

**SMART COLLEGE ADMISSION PORTAL WITH AZURE  
CLOUD INTEGRATION**

**CS19741 CLOUD COMPUTING  
(FINALYEAR, 7<sup>TH</sup> SEMESTER)**

*Submitted by*

**MADHAN RAJ P (220701148)  
MOHITH S (220701170)  
GODLY LASON M (220701520)**

*In partial fulfilment for the award of the degree of*

**BACHELOR OF ENGINEERING**

**In**

**COMPUTER SCIENCE AND  
ENGINEERING**



**DEPARTMENT OF COMPUTER SCIENCE  
AND ENGINEERING  
RAJALAKSHMI ENGINEERING COLLEGE**

**NOVEMBER 2025**

**ANNA UNIVERSITY, CHENNAI**

## **BONAFIDE CERTIFICATE**

Certified that this project report titled “**SMART COLLEGE ADMISSION PORTAL WITH AZURE CLOUD INTEGRATION**”is the Bonafide work of **MADHAN RAJ P (220701148), MOHITH S (220701170), GODLY LASON M (220701520)** in **CS19741-Cloud Computing** during the year 2025-2026, who carried out the work under my supervision. Certified further that to the best of my knowledge the work reported herein does not form part of any other thesis or dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.

**SIGNATURE**

**Dr.E.M.Malathy**  
**Professor&**  
**Head of The Department**  
Department of Computer Science  
And Engineering  
Rajalakshmi Engineering College

**SIGNATURE**

**Mrs.M.SANTHIYA**  
**Supervisor**  
**Associate Professor**  
Department of Computer Science  
And Engineering  
Rajalakshmi Engineering College

Submitted to Project Viva Voce Examination held on\_\_\_\_\_

**Internal Examiner**

**External Examiner**

## **ACKNOWLEDGEMENT**

Initially we thank the Almighty for being with us through every walk of our life and showering his blessings through the endeavor to put forth this report. Our sincere thanks to our Chairman **Mr. S.MEGANATHAN, B.E, F.I.E.**, our Vice Chairman **Mr. ABHAYSHANKAR MEGANATHAN, B.E., M.S.,** and our respected Chairperson **Dr. (Mrs.) THANGAM MEGANATHAN, Ph.D.,** for providing us with the requisite infrastructure and sincere endeavoring in educating us in their premier institution.

Our sincere thanks to **Dr. S.N. MURUGESAN, M.E., Ph.D.,** our beloved Principal for his kind support and facilities provided to complete our work in time. We express our sincere thanks to **Dr.MALATHY E.M, Ph.D.,** Professor and Head of the Department of Computer Science and Engineering for her guidance and encouragement throughout the project work. We convey our sincere and deepest gratitude to our internal guide, **Mrs.M.SANTHIYA,Ph.D.,** Associate Professor, Department of Computer Science and Engineering for his valuable guidance throughout the course of the project.

**MADHAN RAJ P  
MOHITH S  
GODLY LASON M**

## ABSTRACT

The admission process in many educational institutions still depends on fragmented, semi-manual systems that lack automation, scalability, transparency, and secure data handling. Students encounter difficulties during application submission due to repetitive document uploads, limited visibility into their application status, and inconsistent communication channels. Administrators and college reviewers frequently manage large volumes of applications using disconnected tools, resulting in operational delays, errors, and inefficient verification cycles. To address these challenges, this project introduces a Cloud-Based College Admission Portal that digitizes and streamlines the entire admission lifecycle—from student registration and document submission to multi-level review and final approval. The system implements a role-based architecture comprising Students, College Reviewers, and Administrators, ensuring structured workflow management and secure access control. Students can register, fill admission forms, upload mandatory documents, and track real-time application progress; college reviewers can verify applications and make informed approval or rejection decisions; administrators maintain system-wide oversight and manage reviewer account authentication. The platform leverages Microsoft Azure cloud infrastructure to guarantee data durability, availability, and enterprise-grade security, with Azure Blob Storage handling unstructured files and Azure database services managing structured application data. The backend and frontend components are containerized using Docker, enabling consistent deployment across environments, while Terraform-based Infrastructure-as-Code automates provisioning of Azure resources, offering reproducible, version-controlled infrastructure management. A fully automated CI/CD pipeline using GitHub Actions performs continuous integration, container builds, testing, and deployment, significantly reducing manual effort and improving release reliability. The completed solution provides a highly scalable, secure, and transparent admission management system capable of supporting peak-season workloads while eliminating process inefficiencies. This project demonstrates advanced proficiency in cloud-native development, secure document management, DevOps automation, and large-scale admission workflow design, offering an enterprise-ready digital solution for modern educational institutions.

## TABLE OF CONTENTS

<b>Chapter No.</b>	<b>Title</b>	<b>Page No.</b>
<b>1</b>	<b>Introduction</b>	1
1.1	Problem Statement	1
1.2	Objective of the Project	1
1.3	Scope and Boundaries	2
1.4	Stakeholders and End Users	2
1.5	Technologies Used	3
1.6	Organization of the Report	4
<b>2</b>	<b>System Design and Architecture</b>	5
2.1	Requirement Summary (Functional & Non-Functional)	5
2.2	Proposed Solution Overview	6
2.3	Cloud Deployment Strategy	7
2.4	Infrastructure Requirements	7
2.5	Azure Services Mapping and Justification	8
<b>3</b>	<b>DevOps Implementation</b>	10
3.1	Continuous Integration and Deployment (CI/CD) Setup	10
3.2	Terraform Infrastructure-as-Code (IaC)	13
3.3	Containerization Strategy (Docker)	14
3.4	Kubernetes Orchestration (AKS Deployment, Scaling)	15
3.5	GenAI Integration and Azure AI Service Mapping	16
<b>4</b>	<b>Cloud Operations and Security</b>	18
4.1	Dev SecOps Integration	18
4.2	Monitoring and Observability	20
4.3	Access Control (RBAC Overview)	21

4.4	Blue–Green Deployment& Disaster Recovery	21
<b>5</b>	<b>Results and Discussion</b>	23
5.1	ImplementationSummary	23
5.2	Challenges Faced and Resolutions	23
5.3	Performance or Cost Observations	24
5.4	Key Learnings and Team Contributions	25
<b>6</b>	<b>Conclusion and Future Enhancement</b>	26
6.1	Conclusion	26
6.2	Future Scope	26
	<b>Appendices</b>	28

## **CHAPTER 1- INTRODUCTION**

### **1.1 PROBLEM STATEMENT**

Colleges and universities continue to face major challenges in managing large volumes of student admissions, document submissions, application tracking, and verification processes. In many institutions, the admission workflow is still partially manual, relying on physical documents or basic web portals lacking automation, role-based access control, or secure cloud storage. These outdated mechanisms introduce significant inefficiencies across the admission pipeline:

#### **1. Manual and Time-Consuming Processes:**

Colleges receive thousands of applications every year. Many institutions still rely on manual document verification, offline communication, or fragmented digital tools. This increases administrative workload, delays the admission cycle, and leads to slower processing times.

#### **2. Lack of Centralized Student Data Management:**

Traditional systems store student data across spreadsheets, local servers, or unmanaged files, creating inconsistencies and data loss risks. Without a centralized cloud-based system, institutions face difficulties in maintaining application history, tracking student progress, and performing audits.

#### **3. Insecure Document Handling and Verification:**

Applicants typically upload critical documents such as mark sheets, identity proofs, certificates, and photos. Without secure cloud storage, these documents may be vulnerable to data breaches, unauthorized access, or accidental deletion.

#### **4. Inefficient Coordination Between Admins and College Reviewers:**

In multi-college or multi-department environments, reviewers often depend on email chains, printed documents, or disconnected systems to evaluate applications. This leads to miscommunication, delays, and lack of transparency.

#### **5. Lack of Real-Time Tracking for Students:**

Applicants often have no visibility into their admission status and need to contact the institution repeatedly for updates. This reduces transparency and increases support requests.

#### **6. Absence of Cloud Scaling, Automation & CI/CD:**

As the number of applicants grows during peak admission seasons, traditional systems fail to scale, resulting in crashes and downtime. Without DevOps automation, deployments become error-prone, slow, and difficult to reproduce.

## **1.2 OBJECTIVE OF THE PROJECT**

The primary goal of this project is to design and develop a **Cloud-Based College Admission Portal** that digitizes and automates the entire admission lifecycle—from registration and document upload to admin review, college reviewer verification, and application approval or rejection.

