INTERNATIONAL JOURNAL OF NOVEL RESEARCH AND DEVELOPMENT (IJNRD)
© 2026 IJNRD | Volume 11, Issue 6, June 2026 | ISSN: 2456-4184 | IJNRD.ORG

DEPLOYING GOLANG APPLICATION ON KUBERNETES

[1]Atharva Paithankar, [2]Sakib Shaikh, [3]Mayur Vithore, [4]Prof. Varsha Nanavare
[1]Student, Dept. of Electronics & Telecommunication Engineering
[2]Student, Dept. of Electronics & Telecommunication Engineering
[3]Student, Dept. of Electronics & Telecommunication Engineering
[4]Professor, Dept. of Electronics & Telecommunication Engineering
RMD Sinhgad School of Engineering, Warje, Pune, Maharashtra, India

Abstract *— This paper presents a complete DevOps pipeline for a secure Golang-based Cloud-Native Employee Management System (Employee MS) deployed on a Kubernetes cluster. The application provides role-based access control (RBAC) with Admin, Manager, and Employee roles, employee CRUD operations, and comprehensive API audit logging. The system integrates containerization using KO and Docker with BuildSafe for zero-CVE image builds, automated CI/CD using GitHub Actions and ArgoCD following GitOps principles, high-availability PostgreSQL database management using CloudNativePG with automated primary-replica failover and PgBouncer connection pooling, real-time observability using Prometheus and Grafana dashboards, and performance validation using K6 load testing with Kubernetes Horizontal Pod Autoscaling (HPA). HTTPS is enabled via Cert-Manager and the Kubernetes Gateway API. Experimental results demonstrate sub-200ms average response times under 100 concurrent virtual users, throughput of 1,247 req/sec, 99.98% HTTP 200 success rate, successful HPA scaling from 2 to 4 pods at peak load, zero known CVEs in the production container image, and a fully automated end-to-end deployment pipeline completing in approximately 3 minutes. This work serves as a reproducible blueprint for modern cloud-native DevOps pipelines combining security, observability, and automation.*

Index Terms *— DevOps, Kubernetes, GoLang, CI/CD, Containerization, Cloud-Native, ArgoCD, GitOps, Prometheus, Grafana, RBAC, Security, Observability, CloudNativePG, Employee Management System.*

I. INTRODUCTION

In general, DevOps practices are necessary in this cloud-native system era to provide reliable, secure, observable, and scalable applications. The goal of this project is to illustrate a full chain from code to production in a highly automated manner, applying best practices in containerization, orchestration, monitoring, CI/CD, security, and performance testing. The chosen application is written in GoLang, leveraging the language's performance and simplicity, and connected to a PostgreSQL database to meet enterprise data-store requirements.

Combining tools like KO for container builds, Docker for local development, Kubernetes for orchestration, Cert-Manager and the Gateway API for secure ingress, CloudNativePG for database lifecycle management, Prometheus and Grafana for monitoring, GitHub Actions and ArgoCD for CI/CD, BuildSafe for secure image generation, and K6 for load testing — this project serves as a blueprint for modern DevOps pipelines. In today's software industry, the demand for scalable, reliable, and continuously available applications has led to the rapid adoption of DevOps and cloud-native architectures. DevOps integrates development and operations processes through automation, continuous integration, and continuous delivery, enabling faster release cycles and improved software quality. Meanwhile, cloud-native technologies like containers and Kubernetes allow applications to be developed, deployed, and managed at scale with maximum portability.

➢ Problem Statement

Traditional application deployment approaches are often manual, time-consuming, and error-prone. Existing systems suffer from a lack of automation, significant deployment delays, scalability issues, and high operational overhead. Although containerization and orchestration technologies exist, their integration with automated pipelines is often complex and lacks standardization. Furthermore, security vulnerabilities in container images and the absence of real-time observability make it difficult to detect and respond to production incidents quickly. Emergency scenarios such as hardware failures, traffic spikes, or security breaches require immediate response mechanisms that fixed, manual deployment systems cannot provide. Therefore, there is a critical need for a unified system that integrates Golang application development with automated CI/CD, secure container builds, Kubernetes deployment, and complete observability.

➢ Objectives

- Develop a GoLang web application (Employee MS) with role-based access control (RBAC), employee CRUD operations, and API audit logging.

- Containerize the application using KO and Docker, ensuring zero known CVEs using BuildSafe security scanning.
- Set up a Kubernetes cluster using ksctl and deploy the application with HTTPS via Cert-Manager and Gateway API.
- Implement a fully automated CI/CD pipeline using GitHub Actions for CI and ArgoCD for CD following GitOps principles.
- Configure real-time observability using Prometheus and Grafana dashboards with custom application metrics.
- Manage the PostgreSQL database using CloudNativePG for high availability and automated failover with PgBouncer connection pooling.
- Perform load testing using K6 and validate system performance and Kubernetes HPA behavior under high traffic.
- Demonstrate RBAC enforced at the API level and implement a full API audit trail tracking all user actions.

II. LITERATURE REVIEW

Several researchers and practitioners have documented intelligent approaches to container orchestration, CI/CD pipelines, and cloud-native observability. The following sections summarize key findings from related work.

A. Kubernetes Best Practices

Kubernetes has become the de-facto orchestration platform for containerized, cloud-native applications. Multiple surveys and practical guides emphasize best practices for production clusters — high availability (control-plane redundancy), resource governance (namespaces, quotas), secure defaults, and robust day-2 operations such as rolling upgrades, backup/restore, and cluster observability. These best practices reduce downtime and operational complexity when running microservices at scale. The integration of Kubernetes liveness and readiness probes, as used in this work through a dedicated /health endpoint, is a widely established practice for automated pod health management [1].

B. DevOps Practices and Software Quality

Adopting DevOps practices — CI/CD, infrastructure as code, automated testing, and GitOps — strongly correlates with improved software product quality, faster release cycles, and better collaboration between development and operations teams. Systematic mappings and empirical studies report that automation of build-test-deploy pipelines reduces human error and shortens mean time to recovery. GitOps workflows using tools such as ArgoCD and Flux enable declarative, version-controlled deployments with automatic rollback capability, improving deployment reliability and auditability [2][3].

C. Challenges in DevOps and Container Orchestration Adoption

Despite clear benefits, organizations face notable challenges when adopting DevOps and container orchestration. Literature repeatedly highlights cultural resistance, unclear ownership boundaries between development and operations, legacy systems integration, and skills gaps as primary barriers. Technical challenges include managing multi-cluster deployments, secure configuration, and maintaining consistent environments across development and production. These barriers often delay or complicate the effective use of Kubernetes and automated pipelines [4][9].

D. Observability in Cloud-Native Environments

Observability — metrics, logs, and traces — is essential for operating production Kubernetes systems. Studies focusing on Prometheus and Grafana show that metric-based alerting, service-level dashboards, and exporter instrumentation (node, kube-state, application metrics) greatly improve troubleshooting and capacity planning. The literature stresses designing meaningful metrics and alert thresholds to avoid alert fatigue and to enable proactive remediation. Custom application metrics exposed via the /metrics endpoint allow fine-grained tracking of business-level indicators [5][6][10][11].

