DevOps Project Report 2025

By Hardik Ajmeriya

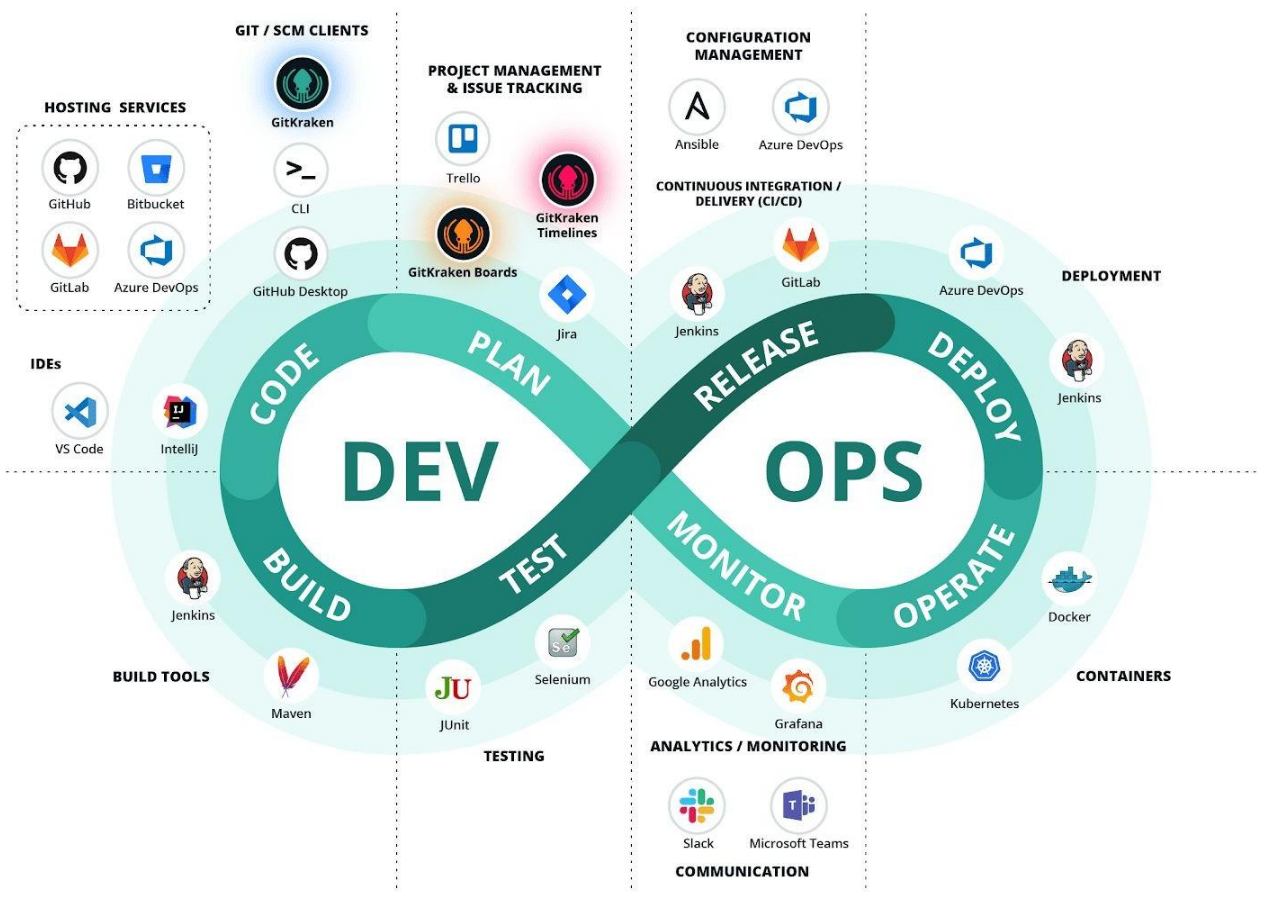

Table of Contents

EXECUTIVE SUMMARY

This project demonstrates the successful implementation of a complete end-to-end DevOps pipeline, integrating modern tools and cloud-native technologies to achieve automated, scalable, and reliable application delivery. The primary objective was to design and deploy a microservices-based application using a fully automated CI/CD and GitOps pipeline on AWS cloud infrastructure.

The project follows the DevOps lifecycle — from code management, build automation, testing, and containerization, to deployment, monitoring, and continuous delivery. Each stage has been configured with an emphasis on automation, security, and efficiency to replicate real-world DevOps workflows used in production environments.

At the infrastructure level, Terraform was used for Infrastructure as Code (IaC) to provision and manage AWS resources such as VPCs, EC2 instances, IAM roles, S3 buckets, and EKS clusters. This ensures consistency, repeatability, and simplified management of infrastructure across environments.

The application source code was managed through Git and GitHub, with Jenkins configured as the automation server for continuous integration and build management. Docker was used to containerize multiple microservices (developed in Go, Java, and Python), ensuring isolated, reproducible, and portable deployments. All container images were versioned and stored in Docker Hub.

For deployment, the project utilized AWS Elastic Kubernetes Service (EKS) as the orchestration layer, handling multi-service scaling, load balancing, and resilience. The deployment process was further enhanced by integrating Argo CD — a GitOps tool that automates continuous delivery directly from Git repositories, ensuring that the cluster state always remains synchronized with the desired configuration stored in Git.

Additionally, AWS Load Balancer Controller and IAM OIDC provider were configured for secure access management and efficient routing of traffic through a custom production domain — **devopsbyhardik.shop.**

(Note: The domain is currently inactive due to the suspension of AWS resources, as maintaining live infrastructure involves recurring cloud costs that are difficult to sustain for a student project.)

Monitoring and observability were achieved through Argo CD dashboards and AWS CloudWatch, enabling real-time visibility into deployment health, logs, and application performance.

