Fall 2024 – Data Center Scale Computing

Final Project

AI -PHOTO MANAGEMENT SYSTEM

GitHub **YouTube**

(Detailed URLs are provided at the end)

Teammates: 1) Venkateswarlu Mopidevi 2) Tarun Kumar Nagelli

PROJECT GOAL:

The goal of this project is to create an AI-based photo management system that helps users organize and find their photos easily. When a user uploads a photo, the system will automatically sort and group all pictures featuring that person. This makes it simple to quickly find photos of specific people from events without searching through the entire collection manually.

Additionally, the system will offer a convenient feature where users can upload a photo to filter and find similar images. This makes it easy and quick to locate matching pictures, saving time and effort.

COMPONENTS:

APIs:

Purpose: RESTful APIs will act as the bridge between the frontend (what users see and interact with) and the backend (where data is processed and stored).

Functions: These APIs will allow users to:

- Upload images to the system.
- Google cloud vision API: This is used to check how many faces are detected in an uploaded image. If more than one/no face detects this will pop an error message.
- Get all the photos from the Google cloud bucket (Gallery).
- Filter matched images from the gallery using uploaded image.
- Fetch recently uploaded images from a redis endpoint.

Frontend:

Technology: React.js will be used to create a smooth and responsive user interface.

Features

- Users will be able to upload photos through an intuitive interface.
- View photo albums or search results directly on the platform.
- Interact with the AI-powered filtering and organization features seamlessly.

Backend:

Technology: Built with Node.js to handle core functionalities of the system.

Functions:

- Receives images uploaded by users and processes them.
- Caches uploaded images in **Redis** for quick access.
- Logs details like file name, upload time, image status, etc., into a relational database. This database will be useful for tracking and retrieving specific data when needed.

Cloud Storage:

Tool: Google Cloud Storage will be used to securely store all the uploaded images.

Role:

- Acts like an online photo gallery where the images are organized and stored for retrieval.
- Ensures that the images are safe, backed up, and accessible whenever needed.

Containers:

Tools: Docker and Kubernetes will be used for deploying the backend.

Advantages:

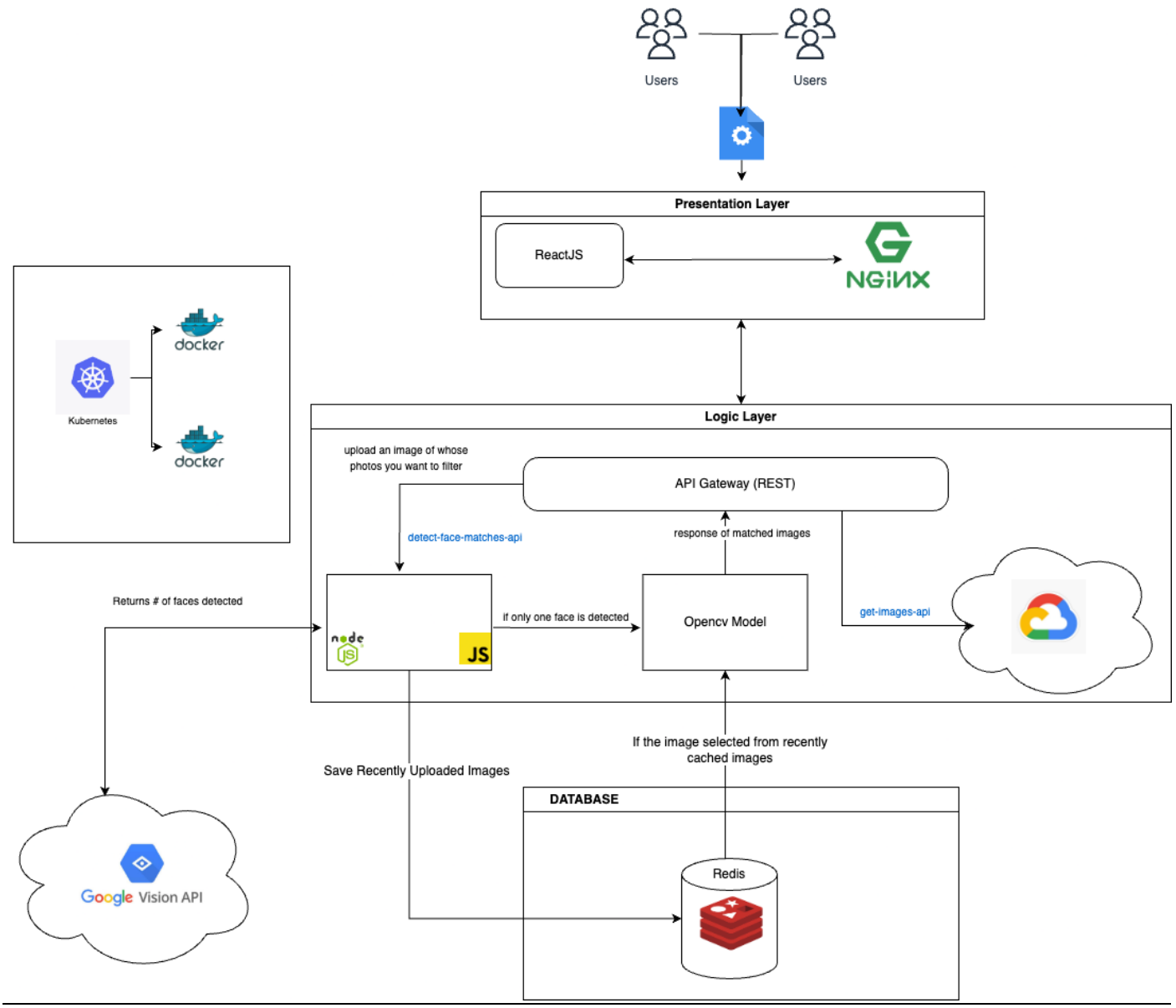
- **Docker**: Ensures that the backend runs consistently, no matter where it is deployed.
- **Kubernetes**: Helps manage multiple instances of the backend efficiently, ensuring the system is reliable and scalable.

