Smart Banking System Using Cloud

Ashraf Uddin Chowdhury Rafat (40272674)*Electrical and Computer Science engineering*Concordia Universityashrafuddinchowdhury.rafat@mail.concordia.ca

Sai Abhinav Tadepalli (40257238)  
*Electrical and Computer Science engineering* Concordia Universityabhinavsai.t@gmail.com

Proddut Kumar Biswas (40262586)  
*Electrical and Computer Science engineering*Concordia Universitypr\_bis@live.concorsssssdia.ca

Adilmanzoor Kalilur Rahman (40271551)  
*Electrical and Computer Science ssengineering*Concordia Universityadilmanzoor.kalilur@live.concordia.ca

**Abstract**

The Smart Banking System using Cloud is a cloud-based application developed to modernize banking services by harnessing technologies such as Flask, MongoDB, RESTful APIs, Docker, HTML, JavaScript, and advanced AI models. The platform offers essential banking functionalities, including secure authentication, real-time transaction processing, balance inquiries, fund transfers, and detailed transaction histories. Enhanced by AI features, it provides OCR-powered cheque processing, AI signature verification, an intelligent chatbot for customer support, and predictive expense analysis. Security is a paramount focus, implemented through JWT-based authentication, encrypted transactions, multi-factor verification, and secure session management. A robust Continuous Integration/Continuous Deployment (CI/CD) pipeline has been established to automate deployment, improving development efficiency and consistency. Built on a monolithic architecture, the application consolidates all features into a single deployable unit. The entire system is deployed on the Google Cloud Platform (GCP), leveraging its services for scalability and availability. To validate the APIs and functionalities, a client-end webpage was deployed for testing purposes. This report explores the system's architectural design, technical implementation, challenges encountered, and insights gained throughout the development journey.

**Keywords—Smart Banking System, Cloud Computing, Monolithic Architecture, CI/CD Pipeline, Flask Framework, Docker, RESTful API, JWT Authentication, AI Chatbot, Optical Character Recognition, Signature Verification, Predictive Expense Analysis**

1. **Introduction**

**1.1 Goal and Objectives**

This project's primary aim is to develop a cloud-based smart banking system that delivers secure, efficient, and intelligent banking services enhanced by AI capabilities. The objectives are:

* **Core Banking Functionalities:** Deliver essential banking services accessible globally through cloud deployment.
* **AI-Driven Features**: Enhance user experience with an intelligent chatbot, expense prediction, and cheque processing.
* **Cheque Processing with OCR:** Allow users to deposit cheques via the application using image processing and Optical Character Recognition (OCR).
* **Expense Tracking and Prediction**: Provide insights into users' spending habits by tracking expenses and predicting future spending patterns.
* **Robust Security Measures:** Ensure security through JWT authentication, encrypted transactions, and multi-factor verification.
* **CI/CD Implementation:** Automate deployment processes using a Continuous Integration/Continuous Deployment pipeline.
* **Cost-Effective Development:** Utilize free-tier cloud services on Google Cloud Platform (GCP) to minimize costs.

**1.2 Problem Statement**

Traditional banking systems often face limitations such as restricted accessibility, manual operations, and insufficient tools for financial planning. Customers today seek real-time services, personalized assistance, and intelligent tools to manage their finances effectively. The absence of predictive financial insights and tailored services can impede users from making informed decisions. While many modern applications are moving towards microservices architectures, monolithic architectures remain prevalent, especially in projects with limited resources, due to their simplicity and ease of development.

Our project adopts a monolithic architecture, where all components—including user interface, business logic, and database interactions—are unified within a single deployable unit. This approach streamlines development and deployment processes, making it suitable for our project's scale and resource constraints. To enhance deployment efficiency and maintain continuous delivery, we have implemented a CI/CD pipeline that automates building, testing, and deployment stages.

