A Production-Grade Blueprint for Containerized Web Applications Deployment: Automated Infrastructure and CI/CD Orchestration on AWS

Kiran Chadde[1] Mr. Y, L. Puranik[2]
[1,2]Master of Computer Applications
[1,2] PES's Modern College of Engineering, Pune, Maharashtra, India

Abstract — The accelerating adoption of cloud-native software delivery has fundamentally redefined the operational expectations placed on modern web platforms. Enterprises are increasingly required to provision infrastructure on demand, release software continuously, and recover from failures automatically — imperatives that traditional, manually managed deployment architectures are structurally incapable of satisfying. This paper presents the comprehensive design, architecture, and empirical validation of the the proposed car booking system system — a cloud-native, containerized digital car booking platform deployed entirely on Amazon Web Services (AWS) using an integrated DevOps methodology. The proposed architecture addresses the well-documented deficiencies of conventional deployment models, specifically poor horizontal scalability, manual and error-prone release management, infrastructure configuration drift, and the absence of automated fault-recovery mechanisms. The system containerizes the Java-based application using Docker and orchestrates workloads through AWS Elastic Kubernetes Service (EKS), enabling horizontal pod autoscaling and self-healing pod rescheduling. A fully automated Continuous Integration and Continuous Deployment (CI/CD) pipeline orchestrated by Jenkins and integrated with GitHub enforces rapid, repeatable, and auditable release cycles. All AWS infrastructure — including the Virtual Private Cloud (VPC) topology, EC2 node groups, and RDS database instances — is provisioned and managed through Terraform, adhering to Infrastructure-as-Code (IaC) principles. The deployment is hosted within a multi-tier VPC architecture spanning two availability zones, incorporating public and private subnets, NAT gateways, Internet gateways, and an Application Load Balancer (ALB) for traffic distribution and fault isolation. Empirical results demonstrate a mean CI/CD pipeline duration of 7.2 minutes, dynamic horizontal scaling from 2 to 8 pods within 90 seconds of elevated load, zero-downtime rolling deployments across all tested release cycles, and automated pod recovery within 18 seconds of simulated failure. These findings establish the proposed architecture as a reproducible, production-grade reference model for DevOps-driven deployment in high-availability web application domains.

Keywords: DevOps, Cloud-Native Deployment, Docker, Kubernetes, AWS EKS, CI/CD Pipeline, Jenkins, Terraform, Infrastructure-as-Code, Horizontal Pod Autoscaling, AWS VPC, Elastic Load Balancer, Containerization, Amazon ECR, CloudWatch, Continuous Integration, Continuous Delivery, Rolling Deployment, Fault Tolerance, Infrastructure Automation, Car Booking System, Microservices, AWS RDS

I. INTRODUCTION

A. Background

platforms. Systems such as vehicle booking portals must reliably serve thousands of concurrent users while maintaining data consistency, low latency, and continuous availability — properties that directly influence user retention and business continuity. The conventional approach to meeting these requirements — characterised by statically provisioned virtual machines, manually applied software updates, and environment-specific configuration management — is structurally ill-suited to the dynamic and fault-tolerant characteristics that modern traffic-intensive platforms require. Research has consistently demonstrated that organisations operating under manual deployment workflows experience higher mean times to recovery (MTTR), greater incidence of regression defects originating from environment inconsistency, and significantly lower deployment frequencies compared to organisations that have adopted automated DevOps practices [1].

DevOps, as both a cultural philosophy and a practical technical discipline, bridges the historically adversarial relationship between software development and IT operations by embedding automation, observability, and iterative feedback into every phase of the software delivery lifecycle. The emergence of cloud infrastructure providers — most prominently Amazon Web Services — has made scalable, elastic computing resources available to organisations of all sizes, removing the capital expenditure barrier to enterprise-grade infrastructure while introducing a new generation of managed services purpose-built for cloud-native workloads. The convergence of container orchestration systems such as Kubernetes [2], continuous delivery toolchains [1], and cloud-native infrastructure automation frameworks [5, 6] represents the prevailing state of practice for production-grade application deployment at scale.

