National College of Ireland

Cloud-Native Application Development Using DevOps Tools

MSc Research Project
Cloud Computing

Sanal Sabu
Student ID: 23382651

School of Computing
National College of Ireland

Supervisor: Sean Heeney

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Sanal Sabu
Student ID:	23382651
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Sean Heeney
Submission Due Date:	11/12/2025
Project Title:	Cloud-Native Application Development Using DevOps Tools
Word Count:	7212
Page Count:	20

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Sanal Sabu
Date:	26th January 2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Cloud-Native Application Development Using DevOps Tools

Sanal Sabu
23382651

Abstract

As organizations continue to migrate to cloud-native architectures, there is an increasing need for ways to effectively maintain security, compliance, and operational stability while also enabling a rapid rate of development. This paper describes a comprehensive methodology for achieving such a balance through the use of DevOps practices in conjunction with automated Policy-as-Code (PaC) processes. Through this research, the author has demonstrated an approach for designing, implementing, and evaluating a fully functioning environment for a cloud-native application. This includes all aspects of developing a cloud-native application; from infrastructure creation, to CI/CD (continuous integration/continuous delivery), to providing an automated mechanism for scanning code for security vulnerabilities, to providing real-time insight into the production environment. This was accomplished by developing and deploying a microservices based application on AWS using modern container and orchestration technologies, and by using Terraform for creating the infrastructure in a declarative manner, and automating the CI/CD through the use of GitHub Actions. An important finding of this research is that integrating security and policy enforcement into the development life-cycle can occur at all phases. By utilizing several different tools (such as Trivy for scanning vulnerabilities, OPA to validate both Infrastructure as Code and the various configurations of Kubernetes), this new "shift left" strategy has been implemented. The effectiveness of this combined solution has been established using a rigorous series of controlled experiments that demonstrated the successful automated identification and remediation of vulnerabilities and misconfigurations in a variety of situations. These results show that a solidly built DevOps and PaC Framework will improve the security, resilience, and agility of cloud-native applications substantially.

1 Introduction

The way that software is developed has changed dramatically as a result of cloud-native computing. This new method of software development focuses on how to build and operate scalable applications within a modern, flexible environment with the use of microservices, containers, and dynamic orchestration (Deng et al., 2024). In addition, with new technologies such as service meshes, immutable infrastructure, and containers, businesses have greater agility, resilience, and scalability (Oyeniran et al., 2024). However, managing this complexity is a challenge for many organisations; it requires managing the underlying infrastructure, security and ensuring operational visibility.

Due to the challenges created by this new model of software development, DevOps emerged as a cultural and engineering practice to help manage that complexity. DevOps encourages collaboration and automation, and by eliminating the silos between Development and Operations teams, allows organisations to deliver software in a faster and more reliable manner (Poojary et al., 2025). One of the key concepts of DevOps is the Continuous Integration and Continuous Deployment (CI/CD) pipeline, which automates all aspects of the Build, Test, Deploy processes. CI/CD provides the foundation for rapid software delivery, but if organisations don't incorporate security throughout that automated process, they run the risk of rapidly deploying vulnerabilities or misconfigurations into their production systems. As a result, DevSecOps has emerged as the next evolution of DevOps, incorporating security practices in the automated CI/CD pipeline, often referred to as "shifting left" (Somi, 2022).

Policy-as-Code (PaC) is one of the necessary enabling factors for implementing DevSecOps in a cloud-native framework. Using PaC means will create, manage and enforce Security and Operational Policies using code just like any other technical part of a system. Policy enforcement can subsequently be done automatically, with consistency and an audit trail, throughout the Application Lifecycle from the Infrastructure provisioned to runtime.Theodoropoulos et.al (2023) has set out to tackle this Research gap around establishing a secure, compliant and efficient cloud-native software development ecosystem. The main question driving this Research project is: **"To what extent does integrating a Policy-as-Code framework into CI/CD pipelines improve automated security compliance and vulnerability management in cloud-native applications?"**

1.1 Research Objective

To assist in answering this central question, the Research has identified these objectives:

- To design and build a Micro Services based Containerised Application that can be deployed in a Cloud Native Environment;
- To create an infrastructure as code based provisioned Cloud Environment;
- To create sufficiently strong Continuous Integration and Continuous Deployment (CI/CD) Pipelines that can automate Building, Scanning For Security Issues and Deploying Applications into a variety of production-like environments.
- Through the integration and enforcement of security and operational policies into the Critical Stages of a CI/CD Pipeline, implement these practices using Policy-as-Code.
- The establishment of an Observability Stack will provide real-time visibility into application performance and Infrastructure Health (including Policy Compliance).
- And finally, empirically assess and validate the findings by performing a series of controlled experiments using integrated security controls.

1.2 Report Structure

This report presents the first official implementation and validation of an end-to-end system that showcases the synergy between these modern practices. While much of the

current literature on this area focuses on the individual aspects of each of these methodologies, this paper illustrates a coherent framework in which all aspects are integrated together and validated. It offers a roadmap for the creation of secure Cloud-Native systems.

To facilitate reading this report and for referencing in future work, the structure of this report can be outlined as;

Section 2 reviews the current state of the literature on these concepts.

Section 3 states the methodology used to conduct the research.

Section 4 describes the system architecture and design.

Section 5 includes implementation details, tools, and related resources.

Section 6 illustrates the findings via a series of controlled experiments.

Section 7 concludes with recommendations for future research on building secure Cloud-Native systems.

2 Related Work

This section places the project into the context of the current body of academic work by conducting a critical review of cloud-native architectures, DevOps, DevSecOps, Policy-as-Code and observability-related research.