Database and Key-Value Store:

Redis: This tool will serve two main purposes:

- **Caching**: Stores recently uploaded images temporarily for fast access.
- **Quick Access**: Makes browsing and retrieving recently added photos much faster compared to fetching them from the main database or cloud storage.

ARCHITECTURAL DIAGRAM (Working System):



INTERACTION AMONG SOFTWARE COMPONENTS:

1. Initial Load

- **Frontend → Backend: GET /images**
    - When the website is first loaded, the frontend sends a request to the **get-images API**.
    - **Backend**: This API interacts with **Google Cloud Storage** to retrieve all the images stored in the bucket.
    - The API responds with a list of image URLs.
    - **Frontend**: Displays these images in the gallery for the user to view.

---

2. Uploading an Image

- **Frontend → Backend: POST /detect-face**
    - The user uploads an image via the frontend. The image is sent to the **detect-face API**.
    - **Backend Process**:
        1. The API sends the image to the **Google Vision API** to detect faces.
        2. **Face Detection Logic**:
            - If **multiple** or **no faces** are detected:
                - The API returns an error to the frontend (e.g., "Multiple faces detected" or "No face detected").
            - If **exactly one face** is detected:
                - The image is passed to the **OpenCV model** for further processing.
                - **OpenCV Model**: Searches for matching faces in the gallery images stored in **Google Cloud Storage**.
                - A list of matching images is returned.

        - The API sends this list of matching images to the frontend for display.

    3. **Redis Caching**:
        - The uploaded image, along with metadata (file name, status: success or failure), is stored in **Redis**.
        - Only the 5 most recent uploads are retained in Redis.

---

3. Dropdown for Recent Uploads

- **Frontend**:
    - Displays a dropdown containing the filenames of the last 5 uploaded images, sourced from **Redis**.
    - This allows the user to quickly select a recently uploaded image instead of re-uploading.

