ABSTRACT

Our project shows designing and implementing a zero-downtime and highly available application deployment architecture using Kubernetes on Amazon EKS and AWS cloud services. The objective is to ensure uninterrupted application availability during deployments, traffic spikes, application failures, and regional outages.

The application is containerized using Docker and deployed on Amazon EKS for orchestration, scalability, and automated recovery. Kubernetes rolling update strategy along with readiness probes is used to achieve seamless application version updates without affecting users. Horizontal Pod Autoscaler (HPA) dynamically increases or decreases pods based on CPU utilization to handle changing traffic conditions efficiently.

To improve application reliability, liveness probes are implemented to detect unresponsive Node.js containers caused by event loop blocking. Kubernetes automatically restarts unhealthy containers, enabling self-healing without manual intervention.

For disaster recovery, a warm standby multi-region architecture is implemented using Amazon Route 53 health checks and automated DNS failover. When the primary region becomes unavailable, traffic is redirected automatically to the secondary region with minimal downtime.

Prometheus, Grafana, and CloudWatch are used for monitoring system health, resource usage, autoscaling behavior, container restarts, and failover status. The project demonstrates resilient cloud-native architecture suitable for modern production environments.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

While working with containerised applications, one practical issue that comes up frequently is that failures are not always visible at first glance. A pod may appear to be running, but the application inside may stop responding. At times, traffic increases suddenly and the existing resources are not sufficient. In other cases, an entire region may become unavailable. In all these situations, the main requirement is not just deployment but ensuring that the application continues to serve users without interruption.

We focused on designing and implementing a **zero-downtime, highly available architecture using Kubernetes on Amazon EKS along with AWS services**. We aimed to maintain continuous application availability even during different types of failures such as pod-level issues, application unresponsiveness, and regional outages. The system is designed by combining multiple mechanisms instead of relying on a single solution for availability.

Kubernetes is used as the core platform to manage containerised workloads. It provides capabilities such as deployment management, scaling, and self-healing. However, in real-world scenarios, these default features need to be extended with proper configurations to handle production-level challenges. Our priority is given to health checks, scaling behaviour, and traffic routing to ensure stable operation.

To handle varying load conditions, **Horizontal Pod Autoscaler (HPA)** is implemented. HPA automatically increases or decreases the number of running pods based on CPU utilisation. When the load increases, additional pods are created to distribute the traffic. Similarly, when the load decreases, unnecessary pods are removed.

The system also makes use of **readiness probes**, which determine whether a pod is ready to handle incoming requests. A pod may be in a running state but still not capable of serving traffic, especially during

startup or temporary overload conditions. Readiness probes ensure that such pods are excluded from the service endpoints until they become stable.

In addition to readiness checks, **liveness probes** are implemented to detect application-level failures. This becomes important in cases where the application becomes unresponsive without crashing. For example, in Node.js applications, CPU-intensive operations can block the event loop, causing the application to stop responding while the container continues to run. In such situations, readiness probes only prevent traffic routing but do not recover the application. Liveness probes address this by continuously monitoring the application health. If failures occur repeatedly, Kubernetes restarts the container.

At the networking level, **Route 53** is used to implement DNS-based failover. Health checks are configured to monitor the application endpoints. When the primary region is healthy, all traffic is routed to it. If the health checks fail, Route 53 automatically redirects traffic to the secondary region. The use of low TTL values helps in reducing the delay during DNS switching.
The disaster recovery approach followed in this project is based on a **Warm Standby model**. In this setup, the primary AWS region handles all user traffic, while the secondary region runs with limited resources and remains ready to take over. During a failure in the primary region, the secondary region is activated to serve requests.

Traffic distribution within each region is managed using **Elastic Load Balancer (ELB)** integrated with Kubernetes ingress. The load balancer receives incoming requests and forwards them to the appropriate Kubernetes services, which then route traffic to healthy pods.

For monitoring, **CloudWatch** is used to track metrics such as health check status and endpoint availability. These metrics help in observing system behaviour during normal operation as well as during failure scenarios. It becomes easier to verify whether failover and recovery mechanisms are functioning as expected.

CHAPTER 2

LITERATURE REVIEW

Ref No.	Paper Title	Author	Drawbacks / Limitations
1	ZERO-DOWNTIME DEPLOYMENTS: Analyzing Blue-Green & Canary Approaches in DevOps	Bruce William	Focuses on methodologies rather than HA measurements; no experiment results.
2	High Availability Solution for Cloud Applications	Wagdy Anis Aziz	Limited container focus; more cloud HA than deployment strategies.
3	Containerized Zero-Downtime Deployments in Full-Stack Systems	Kiran Kumar Pappula	Microservices only; needs extended experiments.
4	Evaluate Solutions for Achieving High Availability or Near Zero Downtime for Cloud Native Enterprise	Antra Malhotra	Focus on enterprise patterns; not strictly container-centric.
5	Migrating Enterprise Applications to AWS: Key Considerations and Strategies	Nagarjuna Putta, Imran Khan, Murali Mohana, Krishna Dandu	Does not measure quantitative performance metrics or specific migration results; strategic/qualitative focus only
6	A Docker-based Operation and Maintenance Method for New-Generation Command and Control Systems	Wei Yongyong	Docker has lower isolation than full VMs (trading performance for speed); relies on improving resource capacity to overcome current limitations.

1. Zero-Downtime Deployments *(Bruce William)*:
This paper explains blue-green and canary strategies for minimizing downtime. It focuses on deployment methodologies but lacks quantitative evaluation and real-world performance validation.

2. High Availability Solution for Cloud Applications *(Wagdy Anis Aziz)***:**
The study discusses cloud-based high availability architectures using redundancy and failover. It emphasises infrastructure-level reliability but gives limited attention to container orchestration and deployment-level strategies.

3. Containerized Zero-Downtime Deployments in Full-Stack Systems *(Kiran Kumar Pappula)***:**
This work explores zero-downtime deployment in containerised microservices environments. It focuses on full-stack systems but requires broader experimentation and validation across different workloads and architectures.

4. Evaluate Solutions for Achieving High Availability or Near Zero Downtime for Cloud Native Enterprise *(Antra Malhotra)***:**
The paper analyses enterprise-level HA patterns and strategies. It mainly focuses on architectural approaches rather than practical container-based implementations or Kubernetes-specific deployment techniques.

5. Migrating Enterprise Applications to AWS *(Nagarjuna Putta, Imran Khan, Murali Mohana, Krishna Dandu)***:**
This paper discusses migration strategies and design considerations for AWS adoption. It provides strategic insights but lacks detailed performance metrics or validation of migration outcomes in production environments.

6. A Docker-based Operation and Maintenance Method for New *(Wei Yongyong)***:**
The study presents Docker-based deployment for modern systems. It highlights operational benefits but notes limitations in isolation compared to VMs and dependency on resource scaling for performance improvement.

CHAPTER 3

SYSTEM ANALYSIS

The proposed system is designed to deliver high availability and near zero downtime for containerized applications using a cloud-native architecture built on Amazon Web Services (AWS) and Kubernetes (Amazon EKS). The system addresses critical operational challenges such as service interruption during deployments, inability to handle traffic spikes, and lack of reliable disaster recovery mechanisms.

3.1 Existing System Analysis

Traditional application deployment models typically rely on single-region infrastructure and manual deployment processes. These approaches suffer from several limitations:

- Downtime during updates due to service restarts or improper traffic handling
- Limited scalability, as resources are not dynamically adjusted based on load
- Single point of failure, where regional outages can bring down the entire application
- Manual disaster recovery, leading to increased recovery time and potential data loss

Such systems are not suitable for modern applications that require continuous availability and responsiveness.

3.2 Proposed System Overview

The proposed system leverages Amazon EKS to orchestrate containerized applications and integrates multiple AWS services to ensure resilience, scalability, and automated failover.

Core Components:

- **Amazon EKS (Elastic Kubernetes Service):** Manages container orchestration, deployments, and scaling across worker nodes.
- **Elastic Load Balancer (ELB):** Distributes incoming traffic and integrates with Kubernetes Ingress for routing.
- **Amazon Route 53:** Provides DNS-based routing with health checks for automatic failover between regions.
- **Amazon EC2 Worker Nodes:** Hosts Kubernetes pods and supports horizontal scaling.
- **IAM (Identity and Access Management):** Controls secure access between AWS services and Kubernetes clusters.

3.3 Functional Analysis

The system is designed to handle three key scenarios:

1. **Zero Downtime Deployment**

- Kubernetes Rolling Update strategy ensures gradual replacement of old pods with new ones
- Readiness probes allow traffic only to healthy pods
- Graceful termination (SIGTERM) ensures ongoing requests are completed before shutdown

2. **Handling Traffic Spikes**

- Metrics Server collects resource utilization data
- Horizontal Pod Autoscaler (HPA) dynamically scales pods based on CPU usage
- Ensures optimal performance during high load without manual intervention

3. **Disaster Recovery (Multi-Region)**

- Primary region hosts full-capacity infrastructure
- Secondary region maintains a warm standby cluster
- Route 53 health checks detect failure and automatically redirect

traffic
- Enables rapid recovery with minimal downtime
- The application is designed to be stateless, allowing seamless failover between regions without requiring data synchronization.

4. **Self-Healing Mechanism**
- Liveness probes detect unresponsive application states caused by issues such as event loop blocking.
- Kubernetes automatically restarts failed containers.
- Ensures application recovery without manual intervention.

