PROJECT - PHOTO STORAGE APPLICATION

**Introduction:**

This course project involves developing, deploying, and enhancing a web application using Google Cloud Run. The web application of the deployed Cloud Run service enables users to upload images, which are then stored in a Google Cloud Storage bucket. Upon each upload, the Gemini API is invoked to generate a JSON file containing a title and description for the uploaded image. This JSON file, named to match the uploaded image, is stored in the same bucket. The application provides a user-friendly interface where images are

displayed as clickable links, each linking to the respective image along with its generated metadata (title and description).

Building upon this foundation, the project further explores automated deployment and revision management within the Cloud Run environment. The existing Cloud Run service is integrated with GitHub, enabling continuous deployment every time changes are pushed to the GitHub repository, a new revision is automatically deployed in the Revisions section of the existing Cloud Run service. Traffic is set to 100% for the latest deployed revision,

ensuring that all users are served the most up-to-date version of the application. This automation streamlines development and ensures efficient, hands-free updates to the live application.

**Goals and Objectives**

* Deploy a scalable and user-friendly web application using Google Cloud Run.
* Enable users to upload images through the web application of the deployed Cloud Run service.
* Store uploaded images securely in Google Cloud Storage.
* Integrate the Gemini API to generate metadata (title and description) for each uploaded image and store this metadata as a JSON file in the same bucket alongside the image.
* Display uploaded images as clickable links for easy access and viewing.
* Ensure that clicking on an image link display both the image and its associated metadata (title and description).
* Implement a deployment process that utilizes Google Cloud Run's revision and traffic management features.
* Set traffic to 100% for the latest deployed revision, ensuring that all users access the most up-to-date version of the application.
* Set up automated deployment so that any changes pushed to the GitHub repository are automatically deployed to the Cloud Run service, keeping the application up to date with the latest version.

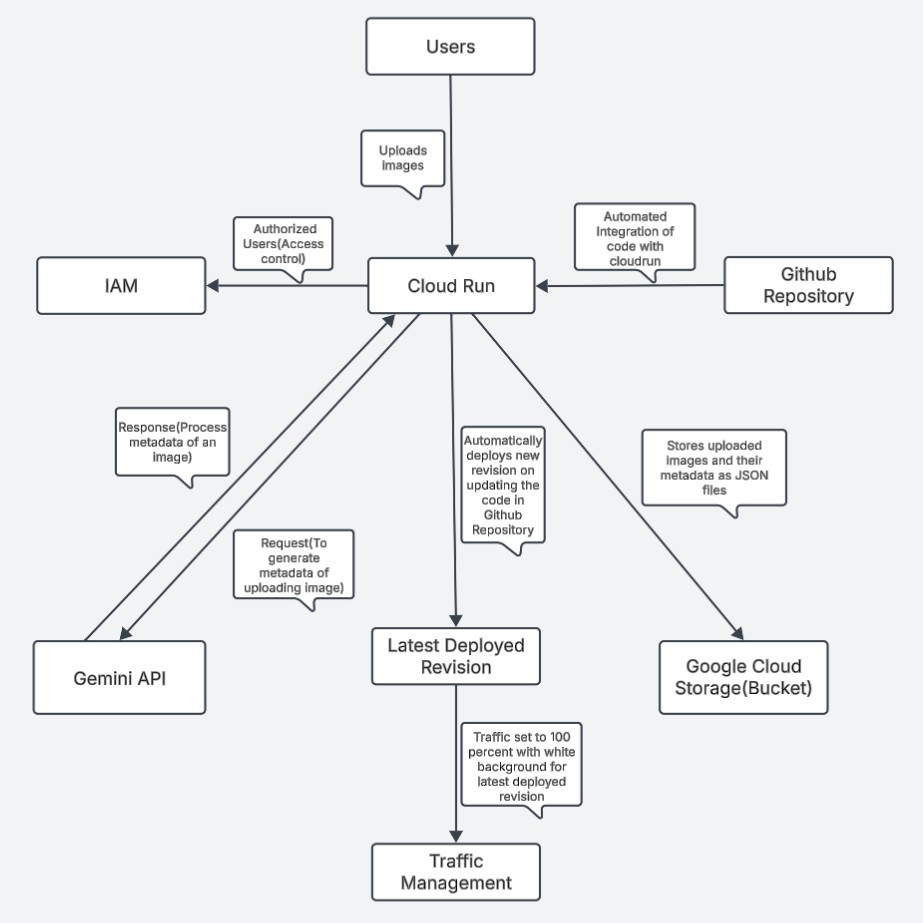
**Implemented Features**

* **Image upload functionality:** Users can select and upload images through a file upload interface.
* **Cloud Storage integration:** Uploaded images are securely stored in a Google Cloud Storage bucket.
* **Gemini API integration:** For each uploaded image, the Gemini API generates a title and description, which are stored as a JSON file in the same bucket with the same name as the image.
* **Image display and interaction:** Uploaded images are shown as clickable links. Clicking on a link opens the image along with its generated metadata (title and description).
* **Scalable deployment:** The application is deployed on Google Cloud Run, allowing it to scale automatically based on traffic, ensuring high availability and

performance.

