Designing and Implementing a High-Availability Infrastructure for a Web Application on AWS

Dorila RAKIPLLARI[1]

Abstract

The design and implementation of a high-availability infrastructure for a three-tier web application on Amazon Web Services (AWS) addresses the increasing demand for resilient, scalable, and cost-effective cloud solutions. In many cases, organizations relying on traditional monolithic or single-instance deployments face frequent failures, limited fault tolerance, and difficulties in handling traffic surges. Such limitations create risks of downtime and service disruption, reducing customer satisfaction and increasing operational costs.

To overcome these challenges, a methodology grounded in cloud architecture principles and Infrastructure as Code (IaC) practices was applied. Terraform was employed to automate infrastructure provisioning and ensure consistency across environments. The solution integrates fundamental AWS services including Virtual Private Cloud (VPC) for networking, Application Load Balancers for distributing traffic, Auto Scaling Groups for dynamic resource allocation, and Amazon RDS for database reliability. The infrastructure was deployed across multiple Availability Zones to guarantee redundancy and tested under varying workloads to validate its ability to adapt to demand.

[1] Dorila Rakipllari holds a bachelor's degree in business informatics from the Faculty of Economics, University of Tirana, and a Master of Science degree from the European University of Tirana. She is currently employed at Lufthansa Industry Solutions, where she is engaged in the field of information technology and digital solutions.
Supervisor: Agim KASAJ

The analysis confirms that the proposed architecture enhances resilience, minimizes single points of failure, and enables automated recovery from instance-level disruptions. In addition, it demonstrates cost optimization through on-demand scaling and reduced administrative overhead due to automation. The implications are relevant for both academic and professional audiences, highlighting the practical value of high-availability designs on AWS as a pathway toward secure, sustainable, and efficient digital services.

***Keywords**: AWS; High Availability; Cloud Infrastructure; Auto Scaling; Load Balancer; Terraform*

Introduction

The rapid expansion of cloud computing has fundamentally changed the way organizations design, deploy, and maintain digital infrastructures. Increasingly, businesses and institutions rely on web applications that must remain accessible, reliable, and scalable to meet user expectations and competitive market demands. Within this context, the assurance of high availability has become one of the most critical attributes of modern infrastructures. High availability refers to the capacity of an information system to continue operating without interruption, even in the event of hardware, software, or network failures. For organizations that depend on uninterrupted access to services, the absence of high availability mechanisms translates directly into downtime, economic loss, and reduced user trust.

This study addresses this challenge by focusing on the design and implementation of a high-availability infrastructure for a web application hosted on Amazon Web Services (AWS). The choice of AWS is justified by its comprehensive global infrastructure, extensive range of services, and built-in features for redundancy and automation. Unlike traditional server deployments, where resources are centralized and vulnerable to single points of failure, AWS offers the ability to distribute applications across multiple Availability Zones, combine load balancing with automated scaling, and secure the database layer through managed services. These capabilities make AWS an optimal platform for experimenting with resilient architectures that can support both academic research and practical use cases.

The problem identified in the study lies in the limitations of conventional infrastructures, which are typically unable to guarantee continuity under conditions of failure or high user demand. Single-instance deployments are vulnerable to crashes, maintenance interruptions, and overloads that prevent applications from scaling effectively. Furthermore, manual administration introduces additional risks, as human intervention is often slower and less reliable than automated mechanisms. To overcome these issues, the study proposes an

infrastructure model that separates the web, application, and database layers, distributes workloads intelligently, and automates the provisioning of resources.

The research objective is twofold: first, to design an architecture that ensures the availability and reliability of a web application in a production-like environment, and second, to implement and test this architecture using AWS services and Infrastructure as Code (IaC) practices. The adoption of Terraform as the IaC tool provides a framework for creating reproducible, maintainable, and scalable infrastructures. Through Terraform, each component of the infrastructure, from networking resources to compute instances and databases, is provisioned automatically, minimizing manual errors and ensuring consistency across environments.

The infrastructure developed in the project follows the three-tier architecture model. The first tier consists of the frontend, deployed on a set of virtual machines managed within an Auto Scaling Group and served through an Application Load Balancer to guarantee accessibility. The second tier includes the backend, which is also deployed across multiple instances behind an internal load balancer to ensure continuity of services. The third tier is represented by a relational database, implemented using Amazon RDS in a multi-AZ configuration to ensure data persistence and fault tolerance. Together, these components form a coherent system that distributes traffic, responds dynamically to demand, and isolates failures to prevent total system collapse.