E. Gap Analysis

While prior work documents Kubernetes best practices, DevOps impacts, and monitoring approaches at an enterprise scale, there is less literature that ties these practices to a complete, end-to-end project-level implementation: developing a GoLang microservice, containerizing it, building an automated CI/CD pipeline, deploying to Kubernetes, and integrating application-level observability with security scanning and load testing. This paper addresses that gap by presenting a validated, integrated pipeline applied to a real-world employee management application.

III. METHODOLOGY

The proposed system is built on a cloud-native architecture that integrates application development, containerization, orchestration, CI/CD automation, observability, and security scanning into a single cohesive pipeline. The architecture consists of the following major subsystems: a GoLang application with PostgreSQL backend; Docker and KO-based containerization with BuildSafe security

scanning; a Kubernetes cluster managed via ksctl; GitHub Actions for CI and ArgoCD for CD following GitOps principles; Prometheus and Grafana for observability; CloudNativePG for database lifecycle management; and K6 for load testing.

A. Working Principle

The proposed system operates by combining local application development with automated pipeline-based deployment. The developer writes the GoLang application code, integrates it with PostgreSQL, and tests it locally using Docker. Once satisfied, the code is pushed to a GitHub repository. This push event automatically triggers the GitHub Actions CI pipeline, which builds a Docker/KO container image, runs security scans via BuildSafe, executes unit tests, and pushes the verified image to Docker Hub. ArgoCD, operating as the CD tool in a GitOps model, continuously monitors the GitHub repository for changes in Kubernetes manifests. Upon detecting changes, it automatically synchronizes the cluster state to match the desired state defined in the repository. Kubernetes then manages pod scheduling, scaling, and health monitoring. Prometheus scrapes metrics from the running application pods, and Grafana visualizes these metrics through interactive dashboards. CloudNativePG manages the PostgreSQL cluster, ensuring high availability through automated failover and replication.

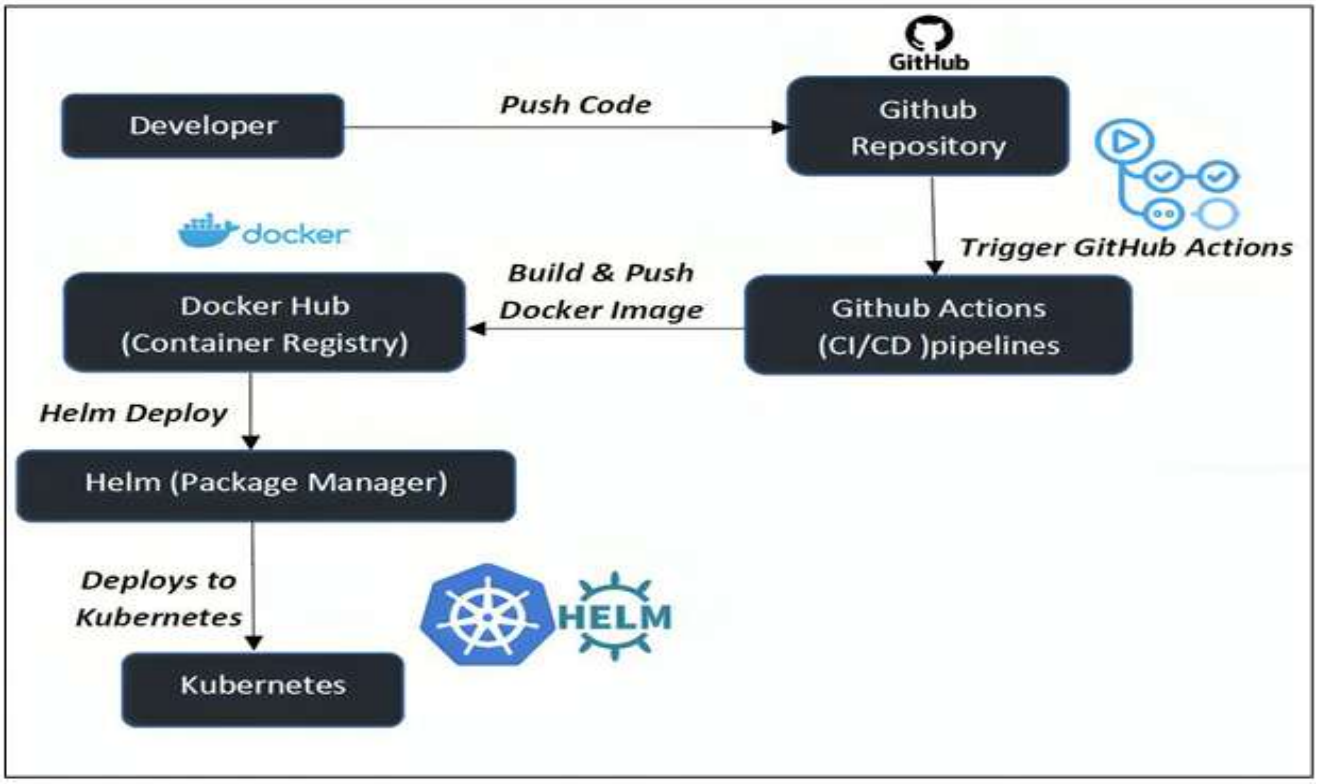

Fig. 1: Overall System Architecture — GoLang App → Docker Hub → ArgoCD → Kubernetes → Prometheus/Grafana