* **Automated deployment:** Any changes made to the GitHub repository are automatically deployed to the Cloud Run service. When a commit is pushed, a new revision is created and deployed without the need for manual steps, ensuring the application stays updated.
* **Traffic management:** The traffic is set to 100% for the latest deployed revision, ensuring that all users access the most up-to-date version of the application.
* **Revision management:** Google Cloud Run creates a new revision each time a change is committed to the integrated GitHub repository. The service automatically updates to the latest revision, maintaining the most current version of the application in production.

**Architecture:**

****

**Users → Cloud Run**: Upload Images

Users interact with the web application to upload images through a simple interface. They are the starting point of the workflow, providing the image that will be processed.

**Importance**: The user input is the fundamental trigger for the entire process. Without user-uploaded images, there is no data to process, making it the key action to initiate the flow.

**Cloud Run → Google Cloud Storage (Bucket)**: Store Images and Metadata

Cloud Run stores the uploaded images in Google Cloud Storage. In addition to images, metadata in JSON format (generated by the Gemini API) is also stored in the same bucket.

**Importance**: Google Cloud Storage provides reliable and scalable storage for both images and metadata. By storing them together, you ensure that they are easily retrievable in one place. The storage service is designed to handle large volumes of data and provide durability, making it an ideal solution for this project.

**Cloud Run → Gemini API**: Request Metadata Generation

After receiving the uploaded image, Cloud Run sends the image to the Gemini API to generate metadata, including a title and description based on the image content.

**Importance**: The Gemini API utilizes AI models to automatically generate relevant and human-readable metadata for the images. This step adds significant value by providing context to each image, enhancing the user experience. Without this step, the uploaded images would lack descriptive information, which is key for

understanding their content.

**Gemini API → Cloud Run**: Return Generated Metadata

The Gemini API processes the image and returns the generated metadata (a title and description) back to Cloud Run.

**Importance**: The response from the Gemini API contains the metadata (title and description) that is critical for providing users with valuable information about the images. This metadata will be saved as a JSON file and stored alongside the image in Google Cloud Storage, completing the image’s context.

**Cloud Run->IAM**: Grant Permissions to Access Cloud Storage and Gemini API IAM (Identity and Access Management) grants Cloud Run the necessary

permissions to access both Google Cloud Storage (for storing and retrieving images and metadata) and Gemini API (for generating metadata).

**Importance**: IAM ensures that Cloud Run has the appropriate permissions to interact with the resources it needs, like Google Cloud Storage and Gemini API. Properly configured IAM policies are essential for securing the application by allowing access only to authorized components. Without these permissions, Cloud Run would not be able to perform essential actions like uploading images or

requesting metadata from the API.

**GitHub Repository->Cloud Run**: Trigger Cloud Run deployment on committing changes to code

The GitHub repository is directly connected to Cloud Run. When user push code changes to the repository, Cloud Run automatically pulls the latest code and deploys it as a new revision. This eliminates the need for manual deployment steps.

**Importance:**

Ensures the most recent version of the application is always live. It supports fast updates, reduces human error, and improves overall deployment efficiency.

**Cloud Run ->Latest Deployed Revision:** Deploy updated code as a new revision

Whenever Cloud Run deploys new code from GitHub, it creates a new revision. Each revision represents a snapshot of the application at that point in time, preserving the previous states for reference or rollback.

**Importance:**

Provides reliable version control and enables zero-downtime updates. Makes it easier to monitor, test, and manage different application versions.

**Latest Deployed Revision->Traffic Management:** Route user traffic to latest revision

After a new revision is deployed, Cloud Run automatically updates the traffic routing to send 100% of incoming traffic to this latest version

**Importance:**

Guarantees users access the most current version of the app. Helps ensure consistency, performance, and availability.

**Implementation:**

1. **Users (Interacting with the Application)**:

Code snippet and Configuration: HTML Form for image Upload:

Endpoint: /

The HTML form serves as the primary interface for users to interact with the application. It allows users to upload image files (JPEG/PNG)



HTTP Requests:

POST Request: When the user uploads an image, a POST request is made to the server. The image file is sent via the file field in the form.

GET Request: After the image is uploaded and metadata is generated, users can view the image and its associated metadata by clicking links generated on the homepage.

Image Display:

The /files/<filename> endpoint serves the image metadata and display it in the browser. This is a dynamic page that shows the title and description, which is stored in the metadata JSON file.

# Code snippet to fetch and display metadata from Google Cloud Storage



Upload Handling:

After the user submits an image, the backend processes the file, uploads it to Google Cloud Storage, and generates a title and description via the Gemini API.

This provides users with an easy-to-use web interface for uploading images and fetching their metadata.

How this relates to the features of the application:

User interaction is streamlined through a simple interface. The code handles the image upload, metadata generation, and displaying metadata seamlessly for the user

1. **Cloud Run (App Deployment)**

Code and Configuration:

App Location on Server:

The Flask application is deployed to Google Cloud Run for production. Cloud Run automatically handles the scaling of the application depending on the traffic.