2.1 Architecture of Cloud-Native Developing

The movement toward cloud-native architecture has been a major focus of research in recent years. Deng et al. (2024) have published a full review of cloud-native computing using a services perspective, illustrating the basic elements of cloud-native computing; including microservices, containers and serverless computing. Oyeniran et al. (2024) also reviewed cloud-native technologies, showing how cloud-native technologies increase scalability and resilience to software development. The article by Kader and Ahmed (2025) examines modern design patterns and best-practices regarding full-stack development using AWS, which is in line with the technology proposed in this project. These articles provide a strong theoretical and architectural backing to what cloud-native applications should be; however, they largely fail to offer a detailed guideline for implementing the necessary security and operational tools to allow for the management of these complex systems. Similarly Chippagiri and Ravula (2021) have published a review of the best practices with regard to creating scalable and resilient web applications; however, their primary focus is on the application design itself, rather than the adjoining DevOps/Security ecosystem. The intention of this project is to extend upon the architectural principles provided in these references to implement them in a fully automated and secure lifecycle.

2.2 Combining DevOps Automation with CI/CD to Build a Scalable Work Environment

The role of DevOps in controlling and managing Cloud infrastructure as well as workflows is widely known within Computer Science. W. Poojary et al. (2025), and M. Ramesh et al. (2025), provide a complete review of how to scale ML workloads in a Cloud environment, as well as how to apply DevOps processes to it via automation within both platforms/technologies, an example of this is how to control and manage Digital Twin applications, which include both complex and large, data-driven operations/. P. Vangala has some specific examples of how to implement optimally manage Cloud Infrastructure via DevOps Process. A primary tool for implementing Infrastructure as Code (IaC) Management in DevOps Processes is Terraform. Additionally, T. Tong, (2025) has some great outcomes through using both Terraform and DevOps to implement Disaster Recovery methods in Multi-Cloud environments. This demonstrates the need for this type of software and DevOps Process automation to achieve resiliency within operation. All of the above research supports the fact that DevOps Methodologies and IaC tools are critical for the Success of Modern Cloud Operations, but the integration of Automation policies during these automated workflows is usually an afterthought. This research will contribute to the existing body of work in this area by requiring that all changes to infrastructure are validated against a Policy requirement prior to being deployed in the CI/CD pipeline.

2.3 Integrated Security and DevSecOps Security

It is combined with DevOps to create DevSecOps and is one of the most necessary integrations in cloud-native environments. Somi's (2022) DevSecOps Review discusses the best practices for integrating security into the CI/CD pipeline using the "shift-left" concept. This involves integrating security checks into the earliest stages of the development life cycle. Theodoropoulos and others (2023) provide an extensive survey of cloud-native services, addressing the potential security risks from the container runtime all the way to the application layer, and presents possible solutions to mitigate those risks across the stack. Arif, Jo, and Park (2025) conducted an extensive survey of privacy-enhancing and trust-centric measures available to mitigate security risks against cyberattacks. In the literature, a key issue identified is the friction that security tools can create in an agile pipeline. According to Syal (2022), one way to solve this friction is through the use of automated testing frameworks, which enable velocity to be maintained in an agile pipeline. I addressed this issue by implementing fast, automated, and policy-driven security tools (e.g., Trivy and OPA) that allow developers to obtain immediate feedback without causing much of a delay.

2.4 Policy-as-Code

Policy-as-Code is a powerful approach to DevSecOps and is becoming the focus of an increasing amount of literature, especially in terms of governance and security automation. The authors (Wei, Madhavji and Steinbacher, 2025) have mapped cloud-native practices and tools that produce the desired characteristics of a system with one of the key components being the use of automated enforcement of policies. One such tool is the Open Policy Agent (OPA), which provides a unified, declarative way to implement policy

across multiple domains (e.g., Kubernetes and Terraform). Historically, multiple tools would have used different tooling-specific checks for configurations, which were unmanageable and unsustainable. This project's method of using a single policy engine (OPA) and different Rego policies for various target contexts, shows a modern, maintainable option. Experiments have confirmed the value of this project through its demonstration of how Policy-as-Code can provide practical evidence supporting its theoretical advantages.

2.5 Observability in Cloud-Native Environments

As explained by Kosińska et al. (2023), observability is a critical capability of cloud-native applications. Using Cloud-Native applications' distributions and dynamics, traditional monitoring approaches are not sufficient. Observability consists of three main areas called pillars of observability: metrics, logs, and. Observability Challenges and Solutions: Sakinala (2025), Ganesan (2022) have provided detailed examples of both challenges and solutions in relation to Monitoring and Observability and recommend various tools to support Monitoring and Observability. The recommended tools include platforms like Prometheus and Grafana. The implementation of the approaches in this document is performed by collecting both standard application metrics and infrastructure metrics and creating a custom Policy Metrics Exporter that connects the observability with policy enforcement. The Policy Metrics Exporter provides the visual representation of Compliance Trends, Security Vulnerabilities, and Blocked Deployments directly within Grafana, thereby incorporating Security and Compliance Monitoring within Observability Monitoring.

To sum up, the current literature provides a comprehensive basis for all aspects of the project. The academic work has been demonstrated in a holistic, practical and validated way from IaC and CI/CD to PaC and observability into one unified structure. this research provides the missing link through the provision of an in-depth blueprint and empirical proof of the effectiveness of such a system.