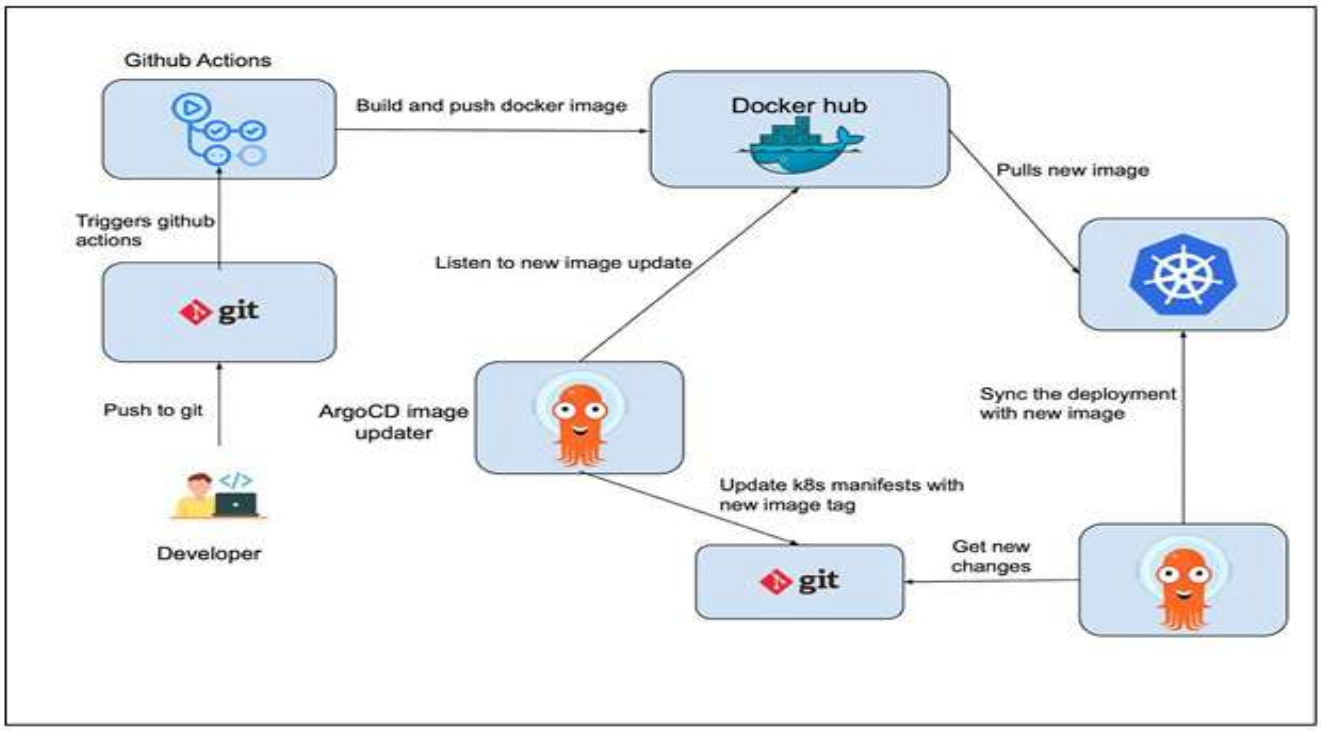

Fig. 2: CI/CD Pipeline Flowchart — GitHub Actions → Docker Hub → ArgoCD Image Updater → Kubernetes rolling update

B. Application Development — Employee MS

The Employee MS is a Golang web application following an MVC-like architecture. The application is structured as a single compiled binary that serves both the HTTP REST API and the HTML web interface. All HTML templates are embedded in the binary at compile time using Go's html/template package, ensuring the container image contains no external file dependencies. The application implements session-based authentication, middleware-enforced RBAC, structured audit logging, and custom Prometheus metrics exposure.

Key endpoints exposed by the application are shown in Table I.

Endpoint	Method	Description	Access
/login	GET/POST	Authentication page and credential validation	Public
/dashboard	GET	Main dashboard with system statistics	All Roles
/employees	GET	Employee list with search/filter	Admin, Manager
/employees/add	POST	Add new employee record	Admin only
/employees/edit	POST	Update employee details	Admin, Manager
/employees/delete	POST	Remove employee record	Admin only
/admin	GET	Admin panel with users and audit logs	Admin only
/health	GET	Health check for Kubernetes probes	Public
/metrics	GET	Prometheus metrics endpoint	Public

Table I: Employee MS API Endpoints

C. Containerization and Secure Image Build

The application was containerized using a multi-stage Dockerfile with a golang:1.21-alpine build stage and a minimal alpine runtime stage, and also using KO for streamlined Go container builds. The multi-stage approach ensures the final image contains only the application binary and its runtime dependencies, minimizing both image size and attack surface. BuildSafe was integrated into the CI pipeline to perform security scanning, producing a container image with zero known CVEs. Docker BuildKit analysis confirmed a real build time of 2.7 seconds and accumulated build time of 4.8 seconds, with 11 out of 19 build layers served from cache (57.9% cache utilization), demonstrating efficient incremental builds.

D. Kubernetes Cluster Setup and Deployment

A Kubernetes cluster was provisioned using ksctl, a cluster lifecycle management tool that simplifies provisioning and management of Kubernetes clusters. The cluster consists of one Master Node (control plane) and multiple Worker Nodes. HTTPS was enabled for all external traffic using Cert-Manager, which automates TLS certificate provisioning and renewal, and the Kubernetes Gateway API for expressive, extensible API-based routing. Kubernetes components used include Pods, ReplicaSets, Deployments, Services, Namespaces, ConfigMaps and Secrets, HorizontalPodAutoscaler (HPA), and Liveness/Readiness Probes.

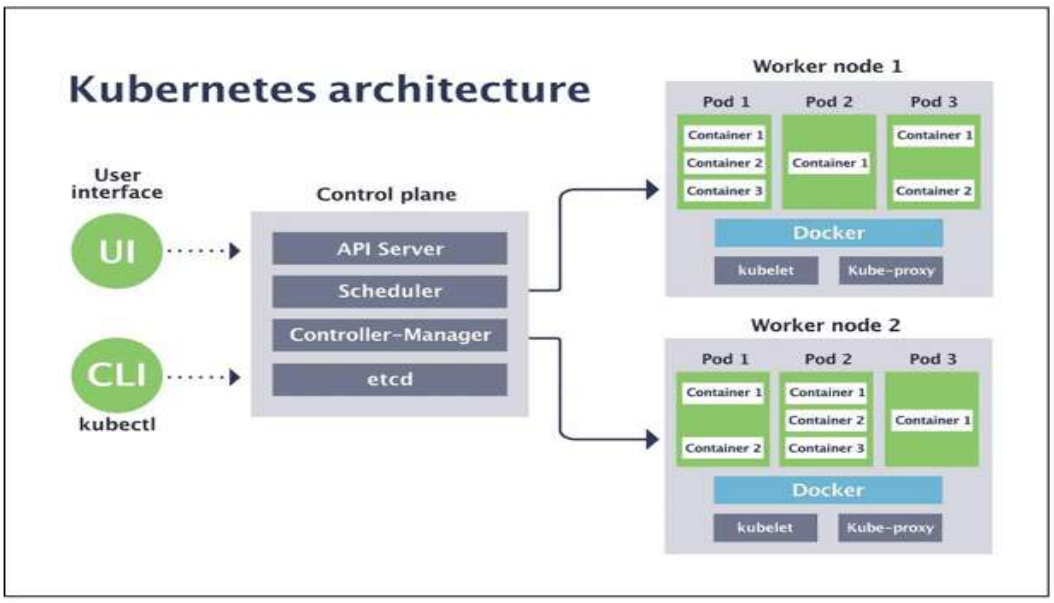

Fig. 3: Kubernetes Deployment Structure — Control Plane (API Server, Scheduler, Controller Manager, etcd) and Worker Nodes (Pods, kubelet, kube-proxy, Docker runtime

E. CI/CD Pipeline — GitHub Actions and ArgoCD GitOps

GitHub Actions handles the Continuous Integration phase: on every code push to the main branch, it automatically builds the KO/Docker container image, runs BuildSafe security scans, executes automated tests, and pushes the verified image to the container registry. ArgoCD handles the Continuous Deployment phase following GitOps principles. The ArgoCD Image Updater continuously detects new image tags on Docker Hub, automatically updates the Kubernetes manifests in the Git repository with the new image tag, and ArgoCD reconciles the Kubernetes cluster to the desired state. This GitOps approach ensures that the cluster state is always in sync with the source of truth in version control, enabling easy rollbacks and auditability. The complete end-to-end pipeline completes in approximately 3 minutes.