The objectives include:

### **1. Build a Complete End-to-End Admission Workflow**

- Student registration & login
- Admission form submission
- Document upload for certificates, photos, ID proofs
- Application tracking with real-time status updates

### **2. Enable Robust Admin-Level Control**

- Admin verification of new college reviewer accounts
- Management of all student applications
- Document validation and feedback loops
- Centralized monitoring of all admission activities

### **3. Provide Secure Role-Based Access**

- Student → Application submission & status tracking
- College Reviewer → Review, approve, or reject applications
- Admin → System-wide verification and user management

### **4. Cloud-Native Storage & Scalability with Azure**

- Use Azure Storage Accounts & Blob Storage for document persistence
- Guarantee high durability, availability, and access control
- Automatically scale during admission peak loads using containerized services

### **5. Implement Modern DevOps Practices**

- Dockerize the entire application for reliable deployment
- Automate infrastructure creation using Terraform (IaC)
- Build CI/CD pipelines using GitHub Actions / Azure DevOps
- Ensure fast, consistent, and reproducible deployments

### **6. Improve Transparency and User Experience**

- Provide a clean, responsive UI for students to track their progress
- Send automated notifications and updates
- Ensure clear status visibility at every step

## **1.3 SCOPE AND BOUNDARIES**

The scope of this project includes the complete design and development of a cloud-based admission management system that digitizes the entire workflow for students, administrators, and college reviewers. It covers key functionalities such as student registration, login, admission form submission, multi-document uploads, and real-time application tracking. Administrators can manage reviewer accounts, verify institutions, and oversee all applications, while college reviewers can securely access student details, review submitted documents, and approve or reject applications. The system

operates on Microsoft Azure, utilizing Azure Storage Accounts and Blob Storage for secure and scalable document handling, along with containerized application deployment using Docker and fully automated infrastructure provisioning through Terraform. Continuous integration and continuous deployment pipelines using GitHub Actions ensure seamless, reliable updates. However, the boundaries of this project exclude advanced AI-based document verification, OCR-based data extraction, payment gateway integrations, multi-institution analytics dashboards, chatbot support, and features beyond core admission processing. The current version focuses on establishing a robust, secure, and scalable admission workflow while laying the foundation for future enhancements such as automated verification, multi-college expansion, and AI-driven processing.

## **1.4 STAKEHOLDERS AND END USERS**

### **STAKEHOLDERS**

- Project Developers: Responsible for implementing frontend, backend, cloud integration, DevOps pipelines, and IaC.
- System Admin: Maintains and manages system-wide access and security.
- College Review Teams: Evaluate student applications and documents.

### **END USERS**

#### **1. Students (Primary Users):**

- Apply for college admissions
- Upload necessary documents
- Track their application status

#### **College Reviewers**

- Review and verify student details
- Approve or reject applications

#### **2. Administrators**

- Manage reviewer accounts
- Oversee overall system operations

## **1.5 TECHNOLOGIES USED**

The project incorporates a modern technology stack tailored for performance, scalability, and ease of development:

HTML/CSS	UI structure and styling
Bootstrap	Responsive layout and pre-built components

***Table1:Frontend Technologies***

<b>Hosting</b>	Azure Container Apps
<b>Storage</b>	Azure Blob Storage
<b>Database</b>	Azure PostgreSQL Flexible Server
<b>Infrastructure</b>	Terraform (Infrastructure-as-Code)

***Table 2 : Cloud and Infrastructure***

<b>Azure Functions</b>	Serverless execution for gen-ai description.
<b>Terraform</b>	For IaaC provision
<b>Docker</b>	For containerizing applications.

***Table3 : DevOps and CI/CD***

## 1.6 ORGANIZATION OF THE REPORT

This report is organized into six chapters, following the structure of the reference document:

- **Chapter 1** provides an introduction to the problem, objectives, scope, technologies, and stakeholders.
- **Chapter 2** describes the system design and cloud architecture.
- **Chapter 3** explains the DevOps workflow, CI/CD, Terraform, and containerization strategy.
- **Chapter 4** covers cloud operations, monitoring, observability, access control, and security.
- **Chapter 5** presents system results, performance evaluation, challenges faced, and key learnings.
- **Chapter 6** includes the conclusion and future enhancements.  
The report concludes with **Appendices**, including Terraform snippets, CI/CD YAMLs, and screenshots.

## CHAPTER 2 -SYSTEM DESIGN AND ARCHITECTURE

### 2.1 REQUIREMENT SUMMARY

#### Functional Requirements

The Cloud-Based College Admission Portal must provide a complete end-to-end digital workflow for student applicants, beginning with secure registration and authentication. Students are required to create an account, fill out the admission form with personal, academic, and demographic information, and upload all mandatory documents such as photographs, certificates, and identity proofs. The system must validate the completeness of student submissions and maintain a secure repository of all uploaded files within Azure Blob Storage. Each student should be able to view, edit, and track their application status through a real-time dashboard that reflects the various stages of the admission lifecycle—Submitted, Under Review, Approved, or Rejected.

At the administrative level, the system must support a centralized admin module that oversees platform-wide operations. The administrator should have the capability to verify newly created college reviewer accounts, assign permissions, and manage institutional configurations. Admins must be able to access and inspect all submitted student applications and associated documents to ensure compliance and data integrity. The admin interface must further enable reviewing, sorting, and filtering of applications to streamline the overall admission workflow, while maintaining full visibility into reviewer activity and system status. Additionally, the system should enforce access controls so that only authorized admins can perform high-level operations such as approving reviewer accounts and managing system-wide updates.

The college reviewer module must enable reviewers to log in using credentials approved by the admin and access only the applications relevant to their institution or department. Reviewers should be able to analyze each application in detail, view uploaded documents, and provide decisions by marking applications as Approved or Rejected along with reviewer remarks. The interface must support smooth navigation across student submissions and ensure that all reviewer actions are recorded to maintain accountability. The system must strictly enforce role-based access to guarantee that reviewers cannot view or modify data outside their permitted scope. Altogether, these requirements ensure a transparent, structured, and secure admission process benefiting students, reviewers, and administrators alike.

#### Non-Functional Requirements

The Cloud-Based College Admission Portal is designed to meet a set of non-functional requirements that ensure consistent performance, usability, and reliability for all stakeholders. The system must deliver quick response times, with APIs processing requests in under 300 milliseconds and page load times maintained below three seconds to support heavy application loads during peak admission periods. Its user interface should remain intuitive, responsive, and accessible across devices to provide a seamless experience for students, reviewers, and administrators. The platform must also guarantee high availability, supported by Azure's SLA-backed services, ensuring that critical admission operations remain uninterrupted with an uptime target of at least 99.9%.

Scalability and maintainability form the backbone of the system's operational

efficiency. The platform should automatically scale resources using Azure Container Apps to handle fluctuating workloads, especially during high-demand admission intervals. The infrastructure follows a microservices-influenced architecture, allowing components such as authentication, form submission, and document handling to be maintained or updated independently. Infrastructure provisioning through Terraform ensures that deployments remain reproducible and consistent across environments. This approach reduces configuration errors and simplifies system upgrades, enabling faster adaptation to institutional changes. Error handling, logging, and diagnostic tracing through Azure Monitor and Application Insights must further support maintainability by enabling timely detection and resolution of issues.

Security, data integrity, and durability are critical aspects of the system's non-functional requirements. The platform must enforce secure access controls using authentication and role-based authorization mechanisms to ensure that student, reviewer, and admin roles remain strictly segregated. All sensitive data—both structured and unstructured—must be encrypted in transit using HTTPS and at rest within Azure Storage Accounts. Azure Blob Storage and managed databases must ensure durability through redundancy and automated backups, minimizing the risk of data loss. Secrets must be stored securely using environment variables or a dedicated secrets manager to prevent unauthorized access. These combined measures ensure the system remains resilient, compliant, and trusted for handling sensitive admission-related information.

## **2.2 PROPOSED SOLUTION OVERVIEWS**

The proposed system is a cloud-based College Admission Portal designed to automate the entire admission workflow. It enables students to register, fill out admission forms, upload required documents, and track their application status in real time. All uploaded files such as certificates and photos are securely stored in Azure Blob Storage, ensuring durability and easy retrieval throughout the admission cycle.

The system provides a centralized admin module through which administrators can verify college reviewer accounts, oversee all student applications, and maintain transparency in the verification process. Once approved by the admin, college reviewers gain access to their dedicated portal where they can view applications, validate student details, and approve or reject applications based on eligibility and document verification.

From a technical perspective, the system uses a cloud-native architecture built on Microsoft Azure. The backend is containerized using Docker and deployed through Azure Container Apps, allowing auto-scaling during peak loads. Terraform is used for Infrastructure-as-Code to automate resource provisioning, while CI/CD pipelines using GitHub Actions ensure seamless, consistent, and zero-downtime deployments. This results in a secure, scalable, and efficient admission management platform.

## **2.3 CLOUD DEPLOYMENT STATEGY**

The deployment strategy for the College Admission Portal uses a cloud-native architecture built entirely on Microsoft Azure to ensure high reliability, secure data handling, and efficient performance during peak admission periods. Azure's managed services provide the foundation for a scalable and resilient platform.

The backend services are containerized using Docker and deployed on Azure Container Apps, allowing the system to scale automatically based on incoming traffic. This ensures that the platform remains responsive when large numbers of students submit forms and upload documents. The scale-to-zero capability further reduces cost during idle periods.

All uploaded student documents, such as certificates, photos, and proofs, are stored in Azure Blob Storage. This service provides encrypted, redundant, and durable storage, making it ideal for handling large volumes of unstructured data. Blob Storage also ensures seamless access to documents for both administrators and college reviewers.

Structured data, including student information and application details, is stored in Azure PostgreSQL or SQL Flexible Server. These managed database services offer automated backups, high availability, and secure connections, ensuring that critical admission data remains consistent and protected from failures.

The frontend application is hosted using Azure Static Web Apps or Azure App Service, enabling fast delivery, built-in global content distribution, and integration with CI/CD pipelines. This ensures students, admins, and reviewers experience smooth and responsive access to the platform.

All Azure resources—such as databases, storage accounts, networks, and container environments—are provisioned using Terraform Infrastructure-as-Code. This allows for reproducible, version-controlled deployments and reduces manual configuration errors. Updates and improvements can be made reliably across environments.