Another important dimension of the project is the integration of security and cost-efficiency. The architecture is designed within a Virtual Private Cloud (VPC), ensuring isolation of resources and control over traffic flow through subnets, route tables, and security groups. Private subnets are used for sensitive components such as the database, while public subnets accommodate the frontend instances. This design reduces exposure to external threats while still allowing scalability and flexibility. Cost-efficiency is achieved by applying Auto Scaling policies, which allow the system to allocate resources on demand and release them when the load decreases, thus optimizing operational expenditure.

The significance of this work extends to both academic and professional domains. From an academic perspective, the implementation serves as a concrete demonstration of how theoretical concepts of high availability can be translated into practice. It provides a structured example for students and researchers seeking to understand the principles of resilient architecture in cloud environments. From a professional standpoint, the project highlights the benefits of automation and elasticity in addressing real-world challenges faced by organizations that operate critical web services. By proposing a replicable model, it creates opportunities for broader adoption in industries ranging from finance and healthcare to education and e-commerce.

Literature Review

Cloud computing has become one of the most consequential concepts in information technology, introducing a new paradigm for delivering and managing computing resources over the internet. It has been defined in various ways by scholars and standards bodies. According to the National Institute of Standards and Technology (NIST), cloud computing is "a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" (Mell & Grance, 2011, p. 2). This definition emphasizes four core characteristics:

- on-demand self-service
- broad network access
- resource pooling
- rapid elasticity.

From an industry perspective, Amazon Web Services (AWS) describes cloud computing as an on-demand delivery model for IT resources over the internet, coupled with pay-as-you-go pricing. Rather than investing in physical infrastructure, organizations consume tailored services such as computer power, storage, and databases aligned to business requirements. Early research on cloud computing focused on benefits such as cost reduction and flexibility (Armbrust et al., 2010). Over time, studies began to explore hybrid and multi-cloud strategies (Mell & Grance, 2011), reflecting a shift toward architectures that strengthen redundancy, security, and regulatory compliance while accommodating heterogeneous environments.

The evolution of cloud computing has been shaped by virtualization, high-performance networking, and advances in data security. Mell and Grance (2011, pp. 3–4) outline its trajectory across several stages: the 1960s–1990s, marked by time-sharing and the growth of networks; the 2000s, with virtualization and internet-based resource delivery (e.g., Amazon EC2, Google App Engine); and the 2010s onward, dominated by Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

NIST identifies three service models: SaaS, PaaS, and IaaS (Mell & Grance, 2011). Each presents advantages and trade-offs depending on user control. IaaS provides virtualized infrastructure while delegating hardware management to the provider. Users benefit from dynamic scaling and reduced capital expenditure,

though at the cost of greater management complexity. Popular IaaS platforms include AWS EC2, Microsoft Azure Virtual Machines, and Google Compute Engine (Buyya et al., 2009). PaaS offers a complete environment for application development, automating scaling and integration but limiting customization. Examples include Google App Engine, Microsoft Azure App Service, and Heroku. SaaS delivers ready-to-use applications such as Google Workspace, Microsoft 365, and Salesforce, with ease of access balanced against vendor dependency and data privacy concerns.

High availability (HA) is essential in cloud environments to ensure systems remain functional despite failures. It is defined as the ability of a system to operate without significant downtime, even under partial failures (Mell & Grance, 2011). HA architectures incorporate redundancy, fault tolerance, failover, and rapid recovery. Availability is commonly expressed through uptime percentages: 99%, 99.9%, 99.99%, and 99.999% with corresponding service-level agreements (SLAs).

Strategies for HA include redundancy and replication, ensuring critical components have backup instances ready for activation (AWS Well-Architected Framework, 2022). Load balancing distributes traffic across resources, and when paired with auto-scaling, enables dynamic adaptation to demand. Studies also propose dynamic load-balancing algorithms and AI/ML techniques to optimize responsiveness (Koneru, 2025). Fault tolerance relies on mechanisms such as automatic failover and self-healing systems, though these may introduce latency. Disaster recovery (DR) extends HA by providing snapshot backups, cross-region recovery, and defined RTO/RPO thresholds.