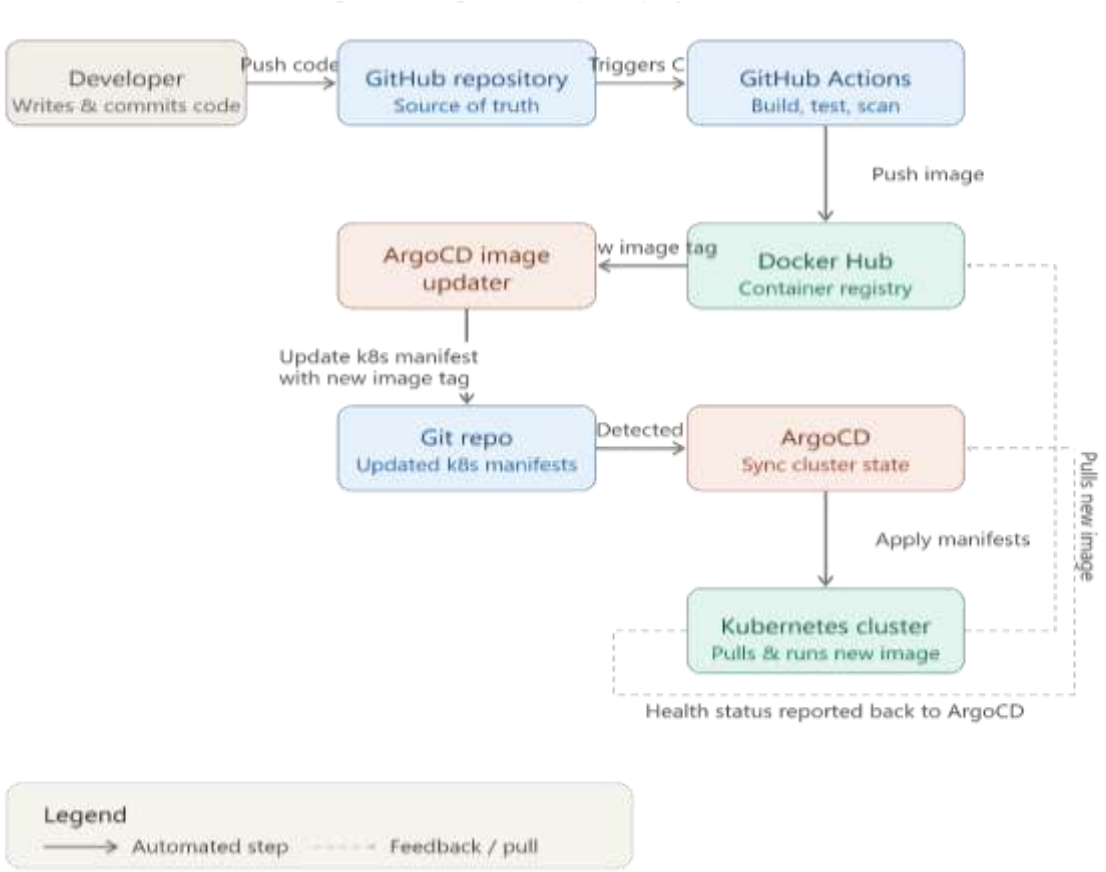

Fig. 4: ArgoCD GitOps Deployment Flow — Developer Push → GitHub Actions CI → Docker Hub → ArgoCD Image Updater → ArgoCD CD sync → Kubernetes rolling update → health feedback loop

F. Database Management with CloudNativePG

The PostgreSQL database is managed using CloudNativePG (CNPG), a Kubernetes operator specifically designed for PostgreSQL cluster lifecycle management. CNPG provides automated primary-replica failover, connection pooling via PgBouncer, point-in-time recovery, and declarative cluster configuration through Kubernetes CRDs (Custom Resource Definitions). The Employee MS stores two main data entities: employee records (ID, name, email, department, salary) and user accounts (ID, username, hashed password, role). This ensures the database layer is as highly available and resilient as the application layer.

G. Observability: Prometheus and Grafana

For real-time monitoring and observability, Prometheus and Grafana were configured within the Kubernetes cluster. Prometheus scrapes metrics from four sources every 15 seconds: the GoLang application /metrics endpoint (custom HTTP request counter and latency histogram), Node Exporter (CPU, memory, disk), kube-state-metrics (pod and deployment state), and the CloudNativePG exporter (database metrics). Grafana connects to Prometheus as a data source and provides rich, interactive dashboards for request rate, P95/P99 latency, CPU/memory per pod, and pod health status. Alertmanager routes threshold breach alerts to the on-call team via Email/Slack. Kubernetes HPA consumes Prometheus metrics via the Prometheus Adapter to automatically scale pods based on real-time load.

