BG_COLOR manually with CLI flags; Cloud Build injects these variables during the deploy steps via --update-env-vars=GENAI_API_KEY=... and --update-env-vars=BG_COLOR=... in each YAML file.
    b. This ensures secrets are not hardcoded, and the color is decided at deploy time rather than in the code itself.
3. **Creating single revesion Blue**
    a. Previously Each trigger deploys to the same Cloud Run service but with a different BG_COLOR value.
    b. As a result now, we set all traffic to 100% on blue rather than split
4. **Running and Testing**
    a. Once the builds finish and the revisions are live, I open the Cloud Run URL.
    b. Upload a JPEG from the home page to test the AI captioning.
    c. The application calls Gemini AI to generate a short description, and both the image (filename.jpg) and its JSON (filename.json) are stored in the private GCS bucket.
    d. The home page (/) lists any uploaded images; clicking on one shows the image and its description.
5. **Usage Summary**
    a. Open the Cloud Run URL in your browser.
    b. Upload a JPEG file using the form.
    c. The system calls Gemini to create a description; image + JSON are saved in GCS.
    d. Refresh or open the link in a new browser/incognito to see if you land on the blue revesion

Pros and Cons

Pros:

1. **Serverless and Auto Scaling**
    a. The Cloud Run platform still handles all scaling automatically, so sudden spikes in image uploads or AI requests do not require manual server provisioning.
2. **Blue Deployment & Traffic Splitting**
    a. Having one revision (blue) allows 100% traffic to one cloud build
3. **Automated CI/CD**
    a. Using Cloud Build triggers eliminates manual deploy commands. Pushes to GitHub seamlessly build and deploy new revisions to Cloud Run.
    b. Reduces human error and ensures consistent deployments.
4. **Secure Data storage**
    a. Files remain in a private GCS bucket. Only the Flask app can access the bucket, preventing unauthorized reads/writes.
    b. Environment variables (e.g., GENAI_API_KEY, BG_COLOR) avoid embedding secrets in code.
5. **Extensible & Modular**
    a. Flask routes, AI calls, and GCS interactions are separated.
    b. Additional backgrounds, new AI endpoints, or more advanced prompts can be introduced with minimal code changes.

Cons:

1. **Increased Deployment Complexity**
    a. Maintaining two triggers (blue, green) and adjusting traffic splits can be more complex for smaller teams. (why we switched back to one traffic flow)
    b. Coordinating branches or YAML files for each color might lead to confusion without clear naming.
2. **Cost Concerns at Scale**
    a. AI calls to Gemini remain a potential cost driver if millions of users upload images daily.
    b. Running two parallel revisions doubles the baseline container overhead, although Cloud Run scales down to zero if no traffic arrives.
3. **Single Region Limitations**
    a. The AI model (Gemini) is currently limited to certain regions (e.g., us-central1), while Cloud Run might be in a different region.
    b. Cross‑region traffic can cause latency or additional egress charges.

4. **No Auth/ Rate limiting**
    a. The current setup is open to anyone who knows the Cloud Run URL, which could be abused if usage spikes.
    b. For enterprise or production usage, more robust authentication, rate limiting, and user management would be needed.
5. **Latency & Performance Bottlenecks**
    a. Although Cloud Run can scale out, high concurrency for uploading large images or AI inference might cause noticeable delays.
    b. Potential solutions might involve caching AI responses, queue‑based patterns, or asynchronous processing for high‑volume workloads.