By the end of this project, a fully functional, automated DevOps ecosystem was established — capable of building, testing, deploying, and monitoring microservices applications with minimal manual intervention. This end-to-end solution reflects the modern cloud-native DevOps architecture used by leading enterprises and demonstrates strong proficiency in CI/CD, Kubernetes, AWS cloud, and GitOps methodologies.

PROJECT OVERVIEW

2.1 Objectives

The primary objective of this project was to **design and implement a fully automated DevOps pipeline** that integrates multiple tools and technologies to manage the entire software development lifecycle — from source code management to automated deployment on the cloud.

This project aims to simulate a **real-world production DevOps environment** where code flows seamlessly through the stages of **build, test, deploy, and monitor,** with minimal manual intervention. The end goal was to achieve **high automation, scalability, consistency, and security** in deployment workflows.

Specific objectives include:

1. **Infrastructure-Automation:**
   Implement Infrastructure as Code (IaC) using Terraform to provision AWS resources such as VPC, EC2 instances, IAM roles, S3 buckets, and EKS clusters automatically.

2. **Continuous Integration:**
   To ensure a seamless and automated build and testing process, GitHub Actions was utilized as the core CI tool. This replaced traditional Jenkins-based pipelines with a fully cloud-native, event-driven automation workflow.

3. **Containerization:**
   Containerize multiple microservices (Go, Java, Python) using Docker, ensuring each application runs in an isolated and reproducible environment.

4. **Image Management:**
   Store and manage container images securely on Docker Hub with proper tagging and version control.

5. **Continuous Delivery:**
   Deploy and manage services on AWS Elastic Kubernetes Service (EKS), achieving automated, declarative deployments through Argo CD using a GitOps approach.

6. **Security and Access Management**
   Utilize AWS IAM roles, OIDC provider, and Load Balancer Controller to manage secure access, authentication, and network routing.

7. **Monitoring and Observability:**
   Integrate AWS CloudWatch and Argo CD dashboards to monitor application performance, health, and deployment synchronization in real time.

8. **Cost Optimization and Resource Management:**
   Use Terraform's destroy functionality after each phase to terminate unused resources and minimize AWS billing.

The project thus combines DevOps, Cloud Computing, and Automation Engineering principles to build a complete production-like deployment pipeline.

2.2 Scope

The scope of this project covers the entire DevOps lifecycle, demonstrating how an application moves from code to deployment in an automated, scalable, and monitored environment. It focuses on implementing each DevOps phase using the most widely adopted tools in the industry.

The project was executed entirely on AWS Cloud, using Free Tier resources and educational credits where possible. The infrastructure was provisioned dynamically using Terraform and destroyed upon completion to ensure cost efficiency — an essential consideration for students working on cloud-based projects.

In-Scope Activities:

- Designing and provisioning cloud infrastructure using Terraform.
- Managing code and version control through Git and GitHub.
- Setting up a Jenkins server for CI pipeline automation.
- Containerizing applications with Docker and pushing images to Docker Hub.
- Configuring and deploying workloads on AWS EKS clusters.
- Implementing continuous delivery through Argo CD and GitOps workflows.
- Configuring AWS Load Balancer Controller for ingress routing and HTTPS traffic management.
- Enabling custom domain routing using Route 53 and domain: devopsbyhardik.shop.
- Implementing monitoring and alerting using CloudWatch and Argo CD dashboards.

Out-of-Scope Activities:

- Maintaining long-term live hosting (the domain and EKS cluster are currently inactive due to cloud cost constraints).
- Advanced production-level scaling features such as auto-scaling groups or multi-region failover.

This project is intended for educational and demonstrative purposes, showcasing hands-on expertise in modern DevOps tools and AWS ecosystem management.

2.3 Architecture Overview

The project architecture is designed as a modular, microservices-based cloud-native system that integrates container orchestration, automation, and monitoring across the DevOps pipeline.

Each layer of the architecture interacts seamlessly, ensuring scalability, maintainability, and traceability across the software delivery process.

1. Infrastructure Layer

- Managed through Terraform using Infrastructure as Code (IaC) principles.
- Provisions all essential AWS components:
  - **VPC and Subnets:** For secure, isolated networking.
  - **EC2 Instance:** Hosting Jenkins and initial configuration.
  - **IAM Roles & Policies:** Providing least-privilege access.
  - **EKS Cluster:** Hosting containerized workloads.
  - **S3 Bucket & DynamoDB Table:** Supporting artifact storage and state management.

2. Source Code and Build Layer

- **Git & GitHub:** Used for source code management and version control.
- **Jenkins Pipeline:** Automatically builds, tests, and packages applications when new code is pushed.
- **Docker:** Converts the application into portable container images.
- **Docker Hub:** Acts as a central image registry for versioned containers.

3. Deployment and Delivery Layer

- **AWS EKS (Kubernetes):** Orchestrates multiple containerized applications.
- **Argo CD:** Enables GitOps-based deployment, automatically syncing the cluster state with the Git repository.
- **AWS Load Balancer Controller:** Routes traffic efficiently to running pods and services.
- **Custom Domain (devopsbyhardik.shop):** Provides a public endpoint for production-like access. *(Currently inactive to avoid recurring cloud charges.)*

4. Monitoring and Observability Layer

- **AWS CloudWatch:** Tracks logs, metrics, and infrastructure health.
- **Argo CD Dashboard:** Monitors deployment status, sync state, and cluster activity.
- Together, they provide complete visibility into both infrastructure and application performance.

5. Security Layer

- **IAM OIDC Integration:** Facilitates secure authentication between AWS services and Kubernetes components.
- **IAM Role Binding:** Ensures least-privilege access policies for Jenkins, Terraform, and Argo CD.

This layered design ensures reusability, automation, and resilience, closely mirroring enterprise-level DevOps workflows.