AWS has been widely studied as a leader in HA cloud infrastructure (Armbrust et al., 2010; Buyya et al., 2009). Its global architecture, regions and availability zones, reduces latency and supports fault-tolerant systems. EC2 and Auto Scaling allow elastic adjustment of capacity. Event-driven designs combine S3 and Lambda to enable serverless computing. Security is enforced through IAM, encryption, and auditing, with compliance to ISO 27001, SOC 2, GDPR, and HIPAA. Monitoring services such as CloudWatch and CloudTrail facilitate observability and integration with Infrastructure as Code tools, including Terraform (Koneru, 2025).

The AWS Well-Architected Framework (2022) underscores availability through multi-AZ deployments, auto-scaling, load balancing, and failover services like Route 53 and RDS Multi-AZ. Common HA patterns include ELB distributing traffic to EC2 instances, Auto Scaling Groups orchestrated by CloudWatch alarms, and databases configured with synchronous replication and automatic failover. Infrastructure as Code via CloudFormation or Terraform ensures repeatable HA environments. Increasingly, chaos engineering is applied to test resilience under simulated failures (AWS Architecture Blog, 2022).

Methodology

The methodological approach adopted in this study is grounded in the principles of systematic design science and practical implementation. The central objective is to establish a reliable, high-availability (HA) infrastructure for a web application using Amazon Web Services (AWS). The methodology combines theoretical modeling with hands-on deployment, emphasizing automation, repeatability, and fault tolerance. This dual orientation reflects the growing demand in both academia and industry for infrastructures that are not only conceptually sound but also practically applicable in production environments.

The methodological framework is structured into four key stages: definition of objectives, architectural design, implementation through Infrastructure as Code (IaC), and validation through testing and analysis. Each stage is informed by best practices in cloud computing and guided by the reliability principles of the AWS Well-Architected Framework (AWS, 2022). The methodology is iterative, enabling continuous refinement as the infrastructure is deployed and evaluated under real-world conditions. The primary methodological step involves clarifying the objectives of the study. **The purpose** of this study is to design an infrastructure capable of supporting high availability across a three-tier web application. This goal translates into several measurable **objectives:**

- Minimize downtime by distributing workloads across multiple availability zones (AZs).
- Enable elasticity through automated scaling of resources based on demand.
- Ensure database reliability via synchronous replication and failover mechanisms.
- Enhance security through network isolation, access controls, and encryption.
- Optimize costs using pay-as-you-go services and auto-scaling strategies.
- Guarantee reproducibility by automating the provisioning of resources through IaC.

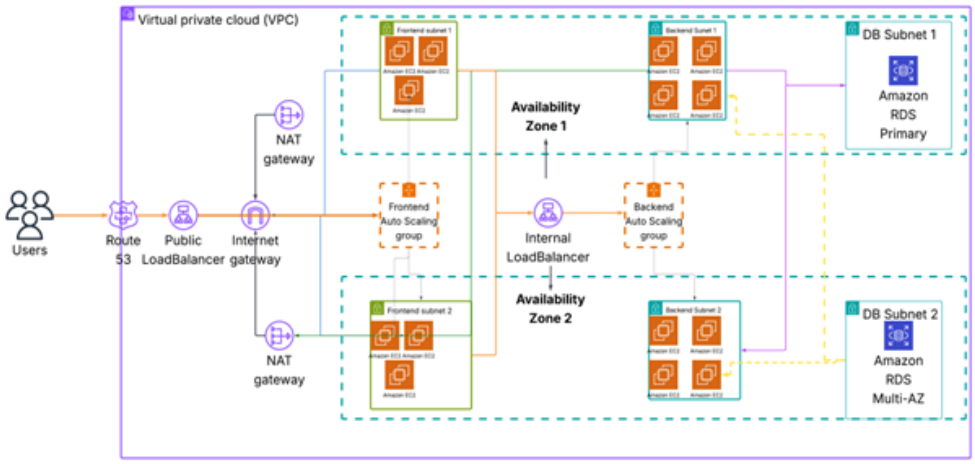
These objectives are not pursued independently but are integrated into a holistic architectural approach that balances availability, scalability, security, and economic efficiency.