3.4 Non-Functional Analysis

- **Availability**: Achieved through multi-region deployment and failover mechanisms
- **Scalability**: Enabled via HPA and cloud elasticity
- **Reliability**: Ensured through readiness and liveness probes, redundancy, and Kubernetes self-healing capabilities.
- **Security**: Managed through IAM roles, VPC isolation, and security groups
- **Performance**: Optimized using load balancing and dynamic scaling

3.5 System Constraints

- Disaster recovery uses warm standby, not active-active (slight failover delay)
- Scaling is based on CPU metrics only, not custom or predictive metrics
- Dependent on proper DNS configuration and health check accuracy

CHAPTER 4

SYSTEM DESIGN & ARCHITECTURE

4.1 Zero Downtime During Application/Version Update

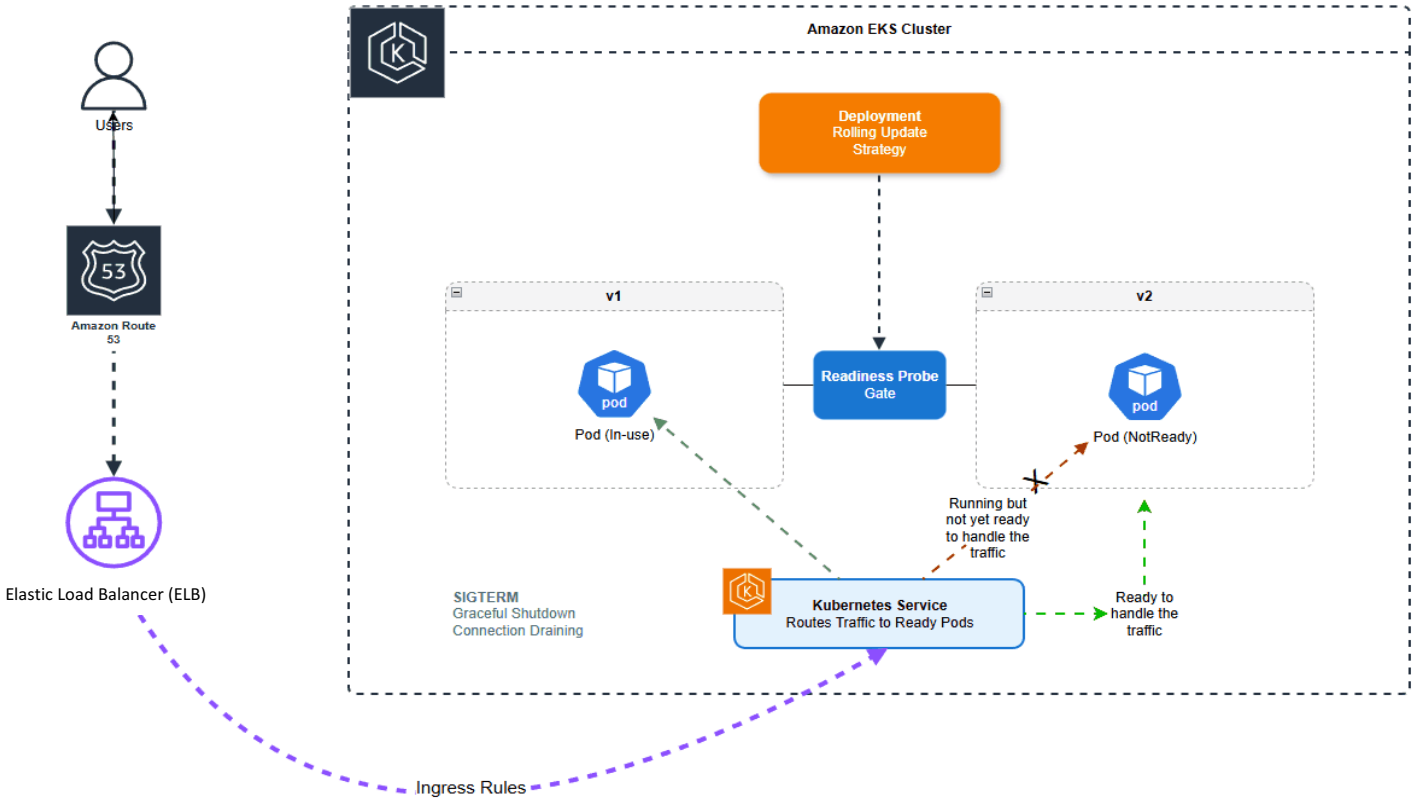


Fig. 4.1.1: **Zero Downtime During Application/Version Update**

In this architecture, the system ensures that application updates happen without affecting end users. When a new version (v2) is deployed in the Amazon EKS cluster, Kubernetes uses a rolling update strategy to gradually replace old pods (v1) with new ones. During this process, readiness probes play a key role by allowing traffic only to pods that are fully ready to serve requests. Even if the new pods are running, they will not receive traffic until they pass the readiness check. At the same time, old pods are not terminated immediately; they are gracefully shut down using SIGTERM, ensuring that ongoing requests are completed without interruption.

Additionally, liveness probes ensure that any unresponsive containers are automatically restarted, improving application reliability.

Example:
Consider an e-commerce website updating its payment service. While the new version is being deployed, users can still complete transactions using the older pods. Once the new pods are fully ready, traffic is smoothly shifted, ensuring zero downtime and a seamless user experience.

4.2 Handling High Traffic with Auto Scaling (HPA)

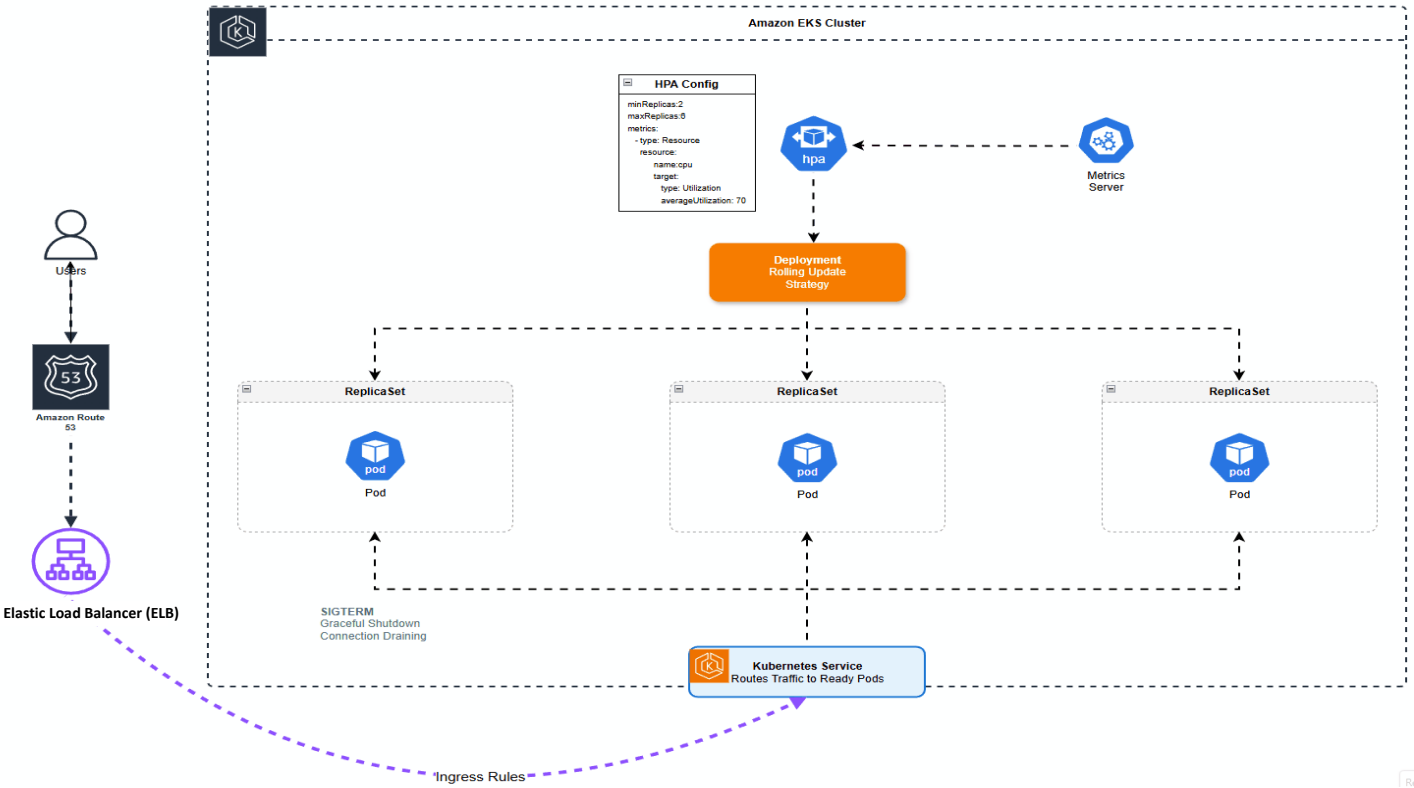


Fig. 4.2.1: **Handling High Traffic with Auto Scaling (HPA)**

In this architecture, the system automatically handles sudden increases in user traffic without affecting performance. When users send requests through Route 53 and the Elastic Load Balancer, traffic reaches the Kubernetes service, which distributes it across available pods. The Metrics Server continuously monitors CPU usage of these pods. Based on this data, the Horizontal Pod Autoscaler (HPA) dynamically increases or decreases the number of pods. During high load, new pods are created and added to the service, ensuring load is balanced properly. When traffic reduces, extra pods are terminated to optimize resource usage. The rolling update strategy and graceful

shutdown ensure that scaling happens without interrupting ongoing requests.

Example:
Consider a ticket booking website during a flash sale. As thousands of users access the platform simultaneously, CPU usage increases. HPA automatically scales up pods to handle the load, preventing slowdowns or crashes, and scales down later when traffic normalizes.