Planning & Management

3.1 Project Plan

The project was structured following an iterative DevOps lifecycle, ensuring continuous improvement and automation across all stages — from code development to production deployment. The plan was divided into well-defined phases to streamline execution and maintain traceability across tools and environments.

Project Phases:

1. **Planning & Setup** – Defined scope, selected cloud provider (AWS), and created a VPC with required networking components using Terraform.
2. **Containerization** – Dockerized microservices (Node.js, Python, and Go-based applications) to ensure consistent deployment environments.
3. **Continuous Integration (CI)** – Configured GitHub Actions workflows to automate build, linting, and testing tasks upon each code push or pull request.
4. **Continuous Delivery (CD)** – Integrated ArgoCD with the EKS cluster for GitOps-based deployment automation.
5. **Monitoring & Optimization** – Set up Prometheus and Grafana for real-time monitoring of pod health, cluster resources, and pipeline performance.

Timeline:

- **Phase 1 (Week 1–2):** Infrastructure provisioning using Terraform
- **Phase 2 (Week 3):** Containerization & Docker Hub setup
- **Phase 3 (Week 4):** CI/CD configuration
- **Phase 4 (Week 5):** EKS deployment & GitOps integration
- **Phase 5 (Week 6):** Monitoring, testing, and final optimization

This structured timeline ensured that each module (Code → Build → Deploy → Operate) was independently functional and testable.

3.2 Issue Tracking (GitHub Projects / Jira)

For issue tracking and agile management, **GitHub Projects** was utilized to organize tasks under “To Do”, “In Progress”, and “Completed” columns.

Each feature, bug, or configuration change was logged as an issue linked with corresponding commits and pull requests to maintain transparency and traceability.

Workflow:

- Each module or service was assigned a GitHub Issue with detailed descriptions and acceptance criteria.
- Labels such as *“frontend”*, *“backend”*, *“infrastructure”*, and *“CI/CD”* were used for quick categorization.
- Milestones represented module completion (e.g., *Containerization Complete*, *EKS Setup Done*, *ArgoCD Deployment Live*).
- For critical bugs or blockers, comments and discussions were maintained directly under the issue thread for collaboration.

This systematic issue-tracking strategy ensured timely resolution, efficient communication, and clear documentation of project progress without requiring external tools like Jira.

3.3 Key Findings

- Terraform automation significantly reduced the manual effort required for setting up AWS infrastructure.
- GitHub Actions proved lightweight and fast for CI compared to Jenkins, integrating easily with Docker and GitHub repositories.
- GitOps with ArgoCD enhanced deployment reliability and transparency, reducing human errors during production syncs.
- Using GitHub Projects for task management simplified workflow visualization and boosted productivity during solo development.
- Time-bound iterations made it easier to maintain focus and complete deliverables within a six-week academic schedule.

Infrastructure as Code (IaC)

4.1 Terraform Project Structure

To ensure scalability, readability, and reusability, the Terraform configuration was organized using a modular structure.

Each module handled a specific component of the infrastructure — VPC, EKS, and backend (for remote state management).

Directory Structure:

```
eks/
├── backend/
│   ├── .terraform.lock.hcl
│   ├── main.tf
│   ├── terraform.tfstate
│   └── terraform.tfstate.backup
├── modules/
│   ├── eks/
│   │   ├── main.tf
│   │   ├── outputs.tf
│   │   └── variables.tf
│   └── vpc/
│       ├── main.tf
│       ├── outputs.tf
│       └── variables.tf
├── main.tf
├── variables.tf
├── outputs.tf
└── provider.tf
```

Highlights:

- The modules/vpc directory manages networking (VPC, subnets, gateways, and routing).
- The modules/eks directory provisions the Amazon EKS cluster, IAM roles, node groups, and associated policies.
- The backend directory holds S3 and DynamoDB configurations for Terraform remote state and lock management.
- The root-level main.tf integrates all modules into a unified infrastructure deployment.

4.2 Terraform Configuration & Backend Setup

The project leverages Terraform's S3 backend for storing state files securely, with DynamoDB for state locking to prevent race conditions during simultaneous deployments.

Backend Configuration:

```
terraform {
  backend "s3" {
    bucket         = "demo-terraform-eks-state-s3-bucket-hardik"
    key            = "terraform.tfstate"
    region         = "us-west-2"
    dynamodb_table = "terraform-eks-state-s3-bucket-locks"
    encrypt        = true
  }
}
```

This setup ensures:

- Remote and versioned state management
- Safe collaboration with locking
- Encrypted and centralized storage

4.3 EKS Module Configuration

The EKS module provisions a fully functional Kubernetes control plane and worker node groups with IAM roles and policies.

Key Components:

- Cluster IAM Role with AmazonEKSClusterPolicy attached
- Node IAM Role with worker node policies (EKSWorkerNodePolicy, EKS_CNI_Policy, ECRReadOnly)
- EKS Cluster using input variables for version, VPC, and subnets

Sample Configuration (modules/eks/main.tf):

```
resource "aws_iam_role" "cluster" {
  name = "${var.cluster_name}-cluster-role"
  assume_role_policy = jsonencode({
    Version = "2012-10-17"
    Statement = [{
      Action = "sts:AssumeRole"
      Effect = "Allow"
      Principal = {
        Service = "eks.amazonaws.com"
      }
    }]
  })
}

resource "aws_eks_cluster" "main" {
  name     = var.cluster_name
  version  = var.cluster_version
  role_arn = aws_iam_role.cluster.arn
  vpc_config {
    subnet_ids = var.subnet_ids
  }
}
```

Outputs:

```
output "cluster_endpoint" {
  value = aws_eks_cluster.main.endpoint
}

output "cluster_name" {
  value = aws_eks_cluster.main.name
}
```

4.4 Workflow & Automation

The infrastructure lifecycle followed a consistent command workflow:

```
terraform init       # Initialize provider & backend
terraform plan       # Preview infrastructure changes
terraform apply -auto-approve   # Deploy resources
terraform destroy -auto-approve # Cleanup after module completion
```

Integration with GitHub Actions:

A CI workflow was implemented using GitHub Actions, automatically triggering Terraform validation and plan steps on pull requests to maintain code quality and prevent misconfigurations before deployment.