**1.3 Assumptions**

* Users have devices with internet connectivity.
* The system complies with relevant financial regulations and data protection laws.
* Users consent to data analysis for expense tracking and personalized financial services.
* The project utilizes free-tier services provided by Google Cloud Platform (GCP) to avoid additional costs

**1.4 Methodology**

An Agile development methodology was embraced to allow iterative development and feature integration. The project progressed through several phases, beginning with requirement analysis and planning, followed by design and architectural considerations. We implemented core banking features before integrating AI components such as the chatbot and cheque processing functionalities. Additional features like the shop and purchase history were added to enhance user engagement. A critical aspect was the implementation of a CI/CD pipeline, automating build, test, and deployment processes, and ensuring rapid and reliable updates. Deployment of a client-end webpage was undertaken to test and validate the APIs and functionalities from the user's perspective. Finally, we conducted thorough testing and deployed the application on GCP, iterating based on feedback and performance.

**1.5 Schedule**

**Key project milestones:**

|  |  |
| --- | --- |
| **Week No** | **Activity** |
| Weeks 1-3: | Team formation, requirement gathering, and initial planning. |
| Weeks 4-6: | Development of core banking functionalities and security features. |
| Weeks 7-9: | Integration of AI chatbot and cheque deposit features. |
| Weeks 10-11: | Implementation of CI/CD pipeline, additional features, testing, and optimization. |
| Week 12: | Final presentation, demonstration, and report submission. |

**2. Project Description**

**2.1 Functions Provided by the Service**

**2.1.1 Core Banking Operations**

Our smart banking system provides a comprehensive suite of core banking functionalities designed to enhance user convenience and security. User registration and authentication are facilitated through a secure sign-up and login process using JWT-based authentication. Passwords are securely hashed with bcrypt, and users provide their signatures during registration, enabling AI-based signature verification for added security in specific transactions.

Account management allows users to view account details and check their balances at any time. The system supports a variety of transactions, including deposits, withdrawals, and transfers. Deposits can be made directly or via mobile cheque deposits, with OCR technology efficiently processing cheque information. Withdrawals and fund transfers are conducted in real-time, ensuring immediate updates to account balances and secure transaction processing using the recipient's email address.

A comprehensive transaction history is maintained, providing users with a detailed view of all past financial activities, such as deposits, withdrawals, transfers, purchases, and cheque deposits. Predictive expense analysis offers users insights into their spending habits, with forecasts for the next three months categorized by spending areas to promote better financial management.

To enhance user engagement, the application includes a shop and purchases feature, allowing users to browse and purchase items within the platform. A purchase history feature is also available, enabling users to review their shopping activities seamlessly. This robust feature set positions the system as a secure, intelligent, and user-friendly banking solution.

**2.1.2 AI-Enhanced Features**

The application is enhanced with several AI-driven features:

* **Intelligent Chatbot Assistance:** An AI-powered chatbot built with Microsoft DialoGPT provides users with a conversational interface to perform banking operations. The chatbot can handle various intents, such as checking balances, transferring funds, viewing transaction histories, and creating support tickets. It accepts natural language inputs, facilitating seamless interactions and improving user convenience. The chatbot maintains conversation history to provide contextual responses and personalize the user experience.
* **Cheque Processing with OCR:** Users can deposit cheques by uploading images through the application. The system employs the Google Cloud Vision API for extracting text from cheque images. AI models, specifically Transformer-based models like Microsoft/swin-tiny-patch4-window7-224, are used for signature verification, comparing the uploaded signature with the one provided at registration. This ensures secure and accurate processing of cheque deposits.
* **Predictive Expense Analysis:** To assist users in financial planning, the system offers predictive expense analysis. It aggregates purchase data by category and time, utilizing a Transformer-based machine learning model to forecast expenses for the next three months. This feature provides insights into future spending patterns, helping users make informed financial decisions.

