Architecture Justification Document

Project: Sock Shop Microservices Platform on AWS
Author: Inalegwu John Aleyi

---

1. Introduction

This document provides the architectural justification for the design and implementation of a highly available, secure, and scalability-ready cloud-native platform deployed on AWS. The architecture supports a microservices-based e-commerce application (Sock Shop) and is designed to meet enterprise-grade requirements around resilience, automation, security, and operational excellence.

The solution leverages Terraform Infrastructure as Code (IaC), Ansible configuration management, Kubernetes container orchestration, and Jenkins continuous delivery pipelines to deliver a robust production-ready environment that can deliver a highly available, secure, and automated microservices platform on AWS, with built-in failover mechanisms.

---

2. Architectural Objectives

The primary objectives guiding this architecture are:

- **High Availability:** Eliminate single points of failure across infrastructure and application layers.
- **Security:** Enforce least-privilege access and network isolation.
- **Automation:** Minimise manual intervention using CI/CD and Infrastructure as Code.
- **Observability:** Provide full visibility into system health and performance.
- **Maintainability:** Enable easy updates, troubleshooting, and future enhancements.
- **Scalability:** The application is scalability-ready and with futuristic outlook and possibility to support horizontal scaling at both infrastructure and application levels.

---

3. Cloud Platform Selection (AWS)

AWS was selected as the cloud provider due to:

- Proven reliability and global availability
- Native support for scalable networking, compute, and storage
- Mature security services (IAM, VPC, security groups)
- Strong integration with Kubernetes, Terraform, and CI/CD tooling

Using AWS aligns the solution with industry-standard enterprise cloud practices.

---

4. Network Architecture Justification

Virtual Private Cloud (VPC)

The entire system is isolated within a dedicated VPC across three Availability Zones (Azs) to protect against data center failures. The VPC is segmented into:

Public Subnets: For internet-facing components

Private Subnets: For internal systems and workloads

This separation enforces network-level security and reduces attack surfaces.

Public Subnets (per AZ)

- Application Load Balancer
- Bastion Host (entry points to resources in private subnets)

Private Subnets (per AZ)

- Master nodes - Kubernetes control plane
- Worker nodes
- HAProxy instances
- Ansible control node

This design ensures that critical workloads are not directly exposed to the internet. This ensures security by isolation.

---

5. Access & Security Justification

A. Bastion Host with Auto Scaling

A bastion host is used as the sole entry point into private infrastructure:

- Prevents direct SSH access to private resources
- Auto Scaling ensures high availability
- Supports secure operational (admin) access and auditing

B. IAM & Security Controls

- Role-based access control
- Least-privilege permissions

- Secure authentication and authorisation boundaries

These controls reduce the risk of unauthorised access.

---

6. Traffic Management & High Availability

A. Application Load Balancer (ALB)

The ALB provides:

- Internet-facing access
- Health checks
- Load distribution
- TLS termination

This ensures reliable and secure access to the application.

B. HAProxy with Keepalived (VIP Management):

A critical requirement for modern applications is **zero downtime**. This is achieved through a redundant load-balancing layer.

- **HAProxy:** Dual instances distribute incoming traffic efficiently across the Kubernetes cluster.
- **Keepalived:** This service monitors the health of the load balancers. If the primary instance fails, a "Virtual IP" is instantly failed over to the backup, ensuring the application remains reachable without human intervention.

HAProxy instances run in private subnets and are protected by Keepalived:

- Keepalived assigns a Virtual IP (VIP)
- Automatic failover if one HAProxy node fails - If HAProxy-01 fails → HAProxy-02 automatically takes over
- Ensures uninterrupted traffic flow to Kubernetes services

This eliminates ingress-level single points of failure.

---

7. Kubernetes Architecture Justification

A. Multi-Master Kubernetes Cluster (Core of the Architecture)

The Kubernetes cluster is deployed with:

- **3 master (control plane) nodes -** Three master nodes ensure the cluster's "brain" is always available.

- **3 worker nodes -** Three worker nodes provide the muscle to run the application containers(microservices).
- Nodes distributed across multiple availability zones

This design ensures self-healing vis-a-vis:

- Control plane resilience with high availability -
- Fault tolerance during node or AZ failures
- Continuous workload availability

B. Namespace Segregation

Namespaces are used to separate concerns: - Using Namespaces, we separate the environment into Production, Staging, and Monitoring, preventing configuration "leakage" between environments.

