Online Bookstore

Yash Gadodia, Amlan Baruah, Shobika Rajeskanna, Boyang Wang, and Vaibhav Patil

Newcastle University, Newcastle Upon Tyne, UK

Abstract. The landscape of software development has been revolutionized through cloud computing, microservices, and automation-based practices. As more organizations move towards cloud-native architectures, the need for systems that demonstrate scalability, resilience, and velocity is of the utmost importance. However, conventional approaches to software development fail to meet such demands, leading to persistent issues like integration complexity, longer deployment times, and higher failure rates. These constraints have resulted in lengthy release cycles and decreased operational efficiency, necessitating the adoption of more flexible and resilient development practices. The application of DevOps practices has proven useful in addressing the complexities of modern software development by providing a unifying platform that integrates development, operations, and automation to enhance deployment efficiency. Core practices such as Continuous Integration (CI), Continuous Deployment (CD), and Containerization enable the construction of systems that are readily accessible, scalable, and fault-tolerant. These practices ensure that modern applications can efficiently manage high traffic, recover quickly from outages, and roll out updates with minimal downtime. This project entails the development of a cloud-native bookstore application using the MERN (MongoDB, Express.js, React.js, Node.js) stack, deployed on Google Kubernetes Engine (GKE). The application is built with Blue-Green Deployment, Horizontal Pod Autoscaling (HPA), and persistent storage via StatefulSets to ensure seamless upgrading, auto-scaling, and fault tolerance. The Jenkins-based CI/CD pipeline automates the build and deployment processes, while Velero provides backup solutions. Kubernetes monitoring systems such as Prometheus and Grafana enhance system resiliency and observability. This report presents a detailed examination of the design, implementation, and testing approaches followed during the project, demonstrating how DevOps principles can be leveraged to develop scalable, fault-tolerant, and automated cloud systems.

1 Introduction

The Cloud-Based Bookstore App provides an affordable, scalable, and high-availability online bookstore solution, driven by state-of-the-art microservices architecture and best practices in DevOps. The solution integrates Google Kubernetes Engine (GKE), CI/CD automation, and a blue-green release strategy to enable seamless upgrades without service disruption, adhering to industry best practices for deploying and managing software. This project employs a microservices architecture with containers, dividing the system into standalone services such as User Management, Catalogue, and Order Processing, each running in separate Kubernetes Pods [1]. The system's backend is implemented using NodeJS and ExpressJS, while the frontend is built with React.js to provide a responsive and interactive user interface. For improved operational efficiency and reliability, Jenkins is used for CI/CD automation, MongoDB as the persistent database, and Prometheus and Grafana for monitoring system performance. The blue-green deployment strategy is a key feature of this setup, allowing updates to be deployed without affecting active users by gradually transferring traffic between two environments (blue and green) after validation [2]. The primary objective of this project is to ensure high availability and fault tolerance for the bookstore application while providing scalability through Kubernetes auto-scaling features such as Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler. Automated deployments are achieved via a CI/CD pipeline integrating Google Artifact Registry, enabling rapid and reliable software delivery. The project also includes a comprehensive monitoring setup with Prometheus, Grafana, and the Kubernetes Dashboard to track system performance and resource utilization in real time. Data persistence and disaster recovery are handled through MongoDB StatefulSet and Velero backups, ensuring the system remains fault-tolerant and resilient to disruptions [3]. The motivation behind this project is to develop a real-world cloud-native application capable of scaling with increasing user loads while maintaining system reliability. By leveraging a microservices-based and Kubernetes-powered architecture, this system serves as a foundation for building highly available, automated, and scalable cloud-native applications. The subsequent sections will elaborate on the design, development, and evaluation of this bookstore application, highlighting challenges and solutions for creating a robust cloud-based system.

2 Project Specification

2.1 Overview

The Cloud-Based Bookstore Application is built on top of a Kubernetes framework that is achieved through Google Kubernetes Engine (GKE), with a focus on important considerations including scalability, high availability, and automation. The application utilizes a microservices architecture and blue-green deployment techniques in combination with Continuous Integration/Continuous Deployment (CI/CD) pipelines, thus enabling a seamless flow with minimal downtime and improved efficiency in terms of resources. In contrast to traditional architectures that are monolithic, those present here have issues with limited scalability, longer downtimes related to update operations, and inefficient usage of resources. As a cloud-born application by design, the Cloud-Based Bookstore Application efficiently addresses such inefficacies.

The architecture of the system consists of several stand-alone services that deal with different business operations. The user interface that is modern and interactive has been built using React.js [4]. The backend is designed using a microservices architecture with several components like the Catalog Service, Order Service, and User Service with each component being a different Pod in Kubernetes [1]. The database architecture uses MongoDB and is done through a StatefulSet that allows it to store persistently while maintaining the required attributes of consistency and availability.

2.2 Functional Specifications

The Cloud-Based Bookstore Application offers a wide range of features to make users more convenient, simplify transaction processing, and minimize challenges that come with managing inventory. The user management feature provides safe registration, authentication, and management of profiles through applying security measures using JSON Web Token technology. In addition, using Role-Based Access Control (RBAC) allows regulated access in accordance with user permissions to enhance security measures and increase user convenience.

The bibliographic cataloging system offers administrators the capability to create, edit, and delete book records. In addition to that, customer account capability enables users to browse through the catalog, thus increasing their efficiency and effectualness in their searching operations. The Order Processing System enables users to place products in their shopping cart and place orders at once. The Order Service, on the other hand, takes care of managing orders processing, tracking inventory levels, and shipping orders to minimize discrepancies with the Catalog Service to ensure inventory consistency.

The system uses blue-green deployment techniques that allow software upgrades without causing downtime to maintain continuous availability of services. In addition to that, the application uses monitoring and auditing capabilities through Prometheus, Grafana, and Kubernetes Dashboard that allow real-time performance testing and identify failures. The system also uses MongoDB StatefulSet and Velero to achieve data persistence and create backup routines to reduce data loss and enhance disaster recovery operations.

2.3 Non-Functional Requirements

Apart from meeting functional requirements, the system complies with a set of non-functional requirements to ensure efficiency, fault tolerance, and security [5]. The Kubernetes Horizontal Pod Autoscaler and Cluster Autoscaler have been designed to automatically scale with momentary fluctuations in workload demand, thus maintaining optimum performance irrespective of traffic volume fluctuations. The use of Docker's containerization combined with Kubernetes orchestration ensures consistency in all development, testing, and deployment phases, while eliminating the risks of configuration and delivery complexities. Automation of CI/CD processes using Jenkins and Google Artifact Registry simplifies software delivery, reduces the chances of failure, and enhances overall operational efficiency. The use of real-time system monitoring through Prometheus and Grafana enables timely performance evaluation, thus enabling the detection and rectification of problems before they affect users.

2.4 System Architecture and Deployment Strategy

The bookstore application is built on a cloud-based platform using a microservices architecture that separates every backend service into separate Kubernetes Pods. The backend services include the Catalog Service, Order Service, and User Service that can run separately without dependency after being deployed to increase fault tolerance. The frontend client is written in React.js that communicates with backend services using a RESTful API to maintain modularity and scalability. MongoDB is used as a StatefulSet using Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) to maintain consistency and integrity of information.

Automation of the deployment process is enabled through CI/CD pipelines with Jenkins as their monitor [6]. The pipeline automates building, testing, and deploying software processes in such a way that it provides timely and reliable software upgrades. Blue-green deployment practices allow continuous traffic management between two environments without interruption while upgrading software [2]. The handling of incoming traffic is effectively managed through Kubernetes Ingress Controller that directs traffic to precise locations and balances it accordingly.

2.5 Resource Management and Cloud Infrastructure

Bookstore Application is deployed on Google Kubernetes Engine (GKE using e2-medium machines with 2 CPUs and 4 GB RAM. An Ingress Controller is used to efficiently manage and route traffic to servers and respond to incoming queries. The application uses Google Cloud Persistent Disks to create stable and secure storage options. In addition to that, security features are integrated using Role-Based Access Control (RBAC), limiting unauthorized changes and maintaining sensitive information's integrity.

Component	**Cloud Service**	**Instance Type**	**CPU**	**Memory**	**Storage**
Kubernetes Nodes	Google Kubernetes Engine	e2-medium	2	4	Persistent Disk
MongoDB Database	Google Cloud Storage	StatefulSet	N/A	N/A	Persistent Volume
CI/CD Pipeline	Jenkins on GCP VM	n1-standard-2	2	8	SSD

Table 1. Cloud Resource Allocation for Application Components

2.6 Monitoring and Performance Testing

Extensive monitoring is being implemented with Prometheus used as a tool to aggregate metrics, Grafana to visualize metrics in real time, and Kubernetes Dashboard to manage resources [3]. All three instruments taken together offer deep insights into the system health in general, limits of performance, and utilization [3]. In addition to this, supporting alert mechanisms and notification systems shall also be integrated to facilitate proactive detection and remediation in real time.

Performance is evaluated using K6, a load testing tool designed to generate heavy traffic to test system scalability. Key performance indicators (KPIs), alternatively called critical performance metrics (CPMs), such as response time, throughput, and error rates, are carefully examined to make the overall system effective. The capacity of the infrastructure to handle heavy loads while keeping users satisfied is evaluated using stress testing. In addition, the strength of the system against crashes and sudden halts due to failures is explored using chaos testing software.

2.7 Problems and Obstacles

During all development and implementation phases, several limitation and boundary issues were encountered that required judicious solutions to be adopted. In the initial phases, performance issues arose due to constraints related to using AWS Free Tier instances. These issues were overcome using Google Cloud Platform instances with better resources. In addition to that, routing issues due to issues with the NGINX Ingress Controller caused delays in handling requests; these constraints were overcome through routing policy optimizations and adequate configuration of ingress rules. In addition to that, issues related to MongoDB deployment-related data persistence due to inconsistencies were also encountered [7]. The introduction of StatefulSets and Persistent Volumes were found to be a good solution to those issues and hence maintaining consistency and integrity of data. Lastly, database performance improvements were realized through query optimizations and indexing with respect to connection pooling.

The Cloud-Based Bookstore Application uses basically architecture that natively supports cloud environments so that it not only enables high performance and availability but also supports scalability. In addition to this, it uses Kubernetes coupled with modern-day DevOps practices to create a robust and user-focused platform to manage online bookstore operations [1].

3 Background Research

3.1 Analysis of the Implementation of Cloud-Enabled Bookstores

Cloud commerce platforms have seen remarkable growth over recent years due to increasing demands on scalability, availability, and automation. Traditional on-premises environments that were being utilized by online bookstores have been hampered by issues such as long downtimes, scalability limitation, and longer maintenance windows within business hours. In contrast, cloud platforms take advantage of microservices, containers, and DevOps practices to enhance application performance, reduce delivery complexity, and enhance resiliency against failures.

The current research considers effective methodologies and techniques related to cloud computing, microservices architecture [8], and automation using the principles of DevOps, as well as orchestration through Kubernetes, especially with regard to blue-green deploying techniques. Such techniques allow software to be updated seamlessly to prevent interruption to services and reduce the risk of system crashes. In regard to solution implementation, this chapter offers a comprehensive review of current literature, relevant technology, and real-world examples related to bookstore applications in the context of cloud technology.

3.2 Development of Bookstore Apps

The rise of bookshop apps can largely be credited to modern-day e-commerce platforms like Amazon and Barnes & Noble that used to have their initial architecture composed of monolithic architecture typical of single-server systems. These conventional architecture patterns had several shortcomings regarding scalability, high infrastructure expenses, and efficiency in operations.

The platform of cloud technology has seen significant development with the advent of Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS), both of which offer improved scalability, efficient distribution of resources, and competent management of cloud services. The convergence of microservices architecture and Kubernetes orchestration has enabled more modularity, self-scaling of discrete components, and efficient management of applications [1].

Its shift to a microservices architecture with decentralized architecture has enabled improvements to core services like user authentication, product catalogs management, and payment handling. In line with this evolution, our cloud application bookstore utilizes Google Kubernetes Engine (GKE), continuous integration and continuous delivery (CI/CD) automated pipelines, and persistent storage mechanisms to provide high levels of availability, scalability, and reliability.

3.3 Comparison of Monolithic vs. Microservices Architecture

Monolithic architecture packages three pieces—frontend, backend, and database—within a single codebase [8]. Although such an architectural style enables fast initial development, it poses serious issues with scalability; even small changes require deploying the entire application. Monolithic applications also struggle with handling large traffic volumes that lead to performance bottlenecks and long periods of downtime.

Conversely, the microservices architectural model designs applications as separate services with each of their own designated business functionalities. The architecture enables independent scalability, enables rapid release of new features, and enables isolation of failures, thus providing a best platform to build cloud-native applications. For our project scope, key functions such as catalog management, processing orders, and authentication were built as separate microservices within Kubernetes pods.