**2.2 Architectural Design**

**2.2.1 Overview**

Our system is constructed using a monolithic architecture, where all functionalities are encapsulated within a single Flask application (server.py). This design choice consolidates backend services, API endpoints, AI integrations, and database interactions into a single codebase, simplifying deployment and maintenance.The monolithic application encompasses several key components:

* Authentication Service: This component manages user registration, login, and JWT token issuance. It handles secure user authentication and ensures that only authorized users can access the banking services.
* Banking Service: Responsible for core banking operations, the banking service allows users to perform transactions such as deposits, withdrawals, transfers, and balance inquiries. It interacts with the database to update account balances and record transaction histories.
* AI Services: Integrated AI functionalities enhance user experience and security:

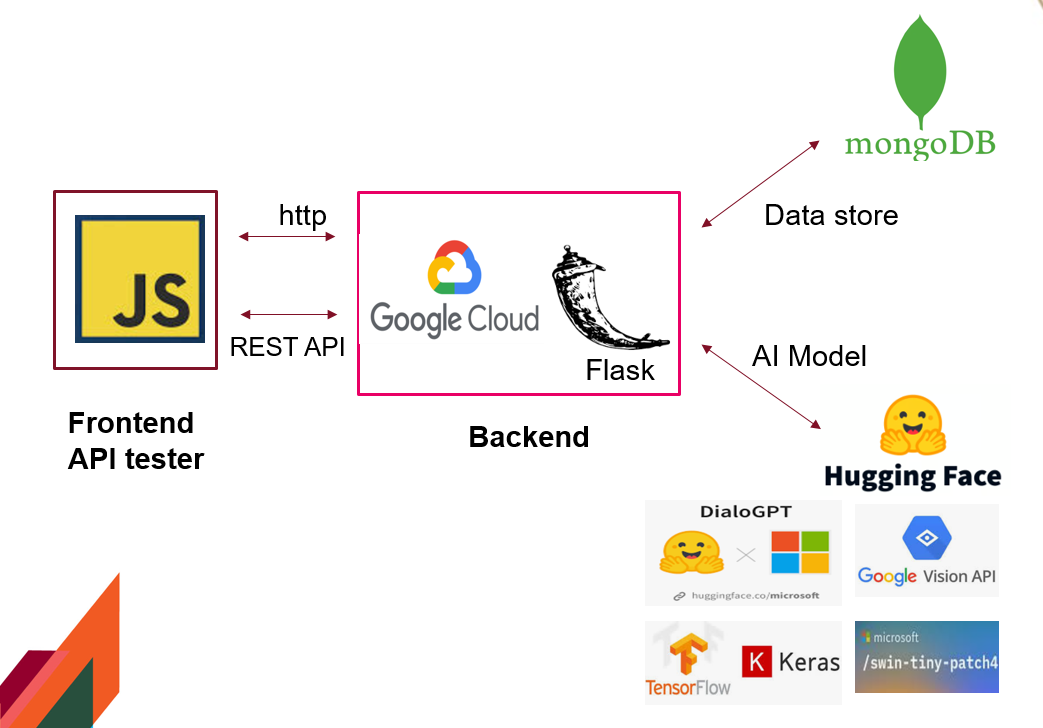
1. **Chatbot Service:** Provides AI-powered conversational assistance using the microsoft/DialoGPT-medium model from Hugging Face. The chatbot can handle user inquiries, facilitate transactions, provide account information, and assist in customer support activities like creating and viewing support tickets.
2. **OCR Service:** Processes cheque images to extract financial information using the Google Cloud Vision API. This service enables users to deposit cheques remotely by capturing and uploading cheque images through the application.
3. **Predictive Analytics Service:** Analyzes transaction data to predict future expenses. By incorporating machine learning models and external financial datasets, it offers users insights into their spending patterns and assists in financial planning.

Communication within the application is facilitated through RESTful APIs using Flask. This allows seamless interaction between the frontend and backend and among different components within the backend. The entire application is hosted on the Google Cloud Platform (GCP), utilizing free-tier services like Cloud Run and Compute Engine. Deployment is managed using Docker for containerization, ensuring consistency across development, testing, and production environments.

A Continuous Integration/Continuous Deployment (CI/CD) pipeline is implemented using GitHub Actions and integrated with GCP services. This pipeline automates the build, test, and deployment processes, enhancing development efficiency and reliability. The system employs MongoDB Atlas (free tier) as the centralized database, accessed by the application as needed. It stores user data, account details, transaction records, shop items, and tickets in MongoDB collection.