Port and Networking:

Cloud Run services use port 8080 by default, so the app listens on this port when deployed.

Code snippet:

Picture

Deployment Process:

The app is built and deployed to Cloud Run, which automatically manages scaling, routing, and networking for the application. Cloud Run routes HTTP requests to the app on port 8080.

Firewall Rules:

Public Access: Cloud Run handles firewall rules by default, meaning the app is publicly accessible over HTTP/HTTPS. No custom firewall rules need to be set manually.

How this relates to the features of the application:

Cloud Run provides an easy deployment platform, eliminating the need for manual infrastructure management. It enables the app to automatically scale with traffic and ensures secure and reliable access to the users.

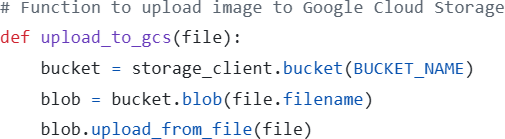
1. **Google Cloud Storage (Bucket)**

Code and Configuration: Storage Location:

Images and their metadata JSON files are stored in Google Cloud Storage Bucket. The app creates the necessary files and stores them within a

specified bucket.

Code snippet:



Metadata is stored in a JSON file associated with each image. The metadata includes the title and description generated by the Gemini API.

Bucket Configuration:

The Bucket Name is stored as an environment variable (BUCKET\_NAME), and it’s used throughout the app to interact with the bucket.

Private Access: Images are accessible via their names displayed in stored bucket, allowing authenticated users to view the uploaded images.

Access and Permissions:

The Cloud Run service interacts with Google Cloud Storage to upload and retrieve files, ensuring that the IAM roles grant the appropriate permissions.

File Storage:

Images are stored with their original filenames (.jpg, .jpeg, .png).

Metadata files are stored with filenames corresponding to the image, but with a .json extension.

How this relates to the features of the application:

Google Cloud Storage provides a scalable and reliable file storage solution. It ensures images and metadata are safely stored, easily accessible, and linked together.

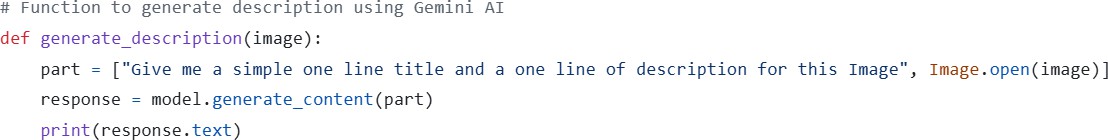
1. **Gemini API (Image Metadata Generation)**

Code and Configuration:

API Request and Response:

After an image is uploaded to Google Cloud Storage, the app sends a request to the Gemini API to generate metadata (title and description) based on the content of the image.

Code snippet:



API Key Configuration:

The Gemini API key is stored as an environment variable (GEMINI\_API\_KEY) and is used to authenticate requests to the Gemini model.

genai.configure(api\_key=os.environ["GEMINI\_API\_KEY"]) Processing the Response:

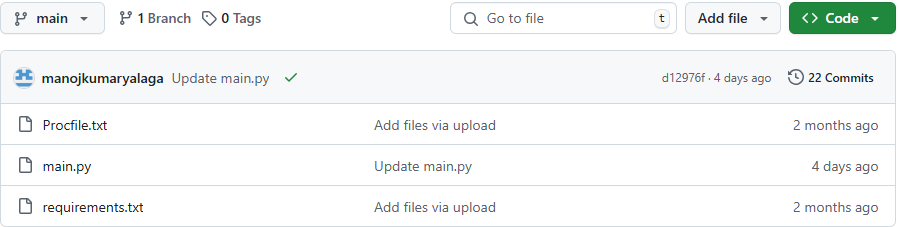
The response from Gemini contains a title and description that are parsed and stored as a JSON file in Google Cloud Storage, associated with the uploaded image.

How this relates to the features of the application:

The Gemini API provides the core functionality of automatically generating meaningful titles and descriptions for images. This step is crucial for creating the metadata associated with each uploaded image.

1. **GitHub Repository**

Below three files are needed for deploying current web application to cloud Run

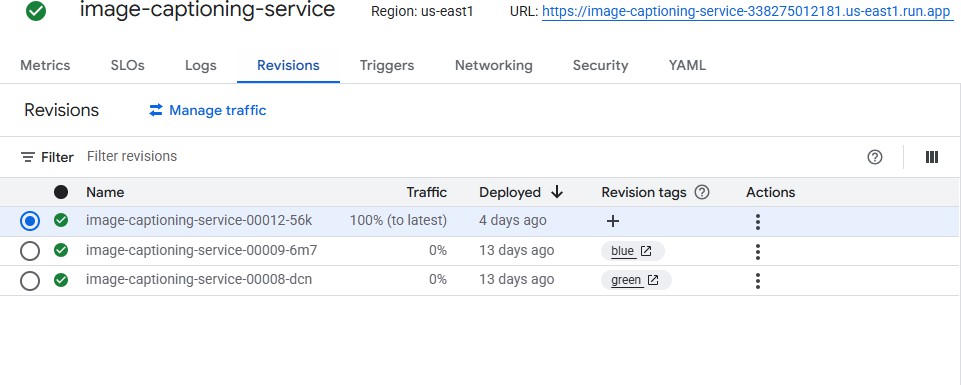


How this relates to the features of the application:

Cloud Run is connected to a GitHub repository for continuous deployment. Whenever a new code update is pushed in main.py, Cloud Run automatically deploys a new revision.