Feature	Monolithic Architecture	Microservices Architecture
Deployment	Single unit, difficult to update	Independent service deployment
Scalability	Limited, scales as a whole	Scales per service demand
Fault Isolation	A failure affects the entire system	Failures are contained within a service
Technology Stack	Limited to a single stack	Can use different technologies per service
Maintenance	Becomes complex over time	Easier due to modular structure

Table 2. Comparison of Monolithic vs. Microservices Architecture

3.4 Cloud Computing and Kubernetes Orchestration

Progress in cloud technologies in recent times has allowed important industry players such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure to offer notable upgrades through services designed to enhance modern applications in the cloud [9]. The orchestration platform of choice now has become Kubernetes due to features such as auto-scaling capabilities, self-healing attributes, and efficient management of resources.

Kubernetes enables bookstore services to be deployed as application containers with each service being self-contained to allow proper isolation in case of fault incidents and scalability [1]. The project takes advantage of using Google Kubernetes Engine (GKE) due to ease of integration with Google Cloud services and more efficient autoscaling compared to constraints with using AWS Free Tier [10].

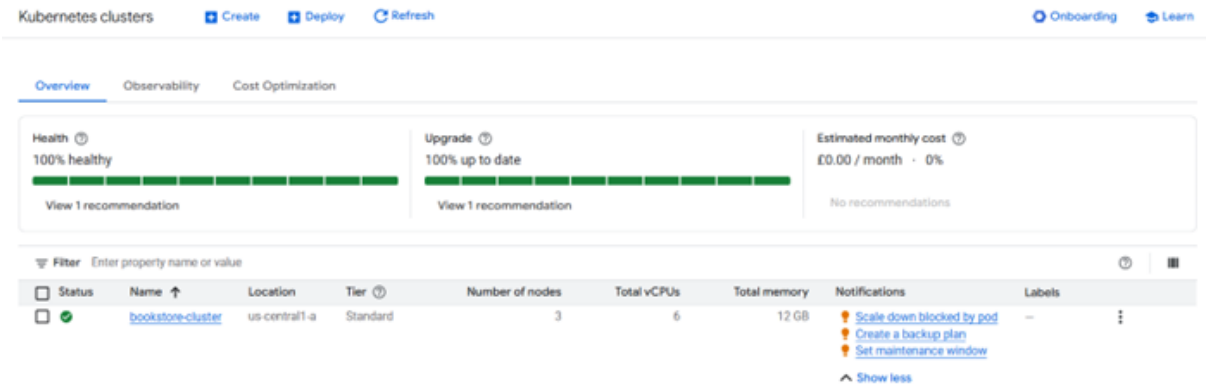


Fig. 1. High-Level System Architecture – System components interaction.

3.5 DevOps Practice: CI/CD and Blue-Green Deployment

CI/CD stands for Continuous Integration and Continuous Deployment, key principles of the DevOps platform that make it possible to release software quickly, reliably, and [2]. CI/CD pipelines improve the software development cycle by using automated routines that take care of operations related to software build, testing, and deploy. CI/CD pipelines expedite application delivery while reducing human error at the same time by reducing human intervention to a minimum.

The tool utilizes Jenkins to monitor automated continuous integration and continuous delivery-related tasks, GitHub to manage continuous control of versions, and Google Artifact Registry to allow management of containerized images [11]. Blue-green deployment is used to allow continuous software development without interruption [12]. Blue-green deployment involves keeping two similar environments: a production-ready environment that handles real traffic and a backup to deploy updates to, with testing and validation being conducted before switchover to production [6]. The user's overall user experience remains seamlessly consistent with this practice as it transfers between different versions of the application.

3.6 Data Persistence and Disaster Recovery

Traditional software used by bookstores usually utilized relational database management systems like MySQL and PostgreSQL. Although both systems have been praised for their strength, they have constraints in their abilities to horizontally scale dynamically when faced with large volumes of information. In contrast, MongoDB being a NoSQL database classifies it to be more efficient in handling unstructured information and allow horizontal scalability, thus making it more compatible with modern applications that have a cloud-based platform.

To maintain persistency of data and increase availability, MongoDB exists as a StatefulSet in Kubernetes cluster using Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) to maintain consistency of data after pod restarts [7]. In addition to that, Velero is used to automate backup operations, disaster recovery routines, and to make it possible to restore in case of system crashes [13]. All together, all such approaches create a basis of better business continuity and more reliable data.

3.7 Auto-Scaling and Performance Monitoring

Performance evaluation, coupled with effective distribution of resources, is essential building blocks to attainment of both cost-effective and reliable systems. The bookstore application uses Prometheus to monitor metrics in real time while utilizing Grafana to visualize metrics to hence make derivation of insights related to CPU utilization, capacity of available memory, and response time [3].

The self-adjustable management of resources allows the system's capability to move both up and down with respect to fluctuating demand levels. In Kubernetes, application pods are regulated by the Horizontal Pod Autoscaler (HPA) based on their CPU and RAM usage while Cluster Autoscaler manages the nodes to be scaled up to meet increased demand. Various research findings state that auto-scaling capability implementation increases application responsiveness, reduces unnecessary spending on cloud resources, and generally increases user satisfaction.

Kubernetes integration with a microservices architecture, automation through DevOps practices, and cloud-native technology provides us with the capability to build a solution that is hugely scalable, fault-resilient, and cost-effective at the same time, making it especially viable for modern-day e-commerce applications.

4 Design Description

4.1 High-Level Architecture Overview

The bookstore application is deployed in a large-scale DevOps setup that uses Google Kubernetes Engine (GKE) mostly, in combination with a blue-green deployment strategy. This setup is meant to ensure both scalability and high availability, while also making deployments possible without causing downtime, aligning with the core DevOps principles discussed in our literature review.

The architectural structure consists of five key pillars. The application context is built with containerized microservices to allow blue and green deployments to run simultaneously to make upgrades seamless. The CI/CD pipeline automates building and deploying through Jenkins to effectively practice principles of Continuous Deployment and Continuous Integration [6]. GKE is utilized to manage orchestration through a Kubernetes-based system, as discussed in previous research. MongoDB is used to store data through StatefulSet with persistent volume to effectively deal with issues of consistency in [7]. The monitoring toolset that incorporates Prometheus, Grafana, and Kubernetes Dashboard provides insightful information regarding system working dynamics and performance metrics.

This design leverages the relationship between CI/CD and containerization, where containerization ensures environment consistency, while CI/CD automates building and deployment of container images.

4.2 Application Architecture

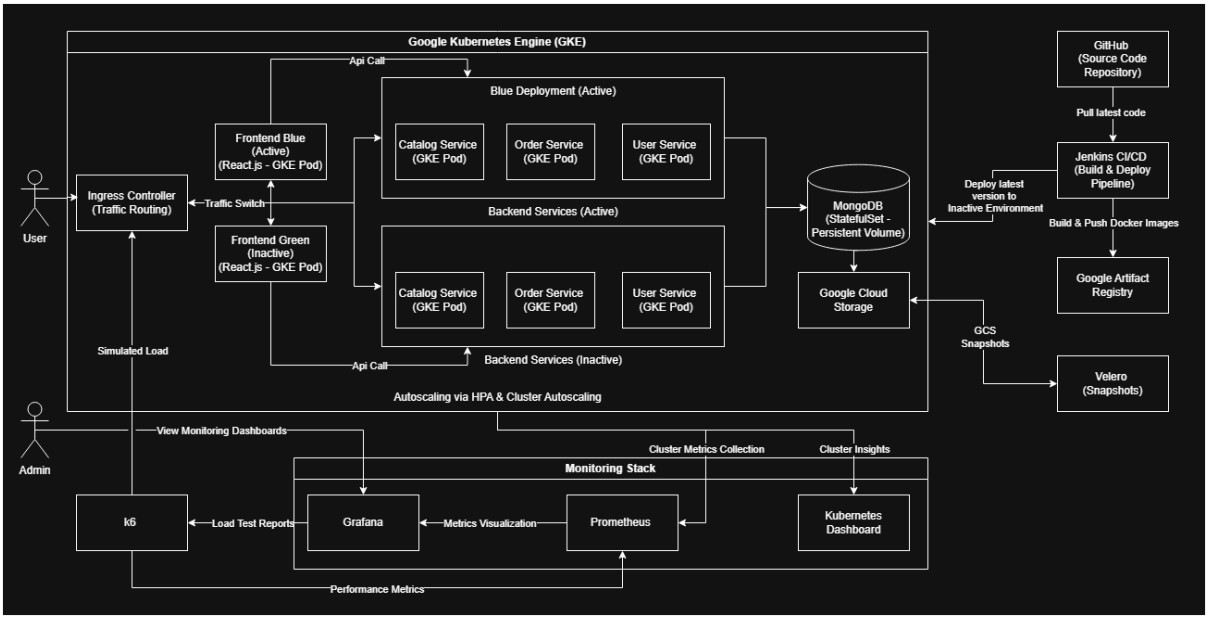


Fig. 2. Application Architecture

The bookstore application uses a microservices architecture that defines the role of each of the frontend service, backend microservices, and database layer separately [8]. The frontend application is built using React that uses blue and green enclosures to make connections to backend applications through Kubernetes service discovery. The Ingress Controller manages traffic to other environments and thus improves routing efficiency.

The backend microservices architecture is built with three separated services: Order, Catalog, and User services with each separated into a separate Kubernetes pod. All services have their business-specific logics with RESTful APIs, thus making it modular and supporting decoupled development practices. Having a blue-green strategy implemented at microservices level allows upgrades to be done in deployments without stopping current services.

To provide ongoing persistence of data, MongoDB is used as a StatefulSet instead of a standard Deployment to provide high availability and persistent storage through using Persistent Volumes (PV) and Persistent Volume Claims (PVC). The consistency of database operations is ensured through using Velero that backs up to Google Cloud Storage to subsequently provide enhanced disaster recovery capability. The microservices architecture allows every component to have separate development and application lifecycle, thus avoiding complexity related to traditional monolithic architectures. A blue-green deployment pattern is used based on previous studies on techniques that allow software to be updated without causing downtime while also enabling instant rollbacks.

4.3 DevOps Pipeline

The CI/CD pipeline created is aimed at providing a reproducible deployment process along with following established best practices in CI/CD practices. The pipeline is linked with a GitHub repository that serves as the central source control for application code along with Kubernetes manifest files [11]. Jenkins is configured to automatically pull code changes, build Docker images, and run test suites. The images built are then pushed to Google Artifact Registry for proper version control and security.

To make automation of release easier to accomplish, the pipeline identifies which environment, blue or green, is idle at present and then installs the latest version within that environment [2]. After installation, health checks are conducted to verify that the system is stable before sending user traffic to the newly deployed environment. The traffic is migrated in a gradual manner from the present working environment to the idle state, hence enabling a transition without interruptions. After finishing migrating traffic, the previous release is retired to prevent wastage of resources. The practice follows best techniques in

Continuous Integration and Continuous Deployment (CI/CD) and containerization alike to create low-risk and reproducible release operations. Blue-green practice enables full testing of new releases before being introduced to end-users, thus supporting current literature on methodological approaches.

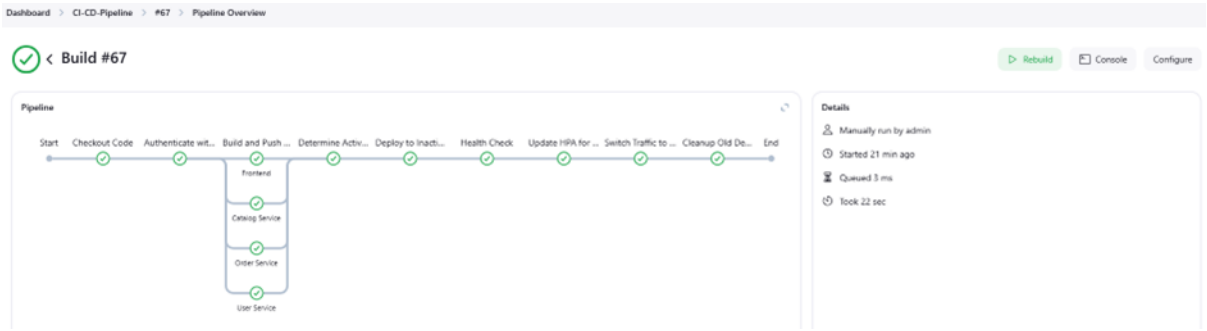


Fig. 3. Successful Pipeline build

4.4 Infrastructure Design

The creation of infrastructure is particularly driven by the benefits offered by containerization coupled with Kubernetes orchestration. The application is hosted on Google Kubernetes Engine (GKE) due to its integrated autoscale capabilities, cluster management options, and natively integrated capabilities with multiple Google Cloud services. Kubernetes provides automated application scaling, application discovery services, and self-healing capabilities with a resultant reliable system.