Security measures include the use of JWT tokens for secure authentication and session management, and passwords are securely hashed using bcrypt for protection against unauthorized access. Secure token handling and validation mechanisms are implemented to maintain session integrity and application security.

**2.2.2 System Architecture Diagram**



**Fig 1 System Archietecture**

**2.3 Technical Implementation Details**

**2.3.1 Technologies Used**

* **Programming Language:** Python
* **Frameworks:**
  + Backend: Flask for RESTful APIs.
  + Database: MongoDB Atlas (free tier).
* **Cloud Platform:** Google Cloud Platform (GCP) using free-tier Compute Engine instances.
* **Containerization:** Docker.
* **Development Tools:** Visual Studio Code.
* **AI Models:**

1. **Chatbot- Microsoft/DialoGPT-medium** from Hugging Face.
2. OCR for Cheque Deposit: Google Cloud Vision API (free tier usage).
3. Signature Verification: **microsoft/swin-tiny-patch4-window7-224 model.**
4. **Prediction Analysis:** Tensor Flow Keras (Neural Network Model)

* **Security:**

1. Authentication: JWT.
2. Encryption: Passwords hashed with bcrypt.

**2.3.2 Implementation Highlights**

1. **Monolithic Architecture:**

Our monolithic architecture consolidates all functionalities into a single Flask application (server.py). This approach simplifies the deployment process by reducing the complexity associated with managing multiple services. Despite being monolithic, the codebase is organized into modular components to improve readability and maintainability. Each module corresponds to different functionalities, such as authentication, banking operations, AI services, and more. This modularity within the monolithic structure allows developers to work on individual components without affecting the entire application.

1. **CI/CD Pipeline:**

A robust Continuous Integration/Continuous Deployment (CI/CD) pipeline is implemented using GitHub Actions and integrated with Google Cloud Platform (GCP). The pipeline ensures that any changes pushed to the repository are automatically tested, built, and deployed.

In the Continuous Integration phase, automated tests are triggered on every push to the main branch. This helps maintain code integrity by catching issues early in the development cycle.

During Continuous Deployment, successful builds are containerized using Docker and pushed to the Google Container Registry. The application is then deployed to Google Cloud Run, a serverless platform that automatically scales the application based on traffic, ensuring high availability.

Configuration files play a crucial role in the CI/CD process:

* + cicd.yml: Defines the steps involved in the CI/CD pipeline, specifying how the application should be built, tested, and deployed.
  + Dockerfile: Contains instructions for building the Docker image, including installing dependencies and setting up the application environment