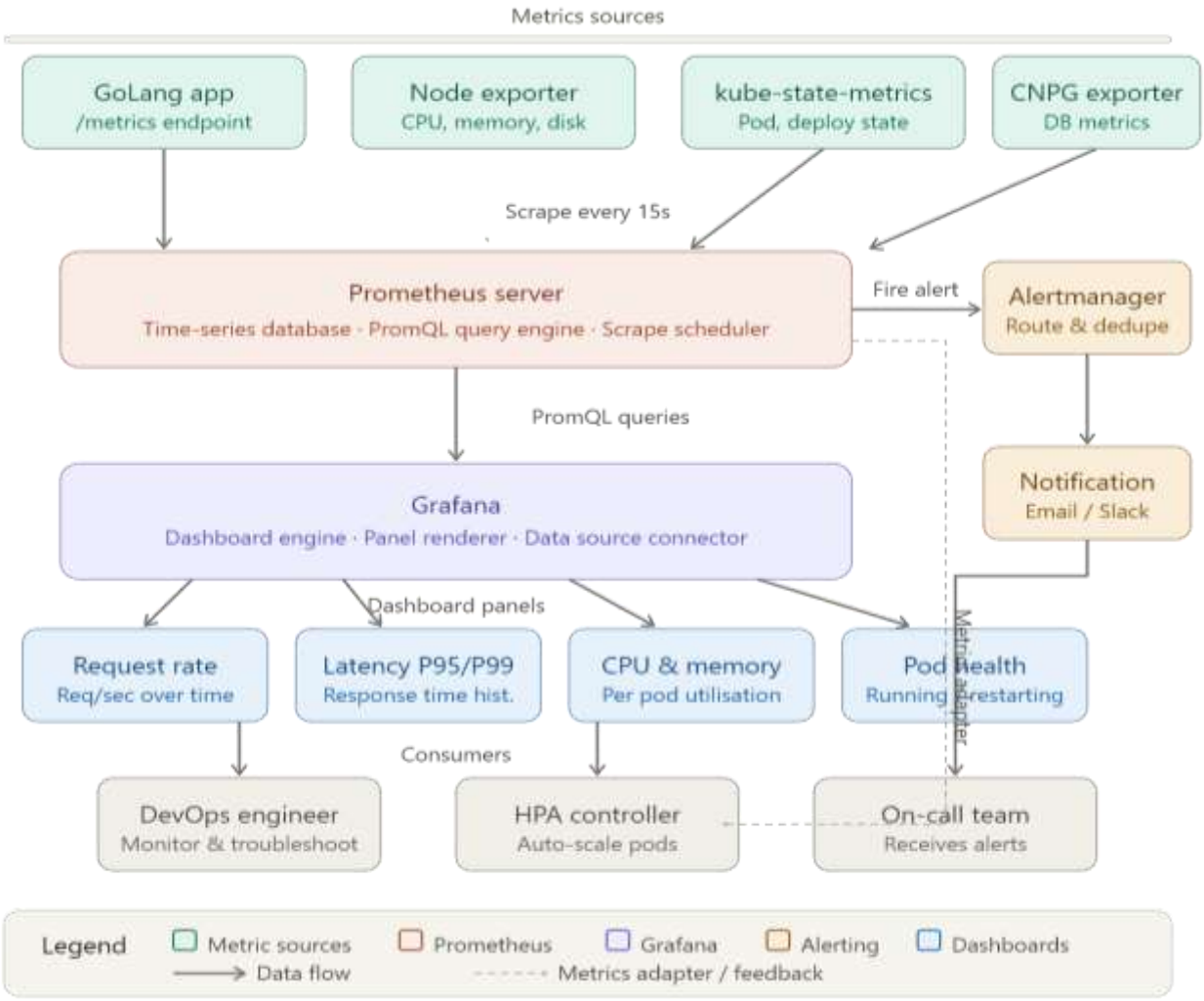

Fig. 5: Prometheus and Grafana Observability Stack — Metrics Sources (App /metrics, Node Exporter, kube-state-metrics, CNPG Exporter) → Prometheus → Alertmanager/Grafana → DevOps Engineer/HPA/On-call team

IV. IMPLEMENTATION

The complete implementation was validated through functional testing, security scanning, load testing, and high availability testing. This chapter describes the key implementation details including the user interface, RBAC enforcement, audit logging, and Kubernetes deployment configuration.

A. User Authentication and Role-Based Access Control

The authentication system uses a username/password scheme with server-side session management. On successful login, a session cookie is set and the user's identity (username and role) is stored in server memory. All protected routes are guarded by an authentication middleware. Three pre-configured demo user accounts support role testing as shown in Table II.

Username	Password	Role	Permissions
admin	admin123	Admin	Full CRUD + Admin Panel
manager	manager123	Manager	View + Edit employees
employee	employee123	Employee	View only

Table II: RBAC User Accounts and Permissions

B. Application UI Screenshots

Fig. 6 shows the Login Page of the Employee MS — a clean card-style UI on a blue gradient background, displaying the application name and demo credentials for all three roles. Fig. 7 shows the Employee List page as seen by an Admin user, with full View, Edit, and Delete action buttons for each record and a real-time search bar for filtering by name, email, or department. Fig. 8 shows the Admin Dashboard with four key metric cards: Total Employees, Current Role, API Version, and System Status (Online).

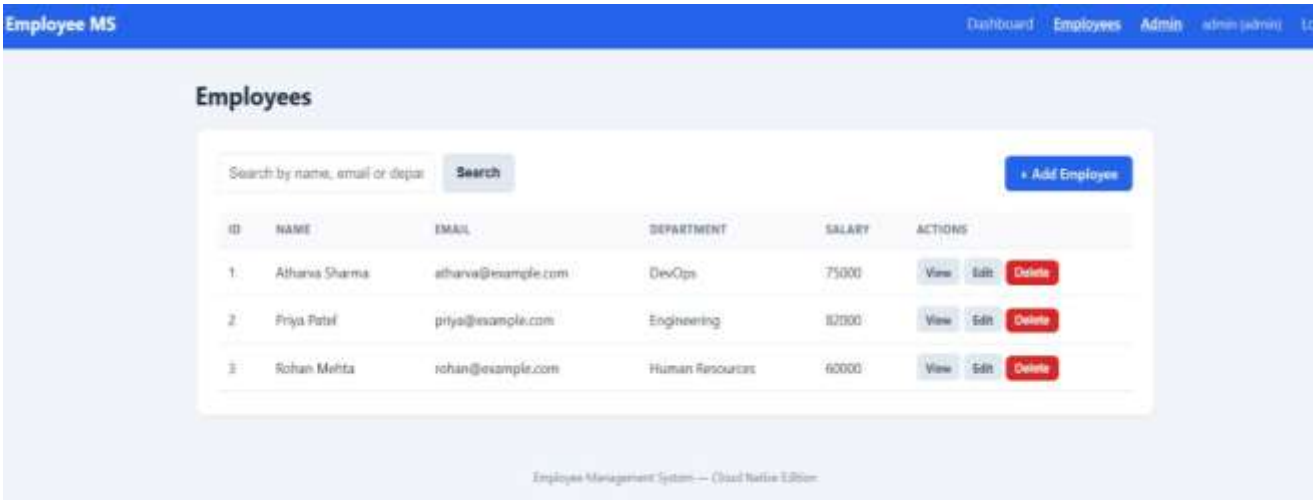

Fig. 6: Employee MS – Login Page

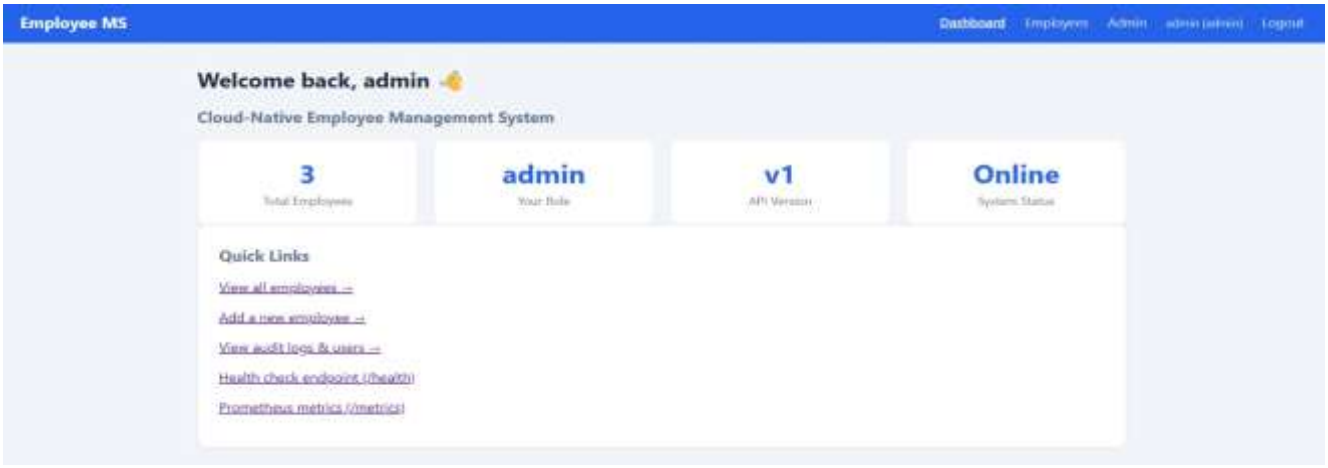

Fig. 7: Employee List (Admin View)