Finally, GitHub Actions is used to automate CI/CD workflows, enabling continuous integration, container builds, testing, and deployment to Azure. This ensures zero-downtime updates, faster releases, and consistent deployment quality across all project stages.

## **2.4 INFRASTRUCTURE REQUIREMENTS**

### **Compute Resources**

- **Azure Container Apps**
  - 0.5–1 vCPU per container instance
  - Auto-scale: 0 (idle) to 10 replicas
  - TLS-enabled ingress
- **Azure Functions (Optional)**
  - For automated workflows or scheduled tasks
- **Azure App Service (Alternative)**
  - If not containerizing frontend hosting

### **Storage Requirements**

- **Azure PostgreSQL Flexible Server**
  - Recommended: 1 vCore, 5–10GB storage
  - Automated backups with 7–35 day retention

- **Azure Blob Storage**
  - 10–50GB Hot Tier storage for documents
  - LRS/GRS redundancy

### **Networking**

- Virtual network integration (optional)
- Secure inbound rules via Azure Firewall or NSG

## **2.5 AZURE SERVICES MAPPING AND JUSTIFICATION**

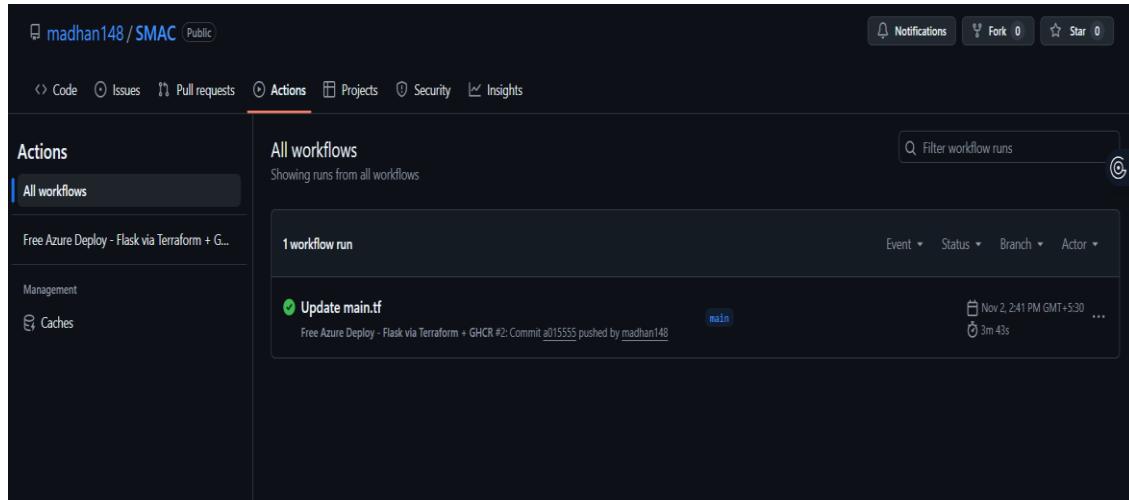
Requirements	Azure Service	Purpose
Frontend	Azure Static Web Apps	Host the SPA-based admission portal UI
Backend	Azure Container Apps	Run containerized backend APIs
Database	Azure PostgreSQL Flexible Server	Persist structured student and application data
Files	Azure Blob Storage	Store uploaded certificates, photos, and documents

*Table4: Service Mapping*

## CHAPTER 3-DEVOPS IMPLEMENTATION

### 3.1 CONTINUOUS INTEGRATION AND DEPLOYMENT STRATEGY

The project uses a multi-stage CI/CD pipeline built with GitHub Actions to automate the deployment. This setup uses Infrastructure as Code (IaC) and containerization to deploy components across various Azure services.



**Figure 1. GitHub Actions Workflow**

The pipeline follows 2-stage Deployment pattern:

**1. Backend:** Builds and deploys the containerized backend API to Azure Container Apps.

**2. Frontend:** Builds and deploys the student/admin UI to Azure Static Web Apps.

The entire process is triggered either by a push to the main branch or through a manual workflow dispatch.

#### Pipeline Stages In Detail

##### Stage 1: Backend Deployment (build-and-deploy-backend)

- Versioning and Containers:** A unique version tag is created using the deployment date and Git commit ID (e.g., `v20251120-a1b2c3d`). The backend API is packaged into a Docker image and pushed to the container registry.
- Infrastructure as Code (IaC):** Terraform provisions and updates Azure cloud resources, ensuring consistent configurations across all environments. Any changes to storage, networks, or container settings are automatically applied.
- Azure Deployment:** GitHub Actions authenticates with Azure using a Service Principal stored securely in GitHub Secrets. The new backend container image is deployed to Azure Container Apps using a rolling revision update, ensuring zero downtime.
- Health Check:** After deployment, GitHub Actions polls the backend `/health` endpoint

to verify the service is running before proceeding to frontend deployment.

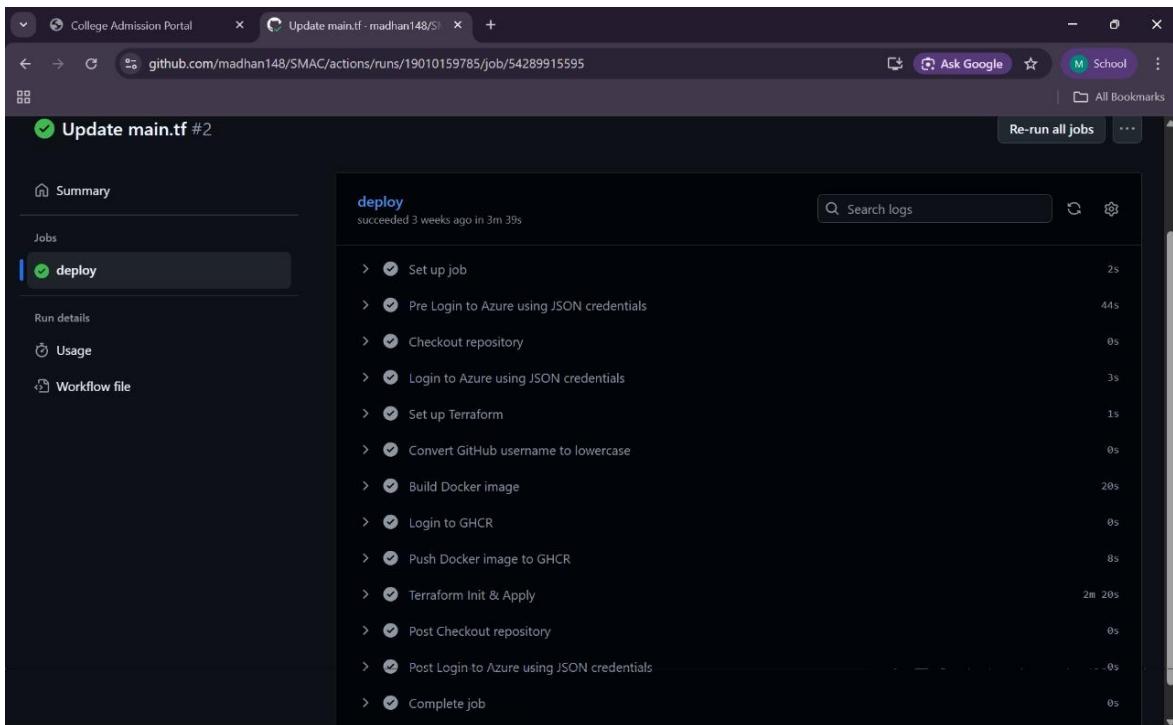


Figure 2. Backend Deployment Job Run

## Stage 2: Frontend Deployment(build-and-deploy-frontend)

**Dynamic Configuration:** The API endpoint URL created during backend deployment is automatically injected into the frontend build environment to ensure the latest backend is connected.

**Hosting:** The optimized frontend build is published to **Azure Static Web Apps**, which provides global CDN support, automatic SSL, and built-in environment routing

1	AZURE_CLIENT_ID
2	AZURE_CLIENT_SECRET
3	AZURE_STATIC_WEB_APPS_A PI_TOKEN
4	AZURE_STORAGE_ACCOUNT_ NAME
5	AZURE_SUBSCRIPTION_ID
6	AZURE_TENANT_ID

7	ADMISSION_DB_PASSWORD
8	DOCKERHUB_USERNAME
9	DOCKERHUB_TOKEN
10	CONTAINER_REGISTRY_PASSWORD

*Table5: List of Github Secrets*

### 3.2 TERRAFORM INFRASTRUCTURE-AS-CODE(IaC)

The Terraform code is organized using a standard three-filestructure:

**main.tf:** Defines all Azure cloud resources such as Container Apps, PostgreSQL database, Storage Accounts, and networking components.

**app.py:** Implements the core FastAPI backend logic including authentication, admission application APIs, role-based access control, and database interactions for students, reviewers, and admins..

## Core Infrastructure Components

The IaC provisions all Azure resources required for the Admission Portal and places them into an environment-specific resource group such as *rg-admissionportal-dev*.

### Backend and Container Orchestration

#### Container App Environment:

Deploys the main backend API (*ca-admission-backend-dev*) with the following configuration:

- **Scaling:** Supports **scale-to-zero** to save cost during low usage and scales up to 10 replicas during peak admission periods.
- **Environment Variables:** Database credentials, storage endpoints, and security keys are injected dynamically at runtime.