Table 1: Critical Review of Related Work

Reference	**Primary Theme**	**Key Contribution / Finding**	**Limitation or Gap Addressed by This Project**	**Relevance to This Project**
Deng et al. (2024)	Cloud-Native Architecture	Provides a comprehensive survey defining the core principles and technologies of cloud-native computing (microservices, containers).	Primarily a theoretical survey. Lacks a practical, end-to-end implementation that integrates automated security and policy enforcement.	Provides the foundational architectural principles upon which the project's application and infrastructure are based.
Oyeniran et al. (2024)	Cloud-Native Architecture	Reviews how cloud-native technologies are leveraged to achieve system scalability and resilience.	Focuses on architectural benefits (scalability, resilience) with less emphasis on the automated security governance required to manage these systems safely.	Justifies the architectural choices aimed at building a resilient system and highlights the need for robust automation.
Tong (2025)	DevOps & Infrastructure as Code (IaC)	Demonstrates the use of Terraform for a specific DevOps task (disaster recovery), showcasing the power of IaC for resilience.	Focuses on a single use case (DR) and does not integrate proactive security validation (like Policy-as-Code) directly into the Terraform workflow.	Validates the choice of Terraform for managing cloud infrastructure and provides a use case for IaC in a DevOps context.
Somi (2022)	DevSecOps	Offers a comprehensive review of DevSecOps practices, strongly advocating for "shifting security left" within the CI/CD pipeline.	As a review paper, it describes what should be done but does not provide a concrete, implemented, and empirically validated system demonstrating how.	Provides the core theoretical justification for the project's security philosophy and the design of the secure CI pipeline.
Theodoropoulos et al. (2023)	DevSecOps & Policy-as-Code	Surveys the security landscape of cloud-native services and identifies automated policy enforcement as a key mitigation strategy.	A high-level survey that does not detail the implementation of a unified policy engine (like OPA) across different domains (IaC and Kubernetes).	Supports the project's decision to use Policy-as-Code as a central mechanism for enforcing security controls automatically.

3 Methodology

The research methodology for this project is based on two principles, those of Design Science and Constructive Research. The methodology selected for this project was appropriate because the primary goal was to design, build, and evaluate an artifact (an integrated DevOps and Policy-as-Code ecosystem) to solve a real-world problem. The methodology included several distinct phases, which were repeated until the desired outcomes were achieved.

Literature Review and Problem Formulation: A comprehensive review of the literature on cloud-native development, DevOps, DevSecOps, and Policy-as-Code. Through this phase, the best practices, state-of-the-art tools, and gaps in knowledge were discovered that provided the basis for the development of the central research question and research objectives.

System Design: Following completion of the literature review, a complete architecture for the entire system was designed. This included designing the Application Architecture (Microservices), Cloud Infrastructure (AWS), CI/CD Pipeline Workflows, Policy Enforcement Framework, and Observability Stack. In the architecture, Automation, Security, and Modularity were prioritized.

Iterative Implementation of Each Component: The architecture was constructed through iterative implementations using a variety of tools and technologies.

- The microservices application is built and containerized.
- The infrastructure on AWS has been defined using Terraform modules.
- The CI/CD pipeline has been defined using GitHub Actions.
- The security and operational policies have been defined using Rego for OPA.
- The observability stack (Prometheus, Grafana, and custom metric exporters) has been set up and deployed.

Empirical Evaluation and Validation: To properly evaluate the effectiveness of the implemented system, 4 separate and controlled experiments were designed. Each experiment focused on a particular operational or security concern and included a defined hypothesis.

Hypothesis: The policies that are defined and automatically enforced in the CI/CD pipeline will allow for the detection and prevention of deployment of non-compliant/insecure artifacts.

For each of the 4 experiments, there were two separate versions of a configuration artifact established: a "non-compliant" version that contained the violation that was being addressed in the experiment, and a "compliant" version indicating that the violation had been resolved.

The policy enforcement experiments were executed in both a local development environment as well as within the automated CI/CD pipeline to validate the policy enforcement mechanisms as they were integrated within the application.

Data Collection & Analysis: Throughout the experimentation phase, both quantitative and qualitative sources of data were collected.

- **CI/CD Pipeline Logs:** The logs provided by the GitHub Actions runs captured details on which jobs passed or failed (along with the output generated from tests, scans, and policy validations).
- **Security Scan Reports:** The JSON format reports generated from tools such as Trivy and OPA contained information relating to detected vulnerabilities and violations of policies.
- **Observability Metrics:** Data collected via Prometheus, visualized through Grafana, provided both real-time and historical insight into the performance of the application and policy compliance metrics.
- **Command-line Output:** The logs from the manual provision and deployment scripts provided evidence of the successful provisioning and operation of a system.

The raw data collected from these sources were analysed to determine whether research objectives had been met and to answer the research question. Focus areas of the analyses included the successful automation of tasks, the security gates functioning correctly, and the system's capability to prevent insecure configurations from being deployed.

4 Design Specification

This section details the architecture and design of the integrated system, covering the application, infrastructure, CI/CD processes, policy framework, and observability stack.

Table 2: Key Technologies Used in the System

Domain	**Technology / Tool**	**Purpose**
Cloud Provider	Amazon Web Services (AWS)	Hosting infrastructure
Application Framework	Python (Flask)	Microservices development
Containerization	Docker	Packaging applications
Orchestration	Amazon EKS (Kubernetes)	Container management and orchestration
Infrastructure as Code	Terraform	Provisioning and managing AWS resources
CI/CD	GitHub Actions	Automating build, test, and deployment pipelines
Policy as Code	Open Policy Agent (OPA)	Enforcing policies on Terraform and Kubernetes
Vulnerability Scanning	Trivy	Scanning container images for CVEs
Configuration Management	Helm	Packaging and deploying Kubernetes applications
Observability	Prometheus, Grafana	Metrics collection and visualization

4.1 Application Architecture

The application is structured based on a simple yet realistic microservice-based way of doing things that allows for both stateless functionality and scaling.

API Service: This is a RESTful API created using Python's Flask framework. It provides endpoints for performing basic Create, Read, Update and Delete type operations on data items. It also uses the "prometheus_client" library to track custom metrics about the application, such as the amount of time it took to complete a request or how many errors occurred.

Worker Service: This is a background task/service built using Python as well. It provides a way for users to offload work that can be performed asynchronously. This type of work can include things like cleaning up old data periodically, or processing jobs from a queue of jobs that need to be completed. For this implementation, it will also connect to the database at intervals to perform maintenance tasks on the database.

Database: A PostgreSQL database that is hosted on AWS RDS is used as the persistent data store for both microservices. This gives the benefit of decoupling the logical application layers from the data layers, allowing both layers to be scaled and managed independently.

4.2 Infrastructure Architecture

The design of cloud infrastructure aims to satisfy Security, Scalability and Resilience, with Terraform embodying the automating and provisioning of all components of an entire stack.

