RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM – 602 105

CP23211 ADVANCED SOFTWARE ENGINEERING LABORATORY

LABORATORY RECORD

Name : _____SUCHINDHAR R M _____

Year / Branch : _____I – M.E. CSE_____

University Register No. :___2116230711011_____

College Roll no : _____230711011_____

Semester : _____ II_____

Academic Year : _____2023 – 2024_____

RAJALAKSHMI ENGINEERING COLLEGE

RAJALAKSHMI NAGAR, THANDALAM – 602 105

BONAFIDE CERTIFICATE

Name : _____SUCHINDHAR R M _____

Academic Year : 2023 - 2024 Semester :_II_ Branch :___CSE_

Register No : 2116230711011

Certified that this is the bonafide record of work done by the above student in the CP 23211- Advanced Software Engineering Laboratory during in the year 2023 – 24

Signature of the Faculty In-charge

Submitted for the Practical Examination held on __22/06/2024____

Internal Examiner External Examiner

INDEX

SOFTWARE REQUIREMENT SPECIFICATIONS (SRS)

EXP NO : 1 **DATE : 05.03.2024**

CONTENTS

Integrating Jenkins for Efficient Deployment and Orchestration across Multi-Cloud Environments

OVERVIEW OF THE PROJECT:

The project addresses the challenges inherent in managing multi-cloud environments by leveraging automation and Infrastructure as Code (IaC) principles. It integrates tools like Terraform, Kubernetes, and Jenkins to streamline cloud resource management across AWS, Azure, and GCP. This approach not only reduces operational costs through optimized resource allocation and automated scaling but also enhances efficiency by automating CI/CD pipelines. By adopting a microservices architecture and adhering to design principles like modularity and scalability, the system ensures robustness and flexibility in deploying and managing applications seamlessly across diverse cloud platforms. Overall, the project aims to deliver significant benefits including cost savings, operational efficiency improvements, and enhanced scalability for users in DevOps, cloud architecture, and software development roles.

In summary, the project tackles the complexities of multi-cloud management through automation and IaC, offering a comprehensive solution for optimizing cloud resources and enhancing deployment processes across AWS, Azure, and GCP. By integrating Terraform, Kubernetes, and Jenkins, it aims to deliver cost efficiencies, operational streamlining, and scalability benefits, underpinned by a robust microservices architecture. This approach caters to the needs of DevOps engineers, cloud architects, and software developers, ensuring seamless application deployment and management while addressing current challenges in manual and siloed cloud resource management practices.

Integrating Jenkins for Efficient Deployment and Orchestration across Multi-Cloud Environments

1.Introduction

1.1.Purpose:

The purpose of this project is to address the complexity and inefficiency associated with managing multi-cloud environments. By leveraging automation and Infrastructure as Code (IaC) practices, the project aims to optimize cloud resource management across AWS, Azure, and GCP. Key tools such as Terraform, Kubernetes, and Jenkins are integrated to automate provisioning, scaling, and deployment processes, thereby reducing operational costs and improving overall efficiency.

2.Overall Description

2.1Product Perspective:

The project operates within the realm of cloud infrastructure management, focusing on automating the deployment and scaling of applications across multiple cloud providers. It interfaces directly with cloud platforms (AWS, Azure, GCP) through APIs and utilizes infrastructure orchestration tools like Terraform and Kubernetes to achieve seamless integration and automation. The system ensures compatibility with existing CI/CD pipelines by incorporating Jenkins for automated build, test, and deployment workflows.

Features

2.2.1.Multi-Cloud Management:

- Provision and manage resources across AWS, Azure, and GCP using Terraform.

2.2.2.Container Orchestration:

- Deploy and manage containerized applications at scale with Kubernetes.

2.2.3 Automated CI/CD:

- Streamline software delivery pipelines with Jenkins integration for continuous integration and deployment.

2.2.4.Monitoring and Optimization:

- Monitor resource utilization and optimize costs across multiple cloud environments.

2.2.5.Security and Compliance:

- Integrate security scans and ensure compliance with industry standards throughout the deployment lifecycle.

3.Specific Requirements

3.1Functional Requirements:

3.1.1.Automated Infrastructure Deployment:

- Provision cloud resources automatically using Terraform scripts.

3.1.2.Containerized Application Deployment:

- Deploy applications to Kubernetes clusters for scalability and resilience.

3.1.3.Continuous Integration and Deployment:

- Automate build, test, and deployment workflows using Jenkins pipelines.

3.1.4. Resource Monitoring and Optimization:

- Monitor and optimize resource usage across AWS, Azure, and GCP.

3.1.5.Security Integration:

- Integrate security scans into CI/CD pipelines to ensure compliance and mitigate risks.

3.1.6. Logging and Auditability:

- Log deployment activities and provide audit trails for troubleshooting and compliance purposes.

3.2. Non-Functional Requirements:

3.2.1.Scalability:

- Support up to 1000 concurrent deployments without performance degradation.