Integration with Argo CD & Kubernetes:

Once the EKS cluster was active, Argo CD was connected to automate application deployment directly from GitHub repositories — establishing a complete GitOps workflow.

4.5 Key Findings

- **Modular Terraform architecture** enabled code reuse and easy updates.
- **S3 + DynamoDB backend** ensured reliable and collaborative infrastructure state management.
- **EKS + Argo CD integration** established a seamless GitOps pipeline.
- **GitHub Actions CI** reduced manual execution errors in Terraform provisioning.
- **Cost optimization** was achieved by destroying inactive resources between modules.

4.6 Tools Used

Tool / Service	Purpose
Terraform	Infrastructure as Code for provisioning AWS services
AWS EC2	Compute instances for testing and cluster access
AWS VPC	Networking foundation with public/private subnets
AWS IAM	Role-based access and OIDC integration
AWS S3	Remote backend for Terraform state storage
AWS DynamoDB	State locking and concurrency control
AWS EKS (Kubernetes)	Managed Kubernetes cluster for application hosting
Argo CD	GitOps-based Continuous Delivery and cluster sync
Git / GitHub	Version control and repository management
GitHub Actions	Continuous Integration and Terraform automation
kubectl	Kubernetes cluster management CLI
AWS CLI	AWS service management and validation

Source Code Management

5.1 Version Control with Git & GitHub

Version control was implemented using Git, with GitHub serving as the central repository hosting platform. The project is based on a forked open-source repository —ultimate-devops-project-demo — originally inspired by the Open Telemetry Astronomy Shop microservices demo.

This repository provided a real-world multi-service codebase to demonstrate DevOps automation, observability, and cloud-native deployment practices. I extended and customized the project by integrating Terraform-based infrastructure provisioning, GitHub Actions for CI, and Argo CD for GitOps-driven CD on AWS EKS.

Repository Structure

The repository was structured into modular directories to promote separation of concerns and ease of automation:

Directory / File	Description
`.github/workflows/`	Contains CI pipelines implemented via GitHub Actions
`kubernetes/`	Kubernetes deployment manifests for each microservice
`terraform/`	Terraform IaC modules for AWS VPC, EKS, and IAM resources
`src/` & `internal/`	Application source code (Go, Java, and Python services)
`docker/`	Dockerfiles for each microservice
`README.md`	Documentation for setup, configuration, and contribution

Each commit was carefully tagged and versioned to reflect changes across both application and infrastructure layers, following semantic versioning principles.

Key Highlights:

- Maintained a clean and auditable Git history
- Integrated IaC and application source in a single repository for unified management
- Implemented conventional commit messages (`feat:`, `fix:`, `refactor:`) for better traceability

5.2 Branching Strategy

A feature-driven branching model was adopted to facilitate parallel development and stable integration cycles.

This model followed a simplified Git Flow strategy:

Branch Name	Purpose
`main`	Stable production-ready code automatically deployed via Argo CD
`dev`	Active integration branch where new features are merged and tested
`feature/*`	Individual branches for implementing new services, Terraform modules, or CI improvements
`hotfix/*`	Used for urgent production patches or configuration fixes

All new features were developed in isolated branches derived from dev.
Pull Requests (PRs) were created for review, validated via GitHub Actions CI, and merged upon successful testing.
This process ensured code consistency, review transparency, and safe deployment to the main branch.

Example Workflow:

1. Create branch → feature/add-argo-cd-pipeline
2. Commit & push → triggers GitHub Actions
3. Validate CI → merge into dev
4. Stable version → merge into main
5. Argo CD detects changes → auto-syncs to AWS EKS

5.3 GitHub Actions Integration

To automate the Continuous Integration (CI) workflow, GitHub Actions was configured using YAML-based workflow definitions stored in .github/workflows/.This eliminated the need for a separate Jenkins setup and provided a lightweight, cloud-native CI solution fully integrated within the repository.

Key Workflow Features:

- **Trigger:** On every push or pull request to dev or main
- **Jobs:** Build → Test → Dockerize → Push to Docker Hub → Notify Argo CD
- **Environment:** Runs on ubuntu-latest virtual runners
- **Docker Hub Integration:** Builds versioned images using commit SHAs

Example Workflow (CI Build & Deploy)

```
name: CI Build & Deploy

on:

  push:

    branches: [ "main", "dev" ]

  pull_request:


jobs:

  build:

    runs-on: ubuntu-latest

    steps:

      - name: Checkout Repository

        uses: actions/checkout@v3


      - name: Set up Node.js

        uses: actions/setup-node@v3

        with:

          node-version: '18'


      - name: Install Dependencies

        run: npm ci


      - name: Run Tests

        run: npm test
      - name: Build Docker Image

        run: docker build -t hardikajmeriya/devopsapp:${{ github.sha }} .



      - name: Push to Docker Hub

        run: |

          docker login -u ${{ secrets.DOCKERHUB_USERNAME }} -p ${{
secrets.DOCKERHUB_TOKEN }}

          docker push hardikajmeriya/devopsapp:${{ github.sha }}
```

This automation ensures that each code change is built, tested, and containerized immediately after it's committed.

Successful builds are automatically tagged and pushed to Docker Hub, triggering Argo CD to update the Kubernetes deployments in EKS — completing the GitOps pipeline.

