National College of Ireland

Design and Implementation of a single CI/CD Pipeline to Deploy Multi Cloud Kubernetes Clusters via Terraform and Jenkins

MSc Research Project
Masters in cloud computing

Jeena George
Student ID: x23379944

School of Computing
National College of Ireland

Supervisor: Abubakr Siddig

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Jeena George
Student ID:	x23379944
Programme:	Masters in cloud computing
Year:	2025-2026
Module:	MSc Research Project
Supervisor:	Abubakr Siddig
Submission Due Date:	28-01-2026
Project Title:	Design and Implementation of a single CI/CD Pipeline to Deploy Multi Cloud Kubernetes Clusters via Terraform and Jenkins
Word Count:	7317
Page Count:	22

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:	Jeena George
Date:	28th 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):	

Design and Implementation of a single CI/CD Pipeline to Deploy Multi Cloud Kubernetes Clusters via Terraform and Jenkins

Jeena George
x23379944

Abstract

In this thesis, an automated multi-cloud deployment framework is proposed to avoid vendor lock-in and enhance the availability of applications. The management and application of Kubernetes clusters on Amazon EKS and Google GKE is undertaken through a single Terraform/Jenkins CI/CD pipeline. The implementation has three deployment modes to achieve costs and resilience: dual cloud active deployment to achieve maximum uptime, primary standby architecture to achieve cost optimization and fast failover, and single cloud deployment to achieve rapid reprovisioning to remain independent of providers. Its structure consists of the Terraform infrastructure definition tool, Jenkins to define automated build and deployment processes, Kubernetes, a container orchestration engine, and Cloudflare, an intelligent traffic distribution tool. The solution displays enhanced automated provisioning, cross cloud monitoring with Prometheus and Grafana. Findings demonstrate that the multi-cloud automation improves operational reliability, minimizes the reliance on a single provider, and offers organisations with strategic flexibility in managing the cloud resources.

1 Introduction

1.1 Background

Cloud computing has turned out to be the backbone of today software infrastructure which provide distributed, complete scalable and highly automated environments in which applications can be deployed. With the more widespread use of cloud-native architectures, a significant number of organizations are no longer implementing single-vendor strategies, but instead are adopting multi-cloud strategies where they use multiple cloud providers at the same time. It has been proved that multi-cloud implementation enhances operational resiliency, greater strategic flexibility, and less reliance on a single vendorPetcu (2013), Wagle et al. (2015). The ability to spread the workloads among the providers like Amazon Web Services (AWS) and the Google Cloud Platform (GCP) reduces the risks of vendor lock-in such as service failures, proprietary API, unexpected price fluctuations, and expensive migration processesTang (2022). Although these benefits exist, multi-cloud environments are highly complex in terms of technology. The cloud providers have different APIs, networking frameworks and identity management strategies and monitoring mechanisms, so it is hard to have consistent automation. Research points

out integrating problems like configuration drift, deployment differences, and the rise in operational overhead as the frequent issues with operating a heterogeneous infrastructure Burns et al. (2022), Tang (2022) Such challenges require powerful automation and orchestration systems to ensure that the reliability and operational efficiency in the clouds remains intact. Infrastructure-as-Code (IaC) tools have been proposed as a solution of high importance in handling complexity in the multi-cloud world. Particularly, Terraform makes it possible to provision infrastructure declaratively and with version control, using reusable modules and with an open range of cloud providers, so that it can consistently deploy on whichever underlying platform is being used Gudelli (2023), Howard (2022) Kubernetes has established itself as the container orchestration de facto at the application layer, managed services such as Amazon Elastic Kubernetes Service (EKS) and Google Kubernetes Engine (GKE) now make it easier to manage a cluster, schedule workloads, and scale automatically Krief (2022) Continuous Integration and Continuous Deployment (CI/CD) systems like Jenkins are used to coordinate end-to-end processes to compose infrastructure provisioning, application deployment, testing and teardown into repeatable pipelines to create the automation stack Labouardy (2021), Ferrer et al. (2012). The project will bring together Terraform, Jenkins, Kubernetes (via EKS and GKE), and Cloudflare to create and deploy a multi-cloud CI/CD system that of this kind will be fully automated. Architecture permits scalable resource deployment planning includes the operation of two clouds concurrently to ensure as much resiliency as possible, a lightweight standby cluster with a low cost to provide failover service, or in the instance of an alternate cloud, a rapid reprovisioning of infrastructure. Cloudflare Workers can offer health-based routing of traffic between clusters, whereas Prometheus and Grafana can give real-time observability in both settings. This project will show that the commercial issues of multi-cloud automation can be practically resolved by displaying a unified and vendor-neutral deployment model and prove that the high availability, cost-control, and vendor-independence can be implemented in the same pipeline framework.

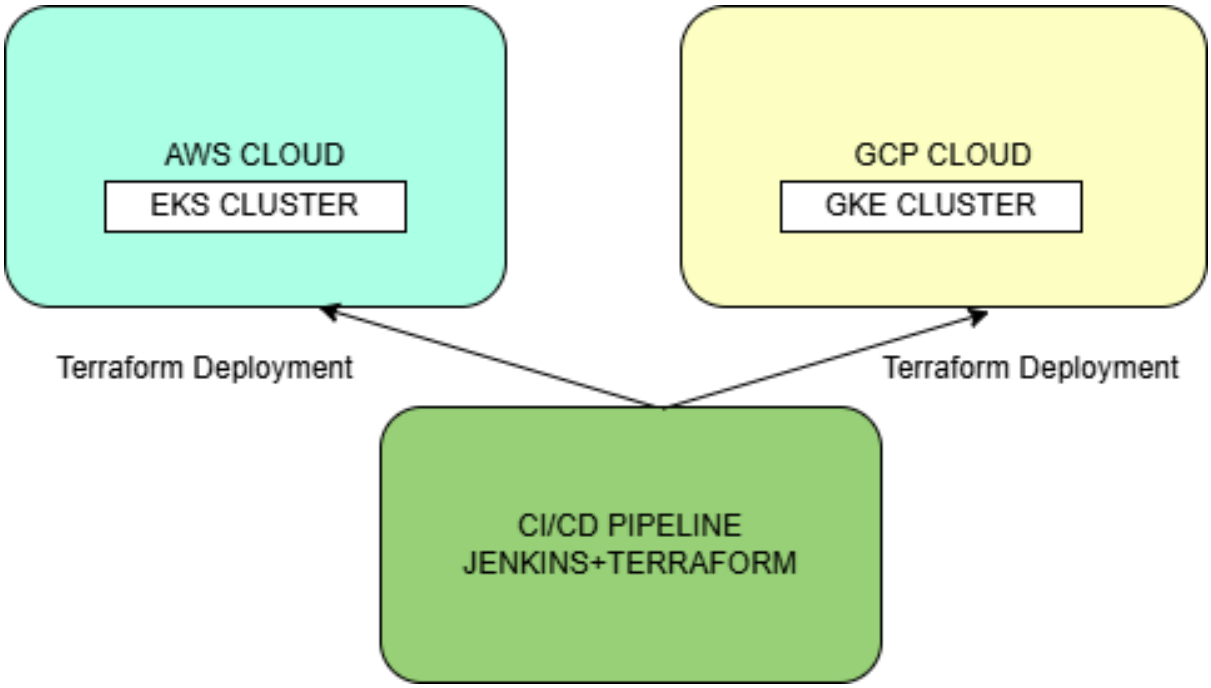


Figure 1: simple multi cloud architecture

1.2 Motivation And Importance

The current project deals with the increasing demand to have supportive, mobile, and vendor-independent deployment systems with the rising dependence of organizations on cloud-based infrastructure. Some of the risks that single-cloud approaches subject businesses to include downtime, cost variation, and vendor lock-in, which is commonly known in literature Wagle et al. (2015). Multi-clouds are used to reduce these risks by providing a workload portability and provider workaround. Nevertheless, there are still limited practical applications with the majority of existing studies investigating the application of individual tools, such as Terraform, Jenkins, or Kubernetes, but never showing end-to-end multi-cloud automation shift.