The Horizontal Pod Autoscaler (HPA) works by adjusting the number of pods to suit varying requirements of CPUs. In addition to that, Cluster Autoscaler changes the number of nodes based on aggregate demand of resources to maintain proper distribution of resources [14]. The move to make use of GCP over AWS was due to constraints that had been experienced at the implementation stage. The recognition that instances under AWS Free Tier (t2.micro with 1 GB RAM) were not adequate to manage Kubernetes workloads compared to those of GCP's e2-medium instances (2 vCPUs, 4 GB RAM), were adequate enough to allow efficient Kubernetes cluster operations led to this decision.

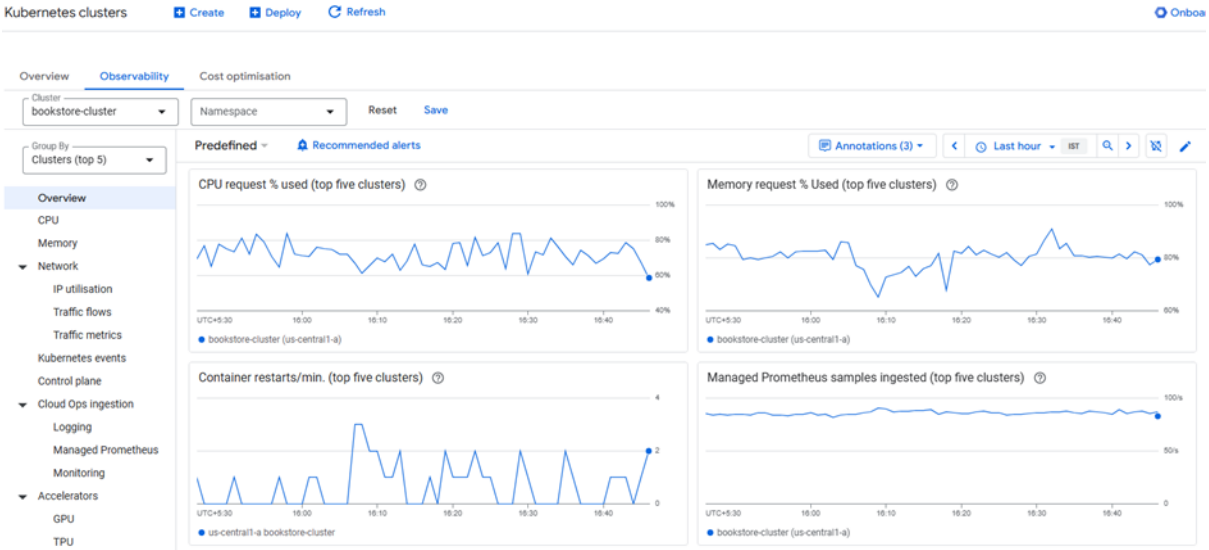


Fig. 4. GKE Cluster Overview

4.5 Monitoring and Observability

For increasing system reliability and performance, a proper framework of observability and monitoring has been implemented. All application services and Kubernetes services send their metrics to be collected through Prometheus, while Grafana provides a platform to create graphical dashboards that allow real-time monitoring of the system in an efficient way. In addition to that, Kubernetes Dashboard provides cluster-level visibility to administrators that allow system health and utilization of resources to be evaluated.

To evaluate performance at increased levels of stress, we used K6 as a load testing tool that enables auto-scaling configuration analyses at both pod level through the Horizontal Pod Autoscaler and at node level through Cluster Autoscaler. The monitoring framework is essential to proactively identify performance constraints and to ensure consistent availability and responsiveness of the application.

Our system's overall architecture is built on three core principles of DevOps: Continuous Integration, Continuous Deployment, and Containerization. The combination of these principles with the adoption of blue-green deployment enabled through the Google Kubernetes Engine equates to a very available and scalable architecture with a efficient design that provides rapid software delivery with lower risk [2].

5 Implementation Overview and Related Issues

5.1 Implementation Overview

The cloud-native application for the bookstore used Google Kubernetes Engine (GKE) for deployment using a blue-green deployment strategy. Blue-green deployment was used for the purpose of taking advantage of Continuous Integration (CI), Continuous Deployment (CD), and containerization concepts and hence providing a highly available and scalable architecture that allows upgrades without suffering downtime.

Core Infrastructure Components

The Google Kubernetes Engine cluster played the core role of being our deploy component that held within it all aspects of the application within containers. Blue-green deployment was achieved through making a working blue instance handle production traffic while leaving idle green instance to stage new releases to be promoted later. The blue-green traffic management between instances was achieved through using an Ingress Controller configuration.

The application was built using a three-tier backend microservices architecture consisting of Catalog Service, to manage book information and inventory; Order Service to deal with payment processing and filling orders; and User Service to control user authentication with user-profile management. All three of the microservices were contained in Docker containers and run as separate Kubernetes pods and MongoDB implemented as a StatefulSet using persistent volumes to store reliable data [7].

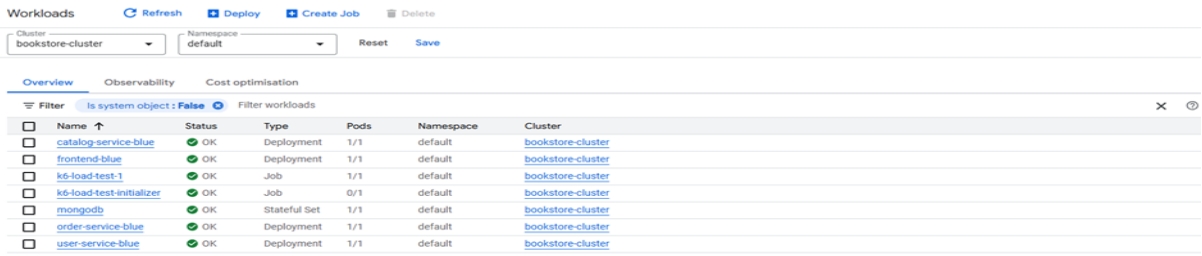


Fig. 5. GKE Cluster and Services

Implementing CI/CD Pipeline

We built a Jenkins CI/CD pipeline to make automated full deployment possible. GitHub used as the code management tool, and Jenkins was used to fetch code changes, build Docker images, and then deploy those into the Google Artifact Registry. The pipeline was configured to detect if the inactive environment is blue or green, deploy a new version, make health tests, and make a traffic switch subject to a successful deploy [2]. The automation has caused a tremendous reduction in time taken to deploy and also minimized human error while releasing.

Monitoring and Evaluation

To make it possible to continuously check system health and performance, a full-fledged monitoring infrastructure was implemented that included Prometheus to gather metrics, Grafana to visualize using dashboards, and Kubernetes Dashboard to increase cluster visibility. K6 was also used to perform load tests, thus enabling it to test system scalability under difficult conditions [3]. This software suite offered considerable observability that enabled instant detection and troubleshooting of any issues that would arise [15].

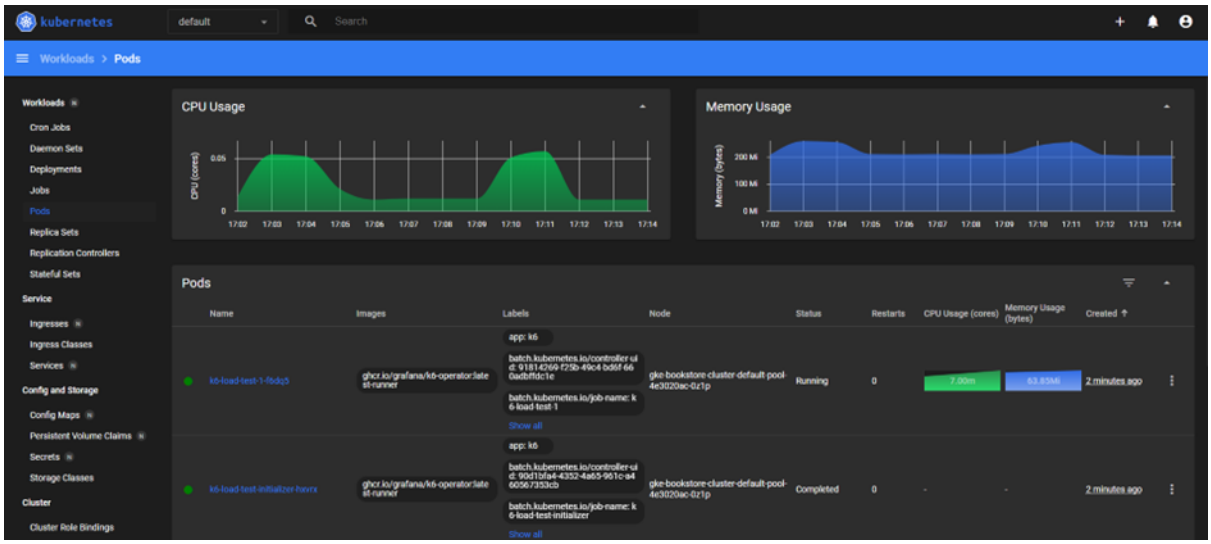


Fig. 6. Kubernetes Dashboard

Fig. 7. Grafana Dashboard

5.2 Transition to GCP: Resource-related Issues

Our team decided to use AWS Free Tier EC2 instances as the deployment environment during early development as we were short on budget. However, while designing Kubernetes we started running into serious limitations with the resources available. As of now there was only t2 on AWS Free Tier. micro instances that had 1 vCPU and 1GB RAM. Although this setup worked well for simple development, it became immediately obvious that this was not appropriate for running a production-ready Kubernetes cluster. The allocated memory size was too small and resulted in many out-of-memory (OOMKilled) errors, which caused the containers to crash and affected the availability of the application. When we deployed MongoDB we faced significant performance issues (MongoDB extremely sensitive to CPU and memory allocation problems in terms of data integrity and high invite transaction) [7]. Moreover, our Kubernetes nodes appeared unable to properly manage workloads due to bumping against their CPU limits (throttling), and performance would tend to degrade rather dramatically under relatively moderate levels of traffic.

Apart from the potential resource constraints, AWS itself added additional networking and storage models. With limited networking capabilities, Kubernetes pods would frequently fail, and inter-service communication became unreliable. AIOps Data collector- the challenge of persistent volume provisioning on AWS Elastic Block Store (EBS) was the reason behind I/O operations per second (IOPS); the read/write operation was slow which reduced database performance as well. Also, since AWS Elastic Kubernetes Service (EKS) did not come under AWS Free Tier, we had to manage deployment of Kubernetes manually, which resulted in extra operational overhead and complexity. This meant that Kubernetes cluster management on AWS needed extra configurations and manual tuning that was in many cases tedious and inefficient.

Keeping this limitation in mind, we explored other cloud platforms to find a more scalable and budget-friendly solution. Based on resource allocation and pricing flexibility, Google Cloud Platform (GCP) proved to be the best prospect. GCP also offered its e2-medium instances, which came with 2 vCPUs and 4GB RAM, unlike AWS on the Free Tier. With this transition, we solved our memory allocation problems, getting rid of all OOMKilled errors and stabilizing our application. Moreover, advanced processing capabilities led to improved performance of the Kubernetes pods without any CPU throttling, which in turn promoted a smooth functioning of the microservices. Furthermore, GCP's managed Kubernetes solution, known as Google Kubernetes Engine (GKE), made it easier to manage the cluster by automating some key tasks such as node scaling, networking configuration, and security updates [14]. This move greatly diminished the operational burden we used to have with AWS EKS.

A major reason to switch GCP wasa more generous free-tier credit cap. While reading AWS documentations I found that they have a limited free-tier resources to let you test few things, but GCP offers $300 in credits for new users where we can deploy a complete Kubernetes environment without adding anything extra. This credit enabled us to try out various configurations, fine-tune resource distribution, and perform extensive performance testing prior to the rollout of our infrastructure. Additionally, GCP offered sustained usage discounts in its pricing model, which would be advantageous for long-term cost optimization.

The migration had to be done in a way that would cause minimal disruption. We created a new GKE cluster and also had to reconfigure our networking and security settings to match GCP's infrastructure. Kubernetes deployment specs were modified using GCP-specific enhancements like integrated load balancers, networking policies, etc. This meant changing our CI/CD pipeline that we initially designed for AWS to adopt the GCP way. This involved configuring Jenkins to work with Google Artifact Registry to store Docker images and changing deployment scripts for the GKE cluster.

Retrospectively, the migration from AWS to GCP was a milestone that led to much greater stability, performance, and scalability for our online bookstore app. The increased resource availability removed performance bottlenecks, while the managed Kubernetes services lowered the chore of managing infrastructure. Moreover, the cost optimizations ensured long-term sustainability, solidifying GCP as a more feasible option for hosting scalable and resilient cloud applications. The experience taught us to carefully plan our resources while developing Kubernetes-based applications to avoid performance and reliability bottlenecks.