5.4 Key Findings

- Simplified DevOps Workflow: Using GitHub Actions replaced complex Jenkins configurations with a natively integrated CI platform.
- Improved Code Integrity**:** Automated testing and build validation prevented broken commits from reaching production.
- GitOps Integration: Argo CD automatically synchronized infrastructure and code changes from GitHub to EKS.
- Cost Efficiency: GitHub-hosted runners reduced the need for maintaining dedicated build servers.
- End-to-End Automation**:** CI pipelines seamlessly connected to Terraform, Docker, and Kubernetes layers.

5.5 Tools Used

Category	**Tool / Service**	**Purpose**
Version Control	**Git**	Local version tracking
Repository Hosting	**GitHub**	Centralized repository management
CI Automation	**GitHub Actions**	Automated builds, tests, and image deployments
IaC	**Terraform**	Infrastructure provisioning for AWS resources
Containerization	**Docker**	Building and packaging application services
CD / GitOps	**Argo CD**	Continuous deployment via GitOps principles
Orchestration	**Kubernetes (AWS EKS)**	Application scaling and cluster management

Build & Test Automation

6.1 Build Process (Docker, GitHub Actions Pipeline)

The project's build process was designed to ensure every code change could be automatically packaged, tested, and deployed without manual intervention.The Continuous Integration (CI) phase was primarily implemented using GitHub Actions, while Docker was used for containerization.

Each microservice and infrastructure component was containerized to ensure portability and consistency across all environments — from local development to AWS EKS.

Build Workflow Overview:

1. Source Code Checkout – Upon each commit or pull request, GitHub Actions automatically cloned the repository and initialized the workflow.
2. Dependency Installation – The required dependencies (Node.js packages, Terraform providers, etc.) were installed.
3. Linting and Code Quality Checks – Automated syntax validation and lint checks were performed for YAML, Dockerfile, and Terraform configurations.
4. Docker Image Build – Each service was built into a Docker image using its dedicated Dockerfile.

```
docker build -t hardikajmeriya/<service-name>:latest .
```

5.Image Tagging & Push – Successfully built images were tagged with version identifiers and pushed to a container registry (e.g., Docker Hub or AWS ECR).

6.Artifact Verification – Build logs, image digests, and workflow artifacts were verified before triggering deployment.

Integration with Deployment Tools:

Once the Docker image was successfully pushed, the workflow triggered **Argo CD**, which automatically pulled the updated image into the Kubernetes cluster. This completed the CI → CD flow.

6.2 Automated Testing Stages

Testing was integrated directly into the CI pipeline to ensure build quality, reliability, and performance before deployment.

Testing Stages Implemented:

1. **Unit Testing**
   - Focused on verifying individual modules and API endpoints.
   - Implemented using lightweight Node.js test frameworks (e.g., Jest or Mocha).
   - Automatically executed after the build stage via GitHub Actions YAML workflow.
2. **Integration Testing**
   - Verified communication between microservices after containerization.
   - Ensured that APIs, databases, and other service dependencies were functioning correctly.
   - Used temporary Docker containers spun up during workflow runs.

3. **Infrastructure Validation Tests**
    - Checked Terraform syntax (terraform validate) and Kubernetes YAMLs (kubectl apply --dry-run=client).
    - Ensured that IaC and deployment manifests were properly formatted before applying.
4. **Container Image Security & Linting**
    - Used tools like **Hadolint** to scan Dockerfiles for best practices.
    - Ensured images were minimal, secure, and efficient.
5. **Deployment Readiness Check**
    - Post-build verification ensured that the final images could be successfully pulled, deployed, and run inside Kubernetes pods without errors.

All test results, logs, and failure traces were made visible in the GitHub Actions "Checks" tab, ensuring full transparency and easy debugging.

6.3 Key Findings

- **Docker-based builds** simplified environment consistency, reducing "it works on my machine" issues.
- **GitHub Actions** provided seamless integration with version control, replacing the need for Jenkins servers in this student project.
- **Automated linting and validation** prevented deployment of misconfigured YAML or Terraform files.
- **Test automation** significantly reduced manual verification time.
- **Argo CD integration** ensured that only successfully tested builds were deployed to Kubernetes.
- The overall **CI/CD chain** — from code commit → build → test → deploy — ran entirely in the cloud with minimal manual involvement.

6.4 Tools Used

Tool / Platform	Purpose
Docker	Containerization of microservices for environment consistency.
GitHub Actions	Automated CI workflows for building, testing, and deploying applications.
Terraform	Infrastructure provisioning for AWS using IaC principles.
Kubernetes (EKS)	Container orchestration for scalable application deployment.
Argo CD	Continuous Deployment tool to automate release management using GitOps.
YAML Lint	Static code analysis for Dockerfiles and configuration files.
Node.js	Unit and integration testing framework for backend services.

Continuous Integration & Continuous Delivery (CI/CD)

7.1 Overview

The CI/CD pipeline forms the core automation engine of this DevOps project, enabling a seamless transition from code commit to deployment on a live Kubernetes environment.
In this project, GitHub Actions handled Continuous Integration (CI), while Argo CD implemented Continuous Delivery (CD) using a GitOps approach on AWS EKS.

This pipeline ensured that every code update triggered an automated sequence — build, test, image creation, push to registry, and deployment — ensuring continuous feedback, consistency, and reliability throughout the software lifecycle.

7.2 Continuous Integration (GitHub Actions)

Instead of using Jenkins, the project leveraged GitHub Actions for CI automation because of its tight integration with GitHub repositories and cost-effectiveness for individual developers.

Pipeline Workflow:

1. Trigger:
   The CI pipeline is triggered automatically on every push or `pull_request` event to the repository's main or `githubcicheck` branches.
2. Checkout & Setup:
   The workflow pulls the latest code and sets up the environment with required dependencies (Node.js, Docker CLI, etc.).
3. Build & Test:
   - Executes automated linting, unit tests, and integration tests.
   - Validates Terraform syntax (terraform validate) and Kubernetes YAML (`kubectl apply --dry-run=client`).
4. Docker Build:
   Builds Docker images for each microservice using its `Dockerfile`.