4.3 Liveness Probe-Based Self-Healing

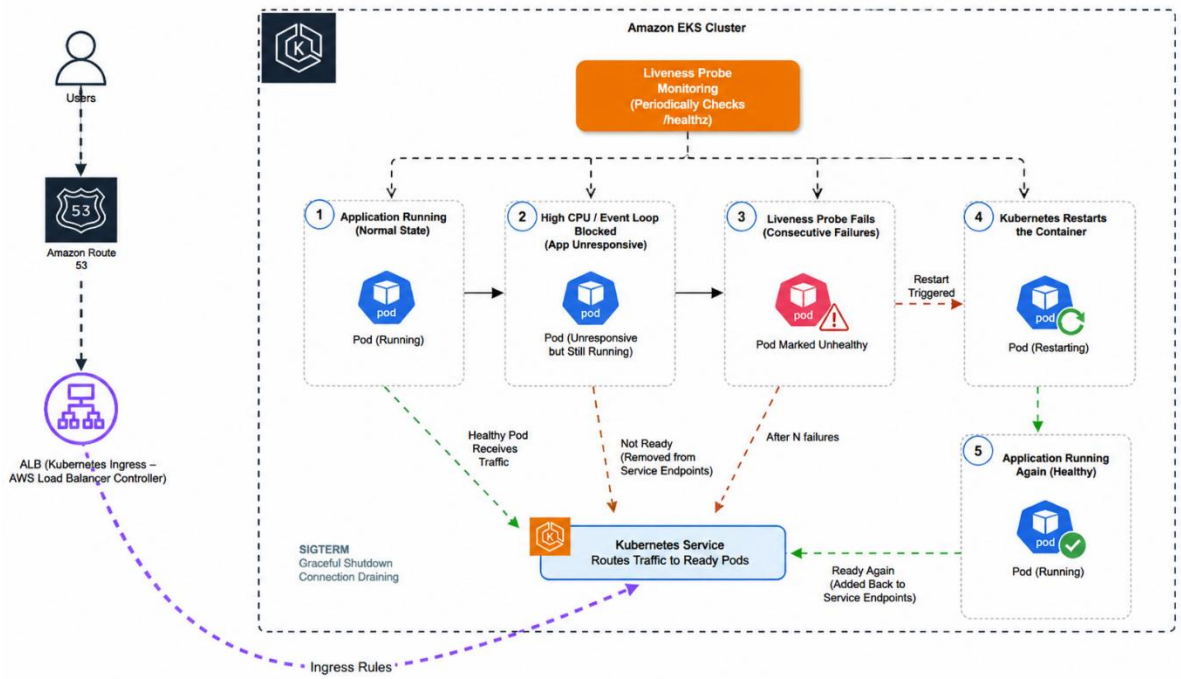


Fig. 4.3.1: **Liveness probe architecture for detecting unresponsive containers and automatically restarting them to ensure application self-healing in Kubernetes**

The diagram shows the working of the liveness probe in the Kubernetes deployment. The probe periodically checks the application health via the /healthz endpoint.

Under normal conditions, the application responds successfully and the pod continues running. However, CPU-intensive operations in Node.js can block the event loop, making the application unresponsive even though the container is still alive.

If the liveness probe fails repeatedly beyond the threshold, Kubernetes marks the container as unhealthy and restarts it. This creates a fresh instance, restoring normal application behaviour.

During this time, traffic is routed only to healthy pods. Once the restarted pod becomes stable and passes readiness checks, it is added back to the service.

This ensures automatic recovery, improving system availability and resilience without manual intervention.

4.4 Disaster Recovery with Multi-Region Failover

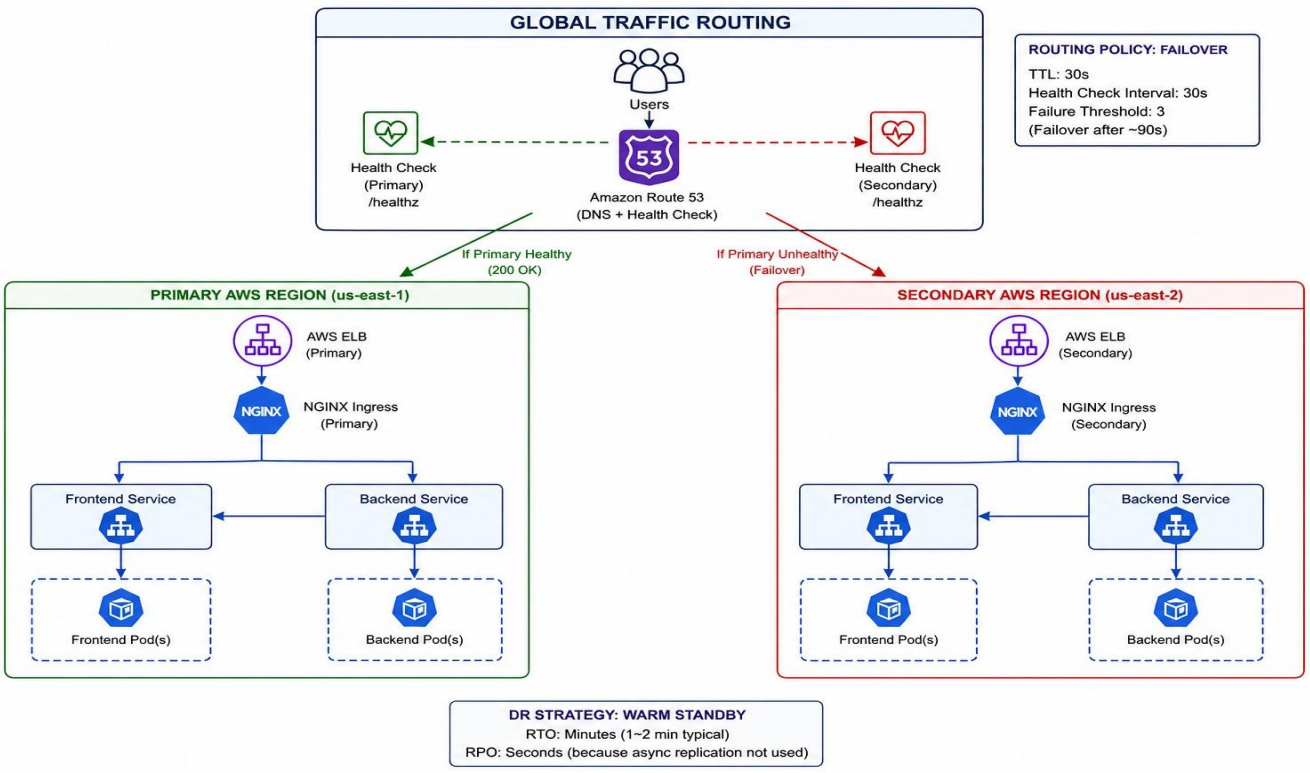


Fig. 4.4.1: **Multi-region failover architecture using Route 53 health checks and warm standby EKS clusters**

In this architecture, disaster recovery is implemented using a multi-region failover strategy with a warm standby Kubernetes cluster. The primary AWS region handles all incoming traffic under normal conditions, while the secondary region maintains a pre-configured standby EKS cluster with application components deployed.

Amazon Route 53 continuously performs health checks on the primary region's load balancer endpoint (/healthz). If the primary region becomes unhealthy, Route 53 automatically redirects traffic to the secondary region. The secondary cluster, already running with frontend and backend services, immediately starts serving user requests.

This approach eliminates the need for infrastructure provisioning during failure and ensures minimal recovery time with near-zero downtime.

Example:
Consider a banking application running in one region. If that region goes down due to a network outage, Route 53 quickly shifts traffic to the secondary region. Users can continue transactions without noticing the backend failure, ensuring business continuity.

CHAPTER 5

SYSTEM REQUIREMENTS

The proposed system requires the following AWS services and cloud components to implement zero-downtime, highly available architecture:

5.1 Core AWS Services

- **Amazon EKS (Elastic Kubernetes Service):** Used to manage and orchestrate containerized applications with support for rolling updates and auto-scaling.
- **Amazon EC2 (Worker Nodes):** Provides compute resources to run Kubernetes pods inside the EKS cluster.
- **Amazon VPC (Virtual Private Cloud):** Defines the network architecture, including public and private subnets, routing, and isolation.

5.2 Networking & Traffic Management

- **Elastic Load Balancer (ELB):** Distributes incoming traffic across Kubernetes services and integrates with Ingress.
- **AWS Load Balancer Controller:** Connects Kubernetes Ingress resources with ELB for dynamic traffic routing.
- **Amazon Route 53:** Handles DNS management, health checks, and automatic failover between regions.

5.3 Security & Access Control

- **AWS IAM (Identity and Access Management):** Manages authentication and authorization for users, services, and Kubernetes access.
- **Security Groups:** Acts as a virtual firewall controlling inbound and outbound traffic.

5.4 Monitoring & Scaling

- **Metrics Server:** Collects resource utilization data required for auto-scaling.
- **CloudWatch:** Used for monitoring system health and DNS failover conditions.
  - Route 53 health check metrics:
    - **HealthCheckStatus**: Indicates whether the endpoint is healthy or unhealthy
    - **HealthCheckPercentageHealthy**: Shows percentage of healthy responses across regions
  - These metrics help detect primary region failure and trigger automated failover to the secondary region.
- **Prometheus and Grafana:** Used for Kubernetes monitoring, pod health visualization, CPU utilization tracking, liveness probe monitoring, and autoscaling observation. Grafana dashboards provide real-time visibility into pod restarts, resource consumption, and application health during failure and recovery scenarios.