Networking: A custom-created AWS Virtual Private Cloud (VPC) acts as a logically isolated network. The VPC consists of multiple Availability Zones (AZ), providing both public and private subnets, and thus enabling high availability. Public Subnets have placed NAT Gateways to enable resources residing within the Private Subnets (e.g., EKS Worker Nodes) to communicate over the internet.

Compute and Orchestration: Amazon Elastic Kubernetes Service (EKS) Clusters are the foundation of the Compute Infrastructure. EKS Worker Nodes are implemented within the Private Subnets, isolating them from direct exposure to the internet. The Kubernetes Control Plane is managed by EKS, providing a highly resilient and scalable Orchestration Layer.

Container Registry: The Amazon Elastic Container Registry (ECR) is used to manage, store and deploy images of Docker containers for api-service and worker-service.

Data Storage: The underlying database for the project is an Amazon RDS instance running on PostgreSQL. This instance is deployed on Private Subnets and configured using a security group that only allows inbound connections from the security group associated with the EKS cluster, which ensures that with strict network segmentation.

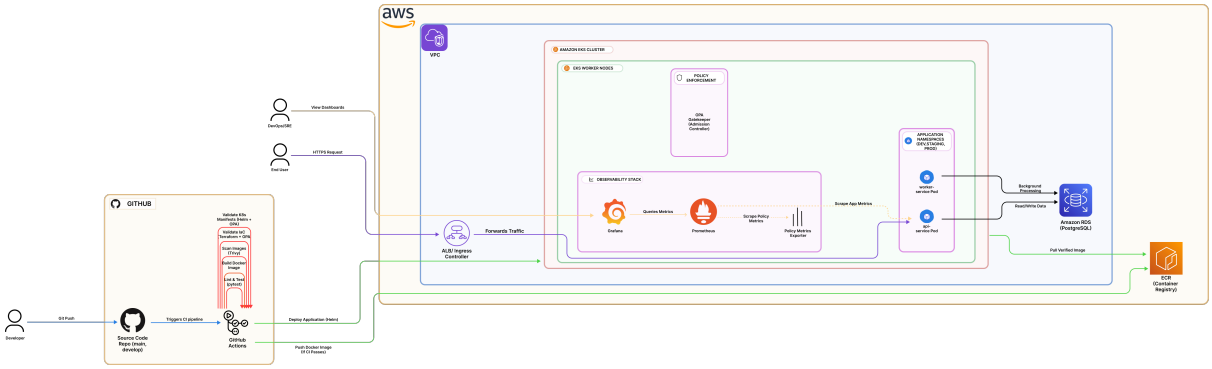

Figure 1: System Architecture

4.3 CI/CD Pipeline Design

Two separate pipelines (GitHub Actions) have been defined to automate the application development lifecycle.

Continuous Integration (CI) Pipeline: The CI pipeline serves as a complete Quality & Security Gate, in that it will run for every code change made to the codebase.

- **Lint & Test:** The first thing that the pipeline does is check out the source code and install its dependencies, then run Linter (Flake8) and Unit Tests (Pytest). Once complete, Code Coverage Reports are generated and uploaded to the CI pipeline.
- **Build & Scan:** After Linting and Testing, the pipeline will build Docker Images for the Microservices. Following that step, the Trivy Vulnerability Scanner will be run against the Docker Images looking for High and Critical Severity Vulnerabilities. If there is a failure in the scan, the pipeline cannot advance.
- **Validate Infrastructure as Code (IAC):** The pipeline will validate the Terraform Infrastructure as Code (IAC) by initializing and validating its build artefacts (Terraform Code). Once validated, a Terraform Plan will be created and converted to JSON, and will then be evaluated against Security Policies and Cost Policies defined using Rego via OPA.
- **Validate Kubernetes Configuration:** The pipeline will render the Helm Charts into Kubernetes manifests. After the manifests have been created, they will be evaluated against Kubernetes Security Best Practices using OPA.
- **Create SBOM:** Syft generates a Software Bill of Materials (SBOM) for each Container Image.

Continuous Delivery (CD): The CD Pipeline is the system responsible for placing the developed application into an EKS Cluster.

- **Build & Push:** Once a valid trigger occurs (e.g. pushing code to the 'main' Branch or adding a New Version Tag), Docker Images are built, assigned Unique Identifying Tags (e.g. Commit SHA or Version) and uploaded to the ECR.
- **Dev Deployment:** Once the code has been pushed up to the 'main' Branch, the latest version is automatically deployed to the Development Namespace in EKS via Helm.

Staging Deployment: A new version tag triggers the Staging Deployment. A Smoke Test is executed in this stage to assess the health of the deployment.

Production Deployment: After the Successful Staging Deployment, the Production Deployment occurs. Canary Deployment is used; the new version is deployed to a single instance first, a delay is inserted for Reviewing the Health of the Instance, and then the deployment of all instances is commenced.

4.4 Policy as Code Framework Design

The Policy as Code framework has been developed to create a centralized and automated method for governing the infrastructure and application configurations. The OPA serves as the central governing engine for the Policy as Code framework.

Kubernetes Policies: A collection of Rego policies (security.rego, bestpractices.rego) outline the security and operational standards for workloads running in Kubernetes. Examples of rules enforced by these policies include:

- Containers cannot execute as the root user
- Containers cannot execute in privileged mode
- Resource limits (e.g. CPU, memory) must be defined
- Liveness and readiness probes must be configured to ensure correct functioning
- Images using the :latest tag are not to be used

Policies are enforced in two ways: proactively during the CI pipeline and reactively during runtime using the OPA Gatekeeper admission controller installed within the Kubernetes cluster.

Terraform Policy Management: At the infrastructure level, Terraform policy management involves the validation of Terraform plans through 'Rego Policies', such as the 'cost' and 'security'. These two policies have rules in place that ensure,

- RDS databases and EBS volumes are encrypted.
- RDS databases are private and cannot be accessed publicly.
- Security groups do not allow unrestricted inbound access on critical ports (0.0.0.0/0).
- Resources must have appropriate tags for the purpose of cost allocation and governance.
- Use of expensive instance types in non-production environments is discouraged.