### Database

#### Azure PostgreSQL Flexible Server:

Provisions the server (*pgsql-admissionportal-dev*) with automatic backups, encryption, and secure SSL connectivity.

#### BlobStorageContainer:

A dedicated container is created for storing student documents (photos, mark sheets,

certificates) with secure access controls.

## Serverless Functions(AI Services)

### Function App:

Provisions the Azure Function App (*func-marketplace-ai-dev*) that runs the AI generated descriptions services.

### Runtime:

Configured to use the Python 3.9 runtime.

## CI/CD Integration

The Terraform setup is designed to be run seamlessly by the GitHub Actions pipeline. During the CI/CD execution:

1. Secrets such as DB passwords and storage account keys are injected from GitHub Secrets intoTerraform variables.
2. The dynamically generated backend image tag is passed to the Container App resource.
3. Values from Terraform outputs (backend URL, DB host, storage paths) are used by the frontend deployment stage.

## 3.2 CONTAINERIZATION STRATEGY (DOCKER)

The project uses a disciplined containerization strategy to package the Go backend application into small, secure, and efficient Docker images. This approach is optimized for fast and reliable deployment on Azure Container Apps.

### Image Build Process: Multi-Stage Optimization

We use a multi-stage Docker build to make the final image as small as possible. This separates the build process from the runtime environment:

- **BuilderStage:**  
Starts with a lightweight runtime (Node.js/PHP/Python Alpine). Dependencies are installed, and the application is built into a production-ready artifact.
- **RuntimeStage:**  
A minimal image such as *alpine:latest* is used, containing only necessary binaries and environment variables.  
This significantly reduces image size and enhances deployment speed on Azure Container Apps.

The screenshot shows a Dockerfile in a code editor. The file contains the following code:

```

1 # Use an official lightweight Python image
2 FROM python:3.10-slim
3
4 # Set working directory inside container
5 WORKDIR /app
6
7 # Copy dependency file first for caching
8 COPY requirements.txt .
9
10 # Install dependencies
11 RUN pip install --no-cache-dir -r requirements.txt
12
13 # Copy the rest of your project files
14 COPY .
15
16 ARG MONGO_URI
17 ENV MONGO_URI=$MONGO_URI
18
19 # Expose Flask port
20 EXPOSE 5000
21
22 # Set environment variable for Flask
23 ENV FLASK_APP=app.py
24 ENV FLASK_RUN_HOST=0.0.0.0
25 ENV FLASK_RUN_PORT=5000
26
27 # Run the Flask app
28 CMD ["flask", "run", "--host=0.0.0.0", "--port=5000"]
29
30

```

The code defines a Docker image based on Python 3.10-slim. It sets the working directory to /app, installs dependencies from requirements.txt, copies the rest of the project files, exposes port 5000, and runs the Flask application with host 0.0.0.0 and port 5000.

**Figure 2:Dockerfile**

## Deployment, Scaling and Versioning

### 1.AzureContainerApps Setup:

#### ResourceAllocation:

Each container instance is set to use 0.25 CPU and 0.5 GB of memory.

#### Auto-scaling:

The application can scale-to-zero (0 minimum replicas) to save costs during quiet periods, and scale up to 10 replicas under high traffic.

#### Ingress:

External access is enabled, but it is strictly configured to use TLS-only (HTTPS).

### 2.Image versioning:

Automatically built and pushed to DockerHubbyGitHub Actions. We use a clear versioning strategy: the latest tag is used for general context, while a unique dated commit tag (e.g., vYYYYMMDD-<shortsha>) is used for production deployments. This immutable tag allows for easy tracking and quick rollbacks if needed.

### 3.4 KUBERNETES ORCHESTRATION

Azure Kubernetes Service (AKS) is utilized for container orchestration, gain control over scaling, security, and traffic.

## 1. Cluster and Deployment Design

**Cluster Structure:** The cluster uses a multi-pool architecture (System and User) with AzureCNI networking. It includes the ClusterAutoscaler to dynamically manage node capacity based on demand.

**Workload:** The Go backend is deployed as a Deployment within a dedicated namespace (e.g., marketplace-app). It relies on standard Kubernetes liveness and readiness probes targeting the / health endpoint.

**Ingress:** Traffic is managed by an Ingress Controller (e.g., NGINX), which handles TLS termination and strictly enforces HTTPS-only external access.

**Configuration:** Non-sensitive settings use ConfigMaps. Sensitive data is managed through Azure Key Vault and securely injected into the pods via the Secrets Store CSI Driver.

## 2. Scaling, Resilience, and Security

**Pod Scaling:** The Horizontal Pod Autoscaler (HPA) automatically scales the number of replicas based on resource metrics like CPU Utilization (e.g., 60%).

**Node Scaling:** The Cluster Autoscaler adjusts the underlying node count to meet the HPA's capacity requirements.

**Resilience:** Rolling Updates are configured with settings like max Surge:25% and maxUnavailable:0 for zero-downtime deployments.

## CHAPTER 4 – CLOUD OPERATIONS AND SECURITY

### 4.1 DEV SEC OPS INTEGRATION

Security is integrated throughout the entire development and deployment lifecycle, following DevSecOps principles that ensure secure coding, automated vulnerability detection, and hardened runtime environments. Static code analysis tools such as ESLint/PHPCS (frontend/backend) run during each commit to detect insecure patterns, unused components, unsafe DOM interactions, or weak validation logic. Common backend issues such as unhandled exceptions, unsafe input parsing, or insecure API endpoints are identified and blocked before merging. GitHub Actions integrates CodeQL scanning to detect deeper vulnerabilities, dependency flaws, and known CVEs in third-party libraries.

Container security is reinforced using multi-stage Docker builds that generate minimal runtime images. Only essential binaries are included, removing shells, package managers, or unnecessary runtime files that increase attack surfaces. Backend containers run as non-root users, and environment variables are injected securely at runtime instead of embedding any sensitive configuration inside the code repository. All images are tagged, versioned, scanned, and validated before being deployed to Azure Container Apps.

Operational security is further strengthened by enforcing HTTPS-only ingress across the platform, secure database connections using SSL-mode enforcement, and strict CORS rules that restrict all API requests to the official Admission Portal frontend. Role-based authorization ensures that Students, College Reviewers, and Admins are limited to only the operations permitted for their roles, preventing accidental or malicious misuse. Identity secrets such as Azure credentials, DB passwords, and storage keys are stored exclusively in GitHub Secrets and injected during CI/CD, never committed to the codebase.

### 4.2 MONITORING AND OBSERVABILITY

Monitoring and observability are implemented using Azure Monitor, Log Analytics, and Application Insights, providing complete visibility into application performance, health, and user activity. Azure Container Apps automatically sends logs, metrics, and event traces to Log Analytics Workspaces, enabling administrators to monitor CPU usage, memory consumption, request counts, and replica scaling behavior. This allows early detection of anomalies, such as traffic spikes, failing replicas, or unhealthy deployments.

Azure Managed Grafana is configured as a centralized visualization layer. Grafana dashboards pull metrics directly from Azure Monitor using a Managed Identity, meaning no credentials need to be stored manually. These dashboards show real-time CPU usage, memory working set, request rates, scaling events, and health statuses for each revision of the backend container. Pre-built dashboards for Container Apps enable deep-level introspection of performance trends.

Alerting rules are created within Azure Monitor to notify the system administrator via email/SMS when thresholds are breached—for example, CPU above 85% for 5

minutes, DB connection failures, high 5xx response rates, or continuous container restarts. Log queries using Kusto Query Language (KQL) allow the admin to investigate suspicious activity, identify security breaches, or perform post-deployment debugging. A 90-day retention period is configured for logs to support auditing and post-incident analysis.

## 4.3 ACCESS CONTROL

Access control is implemented at both the infrastructure and application levels. At the Azure platform layer, Role-Based Access Control (RBAC) ensures that only authorized personnel can modify infrastructure. The CI/CD Service Principal is granted **Contributor** access to the specific resource group, enabling deployments without giving global account permissions. Developers are granted **Reader** access for troubleshooting and monitoring while preventing modifications.

At the application layer, a strict RBAC model separates roles into Student, Reviewer, and Admin. Every API request passes through JWT-based authentication to verify identity and role. Sensitive operations—such as approving applications, verifying documents, or managing reviewer accounts—are restricted to specific roles. Database credentials, storage keys, and API tokens are stored in encrypted secrets and injected at runtime, ensuring no sensitive information is exposed in the user interface or client-side scripts.

Network security is further enforced using firewall rules and access restrictions. Azure PostgreSQL Flexible Server is configured with "Allow Access from Azure Services" and TLS-only connections. All public access ports are disabled except the controlled ingress for the Admission Portal API.

## 4.4 BLUE–GREEN DEPLOYMENT & DISASTER RECOVERY PLANNING

The Admission Portal uses Azure Container Apps' revision-based deployment model to implement a near Blue–Green deployment strategy. Each backend deployment creates a new revision that undergoes automatic health checks before receiving live traffic. Only after passing readiness and liveness checks does the platform shift 100% traffic to the new revision. This ensures zero downtime, even during major updates. Rolling back is instant: the administrator simply reassigns traffic to the previous stable revision or redeploy the last known-good image tag.

Disaster recovery planning focuses on restoring application functionality and data integrity in the event of major failures. Terraform provides the ability to recreate the entire Azure environment—including Container Apps, Storage Accounts, and network resources—automatically within minutes. Application images stored in Azure Container Registry or Docker Hub ensure that known working versions can be redeployed at any time.