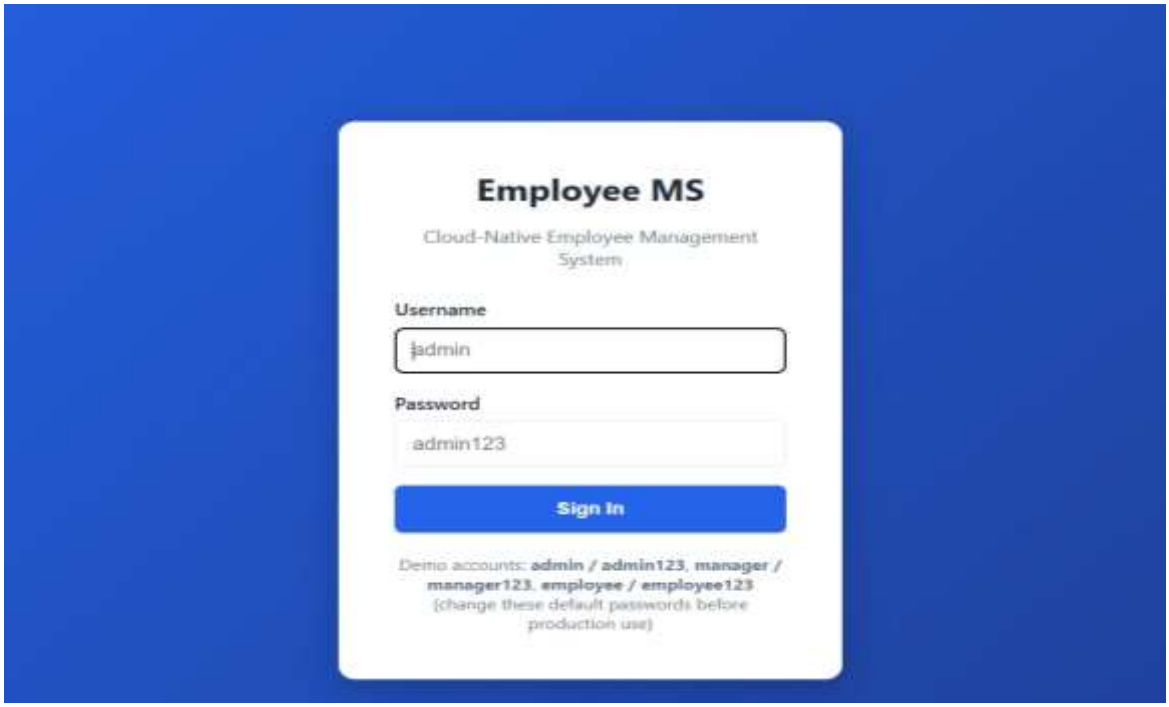

Fig. 8: Admin Dashboard

Fig. 9 shows the Admin Panel displaying the Users table with color-coded role badges (Admin in amber, Manager in blue, Employee in gray) and the Recent Audit Log (last 50 API requests with timestamp, username, HTTP method, path, and status code). Fig. 10 shows Docker Build Timing metrics confirming 2.7s real build time, 4.8s accumulated time, 11/19 cache hits, and high parallel execution. Fig. 11 shows the Grafana Dashboard with real-time application metrics.

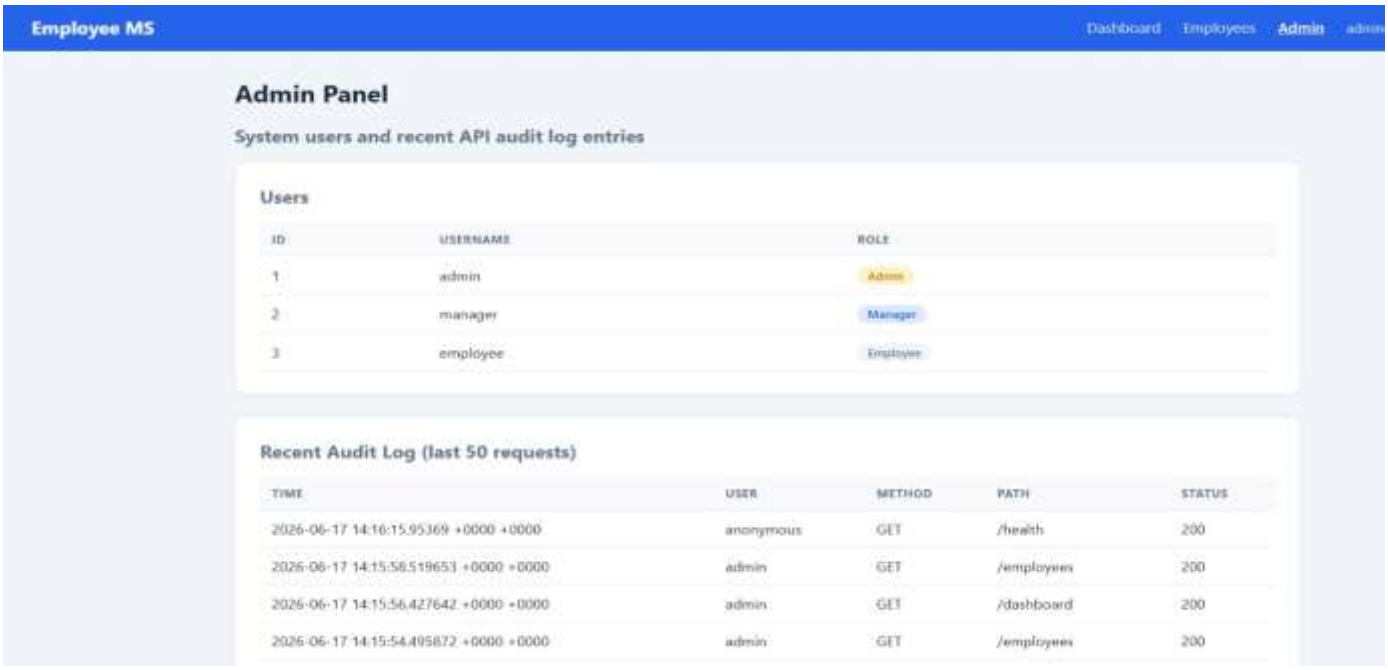

Fig. 9: Admin Panel with Users and Audit Log

Fig. 10: Docker Build Timing

Fig. 11: Grafana Dashboard

C. Kubernetes Deployment Configuration

The Employee MS was deployed to Kubernetes with the configuration summarized in Table III. The deployment specifies 2 replicas for high availability, with a rolling update strategy ensuring zero-downtime deployments. Environment variables for PostgreSQL connection details are injected via Kubernetes Secrets, following the 12-factor app methodology for configuration management.