- Implement load balancing mechanisms to evenly distribute traffic across multiple instances and ensure optimal performance during peak usage
- Utilize auto-scaling capabilities to dynamically adjust resources based on workload demands

3.2.2. Security:

- Ensure all data transmissions are encrypted using TLS 1.2 or higher.
- Implement mechanisms to verify the integrity of transmitted data
- Enforce strict access controls and authentication mechanisms to restrict access to sensitive data

3.3.3.Reliability:

- Maintain a system uptime of 99.9% for continuous availability.
- Implement fault-tolerant strategies to mitigate the impact of failures.
- Continuously monitor system performance and availability metrics to identify issues.

4 External Interface Requirements

4.1.Cloud Providers:

- Integration with APIs of AWS, Azure, and GCP enables automated provisioning and management of cloud resources
- Utilizes cloud provider APIs to dynamically allocate and manage compute, storage, and networking resources across multiple cloud platforms

4.2.Version Control Systems:

- Integration with Git repositories to trigger automated CI/CD pipelines.
- Facilitates continuous integration and deployment (CI/CD)automatically.

4.3.Monitoring Tools :

- Tools for monitoring resource utilization and application performance.
- Monitors and analyzes application performance metrics.

4.4Security Tools:

- Integration with security tools to ensure secure deployments and compliance.
- Ensures adherence to security standards and regulatory requirements through automated security scans

5. Conclusion

This project aims to revolutionize cloud resource management by automating and optimizing processes across multiple cloud environments. By leveraging modern DevOps practices and technologies such as Terraform, Kubernetes, and Jenkins, it promises significant improvements in efficiency, scalability, and cost-effectiveness for organizations embracing multi-cloud strategies. Through robust automation, seamless integration, and adherence to stringent security and reliability standards, the project seeks to empower DevOps engineers, cloud architects, and software developers to deploy and manage applications with confidence and agility in complex multi-cloud environments.

SCRUM METHODOLOGY

EXP NO 2 **DATE:**

SCRUM METHODOLOGY

1.Introduction

Scrum is an agile framework that facilitates collaborative effort in complex projects. It emphasizes iterative development, transparency, and adaptability to deliver high-value products efficiently.

2.Objectives

The objectives of Scrum include enhancing team productivity, improving product quality through frequent feedback loops, and enabling rapid adaptation to changing requirements.

3.Product Backlog Introduction

The product backlog serves as a prioritized list of all desired features, enhancements, and fixes for a product. It evolves throughout the project based on stakeholder input, market changes, and team insights.

4.Product Backlog

4.1For CI/CD System

4.1.2 Automated Build:

- Automated build processes triggered by code commits to ensure efficient compilation and testing of code changes.

4.1.3. Deployment Automation:

- Automate deployment pipelines to streamline the release process across different environments.

4.1.3.Monitoring and Alerting:

- Monitoring and alerting set up for CI/CD pipelines to quickly identify and respond to build or deployment failure**s.**

5.1.User Stories

5.1.1.As a Developer:.

- Automated build processes triggered by code commits
- verify and validate code changes**.**

5.1.2.As a DevOps Engineer:

- Automated deployment pipelines to streamline the deployment of applications across different environments.

5.1.3.As a Performance Engineer

- Automate performance tests included in CI/CD pipelines to validate system scalability and response times under load

6.Sprint

A sprint is a time-boxed iteration in Scrum, usually lasting 1-4 weeks, where a cross-functional team works collaboratively to complete a set of prioritized items from the product backlog.

7.Sprint Backlog

The sprint backlog is a subset of the product backlog items selected for implementation during a sprint. It details the tasks, responsibilities, and commitments of the team members for the sprint duration.

8.Sprint Review

The sprint review is a meeting held at the end of each sprint where the team demonstrates the completed work to stakeholders. Feedback is gathered, and

adjustments to the product backlog are made based on insights gained.

9.Software Used

Common software tools used in Scrum include:

- Jira:For managing the product backlog, sprints, and tasks.
- Trello: Visual project management tool for organizing tasks on boards.
- Azure DevOps: Integrated suite for managing software development projects using Scrum practices.
- VersionOne:Agile project management software that supports Scrum methodologies.

10.Conclusion

Scrum methodology provides a structured approach to project management, emphasizing collaboration, iterative development, and responsiveness to change. By focusing on delivering incremental value and continuous improvement, Scrum enables teams to adapt to evolving requirements and deliver high-quality products efficiently.

USER STORIES

EXP.NO: 3 **DATE: 12.3.2024**

Automated Deployment Trigger:

User story

- As a DevOps engineer, changes pushed to the main branch of the repository should automatically trigger the deployment pipeline, ensuring new features are quickly deployed to production.

Integration Testing:

User story

- As a QA engineer, integration tests must cover critical user workflows across different modules of the application to ensure seamless functionality across all components.

Configuration Management Flexibility:

User story

- As a system administrator, configuration settings for deployment environments (e.g., database connections, API endpoints) should be easily configurable through environment-specific variables, ensuring flexibility and security.