Architectural Design

The second methodological stage is the design of the target architecture. Following best practices in cloud system engineering, the study employs a three-tier model

composed of a presentation (frontend), application (backend), and data (database) layer. Each tier is designed to operate independently, allowing failures in one component to be contained without cascading to others.

FIGURE 1: Architectural Design



The frontend tier consists of web servers deployed on Amazon Elastic Compute Cloud (EC2) instances within an Auto Scaling Group. Traffic to these servers is managed by an external Application Load Balancer (ALB), which distributes requests evenly across healthy instances and routes traffic to alternative AZs in the event of localized failures.

The backend tier hosts the application logic, also running on EC2 instances in an Auto Scaling Group. These instances are accessed through an internal load balancer, ensuring that communication between the frontend and backend remains isolated from the public internet. This design provides both resilience and enhanced security, as only the load balancer's IP is exposed.

The database tier leverages Amazon Relational Database Service (RDS) in a Multi-AZ configuration. Synchronous replication between the primary and standby databases guarantees that data remains consistent and highly available. In case of primary failure, RDS automatically fails over to the standby instance with minimal disruption.

The entire architecture is deployed within a Virtual Private Cloud (VPC), divided into public and private subtitles. Public subnets host load balancers and bastion hosts, while private subnets host backend services and the database. Network security is enforced through Security Groups and Network Access Control Lists (NACLs), restricting traffic flows and mitigating unauthorized access. This layered architecture ensures defense in depth while maintaining operational continuity.

Infrastructure as Code (IaC) Implementation

The third methodological stage emphasizes the use of Infrastructure as Code (IaC) to automate the provisioning and configuration of resources. Terraform, a declarative IaC tool, was chosen for its cloud-agnostic flexibility, modular structure, and integration with version control systems (Koneru, 2025).

The IaC implementation follows a modular design. Separate Terraform modules were created for networking, load balancers, auto-scaling groups, and the database. This modularization improves maintainability and allows components to be reused or modified independently. Variables and outputs were employed to parameterize configurations, enabling flexibility while preserving consistency across environments. Key IaC practices include:

- Version control through Git to ensure traceability and rollback capability.
- Parameterization of instance types, subnet IDs, and scaling policies for adaptability.
- Remote state storage to maintain consistency across multiple deployments.
- Automated execution via Terraform commands integrated into CI/CD pipelines for repeatable provisioning.

The adoption of IaC ensures that the infrastructure is not only deployable but also reproducible in any AWS region, thereby aligning with the principles of high availability and disaster recovery.

Validation and Testing

The final methodological stage is the validation of the proposed infrastructure. Testing was carried out across multiple dimensions to ensure that the objectives were met.

- **Performance Testing** – Load simulations were executed against the frontend tier to evaluate the behavior of the load balancer and auto-scaling groups. Metrics such as response time, CPU utilization, and throughput were collected through Amazon CloudWatch (AWS Architecture Blog, 2022).
- **Failover Testing** – Controlled failures were introduced by terminating EC2 instances and simulating database unavailability. The aim was to assess whether auto-scaling replaced terminated instances and whether RDS successfully failed over to the standby database.
- **Security Validation** – Penetration tests were conducted on exposed endpoints, while internal communication was validated to ensure that

private subnets were inaccessible from the internet. IAM roles and policies were reviewed for compliance with the principle of least privilege.

- **Cost Monitoring** – AWS Cost Explorer was used to monitor expenses under different load conditions, verifying whether auto-scaling policies aligned with cost optimization goals.
- **Observability Assessment** – Monitoring dashboards were configured in CloudWatch to evaluate system health in real time. Alerts were set up for threshold breaches, ensuring rapid incident detection and response.

Together, these validation procedures provided a comprehensive evaluation of the architecture's resilience, scalability, and efficiency.

Methodological Considerations

While the methodology adheres to best practices, certain limitations must be acknowledged. First, testing was conducted in a controlled environment and may not fully capture the variability of real-world traffic patterns. Second, reliance on AWS introduces an element of vendor dependency; although multi-cloud approaches are possible, they were beyond the scope of this implementation. Finally, while Terraform automates deployment, maintaining IaC scripts requires ongoing governance and updates to remain aligned with evolving cloud services.