5.3 Implementation of MongoDB StatefulSet and Data Persistence

For the initial implementation, MongoDB was executed as a normal Kubernetes Deployment that supported dynamic pod scheduling to nodes [16]. Subsequently, this setup proved to be inadequate due to built-in Kubernetes Deployment management shortcomings with regard to pod identity and permanent storage. Because Kubernetes Deployment does not guarantee that a restarted pod will have the same identity or storage, each time MongoDB restarted or got rescheduled to another node, it lost access to already stored data [7]. This resulted in inconsistencies, and between consecutive instances of the application, consistency could not be achieved [7].

The other serious flaw in using the Deployment strategy was that MongoDB requires stable network identifiers to be able to create stable connections between replicas in a distributed environment. In our case, with every pod restart receiving a new hostname that would make previous database connections invalid. This was particularly irritating at MongoDB replication startup time because secondary replicas

were no longer able to communicate with the primary instance efficiently and thus presented risks of data inconsistencies. Also with a normal Deployment, Kubernetes can restart and exit pods at will, that is, with undesired outages and lost data.

We had to overcome such challenges by transitioning to a StatefulSet architecture that's stateful application modelled after databases. In contrast to a normal Deployment, StatefulSet gives each pod a stable and permanent identity so that MongoDB instances maintain their network identity even after restarting. We wanted to have persistent connections to databases and make it work with good replication without losing sync between primary and secondary nodes to have it important to us [14].

```
PS C:\Users\Shobika Rajeskanna> kubectl get pvc
NAME                    STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
mongo-storage-mongodb-0   Bound    pvc-2c5dfb75-b361-42b0-a876-22fa3dbc2d2b   5Gi        RWO            standard-rwo   <unset>                 5d9h
PS C:\Users\Shobika Rajeskanna>
```

Fig. 8. PVC Details

One of the greatest benefits of using a StatefulSet is being able to maintain persistent storage with Persistent Volume Claims (PVCs). In our initial configuration, Kubernetes automatically provisioned ephemeral storage to every MongoDB pod so that data would be lost if a pod were to restart or be rescheduled. When we converted to using a StatefulSet, we configured Persistent Volume Claims so that every MongoDB instance would have storage even if it were to restart or be rescheduled to a new node [14]. We accomplished this by creating a Persistent Volume (PV) and a StorageClass so that we had control over storage being provisioned and delivered to each instance of MongoDB.

Along with that, however, came some issues. The transition to a StatefulSet wasn't without its issues. The very first of those issues with us was with PVC binding errors in instances of mismatching storage class configuration. We had not been binding our PVC claims to back-end storage volumes as they were supposed to at first, and that was preventing MongoDB pods from initializing [7]. We resolved that by including explicit storageClassName references in Persistent Volume definitions and ensuring default storage class in our Kubernetes cluster was configured to allow for our MongoDB requirements.

Another point of concern was volume mount points. MongoDB will default to checking /data/db to find where it's supposed to store its data directory, but our initial StatefulSet configuration wasn't setting up the persistent volume to that directory. As such, MongoDB wasn't able to find its persisted data and startup would fail continuously. We were able to resolve this by setting up the volume mount point in the StatefulSet configuration to allow MongoDB to find the proper directory.

We also encountered issues with access mode settings. We used Persistent Volume with the ReadWriteMany access mode generally utilized for shared storage among multiple nodes. MongoDB does not permit concurrent writes by several instances and therefore caused database corruption with this setting. We resolved it by changing to ReadWriteOnce access mode such that every instance of MongoDB had single access to their storage volume.

```
PS C:\Users\Shobika Rajeskanna> kubectl get statefulset
NAME      READY   AGE
mongodb   1/1     11h
PS C:\Users\Shobika Rajeskanna>
```

Fig. 9. Statefulset

Finally, we configured our MongoDB StatefulSet to have rolling upgrades and failovers. Rolling upgrades allow us to deploy new MongoDB versions with zero downtime so that upgrades are sequentially applied to every instance without affecting the overall database cluster. We included readiness and liveness probes to check MongoDB instances to see if they are healthy so that Kubernetes will automatically restart pods that become unresponsive.

The transition to a StatefulSet architecture over a Deployment-based architecture considerably improved the reliability and performance of our MongoDB setup. Persistent storage, stable network names, and improved fault tolerance with persistent storage allowed our database to now handle high-workload operations without the risk of lost information or broken connections. The transition also served to highlight using appropriate Kubernetes configuration settings to run with stateful applications since inappropriate settings will yield disastrous operations issues.

Overall, the application of MongoDB through a StatefulSet was a key point in long-term application stability and application data store scalability. The exercise served to highlight learning Kubernetes storage operations, proper installation of StatefulSets, and verifying that persistent volumes were being allocated properly to prevent loss of information [7]. In long-term terms, this configuration provides us with a good platform to scale up the application even more because we can now scale up our MongoDB cluster with ease without concerning ourselves with storage inconsistencies or network issues.

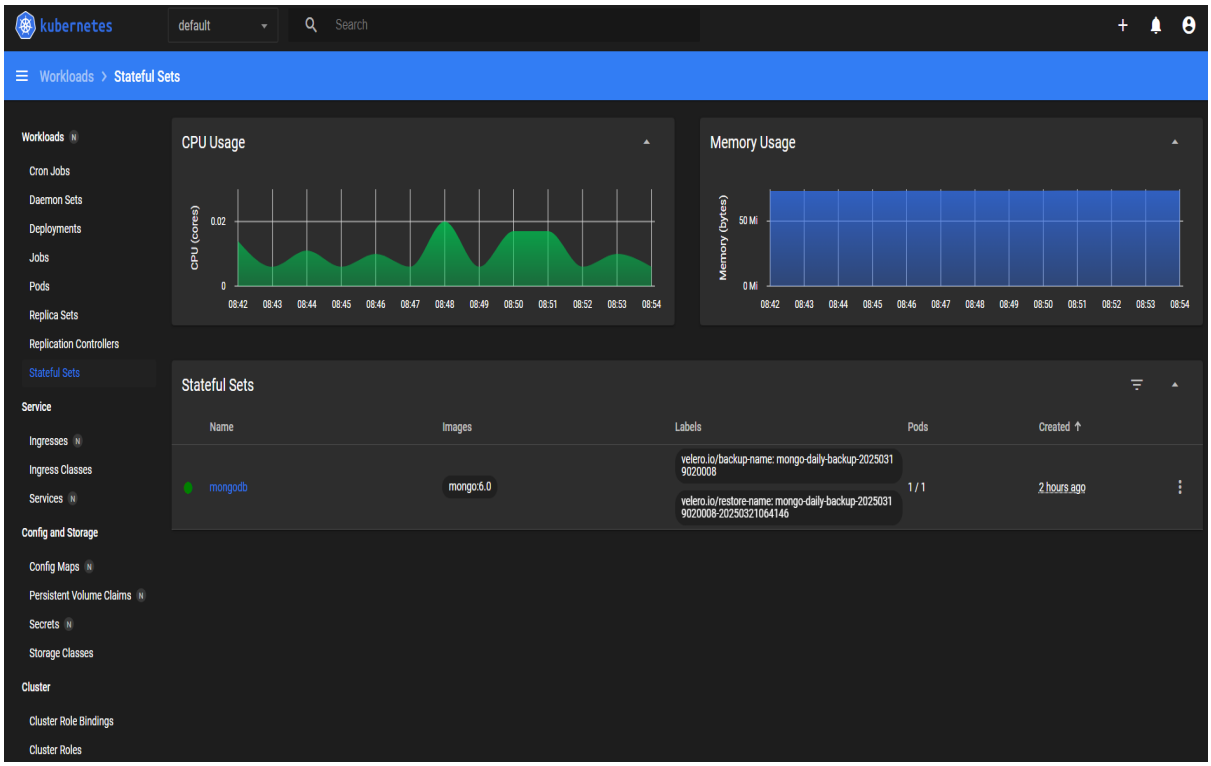


Fig. 10. MongoDB-StatefulSet

5.4 Kubernetes Networking, Ingress, and Traffic Management Problems

Networking and traffic management were also among the hardest issues to overcome while developing and deploying our bookstore application to cloud. Kubernetes does have a mature networking stack with built-in capabilities to allow service discovery, intra-pod-to-pod networking, and networking to outside services using Ingress controllers [17]. Despite that, complexity in Ingress rules, naming convention inconsistency of services, namespace hierarchy, and Ingress rules resulted in connectivity issues in the initial stage, broken API calls, and traffic management issues. Such issues had a tremendous impact on system stability, especially while deploying changes with blue-green deployment strategy.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: bookstore-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  ingressClassName: nginx
  rules:
  - host: csc-8113-bookstore.duckdns.org
    http:
      paths:
      - path: /catalog(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: catalog-service
            port:
              number: 80
      - path: /order(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: order-service
            port:
              number: 80
      - path: /user(/|$)(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: user-service
            port:
              number: 80
```

Fig. 11. Ingress-config

The primary issues that we were facing were Kubernetes cluster services discovery failures. As we deploy services such as Catalog Service, Order Service, and User Service as microservices, each of those services is assigned a unique ClusterIP automatically by default through Kubernetes. Such IP addresses being of type that is to say that each time a pod restarts they can be renewed was not acceptable. All our microservices were communicating with each other through static IPs instead of Kubernetes native services discovery mechanism at initial stage. This was causing frequent connectivity issues with every reschedule of a pod. We enhanced it through normalising our services with fully qualified domain names (FQDNs) and utilizing Kubernetes built-in headless services with stateful services like MongoDB. All services communicating with each other through DNS instead of static IPs allowed us to remove risk of broken connections while pods restarted.

Cross-namespacing communications failures were another top priority. Kubernetes namespaces provide you with default namespace isolation among services you can leverage to aid in organization and security. During the initial release, though, services were being deployed to multiple namespaces without namespace-to-namespace communication. This resulted in API call failures among microservices because requests in a single namespace were not being received by services in another namespace. We fixed that by referring services to their fully qualified domain names (e.g., catalog-service.default.svc.cluster.local) and introduced Network Policies to allow traffic among some namespaces. This stabilized service interactions and allowed microservices to communicate cleanly [8].

The Ingress configuration was also causing issues with traffic routing and blue-green load balances. We were routing traffic between application versions through straightforward path-based routing through the NGINX Ingress Controller. But through poorly configured path mappings and host-name-based

virtual host rules, traffic inconsistencies were being introduced such that users were being redirected to the wrong application version. It was being compounded with header-based traffic inconsistencies such that HTTP headers were not being properly picked up and traffic was being sent to dead services.

We updated our Ingress rules to conform to annotation-based definitions that were designed to be utilized with the NGINX Ingress Controller. We used direct mappings to Kubernetes services rather than relying on indirect DNS resolution to make routing more accurate. We enabled sticky sessions to maintain session consistency of users in blue to green swappage operations. It assisted us in keeping users on the right instance of the application even while routings were being dynamically changed.

Automation of traffic switchovers was at the core of our networking design. Blue-green traffic switchovers in our initial design were buggy and required human intervention with each deploy. These introduced unnecessary downtime and risk of misconfiguration. We implemented canary releases and weighted routing methods to auto-switch such traffic switchovers in Ingress. We utilized Kubernetes natively offered traffic switch feature to route traffic incrementally to a small fraction to the new release (green), then cut over to it solely off of the old release (blue). Risk-ran through by enabling us to see system behavior without cut over.

Our network and traffic management in Kubernetes were made extremely stable by efficient microservices communication, quality traffic management, auto-fail over of deploy and other security through such optimizations [8]. They were also key to facilitating zero-down time upgrades with a guarantee that user requests would always be routed to their correct application instance. The exercise served to affirm the importance of using standard service discovery, quality Ingress configuration, and auto-traffic management in Kubernetes applications.

5.5 Challenges and Solution in CI/CD Pipeline

We required a fully automated Continuous Integration and Continuous Deployment (CI/CD) pipeline to facilitate rapid development with reliable software releases and zero-down time deployments for our cloud-based bookstore application. The pipeline had to integrate code automatically, build it automatically, automatically test it, and automatically deploy it with minimal human intervention. In building it, several issues arose that required to be taken care of very meticulously. They were issues with Jenkins configuration, integration with the Google Cloud Platform (GCP), blue-green automated deployment, and rollbacks.

One of the initial issues was setting Jenkins to deploy using Kubernetes. Jenkins had to be linked to our Kubernetes cluster in a way that it would do automated build, testing, and deploying. In initial setup, though, we were facing Java-Jenkins version compatibility issues wherein jobs in the pipeline were continuously failing without explanation. Jenkins had to run stably with Java 11 while in default setup it was using Java 8 and thus crashing and failing plugins continuously. We fixed this by actually defining in our environment settings the Java that we required and upgrading all Jenkins plugins to Java 11-supported ones.

Fig. 12. Failed Deployment with Auto-Rollback in Jenkins

Another key challenge was authentication and permissions management with GCP services. The application was running on Google Kubernetes Engine (GKE), and Jenkins required permissions to deploy changes, control resources, and store artifacts in Google Artifact Registry. The Jenkins pipeline was failing due to a lack of permissions due to Jenkins's default service account not possessing the

required IAM roles. We fixed this by creating a single-purpose service account in GCP with required IAM roles to deploy Kubernetes and store artifacts [18]. We then encrypted and stored the service account key securely in Jenkins credentials so that the pipeline would authenticate with GCP without leaving sensitive credentials in plain text.

We integrated into our blue-green deployment plan that we needed to ensure that the pipeline would be able to dynamically route traffic between both green and blue environments and be able to smoothly switch. In initial implementation, Jenkins had no way of knowing what was up and therefore had a risk of misconfigurations with traffic being sent to a stale build. We overcame this by introducing a script to the Jenkins pipeline that would check what was up currently before it would make changes. It would check if blue or green was handling traffic and would deploy the other idle environment before it would switch over production traffic. This automated it out of human intervention, making it more consistent and reliable to deploy.

Having auto-rollback on failed deploy was also important. In early deploy pipeline revisions, a deploy had to be rolled back manually to a stable release, causing delay and potential downtime. We introduced an auto-rollback capability in Jenkins to circumvent that. We introduced this capability with Jenkins continuously monitoring health of newly deployed software through liveness and readiness probes. When a deploy did not pass such health probes, Jenkins would roll back to a previously working version automatically. This capability improved deploy reliability substantially and avoided users being affected by buggy upgrades.