Real-time Monitoring Alerts:

User story

- As an operations team member, real-time alerts via Slack or email should be received when the CI/CD pipeline fails or when there's a significant drop in performance, ensuring prompt investigation and resolution.

Incremental Rollout Capabilities:

User story

- As a product manager, the CI/CD pipeline should support incremental

rollout of new features to a subset of users, allowing feedback gathering and risk mitigation before full deployment.

Documentation Generation:

User story

- As a technical writer, updated documentation (e.g., API reference guides, user manuals) should be automatically generated whenever a new feature is deployed, ensuring up-to-date documentation.

Security Scanning Integration:

User story

- As a security analyst, automated security scans (e.g., static code analysis, vulnerability assessments) should be integrated into the CI/CD pipeline to ensure application code security before deployment.

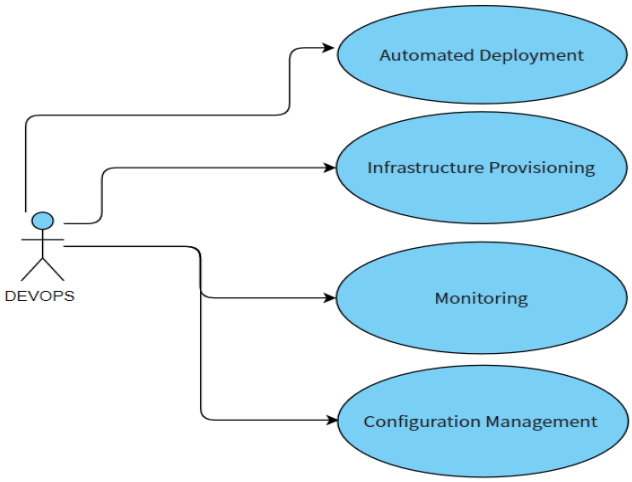
Performance Benchmarking:

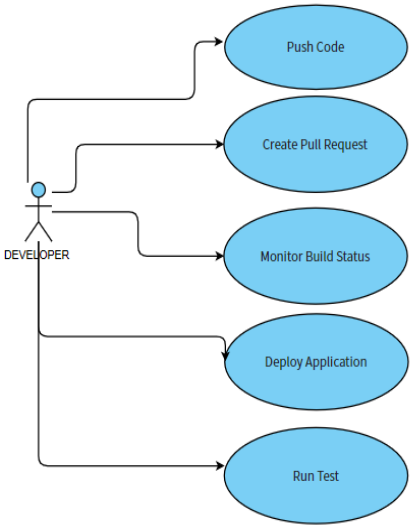
User story

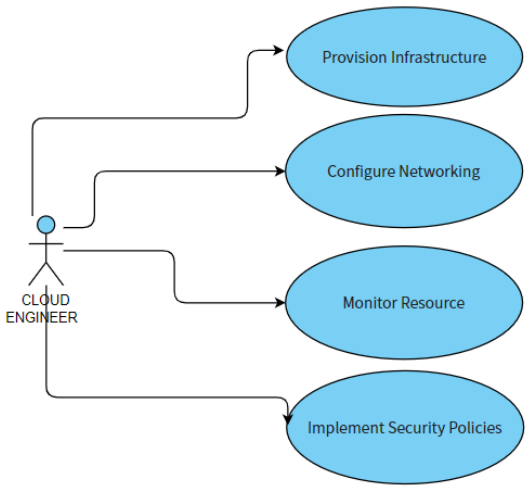
- As a performance engineer, the CI/CD pipeline should include automated performance benchmarks (e.g., load testing, response time analysis) for each deployment, ensuring the application meets performance requirements under varying conditions.

USE CASE DIAGRAM

EXP.NO: 4 **DATE: 19.3.2024**






Provision Infrastructure
Configure Networking
Monitor Resource
Implement Security Policies
CLOUD
ENGINEER

NON FUNCTIONAL REQUIREMENT

EXP.NO: 5 **DATE: 29.3.2024**

1. **Performance:**
   - The system should handle concurrent builds and deployments efficiently.
   - Response times for build and deployment actions should not exceed predefined thresholds (e.g., 5 seconds for triggering builds).

2. Scalability:
   - The system should scale horizontally to accommodate increased load during peak usage times.
   - Ability to handle a 50% increase in concurrent build and deployment requests without performance degradation.

3. Reliability:
   - The system should ensure high availability and minimize downtime during deployments.
   - System uptime of at least 95% over a period of one year, excluding scheduled maintenance windows.

4. Security:
   - All data transmissions and sensitive information should be encrypted using industry-standard protocols (e.g., TLS 1.2 or higher).
   - Regular security audits and compliance checks to ensure adherence to security policies and standards.

5. Maintainability:
   - The system should be easy to maintain and upgrade without disrupting ongoing deployments.
   - Average time required for applying updates or patches should not exceed 30 minutes per deployment.

6. Usability:

- The system interface should be intuitive and user-friendly for both developers and DevOps engineers.
- Average time for a new user to perform basic tasks (e.g., triggering a build, deploying an application) should not exceed 10 minutes.