- Production
- Staging
- Monitoring

This improves security, organisational clarity, scalability-readiness and operational control.

---

8. Automation & Configuration Management

A. Terraform (Infrastructure as Code)

Terraform is used to:

- Provision AWS infrastructure
- Manage VPCs, subnets, load balancers, compute resources
- Store state remotely in S3 for consistency, state locking, collaboration and disaster recovery

This ensures repeatability and eliminates configuration drift.

B. Ansible (Configuration Management)

Ansible is used to:

- Configure operating systems
- Bootstrap Kubernetes nodes
- Install required packages and dependencies
- Configure HAProxy and Keepalived
- Enforce standardised system configurations

The separation of Terraform and Ansible responsibilities follows best practices. This ensures consistent and repeatable configuration across nodes.

---

9. Github Repositories justification:

The architecture intentionally adopts a **multi-repository (multi-repo) strategy**, separating application code and infrastructure code into two distinct GitHub repositories:

- **Application Repository (App Repo)**
  Contains microservices source code, Dockerfiles, Kubernetes manifests, and deployment configurations.
- **Infrastructure Repository (Infra Repo)**
  Contains Terraform, Ansible, CI/CD pipeline definitions, and environment-level configurations.

Having separate Github repositories confers the following benefits:

- **Clear Separation of Concerns** - Prevents accidental infrastructure changes during application updates.
- **Improved Security & Access Control** - Developers can deploy applications without direct access to cloud infrastructure, supporting the principle of least privilege.
- **Reduced Operational Risk** - Application rollbacks do not affect infrastructure, reducing deployment risk.

The separation of application and infrastructure repositories enforces clear ownership boundaries, improves security and enables independent deployment pipelines.

---

10. CI/CD Pipeline Justification

Jenkins is the central automation engine:

- Pulls code from both repositories
- Uses plugins such as:
  - Terraform → infrastructure provisioning
  - AWS → cloud integration
  - OWASP ZAP → security scanning
  - Slack → build notifications

Jenkins orchestrates:

- Infrastructure provisioning
- Application deployment
- Security scanning (OWASP ZAP)

- Notifications via Slack

This enables continuous integration and delivery while reducing human error.

---

11. Observability & Monitoring

The platform includes:

- **Prometheus**: Continually scrapes performance metrics from every container and server.
- **Grafana:** Provides real-time visual dashboards for management, showing system health, traffic patterns, and resource consumption at a glance.
- Alerting for proactive incident response

This ensures operational visibility and supports SRE best practices.

---

12. Scalability & Resilience

Within this architecture, only the Bastion host attained scalability. However, and as pointed out earlier, the architecture is scalability-ready at multiple layers:

- Auto Scaling Groups for infrastructure
- Kubernetes horizontal pod autoscaling and vertical pod autoscaling.
- Kubernetes events driven autoscaling
- Load-balanced ingress traffic

When fully configured, failures at the node, service, or availability zone level can be handled gracefully without downtime.

---

13. Risk Mitigation & Failure Handling

The architecture mitigates common risks by:

- Using multi-AZ deployments
- Removing single points of failure
- Enforcing strict network segmentation
- Automating recovery and failover mechanisms

These measures significantly improve system reliability.

---

14. **A Look At The Deployment Work Flow**

1. Developer pushes code to GitHub
2. Jenkins polls for changes and triggers pipeline
3. Terraform provisions infrastructure
4. Ansible configures servers
5. Kubernetes deploys app into appropriate namespace
6. Monitoring tools track performance
7. Slack notifies team of pipeline status

---

15. Why these matters for the Business:

- Improved customer experience through high availability - The "self-healing" nature of Kubernetes and Keepalived means the system recovers from failures automatically.
- Reduced operational overhead via automation
- Faster time-to-market with CI/CD
- Lower risk through proactive security checks - multi-layer firewalls and private networking minimize the risk of data breaches.
- Future-proof and future-ready platform capable of auto-scaling with proper configurations. Even the barest implemented here goes a long way to handle traffic spikes.

---

16. Conclusion

This architecture provides a secure, highly available, scalability-ready and production-ready Kubernetes platform suitable for modern microservices workloads. By combining AWS-native services, Kubernetes, Infrastructure as Code, and CI/CD automation, the solution meets enterprise standards for **AVAILABILITY**, **SECURITY**, and **OPERATIONAL EXCELLENCE**.

The design demonstrates best practices in cloud architecture and DevOps engineering.