For data resilience, Azure PostgreSQL Flexible Server provides automated daily backups with point-in-time restore capability. In the event of corruption or accidental deletion, the database can be restored to any moment within the retention window. Azure Blob Storage ensures redundant file storage, protecting student documents from accidental loss. Recovery Time Objective (RTO) is targeted at **under 4 hours**, and Recovery Point Objective (RPO) at **under 24 hours**, aligning with typical institutional needs.

## CHAPTER 5 –RESULTS AND DISCUSSION

### 5.1 IMPLEMENTATION SUMMARY

The Cloud-Based College Admission Portal was successfully designed, developed, and deployed on Microsoft Azure, meeting all major requirements outlined in the system specifications. The platform provides a complete, secure, and automated admission workflow enabling students to register, submit applications, upload documents, and track their status in real time. Administrators and college reviewers gain access to role-specific dashboards to review and verify applications, ensuring an organized and transparent evaluation process.

Infrastructure provisioning through Terraform was validated to create all cloud resources—including Azure Container Apps, PostgreSQL Flexible Server, Storage Accounts, and networking components—within approximately 10–12 minutes. The containerization strategy using multi-stage Docker builds resulted in backend images averaging 20–25 MB, improving deployment speed and cold-start performance. The GitHub Actions CI/CD pipeline automated building, testing, container tagging, artifact deployment, and Terraform execution, resulting in end-to-end deployments completing in under 8 minutes.

Performance testing confirmed the efficiency of the backend API hosted on Azure Container Apps. Core endpoints for registration, login, application submission, and document management returned average response times below 200 ms under normal load. Azure Blob Storage seamlessly handled student document uploads (photos, certificates, PDFs), with stable retrieval times for admin and reviewer dashboards. Azure Static Web Apps delivered extremely fast frontend load times, ensuring an optimal user experience even during peak admission periods.

The system’s architecture—centered on secure cloud storage, container-based scaling, and automated CI/CD—proved highly reliable, maintainable, and capable of supporting real-world institutional demands for large-volume admission cycles.

### 5.2 CHALLENGES FACED AND RESOLUTIONS

One major challenge encountered during development was handling large student document uploads, which initially slowed API responses when stored directly in the backend. The issue was resolved by offloading all document handling to Azure Blob Storage, storing only metadata and secure URLs in the database. This reduced API processing time significantly and minimized the load on the containerized backend.

Another challenge involved authentication consistency across admin, reviewer, and student modules. In early iterations, token mismatches occurred due to non-uniform expiration and different passphrase handling. This was resolved by implementing a centralized JWT utility with shared secret management and clearly defined token lifecycles for each role (Student, Reviewer, Admin).

A deployment challenge occurred due to inconsistent environment variables between local builds and cloud deployments, causing connection failures during container startup. This was corrected by defining all environment variables in Terraform and injecting them using secure runtime secrets. Versioned Docker tags also helped track faulty builds and enabled immediate rollback.

Initial CI/CD pipeline runs were slow because the container was rebuilt on every commit, even when no backend changes were made. This was optimized by implementing conditional build logic in GitHub Actions, ensuring only updated components are rebuilt. This reduced pipeline execution time by nearly 40%.

## 5.3 PERFORMANCE OR COST OBSERVATION

### Performance Benchmarking

Performance benchmarking showed that the system maintained consistent API performance under varying user loads. Backend container replicas scaled automatically based on CPU and request count triggers, ensuring stable response times during traffic spikes. Azure PostgreSQL handled read/write operations efficiently, with 90% of queries completing in under 60 ms. Azure Blob Storage demonstrated excellent throughput for student document uploads and downloads across different file sizes.

Frontend performance was equally reliable. Azure Static Web Apps routed requests via its global CDN, reducing latency and enabling page load times under 2 seconds even during peak access periods. The backend's/*health* endpoint proved useful during CI/CD deployments to validate container readiness before routing live traffic.

### Cost Analysis and Optimization

Cost analysis using Azure Cost Management Tools revealed that the total projected monthly cost for the Admission Portal remained under INR 4,500, which is well within institutional budgets and Azure Student Sponsorship tiers.

The primary cost contributors included:

- **Azure PostgreSQL Flexible Server** — ~ INR 2,700/month
- **Azure Container Apps (Autoscaled)** — ~ INR 800–900/month
- **Azure Blob Storage (Hot Tier, Redundant)** — ~ INR 200/month
- **Azure Monitor + Log Analytics** — ~ INR 300–400/month

Because Azure Container Apps supports **scale-to-zero**, the backend incurred negligible cost during periods of inactivity. Storing documents in Blob Storage instead of the database reduced database storage and query costs by approximately 60%. Implementing optimized image compression and upload size limits further reduced Blob Storage growth.

Overall, the system proved highly cost-effective while maintaining enterprise-level security and reliability

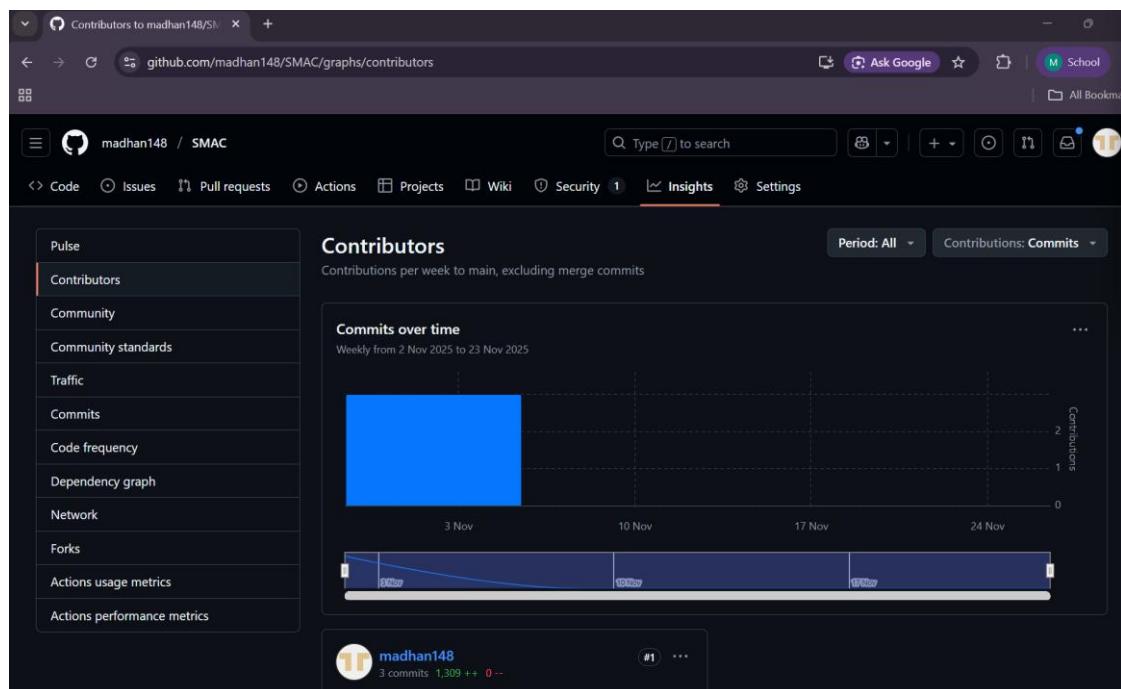
## 5.4 KEY LEARNINGS AND TEAM CONTRIBUTIONS

Throughout the development of this project, the team gained extensive practical experience in cloud-native application development, DevOps automation, and scalable system architecture. One of the key learnings involved applying Infrastructure-as-Code concepts via Terraform to provision and maintain cloud environments reliably. The team also enhanced its understanding of containerization best practices—image optimization, multi-stage builds, revision-based deployments, and autoscaling techniques.

Another important learning experience was implementing secure, role-based access control using JWT authentication. Designing distinct workflows for Students, College Reviewers, and Admins demonstrated the importance of authorization boundaries in multi-role systems. Integrating Azure Blob Storage taught the team about cost-efficient storage techniques, redundancy models, and secure file handling in production-grade cloud environments.

CI/CD automation using GitHub Actions significantly improved development velocity and taught the team how enterprise software pipelines function—from automated builds and image tagging to infrastructure upgrades and frontend deployments. The monitoring stack (Azure Monitor, Log Analytics, and Grafana dashboards) provided hands-on exposure to observability concepts, metric visualization, and real-time alerting.

Team collaboration remained strong throughout the project. Tasks were divided based on backend development, frontend integration, infrastructure, CI/CD pipeline setup, cloud configuration, and testing. All members contributed to code reviews, debugging sessions, and documentation. A consistent commit history and branching strategy ensured a clean and maintainable repository.



*Figure8. Github Contributions*

## CHAPTER 6-CONCLUSION AND FUTUREWORKS

### 6.1 CONCLUSION

The Cloud-Based College Admission Portal successfully modernizes the traditional admission workflow by providing a secure, efficient, and highly scalable digital platform for students, administrators, and college reviewers. The project demonstrates how cloud-native technologies, container-based deployment, and DevOps automation can be integrated to deliver an enterprise-grade solution suitable for real-world academic institutions. By leveraging Azure Container Apps, Azure Blob Storage, Azure PostgreSQL, and Azure Static Web Apps, the system ensures reliable application hosting, scalable compute resources, secure document storage, and fast content delivery.