7. Integration:

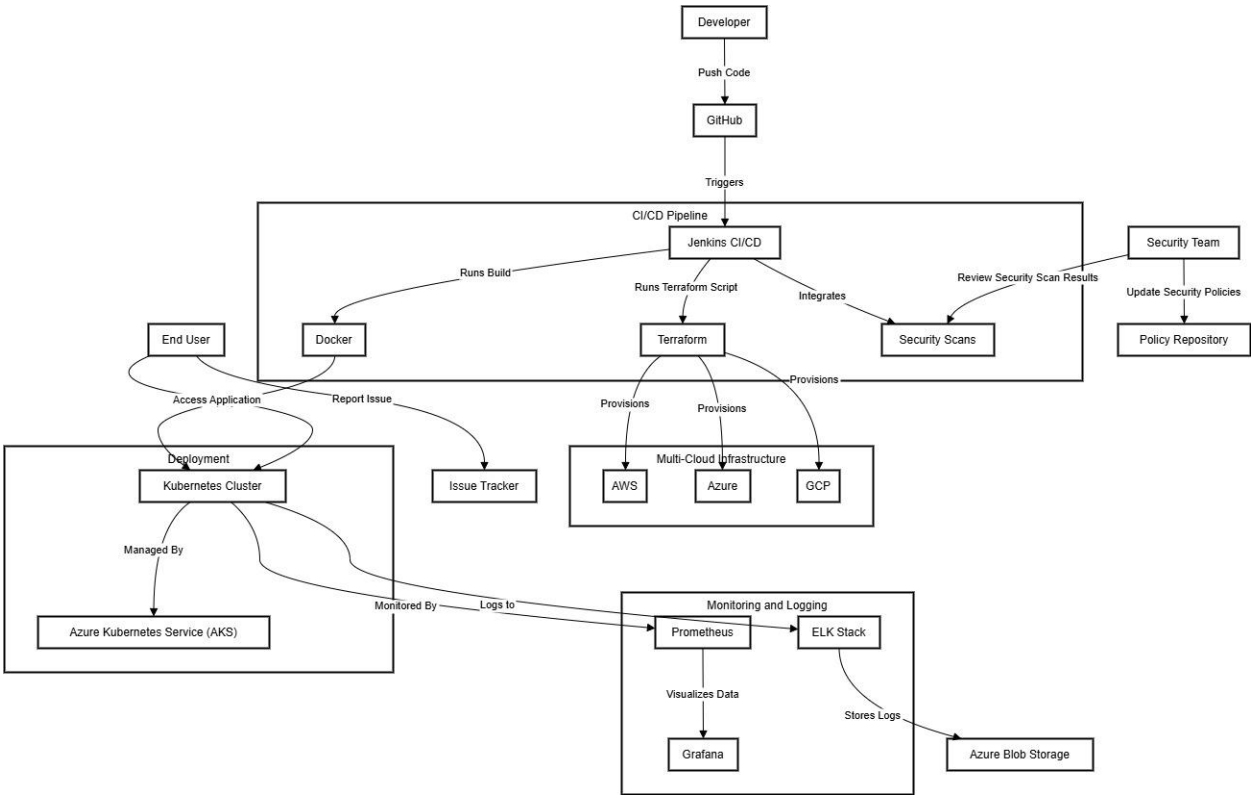
- The system should integrate seamlessly with version control systems (e.g., Git), issue tracking tools (e.g., Jira), and monitoring platforms (e.g., Prometheus).
- Successful integration and synchronization of data across platforms without data loss or inconsistencies.

8. Performance Monitoring:

- The system should have monitoring capabilities to track build times, deployment success rates, and resource utilization.
- Real-time dashboards displaying build metrics and deployment status, with alerts for abnormal behavior or failures.

OVERALL PROJECT ARCHITECTURE

EXP.NO: 6 **DATE: 09.4.2024**



The business architecture of a CI/CD and deployment system involves several key components that work together to ensure smooth development, testing, deployment, monitoring, and management of applications. Below is an explanation of each component and its role:

1. Developer:

Writes code and pushes changes to a version control system (e.g., GitHub). Initiates builds and monitors build status.

2. DevOps Engineer:

Configures and manages the CI/CD pipeline, provisions infrastructure, monitors infrastructure, handles deployments, and integrates security scans.

3. End User:

Accesses the deployed application and reports any issues.

4. Security Team:

Reviews security scan results and updates security policies.

Components and interaction

Developer:

- Pushes code to GitHub, which triggers the CI/CD pipeline.

GitHub:

- Hosts the code repository. Any push to the repository triggers Jenkins to start the CI/CD pipeline.

Jenkins CI/CD:

- Manages the continuous integration and continuous deployment pipeline. It builds the application using Docker and runs infrastructure provisioning scripts using Terraform.

Docker:

- Builds application images that are then deployed to a Kubernetes cluster.

Terraform:

- Automates the provisioning of infrastructure on cloud platforms such as AWS, Azure, and GCP.