```
PS C:\Users\Raj> kubectl get pods
NAME                                    READY   STATUS        RESTARTS        AGE
catalog-service-blue-5c5d79c8f5-dm4fr   1/1     Terminating   0               59s
catalog-service-green-7d7cd84b69-f92tr  1/1     Running       2 (118s ago)    4m10s
catalog-service-green-7d7cd84b69-x7ct6  1/1     Running       0               2m11s
frontend-green-5d45fc4db7-mvfh6         1/1     Running       0               4m9s
mongodb-0                               1/1     Running       0               4d6h
order-service-blue-76cc9db8b-8g6hq      0/1     Terminating   0               58s
order-service-green-7c4d945c8-v2qmp     1/1     Running       3 (59s ago)     4m10s
order-service-green-7c4d945c8-vbxmc     1/1     Running       0               2m19s
user-service-blue-79f45cb478-cdktz      1/1     Terminating   0               58s
user-service-green-65b59ff8cb-2hs8s     1/1     Running       0               2m12s
user-service-green-65b59ff8cb-8sfcn     1/1     Running       3 (58s ago)     4m9s
PS C:\Users\Raj>
```

Fig. 13. Failed Deployment with Auto-Rollback via CLI

We were encountering a permissions issue with Google Artifact Registry access. We had configured our pipeline to push Docker containers to Google Artifact Registry but were encountering issues with Jenkins due to incorrect authentication settings. The issue was a result of incorrect Identity and Access Management policies that were refusing to let Jenkins publish. We fixed it by granting Artifact Registry Writer to our Jenkins service account with full rights to push and pull container images without issues.

Secure management of secret in the pipeline was also a top concern. API keys and database connection strings were plaintext environment variables with sensitive data and presented a security risk. We resolved this by incorporating Google Secret Manager into our Jenkins pipeline and securely storing and loading all sensitive data on a need-to-use basis. This improved security and did not permit key pieces of infrastructure to be accessed by unauthorized people.

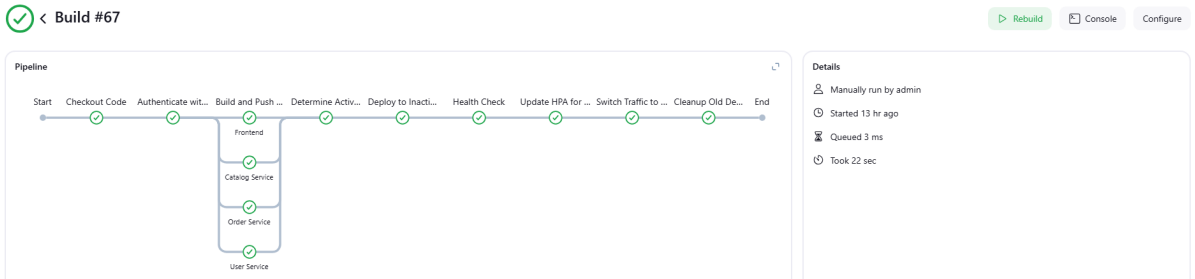


Fig. 14. Successful Deployment in Jenkins

Despite all this, we were able to create a robust automated, fault-tolerant, and secure CI/CD pipeline with swift and predictable zero-downtime deployments. The practice concluded that IAM role assignment

must be correct, traffic automation must be regulated, it must have rollbacks in place, optimizations through cache must be performed, and best security practices must be adopted in building a perfect CI/CD pipeline to deploy cloud-native applications.

```
PS C:\Users\Raj> kubectl get pods
NAME                                    READY   STATUS    RESTARTS   AGE
catalog-service-green-7d7cd84b69-f92tr   1/1     Running   0          57s
frontend-green-5d45fc4db7-mvfh6          1/1     Running   0          56s
mongodb-0                                1/1     Running   0          4d6h
order-service-green-7c4d945c8-v2qmp      1/1     Running   0          57s
user-service-green-65b59ff8cb-8sfcn      1/1     Running   0          56s
PS C:\Users\Raj>
```

Fig. 15. Successful Deployment via CLI

5.6 Conduct Load Testing and Monitoring

Our cloud bookshop application required good load testing practice to be maintained. Because the application had to handle lots of users searching books, purchasing books, and handling user profiles at a time, we had to simulate real traffic to make sure that the system would work at varying workloads. We had to have monitoring infrastructure to detect performance bottlenecks so that we can prevent failures and reduce waste of resources.

We were easily able to establish through initial testing that our system lacked proper facilities to load it up and thus it was difficult to identify the number of concurrent users it would be able to handle under heavy load before it would start to slow down in performance. In attempting to rectify this situation, we used K6, a more recent load-testing tool, to simulate tens of thousands of concurrent users issuing API calls against a subset of microservices. The initial K6 test ramped up slowly over 50 seconds to 500 concurrent users starting at 0 and then maintained load level at that point for ten minutes before slowly ramping back down. The objective was to monitor what effect small and large volumes of traffic would have on the system.

```
export let options = {
    stages: [
        { duration: '10s', target: 100 },  // 10s - ramp up to 100 users
        { duration: '10s', target: 200 },  // 10s - ramp up to 200 users
        { duration: '10s', target: 300 },  // 10s - ramp up to 300 users
        { duration: '10s', target: 400 },  // 10s - ramp up to 400 users
        { duration: '10s', target: 500 },  // 10s - ramp up to 500 users
        { duration: '10m', target: 500 },  // 10m - hold at 500 users
        { duration: '50s', target: 0 },    // 50s - ramp down to 0 users
    ],
```

Fig. 16. K6-stages

We applied Horizontal Pod Autoscaler (HPA) at work with load testing. We wanted to see if Kubernetes was autoscaling up and down pods in sync with traffic increases. We found that pods were not getting scaled up at first, even though both CPU and RAM were well over their thresholds. In some HPA configuration troubleshooting that we were doing, we found that we had a percent target that was way off and that system autoscale wasn't occurring timely. By lowering that percent to a target value, Kubernetes would autoscale very nicely up to eight pods at traffic highs and back down to one pods at traffic lows, evening out load and avoiding degradation of service.

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: catalog-service-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: catalog-service-blue
  minReplicas: 1
  maxReplicas: 8
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 60
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 70
```

Fig. 17. HPA config

We deployed K6-Operator by Helm Chart, which simplifies the K6 configuration. We can easily use a mainfest file to deploy the K6 job to test the catalog service and send the test metrics to Prometheus. However, we encountered a problem with Prometheus not receiving the data. This is because Prometheus does not accept remote writes by default. Since we used Helm Chart to deploy Prometheus-Operator, there is no local manifest file. Hence, we changed this configuration by CLI. Eventually, Prometheus was able to receive the metrics transmitted by the K6 properly, and we used the official dashboard provided by K6 to visualize this data [19].

```yaml
apiVersion: k6.io/v1alpha1
kind: TestRun
metadata:
  name: k6-load-test
spec:
  parallelism: 1
  script:
    configMap:
      name: k6-script
      file: "script.js"
  arguments: --out experimental-prometheus-rw
  runner:
    env:
      - name: K6_PROMETHEUS_RW_SERVER_URL
        value: "http://prometheus-operated.monitoring.svc:9090/api/v1/write"
```

Fig. 18. K6 manifest

Overall, with K6 load testing, Prometheus and Grafana monitoring, and Kubernetes HPA autoscaling together, performance and availability of our book store application increased tremendously [19]. Because all such tools were being observed with their observability, we were able to scale pods more efficiently, have more efficient utilization of resources and minimize cloud expenses. Because of continuous monitoring, we were able to provide high availability of application even in case of high traffic. All such improvements were a good building block to scale up in long term and system was well able to take care of increasing user traffic without sacrificing performance.

5.7 Data Recovery Implementation: Migrating from MongoDB to PVC to Velero

Data resilience and recovery were among the top priorities of our cloud-based bookshop application. As MongoDB was being used as the central database to store user information, book catalogs, and transaction histories, it was essential that a good mechanism of recovery of data was in place that would be robust against crashes like pod crashes, node crashes, accidental overwrite, or cloud infrastructure crashes. Our application started with MongoDB deployments with non-persistent storage and hence lost data every time pods were restarted or rescheduled by Kubernetes [7]. In order to prevent that, we transitioned to using Persistent Volume Claims (PVCs) to store long-term data. But with us starting to scale up our application, we felt that we required a good backup and disaster recovery mechanism to be in place and hence implemented Velero—open-source backup and restore tool for Kubernetes.

Our initial experiment with Kubernetes had used ephemeral storage and therefore every time a pod crashed or was rescheduled to another node, everything would be lost. We realized it while doing some initial load testing and experiencing frequent inconsistencies in data because new pods had new volumes of storage without being able to see previous records. It led us to realize that we required persistent storage and therefore we included Persistent Volume Claims (PVCs) to allow MongoDB to see its records after pod restarts [7].

However, we were encountering issues with our initial roll-out of PVC. The issues were mainly Persistent Volume (PV) binding errors in that Kubernetes was unable to bind MongoDB's PVC to a corresponding PV. The real cause was a StorageClass configuration issue leading to volume provision failures. We solved this by manually configuring storageClassName in our PVC definitions and ensuring that storage class matched available storage class in our Google Kubernetes Engine (GKE) cluster. We were also encountering issues with wrong volume mount locations as MongoDB required mounting folder /data/db. Initially, wrong mount locations resulted in MongoDB being unable to access stored

information and hence causing repeated pod failures. By correcting volume mount configuration, the database was able to store information persistently even when pods were terminating.

```
PS C:\Users\Raj> velero backup describe mongo-daily-backup-20250320020011
Name:         mongo-daily-backup-20250320020011
Namespace:    velero
Labels:       velero.io/schedule-name=mongo-daily-backup
              velero.io/storage-location=default
Annotations:  velero.io/resource-timeout=10m0s
              velero.io/source-cluster-k8s-gitversion=v1.31.6-gke.1020000
              velero.io/source-cluster-k8s-major-version=1
              velero.io/source-cluster-k8s-minor-version=31

Phase:  Completed


Namespaces:
  Included:  default
  Excluded:  <none>

Resources:
  Included:        *
  Excluded:        <none>
  Cluster-scoped:  auto

Label selector:  <none>

Or label selector:  <none>

Storage Location:  default

Velero-Native Snapshot PVs:  auto
Snapshot Move Data:          false
Data Mover:                  velero

TTL:  720h0m0s

CSISnapshotTimeout:    10m0s
ItemOperationTimeout:  4h0m0s

Hooks:  <none>

Backup Format Version:  1.1.0

Started:    2025-03-20 07:30:11 +0530 IST
Completed:  2025-03-20 07:30:24 +0530 IST

Expiration:  2025-04-19 07:30:11 +0530 IST

Total items to be backed up:  82
Items backed up:              82

Backup Volumes:
  Velero-Native Snapshots:
    pvc-2c5dfb75-b361-42b0-a876-22fa3dbc2d2b: specify --details for more information

  CSI Snapshots: <none included>

  Pod Volume Backups: <none included>

HooksAttempted:  0
HooksFailed:     0
```

Fig. 19. Velero Backup Description

While PVCs had answered long-term data storage issues, they had not answered backup and restore operations if we were to have to restore everything in the database. Having to rely solely on PVCs meant that if a node or storage infrastructure crashed, we were not able to restore MongoDB to some point in time. We rectified this by including Velero, a Kubernetes-aware backup and restore tool that allowed us to schedule MongoDB data and cluster configuration backups.