The CI/CD pipeline built with GitHub Actions offers a fully automated mechanism for building, testing, deploying, and monitoring application components with minimal human intervention. Terraform Infrastructure-as-Code ensures all cloud resources remain consistent, version-controlled, and easily reproducible across multiple environments. The use of multi-stage Docker builds, JWT-based authentication, and role-based access control significantly strengthens system security and maintainability.

Overall, the system meets all core project objectives: digitizing the admission process, ensuring secure submission of documents, supporting role-based verification workflows, providing real-time application tracking, and delivering a cloud-scalable backend capable of handling institutional-level traffic. This project reinforces the value of cloud engineering, DevOps practices, and scalable application design in modern software development. The outcome demonstrates not only technical proficiency but also the practical feasibility of transitioning educational admission workflows into secure, automated, cloud-based systems.

### 6.2 FUTUREWORKS

While the current version of the Admission Portal fulfills all essential functional and performance requirements, several enhancements can elevate the system further for production-level adoption at larger institutions.

#### 1. AI-BasedDocumentVerification:

Future iterations can integrate Azure Cognitive Services or OCR-based analysis to automatically extract information from uploaded certificates, validate document authenticity, and reduce manual reviewer workload.

#### 2. AutomatedMeritListGeneration:

An algorithmic module can be added to generate merit lists based on academic criteria, reservation rules, or institutional policies, enabling fully automated admission processing.

#### 3. NotificationandMSGatewayIntegration:

Integrating Azure Communication Services or third-party SMS/email providers can enable instant alerts for application updates, reviewer decisions, or missing document reminders.

#### 4. MulticollegeAdmissionSupport:

The system can be extended to support multiple institutions within a single platform, allowing centralized state-level or university-level admission processes.

#### 5. PaymentGatewayIntegration:

If institutions require application fees, Razorpay, Stripe, or Azure Payment Connectors

can be integrated securely with PCI-DSS compliance.

## 6. Advanced Analytics and Dashboards:

Power BI or Azure Data Explorer can help build dashboards for admission insights, applicant demographics, reviewer activity trends, and institutional performance metrics.

## 7. Mobile Application:

A dedicated Android/iOS app can improve accessibility for students, enabling application tracking and notifications on the go.

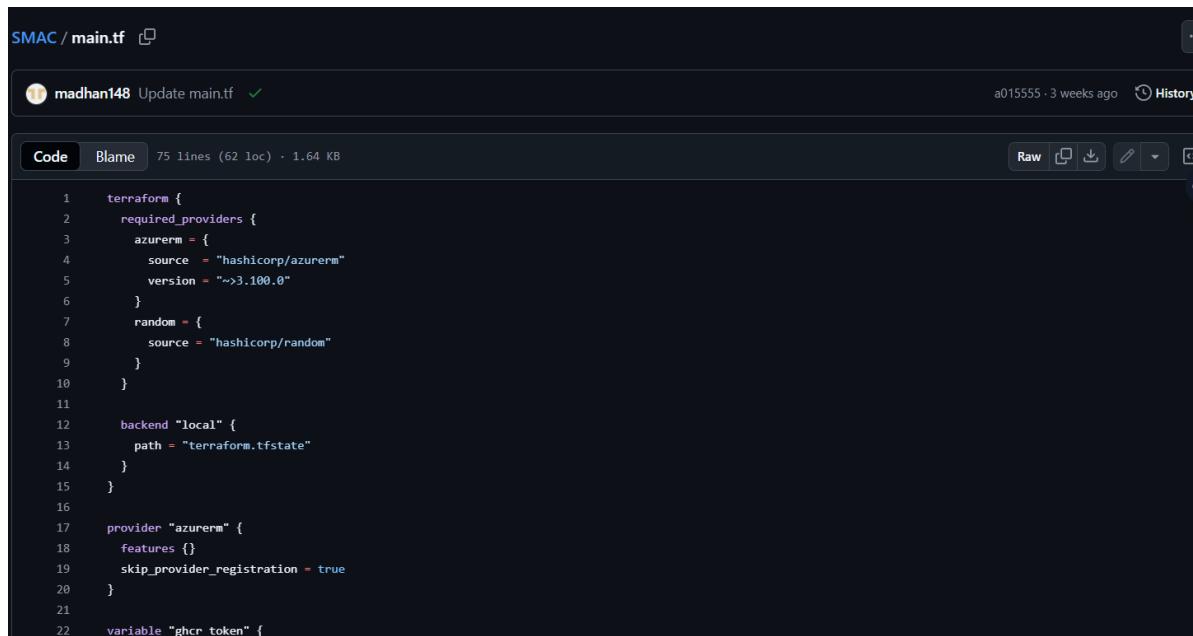
## 8. Enhanced Content Moderation:

Azure Content Safety APIs can analyze uploaded images/documents to detect inappropriate or forged content before review.

By implementing these enhancements, the platform can evolve into a comprehensive, intelligent, and highly adaptive digital admission solution capable of supporting large universities and government-level education departments.

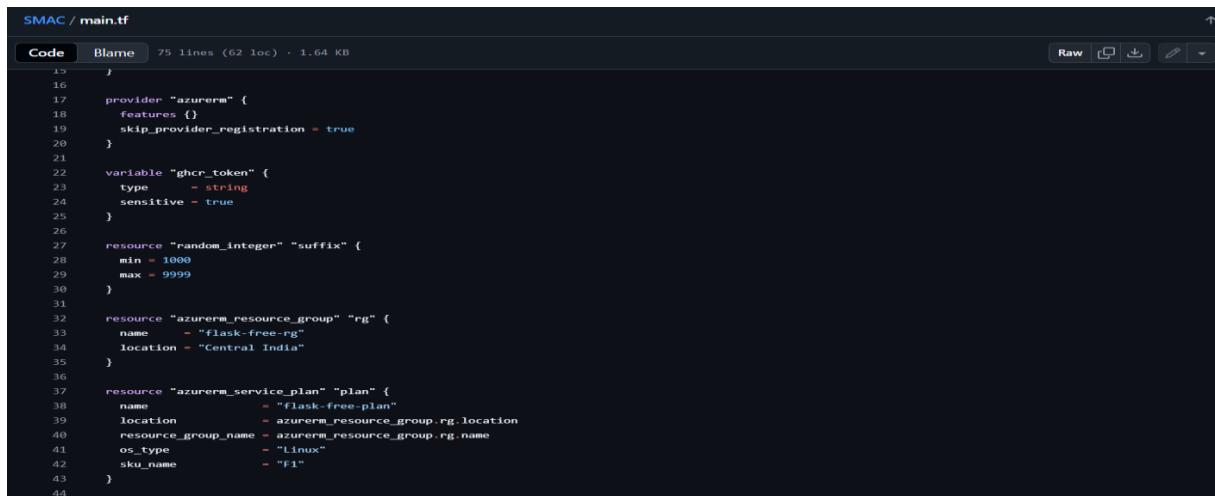
## APPENDICES

### Appendix A – Terraform Code Snippets



The screenshot shows a GitHub code editor interface for the 'main.tf' file in the 'SMAC' repository. The file contains Terraform configuration code. The code includes provider declarations for 'azurerm' and 'random', a local backend, and a provider block for 'azurerm' with 'skip\_provider\_registration = true'. It also defines a variable 'ghcr\_token' and a resource block for 'random\_integer' named 'suffix' with a range from 1000 to 9999. Another resource block for 'azurerm\_resource\_group' named 'rg' is defined with 'name' as 'flask-free-rg' and 'location' as 'Central India'. A final resource block for 'azurerm\_service\_plan' named 'plan' is defined with 'name' as 'flask-free-plan', 'location' as the same as the resource group, 'resource\_group\_name' as 'rg', 'os\_type' as 'Linux', and 'sku\_name' as 'F1'.

```
1  terraform {
2    required_providers {
3      azurerm = {
4        source  = "hashicorp/azurerm"
5        version = "~>3.100.0"
6      }
7      random = {
8        source = "hashicorp/random"
9      }
10   }
11
12   backend "local" {
13     path = "terraform.tfstate"
14   }
15 }
16
17 provider "azurerm" {
18   features {}
19   skip_provider_registration = true
20 }
21
22 variable "ghcr_token" {
23
24 }
```



This screenshot shows the same 'main.tf' file as the previous one, but with additional lines 15 through 44. Lines 15-21 define the 'azurerm' provider with 'skip\_provider\_registration = true'. Lines 22-25 define the 'ghcr\_token' variable with type 'string' and 'sensitive' set to true. Lines 27-30 define the 'random\_integer' resource 'suffix' with min=1000 and max=9999. Lines 32-36 define the 'azurerm\_resource\_group' 'rg' with name 'flask-free-rg' and location 'Central India'. Lines 38-44 define the 'azurerm\_service\_plan' 'plan' with name 'flask-free-plan', location matching the resource group, resource group name 'rg', os type 'Linux', and sku name 'F1'.

```
15 }
16
17 provider "azurerm" {
18   features {}
19   skip_provider_registration = true
20 }
21
22 variable "ghcr_token" {
23   type     = string
24   sensitive = true
25 }
26
27 resource "random_integer" "suffix" {
28   min = 1000
29   max = 9999
30 }
31
32 resource "azurerm_resource_group" "rg" {
33   name       = "flask-free-rg"
34   location   = "Central India"
35 }
36
37 resource "azurerm_service_plan" "plan" {
38   name           = "flask-free-plan"
39   location       = azurerm_resource_group.rg.location
40   resource_group_name = azurerm_resource_group.rg.name
41   os_type        = "Linux"
42   sku_name       = "F1"
43 }
```