AWS, Azure, GCP:

- Cloud platforms where the infrastructure is provisioned and managed.

Kubernetes Cluster:

- Orchestrates the deployment of Docker containers, ensuring the application is properly managed and scaled.

Azure Kubernetes Service (AKS):

- Manages the Kubernetes cluster on Azure.

Prometheus:

- Monitors the resource utilization of the Kubernetes cluster.

ELK Stack:

- Collects and stores logs for analysis.

Grafana:

- Visualizes monitoring data from Prometheus.

Azure Blob Storage:

- Stores logs and other relevant data.

Security Scans:

- Integrated into the CI/CD pipeline to ensure code security before deployment.

End User:

- Accesses the application and interacts with it. Reports issues through an issue tracker.

Issue Tracker:

- Collects and manages user-reported issues.

Security Team:

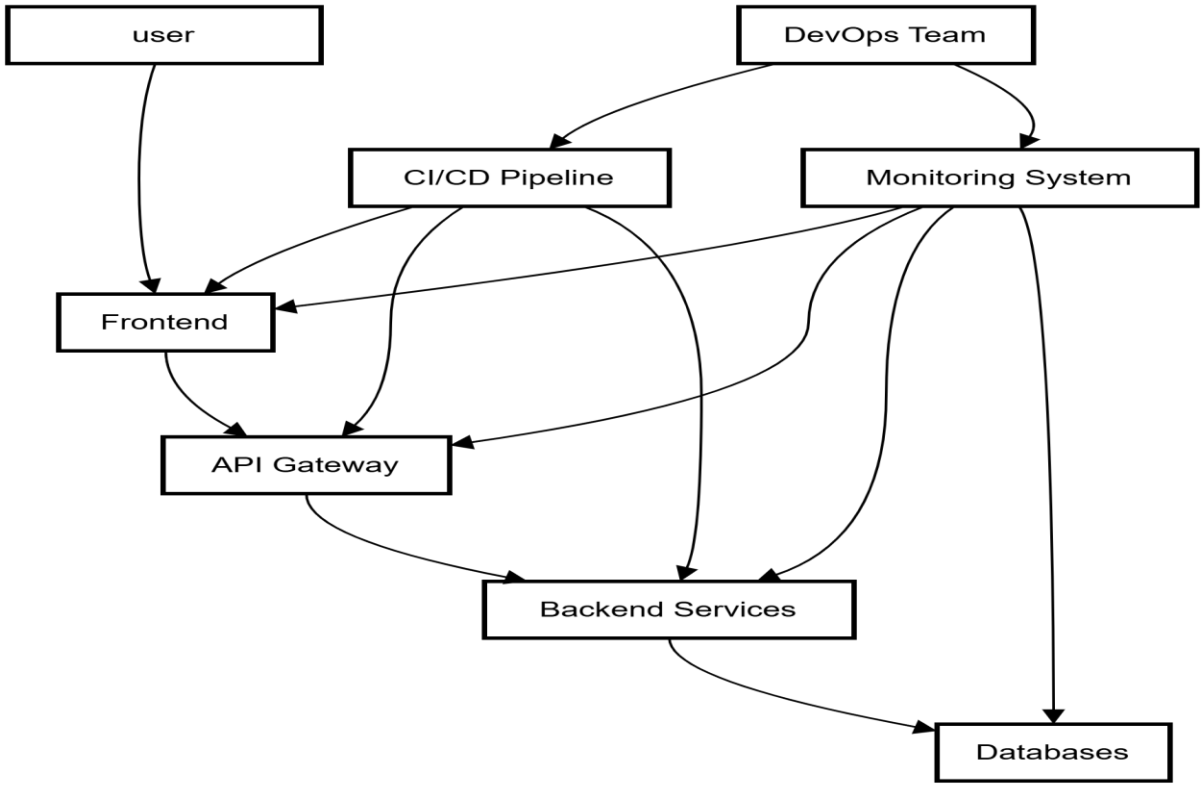
- Reviews security scan results and updates security policies.

Policy Repository:

- Stores security policies that need to be updated based on scan results.

BUSINESS ARCHITECTURE DIAGRAM

EXP.NO: 7 **DATE: 19.4.2024**



Module Explanation

Users:

- Interact with the system through the frontend and consume data

Frontend:

- User interface for the application
- Provides access to various features and data visualizations

API Gateway:

- Single entry point for all client requests

- Routes requests to appropriate backend services

Backend Services:

- Core business logic and data processing
- Includes services like user management, project management, and data analysis

Databases:

- Store and manage application data
- May include different types of databases for various services

DevOps Team:

- Responsible for development, deployment, and maintenance of the system
- Manages the CI/CD pipeline and monitoring system

CI/CD Pipeline:

- Automates the build, test, and deployment processes
- Ensures consistent and reliable software delivery

Monitoring System:

- Tracks system performance, user activities, and business metrics
- Provides insights to both the DevOps team and business users

CLASS DIAGRAM

EXP.NO: 8 **DATE: 30.4.2024**

User:

- Represents an individual who interacts with the system, such as a developer or a DevOps engineer. Users can log in, log out, and manage their roles within the system.