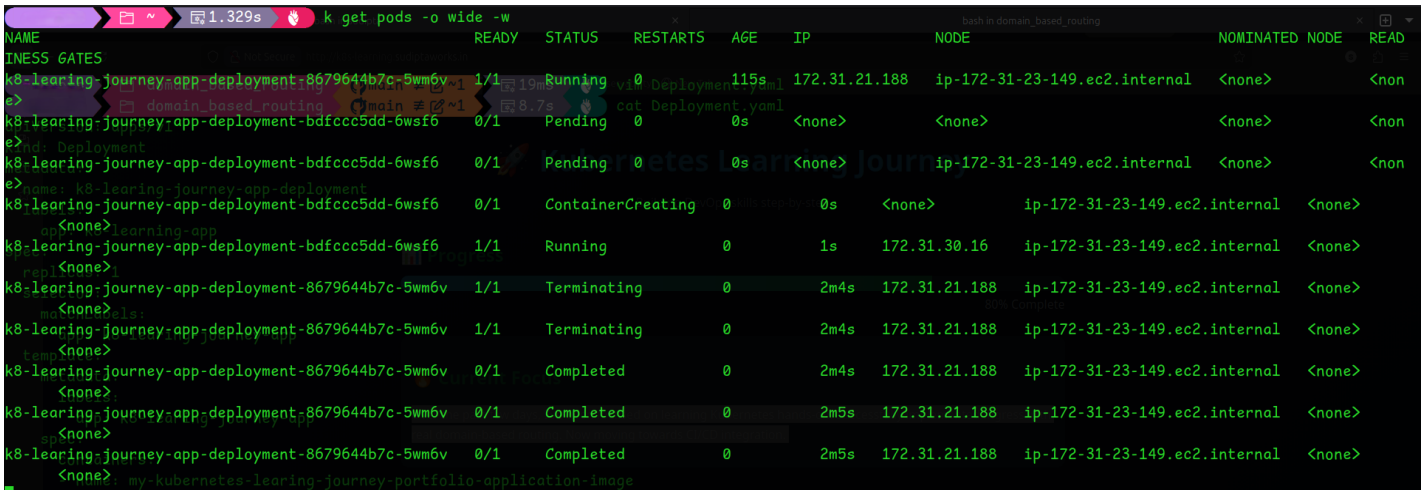
These AWS components together enable scalability, high availability, automated failover, and secure deployment, forming the backbone of the zero-downtime application architecture.

CHAPTER 6

IMPLEMENTATION

6.1 Rolling Update with Zero Downtime

- kubectl apply -f deployment.yaml



```
1.329s  k get pods -o wide -w
NAME                                                READY   STATUS              RESTARTS   AGE    IP              NODE                                NOMINATED NODE   READINESS GATES
k8-learing-journey-app-deployment-8679644b7c-5wm6v  1/1     Running             0          115s   172.31.21.188   ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-bdfccc5dd-6wsf6   0/1     Pending             0          0s     <none>          <none>                              <none>           <none>
k8-learing-journey-app-deployment-bdfccc5dd-6wsf6   0/1     Pending             0          0s     <none>          ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-bdfccc5dd-6wsf6   0/1     ContainerCreating   0          0s     <none>          ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-bdfccc5dd-6wsf6   1/1     Running             0          1s     172.31.30.16    ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-8679644b7c-5wm6v  1/1     Terminating         0          2m4s   172.31.21.188   ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-8679644b7c-5wm6v  1/1     Terminating         0          2m4s   172.31.21.188   ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-8679644b7c-5wm6v  0/1     Completed           0          2m4s   172.31.21.188   ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-8679644b7c-5wm6v  0/1     Completed           0          2m5s   172.31.21.188   ip-172-31-23-149.ec2.internal       <none>           <none>
k8-learing-journey-app-deployment-8679644b7c-5wm6v  0/1     Completed           0          2m5s   172.31.21.188   ip-172-31-23-149.ec2.internal       <none>           <none>
```

Fig 6.1.1: **Rolling Update with Zero Downtime**

This is showing the rolling update mechanism in the Kubernetes deployment. During an application update, new pods are created while old pods are still serving traffic. Kubernetes ensures that only ready pods receive traffic using readiness probes, and old pods are terminated gracefully after completing ongoing requests.

6.2 HPA Scaling During High Load

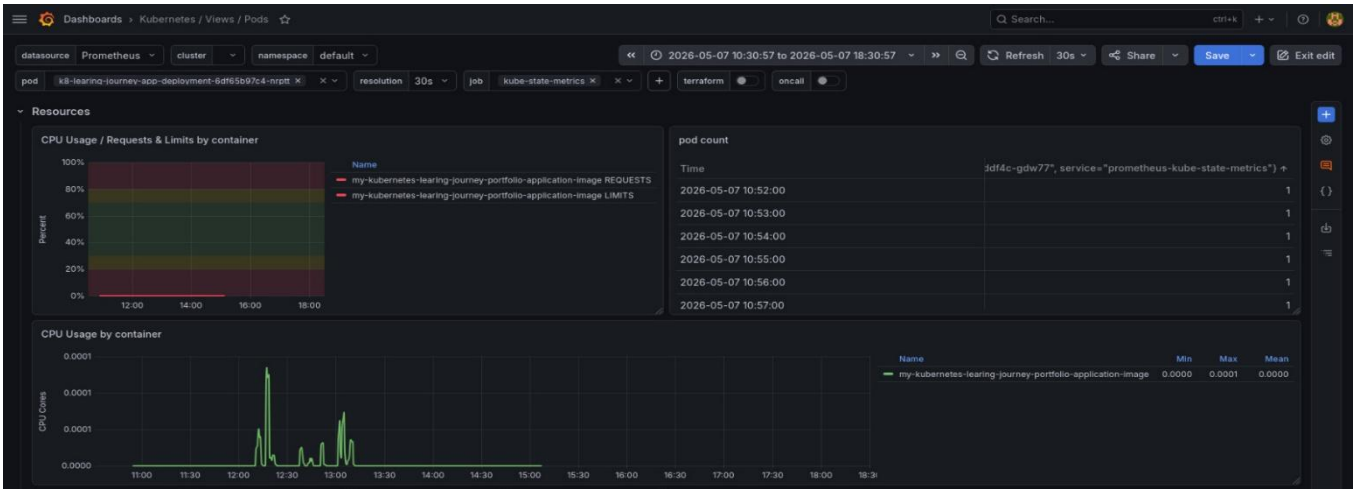


Fig 6.2.1: **Increased number of running pods after HPA scales the application**

The Grafana dashboard displays the normal CPU utilization of the application pod before load generation. Resource usage remains stable, indicating that the current number of pods is sufficient to handle the incoming traffic without triggering autoscaling.

- kubectl get hpa -w

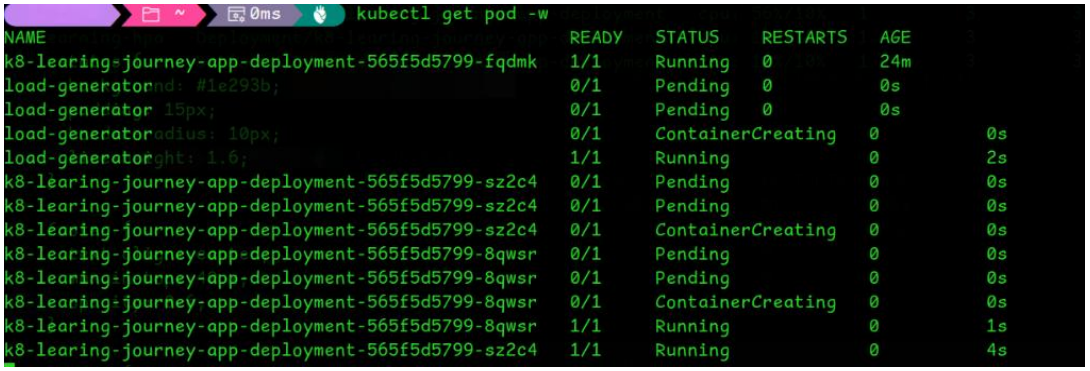


Fig 6.2.2: **Pods being created dynamically by HPA in response to increasing load**

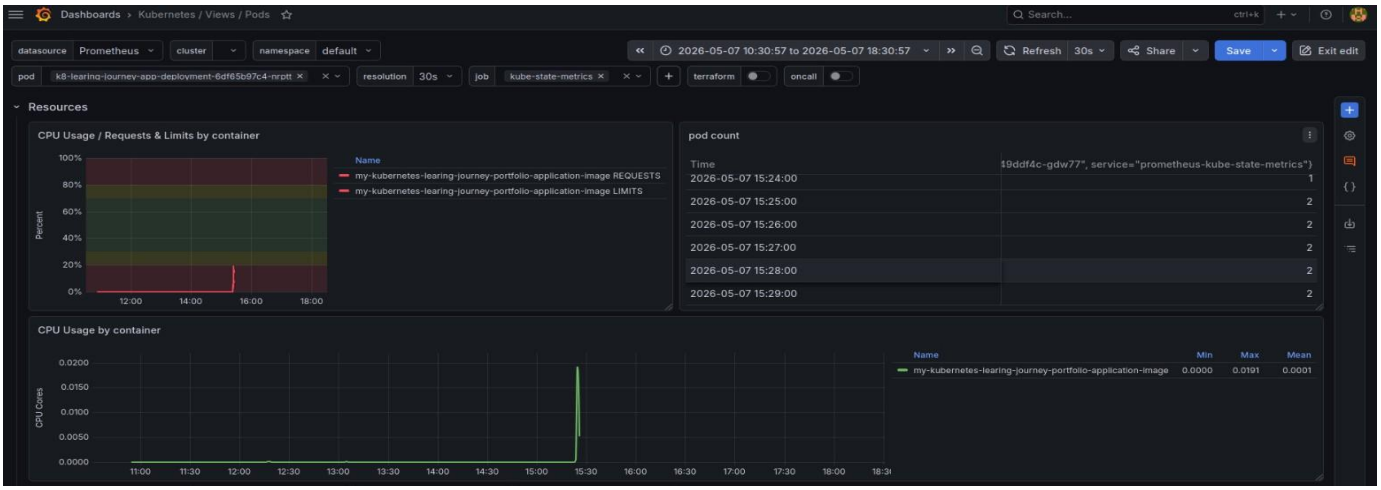


Fig 6.2.3: **Grafana dashboard showing increased CPU utilization during load generation**

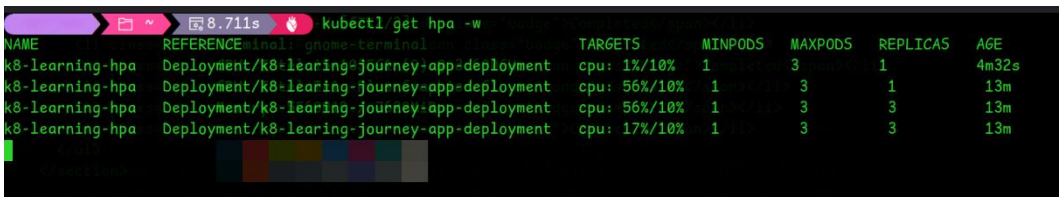


Fig 6.2.4: **Increased number of running pods after HPA scales the application under high CPU load**

This shows how the Horizontal Pod Autoscaler (HPA) responds to increased CPU usage. When load increases, HPA automatically scales the number of pods from the minimum to the maximum defined limit. New pods are created dynamically and start serving traffic.