Example:

```
docker build -t hardikajmeriya/app-service:v1 .
```

1. Push to Docker Hub:
   Successfully built images are tagged and pushed to Docker Hub, serving as the image repository for deployment.
2. docker push hardikajmeriya/app-service:v1
3. Notify Argo CD:
   Once new images are pushed, Argo CD detects the change in the Git repository and automatically triggers deployment synchronization.

The CI pipeline ensures that only successfully tested, validated builds are moved to the deployment stage, maintaining production stability.

7.3 Continuous Delivery (Argo CD)

Argo CD was the backbone of the Continuous Delivery pipeline. It enabled GitOps-based automation — where the desired state of applications was defined in a Git repository, and any change was automatically reflected in the Kubernetes cluster.

Key Deployment Steps:

1. Application Manifests in Git:
   The Kubernetes manifests and Helm charts for each microservice are stored in a Git repository (/deploy folder).
2. Argo CD Configuration:
   - Deployed inside the EKS cluster using a dedicated namespace argocd.
   - Connected to the GitHub repository for automatic sync.
3. Auto-Sync Mechanism:
   Whenever a new image tag or configuration update is pushed to the repo, Argo CD automatically updates the running application in EKS.
4. Load Balancer Configuration:
   Integrated with AWS Load Balancer Controller to expose applications externally using a LoadBalancer service type.
5. Custom Domain Routing:
   A domain devopsbyhardik.shop was configured to route production traffic to the Kubernetes service.
   (Currently inactive due to limited AWS credits as a student project)

The complete deployment flow — from Git commit to production — required no manual kubectl commands, representing a true GitOps implementation.

7.4 Key Findings

- Replaced traditional Jenkins CI/CD with a fully GitHub Actions + Argo CD pipeline — modern, lightweight, and cost-free for personal projects.
- Automated Docker image versioning, build validation, and Kubernetes deployment significantly reduced manual work.
- Argo CD's auto-sync ensured real-time updates whenever the Git repository changed.
- Integration with AWS EKS provided scalability and reliability for microservices.
- Successfully implemented end-to-end GitOps workflow — from source to running pods — within a secure AWS environment

7.5 Tools Used

Tool / Platform	Purpose
GitHub Actions	Continuous Integration – automates build, test, and Docker image push on every commit.
Docker / Docker Hub	Containerization of services and centralized image hosting.
Argo CD	Continuous Delivery – GitOps-based automated deployment on EKS.
Kubernetes (EKS)	Orchestration platform for deploying and managing containers.
Terraform	Infrastructure provisioning for AWS resources (EKS, VPC, IAM, etc.).
AWS Load Balancer Controller	Traffic routing to deployed services via LoadBalancer type.
AWS Route 53 / Domain Setup	Custom domain (devopsbyhardik.shop) routing for production deployment.

Deployment

8.1 AWS EKS Cluster Setup

The project deployment environment was based on Amazon Elastic Kubernetes Service (EKS). Using Terraform as Infrastructure as Code (IaC), the following was provisioned:

- EKS Cluster with specified Kubernetes version.
- VPC, subnets, and networking resources for cluster communication.
- IAM roles for cluster control plane and worker nodes.
- Node groups for running containerized microservices.

Terraform modules allowed modular and reusable infrastructure setup. Once the cluster was created, kubectl was configured with the generated kubeconfig to interact with the cluster and validate resources.

8.2 Argo CD Configuration & GitOps Workflow

Argo CD was deployed on the EKS cluster to enable GitOps-based continuous deployment. Key configuration steps included:

- Installing Argo CD in a dedicated namespace (argocd).
- Connecting Argo CD to the GitHub repository containing Kubernetes manifests.
- Creating applications in Argo CD, defining the source repository, target cluster, and destination namespace.
- Enabling auto-sync so that any updates in the Git repository automatically reflect in the cluster.

This workflow ensured that deployments were fully automated, consistent, and auditable, eliminating manual deployment errors.

8.3 AWS Load Balancer Controller Integration

To expose the applications externally:

- AWS Load Balancer Controller was installed on the EKS cluster.
- Required IAM roles and permissions were configured using Terraform.
- Kubernetes Service of type LoadBalancer was applied to each microservice.
- Traffic routing, SSL termination, and health checks were managed automatically by the controller.

This allowed external users to access deployed applications reliably while maintaining high availability.

8.4 Custom Domain Setup (devopsbyhardik.shop)

A custom domain was configured to provide a production-level endpoint for applications:

- Domain registration and DNS configuration were prepared in Route 53.
- External access was routed via AWS Load Balancer to the services.
- SSL certificates and routing rules were planned for secure traffic.

Note: The domain is currently inactive due to AWS cost constraints as this is a student project.

8.5 Key Findings

- Terraform modularization enabled consistent and reproducible cluster deployments.
- Argo CD's GitOps workflow allowed automatic synchronization from Git repository to EKS.
- AWS Load Balancer Controller simplified service exposure and routing configuration.
- Custom domain integration demonstrated a production-like environment, though temporarily inactive.
- This deployment approach closely mirrors professional DevOps workflows used in industry-scale applications.

8.6 Tools Used

Tool / Platform	Purpose
AWS EKS	Kubernetes cluster orchestration and management
Terraform	Infrastructure provisioning for EKS, VPC, IAM, subnets, and node groups
Argo CD	GitOps-based continuous deployment
AWS Load Balancer Controller	External access, load balancing, and routing to services
AWS Route 53	Custom domain DNS configuration

Operations & Monitoring

9.1 Container Management (EKS Pods, Services, Ingress)

After deployment, proper container management was essential to ensure application stability and scalability:

- Pods:
  Each microservice was deployed as a Kubernetes pod in the EKS cluster.
  Pod status, health, and logs were monitored using kubectl and Argo CD dashboard.
- Services:
  Kubernetes Services were used to provide stable endpoints for pods and enable internal communication between microservices.
  Type LoadBalancer services were integrated with AWS Load Balancer Controller to provide external access.
- Ingress:
  Kubernetes Ingress resources were configured for routing traffic from the Load Balancer to specific services based on paths and hostnames.
  This allowed multiple applications to share the same LoadBalancer efficiently.