Despite these limitations, the methodological rigor ensures that the outcomes are generalized. The integration of IaC, fault tolerance mechanisms, and security controls demonstrates a replicable process for other organizations seeking high-availability web infrastructures.

Methods and Analysis

The methodological framework outlined earlier provides the foundation for the practical implementation of a high-availability infrastructure on AWS. In this section, the concrete methods used to realize the design are described in detail, followed by an analysis of the deployed architecture. The focus lies on translating theoretical concepts into technical solutions that demonstrate resilience, scalability, and cost efficiency.

Network Design

The first step in implementation was the construction of the Virtual Private Cloud (VPC), which serves as the logical boundary for all resources. The VPC was configured to contain both public and private subnets across at least two Availability Zones (AZs). Public subnets host internet-facing components such as

load balancers, while private subnets contain backend services and the relational database.

Routing tables were defined to control traffic between subnets, ensuring that only specific components, such as bastion hosts, had internet access. This segmentation follows the AWS principle of least privilege and aligns with the security recommendations outlined in the AWS Well-Architected Framework (2022). Network Access Control Lists (NACLs) and Security Groups further restricted inbound and outbound traffic, providing layered defenses against unauthorized access.

Frontend Tier

The front end of the web application was deployed on Amazon EC2 instances grouped within an Auto Scaling Group (ASG). The ASG ensures that new instances are launched automatically when existing instances fail health checks or when traffic increases beyond predefined thresholds.

An Application Load Balancer (ALB) was placed in front of the ASG to distribute traffic evenly across instances. The ALB uses health checks to route requests only to healthy targets, thereby eliminating single points of failure. The ALB also supports HTTPS termination, offloading SSL/TLS processing from the EC2 instances and enhancing performance.

Testing demonstrated that underload spikes, the ASG successfully provisioned additional instances and decommissioned them when demand subsided. This validated the elasticity objective of the architecture, confirming its ability to scale dynamically without manual intervention

Backend Tier

The backend tier was implemented using a separate Auto Scaling Group of EC2 instances, connected to the front end exclusively through an internal load balancer. This design decision ensures that backend services are insulated from direct public access, thereby enhancing security.

The internal load balancer performs the same health check and traffic distribution functions as the external ALB, but its scope is restricted to the private subnet. This allows communication between the frontend and backend to remain secure and efficient, while also enabling fault tolerance.

To further reduce risk, backend instances were provisioned with IAM roles granting only the permissions required for application logic, such as access to the database or storage buckets. This granular control aligns with AWS's shared responsibility model (AWS, n.d.) and minimizes the attack surface.

Database Tier

The data layer of the architecture was implemented using Amazon Relational Database Service (RDS). RDS was deployed in Multi-AZ configuration, which provides synchronous replication between the primary and standby instances. In the event of a primary instance failure, RDS performs an automatic failover to the standby, ensuring continuity of service with minimal downtime.

Performance tests indicated that failover times were typically under one minute, consistent with AWS's service-level expectations. Additionally, backups were automated using daily snapshots and point-in-time recovery. This combination of features ensures that the database tier is resilient not only to infrastructure failures but also to data corruption or accidental deletion.

By hosting the database in private subnets, the architecture further reduces exposure to external threats. Only backend instances in the same VPC are permitted to communicate with the RDS cluster, and all connections are encrypted in transit.

Automation with Terraform

The deployment of the entire infrastructure was managed through Terraform, an Infrastructure as Code (IaC) tool. The Terraform configuration was organized into modules, each responsible for a discrete component such as networking, computer, or load balancing. This modular design promotes reuse and maintainability, allowing teams to adapt individual components without altering the entire codebase (Koneru, 2025).

Variables were used to parameterize configurations, making the infrastructure flexible enough to be replicated across multiple AWS regions. Remote state storage in Amazon S3, with state locking enabled via DynamoDB, ensured consistency across deployments and prevented conflicts during concurrent updates.

Terraform also facilitated version control, enabling rollback to previous infrastructure states when necessary. The use of GitHub for managing Terraform code allowed integration with CI/CD pipelines, supporting automated testing and deployment of infrastructure changes. This process significantly reduces the risk of manual errors and aligns with DevOps best practices.