6.3 HPA Scaling Down After Load Reduction

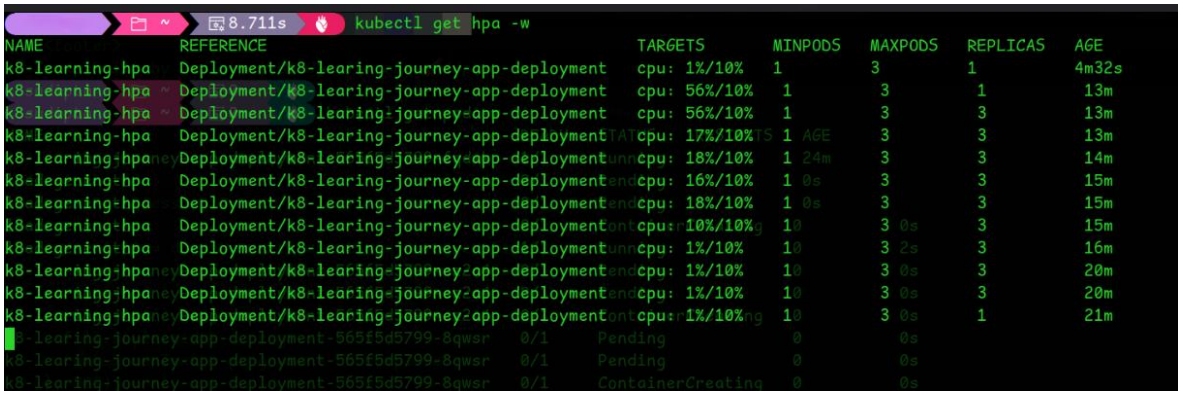


Fig 6.3.1: **HPA Scaling Down After Load Reduction**

Prometheus metrics collected from the Kubernetes cluster are visualized using Grafana dashboards. CPU utilization graphs clearly show the increase in resource usage during load generation, followed by automatic scaling performed by HPA.

This shows the scaling down process. When the system load decreases, HPA reduces the number of running pods to optimize resource usage. This ensures cost efficiency while maintaining performance.

6.4 Application Failure and Self-Healing using Liveness Probe

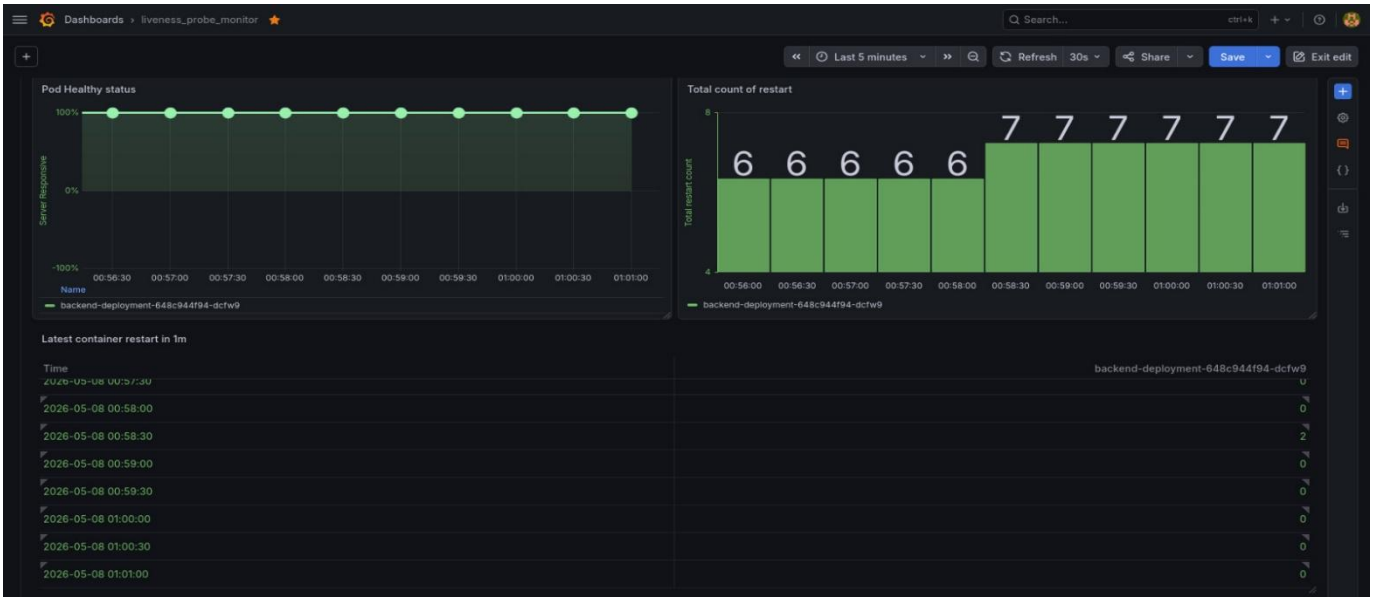


Fig 6.4.1: **Grafana dashboard showing healthy pod status before liveness probe failure**

Fig 6.4.2: **Grafana dashboard showing pod becoming unhealthy due to CPU-intensive event loop blocking**

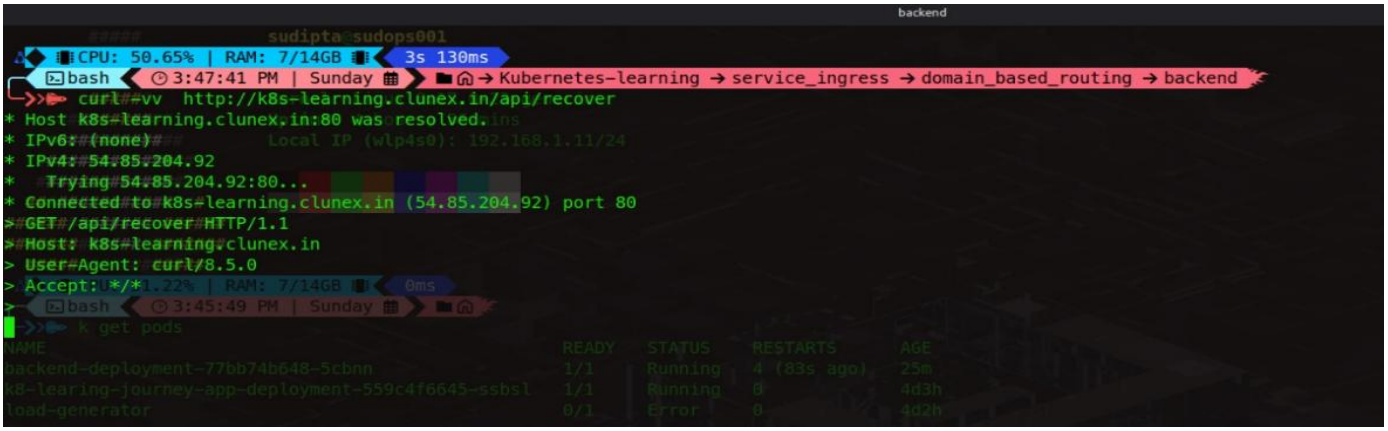


Fig 6.4.3: **Kubernetes pod entering unhealthy state after application becomes unresponsive**

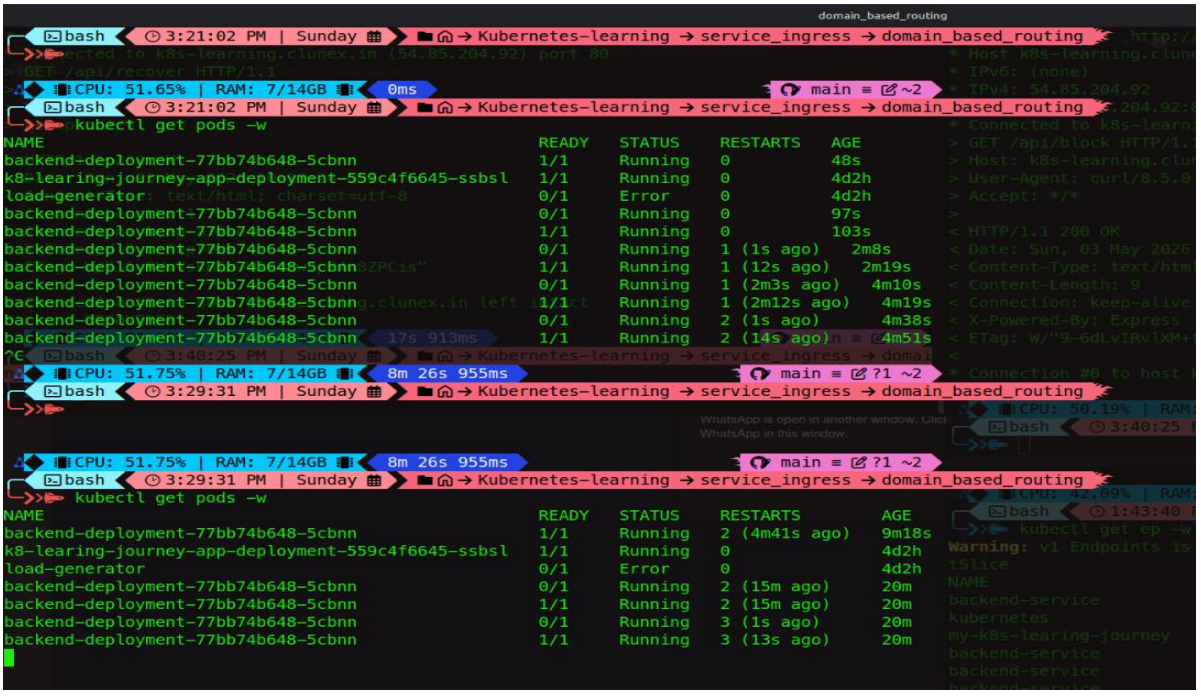


Fig 6.4.4: **Kubernetes liveness probe detecting failure and automatically restarting the unhealthy pod**