This container management setup ensured high availability, scalability, and fault tolerance for the microservices.

9.2 Application Monitoring & Logging (CloudWatch / Argo CD Dashboard)

Monitoring and logging are crucial for identifying issues and maintaining system health:

- Argo CD Dashboard:
  Provided a real-time view of deployed applications, their synchronization status, health, and any configuration drift.
  Enabled developers to track deployment progress and detect inconsistencies quickly.
- AWS CloudWatch:
  Used for logging container outputs, pod metrics, and cluster-level monitoring.
  CloudWatch metrics included CPU, memory usage, pod health, and network traffic.
  Alerts could be configured to notify of any abnormal behavior or resource constraints.
- Log Aggregation:
  Application logs were aggregated for easier troubleshooting and debugging, reducing mean time to recovery (MTTR) for failures.

This monitoring setup ensures a proactive approach to application management and aligns with industry-standard DevOps practices.

9.3 Key Findings

- Kubernetes pods, services, and ingress provided efficient container orchestration for microservices.
- Argo CD offered visual monitoring and management of deployments, ensuring synchronization with Git repository changes.
- CloudWatch provided centralized logging and resource utilization metrics, improving observability of the EKS cluster.
- Combining Argo CD and CloudWatch created a robust monitoring ecosystem, which is essential for production-grade deployments.
- Operations workflows closely mimic professional DevOps practices used in enterprise cloud environments.

9.4 Tools Used

Tool / Platform	Purpose
Kubernetes (EKS)	Container orchestration and pod management
Argo CD	Deployment monitoring, health status, and GitOps synchronization
AWS CloudWatch	Metrics collection, logging, and alerting for cluster and applications
kubectl	Command-line tool to manage pods, services, and ingress resources
AWS Load Balancer Controller	Traffic routing to services and ingress management

Security & Access Management

10.1 IAM Role Configuration

AWS Identity and Access Management (IAM) roles were used to securely control access to AWS resources:

- Cluster Role:
  An IAM role was created for the EKS control plane, granting permissions to manage cluster resources and communicate with AWS services.
- Node Role:
  Worker nodes received a dedicated IAM role with permissions to pull container images, access VPC resources, and integrate with AWS services such as CloudWatch.
- Policy Attachments:
  Standard AWS managed policies such as AmazonEKSClusterPolicy, AmazonEKSWorkerNodePolicy, AmazonEC2ContainerRegistryReadOnly, and AmazonEKS_CNI_Policy were attached to the respective roles to enforce the principle of least privilege.

This configuration ensured that each component had only the permissions required to perform its tasks, reducing security risks.

10.2 OIDC Provider Setup

To enable secure authentication between the EKS cluster and AWS:

- IAM OIDC Provider was configured for the cluster.
- This allowed Kubernetes service accounts to assume IAM roles, enabling fine-grained access to AWS resources without using long-lived credentials.
- This approach supported workload identity, ensuring that pods could securely interact with AWS services (e.g., S3, DynamoDB) with minimal manual credential management.

10.3 Secure Secret Management

Managing secrets and sensitive information was crucial for secure deployments:

- AWS Secrets Manager and S3 were used to store sensitive configuration data such as database credentials and API keys.
- Secrets were mounted as environment variables or Kubernetes secrets in pods to avoid storing sensitive data directly in code repositories.
- Access to secrets was controlled via IAM policies and the OIDC provider, ensuring least privilege access and auditing capabilities.

This setup ensured that secrets were encrypted at rest, securely injected into workloads, and only accessible to authorized pods and users.

10.4 Key Findings

- IAM roles and policies ensured least privilege access across cluster and worker nodes.
- OIDC provider integration enabled secure, automated authentication for Kubernetes workloads.
- Secrets management using AWS services ensured confidentiality, integrity, and secure distribution of sensitive data.
- Security configurations closely mimic industry best practices for cloud-native DevOps pipelines.

10.5 Tools Used

Tool / Platform	Purpose
AWS IAM	Role-based access control for EKS cluster and worker nodes
IAM OIDC Provider	Workload identity for secure authentication without long-lived credentials
AWS Secrets Manager	Secure storage and management of sensitive data and credentials
AWS S3	Storage for encrypted configuration and secret data
Kubernetes Secrets	Inject secrets securely into pods for application use

Communication & Collaboration

11.1 Team Communication (Slack / GitHub Issues)

Effective communication was crucial for coordinating project tasks, tracking progress, and resolving issues. Slack was used for real-time discussions, sharing updates, and clarifying tasks or technical challenges. This setup allowed fast feedback and seamless collaboration, even when the team was distributed across different locations. Slack channels were organized by project modules such as CI/CD, deployment, and monitoring, which helped reduce noise and maintain focus on relevant discussions. In addition, GitHub notifications, including pull requests, commits, and issue updates, were integrated into Slack to provide automated alerts and maintain situational awareness.

GitHub Issues served as the primary platform for task tracking, bug reporting, and feature requests. Each issue was assigned a priority, status, and labels to ensure structured project management. Issues were linked to specific branches and pull requests for full traceability, enabling the team to track progress from identification to resolution. Milestones were used to group tasks according to project modules, ensuring alignment with project objectives. The combination of Slack and GitHub Issues provided a structured communication framework, ensuring all activities were transparent, traceable, and efficiently managed, reducing delays and minimizing miscommunication.

