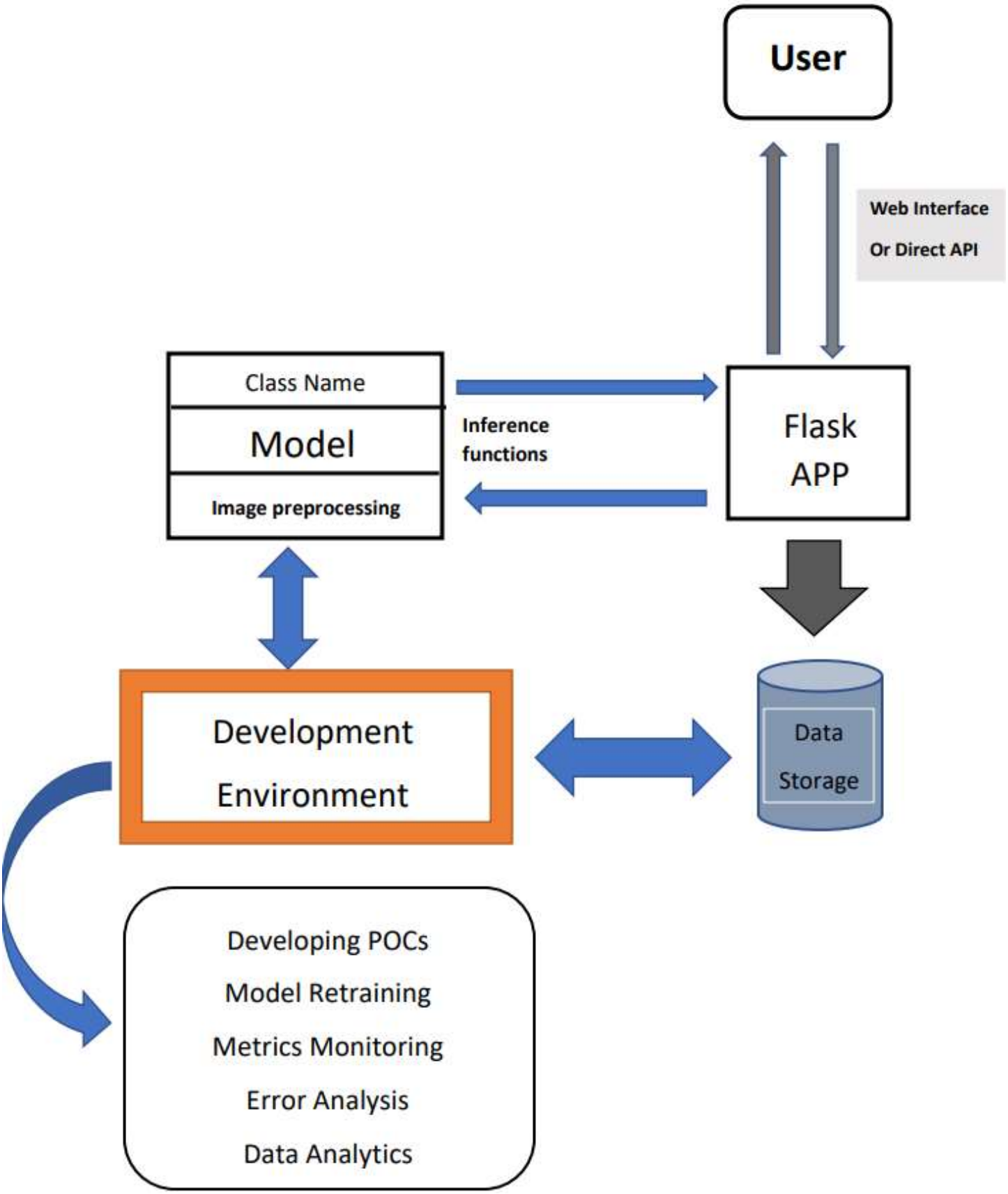
System Architecture:



Description:

1. User Client: Represents the user interface (e.g., web browser, mobile app) through which users interact with the system. Users upload images and make API requests to classify the images.
2. Flask Web App: The core component of the system. It is built using Flask, a Python web framework. The Flask app receives API requests from the User Client, processes the uploaded images, and uses the pre-trained Image Classification Model to predict the class of the image.
3. Image Classification Model: The pre-trained model (e.g., CNN) used for image classification. It takes the processed image as input and returns the predicted class of the image.
4. SQLite Database: The relational database used for data storage. The Flask app saves API request data, including timestamps, user information, image filenames, predicted classes, and processing time, into the SQLite database.

Data Flow:

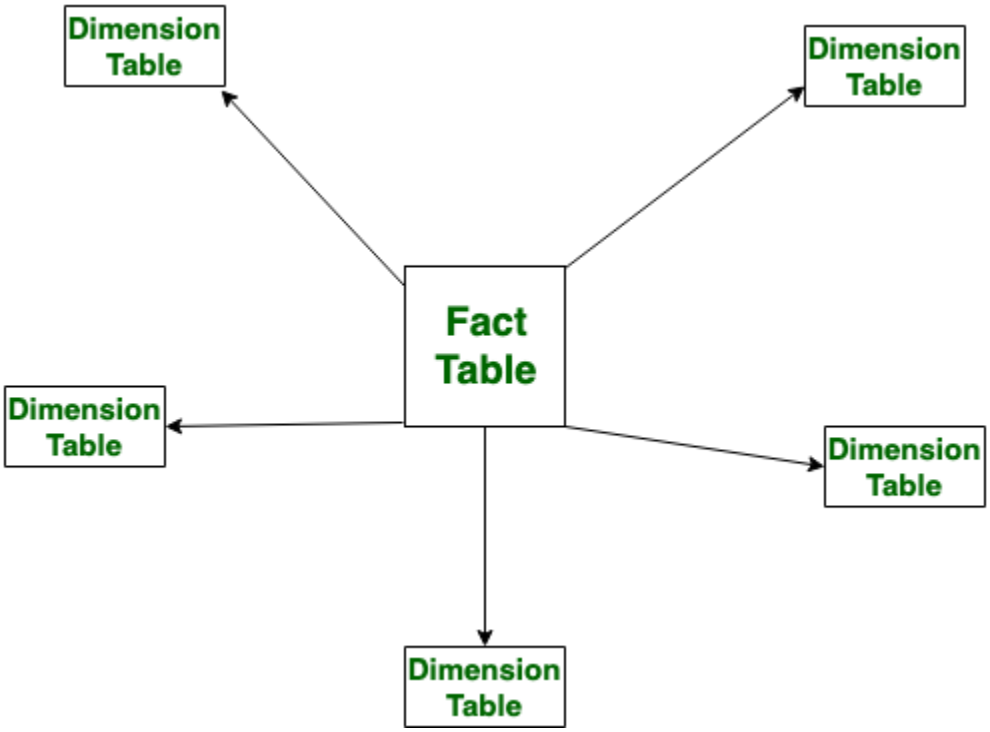
1. The User Client sends an API request to the Flask Web App with an uploaded image.
2. The Flask Web App receives the API request and processes the image, preparing it for classification.
3. The processed image is passed to the Image Classification Model for prediction.
4. The Image Classification Model predicts the class of the image.
5. The Flask Web App saves the API request data, including the predicted class, into the SQLite Database.
6. The Flask Web App responds to the User Client with the predicted class.

The App System Architecture will vary when be in production with scaling

Data Storage and Management:

In the provided submission for the Flask app with the image classification API, the proposed system design includes the usage of a relational database, specifically SQLite, to store the data in a star schema-like format. This design enables efficient data storage and facilitates OLAP (**Online Analytical Processing**) operations for data analysis.