1. **Latest Deployed Revision**

Below is a picture of the Latest Deployed Revision, where each new revision is assigned a unique ID and appears above the previous revisions.



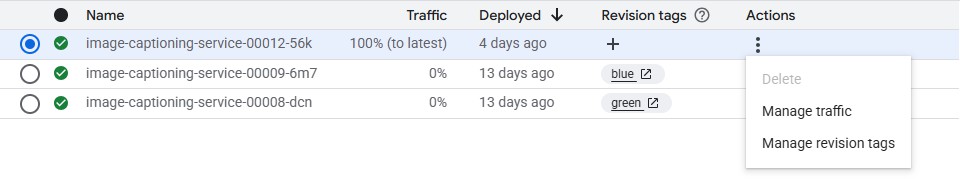
How this relates to the features of the application:

Whenever Cloud Run deploys new code from GitHub, it creates a new revision. Each revision represents a snapshot of the application at that point in time, preserving the previous states for reference or rollback.

1. **Traffic Management**

Below is a screenshot showing traffic set to 100% for the latest deployed revision, which can be configured using the Manage Traffic option in the Actions section. The application's background color is white. Older revisions with green and blue backgrounds are listed

beneath the new revision





How this relates to the features of the application:

Traffic management directly controls which version of the application users interact with, ensuring that the desired features and UI such as the white background are consistently delivered. By setting traffic to 100% for the latest deployed revision, only that version with the finalized features is exposed to users, while older versions remain inactive but accessible for rollback if needed.

**Structure of code:**

**deployment/**

**│**

├ **main.py # Main application logic (Flask app routes and functionality)**

**│**

├ **Procfile # Specifies how the application should be run on Cloud Run**

**│**

├ **requirements.txt # Python dependencies required for the app**

**Pros:**

* Cloud Run dynamically adjusts to traffic, ensuring the app scales based on the number of requests, especially useful for fluctuating workloads.
* Since Cloud Run instances are stateless, multiple instances can handle requests concurrently without affecting the app’s functionality.
* Google Cloud Storage is used to store images, making it globally accessible for any instance, preventing data loss during scaling.
* Deploying the app is straightforward with gcloud run deploy and can be done directly from the source directory.
* Since images and metadata are stored in Cloud Storage, users can access their uploaded content from any instance seamlessly.
* The app integrates well with other Google services like Cloud Storage and Gemini AI for efficient operations and management.
* Changes pushed to GitHub are automatically deployed to Cloud Run, reducing manual intervention and ensuring continuous delivery of the latest code.
* All traffic is routed to the most recent revision, ensuring that users always experience the latest features, design updates and improvements.
* The deployment process is streamlined, as older revisions remain inactive but are retained for rollback if needed, maintaining operational safety without affecting the live app.
* The application scales automatically based on incoming traffic, ensuring consistent performance and responsiveness regardless of user demand.
* Manual traffic controls still provide flexibility, allowing you to reassign 100% of traffic back to a previous revision quickly if a rollback becomes necessary.

**Cons:**

* With the stateless nature of Cloud Run, user sessions and context might be lost when requests are routed to different instances.
* When the app is idle for some time, starting a new instance can introduce delays in handling user requests.
* Due to the distributed nature of Cloud Run, user requests may be handled by different instances, potentially causing issues in user session continuity.
* Heavy traffic and high upload frequency might lead to throttling or delays when accessing Cloud Storage, especially with large files.
* Uploading and processing large images may increase response times, leading to higher latency.
* The solution lacks built-in user authentication, so it’s necessary to implement custom security features.
* Instances are ephemeral, so any data stored on them is lost when they scale down, making state management more difficult.
* Relying entirely on the latest deployed revision means any bugs or issues

immediately impact all users, increasing the risk associated with untested changes.

* If the latest revision fails after deployment, all users are immediately affected until traffic is manually redirected to a previous stable revision, which can lead to

downtime or degraded experience during rollback.

**Problems encountered and Solutions:**

While deploying the app to Cloud Run two error logs arised like worker spawn failed (gunicorn unable to read the main.py application) and other one was unable to recognize the pillow.To resolve this the pip freeze command is used to cross check version numbers of dependencies and change them accordingly in requirements.txt file where for pillow it is corrected.Also I modified the main.py to automatically bind to the environment's assigned port instead of specifying a fixed port like 5000. This was done by modifying the Flask run method to use host="0.0.0.0" and reading the port from the PORT environment

variable,ensuring compatibility with Cloud Run's dynamic environment.