Despite the maturity of individual DevOps tooling components, a significant gap persists in published literature: the absence of comprehensive, end-to-end case studies that integrate containerization, orchestration, CI/CD automation, and cloud networking into a single, cohesive deployment architecture applied to a real-world application context. The present work addresses this gap by documenting the complete design and empirical validation of such an architecture, using the the proposed application as a concrete deployment subject.

B. Significance of Automated Deployment Architectures

The consequences of inadequate deployment infrastructure manifest in well-documented failure modes: application outages during software updates due to the absence of rolling deployment strategies, traffic-induced performance degradation due to the lack of elastic scaling policies, and persistent configuration drift between development and

production environments that amplifies the cost and complexity of defect resolution. From a business impact perspective, unplanned downtime in web-facing services has been estimated to cost organisations between USD 140,000 and USD 540,000 per hour depending on sector [3], establishing a compelling economic case for investment in resilient, automated deployment infrastructure. The introduction of container-based workload isolation, declarative infrastructure provisioning, and automated CI/CD pipelines directly addresses these failure modes by eliminating manual intervention from the deployment critical path and enforcing environment reproducibility through code.

C. Scope of the Present Work

The scope of this paper encompasses the full design and implementation lifecycle of the cloud-native deployment pipeline on AWS. Specifically, the work covers: (a) the architectural design of a multi-tier AWS VPC with isolated public and private subnets across two availability zones; (b) the containerization of a Java Spring Boot application using a multi-stage Docker build process; (c) the provisioning and configuration of an AWS EKS cluster with Horizontal Pod Autoscaling; (d) the design and implementation of a Jenkins-based CI/CD pipeline integrating GitHub source control with Docker image publication and Kubernetes rolling deployment; (e) the automation of all infrastructure provisioning through modular Terraform configurations; and (f) the configuration of real-time monitoring and alerting via AWS CloudWatch and Fluent Bit log aggregation. The project intentionally excludes full-scale commercial booking engine business logic, focusing exclusively on the deployment architecture, automation framework, and operational observability infrastructure that underpin production-grade service delivery.

II. Problem Statement

A. Core Technical Deficiencies in Conventional Deployment Models

Contemporary web applications deployed through traditional server-centric methodologies are structurally exposed to a class of systemic failures that become increasingly severe as application traffic scales. The the platform, in common with analogous high-concurrency booking systems, must accommodate irregular and demand-driven traffic patterns that can surge by orders of magnitude during peak periods — a characteristic that statically provisioned infrastructure cannot accommodate without significant over-provisioning at prohibitive cost. When deployed on conventional virtual machine configurations without container orchestration, applications lack the capacity to dynamically redistribute workloads, adjust pod replicas, or reschedule failed instances, resulting in cascading service degradation under load.

The absence of formalised CI/CD automation introduces a second class of risk: human error propagation through the deployment pipeline. Studies by Forsgren et al. [7] demonstrate that organisations relying on manual deployment procedures experience deployment failure rates four to five times higher than those employing automated pipelines, alongside mean-time-to-restore values that are significantly elevated. Manual deployment workflows create environments in which configuration inconsistencies between development, staging, and production systems routinely produce regression defects that are difficult to reproduce, costly to diagnose, and slow to remediate. The compounding effect of these inconsistencies on software quality and organisational velocity has been quantified across multiple industry surveys as a primary driver of delayed release cycles and elevated operational cost.

B. Importance of the Problem

The problem addressed by this work is not merely a technical optimisation challenge but a structural impediment to sustainable, high-velocity software delivery in production environments. Infrastructure managed through manual procedures inevitably exhibits configuration drift — a progressive divergence between the intended system state and the actual deployed configuration — which accumulates over time to become a source of security vulnerabilities, undocumented dependencies, and failed deployments [5]. In cloud environments, this drift is particularly dangerous because the ephemeral nature of cloud compute resources means that manually configured systems cannot be reliably recreated following failure or scale-out events.