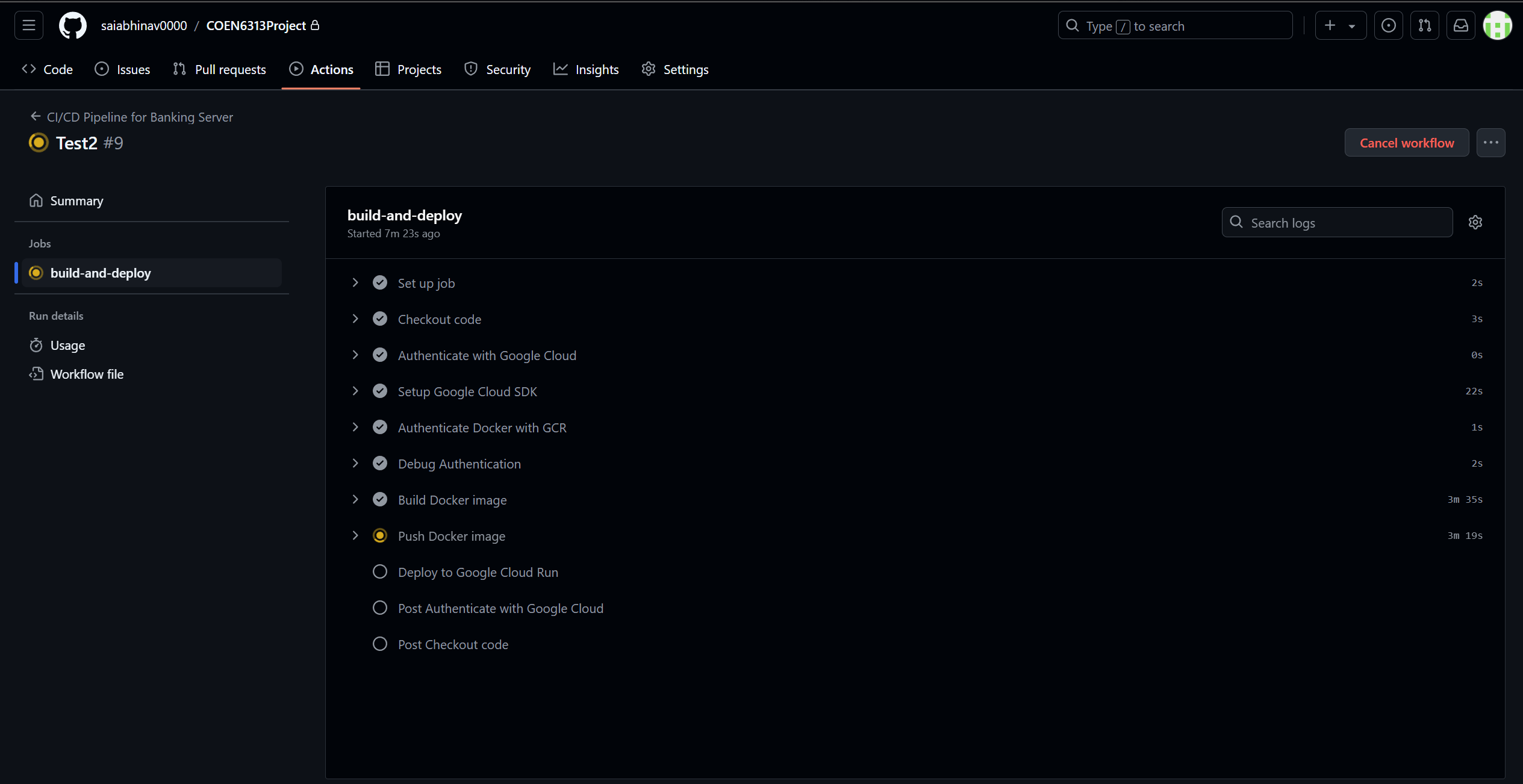


Fig 2 Continuous Integration and Deployment Pipeline

1. **Usage of Free-Tier Services:**

* **Google Cloud Platform:** Leveraged free-tier offerings for deploying the application, including services like Cloud Run and Compute Engine.

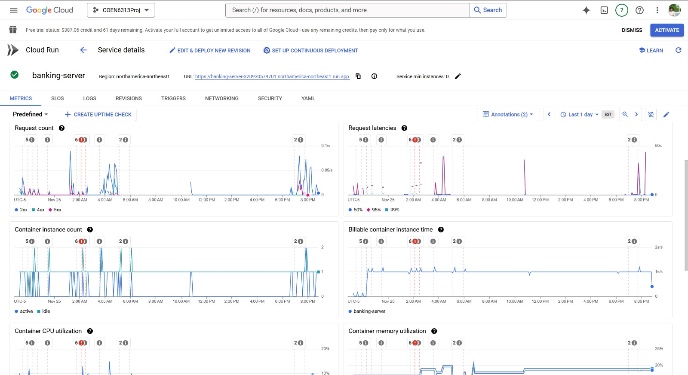


Fig 3 Google Cloud Run Performance Metrics for Smart Banking System

The above figure displays the key performance metrics of our smart banking system monitored via Google Cloud Run. It shows request counts, latencies, and container statistics like instance count and memory use. The data reveals that our system efficiently manages fluctuating workloads, maintaining stable latencies and effective resource use even during peak usage. This monitoring helps us ensure that the banking services we offer are both reliable and cost-effective, essential for maintaining a high-quality user experience.

* **MongoDB Atlas:** Leveraged the free-tier for database hosting.