**Instructions to start/deploy the application**:

Steps to deploy using google cloud shell:

1. For deploying the application first, the path needs to be set to current working directory cd path/ to/directory
2. Navigation to the folder of the application need to be set properly where in this project the folder name is deployment that contains the executable files below is command used

cd foldername

1. Before the application is tested on local host the service account key needs to be created and then downloaded to local machine to set the path later. Below is following command to be run in shell to set the path

set GOOGLE\_APPLICATION\_CREDENTIALS=path/to/downloaded Json key file

1. For automatically generating requirements.txt file pip freeze > requirements.txt
2. Later we can make sure all the required dependencies are included in the

requirements.txt file with proper version numbers where we can check version of dependencies using below command

pip freeze

1. Cloud run need Procfile for deploying the flask app on server

web: gunicorn --bind :$PORT --workers 1 --threads 8 --timeout 0 main:app

This command instructs Cloud Run to use Gunicorn a WSGI HTTP server to deploy the Flask app.

1. We need to explicitly export the environment variables like GEMINI\_API\_KEY, PROJECT\_ID and BUCKET\_NAME before running the main.py file as the secrets should not be included in the code

export KEY1=VALUE1 export KEY2=VALUE2 export KEY3=VALUE3

1. Now main.py file is tested on local host port number to check whether the functionality is working or not

python main.py

1. Before deploying to cloud run the authorization related commands need to be run in shell gcloud auth login

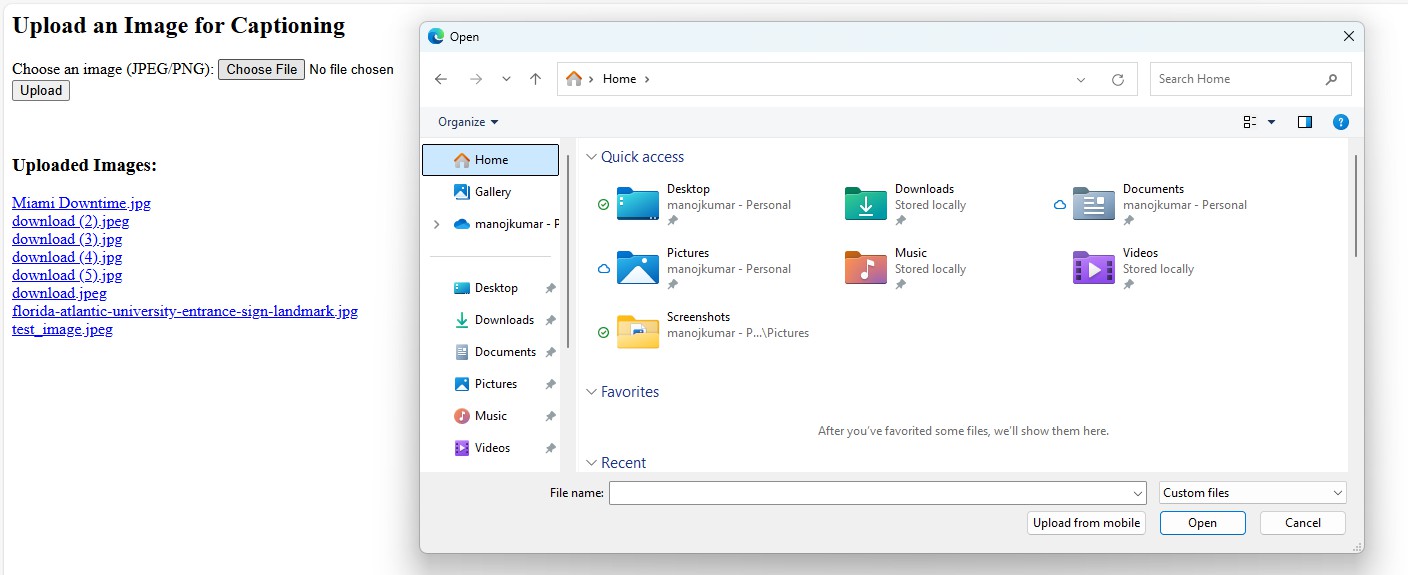
gcloud config set project PROJECT\_ID

1. Once the functionality of main.py is working the app will be deployed using cloud run by explicitly including the environment variables or also can be saved manually in the gcloud platform service and then command (gcloud run deploy) used for deploying the flask app along with the region selected and we can customize the service name and then the service link will be generated and displayed in cloud shell. Below is the cloud deploying command for env variables exported from the shell.