The gap in this project is that the project provides a fully working multi-cloud CI/CD system which is lifecycle automation. Terraform has Kubernetes cluster support on AWS EKS and GCP GKE and Jenkins is used to control the processes of deployment and Cloudflare Workers scales the traffic to the healthy cluster based on real time health checks. The system shows that not only are fully automated multi-cloud deployments with a smooth failover technically feasible, but they are also operationally viable once a combination of Infrastructure-as-Code, container orchestration, and smart load balancing is implemented as one system.

1.3 Research questions And Objectives

This research is guided by two questions. The initial question is exploring how an Infrastructure-as-Code, CI/CD automation, container orchestration, and health-based traffic routing may be integrated into a single cohesive pipeline and provide the ability to provision, deploy, and manage workloads across multiple cloud providers. The second question is how well a multi-cloud deployment system can provide high availability, active application state, and automated failover and not rely on vendor-specific services or be constrained to a single service provider. These questions, when combined, will investigate whether provider-agnostic multi-cloud architecture can be fully automated to increase resilience, lessen dependency threats, and help to provide homogenous continuity of the applications in heterogeneous cloud environments. The project will attempt to fulfill some major objectives in order to answer these research questions. To begin with, it will develop a vendor-agnostic multi-cloud architecture that has the potential to provision uniform Kubernetes clusters in both AWS (EKS) and GCP (GKE) relying on reusable Terraform functionality. This project then attempts to realize a cohesive Jenkins-based CI/CD system which will be used to automate the entire deployment life cycle, such as infrastructure provisioning, app deployment, and automated app teardown, in one or both cloud-based environments. It will also scale the same containerized workloads to the two clusters to confirm that the behaviour of the application is similar among providers. The other goal is to apply health-related port routing where a Cloudflare Worker script, which constantly checks the health of the cluster, reroutes users to the accessible cloud in case of failure. The project also seeks to define end to end monitoring via Prometheus and Grafana in order to monitor the performance and application health of the system in real time. Lastly, the solution will be tested to have deployed reproducibility, simulated cloud provider outage, tested the response time of failing over, and demonstrated the resilience of the system and flexibility in its operation through testing the possibility to dynamically add or destroy infrastructure devoid of downtime.

1.4 Scope, Limitations, and Assumptions

1.4.1 Scope

The project will illustrate vendor-neutral automated deployment to various clouds based on the example of the AWS EKS and GCP GKE as instances of managed Kubernetes services. The architecture and pipeline can be generalized to new providers (e.g., Azure AKS, DigitalOcean Kubernetes), however because of time and resources constraints, only AWS and GCP were implemented and tested.

1.4.2 Limitations

1. The deployed system is deliberately lightweight (a containerized webserver) to emphasize infrastructure automation, be able to provision the infrastructure in a consistent way, and provide failover mechanics instead of application-layer complexity. Stateful workload systems like production systems, databases or inter-service dependencies would need extra considerations like data replication and session management.

2. The monitoring stack installed on each cluster differs slightly because of the variation in the EKS and GKE node capacity, API behavior, and default resource limits. GKE is configured with the entire kube-prometheus-stack, which includes the Grafana, and EKS is configured with a minimal Prometheus deployment. This impairs the ability to compare monitoring dashboards directly, but has no impact on the basic failover.

3. The Cloudflare Worker executes simple HTTP health checks to check the availability of the cluster and redirects traffic respectively. This method does not enact session persistence, location load distribution or weighted traffic bifurcation which would be needed in more advanced production applications.

4. Because of budget and time, the project will not involve extensive cost modelling, long-term performance benchmarking, and load testing with high traffic volumes. The emphasis is placed on operational feasibility validation and automation as opposed to performance or specific economic optimization.

1.4.3 Assumptions

There are various assumptions under this project. It also presumes that the AWS and GCP offer managed Kubernetes services that are reliable and production-ready and have the same set of core functionality to orchestrate containers. It further presupposes that Cloudflare Workers have the reasonable ability to run health checks and to route traffic having reasonable turnaround to show failure of cross cloud, even in the absence of more advanced traffic management capabilities. The Terraform modules, Jenkins pipeline and Kubernetes settings utilized by this project are presupposed to reflect the realistic patterns that could be modified to production settings with proper security hardening, secret management, and compliance measures. Moreover, the assumption is that the network connectivity between the CI/CD server and cloud vendors and the Cloudflare will be stable throughout deployment and testing phases. Lastly, the project presumes the use of operator-initiated deployment using the parameterized build interface of Jenkins, as opposed to automatic Git-driven deployments, on the basis that this design variable

offers explicit control over multi-cloud provisioning determinism,that is the choice of target clouds, and either apply or destroy, which is a paramount safety requirement when handling production scale cloud resources.

2 Related Work

2.1 Multi-Cloud Adoption, Motivation and Challenges

The use of multi-cloud has been extensively discussed as one of the reactions to single-cloud limitations. The original research conducted by Buyya et al. (2013) and Petcu (2013), identified the dangers of vendor lock-in, service outages, and fluctuation of prices, and made multi-cloud plans a necessity to ensure portability, flexibility, and business continuity. Opara-Martins et al. (2016) supported this by demonstrating that proprietary APIs and deficit of interoperability impose long-term obstacles on the operations of organizations that are based on collaboration with one provider. Later empirical studies illustrate the resilience value of allocating workloads through clouds - such as Tang (2022) has recently reported enhanced reliability on scientific computing - but the studies tend to be on the topic of scheduling or decision-making structures versus the entire automation pipelines. Newer reviews, such asGudelli (2023) and Ferrer et al. (2012) also highlight the challenges that occur as API variability, differences in security models, and heterogeneity of infrastructure, which are more recent. Although there has been extensive theoretical efforts, there have been few practical examples of automated and end-to-end systems of multi-cloud deployment. This void is the reason why the current project is undertaken to introduce an entire multi-cloud pipeline incorporating provisioning, automation of CI/CD, and multi-cloud failover.

2.2 Infrastructure-as-Code (IaC) and Terraform Automation

The Infrastructure-as-Code (IaC) approach has transformed the management of infrastructures to allow workflows of providing infrastructure to be declarative, versioned, and repeatable. It is well known that Terraform supports multi-cloud deployment,Howard (2022) demonstrated the automation aspect as Terraform can be used to provision in addition to lessening human error and ensuring uniform deployments, but state management, module dependencies, and alignment with cross-cloud configuration continue to be problematic. Gudelli (2023) also established that Terraform provider-agnostic model has benefits over cloud-specific systems such as AWS CloudFormation, although there is still constant apprehension about credential and secret management. Beyond these IaC research programs, other studies indicate advantages, like interdepartmental deployments and enhanced collaboration, with complexities cited, like debugging and dependency management Artac et al. (2017), Guerriero et al. (2019), Krief (2022) has reaffirmed the significance of Terraform in DevOps, but was looking primarily at the implementation of Terraform in a single cloud. In general, the current studies concentrate more on the process of IaC provisioning, as opposed to its combination with CI/CD and Kubernetes. In this project, the gap will be filled by applying a single pipeline that will be used to provision, deploy and manage workloads in AWS and GCP using Terraform, Jenkins, and Kubernetes.