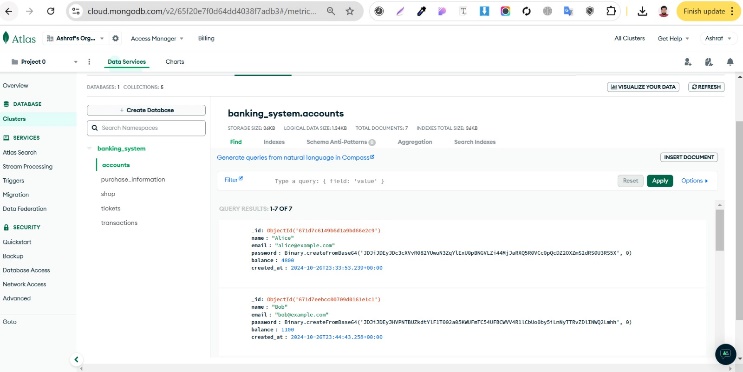


Fig 4 Database Schema Overview for Smart Banking System

The Above figure display the structured database collections that our smart banking system uses to store critical data:

1. **Accounts Collection:** Here, we keep user details like names, emails, encrypted passwords, and balance information. It's the backbone for managing user identities and security.
2. Transactions Collection: This logs every transaction—deposits, withdrawals, and transfers, noting the type, amount, and relevant account details. It's crucial for keeping precise financial records and updating balances in real time.
3. Shop Collection: This section lists products available for purchase, detailing names, categories, prices, and the stores selling them. It enables a dynamic shopping experience within the app.
4. Purchase Information Collection: It tracks each purchase made, including product details, buyer's account, quantity, total price, and the time of purchase. This supports the app’s e-commerce features by monitoring purchases and integrating them with overall account activities.

Together, these collections form a comprehensive database that supports secure data handling and enhances user interactions within our banking platform.

* **Google Cloud Vision API**: Used within free-tier limits for OCR processing.
* **Hugging Face Models**: Open-source models without additional costs.

1. **Cheque Processing:**

Initially, we experimented with various OCR models like Donut, BERT, custom models, and Tesseract for cheque deposit functionality. However, we faced challenges related to accuracy and integration complexity. To overcome these issues, we adopted the Google Cloud Vision API within the free-tier limits. This service provided reliable OCR capabilities and accurate extraction of cheque information, significantly improving the efficiency of the cheque deposit process.

1. **Predictive Expense Analysis:**

An obstacle we encountered was the insufficiency of internal transaction data for accurate expense prediction. To enhance the predictive analytics service, we incorporated external datasets, such as open financial datasets, to supplement our data. By integrating these datasets, we improved the accuracy of our predictive models, enabling us to offer valuable financial insights to users.

1. **Security Implementation:**

* JWT tokens used for secure API access.
* Passwords securely hashed using bcrypt.
* Implemented secure token handling and validation.

**2.4 URLs**

Source Code Repository: GitHub Repository

Web Service Access: (Provide the live URL if available, ensuring it's accessible within the free-tier limits)

**3. Discussion**

**3.1.1 Challenges Faced**

1. **Docker Image Size and Deployment Issues:**

One of the initial challenges was the large size of the Docker image due to heavy dependencies, which exceeded free-tier resource limits on GCP and caused deployment complications. To address this, we optimized the Docker images by using lightweight base images like python:3.9-slim. We removed unnecessary dependencies and files, and implemented multi-stage builds to minimize the image size. Additionally, we adjusted cloud configurations to conform to free-tier limitations. These optimizations allowed us to successfully deploy the application on GCP's free-tier without incurring additional costs.

1. **OCR Model Selection for Cheque Processing:**

Another challenge we faced was selecting an appropriate Optical Character Recognition (OCR) model for the cheque processing functionality. Initially, we experimented with various models, including Donut, BERT, custom models from Hugging Face, TrOCR, and Tesseract. However, these models presented issues either in terms of accuracy or integration complexity. The inconsistent results and integration challenges impeded our progress in delivering a reliable cheque processing feature. To resolve this, we shifted to using the Google Cloud Vision API within the free-tier usage limits. This service provided robust OCR capabilities with high accuracy and seamless integration into our application. By utilizing the Google Cloud Vision API, we enhanced the accuracy of the cheque processing feature and successfully extracted the necessary information from cheques without incurring additional costs.