4.5 Designing an Observability Stack

The observability stack is designed to provide detailed real-time visibility into the status and performance of the complete system.

Metrics Collection: The kube-prometheus-stack Helm chart is used to deploy Prometheus into the cluster, allowing the collection of metrics from various data sources, including:

- **Application Metrics:** Application metrics are custom metrics exposed via the api-service (e.g., requests made, time to process requests).
- **Infrastructure Metrics:** Infrastructure metric data is provided by Kube-system component level metrics (e.g., CPU/memory usage for nodes and pods). These infrastructure metrics are collected by the node-exporter and/or kube-state-metrics.
- **Policy Compliance Metrics:** The policy-compliance-metrics-exporter generates custom metrics that quantify the number of policy violations, number of blocked deployments, and number of vulnerabilities.

Visualization and Alerting: Grafana is used to visualize the metrics collected by Prometheus and create two custom dashboards:

- **Application Dashboard:** The Application Dashboard visualizes critical performance indicators (KPI's) related to the api-service, such as request rate, error rate, response time percentiles, and resource consumption.
- **Policy-as-Code Compliance Dashboard:** The Policy-as-Code Compliance Dashboard visualizes the security and compliance metrics related to the system's security posture in real-time. The Policy-as-Code Compliance Dashboard contains panels that show the number of policy violations over time, the number of critical vulnerabilities, and the number of blocked deployments.

5 Implementation

In this area, various instruments, dialects, and actual setup information that were utilized to build the proposed solution will be discussed.

5.1 Terraform for Infrastructure Provisioning

The total AWS infrastructure was built with Terraform, with reusable components created for all aspects of AWS (with common modules for VPCs, EKS clusters, RDS instances, and ECRs) so that they would be reusable and consistent with one another and throughout the system. All assets are linked together in a root module so that the variables and outputs can be passed between all different components of the overall construction. A remote backend using an S3 bucket was set up for state storage, and a DynamoDB table was set up for state locking so that it would allow individuals working together to build AWS infrastructure via Terraform to share the Terraform state's locking and unqualified state stored in the S3 bucket to avoid corrupted or missing states. Different environments (development, staging, production) were configured with `.tfvars` files with specific variables for each environment; therefore, changing the variables for each environment was straightforward. The logs from executing commands using the command line verify that the Terraform code was applied successively so that the dev environment was created, including the VPC, EKS, and RDS instances for the dev environment.

5.2 Application Development and Containerisation

The api-service and worker-service were both developed with the Python 3.11 programming language, using the Flask web framework (lightweight). A key aspect of the API's

implementation was its instrumentation using the `prometheus_client` library to expose a `/metrics` endpoint for Prometheus scraping.

Both api-service and worker-service were containerised with Docker. The Dockerfiles were built using multi-stage builds, following best practices to keep the size of the final image (containerised version) down by separating build environment (build tools, build dependencies) from the final runtime (production) environment. The final stage only includes a minimal base image (`python:3.11-slim`) and a copy of only the application code and runtime dependencies that are absolutely required. Security best practices were also considered during the Dockerfile creation, i.e., the application runs as a non-root user.

5.3 GitHub Actions Automating CI/CD

The CI and CD pipelines are set up as GitHub actions that are defined within YAML formatted files, located within the `.github/workflows` document structure.

CI Pipeline (ci.yml): This action uses a matrix strategy to perform parallel execution of the jobs. Some notable actions to be included in this pipeline are: `actions/checkout`, `actions/setup-python`, `aws-actions/configure-aws-credentials` for logging into the AWS account, `docker/build-push-action` to create the image, and `codecov/codecov-action` which is being used to post the code coverage reports. In addition, a security scan through shell commands that run Trivy and OPA for vulnerability testing is implemented. The CI pipeline will be configured for "fail fast", meaning that if Trivy detects critical vulnerabilities at the vuln-check stage, the job will terminate with an exit status code of non-zero.

The CD Pipeline (cd.yml) establishes a method for orchestrating when to deploy an application based on the ability of previous deployments to finish before another can start (i.e., if need to deploy to production, must first deploy to staging). The workflow connects to AWS ECR using a login via the `aws-actions/amazon-ecr-login` action and pushes images that were built for deployment. Deploying an application happens using the `aws/setup-helm` action along with `helm upgrade --install` commands. The deployment of the application is customized through the use of environment-specific value files (`values-dev.yaml`, `values-staging.yaml`), which allow Helm to create a uniquely customized deployment in every environment. When performing a canary release on production, two `helm upgrade` commands are executed back-to-back, interspersed with a sleep command to allow time between them.

5.4 Policy Implementation to Enforce Policy

The implementation of Policy-as-Code is using OPA and its associated tools.

OPA Policies: Security and best practice rules were defined using the Rego policy language. For instance, the Kubernetes security policy contains rules prohibiting `container.securityContext.runAsNonRoot` to prevent containers from running with non-root users unless explicitly specified. The Terraform policies reference the JSON formatted representation of the Terraform plan to evaluate properties such as `resource.change.after.storage_encrypted`.

CI Integration: The `ci.yml` workflow contains the OPA installation and command line invocation through `opa eval` to evaluate the JSON format of the Terraform plan and rendered Kubernetes manifests for conformance to the Rego policy files.

Enforcement in Runtime: The OPA Gatekeeper was installed in the EKS cluster via its official Helm chart, and the ConstraintTemplate resources were created to define the new policy schemas, followed by the creation of Constraint resources to apply the policies to specific namespaces or resource types (i.e. requiring that all Deployments in the production namespace have certain labels).