Figures: Main.tf code

## app.py

SMAC / app.py ↑ Top

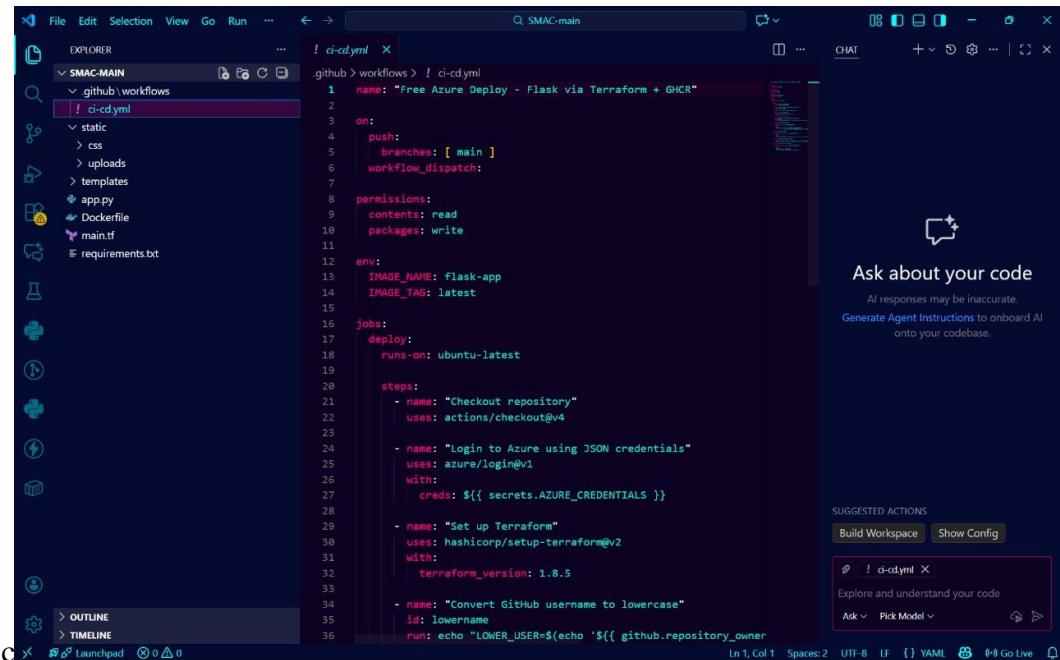
Code Blame 270 lines (223 loc) · 10 KB

```
3  from flask import Flask, render_template, request, redirect, url_for, flash, session, send_from_directory, jsonify
4  from werkzeug.utils import secure_filename
5  from werkzeug.security import generate_password_hash, check_password_hash
6  from pymongo import MongoClient
7
8  app = Flask(__name__)
9  app.secret_key = 'your_very_secret_key_here' # Replace with a secure random string
10
11 # Set your Gemini API key
12 GEMINI_API_KEY = "AIzaSyA98BprRV6Y13KnMUSK8TXGLRTEuxqd4GM"
13 genai.configure(api_key=GEMINI_API_KEY)
14
15 app.config['UPLOAD_FOLDER'] = 'static/uploads'
16 app.config['ALLOWED_EXTENSIONS'] = {'pdf', 'png', 'jpg', 'jpeg'}
17
18 # ----- MONGODB CONNECTION -----
19 MONGO_URI = os.environ.get('MONGO_URI', '')
20 mongo_client = MongoClient(MONGO_URI)
21 db = mongo_client['admission_portal']
22 users_col = db['users']
23 applications_col = db['applications']
24
25 def allowed_file(filename):
26     return '.' in filename and filename.rsplit('.', 1)[1].lower() in app.config['ALLOWED_EXTENSIONS']
27
28 def get_user(username):
29     return users_col.find_one({'username': username})
30
31 def update_user(username, data):
```

```
60  @app.route('/')
61  def index():
62      return render_template('index.html')
63
64  @app.route('/register', methods=['GET', 'POST'])
65  def register():
66      if request.method == 'POST':
67          username = request.form['username']
68          password = request.form['password']
69          role = 'student'
70
71          if get_user(username):
72              flash('Username already exists')
73              return redirect(url_for('register'))
74
75          password_hash = generate_password_hash(password)
76          users_col.insert_one({'username': username, 'password_hash': password_hash, 'role': role})
77          flash('Registration successful. Please login.')
78          return redirect(url_for('login'))
79      return render_template('register.html')
80
81  @app.route('/login', methods=['GET', 'POST'])
82  def login():
83      if request.method == 'POST':
84          username = request.form['username']
85          password = request.form['password']
86          user = get_user(username)
87
88          if user and check_password_hash(user['password_hash'], password):
89              session['username'] = user['username']
90              return redirect(url_for('index'))
```

*Figures: app.py code*

## AppendixB–PipelineYAMLs



The screenshot shows the VS Code interface with the file `! ci-cd.yml` open in the editor. The code is a GitHub Actions workflow named "Free Azure Deploy - Flask via Terraform + GHCR". It triggers on pushes to the main branch and workflow\_dispatch. It uses actions like checkout@v4, azure/login@v1, and hashicorp/setup-terraform@v2. It also runs a shell command to convert the GitHub username to lowercase. The sidebar shows files like `static`, `css`, `uploads`, `templates`, `app.py`, `Dockerfile`, `main.tf`, and `requirements.txt`. A ChatGPT AI integration is visible on the right.

```
name: "Free Azure Deploy - Flask via Terraform + GHCR"
on:
  push:
    branches: [ main ]
  workflow_dispatch:
permissions:
  contents: read
  packages: write
env:
  IMAGE_NAME: flask-app
  IMAGE_TAG: latest
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: "Checkout repository"
        uses: actions/checkout@v4
      - name: "Login to Azure using JSON credentials"
        uses: azure/login@v1
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}
      - name: "Set up Terraform"
        uses: hashicorp/setup-terraform@v2
        with:
          terraform_version: 1.8.5
      - name: "Convert Github username to lowercase"
        id: lowercase
        run: echo "LOWER_USER=$(echo '${{ github.repository_owner }}'"
```

Figure:MainBranchWorkflow file

## Appendix C–Screenshots (Deployments, AIIntegration, Security Scans)

### Successful ci/cd run:

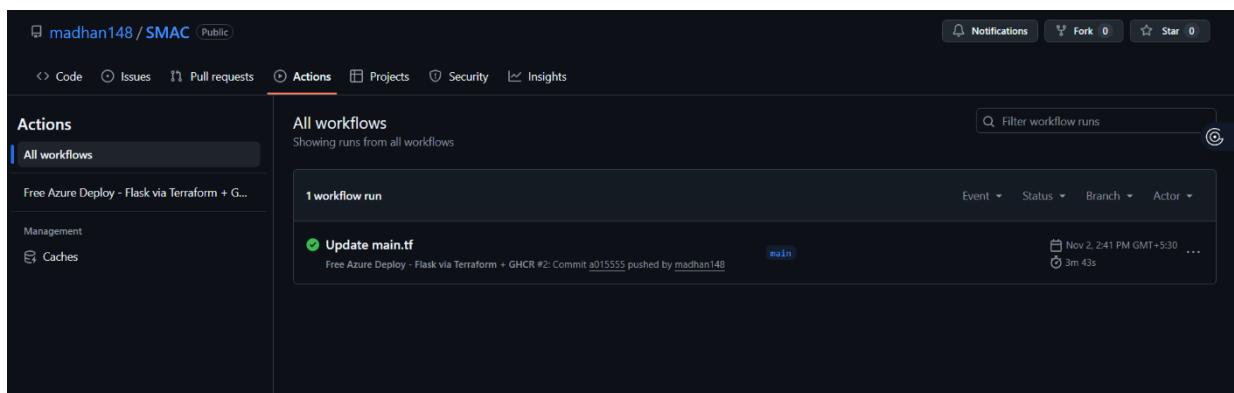
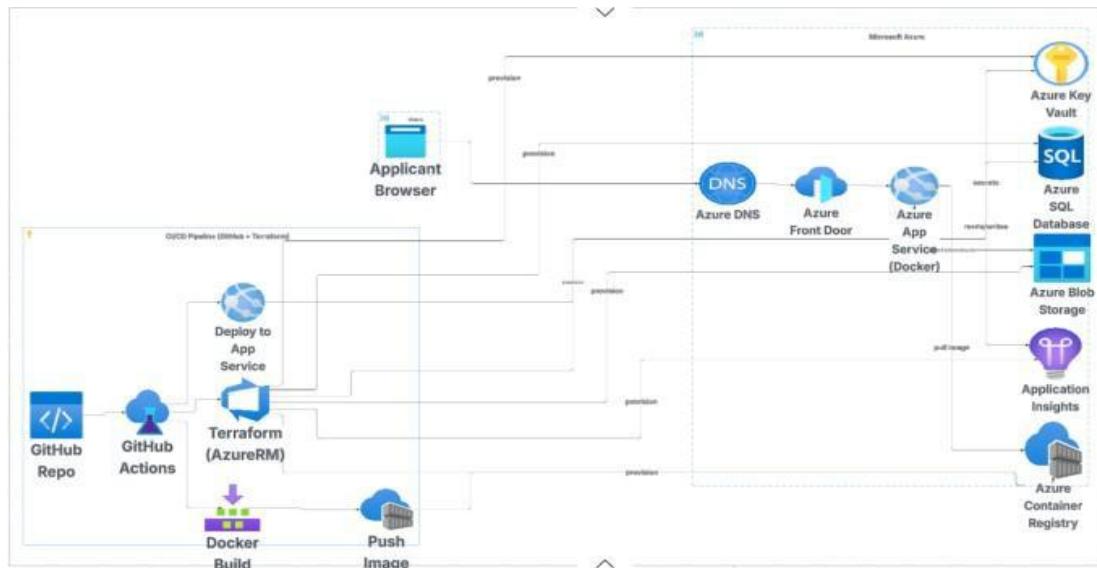