1. **Data for Predictive Expense Analysis:**

Data sufficiency was another significant challenge, particularly concerning the predictive expense analysis feature. Our internal transaction history lacked sufficient volume and diversity to support accurate expense predictions. This limitation hindered our ability to provide users with meaningful insights into their future spending patterns. To overcome this, we incorporated external open-source datasets to supplement our data. Integrating these datasets enriched the predictive model's training process, enhancing its performance and accuracy. Consequently, we achieved more reliable expense predictions, offering valuable insights to users and enhancing the overall utility of our application.

**3.1.2 Development Practices**

To address the challenges encountered, we adopted several key development practices that proved instrumental in the project's success. Resource optimization was a central focus; we prioritized writing efficient code to minimize computational overhead, which is especially important when working within limited resource allocations. This involved refining algorithms, eliminating redundancies, and ensuring that our codebase was as lean as possible. We also optimized the usage of AI models to reduce memory consumption without compromising functionality or performance.

Leveraging free-tier cloud services required us to explore and utilize GCP's offerings thoroughly. We took the time to understand the limitations and capabilities of services like Google Cloud Run, Compute Engine, and the Vision API. By staying informed about service limitations, we avoided unexpected costs and made strategic decisions that aligned with our resource constraints. This informed approach allowed us to maximize the benefits of GCP's free-tier services effectively.

Implementing a Continuous Integration/Continuous Deployment (CI/CD) pipeline was another critical practice that enhanced our development workflow. Using GitHub Actions integrated with GCP, we automated the build, test, and deployment processes. This automation ensured consistent code integration, facilitated early detection of issues, and expedited deployment cycles. The CI/CD pipeline also improved collaboration among team members, as changes were continuously integrated and tested, reducing integration problems and enabling a more agile development process.

* + 1. **Lessons Learned**

The development of the Smart Banking System provided valuable insights into cost-effective application development within cloud environments. A significant lesson learned is that it is possible to build robust applications using free-tier services, provided that careful planning and optimization are applied. Understanding the specific limitations and capabilities of free-tier offerings is crucial; this knowledge allows developers to design solutions that fit within these constraints without sacrificing core functionalities.

We also recognized the importance of selecting an appropriate architectural approach. While microservices architectures offer advantages in scalability and flexibility, they may not always be practical for projects with limited resources or smaller scopes. In our case, a monolithic architecture was more suitable, simplifying the deployment process and reducing overhead. However, we learned that organizing the code into modular components is essential even within a monolithic structure to maintain code quality and ease of maintenance.

Effective resource management emerged as a critical success factor. Optimization must be considered at every stage, from writing efficient code to configuring deployment settings and cloud resources. This holistic approach to optimization ensures that applications remain performant and cost-effective, especially when operating under resource constraints like those of the free-tier services.

Finally, the selection of appropriate tools and services significantly impacts the project's outcomes. Choosing reliable and cost-effective solutions, such as the Google Cloud Vision API for OCR processing, can enhance functionality without incurring additional expenses. This approach allows developers to leverage existing, well-supported technologies, accelerating development and improving the quality of the application.

**4. Contribution of Each Member**

**ss**

|  |  |  |
| --- | --- | --- |
| **Name** | **Role** | **Contribution** |
| **Ashraf Uddin Chowdhury Rafat** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Proddut Kumar Biswas (40262586)** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Adilmanzoor Kalilur Rahman (40271551)** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| **Sai Abhinav Tadepalli (40257238)** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

**5. References**

1. Flask Framework Documentation
2. Google Cloud Platform Documentation
3. MongoDB Atlas Free Tier
4. Docker Documentation
5. GitHub Actions for CI/CD
6. Google Cloud Vision API
7. Hugging Face Models
8. Cybersecurity Best Practices for Web Developers

**6. Project Source Code**

GitHubRepository: <https://github.com/saiabhinav0000/COEN6313Project>

**7. Web Service Access**

Web Application URL: sss

Server:

<https://banking-server-320930579701.northamerica-northeast1.run.app>

Client (for testing API):ss

<https://storage.googleapis.com/smartbanking/Test.html>