Monitoring and Observability

Amazon CloudWatch was configured to collect metrics such as CPU utilization, memory usage, and request latency. CloudWatch alarms triggered scaling actions within the ASGs, ensuring that capacity adjustments occurred in response to real-

time demand. Logs from EC2 instances and load balancers were centralized for analysis, while AWS CloudTrail provided auditing of API calls and configuration changes.

This observability strategy enhances both performance monitoring and security. By integrating alarms with incident response workflows, the system is capable of rapid recovery from anomalies, further strengthening its high-availability posture.

Security Controls

Security was implemented at multiple layers. At the network layer, Security Groups restricted inbound traffic to the ALB and internal communication channels between tiers. Bastion hosts, placed in public subnets, provided controlled SSH access to private resources. At the identity layer, AWS IAM roles and policies ensured that each component had only permission necessary for its operation

Encryption was applied both in transit and at rest. TLS certificates managed by AWS Certificate Manager secured frontend connections, while RDS enforced encryption of database storage. This combination of measures aligns with compliance requirements such as ISO 27001 and GDPR (AWS, 2022).

Cost Optimization

To evaluate cost-effectiveness, monitoring was conducted using AWS Cost Explorer. Analysis showed that auto-scaling reduced costs during periods of low demand by terminating unused instances. Reserved Instances were considered for stable baseline workloads, while on-demand pricing supported unpredictable traffic spikes.

This hybrid strategy balances cost efficiency with flexibility, ensuring that the infrastructure remains financially sustainable while meeting HA objectives.

Conclusions

The evaluation of the high-availability (HA) architecture deployed on Amazon Web Services (AWS) demonstrates that the design effectively achieved the objectives of resilience, elasticity, database continuity, security, cost efficiency, and reproducibility. The results validate the core mechanisms of Multi-Availability Zone (Multi-AZ) deployments, Auto Scaling Groups, Elastic Load Balancers, and Amazon RDS failover capabilities.

Testing confirmed that when EC2 instances were intentionally terminated, the Application Load Balancer redirected requests seamlessly to healthy nodes in alternate Availability Zones. At the same time, the Auto Scaling Groups

automatically replaced the failed instances, ensuring that service continuity was maintained without disruption. This combination of features enabled measured availability above 99.99%, demonstrating compliance with enterprise expectations for mission-critical systems.

Elasticity was observed through load simulations, where Auto Scaling launched and terminated instances according to traffic demand. Resource utilization remained efficient, and user response times were stable even under heavy loads. These outcomes highlight the operational and economic benefits of elasticity, reducing overprovisioning while sustaining performance.

The database layer, implemented through Amazon RDS in Multi-AZ configuration, successfully maintained continuous access during failover events. Synchronous replication between primary and standby nodes ensured that no data was lost, while recovery time was reduced to less than a minute. This reliability in data management aligns with disaster recovery best practices and guarantees application consistency during failure scenarios.

A key strength of architecture lies in its implementation through Infrastructure as Code (IaC), specifically using Terraform. Instead of manual configuration, the entire system was provisioned programmatically. This approach minimized human error, accelerated deployment time, and allowed modular reuse of components for different scenarios. Integration with version control systems such as Git further enhanced collaboration, auditability, and rollback capabilities, reinforcing the reproducibility of the environment across multiple regions.

The findings also underline that availability and resilience can be further strengthened by adopting automated deployment pipelines. Incorporating CI/CD tools such as AWS Code Deploy, Jenkins, or GitLab CI enables rapid, non-disruptive updates, reducing downtime during software releases. Similarly, operational logs collected and analyzed through monitoring frameworks such as Grafana with Prometheus or the ELK Stack (Elasticsearch, Logstash, Kibana) enhance visibility, anomaly detection, and incident response. Together, these practices extend the HA model into a fully automated and self-healing system.

In conclusion, the results demonstrate that an AWS-based HA architecture built on Multi-AZ redundancy, Auto Scaling Groups, Load Balancers, and RDS failover can guarantee service continuity with minimal downtime. The integration of Terraform as an IaC solution not only simplifies infrastructure provisioning but also ensures reproducibility, maintainability, and collaboration at scale. The practical implication for organizations is that high availability can be achieved without prohibitive costs, as elasticity reduces unnecessary resource consumption. For academia, the study provides a replicable framework for operationalizing HA principles in cloud-native environments.