Project:

- Represents a project managed within the system. Each project has a unique ID, a name, and an owner. Projects can be created, deleted, and assigned to users.

Pipeline:

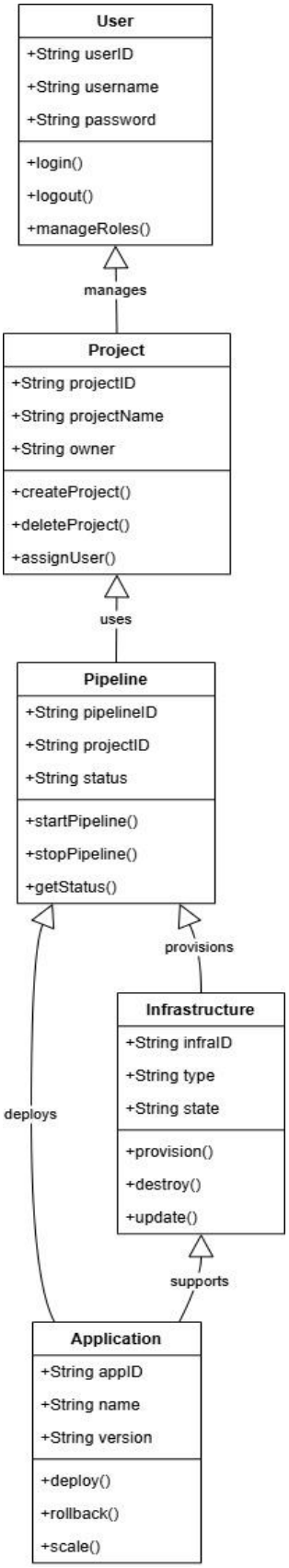
- Represents a CI/CD pipeline associated with a project. Pipelines manage the flow of tasks from code integration to deployment and can be started, stopped, and queried for their status.

Infrastructure:

- Represents the infrastructure resources provisioned for a project. It includes information about the type and state of the infrastructure, and methods to provision, destroy, and update the infrastructure.

Application:

- Represents the application that is built, deployed, and managed within the system. It includes details about the application ID, name, and version, and methods to deploy, rollback, and scale the application.


User
+String userID
+String username
+String password
+login()
+logout()
+manageRoles()
manages
Project
+String projectID
+String projectName
+String owner
+createProject()
+deleteProject()
+assignUser()
uses
Pipeline
+String pipelineID
+String projectID
+String status
+startPipeline()
+stopPipeline()
+getStatus()
provisions
Infrastructure
+String infraID
+String type
+String state
+provision()
+destroy()
+update()
deploys
supports
Application
+String appID
+String name
+String version
+deploy()
+rollback()
+scale()

SEQUENCE DIAGRAM

EXP.NO: 9 **DATE: 10.5.2024**

Automating Infrastructure Deployment:

- **DevOps Engineer triggers deployment**: The DevOps engineer initiates the deployment process by triggering a job in Jenkins.
- **Jenkins runs Terraform script**: Jenkins executes a Terraform script to provision the necessary infrastructure.
- **Terraform provisions infrastructure on AWS**: Terraform communicates with AWS to provision the required resources.
- **AWS confirms provisioning**: AWS confirms that the resources have been successfully provisioned.
- **Terraform reports status to Jenkins**: Terraform sends the status back to Jenkins.
- **Jenkins confirms deployment to DevOps Engineer**: Jenkins notifies the DevOps engineer that the infrastructure has been provisioned.

Deploying Application to Kubernetes:

- **Developer pushes code to GitHub**: The developer pushes code changes to the GitHub repository.
- **GitHub triggers Jenkins build**: GitHub sends a webhook to Jenkins to trigger a new build.
- **Jenkins builds Docker image**: Jenkins builds a Docker image of the application.
- **Docker confirms build to Jenkins**: Docker confirms that the image has been built.
- **Jenkins deploys application to Kubernetes**: Jenkins deploys the Docker image to the Kubernetes cluster.
- **Kubernetes manages deployment on AKS**: The Kubernetes cluster, managed by Azure Kubernetes Service (AKS), handles the deployment.
- **AKS confirms deployment to Jenkins**: AKS confirms that the deployment is successful.

- **Jenkins notifies the Developer**: Jenkins notifies the developer that the application has been deployed.

Monitoring Resource Utilization:

- **DevOps Engineer triggers monitoring script in Jenkins**: The DevOps engineer initiates a monitoring script in Jenkins.
- **Jenkins collects data using Prometheus**: Jenkins collects monitoring data from the Kubernetes cluster using Prometheus.
- **Prometheus visualizes data in Grafana**: Prometheus sends the data to Grafana for visualization.
- **Grafana displays the dashboard to DevOps Engineer**: Grafana displays the monitoring dashboard to the DevOps engineer.