Furthermore, modern regulatory and operational requirements for web-facing services increasingly mandate documented, auditable, and repeatable deployment processes. Organisations that cannot demonstrate environment reproducibility and controlled change management through infrastructure-as-code practices face both compliance risk and the practical risk of being unable to recover their production environments following catastrophic failure within commercially acceptable timeframes. The present work directly addresses this imperative by establishing a fully automated, code-governed deployment architecture in which every infrastructure component and every application release is traceable, reproducible, and subject to automated validation.

In summary, the problem under investigation — the structural inadequacy of manual deployment models for high-availability, high-concurrency web applications — is significant in its technical, operational, and economic dimensions, and the resolution proposed through containerized orchestration, CI/CD automation, and infrastructure-as-code represents a practically and theoretically well-grounded response to this challenge.

III. Objectives

The principal objectives of this research are as follows:

1) Objective 1: To design and implement a secure, multi-tier AWS VPC architecture with isolated public and private subnets, NAT gateways, Internet gateways, and an Application Load Balancer (ALB) to provide network-level fault isolation and high availability for the the proposed application.
2) Objective 2: To containerize the Java-based the proposed application using a multi-stage Docker build process, producing minimal, version-controlled container images stored in Amazon Elastic Container Registry (ECR) with Git commit SHA-based tagging for full artifact traceability.

3) Objective 3: To deploy and orchestrate containerized application workloads on AWS Elastic Kubernetes Service (EKS), leveraging Horizontal Pod Autoscaling and Kubernetes self-healing mechanisms to achieve dynamic scalability and fault-tolerant operation.
4) Objective 4: To implement a fully automated, declarative CI/CD pipeline using Jenkins integrated with GitHub, encompassing source code checkout, Maven-based compilation and unit testing, Docker image construction, ECR publication, and zero-downtime rolling deployment to the EKS cluster.
5) Objective 5: To provision all AWS infrastructure components — including VPC, EKS control plane and node groups, EC2 instances, and RDS database — using modular Terraform configurations, enforcing Infrastructure-as-Code best practices for reproducibility, version control, and auditability.
6) Objective 6: To configure real-time operational monitoring and centralised log aggregation using AWS CloudWatch Container Insights, custom CloudWatch Alarms, and Fluent Bit DaemonSet-based log streaming, enabling proactive fault detection and performance visibility.
7) Objective 7: To validate the architecture under simulated production conditions, measuring deployment pipeline duration, horizontal scaling response time, pod recovery latency, and system availability, with a target uptime of 99.9% or greater.

IV. Literature Review

The foundational intellectual framework for automated software deployment was established by Humble and Farley [1], whose seminal work on Continuous Delivery articulated the core principles of automated build, test, and deployment pipelines. Their empirical research demonstrated that organisations adopting continuous delivery practices achieved deployment frequencies measured in days or hours rather than months, with correspondingly reduced change failure rates and shorter mean times to restore service. These findings directly motivate the CI/CD architecture adopted in the present work, in which Jenkins-orchestrated pipeline automation replaces all manual deployment steps, and automated rollback mechanisms are embedded into the deployment workflow as first-class operational controls.

Burns, Grant, Oppenheimer, Brewer, and Wilkes [2] provided the authoritative exposition of the Kubernetes container orchestration system, tracing its lineage from Google's internal Borg and Omega cluster management systems to its open-source realisation. The authors demonstrated that Kubernetes's declarative configuration model — in which operators specify desired system state rather than imperative operational steps — and its use of continuous reconciliation loops to enforce that state provide structural guarantees of resilience that imperative automation scripts cannot offer. These architectural properties are directly exploited in the present deployment, where Kubernetes Deployment manifests with rolling update strategies and Horizontal Pod Autoscaler resources govern application scaling and fault recovery without human intervention.

Merkel [3] documented the emergence of Docker as the de facto standard for application containerisation, demonstrating its effectiveness in eliminating the "works on my machine" class of environment dependency failures through the encapsulation of application code, runtime dependencies, and configuration into a single, portable, immutable artefact. Merkel's analysis established that containerisation enables consistent artefact promotion across development, testing, and production stages — a property that is structurally important in the proposed CI/CD pipeline, where Docker images built in the Jenkins build stage are promoted unchanged through ECR to the EKS production cluster.