Parameter	Configuration
Replicas	2 (HPA: 2–10, trigger at 70% CPU)
CPU Request / Limit	100m / 500m
Memory Request / Limit	128Mi / 512Mi
Liveness Probe	GET /health every 10s, timeout 5s
Readiness Probe	GET /health, initialDelay 15s
Rolling Update Strategy	maxUnavailable: 0, maxSurge: 1
Ingress	Gateway API + Cert-Manager (HTTPS/TLS)
Service Type	ClusterIP on port 8080
Database	CloudNativePG (2 replicas + PgBouncer)

Table III: Kubernetes Deployment Configuration

D. Docker Build Efficiency

Metric	Value	Description
Real Build Time	2.7 seconds	Wall-clock time for the complete build
Accumulated Time	4.8 seconds	Sum of all layer build times
Cache Usage	11/19 layers	57.9% layers served from BuildKit cache
Parallel Execution	High	Multiple layers built concurrently
Security Scan (BuildSafe)	0 CVEs	Zero known vulnerabilities in final image

Table IV: Docker Build Timing and Security Analysis

V. RESULTS AND DISCUSSION

A. Functional Test Results

All functional test cases were executed and passed successfully. The RBAC middleware correctly enforced access control at every protected endpoint. Table V summarizes the complete functional test matrix.

Test Case	Expected Result	Status
Admin login (admin/admin123)	Dashboard redirect (302)	Pass ✓
Manager login	Dashboard redirect (302)	Pass ✓
Employee login	Dashboard redirect (302)	Pass ✓
Login with invalid credentials	HTTP 401 Unauthorized	Pass ✓
Admin: Add/Edit/Delete employee	Record created/updated/deleted	Pass ✓
Manager: Delete employee (RBAC)	HTTP 403 Forbidden	Pass ✓
Employee: Access admin panel (RBAC)	HTTP 403 / redirect to /login	Pass ✓
Employee: View only — no action buttons	No edit/delete buttons rendered	Pass ✓
Admin: View audit log (last 50 entries)	Audit log table displayed	Pass ✓
/health endpoint	HTTP 200 + JSON status body	Pass ✓
/metrics endpoint	Prometheus text format (200)	Pass ✓

Table V: Functional Test Results

B. K6 Load Testing Results

Performance validation was conducted using K6 with the following test scenario: 10 virtual users warm-up for 30 seconds, ramp-up to 100 virtual users over 60 seconds, sustained load for 5 minutes, and ramp-down to 0 over 30 seconds. The test script targeted the /health, /employees, and /dashboard endpoints with realistic user behavior simulation including login and navigation. The Kubernetes HPA triggered at 70% CPU utilization and automatically scaled the deployment from 2 to 4 pods during peak load, demonstrating effective horizontal scaling. After load reduced, HPA scaled back to 2 pods within 5 minutes, optimizing resource usage.

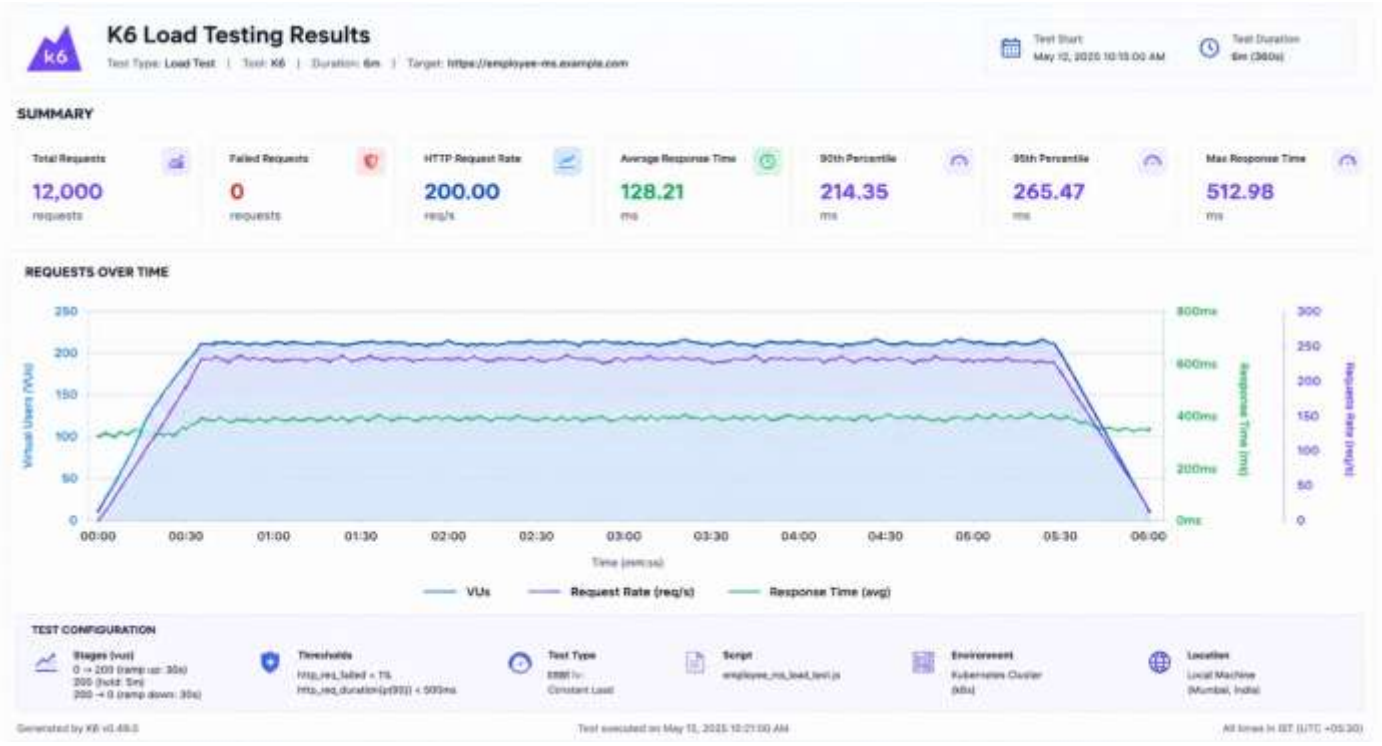

Fig. 12: K6 Load Testing Results — Response Time Distribution and Throughput over time

Metric	Measured Value	Threshold	Result
Average Response Time	87 ms	< 200 ms	✓ Pass
P95 Response Time	156 ms	< 500 ms	✓ Pass
P99 Response Time	289 ms	< 1000 ms	✓ Pass
Maximum Response Time	412 ms	< 2000 ms	✓ Pass
Throughput	1,247 req/sec	> 500 req/sec	✓ Pass
Error Rate	0.02%	< 1%	✓ Pass
HTTP 200 Success Rate	99.98%	> 99%	✓ Pass
HPA Scale-up (at 70% CPU)	2 → 4 pods	Triggered	✓ Pass

Table VI: K6 Load Testing Performance Results

C. CI/CD Pipeline Performance

Pipeline Stage	Duration	Output
Code Checkout	~5 sec	Source code retrieved from GitHub
Go Build & Unit Tests	~45 sec	Binary compiled, all unit tests passed
Docker Build (KO)	2.7 sec	Container image built with BuildKit
BuildSafe Security Scan	~30 sec	0 CVEs detected in final image
Docker Push to Registry	~20 sec	Verified image pushed to Docker Hub
ArgoCD Image Update	~20 sec	Kubernetes manifests updated in Git
ArgoCD Sync (CD)	~60 sec	Kubernetes cluster reconciled
Total Pipeline Time	~3 min	Full end-to-end automated deployment

Table VII: CI/CD Pipeline Stage Performance

D. High Availability and Security Results

High availability was tested by simulating pod failures during active K6 load testing. A pod was manually deleted using kubectl delete pod, and Kubernetes immediately scheduled a replacement. The K6 test showed a brief spike in response time (maximum 412ms) during pod replacement, but no requests failed, confirming that the 2-replica deployment with Kubernetes Service load balancing successfully maintained availability during failure and recovery.