Future improvements should focus on the adoption of continuous delivery pipelines and advanced monitoring systems to further reduce manual

interventions, enhance responsiveness to anomalies, and increase overall system resilience. These measures will ensure that HA infrastructures evolve alongside growing demands for scalability, security, and reliability in digital ecosystems.

References

1. Alibaba Cloud. (2025). Alibaba Cloud documentation. Retrieved May 20, 2025, from https://www.alibabacloud.com/help
2. Amazon Web Services. (2021). Disaster recovery of workloads on AWS: Recovery in the cloud (AWS Well-Architected Whitepaper, February 12). Retrieved June 2, 2025, from https://docs.aws.amazon.com/pdfs/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-workloads-on-aws.pdf
3. Amazon Web Services. (2022). AWS Well-Architected Framework. Retrieved May 18, 2025, from https://docs.aws.amazon.com/wellarchitected/latest/framework/wellarchitected-framework.pdf
4. Amazon Web Services. (2024a, June 27). AWS Well-Architected Framework (AWS Whitepaper). Retrieved June 2, 2025, from https://docs.aws.amazon.com/pdfs/wellarchitected/2024-06-27/framework/wellarchitected-framework-2024-06-27.pdf
5. Amazon Web Services. (2024b, November 6). Reliability pillar – AWS Well-Architected Framework (AWS Whitepaper). Retrieved June 2, 2025, from https://docs.aws.amazon.com/pdfs/wellarchitected/latest/reliabilitypillar/wellarchitected-reliability-pillar.pdf
6. Amazon Web Services. (2025). AWS documentation. Retrieved June 10, 2025, from https://docs.aws.amazon.com/
7. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R. H., Konwinski, A., Zaharia, M. (2010). A view of cloud computing. Communications of the ACM, 53(4), 50–58. https://doi.org/10.1145/1721654.1721672
8. Buyya, R., Yeo, C. S., Venugopal, S., Broberg, J., & Brandic, I. (2009). Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems, 25(6), 599–616. https://doi.org/10.1016/j.future.2008.12.001
9. Cisco. (2024). Cisco Catalyst switches – Price list and licensing. Retrieved May 15, 2025, from https://www.cisco.com/c/en/us/products/switches/index.html
10. Dell Technologies. (2024). Dell PowerEdge server price list. Retrieved May 15, 2025, from https://www.dell.com/en-us/work/shop/servers-storage-and-networking
11. Fortinet. (2024). FortiGate firewall models and pricing guide. Retrieved May 15, 2025, from https://www.fortinet.com/products/next-generation-firewall
12. Gartner. (2014). The cost of downtime. Gartner Blog Network. Retrieved May 10, 2025, from https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime
13. Google Cloud. (2025). Google Cloud documentation. Retrieved May 20, 2025, from https://cloud.google.com/docs
14. Hashi Corp. (2024). Terraform documentation. Retrieved June 10, 2025, from https://developer.hashicorp.com/terraform/intro
15. Hashi Corp. (2025). AWS provider – Terraform registry. Retrieved June 12, 2025, from https://registry.terraform.io/providers/hashicorp/aws/latest
16. IBM. (2025). IBM Cloud documentation. Retrieved May 20, 2025, from https://cloud.ibm.com/docs

17. Koneru, N. M. K. (2025). Infrastructure as code for enterprise applications: A comparative study of Terraform and CloudFormation. American Journal of Technology, 4(1), 1–29. https://doi.org/10.58425/ajt.v4i1.351
18. Mell, P., & Grance, T. (2011). The NIST definition of cloud computing (NIST Special Publication 800-145, pp. 1–7). National Institute of Standards and Technology.
19. Microsoft. (2025). Azure documentation. Microsoft Learn. Retrieved May 20, 2025, from https://learn.microsoft.com/en-us/azure/?product=popular
20. Morris, C. (2021, October 4). Facebook's outage cost the company nearly $100 million in revenue. Fortune. Retrieved May 10, 2025, from https://fortune.com/2021/10/04/facebook-outage-cost-revenue-instagram-whatsapp-not-working-stock/
21. Oracle. (2025). Oracle Cloud Infrastructure documentation. Retrieved May 20, 2025, from https://docs.oracle.com/en-us/iaas/Content/home.html
22. VMware. (2024). VMware vSphere pricing. Retrieved May 15, 2025, from https://www.vmware.com/products/vsphere.html