The initial setup was to deploy it to backup MongoDB's PVC data and snapshot it to GCS. The initial setting failed at backups due to IAM role constraints. Because Velero required roles to create backup, read backup, and restore backup on GCS, we had to deploy a new service account with relevant IAM policies. The automated storing of the backup data and loading of it by the Velero plugin without human intervention was realized through deploying the roles to it.

Having resolved authentication issues, we introduced automated backup jobs utilizing Kubernetes CronJobs such that snapshots of MongoDB content would be taken at regular intervals. The backup procedure was to create full backup every day to create full database snapshots. Incremental hourly backups to save only what's different since last backup. Retention policies that automatically deleted expired backups to reduce storage costs.

After getting the backup system online, we verified that it could restore using disaster recovery capabilities by manually setting up real-world scenarios. We began by manually deleting MongoDB pods and PVCs to check if Velero could restore them. During early testing, restoration was failing due to incorrect restore settings because Velero wasn't reattaching volumes that were being restored to new pods. We updated restore policies to reconstruct MongoDB pods with identical persistent volume claims to address that issue. Having updated those settings, we were able to restore the database within 15 minutes to verify that backup and recovery were working as designed.

The other crucial test was to simulate a complete node failure of the GKE cluster. We manually turned off the node on which MongoDB was being run. We observed what Kubernetes and Velero would do. The cluster scheduler tried to restart the pod on another node at first but due to PV attachment constraints, it was not possible to mount it with its original storage. As a response to this, we assigned ReadWriteOnce (RWO) mode to the PV so that it can be attached again after MongoDB was rescheduled to another node [14]. After applying this patch, the database automatically restarted without assistance.

The integration with Velero also had some additional benefits that were unrelated to database backup. We used Velero to backup Kubernetes cluster settings so that if disaster struck, we could restore both MongoDB and the entire application ecosystem such as deployments, services, Ingress rules, and ConfigMaps. This allowed us to restore the entire cluster if required with minimal downtime and loss of information.

Among some of the key optimizations that we implemented was off-peak backup job execution to minimize performance degradation. We started off with realizing that backup jobs were heavy on resources and caused MongoDB response time to degrade under heavy traffic [7]. As a solution to that effect, we executed full backups at midnight during off-peak hours when users were minimal and incremental backups hourly to record changes to the database without putting much load. Finally, we implemented monitoring and alerting of backup failures. We built dashboards to monitor backup success rate, used storage space, and restore time using Prometheus and Grafana. We also integrated Alertmanager to send us a notification on Slack if a backup job failed so that we could take instant action.

In short, transitioning to a full Velero backup and disaster recovery solution greatly increased the resiliency and reliability of our database infrastructure. Persistent storage of MongoDB data, automated backups, and lightning-fast restoration of it eliminated the risk of lost data due to infrastructure downtime, accidental overwrite, or surprise disasters. After extensive testing and tuning, we built a fast and scalable plan of recovery that made our cloud-based bookstore application more resistant to failures with high availability and performance.

6 Test plan

For testing strength, scalability, and reliability of our cloud bookstore application, a test plan was carefully designed. The primary objective was to test system behavior with multiple user loads, failback to normal on breakdown and graceful update with zero downtime. The Catalog Service being central backend microservice to maintain book inventory and update of it was prioritized exceptionally high in testing. Various tests were performed such as functional verification testing, scalability testing using K6, resilience testing using Velero to simulate disaster recovery and Blue-Green deployment strategy to simulate deployment. Every test was performed to simulate real-world scenarios to identify bottlenecks and check if system can recover without affecting users' experiences. They were immensely helpful in providing input regarding system behavior with diverse loads and stresses.

6.1 Testing Methodology

The testing strategy was designed on four pillars: functional testing, scalability testing, resilience testing, and deployment testing. Functional testing was used to check if Catalog Service processed API requests properly and maintained consistent communication with MongoDB. Scalability testing was used to check if dynamically the resources could be scaled up or down with increasing traffic load using Kubernetes Horizontal Pod Autoscaler (HPA). Resilience testing was used to check with high efficiency the application recovered in case of failures, i.e., database crashes or node termination, using Velero disaster recovery and Kubernetes self-healing [14]. Deployment testing confirmed if software upgrades were deployed efficiently using Blue-Green deployment with rollback feature without affecting services.

6.1.1 Functional testing

Functional testing was crucial to check that the Catalog Service was working as required and was perfect working with MongoDB. The objective was to verify that all API endpoints were working as required and book information was being fetched and presented correctly to the frontend. The testing was done by making HTTP requests to the /api/books endpoint through testing with Catalog Service API using Postman to check proper handling of response. The integration with frontend React application and backend was also verified to check that user requests were bringing correct information being fetched and presented. Some of key test cases that were performed were to check if a GET to /api/books returned a list of books stored in MongoDB correctly, if a POST to create book entry was working, if a PUT to update book details was working and if a DELETE to remove a book entry deleted it from database. The result was that all API endpoints were working as required because every query returned proper HTTP status codes to reflect that system was handling information transfer as required.

6.2 K6 for Scalability Testing

To check if the system would be able to handle heavy traffic due to users, K6 was used to load test it with concurrent queries at once to simulate multiple users query the bookstore at once [20]. The test scenario was configured by introducing Prometheus Operator to gather metrics and using Grafana to visualize it to monitor real-world system behavior at varying loads.

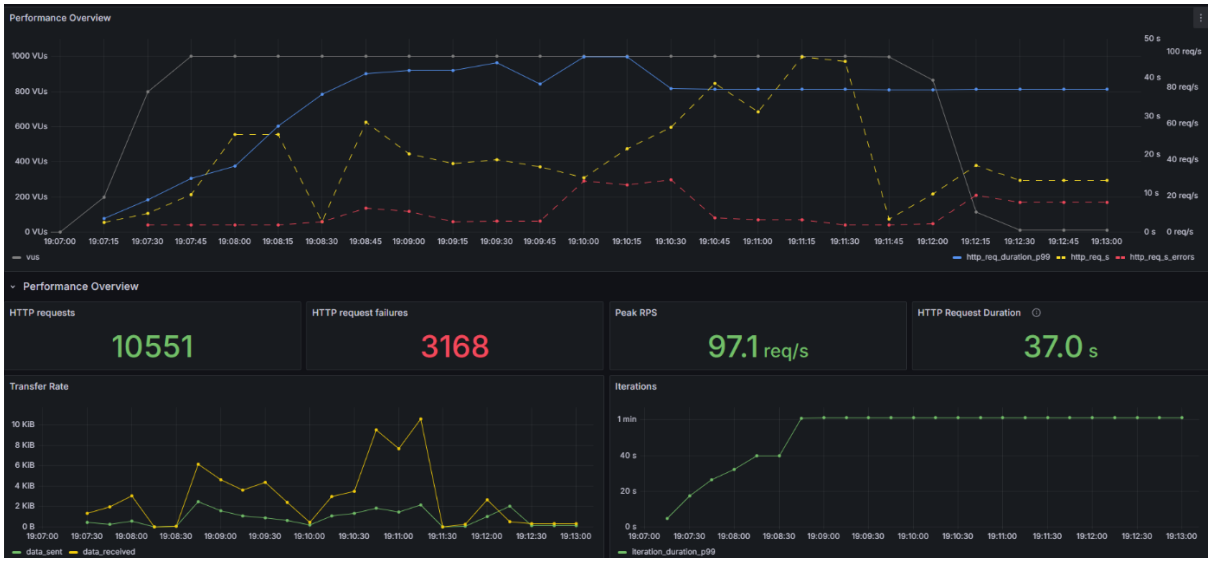


Fig. 20. K6 Grafana Dashboard

The test script was configured to have 500 concurrent users make /api/books query with ramping load to see up to what point the system would be able to handle it. The result, using Grafana to monitor it, was a clear indication of system behavior. The Horizontal Pod Autoscaler (HPA) scaled pods to eight out of one effortlessly, witnessing to the fact that the Kubernetes cluster could dynamically handle resources with real-world demand. All such results testified that the system was extremely scalable with good capacity to stay responsive under heavy traffic while utilizing cloud resources efficiently.

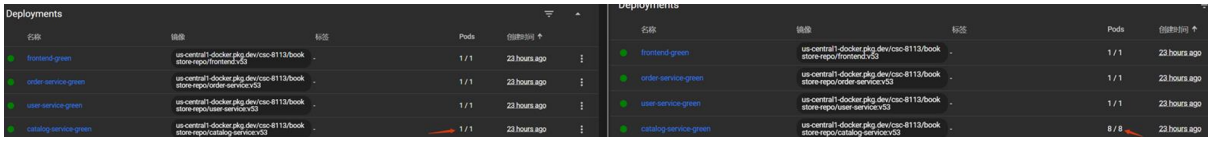


Fig. 21. Horizontal Pod Autoscaling during K6 load test

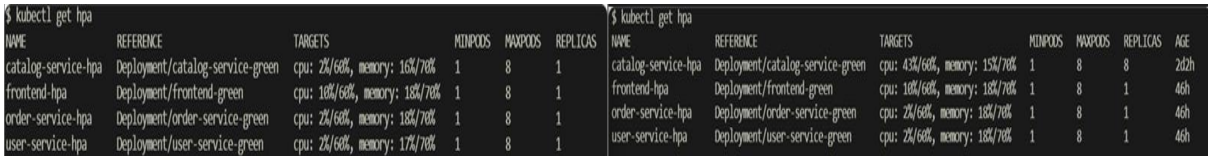


Fig. 22. HPA Status

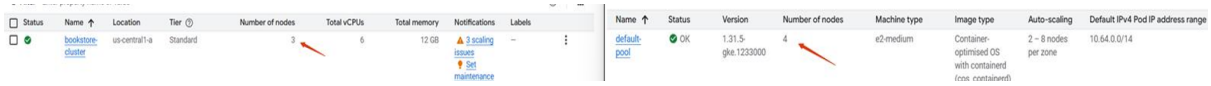


Fig. 23. Cluster Autoscaling

6.3 Resilience Testing

We carried out resilience testing to make sure that failures would not impact availability and that recovery would take place. This included testing Kubernetes self-healing through liveness and readiness probes and Velero through disaster recovery. We integrated liveness probes to make it check periodically if a service's endpoint was healthy so that if a service wasn't available, Kubernetes would automatically restart the failing pod to resume operations. We used readiness probes to drain services to accept traffic until they were initialized properly to make sure that users were unable to access partially working or incomplete services. Velero played a vital role in disaster recovery and allowed us to automatically backup MongoDB data and Kubernetes resources to allow us to have backup features in case of catastrophic failures.

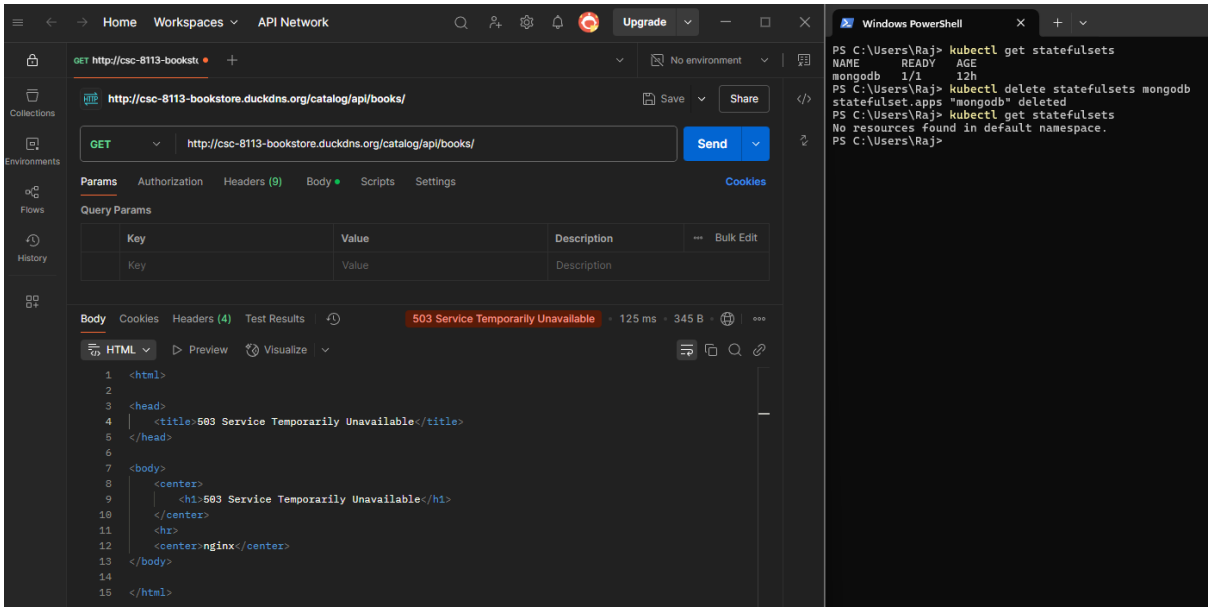


Fig. 24. MongoDB Statefulset Deletion and Testing

The testing procedure involved manually deleting MongoDB pods to simulate sudden crashes and see Kubernetes's response. Failed pods were automatically restarted by the system without human intervention on crashing with Kubernetes self-healing capabilities being witnessed. Further testing to verify disaster recovery involved the utilization of Velero to create a full backup of MongoDB and Kubernetes settings before manually deleting critical database assets. The system was recovered using Velero backups

with full restoration of all information without impacting system performance. The exercise was done within a window of 15 minutes without losing information and therefore showing that backup was efficient and effective. The tests were to verify that the system would survive sudden crashes and automatically recover without much human intervention.