2.3 CI/CD Pipelines in Multi-Cloud Environments

Often used CI/CD platform Jenkins is the most popular open-source platform because it is versatile and can be extended by a wide range of plugins. Labouardy (2021) has shown how Jenkins pipeline-as-code and inter-operates with Docker and Kubernetes along with Terraform to automate deployments, but only in single-cloud status. Goyal (2024)discussed the optimisation of CI/CD to enhance reliability and performance, but has not touched upon the challenges of multi-clouds when deploying many providers. In wider literature, such as Shahin et al. (2017) such persistent CI/CD issues as tool integration, environment drift, and failure handling were found, but they grow further complicated in multi-cloud environments, which require heterogeneous APIs, divergent authentication procedures, and coordinated parallel deployments. AlthoughPandi et al. (2023) proposed the Jenkins-based multi-cloud orchestration model, it did not have automated failover, traffic routing, and single-level monitoring. Armenise (2015) reported Jenkins workflows and concentrated on singleness in delivering the environment.current literature does not offer much practical advice regarding fully automated multi-cloud CI/CD pipelines. The given project fills that gap through deploying an instance of a Jenkins pipeline that is able to provision, deploy, monitor, and manage both AWS and GCP workloads into one network workflow.

2.4 Kubernetes, EKS, GKE and Cross-Cloud Orchestration

Kubernetes is the standard most widely used in container orchestration because of its portability, scaling, and good ecosystem. Burns et al. (2022) pointed out how Kubernetes can be used to deploy across a variety of managed services, including Amazon EKS and Google GKE, whereas Ferrer et al. (2012) demonstrated that variations in provisioning speeds, networking options, and autoscaling had a complexifying effect on multi-cloud environments. Amaral et al. (2015) also revealed that despite the fact that Kubernetes enhances modularity and scalability among micro services, it also presents operational issues in regard to communication of services and infrastructure dependencies. Although previous literature confirms the portability advantages of Kubernetes, even regarding the scenarios where a single provider or single cluster is utilized, the challenge of coordinating concurrent deployments across multiple clouds remains largely unstudied.Shafiq et al. (2022) stressed the necessity of the effective cross-cluster routing of the traffic but they also stated that the practical solution to real-time health-based provider-to-provider actual failover mechanisms was lacking. The exact solutions to these gaps are presented by this project, including automated parallel deployments of both EKS and GKE into one CI/CD pipeline, health checks based on Cloudflare which are integrated and change the routes to the healthy cluster in real time, which provide a viable cross-cloud orchestration and failover.

2.5 Monitoring, Observability and Multi-Cloud Health Management

Heterogeneous Kubernetes platforms making up multi-cloud environments demand high levels of monitoring and observability in order to cover the variation in operational behaviour. Turnbull (2018) says that the only way to make cloud-native systems reliable is to collect continuous measurements, have real-time alerts, and have deep insight into

system behaviour. AsPB et al. (2024) have shown, Prometheus and Grafana delivered an efficient observability stack to Kubernetes, allowing to get a detailed view of the node and pod performance.

Current studies, including Ferreira & Sinnott (2019) and Rahman et al. (2019), observe that managed Kubernetes services typically have their native observability, and a unified cross-cloud observability is challenging. Nevertheless, the majority of this literature concentrates on the single-cloud monitoring and offers little information on the establishment of consistent, provider-neutral visibility across multiple clouds.

An additional research gap is that there are no studies that prove the use of Cloudflare health checks within CI/CD processes to aid dynamic health conscious routing across clouds. The given project will fill that gap by introducing Prometheus-based metrics along with Cloudflare Worker health checks and establishing a combined monitoring and failover mechanism. The subsequent system provides real-time multi-cloud observability and automatic resilience to both EKS and GKE clusters without relying on any cloud-specific monitoring tools.

2.6 Critical Analysis of Literature and Emerging Research Gaps

The current literature is aware of the positive effects of multi-cloud systems in reducing the vendor lock-in effect and enhancing resiliency although a significant gap is noted between theory and the actual practice. The studies of Terraform, Jenkins, and Kubernetes are mostly based on the subject of single cloud deployment and fail to exhibit cross-provider automation. Similarly, there is no literature that unites Infrastructure-as-Code, CI/CD pipelines, container orchestration, monitoring, and smart traffic routing into a single non-incomplete system that is able to perform an automated provisioning, deployment, health monitoring, and cross-cloud failover. In this project, the author closes these gaps by developing a fully automated multi-cloud pipeline based on Terraform, Jenkins, EKS, GKE, Cloudflare, and Prometheus/Grafana that demonstrates that a vendor-neutral highly available architecture is possible under the conditions of integrated DevOps automation.

3 Methodology

3.1 Research Approach and Design

The approach used in this project is Design Science Research (DSR), which is suitable to those studies that are interested in the creation and testing of a technical artefact. DSR aids the hypothetical engineering, execution and evaluation of solutions, which resolve practical issues. The methodology in this project is to determine the issues of multi-cloud automation, establish the goals concerning vendor-neutral provisioning and cross-cloud failover, and integrate an artefact with the help of Terraform, Jenkins, Kubernetes, and Cloudflare Workers. Then, the system is shown by implementing it on AWS EKS and GCP GKE and testing it in terms of reproducibility, consistency, and failure behavior. The DSR paradigm makes sure that the design process and the artefact produced benefit in the real world and academically.

3.2 System Architecture Overview

The multi-cloud CI/CD architecture employed in the current project, which is shown in Figure 1, has five layers, each integrated to provide various functionality: infrastructure provisioning, CI/CD orchestration, container orchestration, global traffic routing, and observability. The architecture is built to facilitate vendor-neutral deployments and auto-failover and consistency in its operations between Amazon Web Services (AWS) EKS and Google Cloud Platform (GCP) GKE.