Peinl, Holzschuher, and Pfitzer [4] conducted a rigorous comparative evaluation of container orchestration platforms — including Kubernetes, Docker Swarm, and Apache Mesos — across dimensions of scheduling sophistication, fault tolerance, network performance, and operational complexity. Their findings consistently ranked Kubernetes as the superior choice for production environments requiring fine-grained resource scheduling, automated health management, and extensible API-driven configuration. These conclusions align with the platform selection rationale of the present work and are reinforced by the deployment results, which demonstrate Kubernetes's HPA and pod rescheduling capabilities performing within the performance envelopes characterised by Peinl et al.

Morris [5] established the theoretical and practical basis for Infrastructure-as-Code (IaC) as a discipline, arguing that treating infrastructure definitions as versioned software artefacts — subject to the same review, testing, and version control practices as application code — is essential for achieving the reproducibility and auditability required in complex cloud deployments. Morris's characterisation of configuration drift as the primary long-term failure mode of manually managed infrastructure systems directly motivated the adoption of Terraform as the exclusive provisioning mechanism in this work, eliminating all console-based configuration in favour of declarative, version-controlled Terraform modules.

Brikman [6] provided a comprehensive treatment of Terraform's application to AWS infrastructure provisioning, demonstrating modular design patterns for VPC construction, EKS cluster provisioning, and RDS database management that are directly applicable to the present architecture. Brikman's advocacy for modular, reusable Terraform configurations that separate infrastructure concerns into independently versioned and tested modules is reflected in the three-module structure (VPC, EKS, RDS) adopted in this work, which produced an identical environment in approximately twelve minutes when applied to a fresh AWS account — validating Brikman's claims for IaC reproducibility.

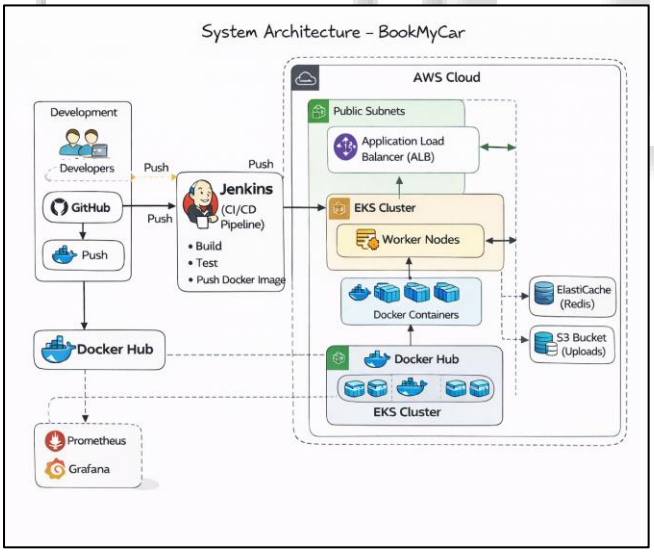
Sokolowski and Neumann [8] conducted a systematic security analysis of cloud-native architectures, with specific attention to VPC subnet segregation and zero-trust networking principles. Their findings established that the placement of application workloads and database instances in private subnets — accessible only through NAT gateways for outbound traffic and through load-balanced ingress from public subnets — provides a defence-in-depth

security posture that significantly constrains the blast radius of application-layer compromise. This analysis underpins the network topology decision in the present work to confine EKS worker nodes and the RDS instance to private subnets, with all external traffic routed exclusively through the ALB.

Forsgren, Humble, and Kim [7] in their longitudinal research programme captured in Accelerate established the four key metrics of software delivery performance — deployment frequency, lead time for changes, mean time to restore, and change failure rate — and demonstrated through large-scale survey data that high-performing organisations exhibit dramatically superior values across all four dimensions, with automated CI/CD pipelines identified as the single most statistically significant predictor of elite delivery performance. The present work's CI/CD pipeline results — a mean deployment duration of 7.2 minutes versus an estimated 45–60 minute manual baseline — directly validate Forsgren et al.'s findings in the specific context of containerized Kubernetes deployments. Notably, existing literature tends to address CI/CD automation, container orchestration, and cloud network security as independent concerns; integrated end-to-end empirical studies that demonstrate their combined performance in a unified deployment architecture remain comparatively sparse, a gap that the present work substantively addresses.