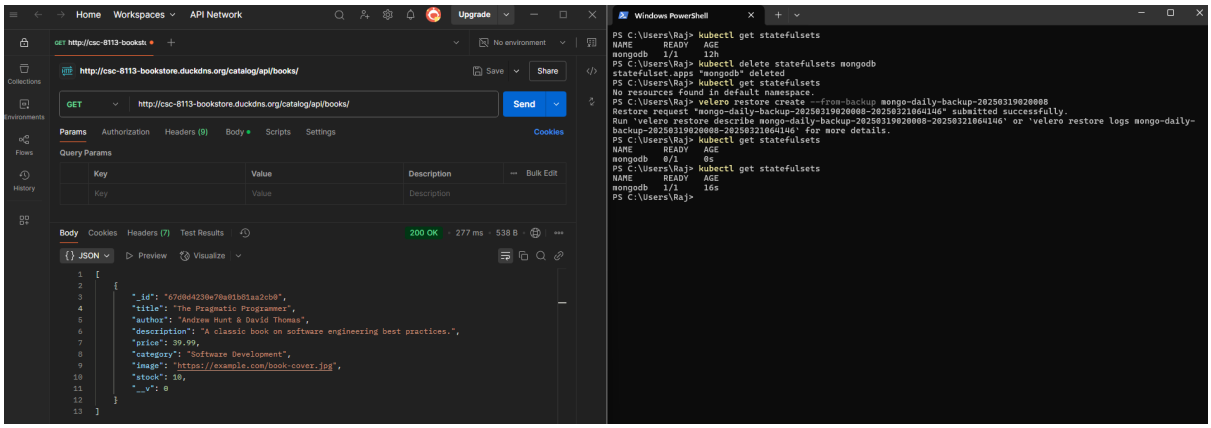


Fig. 25. Successful Velero Restoration for the MongoDB Database

6.4 Deployment Testing

Blue-Green testing was performed to check if software upgrades could be deployed smoothly without affecting traffic. Blue-Green pattern was chosen because it possesses the ability to host two similar production environments with one being used to host current traffic and another on standby to host new upgrades. The rolling out of new upgrades involved rolling out a new release of Catalog Service in idle (Green) mode and it was thoroughly checked and verified before sending out actual traffic. The switch was triggered after health check tests in Green mode confirmed it to be stable. Ingress configuration was updated to divert traffic coming to current Blue mode to newly verified Green mode and switch was completed without downtime.

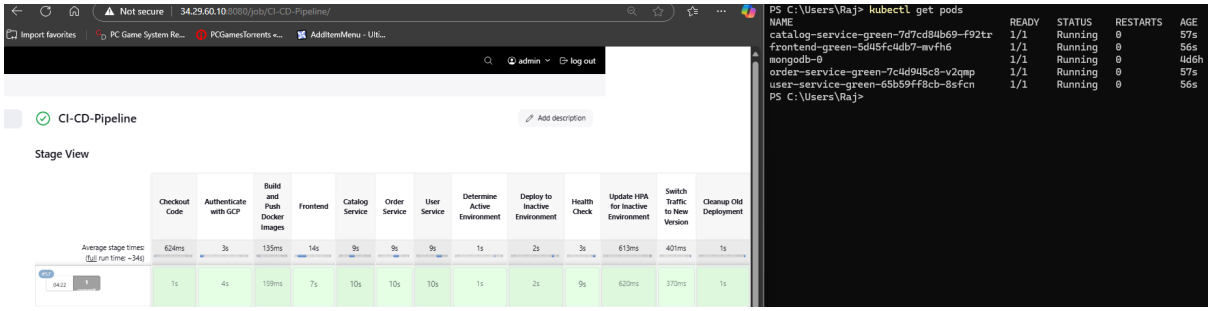


Fig. 26. Successfull CI/CD deployment with Jenkins

The second critical portion of testing with regards to deployment was to verify that rollback within the CI/CD pipeline was functional. If an update did not succeed, the system would roll back to a past stable build automatically to prevent downtime. The testing strategy stimulated failure by deploying a deliberately faulty build of the Catalog Service that caused catastrophic errors [21]. The Jenkins CI/CD pipeline properly diagnosed the failure and automatically backed out and reverted to the good previous build. The mechanism of rolling back prevented traffic ever being sent to the broken-down build, thus preventing downtime and user disruptions.

The result demonstrated that Blue-Green deployment pattern easily facilitated zero-down time releases wherein new software versions were tried out before being deployed to clients. The feature of rollback was also helpful as system stability was maintained even with unsuccessful deployments.

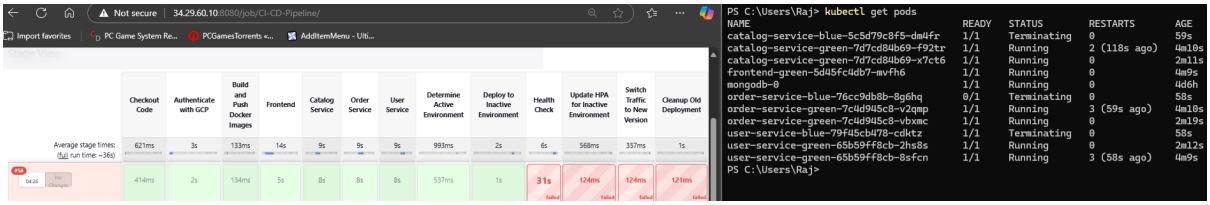


Fig. 27. Failed deployment with rollback

6.5 Obervations and Improvements

We had a few key learnings through testing that led to optimal optimizations to improve system performance. The Catalog Service withstood heavy traffic with low response time with users such that we had a good design with microservices. Kubernetes self-healing capabilities performed well to auto restart crashed pods without contributing to downtime while making it more stable. Integration with Velero to backup and do disaster recovery such that MongoDB data was well safeguarded with full system restore without losing data. Blue-Green deployment pattern led to continuous software upgrades with minimal user disruption.

For optimal continuous system performance tuning, thresholds around liveness probe failures were adjusted to prevent unnecessary restarts due to minor delays that occur in heavy load scenarios so that minor delays would not induce unnecessary restarts. Velero backups were executed at off-peak traffic hours to affect minimal performance while maintaining continuous backup coverage. The thresholds of HPA were also adjusted based on real traffic flows so that Kubernetes would scale resources more efficiently.

6.6 Conclusion of testing

The test plan effectively verified functional correctness, scalability, resiliency, and automated deploy of bookstore application's Catalog Service in a cloud Kubernetes-based environment. K6 load testing confirmed that the system could maintain increasing user load with low response time, showcasing cloud-native efficiency. Velero had a good backup and disaster recovery mechanism in place that offered protectable and recoverable data against failures. Jenkins auto-CI/CD automated the deployments with minimal human intervention while ensuring consistency. Kubernetes orchestration offered self-healing and auto-scaling to maintain system availability and efficient utilization of resources. These findings establish that our DevOps strategy was effective to create cloud applications that were scalable, fault-resilient and resilient and that the system is well positioned to maintain real-world workloads with low operations overhead.

7 Critical Evaluation

7.1 Self Evaluation

Our choices were key to the overall efficiency, scalability, and day-to-day operation of the system. The options enabled us to create a system that is natively flexible and easily functional. A key decision was the choosing Jenkins to deploy Continuous Integration and Continuous Deployment (CI/CD) pipelines, together with utilizing blue-green deployment strategy [2]. The decision greatly improved our release and development workflows through automation of building, testing, and deploying, thus giving us a very seamless integration without downtime in upgrades. The using blue-green deployment strategy eliminated system downtime in upgrades, allowing us to have zero downtime while at the same time offering us a simple rollback mechanism in case of software bugs. In conclusion, this practice improved the system's reliability while at the same time reducing the occurrence of errors.

An important decision that was made was to leverage JavaScript both for client-side programming (using React.js) and server-side programming (using Node.js/Express) [14]. This decision eased the development process by enabling a single codebase to be used, thus minimizing context-switching between two programming languages. It also eased integration between client-side and server-side pieces of code, thus both speeding up development and making it more maintainable to have a single codebase. It

also improved developer efficiency since developers had to deal with a single programming language at both layers of the application, thus making development more efficient and more integrated architectural design.

Helm charts were utilized in this research to manage Kubernetes resources. This hugely enhanced management efficiency in relation to configuration management in those that included the Kubernetes Dashboard as well as the Prometheus Operator [22]. The templating feature presented by Helm enabled upgrades and rollbacks to be administered easily through it, hence making it easier to manage and make changes to Kubernetes deployment. The adoption of Helm presented a reproducible and predictable way of managing resources, hence preventing complexity in managing upgrades of configuration.

We have chosen to use the K6 operator instead of the default K6. The integration of this operator with Kubernetes enabled us to have improved testing capabilities to conduct load testing with optimized performance within the same environment that hosts our application, thus increasing efficiency in management and automation. The integration allowed us to continue to run performance testing to check if our application would be able to handle different levels of traffic.

In database management context, MongoDB was chosen over PostgreSQL because it offered a flexible model of schema together with horizontal scalability that enabled it to build a microservices architecture. The NoSQL features of MongoDB made it effective to handle large amounts of unstructured data efficiently, while horizontal scaling capabilities were particularly helpful to our application. In addition, MongoDB integration with Node.js through Mongoose enabled efficient practices in handling and managing data [23]. Finally, using Ingress to manage traffic routing as opposed to using external IPs improved the networking capabilities of the application. The introduction of Ingress centralized traffic management, thus making it more secure, scalable, and maintainable. The move secured traffic routing through a single domain and avoided issues related to multiple external IPs, thus making it easy to maintain as the application grew.

The design decisions adopted enabled us to create a very scalable and robust system. Jenkins was used together with Helm, K6 operator, and MongoDB with Ingress being used to handle routing. We were able to create a system with high flexibility and maintainability. The system can handle variable load volumes while maintaining a consistent development platform and user platform.

7.2 Relative Evaluation

We analyzed our design choices against industry best practices and standards to allow us to compare methodologies used by other teams. Several key differences were found through this comparative study that highlighted our design's advantages in terms of scalability, management of resources, and overall system efficiency.

The two approaches were distinguished by their traffic routing strategies. Team A used external IP addresses to allow direct traffic routing to their services. As much as this is a relatively simple way to do it, it has several disadvantages with regard to security, scalability, and maintainability [24]. The exposure of services using external IP addresses increases vulnerability risks and increases complexity with regard to system management as it scales. Our team used Ingress to direct traffic through a single domain, thus making traffic management through central management of the routing protocol. In addition to this, Ingress offered several ancillary benefits such as SSL termination, load balancing, and rate limiting that were all beneficial to system security, scalability, and efficiency in resources [25]. This had a more secure traffic routing, lower complexity, and maintainability of the system with regard to services expansion.

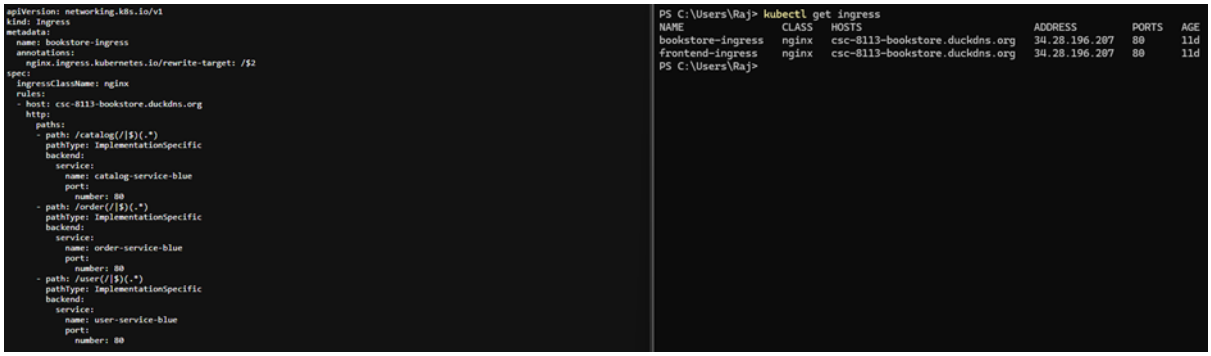


Fig. 28. Ingress Configuration and Deployed Service

We did notice a significant disparity in handling limits on resources. Team B had no pre-set thresholds on resource utilization in their Kubernetes pods and hence had poorly utilized resources. More specifically, lack of pre-set thresholds heightened their pods' probability of using more than their capacity and thus causing OOMKilled errors and creating system instability especially under heavy traffic. Such incidents could lead to eviction of pods or system crashes and thus negatively affect system performance. In contrast to that, our strategy included detailed calculation and setting limits to every single pod of ours. We calculated approximate demands of every microservice and imposed clear constraints on RAM, CPU, and storage space to make every pod consume what it needed without more. This way, we curbed every prospect of competition for resources that could jeopardize application stability with varying loads. This forward thinking allowed us to monitor very carefully against overuse of resources and make sure that capacity of the system could vary with traffic without affecting performance.