5.5 Helm Charts for Microservice Deployment

The microservice deployments within Kubernetes were created using Helm Charts, which provides a templated approach to creating Kubernetes Manifests and enables developers to manage application configuration within a single structure. Each Helm Chart includes a set of Deployment, Service, ServiceAccount, and NetworkPolicy templates for Kubernetes. A `values.yaml` file is included as part of the Helm Chart to define the configuration options, including Replica Count, Image Tag, and Resource Limits. The configuration options are stored within a single `values.yaml` file, but can be separated into different values files that represent different environments (for example, `values-dev.yaml` and `values-prod.yaml`). Environment-specific values can be defined and used rather than defaulting to the `values.yaml` file. All aspects of managing the life cycle of an application within an environment can be accomplished using only the `helm upgrade` command to manage the Helm Charts.

5.6 The Implementation of the Observability Stack.

The observability stack has been deployed to the EKS cluster.

Prometheus and Grafana: The installation of Prometheus, Grafana, Alertmanager, and various exporters using the kube-prometheus stack Helm chart has provided the Kubernetes cluster with a standard monitoring system that includes both metrics collection and visualisation.

Custom Metrics Integration: The creation of a ServiceMonitor Kubernetes resource created Prometheus auto-discovery and scraping of the api-service's `/metrics` endpoint.

Policy Metrics Exporter: A custom Python script implementing a Prometheus exporter was created to generate metrics based upon the knowledge about the results of four security experiments, simulating how a tool would be able to judge the number of policy violations in the real world. Metrics were made available through a port, and a Deployment and Service were created to run the Python script within the cluster.

Grafana Dashboards: Custom dashboards were written as JSON files, which are configured to automatically provision onto Grafana when it starts up and include PromQL (Prometheus Query Language) queries, allowing users to view a tailored representation of Application's performance and compliance with security policies.

6 Evaluation

Evaluation of the security and policy enforcement capacities of the implemented system occurred via a set of four controlled experiments, with the command-line output produced by the `run-all-experiments.ps1` script and the CI pipeline logs being used as the main basis for this evaluation. The overall aim of the evaluation was to determine whether

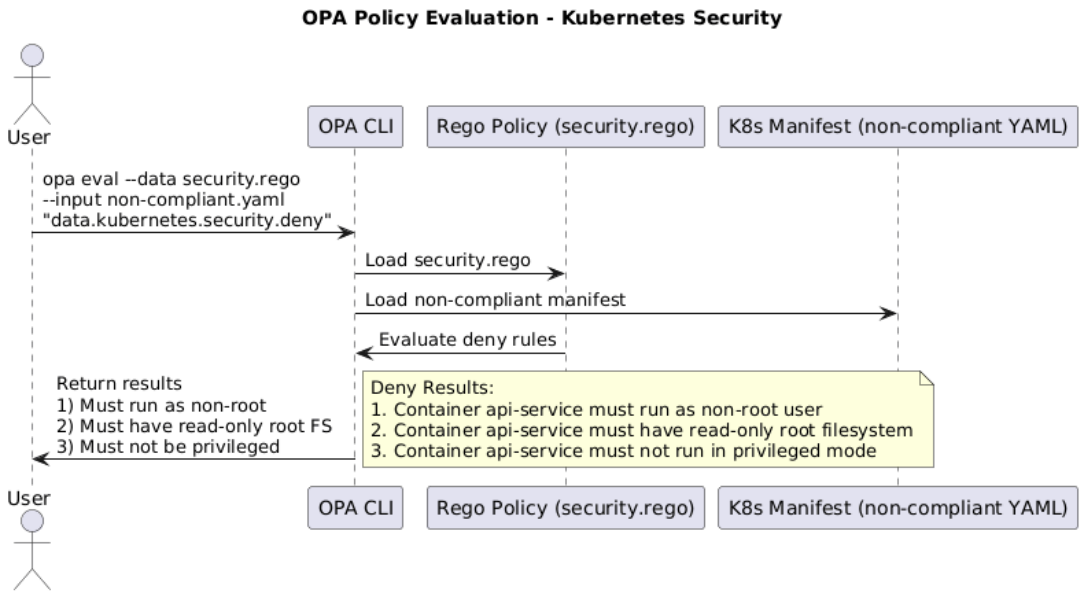

Figure 2: OPA Example Output from Experiment 1

the automated shift-left security controls could effectively identify and remediate routine misconfigurations and vulnerabilities.

6.1 Experiment 1: Preventing the Use of Containers with Insecure Configuration

Purpose: To confirm that policies function to prevent containers from running as root and in privileged mode.

Approach: A non-compliant Kubernetes Deployment manifest (`non-compliant-manifest.yaml`) with a `runAsUser: 0` configuration and a `privileged: true` setting was created. In addition, a compliant manifest (`compliant-manifest.yaml`) was generated that utilized a user other than root, along with safe settings.

Result: The `validate-kubernetes` job, when run in the CI pipeline, was able to catch the violations within the non-compliant manifest using OPA, thus marking the pipeline as failed. If this manifest were applied to a cluster with OPA Gatekeeper already installed, the Kubernetes API server would deny the request and block the deployment of the non-compliant manifest. The compliant manifest, on the other hand, passed all checks and was successfully deployed without issues.

Conclusion: This experiment provides evidence of the effectiveness of the PaC framework at both the CI and runtime stages to prevent the creation of insecure container configurations, which is an essential safeguard against container escape and privilege escalation attacks.

6.2 Experiment 2: Implementing Operational Best Practices

Objective - To confirm that establishments requiring operational best practices to implement defined resource limits and implement health probes are indeed implemen-

ted.

Approach - A Deployment Manifest was generated with an Identical Implementation that did not contain any defined Resource limits (Requests and Limits) or Health probe configurations (Liveness probe and readiness probe). An Implemented Deployment Manifest contained both of those as required configurations.

Findings - The OPA Validation step within the CI Pipeline flagged the missing resource configurations within the Non-Compliant Deployment manifest; Depending on the severity level associated with the relevant Policy (Deny vs. Warn), either the CI Pipeline was prevented from executing, or only a warning was generated. By generating a warning or preventing execution, the developer received immediate notification of any deviation from best practices, which are essential for cluster stability and resource management. The Implemented Deployment Manifest executed without any objections.