Data Models:



Star Schema: The proposed data model follows the star schema, a widely used data modeling technique in data warehousing and OLAP systems. The star schema consists of a central fact table surrounded by dimension tables. In this case, the fact table is **events**, and the dimension tables are **time_dim**, **user_dim**, **image_dim**, and **class_dim**.

Fact Table (**api_event_fact**): The fact table stores quantitative data related to the API requests. It contains the following columns:

- **timestamp_key**: Foreign key referencing the **time_dim** table.
- **user_key**: Foreign key referencing the **user_dim** table.
- **image_key**: Foreign key referencing the **image_dim** table.
- **class_key**: Foreign key referencing the **class_dim** table.
- **Time Taken**: The time taken for the API request to be processed (measure).
- **Request Count**: The count of API requests for each row (measure, sample data).

Dimension Tables:

a. **users** Dimension Table:

- **user_id**: Primary key for uniquely identifying each user.
- **username**: The username of the user who made the API request.
- **ip_address**: The IP address of the user.
- **user_agent**: The user agent string from the request headers.

b. **images** Dimension Table:

- **image_id**: Primary key for uniquely identifying each image.
- **image_filename**: The filename of the uploaded image.

c. **classes** Dimension Table:

- **class_id**: Primary key for uniquely identifying each class.
- **class_name**: The name of the predicted class (e.g., "Cat," "Dog," etc.).

The dimension tables are connected to the fact table through foreign keys, allowing efficient querying and analysis. With this star schema, the system architecture allows efficient querying and analysis of the API request data. The Flask app handles incoming requests and saves the relevant information into the SQLite database. The star schema data model simplifies data analysis, enabling OLAP operations like aggregations, slicing, and dicing.

Scalability and Performance:

Horizontal Scaling: with Container Orchestration (Docker and Kubernetes): We will containerize our Flask app and image classification service using Docker. Kubernetes will be employed as the container orchestration platform to automate the deployment, scaling, and management of containers. Horizontal scaling will be achieved by running multiple replicas of the containers to handle increased user demand.

Or Serverless with AWS Lambda (Optional): To optimize resource utilization and cost efficiency, we can implement serverless architecture for certain components, such as image preprocessing tasks or API request handling. AWS Lambda can be used to automatically scale and run functions in response to incoming requests without the need to manage servers.

Vertical Scaling: Implement auto scaling computation instance like EC2 Instances: For resource-intensive tasks like image classification, we can employ vertical scaling by using AWS EC2 instances with more powerful configurations. As the user base and resource demands grow, we can upgrade the instance types to handle the increased load.

Load Balancing: A load balancer will be utilized to evenly distribute incoming traffic across multiple instances of the Flask app or containers. This ensures optimal resource usage and improved responsiveness.

Caching: We could implement caching mechanisms to store frequently requested data for specific user or area, such as image classifications, in memory. This reduces computation time and minimizes the load on the backend systems.

Fault Tolerance and Reliability:

Redundancy and Replication: We will deploy critical components, including the Flask app, image classification service, and database, redundantly across multiple servers or availability zones. This provides fault tolerance, ensuring continuous service availability even in the face of failures.

Data Backup and Recovery: Regular backups of the database will be taken and stored in a reliable and durable storage solution (e.g., Amazon S3). In case of a database failure or data loss, the backups can be used to recover the data and restore the system.

Automated Monitoring and Alerting: We will set up monitoring using tools like Prometheus and Grafana to collect and visualize metrics related to system health, performance, and machine learning-specific metrics like data and model drifts. Alerts will be configured to notify the operations team in real-time if any metric exceeds predefined thresholds, allowing for prompt actions to address potential issues.

Error Handling and Fallback Mechanisms: The system will be designed with proper error handling and fallback mechanisms. For example, in case of a failure in the image classification service, the system can fallback to using a backup classifier or display a friendly error message to users.

Contact details:

Name: Mohned Moneam

Phone: +201550154178

Email: mohnedmoneam@gmail.com / mohnedmoneam00@gmail.com