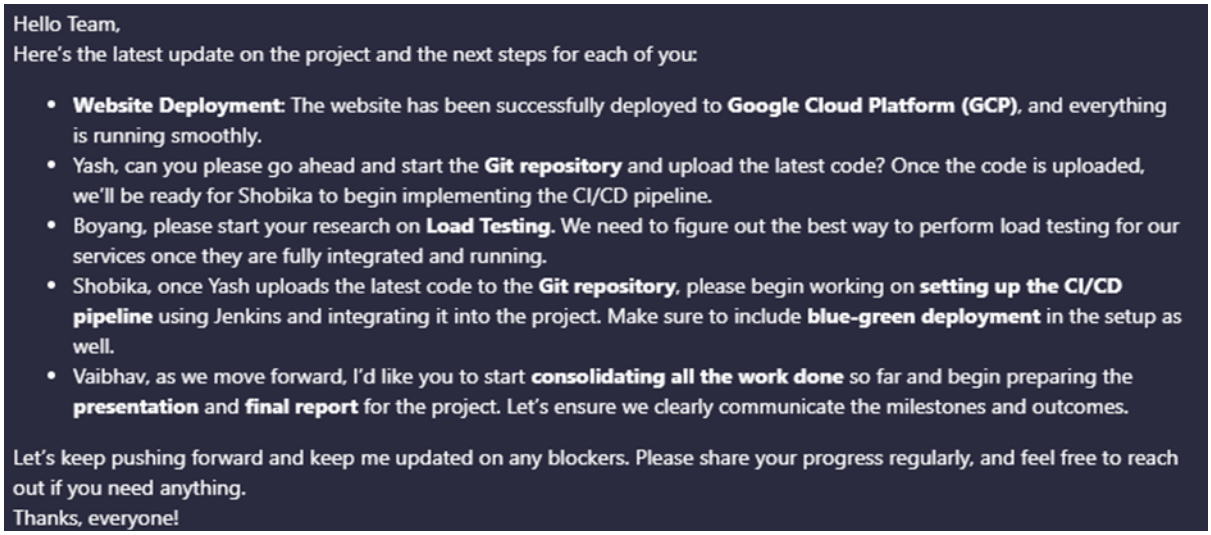
Fig. 29. Catalog Service with Pod limits

The application of Ingress for traffic control and definition of the boundaries of each pod's resources is strategic decision making that is consistent with best practice within Kubernetes environments. The use of centralized traffic control with restricted distribution of resources is vital in enabling scalability while ensuring efficient working conditions. In addition to being used to improve application performance, such methodologies have also positively impacted other factors like security, scalability, and maintainability [24].

The traffic routing and resource planning determinations were critical to overall project success. Ingress traffic routing enabled creation of a system with increased security, scalability, and responsiveness. In addition, prioritized application of constraints to resources allowed effective and reliable management of resources and avoided failures while providing optimal system performance. All together, such design characteristics had a considerable impact on project results while meeting best practices in Kubernetes management and microservices deployment, with the final outcome being a stable application that can provide high performance and scalability to serve a growing user base.

8 Group Working

Our project's triumphant culmination was enabled by organized and effective team working, built on each member's critical contribution to both development and deployment. By drawing on diverse skillsets and professional specialisms, we were able to create a stable and scalable solution. All participants contributed in their areas of specialty that included frontend and backend development, setting up a CI/CD pipeline, orchestration using Kubernetes, database connectivity, and monitoring of systems. All participants' contribution and their percentage contribution to this project is explained in the table below.


Hello Team,
Here's the latest update on the project and the next steps for each of you:

- **Website Deployment**: The website has been successfully deployed to **Google Cloud Platform (GCP)**, and everything is running smoothly.
- Yash, can you please go ahead and start the **Git repository** and upload the latest code? Once the code is uploaded, we'll be ready for Shobika to begin implementing the CI/CD pipeline.
- Boyang, please start your research on **Load Testing**. We need to figure out the best way to perform load testing for our services once they are fully integrated and running.
- Shobika, once Yash uploads the latest code to the **Git repository**, please begin working on **setting up the CI/CD pipeline** using Jenkins and integrating it into the project. Make sure to include **blue-green deployment** in the setup as well.
- Vaibhav, as we move forward, I'd like you to start **consolidating all the work done** so far and begin preparing the **presentation** and **final report** for the project. Let's ensure we clearly communicate the milestones and outcomes.

Let's keep pushing forward and keep me updated on any blockers. Please share your progress regularly, and feel free to reach out if you need anything.
Thanks, everyone!

Fig. 30. Group Updates

Team Member	Contributions	Contribution
Amlan Baruah	Amlan built the website from scratch with all four microservices using a MongoDB database. Besides this, Amlan was involved in both development and deployment aspects of the project in Google Cloud Platform (GCP), using Google Kubernetes Engine (GKE) and Artifact Registry. Furthermore, Amlan adopted a blue-green deployment with a rollback feature as part of the Continuous Integration/Continuous Deployment (CI/CD) pipeline.	20%
Yash Gadodia	Yash played a key role in developing the Kubernetes Dashboard to make it more visible to the GKE cluster. In addition to that, he led in migrating the code to the Git repository and managing access role distribution among team members while integrating the blue-green deployment practice into the CI/CD pipeline.	20%
Shobika Rajeskanna	Shobika led the integration of Jenkins as the CI/CD tool, enabling its connection to the Git repository of the project and creating the first pipeline. In addition, she integrated Velero in the GCP project, creating a connection to the MongoDB pod through PVC, while also running daily snapshots of the database to ensure data persistence.	20%
Boyang Wang	Boyang was tasked with bringing auto-scaling capability into the cluster project. He implemented Horizontal Pod Autoscaling (HPA) to all pods, implemented Prometheus and Grafana to allow monitoring of system performance, and integrated the K6 operator to allow load testing of services under conditions that mimic high traffic volumes.	20%
Vaibhav Patil	Vaibhav played a key role in structuring the project report and content of the final presentation. In addition, he played a key role in making sure that the team was working according to pre-set deadlines by continuously monitoring their work and verifying that the project objectives were completed within the originally allotted time frame.	20%

Table 3. Team Contributions

9 Conclusion

Implementation of microservices architecture, cloud-native technologies, and DevOps principles to build a system renowned for scalability, resilience to faults, and high availability was best demonstrated through

the development of a cloud-based bookstore application. Using Google Kubernetes Engine (GKE) and setting up continuous integration and continuous delivery (CI/CD) pipelines using Jenkins with a blue-green deployment technique enabled automated, efficient, and streamlined delivery of software while ensuring system reliability in the update process [26].

The microservices architectural pattern allowed applications to be built in a modular manner and enabled scaling of individual components like User Management, Catalog, and Order Process services separately within their own Kubernetes Pod. The MongoDB database was deployed using a StatefulSet with Persistent Volumes to maintain consistency and persistency of data; additionally, disaster recovery capabilities were included through using the Velero backup tool. Realtime monitoring facilities were also enabled through using Prometheus and Grafana to allow proactive detection of failures and making it easier to make optimizations to the system.

During the implementation stage, several issues were faced such as challenges related to cloud resources, complications related to database persistence, and complexities related to Kubernetes network operations. The shift to Google Cloud Platform (GCP) from AWS Free Tier really picked up system performance using more efficient resources. Configuring Ingress settings, autoscaling policies, and automating Continuous Integration/Continuous Deployment (CI/CD) operations resulted in improved performance, traffic management with optimal efficiency, and consistent workflow deployments.

The load testing with K6 assured that the system had the capability to dynamically scale to maintain high traffic volumes without compromising on response times. The Cluster Autoscaler and Horizontal Pod Autoscaler both efficiently managed resources to reduce unnecessary expenses while delivering improved services [20]. Additionally, testing with Velero for resilience demonstrated that with selfhealing capability of Kubernetes, the system had good failover and recovery capabilities.

This study highlights the importance of principles of DevOps in modern application development using cloud computing with a focus on important aspects like automation, scalability, and resiliency. The findings and approaches used in this project create a framework that can act as a model to create cloud-based web business systems that efficiently integrate best practices related to microservices, Kubernetes orchestration, and continuous delivery techniques.

Potential enhancements to the system can include performance boosters, increased security measures, and integration with artificial intelligence-powered recommendation engines to enhance user experience even more. Serverless or event-driven architecture can make it more efficient to distribute resources and lead to considerable cost reduction.The project has built a track record of delivering a scalable, effective, and reliable cloud-born application designed for bookstores and thus has proven that modern-day DevOps practices can be used effectively in real-world software release scenarios [27].

References

1. "Kubernetes Documentation," *Kubernetes*. https://kubernetes.io/docs
2. M. Fowler, "bliki: BlueGreenDeployment," *martinfowler.com*. https://martinfowler.com/bliki/BlueGreenDeployment.html
3. "Grafana Documentation," *Grafana Labs*. https://grafana.com/docs
4. "React," *react.dev*. https://react.dev
5. "Using RBAC Authorization," *Kubernetes*. https://kubernetes.io/docs/reference/access-authn-authz/rbac/
6. Jenkins, "Jenkins User Documentation," *Jenkins User Documentation*. https://www.jenkins.io/doc/
7. MongoDB, "What is MongoDB? — MongoDB Manual," *MongDB*. https://www.mongodb.com/docs/manual/
8. S. Newman, *BUILDING MICROSERVICES : designing fine-grained systems*. O'Reilly Media, 2018.
9. P. Mell and T. Grance, "The NIST Definition of Cloud Computing Recommendations of the National Institute of Standards and Technology," NIST, 2011. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf
10. T. Xu *et al.*, "CyberStar: Simple, Elastic and Cost-Effective Network Functions Management in Cloud Network at Scale ," 2024. Accessed: Mar. 21, 2025. [Online]. Available: https://www.usenix.org/system/files/atc24-xu-tingting.pdf
11. Github, "GitHub," *GitHub.com*, 2013. https://github.com
12. D. R. Augustyn, Ł. Wyciślik, and M. Sojka, "Tuning a Kubernetes Horizontal Pod Autoscaler for Meeting Performance and Load Demands in Cloud Deployments," *Applied Sciences*, vol. 14, no. 2, p. 646, Jan. 2024, doi: https://doi.org/10.3390/app14020646.
13. "Velero Docs - Overview," *Velero.io*, 2025. https://velero.io/docs/ (accessed Mar. 21, 2025).
14. NodeJS, "Node.js," *nodejs.org*. https://nodejs.org
15. "Cloud Monitoring," *Google Cloud*. https://cloud.google.com/monitoring
16. L. Bass, I. Weber, and L. Zhu, *DevOps a software architect's perspective*. New York Boston Indianapolis Addison-Wesley New York Boston Indianapolis Addison-Wesley Mai, 2015.
17. M. Armbrust *et al.*, "A View of Cloud Computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010, doi: https://doi.org/10.1145/1721654.1721672.
18. "Docker Documentation," *Docker Documentation*, Jul. 15, 2020. https://docs.docker.com
19. Prometheus, "Overview | Prometheus," *Prometheus.io*, 2016. https://prometheus.io/docs (accessed Mar. 21, 2025).
20. "k6 Documentation," *k6.io*. https://k6.io/docs/
21. J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Upper Saddle River, Nj: Addison-Wesley, 2011.
22. "Docs Home," *helm.sh*. https://helm.sh/docs/
23. Express, "Express - Node.js web application framework," *Expressjs.com*, 2017. https://expressjs.com
24. J. Smith, "DevSecOps Overview," *Snyk*, Mar. 19, 2020. https://snyk.io/learn/devsecops/ (accessed Mar. 21, 2025).
25. "Google - Site Reliability Engineering," *sre.google*. https://sre.google/sre-book/table-of-contents/
26. GOOGLE, "Google Kubernetes Engine (GKE)," *Google Cloud*. https://cloud.google.com/kubernetes-engine
27. "OWASP Top Ten Web Application Security Risks | OWASP," *owasp.org*. https://owasp.org/www-project-top-ten