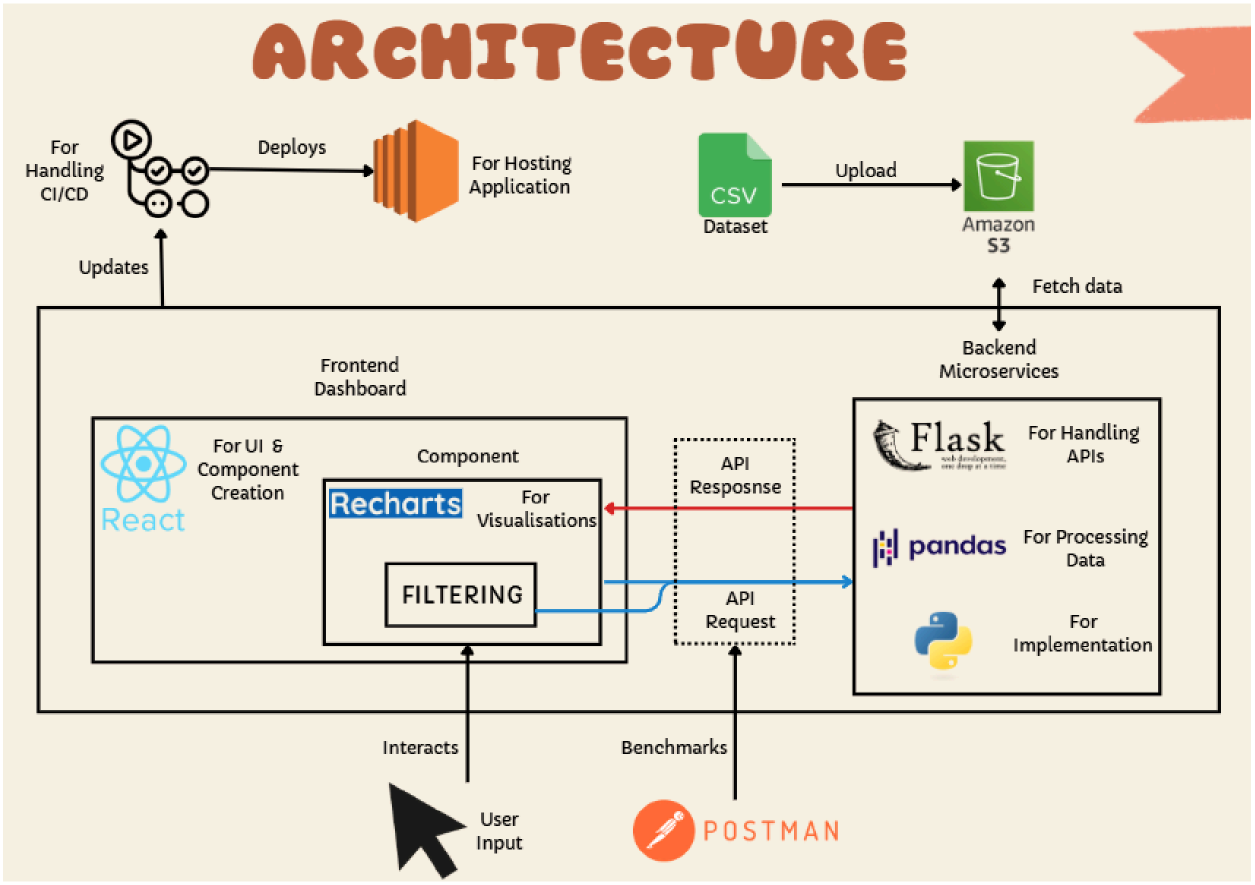

Figure 1: Project Architecture

As seen in Figure 1, this microservice dashboard project follows a modular and scalable architecture that integrates multiple AWS services and technologies for data storage, processing and visualisation. The project is ultimately divided into multiple parts: Storage, Backend Microservices, Frontend Dashboard, CI/CD pipeline and Benchmark.

The Storage part is responsible for storing and managing the dataset, which is crime-related. The crime data is stored in AWS S3 as a CSV file, acting as the main data source. This ensures that the data remains centralised, easily accessible and scalable.

The Backend Microservices part is entirely implemented with Python and consists of a Flask-based microservice that acts as a backend API. This microservice fetches the crime dataset from AWS S3, processes it dynamically and exposes RESTful API endpoints for visualisation in the front end. Each endpoint is responsible for a different process of data transformation: crime heatmaps, yearly crime trends, most affected districts, crime type distribution and year-over-year crime rate changes. The API is designed to be lightweight and efficient, using Flask for request handling and data transformation.

The Frontend Dashboard part provides an interactive web dashboard built with React.js and Recharts. The frontend dynamically calls API endpoints based on user-selected filters, such as year, state, crime type, etc. From there, the backend will process and transform the data accordingly to the specific API endpoint. Once it is completed, the data will be sent back to the frontend as a response and displayed. Various charts (Pie Charts, Bar Charts, Line Charts, etc.) are visualised using Recharts. The dashboard allows users to easily interpret crime trends and distributions.

The CI/CD pipeline for the Flask microservice is automated using GitHub Actions and then deploying to an AWS EC2 instance when pushing to the master branch. When checking out the code, it sets up the appropriate environment (Python 3.11) and installs dependencies to validate the build. The pipeline then secures an SSH connection to EC2 using a private key. Before it deploys, it creates a timestamped backup of the current deployment to enable a rollback mechanism. The updated files are efficiently transferred and once deployed, the pipeline installs dependencies in a virtual environment. It also terminates the previous Flask process and restarts the application. This setup provides automated deployments while ensuring that the Flask API is always updated.

The Benchmarking process utilises Postman to measure the performance and reliability of the Flask API by testing response times, handling concurrent requests, and validating query parameter accuracy. By incorporating Postman benchmarking, the system maintains high robustness, performance and scalability before deployment.