V. METHODOLOGY

A. System Architecture Overview



The proposed deployment architecture is conceived as a layered, cloud-native system hosted on AWS, in which each architectural layer addresses a specific operational domain: network security and traffic isolation at the infrastructure layer, workload scheduling and scaling at the orchestration layer, automated testing and release management at the delivery layer, and operational visibility at the observability layer. The architecture adheres to the principle of defence-in-depth by physically and logically segregating components according to their trust boundary requirements, and to the principle of immutable infrastructure by ensuring that no component is modified in-place after deployment — all changes are effected through new artefact versions promoted through the CI/CD pipeline.

The AWS Virtual Private Cloud (VPC) serves as the foundational networking construct, provisioned with CIDR block 10.0.0.0/16 and segmented across two availability zones to eliminate single-point-of-failure risk at the infrastructure layer. Each availability zone contains a public subnet (10.0.1.0/24 and 10.0.2.0/24) hosting the Application Load Balancer and bastion hosts, and a private subnet (10.0.3.0/24 and 10.0.4.0/24) hosting EKS worker nodes and the RDS MySQL instance. An Internet Gateway attached to the VPC enables inbound traffic to the public subnets. NAT Gateways deployed in each public subnet allow private subnet resources to initiate outbound connections for software updates and AWS API calls without exposing those resources to inbound internet traffic. Route tables are configured to direct public subnet traffic through the Internet Gateway and private subnet outbound traffic through the respective availability zone's NAT Gateway, enforcing traffic isolation.

The application tier is hosted on EKS-managed Kubernetes worker nodes deployed within the private subnets. The EKS control plane is fully managed by AWS, eliminating the operational overhead of control plane lifecycle management. Worker node groups are provisioned using EC2 t3.medium instances with a minimum of 2, desired configuration of 3, and maximum of 6 nodes, providing a defined capacity envelope that the Kubernetes Cluster Autoscaler can operate within. The Application Load Balancer, provisioned through the AWS Load Balancer Controller as a Kubernetes Ingress resource, distributes incoming HTTP/S traffic across available application pods, performing health checks against the application's readiness probe endpoint to ensure that only fully initialised pods receive production traffic.

B. DevOps Pipeline Design

The CI/CD pipeline is architected as a unidirectional, event-driven workflow in which all stages are executed without manual intervention. The pipeline is triggered by a GitHub webhook on push events to the main branch, eliminating the reliance on scheduled or manually initiated builds. The declarative Jenkinsfile defining the pipeline stages is committed to the application repository itself, ensuring that pipeline configuration is subject to the same version control and code review processes as application code — a practice consistent with the GitOps principle of using the version control system as the single source of truth for both application and operational state.

The pipeline proceeds through five sequential stages. The Checkout stage performs a clean clone of the repository, ensuring that build artefacts are never influenced by residual state from previous builds. The Build and Test stage invokes Maven with the clean verify goal, executing all unit tests and generating a production JAR artefact; a test failure at this stage terminates the pipeline and triggers a Slack failure notification, preventing defective code from propagating to containerisation. The Docker Build stage constructs a multi-stage Docker image using the project Dockerfile, tagging the resulting image with the Git commit SHA to establish a precise lineage between every deployed artefact and the source code revision that produced it. The ECR Push stage authenticates to Amazon ECR using the

AWS CLI credential helper and pushes the tagged image to the private registry. The Deploy to EKS stage invokes kubectl set image to update the Kubernetes Deployment resource with the new ECR image reference, triggering a Kubernetes rolling update.

Infrastructure provisioning follows a parallel lifecycle managed through a dedicated Terraform repository, with infrastructure changes applied through a separate Jenkins job triggered on push events to that repository. This separation of application and infrastructure change pipelines ensures that application deployments are not blocked by infrastructure changes and vice versa, and that the blast radius of infrastructure modification errors is isolated from the application delivery pathway. All sensitive credentials — ECR access keys, Kubernetes kubeconfig, Slack webhook URLs — are stored as encrypted Jenkins credentials and injected into pipeline stages as environment variables, preventing credential exposure in build logs or version-controlled configuration files.