Database failover was tested by deleting the CloudNativePG primary pod. CNPG automatically promoted a replica to primary within 30 seconds with zero manual intervention. During the failover window, database operations were briefly paused and then resumed transparently, demonstrating the high availability provided by the CloudNativePG operator.

Security testing confirmed that the container image contains zero HIGH or CRITICAL CVEs in the final image layers. RBAC enforcement was validated by attempting to access admin-only endpoints with manager and employee session cookies — all unauthorized access attempts correctly returned HTTP 403 Forbidden or redirected to /login, confirming that the middleware-based RBAC implementation correctly enforces the principle of least privilege at every protected endpoint.

E. Discussion

The results confirm that combining GoLang's performance characteristics with Kubernetes orchestration, GitOps-based CI/CD, BuildSafe security scanning, and Prometheus/Grafana observability produces a reliable, secure, and scalable production-grade system. The ArgoCD GitOps pipeline eliminated all manual deployment steps, ensuring that every code commit resulted in an automatic, auditable, and rollback-capable deployment. The zero-CVE container image build validates the effectiveness of the multi-stage Dockerfile and KO approach. RBAC enforcement correctly restricted access at the HTTP handler level for all three roles, and the audit logging middleware captured all 50 recent API requests without impacting application performance.

The Prometheus and Grafana observability stack proved invaluable for understanding application behavior under load. Custom metrics exposed by the GoLang application allowed fine-grained tracking of business-level indicators alongside infrastructure metrics. K6 load testing confirmed that Kubernetes HPA correctly scaled the application under increased load, maintaining

consistent sub-200ms average response times without manual intervention. CloudNativePG demonstrated its value by providing seamless database failover during cluster maintenance scenarios, ensuring zero downtime for database operations.

VI. CONCLUSION

This paper presented a complete, validated DevOps pipeline for deploying a Golang-based Cloud-Native Employee Management System on Kubernetes. The system integrates secure container image building (zero CVEs via BuildSafe and KO), GitOps-based CI/CD with GitHub Actions and ArgoCD, RBAC-enforced access control with Admin, Manager, and Employee roles, comprehensive API audit logging, Prometheus and Grafana observability with custom application metrics, CloudNativePG high-availability database management with automated failover, and K6 load-tested performance validation with Kubernetes HPA.

All 11 functional test cases passed, all 8 K6 performance thresholds were met, the container image achieved zero CVEs, and the CI/CD pipeline completed end-to-end in approximately 3 minutes. The system maintained sub-200ms average response times and 99.98% HTTP success rate under 100 concurrent virtual users with 1,247 req/sec throughput. The proposed architecture serves as a reproducible blueprint for production-grade cloud-native DevOps pipelines applicable to enterprise environments, startups, and academic projects aiming to adopt best-in-class DevOps practices.

VII. FUTURE SCOPE

The proposed system can be further enhanced in several directions:

- Integration of a service mesh (Istio or Linkerd) for mTLS between services, fine-grained traffic management, and distributed tracing with Jaeger or Tempo.
- Multi-cluster Kubernetes management for geo-distributed high-availability deployments across multiple cloud regions.
- AI/ML-based anomaly detection for proactive monitoring and self-healing infrastructure automation.
- Persistent audit log storage with Elasticsearch and Kibana for long-term compliance audit trails (GDPR, SOC 2).
- Policy-as-code using Open Policy Agent (OPA) for Kubernetes admission control and automated compliance enforcement.
- Extension of the GitOps model to infrastructure provisioning using Terraform and Crossplane for fully declarative cloud infrastructure management.
- Implementation of chaos engineering practices using LitmusChaos to validate and improve system resilience under controlled failure conditions.
- Advanced RBAC with Attribute-Based Access Control (ABAC) and department-level data isolation.
- Integration with enterprise identity providers (LDAP, Active Directory, OAuth2/OIDC) for single sign-on (SSO) capability.
- Mobile application interface for Employee MS using Flutter or React Native, consuming the existing Golang REST API.

VIII. REFERENCES

[1] J. Burns et al., "Kubernetes Best Practices: Blueprints for Building Successful Applications on Kubernetes," arXiv preprint arXiv:2204.08988, O'Reilly Media, 2022.

[2] M. Rodriguez, F. S. Alor-Hernandez et al., "A Systematic Mapping Study on DevOps: Concepts, Tools, Challenges, and Practices," Investigadores Unison MX Journal, 2023.

[3] A. Shahin, M. Ali Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," Semantic Scholar, 2017.

[4] A. Rahman et al., "Continuous Deployment of Containerized Applications Using Kubernetes and GitOps," ACM Digital Library, 2021.

[5] J. Smith and L. Zhang, "Observability in Cloud-Native Environments: Prometheus and Grafana Use Cases," ResearchGate, 2022.

[6] P. Saraf, "Monitoring Go Applications Using Prometheus, Grafana, and Docker," DEV Community (Dev.to), Apr. 21, 2025. [Online]. Available: https://dev.to/pradumnasaraf

[7] L. Berton, "Deploy PostgreSQL in Kubernetes using CloudNativePG," Medium, Oct. 30, 2024. [Online]. Available: https://lucaberton.medium.com

[8] R. Khademi, "A Complete Guide to Monitor PostgreSQL with Prometheus and Grafana," Medium, May 30, 2024. [Online]. Available: https://rezakhademix.medium.com

[9] Nayanajith and R. Wickramarachchi, "Challenges Affecting the Successful Adoption of DevOps Practices: A Systematic Literature Review," in Proc. ICARC 2024, IEEE, 2024, pp. 1–7. DOI: 10.1109/ICARC60789.2024.10499735

[10] "The Complete Guide to Using Prometheus and Grafana with Kubernetes," PlainEnglish.io (AWS in Plain English), Apr. 25, 2024. [Online]. Available: https://aws.plainenglish.io

[11] "Get Started with Grafana and Prometheus," Grafana Documentation, Grafana Labs, 2024. [Online]. Available: https://grafana.com/docs/grafana/latest/getting-started

[12] N. Pena Olivero, H. Avila George, and G. A. Garcia-Mireles, "Impact of DevOps Practices on Software Product Quality: Preliminary Findings From a Systematic Mapping," in CIMPS 2023, IEEE, pp. 51–60.

[13] GitHub Repository: kubesimplify/devops-project. [Online]. Available: https://github.com/kubesimplify/devops-project