11.2 Documentation Maintenance

Maintaining thorough and up-to-date documentation was critical for project continuity, knowledge sharing, and reproducibility. The README in the GitHub repository provided a comprehensive overview of project setup, workflow, and infrastructure instructions. The GitHub Wiki supplemented this with detailed guides on modules such as CI/CD, Terraform infrastructure setup, Kubernetes deployment, and Argo CD configuration.

Inline comments in Terraform and Kubernetes files explained the purpose of each module, resource configuration, and environment variables, while the GitHub Actions workflow files were annotated to describe each job, stage, and conditional execution. Additionally, OpenTelemetry documentation detailed the observability setup for tracing and metrics, explaining how distributed traces were captured, exported, and visualized. This allowed developers to monitor application performance, latency, and detect bottlenecks across microservices. All documentation updates were version-controlled in Git, ensuring historical reference and minimizing the risk of outdated instructions. This comprehensive documentation approach ensured that any developer or reviewer could understand, replicate, or extend the project setup without ambiguity.

Conclusion

12.1 Project Summary

This project represents the successful completion of a comprehensive end-to-end DevOps implementation, encompassing containerized microservices, infrastructure provisioning, continuous integration and deployment, and robust monitoring and observability. The primary objective was to design, build, and deploy a multi-service application on a cloud-native environment, following modern DevOps practices. The deployment leveraged AWS EKS as the Kubernetes orchestration platform, with networking, security, and cluster resources provisioned via Terraform, ensuring infrastructure-as-code capabilities.

Containerized microservices were built using Docker, with images stored and versioned on Docker Hub. CI/CD workflows were implemented using GitHub Actions, providing automated build, test, and deployment pipelines. For deployment management and GitOps practices, Argo CD was integrated with the EKS cluster to automate synchronization of applications from the Git repository. This workflow ensured that any changes in the repository were automatically reflected in the live cluster, reducing manual intervention and the risk of human error.

Security and access management were implemented using AWS IAM roles for the cluster and node groups, along with OIDC provider integration to enable secure pod-to-AWS service authentication. Secrets were managed via AWS Secrets Manager and Kubernetes secrets, maintaining confidentiality and integrity of sensitive configuration data. Observability was established through OpenTelemetry and AWS CloudWatch, providing detailed metrics, distributed tracing, and logging for all deployed services.

The deployment process also incorporated AWS Load Balancer Controller to expose services externally and demonstrated integration with a custom domain, devopsbyhardik.shop, creating a near-production-level setup. Although the domain and some components are not currently active due to resource constraints as a student project, the architecture and workflows replicate professional-grade cloud deployments, providing significant learning and practical experience.

Overall, the project brought together multiple facets of modern DevOps, including container orchestration, infrastructure automation, CI/CD pipelines, GitOps, monitoring, security, and collaboration tools, creating a holistic, production-ready solution.

12.2 Future Enhancements

While the project achieved its core objectives, several potential enhancements could further improve functionality, scalability, and maintainability. First, autoscaling policies for both pods and node groups could be fully implemented and optimized, allowing the cluster to dynamically adjust resource allocation based on workload demand. This would improve cost efficiency and ensure high availability during peak usage periods.

Second, the project could be extended to include multi-region or hybrid-cloud deployment, ensuring disaster recovery, lower latency for users, and fault tolerance. This would also require a more advanced networking setup with cross-region VPC peering or service mesh integration. Implementing a service mesh such as Istio or Linkerd could enhance traffic management, secure service-to-service communication, and provide better observability across the microservices architecture.

Third, the CI/CD pipeline could be enhanced with automated rollback strategies, chaos testing, and policy-as-code enforcement. Integrating tools like OPA (Open Policy Agent) would ensure compliance and governance at deployment time. Further, advanced monitoring could be added with Prometheus and Grafana dashboards, complementing OpenTelemetry traces, for richer visualization and real-time alerts.

Security could be further strengthened through automatic rotation of secrets, integration with AWS KMS for encryption, and role-based access policies at the application level. Additionally, adding end-to-end testing with integration tests and load tests in the CI/CD workflow would improve reliability and resilience of the deployed microservices.

Finally, the project can be extended to include cost management dashboards, automated alerts for budget thresholds, and optimization recommendations for AWS resources. This would not only provide visibility into cloud expenses but also teach best practices for managing student or startup budgets efficiently while running production-like workloads.

12.3 Learnings & Outcomes

The project provided extensive hands-on experience across multiple domains of cloud computing, DevOps, and modern software engineering. One of the most significant learnings was understanding the interdependence of infrastructure, CI/CD, deployment strategies, and monitoring. By provisioning infrastructure via Terraform and automating deployments with GitHub Actions and Argo CD, it became clear how automation improves reliability, repeatability, and efficiency.

Another key outcome was mastering Kubernetes concepts, including pods, deployments, services, ingress, and node groups, and learning to integrate these components with cloud services like AWS Load Balancer and IAM. Understanding workload identity through OIDC and secure secrets management strengthened knowledge of cloud security practices and least-privilege design principles.

The use of GitHub Actions for CI and Argo CD for CD provided practical exposure to GitOps principles, showing the value of declarative infrastructure and automated application synchronization. Monitoring and observability through OpenTelemetry and CloudWatch taught the importance of tracing, metrics, and proactive alerting in maintaining healthy production systems.

Beyond technical skills, the project improved project planning, documentation, and collaboration. Tools like Slack, GitHub Issues, and comprehensive documentation ensured that tasks were tracked, communication was clear, and knowledge could be shared efficiently. This reinforced the importance of combining technical execution with process discipline for successful DevOps projects.

In conclusion, the project outcomes demonstrate readiness to implement enterprise-grade DevOps pipelines, from infrastructure provisioning and container orchestration to CI/CD, GitOps deployment, monitoring, and secure operations. The hands-on experience gained, coupled with documentation and observability practices, provides a strong foundation for future projects or professional work in cloud computing, DevOps, and full-stack application deployment.