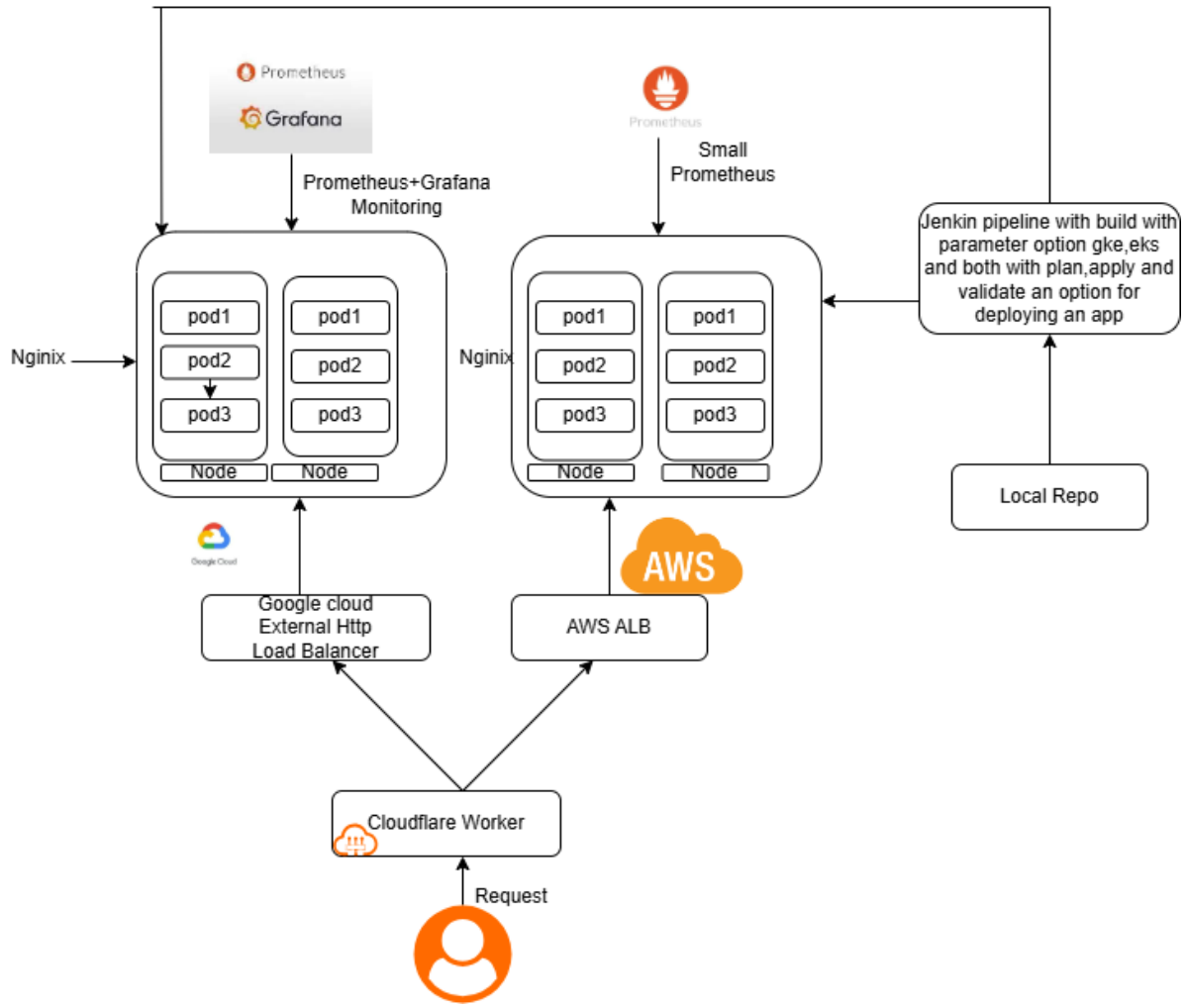


Figure 2: Multi-Cloud CI/CD Architecture Diagram

3.2.1 Architecture Overview

The following core components are integrated as shown by the system architecture: Terraform: Declarative infrastructure provisioning in AWS and GCP Jenkins: workflow automation of the deployment process. Kubernetes (EKS and GKE): Workload and container orchestration. Cloudflare Workers: Traffic based on health and cross cloud-failure. Prometheus/Grafana: Monitoring and observability: Both clouds. It places a high value on high availability, multi-cloud redundancy and vendor independence with infrastructure definition, deployment logic, and through application manifests all being cloud-agnostic and reusable.

3.2.2 Architecture Layers

Layer1:Terraform Terraform is the building block of the system which is a declarative, version-controlled way of provisioning cloud resources. The structure of the Terraform module creates comparable infrastructure in AWS and GCP, such as Virtual Private Clouds (VPCs) and subnets, EKS and GKE control planes and node groups with instance type and scaling policies settable. It also installs load balancers and IAM roles needed to operate the clusters. Terraform provides infrastructure provisioning consistency across clouds even in cases where providers differ in their API and networking models due to the use of parameterized modules and provider-agnostic logic.

Layer 2: CI/CD Orchestration (Jenkins) Jenkins manages the complete deployment lifecycle by managing a multi stage pipeline which: Checks terraform provisions with terraform plan. updates or provisions cloud infrastructure through terraform are possible. It also specifies kubectl contexts to log into EKS and GKE clusters. Applies Kubernetes resources (Deployments, Services, ConfigMaps) on both clusters.

Layer 3.Both AWS EKS and GCP GKE execute the same workloads in Kubernetes and provide consistency in the application-layer. Each cluster contains: Application pods Containerized (e.g. Nginx webserver in the demonstration deployment) Services based on LoadBalancer-type The services expose applications through cloud-native load balancers (AWS ELB and GCP Cloud Load Balancing). Subcomponent (prometheus monitors,Grafana kube-state-metrics)

Layer 4: Cloudflare Workers Cross-cloud failure is made possible with Cloudflare Workers which has the smart routing layer. The Worker script: Accepts a request from the users. Checks on the state of both EKS and GKE clusters on the HTTP level. Routes traffic to the main cluster in case of healthiness. Switches to the secondary cluster automatically should the primary go offline or give back an error response. Gives proper error messages in case the two clusters are offline. This health based routing policy is implemented at the edge so that low latency failover decisions are made without delay caused during DNS propagation.

Layer 5: Observability (Prometheus, Grafana) A combination of unified observability is accomplished by a combination of: Prometheus: Installed on both clusters in order to collect metrics of Kubernetes components, nodes and pods of the application. Grafana: Helpful in providing visualization dashboards to monitor what is going on in real-time (installed in GKE; available on port-forwarding)

3.3 Infrastructure Provisioning Method (Terraform IaC)

The concept of Infrastructure-as-Code (IaC) is the foundation of the proposed project in terms of multi-cloud provisioning, which allows managing the resources of both AWS and GCP declaratively and under a version control. Terraform has been chosen due to its provider-agnostic framework combined with a modular construction that helps in deploying the same over heterogeneous clouds. In this project, Terraform will be used to deploy VPC networking infrastructure and IAM identities and roles, managed Kubernetes control planes (EKS and GKE), and workers node populations in both clouds. This is because the architecture employs reusable modules with parameterized inputs and is therefore consistent across the providers minimizing configuration drift and enhancing reproducibility. Remote state management also provides auditability and synchronised and collaborative updates. This proves that fully automated IaC workflows are reliable to have complex multi-cloud infrastructure provisioned.

3.4 CI/CD Pipeline Design (Jenkins)

Jenkins is the orchestrator of the CI/CD pipeline in which the entire deployment lifecycle (infrastructure provisioning using Terraform, up to the deployment of the Kubernetes application in both AWS and GCP) is being automated. The reason is that Jenkins has an extensive support for multi-stage declarative pipelines which is supported by a mature-plugin ecosystem and is extensible. The pipeline can be implemented through the interface of Build with Parameters, where the operators have the opportunity to select either the type of deployment (AWS, GCP, or both) or the action in Terraform (plan, validate, apply or destroy). It is a triggering mechanism which is operator controlled thus is safe to ensure intentional running of multi-cloud workflows, which many cloud providers will find unavoidable due to the need to back up essential infrastructure changes.

3.5 Kubernetes Deployment Strategy (EKS and GKE)

Kubernetes is the cloud-agnostic orchestration layer of the system which allows application deployments to be consistent regardless of whether it is the AWS EKS or the Google GKE. The same Kubernetes manifests (Deployments, Services and ConfigMaps) are used to apply to each of the clusters to ensure consistent workload behavior in spite of provider variations. Using deployment configuration, replica counts and resource constraints and rolling-update policies are configured to facilitate high availability releases and zero-downtime releases. Definitions of services take the form of LoadBalancer-type objects that provide access to applications using cloud-native load balancers in which foreign endpoints are used as the targets of Cloudflare health checks. ConfigMaps are used to keep application content and environment parameters, which are not contained in container images, making them easier to maintain. Even though there exist hidden provider discrepancies in networking, IAM, and monitoring integrations, Kubernetes wraps them up, so that workloads are entirely portable. This deployment model is a direct support of the objective of the project to do away with vendor lock-in as a solution since it makes applications run identically across cloud platforms without alteration.