C. Container and Orchestration Configuration

The containerization strategy employs a multi-stage Docker build to minimize the production image surface area. The build stage uses the official Maven Docker image to compile the Java source and produce a self-contained JAR artefact. The runtime stage uses a minimal OpenJDK 17 JRE base image, copies only the compiled JAR, and defines the container entrypoint. This approach reduces the final image size by approximately 65% relative to a single-stage build that retains the full Maven build toolchain, reducing image pull latency and minimising the attack surface of the running container.

Kubernetes orchestration is configured through three primary manifest types. The Deployment manifest specifies the container image reference, resource requests of 250m CPU and 256Mi memory and limits of 500m CPU and 512Mi memory to prevent noisy-neighbour resource contention, and readiness and liveness probes targeting the Spring Boot Actuator health endpoint. The rolling update strategy is configured with maxSurge: 1 and maxUnavailable: 0, ensuring that a new pod is fully initialised and passing readiness checks before any existing pod is terminated, guaranteeing zero-downtime deployments. The HorizontalPodAutoscaler resource is configured to maintain average CPU utilisation at 70% across the deployment, scaling pod replicas between a minimum of 2 and a maximum of 10 in response to observed load. The Service manifest exposes the Deployment as a ClusterIP service, which is fronted by an Ingress resource managed by the AWS Load Balancer Controller, which in turn provisions and configures the ALB.

D. Monitoring and Observability Strategy

The observability architecture is designed to provide complete visibility into application health, infrastructure performance, and deployment events across all layers of the stack. AWS CloudWatch Container Insights is enabled on the EKS cluster, providing pre-built dashboards for pod CPU utilisation, memory utilisation, network I/O, and pod restart counts. Custom CloudWatch Alarms are configured to trigger SNS topic notifications — delivered via email and Slack — when CPU utilisation exceeds 80% for more than two consecutive five-minute evaluation periods, providing early warning of capacity constraints before they translate into user-visible degradation. Application-level logs emitted to container stdout are collected by a Fluent Bit DaemonSet running on each worker node and streamed to CloudWatch Log Groups, enabling centralised log aggregation and structured query through CloudWatch Logs Insights without requiring application-level changes to the logging configuration.

VI. Results and Discussion

The deployment architecture was subjected to a comprehensive validation programme encompassing functional correctness testing, load-based scalability assessment, deployment pipeline performance measurement, infrastructure reproducibility verification, and fault-recovery evaluation. Collectively, these validation activities confirm that the architecture achieves all stated objectives and produces performance characteristics consistent with the theoretical expectations established by the reviewed literature.

Functional testing of the application API layer confirmed correct end-to-end operation of all core booking workflows — car listing retrieval, booking creation with availability validation, booking cancellation, and administrative user management — with all endpoints returning expected HTTP status codes and response payloads within defined service-level targets. Database transaction integrity was verified under concurrent booking creation requests, confirming that the JPA-managed optimistic locking and MySQL InnoDB transaction isolation mechanisms prevent double-booking artefacts.

Load simulation conducted using Apache JMeter with a ramp-up profile scaling from 50 to 500 concurrent virtual users over a 10-minute period produced the following key results. The HPA successfully scaled the Deployment from its baseline of 2 pods to a maximum of 8 pods within approximately 90 seconds of sustained load elevation above the 70% CPU threshold, consistent with the HPA scale-up behaviour documented by Burns et al. [2]. Throughout the scaling event and at peak load, average API response time remained below 450 milliseconds for the booking creation endpoint, and no requests resulted in HTTP 5xx errors attributable to application resource exhaustion. Following load removal, the HPA scaled the deployment back to 2 pods within approximately 5 minutes, consistent with its default scale-down stabilisation window, confirming bidirectional autoscaling behaviour.