Conclusions - The conclusions derived from this Testing experiment concluded that the PaC principle can be used to both Deny Critical Security Vulnerabilities, as well as Enforce Operational Standards, which improve system reliability and efficiency.

6.3 Experiment 3: Automated Vulnerability Scan

Goal: To prove that the CI pipeline can automatically identify and block the use of container images with pre-existing critical vulnerabilities.

Methods: A `Dockerfile.vulnerable` was made to have; and contained a `requirements.txt` file that defined previous Python library versions that have been identified to be vulnerable. The CI pipeline also contained Trivy configured to fail the build if HIGH, CRITICAL vulnerabilities were detected.

Results: During the `build-images` job in the CI pipeline, when executing the Trivy scan of the image created from the `Dockerfile.vulnerable`, there were multiple critical vulnerabilities found. during the `vuln-check` step of the pipeline, these results were parsed and included in the summary of the pipeline's work. The build job then exited with a failure code. Therefore, the image that was identified as having critical vulnerabilities was never uploaded to the ECR (Elastic Container Registry) or utilized in any manner.

Conclusion: The evidence produced through Experiment 3 shows the value of utilizing automated vulnerability scans as part of the "shift-left" method of software development. The ability to identify individual CVEs (common vulnerability and exposure references) during a build phase prevents developers from deploying vulnerable code into production, thereby decreasing the size of their attack surface.

Table 3: Sample Trivy Scan Results

Image	CRITICAL	HIGH	Status
Vulnerable	3	12	× **BLOCKED**
Secure	0	0	✓ **PASSED**

6.4 Experiment 4: Validation of Infrastructure Security

A validation experiment was done to see whether CI could prevent the application of insecure Infrastructure-as-Code during the CI process. To begin the experiment, a Terraform variables file (`non-compliant.tfvars`) was created that contained insecure settings. The variables in the `non-compliant.tfvars` file included `db_publicly_accessible = true`; `db_storage_encrypted = false`; and a security group that allows all traffic `0.0.0.0/0` (These insecure variables would allow the creation of an insecure infrastructure). For the purpose of this experiment, the `validate-terraform` job of the CI pipeline was utilized. Terraform produced a plan using the insecure variables specified in the `non-compliant.tfvars` file. An OPA scan was performed on the Terraform plan created using insecure variables; this scan identified multiple violations against the `security.rego` policy for Terraform. The `validate-terraform` failed because of the identified violations, and it reported the reason as "Public access to sensitive resources detected", and the job failed because of these violations. This provided a mechanism to stop the insecure infrastructure from being created as a result. This experiment demonstrated that Policy-as-Code (PaC) can enforce policy compliance for Infrastructure Code at least as well as it can for Application Code, ensuring that the Cloud Foundation conforms to Security before it is created in the real world.

6.1 Discussion

The combination of these four experimental studies corroborates the central premise of this project. These findings indicate that implementing the integration of security scanning and automatic policy enforcement into the CI/CD pipeline will enable the creation of an agile and secure development environment. The "shift-left" methodology allows for immediate feedback to the developer regarding issues related to security, misconfigurations, etc. Creating a production-ready artifact without potential security-related issues results in a significant decrease in the likelihood of having security breaches after deployment.

In addition, the outputs generated during the manual deployment process verify that the combination of application and infrastructure worked as expected when both policy and quality gate requirements were fulfilled. In addition, the ability of deployed services to communicate with the database and the observability stack illustrates that no security measures/dividers had any effect on the overall operation of the solution. Furthermore, the actual data being pulled into the Grafana dashboards provides a clear visual representation of the application's functionality and its compliance with defined policies (in addition to being a real-time window on the application's health).

The novelty of this research is in the design, implementation and validation of the holistic fully integrated cloud-native DevOps ecosystem which integrates Infrastructure as Code, CI/CD automation, Policy-as-Code enforcement and Observability Metrics in real-time into a single system which creates an automated security compliance and vulnerability management layer. The existing literature typically handles these domains separately whereas this project demonstrates the successful integration through these four controlled experiments which demonstrates that policy violations can be tracked in real-time which addresses a critical gap between DevOps velocity and security assurance.

7 Conclusion and Future Work

The purpose of this study is to provide an answer to the question : **"To what extent does integrating a Policy-as-Code framework into CI/CD pipelines improve automated secur- ity compliance and vulnerability management in cloud-native applications?"**

To achieve the goals of the research, a microservices application was built and deployed to a secure, auto-provisioned cloud environment using Terraform technology and a comprehensive CI/CD Pipeline developed using GitHub Actions to manage the entire lifecycle of the microservices application. Most noteworthy, an extensive Policy-as-Code framework was used to integrate security into each stage of the CI/CD Pipeline through using Open Policy Agent (OPA) and integrated vulnerability scanning through Trivy. Evaluation testing of the system showed it can be relied upon to effectively identify and remediate numerous types of vulnerabilities, such as insecure container setups (e.g., exposed ports) and vulnerable third-party software packages, in addition to misconfigured cloud infrastructure. Additionally, using a custom dashboard for real-time Observability/Monitoring and Compliance Tracking helped ensure that ongoing view of how the application was being developed and how well it complied with regulatory and Security standards was maintained.

This work's most significant finding is that a shift-left security model powered by automated policy enforcement provides organizations with a way to secure cloud-native applications without compromising speed of innovation. By classifying policy and security configurations as code, organizations gain the ability to version, audit, and test their policy and security configuration as part of their software development process. In doing so, security is transformed from a manual post facto gating process into an automated, proactive, and collective responsibility.

These findings have substantial implications for the academic and practitioner communities. They offer organizations wanting to adopt DevSecOps a viable and repeatable approach to delivering on that objective, and they show that organizations can use intelligent automation to mitigate the presumed dichotomy between speed of delivery and security.

7.1 Contribution to Existing Work

This research adds to the current body of existing work by giving an initial and verified representation of a complete DevOps + Policy-as-Code integrated pipeline. Although previous studies discuss these components separately, this research shows conclusively how they work together in a complete framework.This research proves to be an intersection between theory and actual implementation by providing reproducible workflows, policy templates and experimental outcomes for researchers to utilize in the future.