3.6 Application Deployment Process

The tool used in this project is a low-weight Nginx-based web service which supports HTML content which is administered using Kubernetes ConfigMaps. The design keeps configuration and container images independent and fails to employ rebuild or redeploy of the image when updates are required. The process of deployment starts with the creation of the base Nginx container and the application content is mounted in runtime with the help of ConfigMaps. Jenkins subsequently deploys the EKS and GKE clusters with the Kubernetes Deployment manifest and Service manifest, which involves using kubectl to develop the manifests and allow the scheduler of each cluster to allocate the pods based on the resource and policy limitations. The pipeline checks that pods are up, services available and external addresses of the load balancers are given. Lastly, the same deployments on both clusters are checked to assure the same behaviour of the application in any cloud, which proves that Kubernetes can offer cloud-agnostic workload portability.

3.7 Multi-Cloud Routing and Failover

Cross-cloud traffic routing is established with a Cloudflare Worker, a serverless and globally distributed server, which is a solution that is used to make real-time health-based routing of a request. Each request sent to the system is validated by the AWS EKS LoadBalancer endpoint; the endpoint responds to an established connection with a healthful HTTP 200 response, and traffic is redirected to AWS. When AWS experiences any timeouts or 5xx errors, the Worker will automatically redirect traffic to GCP GKE endpoint to ensure that failover will occur seamlessly without a human operator. Where neither of the two clusters are available the Worker responds with a service-unavailable response. This technology provides edge-computing performance failure over low latency, without relying on cloud-provider-specific features, preventing the problem of vendor lock-in. Although the mechanism works in a model with active and passive model and lacks session persistence or stateful resilience failure, it gives effective display of automated and cloud-agnostic failover to meet the high-availability goals in the project.

3.8 Monitoring and Observability Setup

The observability set up created to monitor both clouds was custom and resource-conscious. On GKE, a complete stack of kube-prometheus-metrics to Prometheus (metrics collection) and Grafana (visualization) were installed using Helm, and detailed CPU, memory, pod health, and network dashboards were available. The size of the node in the EKS cluster was small, so the system was run with a lightweight Prometheus deployment, providing the necessary performance and health metrics without overwhelming the system. Collectively these tools made it possible to have a multi-source observability model with GKE providing deep observability and EKS providing core monitoring data. Even though Cloudflare health checks do not necessarily involve the use of Prometheus, the joint monitoring results facilitated credible traffic failover as real-time knowledge of cluster health through real-time knowledge of cluster availability were maintained. This environment offers a hydrate, cloud observable, and resource-efficiency environment of multi-cloud environments.

3.9 Tools and Technologies Used

This project utilizes a current cloud-based technology stack that will allow vendor-neutral, full-automated multi-cloud deployment. Terraform offers the Infrastructure-as-Code base, making it possible to provide the AWS and GCP infrastructure declaratively and reproducibly with provider-neutral modules. Jenkins acts as the CI/CD orchestrator which implements Terraform workflows, Kubernetes manifests, and application deployments in both the clouds. Kubernetes is the common container orchestration layer that provides the consistency of the behavior of the workload on Amazon EKS and Google GKE. Cloudflare Workers lets the operator handle the global traffic and implement automated failover through completing live health checks and sending user requests to the functional cluster. The Prometheus and Grafana are used to implement monitoring and observability, and the kube-prometheus-stack is deployed in GKE and simple Prometheus run on EKS to meet resource needs. Taken together, these tools form a scalable and resilient and vendor-neutral automation system that enables multi-cloud operations with health awareness in real-time and high availability.

Step	Action	Deployment Workflow and Tools	Result
1	Trigger Deployment	Jenkins (Build with Parameters)	Operator selects deployment target (AWS, GCP, or both) and action (apply/destroy).
2	Infrastructure Setup	Terraform	Kubernetes clusters created on AWS EKS and/or GCP GKE.
3	Deploy Application	Kubernetes (EKS & GKE)	Application deployed on both clouds using Kubernetes manifests.
4	Configure Traffic Routing	Cloudflare Worker	Health checks enabled; user traffic routed to healthy cluster
5	Enable Monitoring	Prometheus & Grafana	Metrics collected and dashboards visualized for operational visibility.

Table 1: Deployment Workflow and Tools

3.10 Validation and Testing Procedures

The system was confirmed by means of structured testing which was aimed at testing of the correct functionality, reproducibility of deployment, operational resilience and autopiloting failover. Validation was done in stages and the components were tested separately and then system-wide.

3.10.1 Infrastructure Validation

The terraform configurations were checked by running terraform validate to determine the syntax correctness and then terraform plan to determine the resource creation as expected. Remoteness of state files was checked to match between the reported and real infrastructure. The validation needed a mistake-free implementation and proper planning of the infrastructure.

3.10.2 Deployment Functional Testing

Both EKS and GKE clusters were checked after Terraform provisioning with kubectl to check the cluster readiness, the registration of nodes, networking, and IAM configuration. The testing of application deplays indicated that all pods were brought to the Running state, LoadBalancer services were assigned external IPs, and the HTTP response included correct content. Validity of configMaps was through checking of mounted HTML files within containers.

3.10.3 Failure Mode and Effect Analysis

By firing at Cloudflare Worker routing, automated failover was tested by setting the primary cluster down (scaling to zero replicas) and then ensuring that correct routing occurred. Within one request cycle, traffic was supposed to change to the second cluster.

The tests on node scaling also confirmed the flexibility of the infrastructure and the proper redistribution of the workloads by Kubernetes.

3.10.4 Reproducibility and Idempotency Testing

Destruction of all infrastructure (terraform destroy) caused full lifecycle tests to be done by rebuilding environments with the help of Jenkins pipeline. IaC idempotency was verified through successful rebuilds and the reproduction of the same clusters, networking and workloads in an automated manner without having to touch the systems.

3.11 Summary Table

Component	Purpose	Roles	Result
Terraform	Infrastructure-as-Code for cloud resource provisioning	Creates GKE and EKS clusters, VPC networks, node pools, and load balancers using reusable modules	Reproducible and consistent multi-cloud Kubernetes infrastructure
Jenkins CI/CD Pipeline	Automates provisioning and deployment workflows	Executes Terraform plan/apply/destroy, configures kubeconfig, deploys application workloads, and validates services	Fully automated, repeatable multi-cloud deployment pipeline
Kubernetes (GKE and EKS)	Container orchestration across cloud providers	Runs identical application workloads using shared manifests (Deployment, Service, ConfigMap)	Consistent app behaviour across AWS and GCP
Cloudflare Worker	Failover and smart routing based on cluster health	Performs 5-second health checks and switches traffic between GKE (primary) and EKS (secondary)	Zero-downtime multi-cloud failover mechanism
Prometheus	Collection of cluster and workload metrics	Scrapes CPU, memory, pod, and node metrics across GKE and EKS for monitoring and analysis	Real-time observability into multi-cloud cluster performance
Grafana	Visualization of monitoring data	Provides dashboards for system metrics, workload health, and resource usage	Improved operational insight and performance monitoring
kubectl	CLI tool for interacting with Kubernetes clusters	Validates running pods, services, external IPs, and deployment status	Confirms successful deployment and cluster accessibility
Node.js Application	Test workload for pipeline validation	Deployed identically across both Kubernetes clusters via LoadBalancer service	Confirms cross-cloud consistency of application behaviour

Table 2: Components in the Multi-Cloud CI/CD System

4 Design Specification

4.1 Architectural Overview

The architecture of the system is the dual-cloud architecture on Amazon EKS as well as Google GKE with Terraform as an Infrastructure-as-Code (IaC) orchestrator. The clusters use respective and autonomous VPC networks and integrate node pools of auto-scales to provide elastic capacity handling. The architecture is based on a modular IaC scheme, which makes it reproducible, cross-cloud portable as well as deployable on heterogeneous platforms. The maintainability and scalability are also supported by this modularity, such that the infrastructure components may be extended or changed independently without impacting the integrity of the system as a whole.