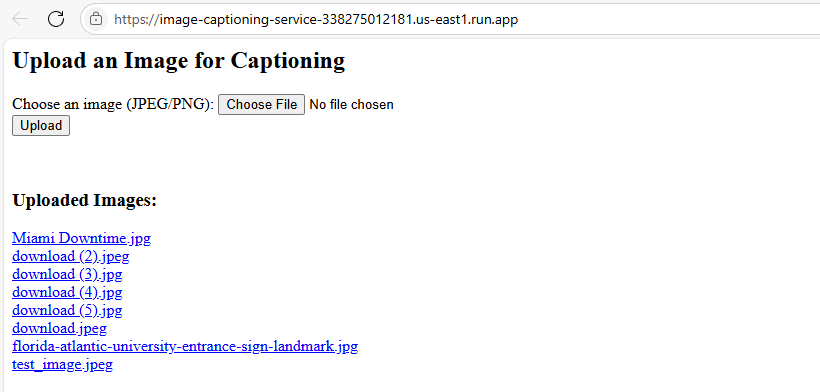
gcloud run deploy service name --source . --region --platform managed --allow- unauthenticated --set-env-vars KEY1=VALUE1 KEY2=VALUE2,KEY3=VALUE3

**Using the application:**

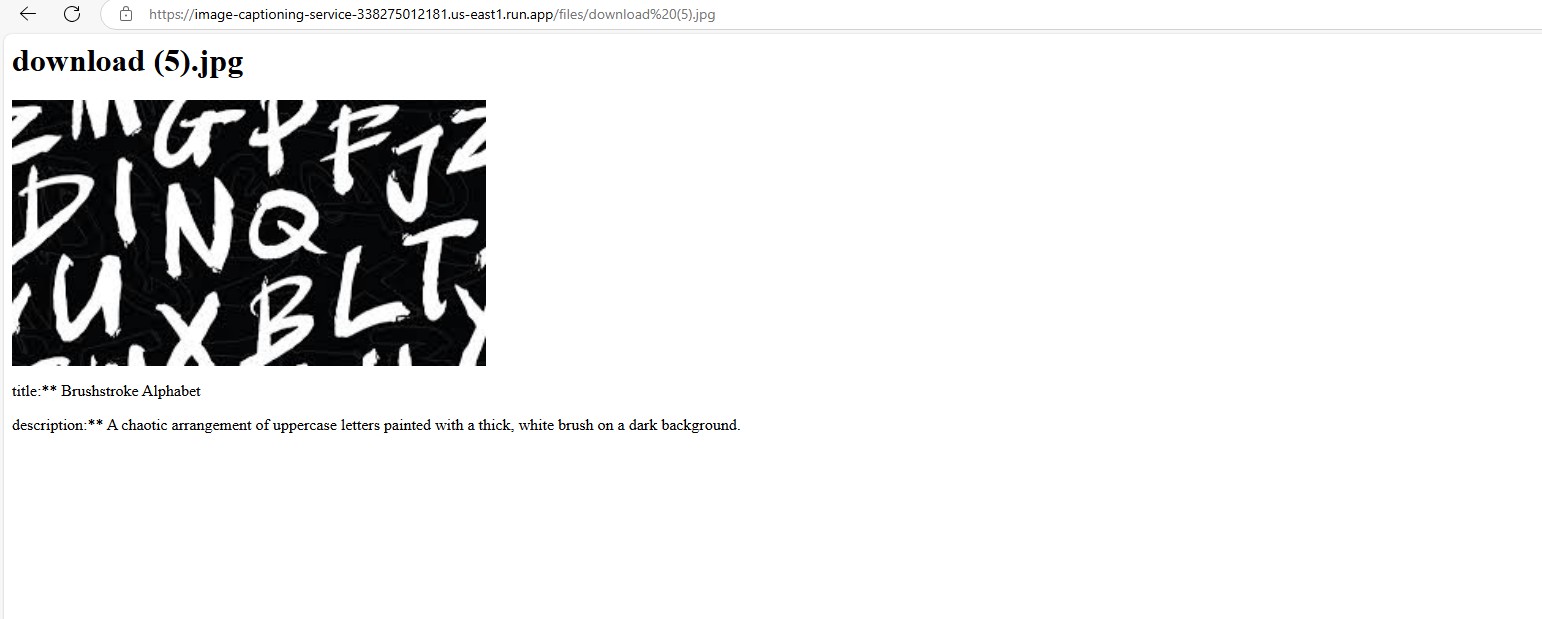
1. Once Cloud Run service link is active on clicking the link the deployed app will be in use where the permitted/authorized Users can access the service link and on clicking choose file button for choosing image from their local machine and using upload button to upload the image.