7.2 Limitations

While the project was successful, there are also some limitations to discuss. The canary deployment strategy used is simplistic and only delays based on a specific time interval, rather than utilizing algorithmic evaluation of performance metrics. The OPA Policy set selected represents a reasonable cross-section of the policies in this domain, but does not provide complete coverage. The manner in which AWS credentials were provided to the

CI/CD pipeline was through the use of stored secrets; a more secure way to implement this in a production environment would have been through the use of an OIDC federated server.

7.3 Future Work

Future work should be focused on three areas based on the previous recommendations: First, a set of capabilities within the CD pipeline should be build out that support automated canary analysis based on metrics from Prometheus. Second, a need exists to develop a more exhaustive library of Rego policies that provide security, compliance (e.g. GDPR and HIPAA), and cost optimization support in greater depth. Third, Dynamic Application Security Testing (DAST) tools should be added to the pipeline in order to scan the application in the staging environment for vulnerabilities that can only be detected while the application is running. Fourth, existing workflows in GitHub Actions should be refactored to authenticate with AWS using OpenID Connect (OIDC) instead of using long-lived access key secrets. The results of this project illustrate how the merging of the principles of Cloud-Native and DevOps automation with Policy-as-Code have created a framework for building software and infrastructure that will allow companies to build, deploy and manage secure, compliant and responsive software systems rapidly and effectively.

References

[1] Anaka, H., 2022. AI and DevOps Integration for Cloud-Native Applications. International Journal of Artificial Intelligence and Machine Learning, 6(8).

[2] Arif, T., Jo, B. and Park, J.H., 2025. A Comprehensive Survey of Privacy-Enhancing and Trust-Centric Cloud-Native Security Techniques Against Cyber Threats. Sensors, 25(8), p.2350.

[3] Chippagiri, S. and Ravula, P., 2021. Cloud-Native Development: Review of Best Practices and Frameworks for Scalable and Resilient Web Applications. Int. J. New Media Studie, 8, pp.13-21.

[4] Deng, S., Zhao, H., Huang, B., Zhang, C., Chen, F., Deng, Y., Yin, J., Dustdar, S. and Zomaya, A.Y., 2024. Cloud-native computing: A survey from the perspective of services. Proceedings of the IEEE, 112(1), pp.12-46.

[5] Ganesan, P., 2022. Observability in Cloud-Native Environments Challenges and Solutions. International Journal of Core Engineering & Management.

[6] Guduru, S.K., Challenges and Development Trends in Cloud-Native Applications: A Comprehensive Survey.

[7] Kader, M.A. and Ahmed, R., 2025. Modern Cloud-Native Architectures: A Review of Full-Stack Design Patterns and AWS Best Practices. Scientia. Technology, Science and Society, 2(11), pp.283-290.

[8] Kader, M.A. and Ahmed, R., 2025. A Comprehensive Review on Full-Stack Development in the Cloud Era: Trends, Tools, and Best Practices. Scientia. Technology, Science and Society, 2(11), pp.310-318.

[9] Kosińska, J., Baliś, B., Konieczny, M., Malawski, M. and Zieliński, S., 2023. Toward the observability of cloud-native applications: The overview of the state-of-the-art. IEEE Access, 11, pp.73036-73052.

[10] Oyeniran, O.C., Modupe, O.T., Otitoola, A.A., Abiona, O.O., Adewusi, A.O. and Oladapo, O.J., 2024. A comprehensive review of leveraging cloud-native technologies for scalability and resilience in software development. International Journal of Science and Research Archive, 11(2), pp.330-337.

[11] POOJARY, K.K., ABHAY, A.R., SOWJANYA, N., POPESCU, V., MITROI, A.T., NIOATA, R.M. and RAJ, K.K., 2025. A Comprehensive Review on Scaling Machine Learning Workflows Using Cloud Technologies and DevOps.

[12] Ramesh, G., Pai, T.V., Birau, R., Poojary, K.K., Shingad, A.R., Sowjanya, N., Popescu, V., Mitroi, A.T., Nioata, R.M. and Raj, K.K., 2025. A comprehensive review on scaling Machine Learning workflows using Cloud Technologies and DevOps. IEEE Access.

[13] Sakinala, K., 2025. Monitoring and observability for cloud-native applications. Journal of Computer Science and Technology Studies, 7(8), pp.101-115.

[14] Somi, V., 2022. Integrating Security in Cloud-Native CI/CD Pipeline: A Comprehensive Review of DevSecOps Practices. Journal of Artificial Intelligence & Cloud Computing, 1(4), pp.1-6.

[15] Syal, M., 2022. Automated Software Testing Frameworks for Cloud-Native Applications. International Journal of Advanced Research in Computer Science & Technology (IJARCST), 5(4), pp.6890-6894.

[16] Theodoropoulos, T., Rosa, L., Benzaid, C., Gray, P., Marin, E., Makris, A., Cordeiro, L., Diego, F., Sorokin, P., Girolamo, M.D. and Barone, P., 2023. Security in cloud-native services: A survey. Journal of Cybersecurity and Privacy, 3(4), pp.758-793.

[17] Tong, W., 2025. Cloud Native Application Disaster Recovery in a Multi-Cloud Environment–A DevOps Approach using Terraform.

[18] Vangala, V., Optimizing Cloud Infrastructure Management in DevOps.

[19] Wang, L., Yang, Z., Zhong, C. and Zhang, W., 2025, July. Cloud-Native Hybrid Deployment Technology Under Narrowband Communication Conditions. In 2025 21st International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD) (pp. 737-746). IEEE.

[20] Wei, H., Madhavji, N. and Steinbacher, J., 2025, March. A Map of Cloud-Native Practices and Tools to Achieve Desirable System Qualities. In 2025 IEEE 22nd International Conference on Software Architecture (ICSA) (pp. 221-231). IEEE.