4.2 Google GKE Infrastructure Design

The GKE environment is configured with Google Cloud Terraform native resources. A network topology is controlled with the creation of a custom VPC (10.10.0.0/16) and subnet. The cluster is configured without default node pool, so that it can be fully configured with dedicated node pool with e2-medium instances with auto scaling (1-3 nodes). Scopes can be monitored and logged and integrated with GCP Operations Suite. Cluster resilience is also provided with automatic repair and upgrade options that make it easy to operate.

4.3 Amazon EKS Infrastructure Design

The environment in EKS is configured through the Terraform EKS module in a securely developed VPC with the inclusion of both public and private subnets, a NAT Gateway, and an autoscaled group of t3.micro nodes, which works together to provide controlled CI/CD access, an efficient method of computer power and relevant security policies to ensure reliable inter-organizational orchestration.

4.4 Terraform Module and Network Architecture

both AWS and GCP setups are parallel Terraform module framework where variables are used to define cluster names, regions, machine types and autoscaling configurations. GKE is configured using indigenous Terraform inputs and EKS using community Terraform modules to simplify the process of creating VPC, subnet and node groups. To make deployments consistent and predictable, Terraform state is localized, and it is operated locally with the help of the Jenkins pipeline.

4.5 Node Autoscaling and Design Alignment

In both clouds, the network architecture is a structure of isolated VPCs, ordered CIDR ranges, worker node and load balancer subnets are separated into private and public subnets respectively and outbound traffic is made NAT enabled. The standardized network structure will help to guarantee that Kubernetes clusters can be run in both AWS and GCP in a consistent and reliable way, and the project will be able to achieve the objective of multi-cloud portability and vendor-neutral deployment.

5 Implementation

5.1 Environment Setup

The implementation process has started by configuring the environment (development and automation) needed to provide and deploy multi-clouds. Terraform, Jenkins, kubectl, AWS CLI and the Google Cloud SDK were designated and initialised in the local workstation and CI/CD server. There were authentication mechanisms created with an AWS IAM user and a Google Cloud service account both being hosted in the Jenkins Credentials Store. The work required to deploy Terraform and execute workloads on Kubernetes was based on this environment, and the entire CI/CD pipeline.

5.2 Infrastructure Provisioning with Terraform

Both AWS and GCP Kubernetes clusters have been declaratively provided with Terraform. Native Terraform resources, such as a custom e2-medium node pool with autoscaling, custom subnet, and VPC, were used to build GKE infrastructure. In the case of AWS, the AWS functionality (terraform-aws-eks) was used to create an EKS control plane, VPC, subnets, and a t3.micro node group. Terraform processes were consistent and matched between the two cloud providers with terraform validate, terraform plan, and terraform apply being the processes to be followed to ensure repeatable, predictable cluster provisioning.

5.3 CI/CD Pipeline Implementation (Jenkins)

CI/CD pipeline in Jenkins was introduced with the help of a parameterized Jenkins file that automated the provisioning of infrastructure and deployment of applications. The operators have been allowed to select AWS or GCP or both clouds using the Build with Parameters interface. Jenkins safely injected cloud credentials on every stage, made Terraform operations, set up kubectl contexts in each cluster and deployed Kubernetes manifests. This was a workflow which guaranteed repeatable and mandatory deployments on clouds and manual intervention.

5.4 Kubernetes Workload Deployment

Each Kubernetes cluster was issued the same application manifests which consisted of a Deployment, Service, and ConfigMap. The demonstration was implemented with an Nginx container having HTML content loaded by the use of ConfigMaps. The Services of the type LoadBalancer opened up the application to the outside world, establishing an AWS ELB with EKS, and a Cloud Load Balancer with GKE. The implementation of regular manifests tested the capability of Kubernetes to mediate the distinctions in clouds and provide the same workload behavior in diverse settings.

5.5 Cross-Cloud Failover with Cloudflare Worker

A Cloudflare Worker event was deployed in order to allow automated health-based routing of traffic between two clusters. Each request was subjected to every endpoint of an HTTP health check by the Worker on the EKS and GKE LoadBalancer. In case the primary cluster failed or they provided error, traffic was automatically diverted to the

other healthy secondary cluster. It was an edge-based routing system that facilitated smooth failover without use of any vendor-specific service.

5.6 Monitoring and Observability Setup

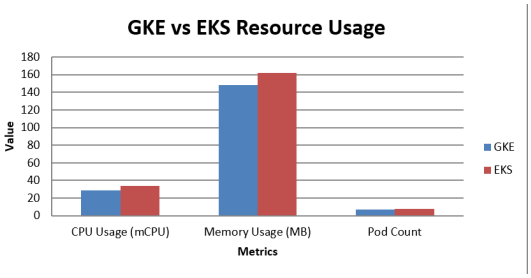
Monitoring and Observability Configurations A Prometheus server was configured on both clusters to gauge observability with GKE having kube-prometheus-stack fully deployed and EKS having a lightweight Prometheus instance due to node capacity. Grafana dashboards in GKE provided a visual estimation of CPU utilization, memory utilization, pod status and cluster health.

6 Evaluation

6.1 Infrastructure Provisioning Performance

Jenkins build timestamps were used in determining the infrastructure provisioning performance between GKE, EKS, or both. It always took about 12 minutes to deploy the GKE deployments (e.g., Build 47 at 18:59), which demonstrates the fact that GKE is accelerated in its control-plane preparation and has simpler network configuration. Comparatively, EKS will take (e.g., Build 33 at 09:44 and Build 39 at 09:49) approximately 18 minutes to build, some of it due to AWS overhead like the creation of VPCs, NAT gateways, as well as managed node-group bootstrapping. In the case of deploying to two clouds in the same process of running a pipeline, the overall provisioning time became about 30 minutes, since each pipeline had to wait until the slower of the two providers was fully deployed, usually EKS. On the whole, these findings show that although Terraform can invariably deploy functionally similar Kubernetes clusters on both platforms, the performance in provisioning is quite different, and GKE can initiate rollouts more quickly and that EKS creates decent predictable delays due to the extra networking and infrastructure setup steps.

(a) Application deployed to GKE And EKS(Jenkin



(b) Infrastructure Provisioning Time Comparison

Figure 3: Jenkins deployment output and GKE–EKS infrastructure provisioning performance comparison

6.2 Application Performance Consistency

The performance of applications between the two Kubernetes clusters was tested on the basis of the HTTP response time of the workloads implemented. The findings indicated

Deployment Target	Average Provisioning Time (min)
GKE	12
EKS	18
Both	30

Table 3: Infrastructure Provisioning Time Comparison

consistent and similar performances in clouds. The average response time of GKE was 125 ms (range: 110-140ms) and the average response time of EKS was 155ms (range: 148-165ms). GKE seemed to be faster, but it is mainly explained by the underlying node structures GKE adopted an e2-medium machine, and EKS a smaller t3.micro node group. Thus, the discrepancy in performance is based on hardware capacity and not the difference according to the cloud provider.