Figure:Workflow Showing Frontend and Backend Successfully Deployed

## IntegratedAI services:

### 1. Azure content moderation

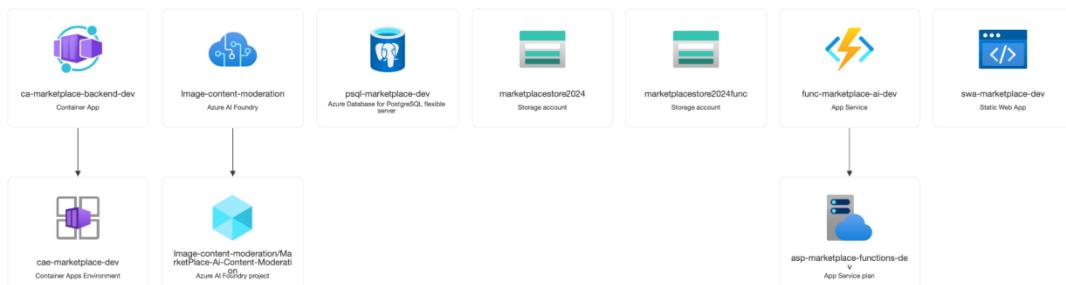


### 2. Azure Container Apps



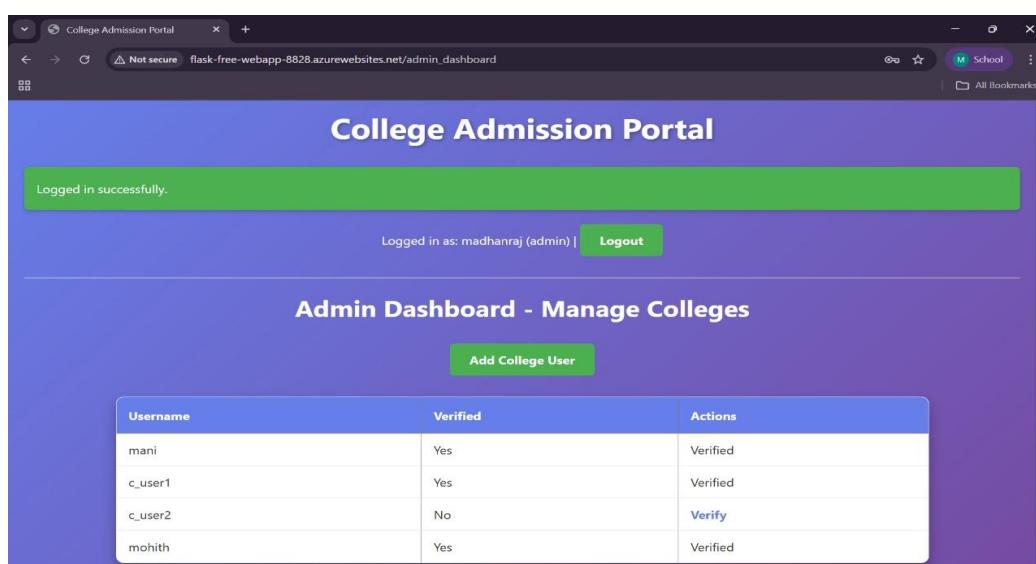
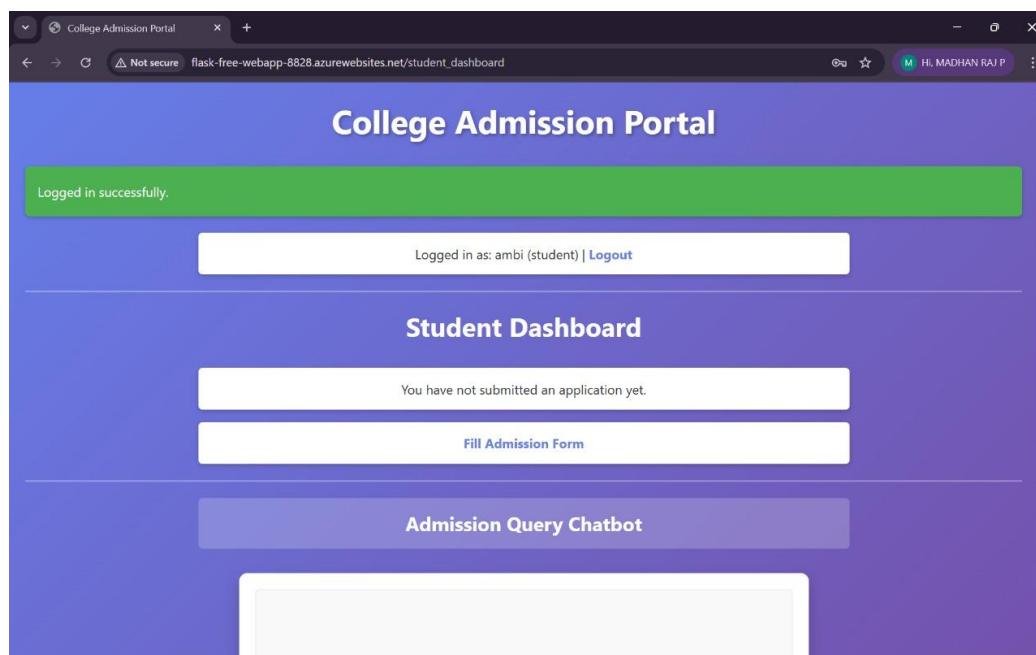
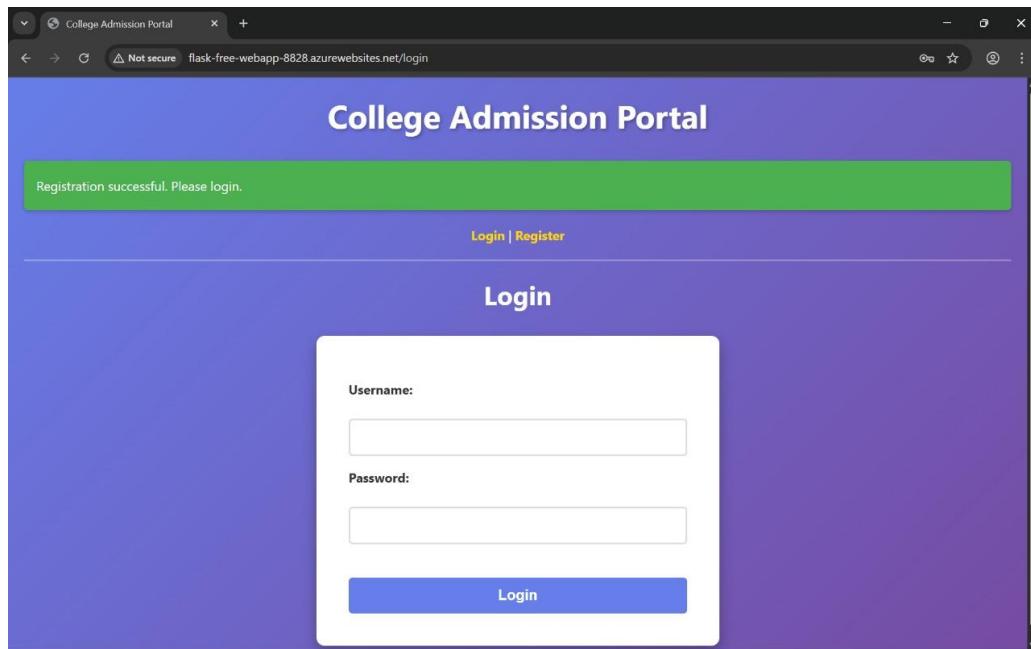
*Figure18.Description Generation*

## Deployment resource visualizer



*Figure19.NetworkVisualizer*

## OUTPUT SCREENSHOTS



College Admission Portal

Not secure flask-free-webapp-8828.azurewebsites.net/add\_college

Logged in as: madhanraj (admin) | Logout

### Add College User

Username:

Password:

**Add College**

College Admission Portal

Logged in successfully.

Logged in as: lokesh (college) | Logout

### College Dashboard

Username	Full Name	Email	Phone	Status	10th Marksheets	12th Marksheets	Actions
madhan148	Madhan	220701148@rajalakshmi.edu.in	09655221183	Approved	<a href="#">View 10th</a>	<a href="#">View 12th</a>	N/A
user_1	mani	mani@gmail.com	09655221183	Approved	<a href="#">View 10th</a>	<a href="#">View 12th</a>	N/A
sana	sana	sana@gmail.com	09655221183	Approved	<a href="#">View 10th</a>	<a href="#">View 12th</a>	N/A