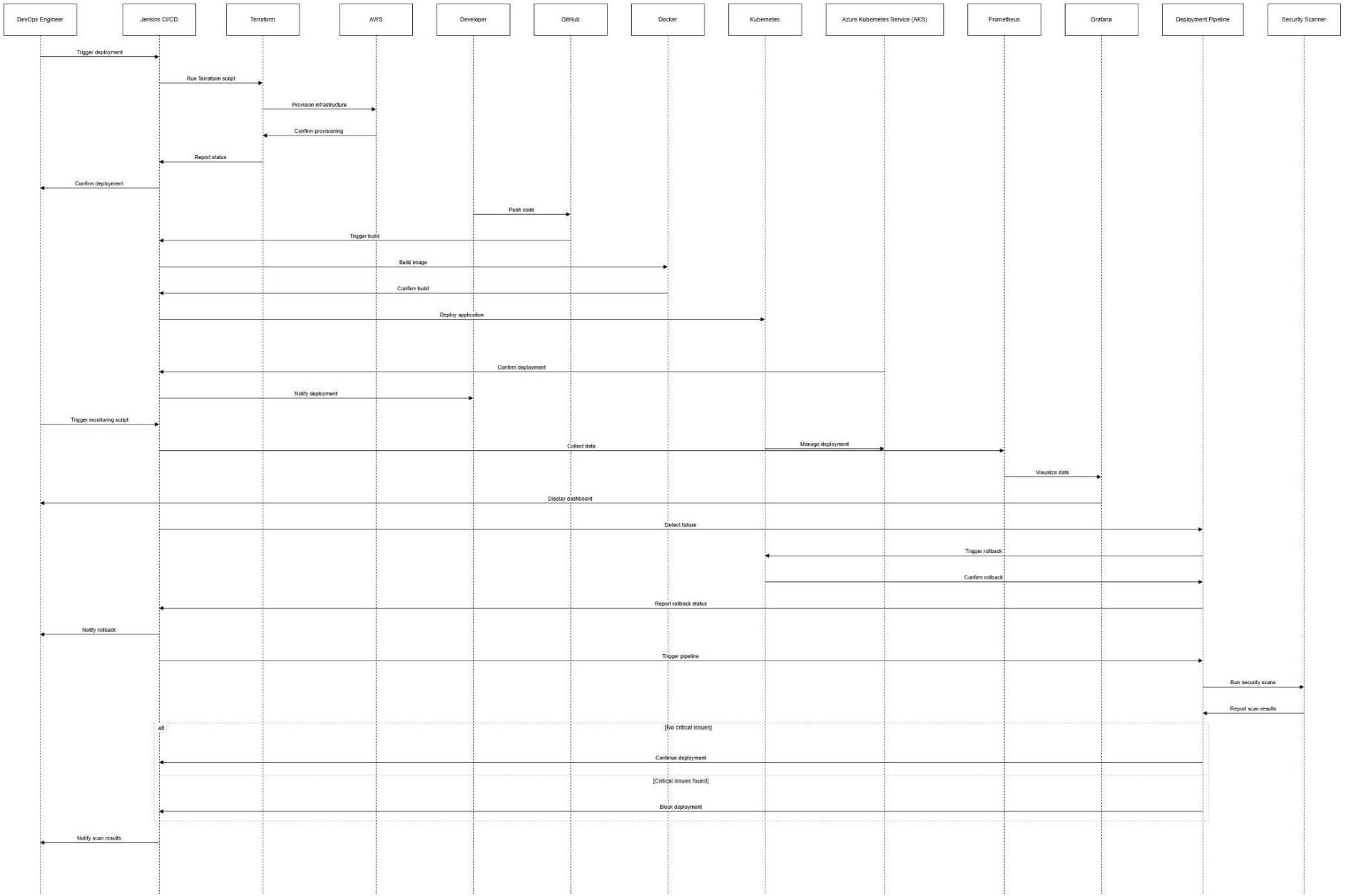
Rolling Back Deployment:

- **Jenkins detects a failure in the deployment pipeline**: Jenkins detects a failure in the deployment process.
- **Pipeline triggers rollback in Kubernetes**: The deployment pipeline triggers a rollback in the Kubernetes cluster.
- **Kubernetes confirms rollback to Pipeline**: Kubernetes confirms that the rollback is successful.
- **Pipeline reports rollback status to Jenkins**: The deployment pipeline reports the rollback status to Jenkins.
- **Jenkins notifies the DevOps Engineer**: Jenkins notifies the DevOps engineer that the rollback has been completed.

Integrating Security Scans:

- **Jenkins triggers pipeline**: Jenkins initiates the CI/CD pipeline.
- **Pipeline runs security scans**: The pipeline runs security scans using an integrated security scanning tool.
- **Security Scanner reports scan results to Pipeline**: The security scanner reports the results of the security scans.
- **Pipeline continues or blocks deployment based on scan results**: If no critical issues are found, the pipeline continues the deployment. If critical issues are found, the pipeline blocks the deployment.