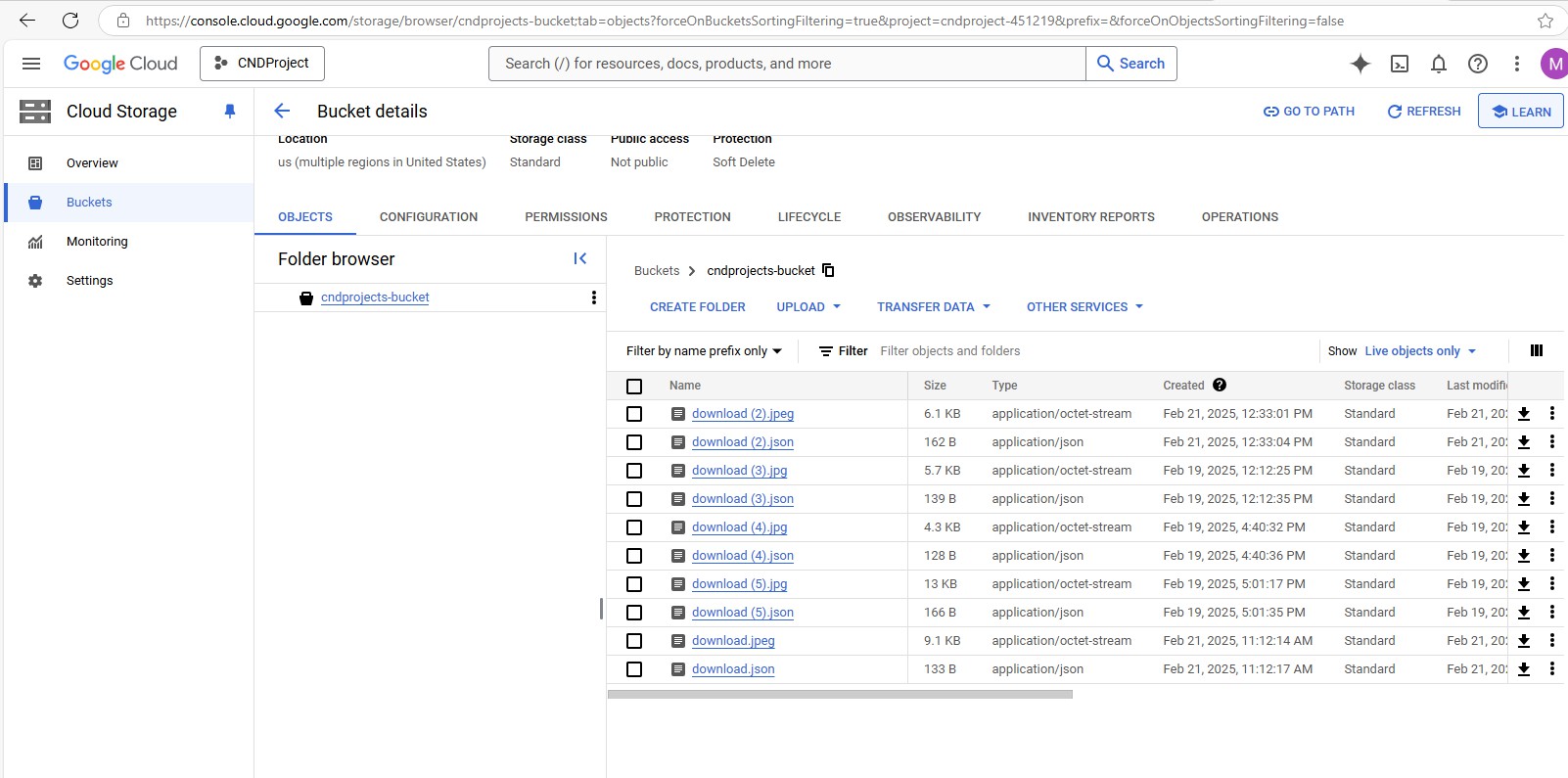
1. The image status of being uploaded successfully is known by seeing it as a clickable link below the upload button once it is uploaded (we can see 8 images in the screenshot with the name download with specified formats like jpg,jpeg)