Where to improve / Future enhancements

1. **Multi-Region or CDN**
    a. Although Cloud Run auto‑scales, it currently deploys in a single region (e.g., us-east1). If the user base becomes global, deploying across multiple regions or leveraging a CDN would reduce latency. Additionally, replicating the GCS bucket or using a multi‑region bucket can minimize cross‑region egress costs and speed up image retrieval.
2. **Authentication**
    a. At present, anyone with the Cloud Run URL can upload images. Introducing user authentication (e.g., via OAuth or Firebase Auth) would enable features like per‑user galleries and usage tracking, while rate limiting would prevent abuse or excessive AI calls from malicious actors.
3. **Stronger Testing & CI/CD Validation**
    a. My Cloud Build pipelines currently run placeholder tests (echo "No tests yet"). Replacing those with real integration and unit tests (e.g., using pytest) would help ensure each deployment is stable. Additional checks like linting or static analysis in the pipeline would further raise code quality.
4. **Asynchronous Processing**
    a. For very high traffic or large images, synchronous AI calls could become a bottleneck. Introducing an asynchronous queue (e.g., Pub/Sub) would let the Flask app quickly accept the upload, then process the Gemini AI call in the background. This avoids tying up the request thread during the AI inference.
5. **Caching & Performance Tuning**
    a. If certain images or captions are accessed frequently, implementing a caching layer (like Redis or Memorystore) could reduce repeated calls to Gemini. Additionally, advanced logging/monitoring in Cloud Monitoring could highlight performance bottlenecks (e.g., slow AI responses, concurrency limits).
6. **Enhanced Traffic Management**
    a. Currently, I use a simple 50/50 split for the “blue” and “green” revisions. In larger production scenarios, strategies like canary releases or more granular traffic weighting might be used (e.g., 5% to green, 95% to blue) for safe rollouts. Tools like Cloud Deploy or advanced Cloud Build triggers can further streamline complex release processes.

Challenges / Troubleshooting

1. **Container Startup Errors**
    a. **Issue:**
        i. Initially, the container would fail on Cloud Run with errors indicating it could not bind to port 8080. Also, running locally on port 5000 sometimes conflicted with Cloud Run's requirement for 8080.
    b. **Troubleshooting:**
        i. Updated the Dockerfile to EXPOSE 8080 and ensured Flask read the PORT environment variable (defaulting to 8080 in production).
        ii. Specified if __name__ == "__main__": app.run(host="0.0.0.0", port=port) in main.py to match Cloud Run's port usage.
        iii. Locally, used a fallback port (5000) if PORT isn't set.
2. **Hardcoded Secrets**
    a. **Issue:**
        i. The Gemini API key was accidentally placed directly in main.py, violating best practices and the assignment's requirements.
    b. **Troubleshooting:**
        i. 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:**
        i. Initial calls to Gemini returned "Invalid JSON payload" or "No candidates" due to outdated or incorrect fields.
    b. **Troubleshooting:**
        i. Reviewed the Generative Language API documentation to ensure the request used contents with inline_data for the image, plus a text part for the prompt.
        ii. Parsed the "candidates" array properly to extract "title" and "description".
        iii. Adjusted the code to fallback gracefully if Gemini returns invalid JSON.

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.
6. **Blue/Green Cloud Build Errors**
    a. **Issue**:
        i. Using two separate Cloud Build YAML files for “blue” and “green” initially caused an error: exec: "run": executable file not found in $PATH.
    b. **Troubleshooting**:
        i. Updated the final Cloud Build step to ['gcloud', 'run', 'deploy', 'imaging-app', ...] instead of ['run', 'deploy', ...].
        ii. Specified options: logging: CLOUD_LOGGING_ONLY to handle logs, avoiding the need for a custom logs bucket.
        iii. Created two separate triggers in Cloud Build console, each referencing the correct YAML and branch.

Conclusion

This project successfully integrates all the essential components of a modern cloud-based application: serverless compute (Cloud Run), object storage (GCS), and AI-driven enhancements (Gemini/PaLM API). By implementing blue/green deployments, we demonstrated how to release new features with minimal risk and downtime, initially splitting traffic to test each revision. The final step—directing 100% of traffic to a single revision—completes the deployment lifecycle, providing a stable version for production use. Overall, this solution is both scalable and maintainable, showcasing reliable CI/CD practices and the flexibility to evolve further if the application gains a larger user base.