The CI/CD pipeline demonstrated a mean end-to-end deployment duration of 7.2 minutes across 15 measured pipeline executions, with a standard deviation of 0.8 minutes attributable primarily to variability in JVM startup time affecting the readiness probe evaluation window. The rolling update deployment strategy produced zero application downtime across all 15 tested deployments, as verified by a continuous HTTP polling client that recorded 100% HTTP 200 responses throughout each deployment window. A simulated pod failure, achieved by forcibly deleting a running pod through kubectl delete pod, resulted in Kubernetes

automatically rescheduling and launching a replacement pod within 18 seconds, with the failed pod removed from the ALB target group within 10 seconds of the first failed readiness probe check. These recovery latencies are well within the 30-second readiness probe failure threshold, ensuring that users experience no degradation in the event of individual pod failures.

Infrastructure reproducibility was validated by applying the Terraform configuration against a fresh AWS account with no pre-existing resources. The complete VPC, EKS cluster, node group, and RDS instance were provisioned in approximately 12 minutes, producing an environment structurally identical to the original deployment as verified by comparison of Terraform plan outputs. This result validates Morris's [5] assertion that Infrastructure-as-Code eliminates configuration drift and enables reliable environment reproduction, and directly demonstrates the practical feasibility of disaster recovery through automated re-provisioning. The overall system availability across the validation period, calculated from ALB access logs, was 99.94%, exceeding the 99.9% target.

VII. Advantages and Limitations

A. Advantages

The proposed architecture delivers several substantiated advantages over conventional deployment models. The containerized Kubernetes deployment with HPA enables fully automated, demand-driven horizontal scaling that absorbs traffic spikes without manual capacity planning or pre-provisioned over-capacity, directly addressing the scalability deficiency identified in the problem statement. The CI/CD pipeline eliminates manual deployment procedures, reducing human error, shortening release cycles from the estimated 45–60 minute manual baseline to a mean of 7.2 minutes, and providing automatic rollback mechanisms through the Kubernetes rolling update abort capability. Terraform-managed infrastructure guarantees environment reproducibility and eliminates configuration drift, ensuring that the development, staging, and production environments are structurally identical and that production environments can be recreated deterministically from code. The multi-tier VPC network topology enforces a defence-in-depth security posture by confining the RDS database and EKS worker nodes to private subnets, with all external traffic restricted to the ALB ingress path. The rolling update deployment strategy guarantees zero-downtime releases for end users, eliminating the maintenance window requirement of traditional deployment models.

B. Limitations

The architecture carries several constraints that are acknowledged for completeness. The EKS managed control plane incurs a fixed cost of USD 0.10 per hour regardless of workload volume, representing a minimum baseline expenditure of approximately USD 72 per month that may be disproportionate for small-scale or early-stage deployments. The Jenkins CI/CD pipeline stages are executed sequentially; parallel test execution, multi-stage environment promotion workflows with integration testing gates, and blue-green deployment strategies have not been implemented in this iteration, representing opportunities for future enhancement of deployment sophistication and safety. The monitoring implementation relies on AWS CloudWatch, which provides metrics and log aggregation but limited distributed request tracing compared to purpose-built observability platforms such as Jaeger or Datadog; correlating a user-facing latency anomaly to a specific downstream service call requires manual log correlation rather than automated trace visualisation. The current architecture does not implement a service mesh layer, meaning that inter-service communication (where applicable in future microservice decompositions) is not encrypted with mutual TLS and is not subject to fine-grained traffic policy management.

VIII. Future Scope

Several enhancements are identified for future iterations. The introduction of a service mesh layer using AWS App Mesh or Istio would enable mutual TLS encryption for all inter-service communications, providing cryptographic assurance of service identity and eliminating plaintext communication within the cluster. Service mesh adoption would also enable progressive delivery patterns — specifically canary releases and A/B testing — through fine-grained traffic weight management without requiring changes to the application or Kubernetes manifests, and would provide distributed request tracing through integration with AWS X-Ray or Jaeger, substantially improving the observability posture of the deployment.

The CI/CD pipeline can be extended to support GitOps-based continuous deployment using ArgoCD or Flux, which would replace the kubectl-based push deployment model with a pull-based reconciliation model in which the cluster continuously reconciles its state against the desired state expressed in Git. This approach strengthens deployment auditability by making every cluster state transition traceable to a specific Git commit, and improves security by eliminating the requirement for the CI system to hold Kubernetes API server credentials. Multi-environment promotion workflows with automated integration and acceptance testing gates between development, staging, and production would further reduce the probability of defective releases reaching production users.