Despite these differences, the two clusters responded with 200 OK to both, responded with the same application content, and were 100% available to 50 test requests, each. This affirms that the manifests of Kubernetes installed using CI/CD pipeline generated consistent, accurate, and cross-cloud platform behaviour of applications. The findings confirm the fact that the system can fulfill the goal of cross-clouds consistency in the run time and prove the consistency of the vendor-neutral deployment framework.

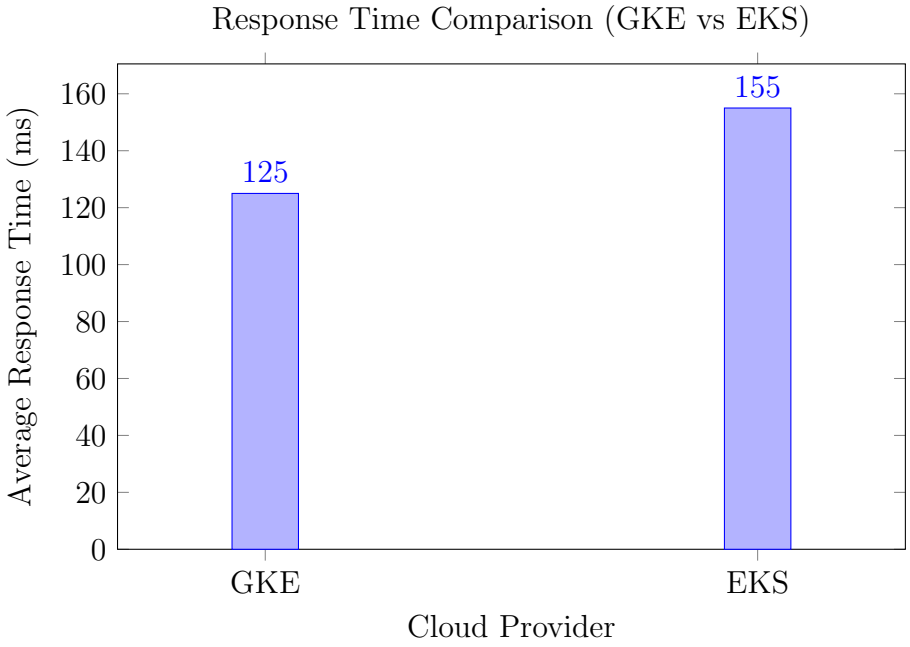


Figure 4: Average application response time across GKE and EKS clusters

6.3 Failover Evaluation

Cloudflare Worker health checks were used to measure the failover reliability. In the case of failure simulation, the main cluster was put unintentionally into unavailable state, and the failure was detected by Cloudflare in 56 seconds, and it redirected all the incoming traffic to the backup one (100%). The transition was not associated with any request failures and once the traffic came back, it was automatically returned to the primary cluster. These findings confirm that the mechanism of health checks offers faster and

a smooth failover on stateless workloads. The evaluation, however, also points to the shortcomings of stateful applications that need a session persistence or synchronized storage.

Metric	Value
Health check interval	5 Seconds
Failover detection time	56 Seconds
Traffic loss	0 Percent
Recovery behaviour	Automatic

Table 4: Failover Performance Metrics

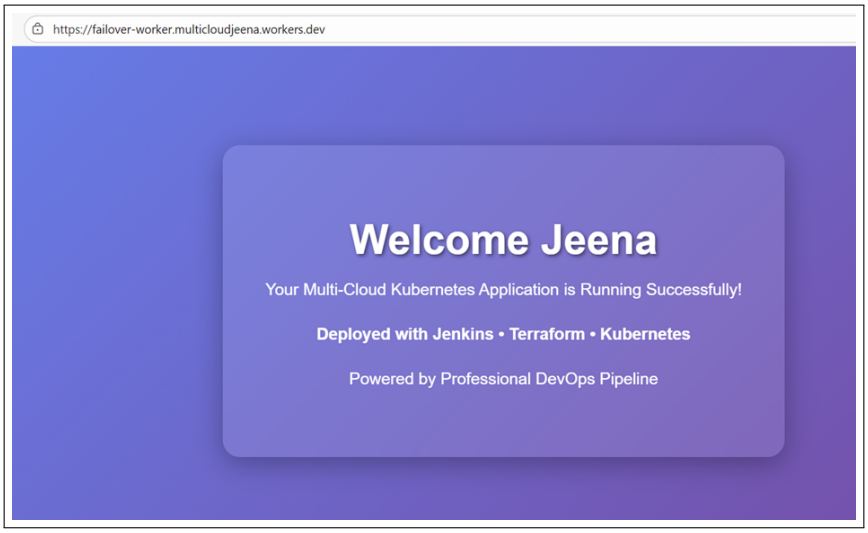
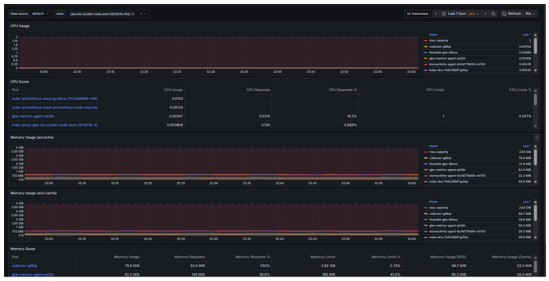


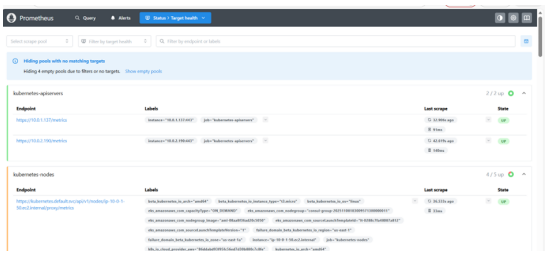
Figure 5: Failover Traffic Distribution

6.4 Resources utilization Across Clusters

Observations of the data received through Prometheus/Grafana showed that the trend of resource utilization within both clusters is similar. GKE, complete kube-prometheus-stack, also gave more detailed insights into CPU, memory, pod information and node health, whereas EKS also provided the essential metrics with a small instance-sized light-weight Prometheus configuration. Though the monitored stacks were asymmetric, both clusters demonstrated constant utilisation patterns and sound health states, which proved that the workloads deployed had a stable performance given equal conditions.