---

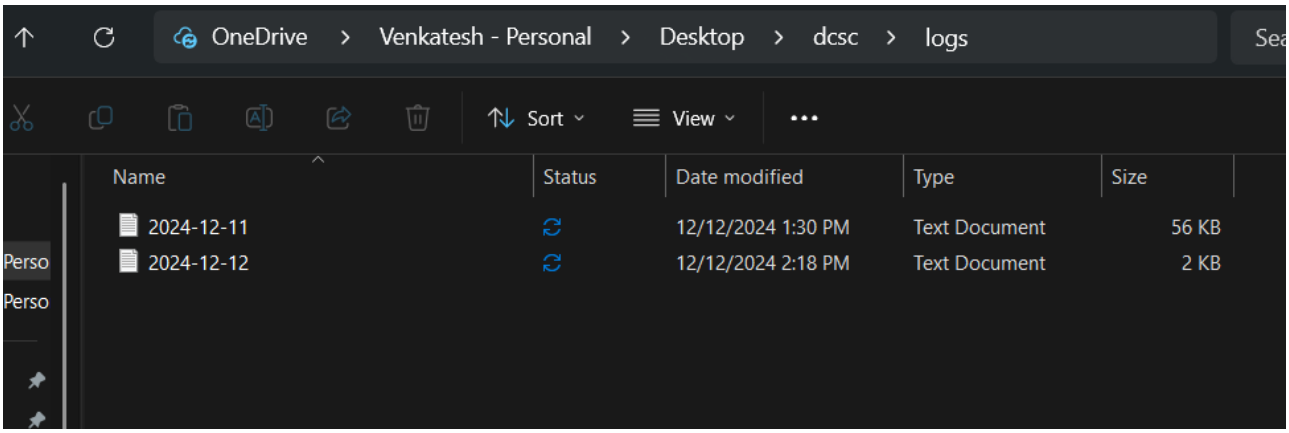
4. Selecting from Recent Uploads

- **Frontend → Backend: POST /detect-face-match-recent**
    - When a user selects an image from the recent uploads dropdown, the frontend calls the **detect-face-match-recent API**.
    - **Backend Process**:
        1. Checks the image's status (stored in Redis):
            - If status = success:
                - The image is sent directly to the **OpenCV model** for face matching.
                - A list of matching images is returned.
            - If status = failure:
                - An error message is returned to the frontend (e.g., "Multiple or no faces detected earlier").
        2. The API responds with the matching images or an appropriate error.

TESTING AND DEBUGGING MECHANISMS:

Overview:

At each step of the process, a robust logging system was implemented to monitor the flow of operations and capture detailed logs for debugging and analysis. Logs are stored in separate files based on the current date, making it easy to trace events and identify issues for specific days. Additionally, the source of logs (e.g., frontend, backend Python, or backend Node.js) is specified, allowing for quick identification of the relevant component in case of errors. This systematic logging ensures that issues are easily traceable, facilitating efficient debugging and smoother operation.




```
[2024-12-11 17:33:11] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:12] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:12] INFO: [Backend-Python] Detected 2 face(s) in the image.
[2024-12-11 17:33:12] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:12] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:13] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:13] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:13] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:13] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:14] INFO: [Backend-Python] Detected 2 face(s) in the image.
[2024-12-11 17:33:14] INFO: [Backend-Python] Detected 2 face(s) in the image.
[2024-12-11 17:33:14] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:15] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:15] INFO: [Backend-Python] Detected 1 face(s) in the image.
[2024-12-11 17:33:15] INFO: [Backend-Python] Detected 0 face(s) in the image.
[2024-12-11 17:33:15] INFO: [Backend-Python] Matching faces found: 14
[2024-12-11 17:33:15] INFO: [Backend-Python] Face matching completed successfully.
[2024-12-11 17:33:15] INFO: [Backend-Python] Temporary image file /tmp/tmpdw9wposa.jpg deleted.
[2024-12-11 17:33:15] INFO: [Backend-Node] Deleted uploaded image: image_1733961858100
[2024-12-11 17:33:15] INFO: [Backend-Node] Face detection and matching completed successfully.
[2024-12-11 17:33:15] INFO: [Frontend] Redis image processed successfully Face detection successful.
```

1. User Uploads a Photo

- **Debugging:**
  Verify that the upload API handles a variety of image formats (e.g., JPEG, PNG) and file sizes. Detailed logs capture any errors encountered during the upload process, such as file format mismatches or size limits exceeded. Error messages from the logs guide troubleshooting efforts to resolve issues efficiently.

- **Testing-Mechanism:**
  Conducted unit tests to validate that the API correctly accepts and processes uploads. Test cases include various file formats, sizes, and multiple simultaneous uploads to ensure stability and robust error handling. Automated tests simulate edge cases such as interrupted uploads or unsupported formats.

2. Image Storage & Metadata Logging

- **Debugging:**
  Check that images are securely uploaded to Google Cloud Storage. Logs capture errors related to storage connectivity, permissions, or metadata inconsistencies, enabling efficient identification and resolution.

- **Testing-Mechanism:**
  Perform integration tests to confirm seamless communication between the storage service and the Backend Node.js. Testing involves uploading sample images and verifying that both the images and metadata are stored as expected. Simulate edge cases like network outages to test retry mechanisms and ensure data integrity.