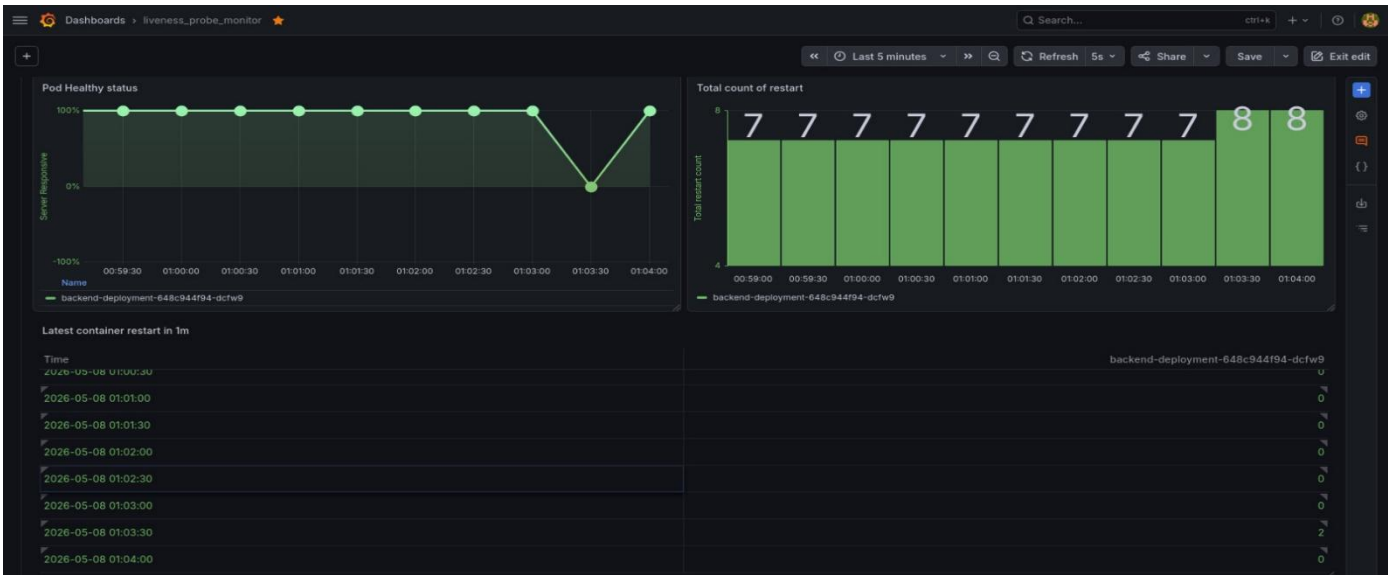


Fig 6.4.5: **Grafana dashboard showing automatic container restart after liveness probe failure**

This shows the application becoming unresponsive after triggering a CPU-intensive request. Although the container is still running, the Node.js event loop becomes blocked, preventing the application from serving requests successfully.

Readiness probes ensure that traffic is only routed to pods that are able to respond successfully. When the application becomes unresponsive, the readiness probe fails, and the pod is removed from service endpoints. However, readiness probes do not restart or recover the application.

Liveness probes continuously monitor the health of the container. When the application fails to respond within the defined threshold, Kubernetes automatically restarts the container. This restores the application by initiating a new process with a clean event loop, ensuring self-healing without manual intervention.

Grafana dashboards were used to monitor pod health status and container restart counts during the liveness probe experiment. The monitoring data confirmed automatic recovery behaviour after the application became unresponsive.

6.5 Disaster Recovery Failover to Secondary Region

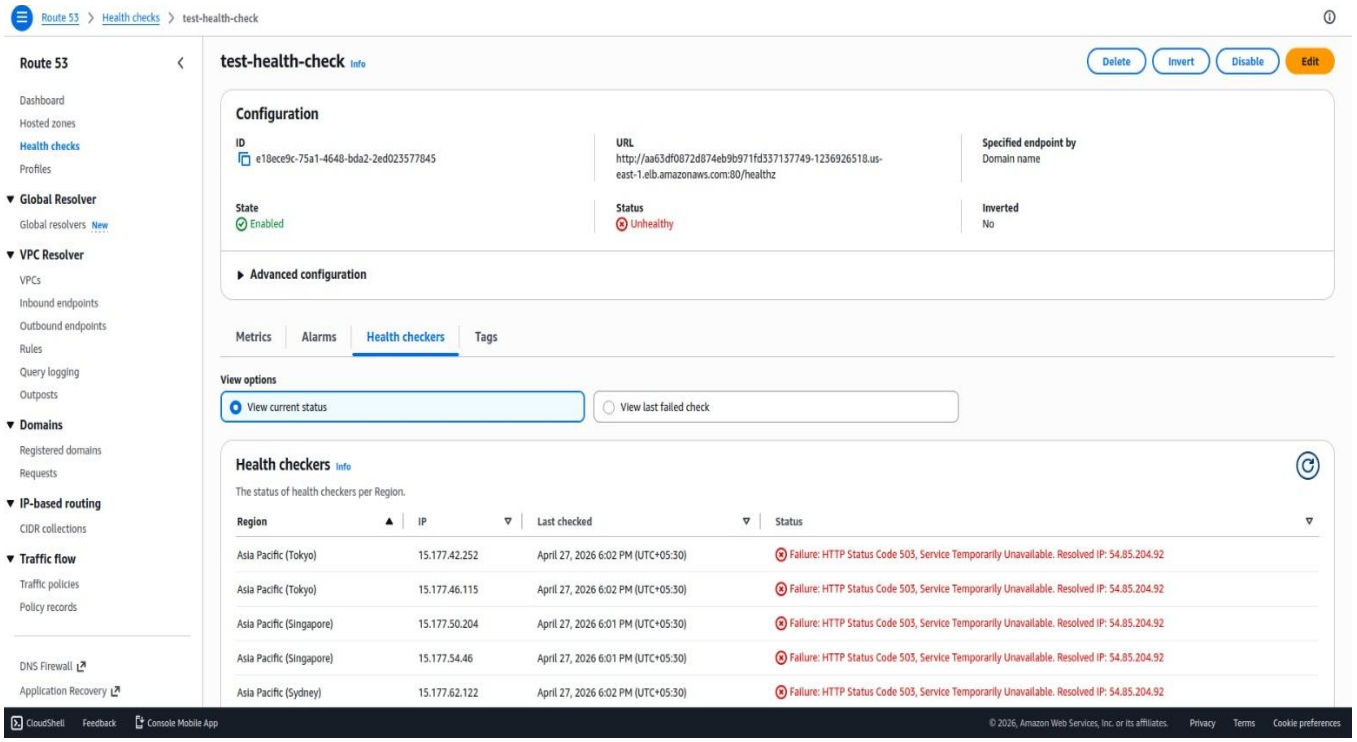


Fig 6.5.1: **System detecting primary region failure and initiating failover process**

- **Live Deployment Link:** http://k8s-learning.clunex.in
  (Note: The system is currently running on AWS infrastructure using free-tier credits. Due to cost constraints, this deployment is temporary and may be decommissioned after the evaluation phase)

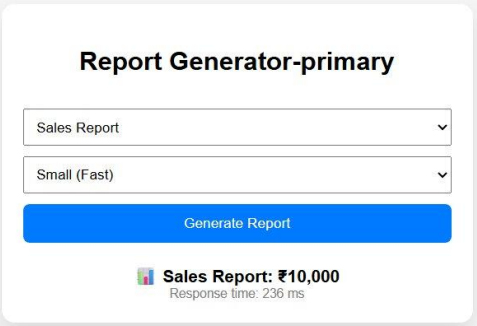


Fig 6.5.2: **Traffic successfully redirected to secondary region after Route 53 failover**

This shows the failover mechanism using Route 53. When the primary region becomes unhealthy, Route 53 automatically redirects traffic to the secondary region based on health checks. The secondary cluster,

already running in warm standby mode, immediately serves traffic without requiring additional provisioning.

In some cases, temporary errors may be observed before recovery mechanisms are triggered.

This system recovery can occur at two levels:

- pod-level self-healing using liveness probes
- infrastructure-level recovery using multi-region failover.

The liveness probe enables automatic recovery by restarting containers when application-level failures occur, even if the process is still running.