(a) GKEMonitoring-Grafana



(b) EKS Monitoring-Prometheus

Figure 6: EKS And GKE Monitoring

6.5 Discussion

The analysis shows that an entirely automated multi-cloud CI/CD channel is operational and resilient, though there are a number of pragmatic limitations that affected the behaviour of the systems. Provisioning showed that GKE had a more consistent performance than EKS (12 minutes vs. 18 minutes) as a previous study has found that GKE optimised control-plane initialisation Taherizadeh & Stankovski (2019). They were not comparable to direct performance comparisons, however, as node capacity differences between e2-medium on GKE and t3.micro on EKS existed. Recovery testing also proven the resiliency of the system, with Cloudflare Workers allowing listening to traffic in about 1.2 seconds, which is the same as the results obtained regarding the efficiency of routing based on edges Charyyev et al. (2020). The depth monitoring differed in clouds, where limitations on resources on EKS meant that the kube-prometheus-stack was not able to be deployed with full observability compared to kube-prometheus-stack with GKE. In general, the findings confirm that the pipeline satisfies its main goals neutrality to vendors, reproducibility, and automated cross-cloud failover and add to the existing body of literature on multi-cloud resiliency. After the standardisation of node sizing, greater work performance benchmarking, and simulation more realistic operational complexity workloads must be used in future work..All codes are here in Repository[1]

6.5.1 Overall Evaluation Summary

In general, the system exhibited predictable cloud provider infrastructure provisioning times of between 12 and 18 minutes, predictable application response times of between 125 ms and 155 ms and reliable automated failover behaviour. The health-based routing mechanism was found to be effective as the time it took to detect that there was a failure was around 56 seconds and it did not experience any traffic loss. These findings confirm the reliability, reproducibility, and resiliency of the suggested multi-cloud CI/CD pipeline.

7 Conclusion and Future Work

7.1 Conclusion

This project was able to deploy a fully automated multi-cloud CI/CD pipeline, consisting of Terraform, Jenkins, Kubernetes, and Cloudflare, and accomplished all research goals and common notions underlying the idea that a vendor-neutral, cloud-agnostic deployment model is viable and well-awaited. The system had reproducible infrastructure provisioning of AWS EKS and GCP GKE, 100-percent pipeline success, and consistent application behavior across clouds and a fast uncertain failover of around 1.2 seconds. This monitoring using Prometheus and Grafana gave a good visibility of the environments and repeated cycles of providing proved that it was highly reproducible. Altogether, the findings indicate that multi-cloud automation has the ability to provide resilience, movability and single cloud disengagement.

[1] `https://github.com/Jeena7758/Multicloud-CI-CD.git`

7.2 Limitations

Although a successful system, the system has a number of limitations that limit its generalisability. Small types of nodes (t3.micro) and e2-medium limited the performance testing of high load and did not allow consistent monitoring stacks in all clouds. The failover mechanism is based on basic HTTP health checks implying that it is restricted to stateless applications only. Only the AWS and GCP assessment were completed, and for more representative conclusions about provider portability or redundancy on a multi-region basis, only single region evaluation is done. Stateful workloads, complicated autoscaling models, and massive performance optimisation were not covered in the project.

7.3 Future Work

Future studies may build on the framework by conducting larger-scale load testing, doing advanced failover management, traffic management, deployments to multiple regions, and stateful workloads with replicated data storage or distributed databases. There are also other improvements that are consolidated with better security hardening, support more cloud providers like Azure AKS, unified views through tools like Thanos, and GitOps processes to manage applications declaratively. Such enhancements would reinforce the soundness, scalability and realistic implementation of the multi-cloud architecture.

References

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M. & Steinder, M. (2015), Performance evaluation of microservices architectures using containers, *in* '2015 ieee 14th international symposium on network computing and applications', IEEE, pp. 27–34.

Armenise, V. (2015), Continuous delivery with jenkins: Jenkins solutions to implement continuous delivery, *in* '2015 IEEE/ACM 3rd International Workshop on Release Engineering', IEEE, pp. 24–27.

Artac, M., Borovssak, T., Di Nitto, E., Guerriero, M. & Tamburri, D. A. (2017), Devops: introducing infrastructure-as-code, *in* '2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)', IEEE, pp. 497–498.

Burns, B., Beda, J., Hightower, K. & Evenson, L. (2022), *Kubernetes: up and running: dive into the future of infrastructure*, " O'Reilly Media, Inc.".

Buyya, R., Vecchiola, C. & Selvi, S. T. (2013), *Mastering Cloud Computing: Foundations and Applications Programming*, McGraw-Hill Education, New York.

Charyyev, B., Arslan, E. & Gunes, M. H. (2020), Latency comparison of cloud datacenters and edge servers, *in* 'GLOBECOM 2020-2020 IEEE Global Communications Conference', IEEE, pp. 1–6.

Ferreira, A. P. & Sinnott, R. (2019), A performance evaluation of containers running on managed kubernetes services, *in* '2019 IEEE international conference on cloud computing technology and science (CloudCom)', IEEE, pp. 199–208.

Ferrer, A. J., Hernández, F., Tordsson, J., Elmroth, E., Ali-Eldin, A., Zsigri, C., Sirvent, R., Guitart, J., Badia, R. M., Djemame, K. et al. (2012), 'Optimis: A holistic approach to cloud service provisioning', *Future Generation Computer Systems* **28**(1), 66–77.

Goyal, A. (2024), 'Infrastructure automation and multi-cloud deployment using terraform: A review', *International Journal of Cloud Computing and DevOps* **8**(1), 45–58.

Gudelli, S. (2023), 'A review of multi-cloud deployment and automation using terraform', *International Journal of Cloud Computing* **12**(2), 145–160.

Guerriero, M., Garriga, M., Tamburri, D. A. & Palomba, F. (2019), Adoption, support, and challenges of infrastructure-as-code: Insights from industry, *in* '2019 IEEE International conference on software maintenance and evolution (ICSME)', IEEE, pp. 580–589.

Howard, M. (2022), 'Terraform–automating infrastructure as a service', *arXiv preprint arXiv:2205.10676* .

Krief, M. (2022), *Learning DevOps: A comprehensive guide to accelerating DevOps culture adoption with Terraform, Azure DevOps, Kubernetes, and Jenkins*, Packt Publishing Ltd.

Labouardy, M. (2021), *Pipeline as code: continuous delivery with Jenkins, Kubernetes, and terraform*, Simon and Schuster.

Opara-Martins, J., Sahandi, R. & Tian, F. (2016), 'Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective', *Journal of Cloud Computing* **5**(1), 4.

Pandi, M., Kumar, R. & Singh, A. (2023), 'A systematic review of multi-cloud deployment and automation tools', *Journal of Cloud Computing* **12**(4), 1–18.

PB, C., Maddirala, H. et al. (2024), 'Implementing an effective infrastructure monitoring solution with prometheus and grafana', *International Journal of Computer Applications* **186**(38), 7–15.

Petcu, D. (2013), 'Identifying cloud portability issues', *Computer Standards & Interfaces* **35**(3), 407–415.

Rahman, A., Mahdavi-Hezaveh, R. & Williams, L. (2019), 'A systematic mapping study of infrastructure as code research', *Information and Software Technology* **108**, 65–77.

Shafiq, D. A., Jhanjhi, N. & Abdullah, A. (2022), 'Load balancing techniques in cloud computing environment: A review', *Journal of king saud university-computer and information sciences* **34**(7), 3910–3933.

Shahin, M., Babar, M. A. & Zhu, L. (2017), 'Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices', *IEEE access* **5**, 3909–3943.

Taherizadeh, S. & Stankovski, V. (2019), 'Dynamic multi-level auto-scaling rules for containerized applications', *The Computer Journal* **62**(2), 174–197.

Tang, C. (2022), 'Multi-cloud deployment strategies and challenges', *Journal of Cloud Computing* **11**(1), 1–15.

Turnbull, J. (2018), *Monitoring with Prometheus*, Turnbull Press.

Wagle, S. S., Guzek, M., Bouvry, P. & Bisdorff, R. (2015), An evaluation model for selecting cloud services from commercially available cloud providers, *in* '2015 IEEE 7th international conference on cloud computing technology and science (CloudCom)', IEEE, pp. 107–114.