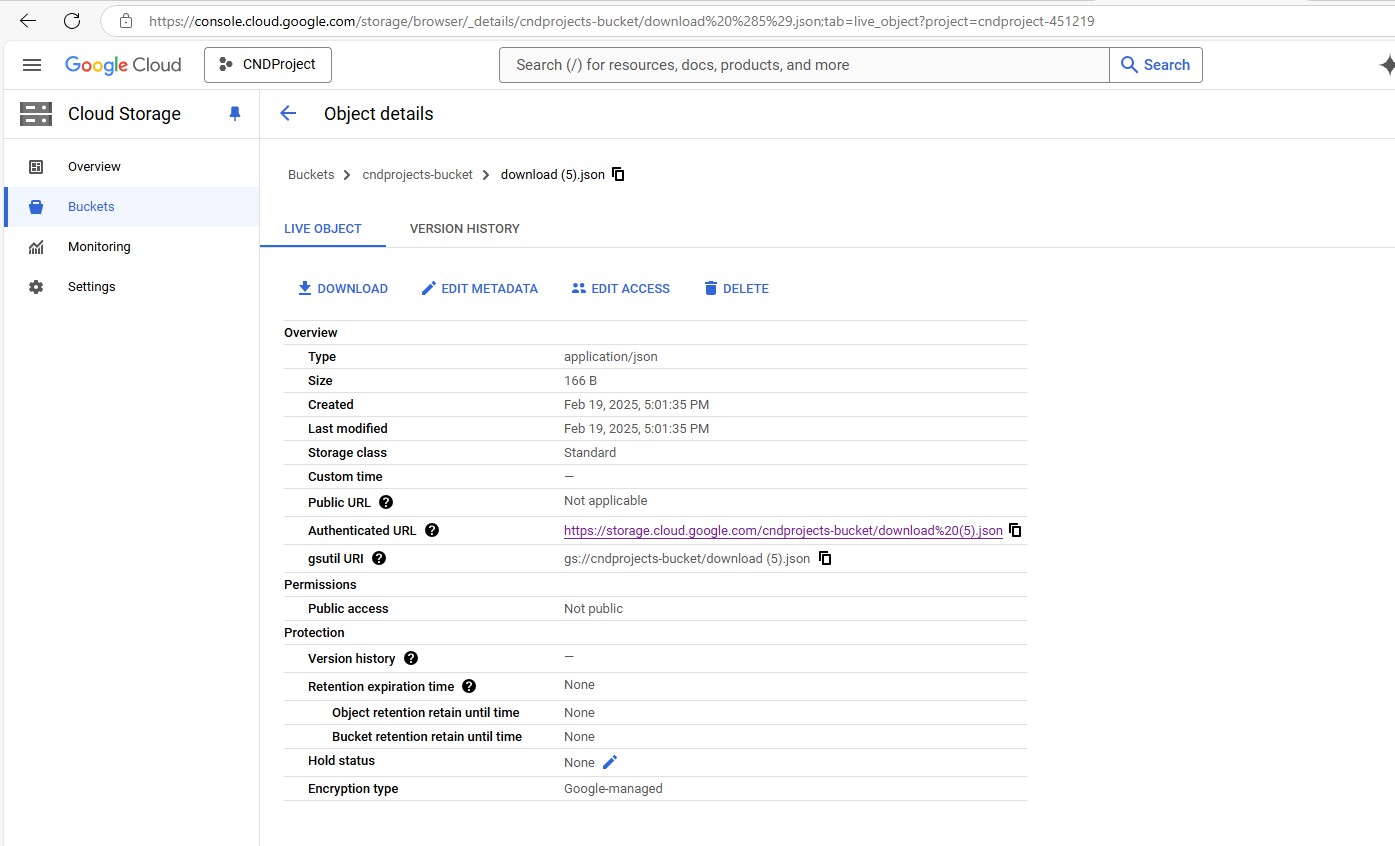
1. Now on clicking the link of the any uploaded image in the deployed app the image content with its title and description are displayed below it.

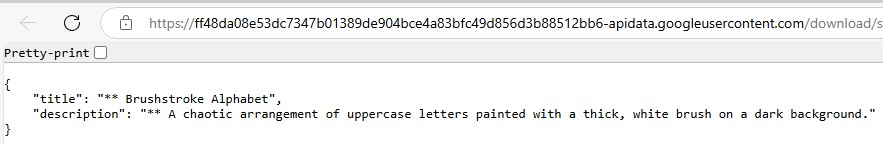


1. Simultaneously the uploaded image file along with the Json file sharing the same name can be checked in the specified bucket of the google cloud storage



1. On clicking the url link of the Json file in the bucket or downloading the file the content of it like the title and the description should match exactly the content showing of its image file





**Lessons Learned while working on this project:**

One of the key lessons learned in this project was the integration of Google Gemini AI for dynamically generating image descriptions. This provided practical experience in working with AI-powered services but also introduced challenges in handling inconsistent AI

responses. To address this, error-handling mechanisms and retry logic were implemented to ensure robustness. Additionally, debugging and error handling were critical aspects of the development process as challenges faced while deploying service to cloud run for generating service link and resolved errors after checking messages in logs so discovered the importance of debugging while working with deployment of cloud services. Also, I learned how to integrate the GitHub repository to cloud run environment that made easy to commit changes in the web page of the deployed service so that the new revision is created each time when we commit changes in main.py of GitHub repository.

**Areas to Improve:**

While the current implementation of this application is functional and meets its core

objectives, there are a few areas where it could be improved. Generated metadata using the Gemini API could be moved to a background task or queue-based system like Cloud Pub/Sub to avoid any delays or timeouts during high traffic. It would also be beneficial to structure the storage system more efficiently and set rules to manage older files automatically. As the user base grows, adding authentication like OAuth2 would help keep the system secure and prevent unauthorized access. On the frontend, a more refined user interface with better feedback during uploads like progress bars or image previews would greatly improve user experience. From a backend perspective, introducing monitoring tools and error reporting would make it easier to catch and fix issues early. If the application were to expand, storing metadata like Big Query would make it easier to search or analyze later.

**Does this application become successful with millions of users?**

Yes, this application has the potential to become successful with millions of users. Since it is built on Google Cloud Run, which automatically scales based on traffic, it can handle

high loads without requiring manual server provisioning. Cloud Storage offers reliable and scalable storage for images and metadata, making it well-suited for large volumes of data.

The automated deployment pipeline ensures that updates are efficiently rolled out with minimal downtime. However, for long-term success at scale, improvements like adding user authentication, rate limiting, enhanced error handling, and monitoring would be important. With these enhancements, the application can remain stable, secure, and responsive as the user base grows.