CHAPTER 7

RESULTS & SCREENSHOTS

7.1 Zero Downtime Deployment

- Shows pods updating without downtime

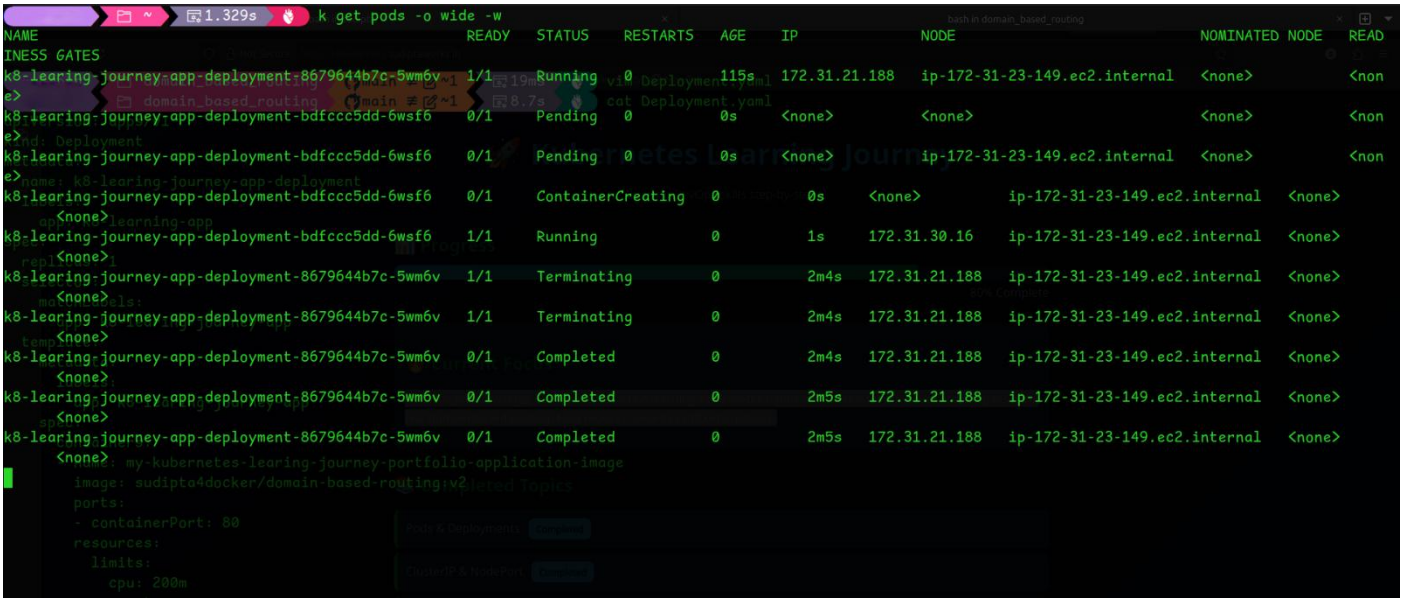


***Fig 7.1.1: P*ods updating without downtime**

7.2 HPA – Scaling Up

- Pods increasing

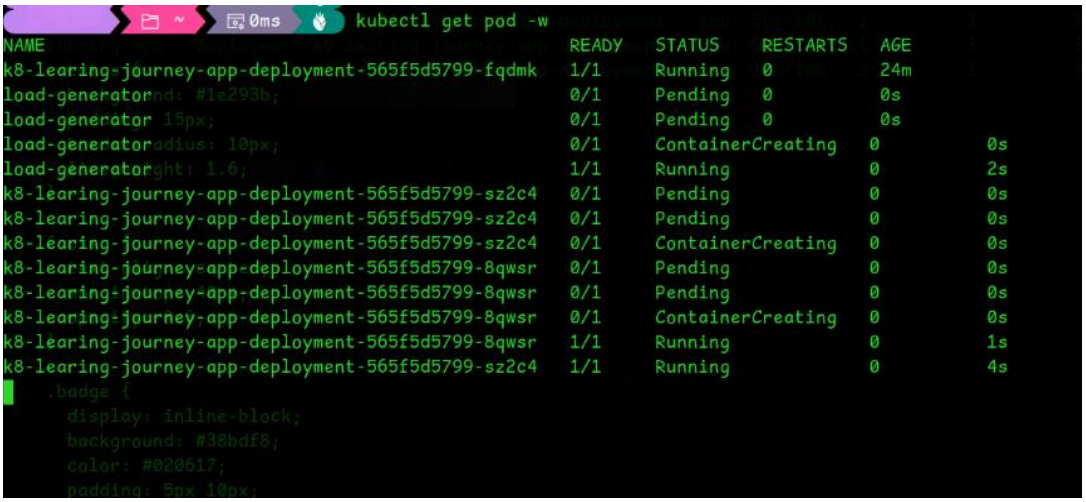


***Fig 7.2.1:* Pods Increasing**

- System reacting to load

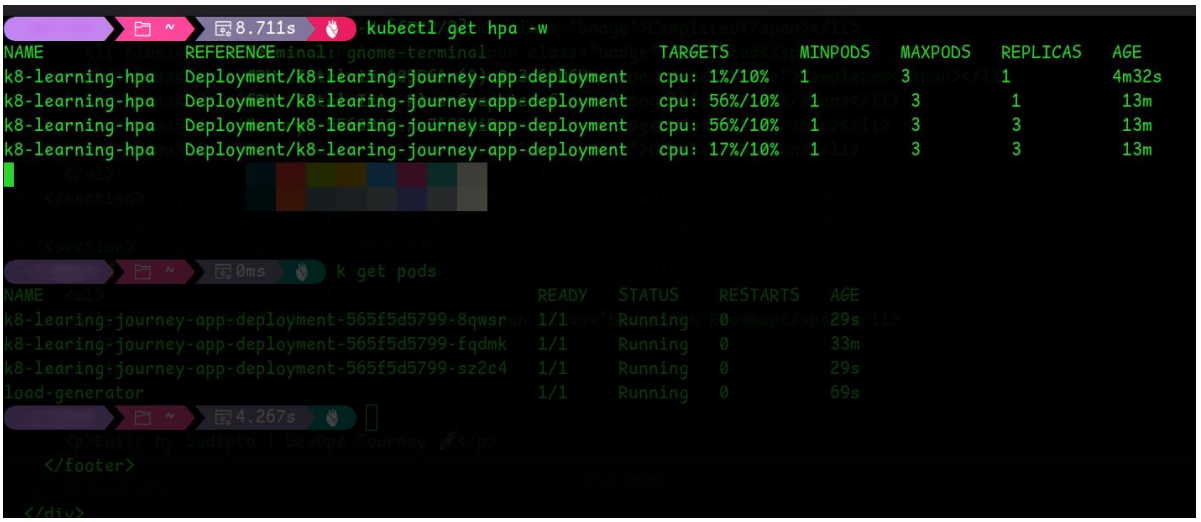


Fig 7.2.2: **System reacting to load**

7.3 HPA – Scaling Down

- Cost optimization and Dynamic scaling

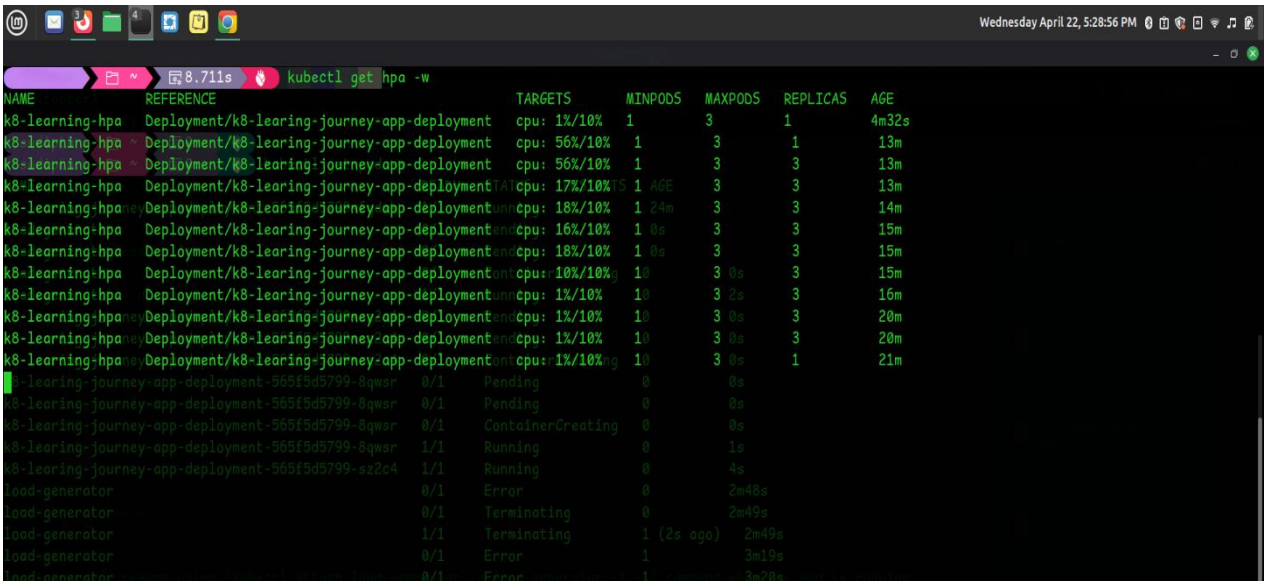


Fig 7.3.1: **Dynamic scaling**

7.4 Disaster Scenario – Failure

- System fails

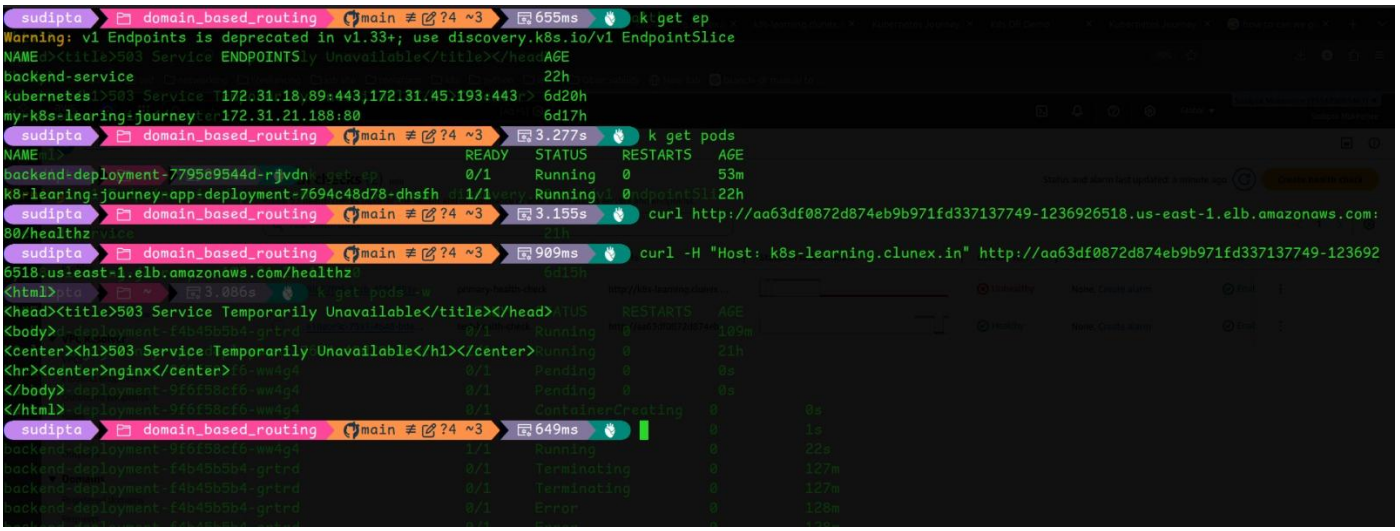


Fig 7.4.1: **Kubernetes cluster status showing pod failures during disaster simulation**