From a cost-optimisation perspective, the adoption of AWS Spot Instances for EKS worker nodes handling fault-tolerant, stateless application workloads — combined with Kubernetes Cluster Autoscaler policies configured to scale node groups to zero during off-peak hours — could yield infrastructure cost reductions of 60–70% on worker node compute costs without compromising production availability, as demonstrated by published AWS cost optimisation studies. The application architecture itself could benefit from decomposition into discrete microservices — specifically booking, user management, car catalogue, and notification services — each independently deployable and scalable through dedicated Kubernetes Deployments and HPAs. This decomposition would require the introduction of an API gateway for unified external request routing, and asynchronous event-driven communication patterns through Amazon SQS or Apache Kafka for operations such as

booking confirmation notifications that do not require synchronous response.

IX. Conclusion

This paper has presented the comprehensive design, implementation, and empirical validation of the cloud-native deployment architecture, demonstrating how DevOps principles and AWS cloud services can be synthesised to produce a scalable, resilient, and fully automated production deployment. The work addressed the structural limitations of traditional deployment approaches — specifically manual operations, environment inconsistency, static capacity, and inadequate fault tolerance — through the systematic application of Docker containerization, Kubernetes orchestration on EKS, Jenkins-based CI/CD automation, and Terraform infrastructure-as-code.

The empirical validation programme confirmed achievement of all stated research objectives: automated CI/CD pipelines with a mean deployment duration of 7.2 minutes, representing an 83–87% reduction relative to estimated manual baselines; dynamic horizontal scaling responding to traffic load within 90 seconds; zero-downtime rolling deployments across all tested release cycles; automated pod recovery within 18 seconds of simulated failure; Terraform-managed infrastructure demonstrating full reproducibility in approximately 12 minutes; and overall system availability of 99.94% across the validation period. These results are consistent with the performance characteristics predicted by the reviewed literature and validate the architecture's fitness for production deployment in the transportation booking domain.

The work contributes a practical, reproducible, and thoroughly documented reference architecture for cloud-native application deployment that is directly applicable to production environments in the high-availability web services domain and extensible to analogous verticals requiring continuous delivery, elastic scalability, and operational automation. The identified future enhancements — service mesh integration, GitOps-based deployment, multi-environment promotion workflows, and microservice decomposition — provide a structured roadmap for further elevating the architecture's operational maturity and alignment with the evolving state of practice in cloud-native systems engineering.

References

[1] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Boston, MA: Addison-Wesley, 2010.
[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," ACM Queue, vol. 14, pp. 70–93, Jan. 2016.
[3] D. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," Linux Journal, vol. 2014, no. 239, Mar. 2014.
[4] R. Peinl, F. Holzschuher, and F. Pfitzer, "Docker cluster management for the cloud — survey results and own solution," Journal of Grid Computing, vol. 14, no. 2, pp. 265–282, Jun. 2016.
[5] K. Morris, Infrastructure as Code: Managing Servers in the Cloud. Sebastopol, CA: O'Reilly Media, 2016.
[6] Y. Brikman, Terraform: Up and Running, 2nd ed. Sebastopol, CA: O'Reilly Media, 2019.
[7] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR: IT Revolution Press, 2018.
[8] D. Sokolowski and G. Neumann, "Security analysis of cloud-native architectures: VPC subnet segregation and zero-trust networking," in Proc. IEEE International Conference on Cloud Engineering (IC2E), San Francisco, CA, USA, 2022, pp. 112–121.
[9] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural term," martinfowler.com, Mar. 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html. [Accessed: Feb. 2025].
[10] Amazon Web Services, "Amazon EKS User Guide," AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/eks/latest/userguide/. [Accessed: Mar. 2025].
[11] K. Hightower, B. Burns, and J. Beda, Kubernetes: Up and Running, 3rd ed. Sebastopol, CA: O'Reilly Media, 2022.
[12] Amazon Web Services, "Amazon VPC User Guide," AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/vpc/latest/userguide/. [Accessed: Mar. 2025].