- **Jenkins notifies the DevOps Engineer of the scan results**: Jenkins notifies the DevOps engineer of the results of the security scans.

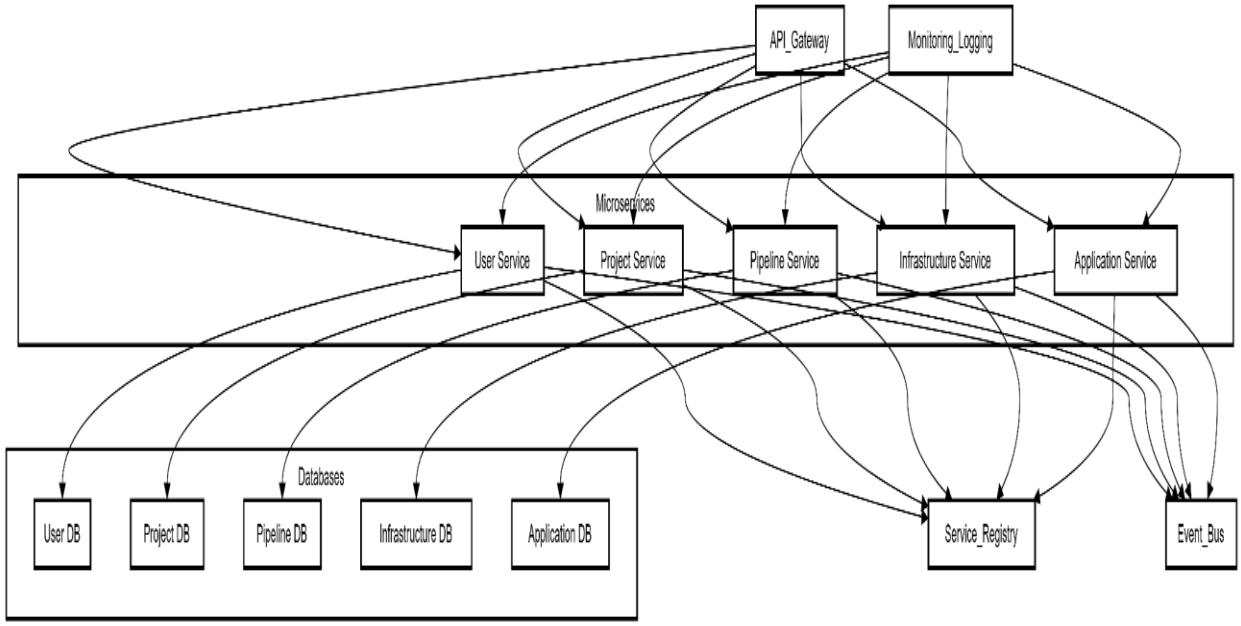

DevOps Engineer
Jenkins CI/CD
Terraform
AWS
Developer
GitHub
Docker
Kubernetes
Azure Kubernetes Service (AKS)
Prometheus
Grafana
Deployment Pipeline
Security Scanner
Trigger deployment
Run Terraform script
Provision infrastructure
Confirm provisioning
Report status
Confirm deployment
Push code
Trigger build
Build image
Confirm build
Deploy application
Confirm deployment
Notify deployment
Trigger monitoring script
Collect data
Manage deployment
Visualize data
Display dashboard
Detect failure
Trigger rollback
Confirm rollback
Report rollback status
Notify rollback
Trigger pipeline
Run security scans
Report scan results
alt
[No critical issues]
Continue deployment
[Critical issues found]
Block deployment
Notify scan results

ARCHITECTURAL PATTERN

EXP.NO: 10 **DATE: 17.5.2024**

Microservices Architecture Pattern

The architectural pattern used in this CI/CD and deployment system is **Microservices Architecture**. This pattern involves designing an application as a collection of small, loosely coupled services that can be developed, deployed, and scaled independently.



Key Components of the Microservices Architecture

1. **API Gateway**: Acts as the entry point for all client requests, routing them to the appropriate microservice.
2. **Microservices**: Each service is responsible for a specific business functionality. For instance:
    - **User Service**: Manages user authentication and roles.
    - **Project Service**: Handles project creation, deletion, and assignment.
    - **Pipeline Service**: Manages CI/CD pipelines.

    - **Infrastructure Service**: Provisions and manages cloud infrastructure.
    - **Application Service**: Deploys and manages application instances.
3. **Service Registry**: Keeps track of all microservices and their instances, facilitating service discovery.
4. **Database per Service**: Each microservice has its own database to ensure loose coupling.
5. **Event Bus**: Facilitates communication between microservices using events.
6. **Monitoring and Logging**: Collects and visualizes logs and metrics from various microservices.

Benefits of Microservices Architecture

1. **Scalability**: Each service can be scaled independently based on demand.
2. **Fault Isolation**: Failures in one service do not affect other services.
3. **Technology Diversity**: Different services can use different technologies best suited for their functionality.
4. **Independent Deployment**: Services can be deployed independently, enabling faster releases and updates.