- Service failure (HTTP error response)

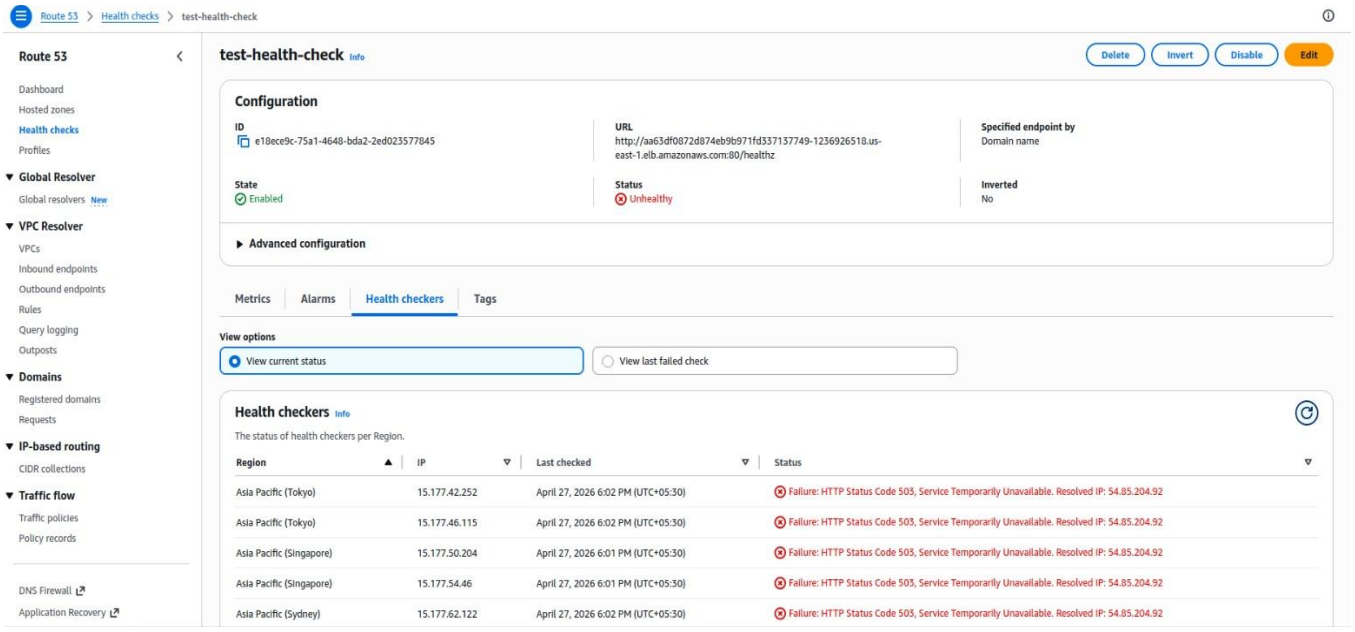


Fig 7.4.2: **Route 53 health check confirming primary region failure (unhealthy status)**

7.5 Disaster Recovery – Failover

- Traffic successfully switched and System recovered

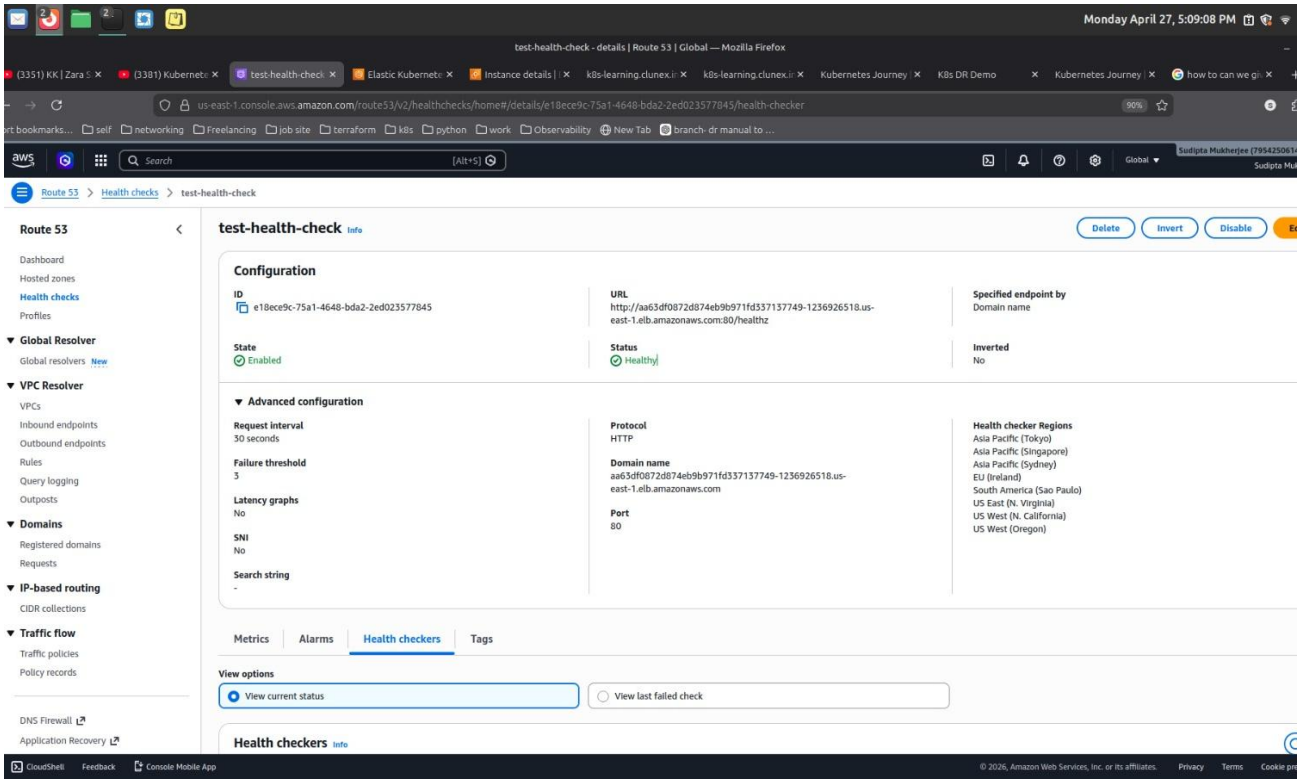


Fig 7.5.1: **System restored with healthy status after successful failover to secondary region**

CHAPTER 8

CONCLUSION & FUTURE ENHANCEMENT

Our project successfully demonstrates zero-downtime and highly available application deployment architecture. Multiple mechanisms were implemented together to ensure uninterrupted service availability under different failure scenarios.

Rolling updates with readiness probes ensured seamless application version upgrades without affecting users. HPA dynamically scaled pods based on CPU utilization during heavy traffic conditions, helping maintain application performance while optimizing resource usage during low traffic periods. Liveness probes provided automatic self-healing by detecting unresponsive Node.js containers and restarting them without manual intervention.

To improve fault tolerance at the infrastructure level, a warm standby disaster recovery architecture was implemented across multiple AWS regions using Amazon Route 53 health checks and automated DNS failover. The system successfully redirected traffic to the secondary region when the primary region became unavailable, reducing service disruption and recovery time. The observed failover recovery time was approximately 1–2 minutes depending on Route 53 health check interval and DNS propagation.

Among all implemented use cases, the liveness probe-based self-healing mechanism provided the fastest recovery at the pod level, typically recovering within seconds through automatic container restart. Disaster recovery failover required slightly higher recovery time due to DNS propagation and health check intervals but ensured business continuity during regional failures.

For future enhancement, the architecture can be extended to an active-active multi-region deployment, integrated with CI/CD pipelines, predictive autoscaling, service mesh technologies, and advanced observability solutions for production-scale environments.

CHAPTER 9

REFERENCES

1 *Kubernetes. (n.d.). Deployments*. Retrieved from https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

2 *Kubernetes. (n.d.). Horizontal Pod Autoscaling*. Retrieved from https://kubernetes.io/docs/concepts/workloads/autoscaling/horizontal-pod-autoscale/

3 *Amazon Web Services. (n.d.). Amazon EKS documentation*. Retrieved from https://docs.aws.amazon.com/eks/

4 *Amazon Web Services. (n.d.). Amazon Route 53 developer guide*. Retrieved from https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/Welcome.html

5 *Amazon Web Services. (n.d.). Disaster recovery (DR) architecture on AWS*: *Pilot light and warm standby*. Retrieved from https://aws.amazon.com/blogs/architecture/disaster-recovery-dr-architecture-on-aws-part-iii-pilot-light-and-warm-standby/

6 *HashiCorp. (n.d.). Zero-downtime deployments*. Retrieved from https://developer.hashicorp.com/well-architected-framework/define-and-automate-processes/deploy/zero-downtime-deployments

7 *Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). Borg, Omega, and Kubernetes*. Communications of the ACM, 59(5), 50–57