3. Image Processing & Facial Recognition

- **Debugging:**
  Implement logging at every step of the image processing pipeline, including image reading, preprocessing, and facial recognition. Log errors when the model fails to detect faces or produces false positives/negatives. Logs aid in fine-tuning the model by identifying problematic input or configurations.

- **Testing-Mechanism:**
  Used a labeled dataset with known faces to test the accuracy of the facial recognition system. Conducted repeated testing and refined the model based on observed shortcomings, retraining as necessary to improve detection accuracy.

4. Data Caching

- **Debugging:**
  Inspect Redis cache entries to confirm that frequently accessed data is properly cached. Verify that expiration policies are applied correctly to avoid stale or unnecessary data. Logs track cache hits and misses to identify inefficiencies in the caching strategy.

- **Testing-Mechanism:**
  Performed performance testing by simulating frequent queries for popular photos. Measure data retrieval times with and without caching to ensure significant performance gains. Validate that the caching mechanism prevents overflows and handles data invalidation efficiently.

5. Photos Retrieval

- **Debugging:**
  Verified that the application accurately retrieves matched photos from Google Cloud Storage based on the image uploaded by the user. This process applies whether the uploaded image comes directly from the user's system or from Redis cache (for recently uploaded images). Logs were used to capture any mismatches, delays, or errors during the retrieval process, ensuring any issues such as incorrect matches or retrieval failures are promptly identified and resolved.

- **Testing-Mechanism:**
  Conducted functional testing to validate that the application retrieves matched photos from Google Cloud Storage correctly based on the uploaded image, irrespective of whether the source of the uploaded image is the user's system or Redis cache. Simulated scenarios with various uploaded images to ensure the accuracy of matching and retrieval. Performed user acceptance testing to confirm that the retrieval process meets expectations for speed and relevance. Tested edge cases, missing links between uploaded images and stored photos, to ensure robust error handling and a seamless user experience.

6. Docker – Containerization & Isolation:

- **Debugging:**
  In the debugging process, Docker provides the ability to isolate services and diagnose issues at the container level using commands like docker logs <container_id>, which allows for inspecting logs of individual containers. When containers fail or crash, inspecting these logs helps identify specific errors related to the container's execution. Additionally, using docker ps ensures that all containers are running as expected and resource allocation is properly managed, avoiding any resource-related failures. Docker's containerization allows for debugging within a contained environment, minimizing cross-service issues.
- **Testing-Mechanism:**
  To test Docker, we use the command docker ps to check if the containers are running correctly. This ensures that the services inside the containers are active and functioning. We also run automated tests within the containers to verify the services work as expected.

7. Kubernetes - Orchestration & Scaling:

- **Debugging:**

  When debugging Kubernetes, commands such as kubectl logs <pod_name> help identify issues within specific pods by providing detailed logs. If a pod is failing, kubectl describe pod <pod_name> reveals critical information about the pod's status, resource allocation, and possible issues like insufficient memory or container image errors. Additionally, using kubectl get pods allows tracking of pod statuses across the cluster, helping identify if pods are stuck or in a pending state, making it easier to isolate and resolve service issues across a distributed environment.

- **Testing-Mechanism:**

  For Kubernetes, we use kubectl get pods to check if the pods are running. This helps ensure that all services are working and scaled correctly. If there's an issue with any pod, we can use additional commands like kubectl describe pod to diagnose the problem.

Conclusion:

In this project, we implemented a comprehensive debugging and testing approach to ensure the reliability and efficiency of the image upload, processing, and retrieval system. By integrating a detailed logging system at every key step, we were able to trace and resolve issues effectively, ensuring smooth operations. The logging system provided insights into potential failures, such as image format mismatches, storage errors, and caching inefficiencies. Alongside this, we employed rigorous testing mechanisms, including integration, and performance tests, to validate that each component functioned as expected under various conditions. Specifically, we focused on ensuring robust handling of user uploads, accurate image processing, seamless data retrieval from Google Cloud Storage, and efficient caching using Redis. Our testing also covered edge cases, such as interrupted uploads, network failures, and expired cache entries, to ensure the system could gracefully handle unexpected scenarios. This multi-layered approach to debugging and testing ensured that the system met performance and accuracy standards, providing a reliable and user-friendly experience.

Github Links:

https://github.com/cu-csci-4253-datacenter-fall-2024/finalproject-final-project-team-51/tree/main

Youtube Link:

https://www.youtube.com/watch?v=H_OzAdSsyGQ