Research Methods and Group Project in Cloud Computing

Tanapon Suwankesawong, Bansi K. Dobariya, Shubh Anand, Sanjana T. Shahu, Aryan Katyayan

School of Computing, Newcastle University
Newcastle upon Tyne, United Kingdom

Abstract

This article discusses the creation and deployment of a cloud-native, microservices-based bookstore application with a focus on the adoption of new DevOps methodologies. The application is developed using Azure Kubernetes Service (AKS) for container deployment and orchestration, which provides scalability, high availability, and efficient traffic handling. The top technologies utilised are React.js as front-end tech, Node.js as back-end micro-services, PostgreSQL as the main database, and Docker for containerisation. The article also mentions using GitHub Actions for automating CI/CD, Prometheus and Grafana for monitoring, and Load Balancer and Nginx Ingress Controller for effective traffic distribution. The paper also mentions the challenges that came in the way while implementing it and calculates the efficiency of using DevOps practices to achieve fault tolerance, scalability, and cloud-nativeness of the application. The report then discusses the pragmatic advantages of these approaches in streamlining the project's development, deployment, and operation.

Keyword

Microservices, DevOps, Kubernetes, CI/CD, Docker

1. Introduction

1.1 Introduction of Members

Our team is composed of individuals with various technical backgrounds who bring their strengths to create the bookstore micro-services application.

Aryan Katyayan

I have experience in machine learning and now how to work with python and machine and deep learning frameworks like scikit-learn and TensorFlow. Working on this project was quite tough for me because I never worked on cloud and DevOps projects but working on this project forced me to learn new tools and develop new skills. This project helped me build confidence for projects like this that I might work on in the future.

Bansi K. Dobariya

I specialize in creating web-based systems by designing both front and back systems which provide users with intuitive web experiences. Through my development experience I have obtained deep skills in JavaScript programming for building dynamic user interfaces with React and Node.js framework. My work at Par Solution involved developing sustainable web applications that allowed me to strengthen API management skills and learn database optimization in addition to rigorous code evaluation. The implementation of role-based authentication for microservices together with complicated database operations required me to overcome professional development challenges. My software development strategy unites accuracy with creative thinking and optimal execution abilities to build secure stable programs.

Tanapon Suwankesawong

I worked at a banking company for four years before joining this university. Initially, my primary responsibility was backend development. As I took on additional projects, I was introduced to DevOps, particularly when tasked with migrating and improving various Jenkins pipelines from legacy systems. This experience provided me with a strong foundation in automation and deployment. With this background, I had no difficulty developing the pipeline for this project. However, I had no prior experience working with the cloud, setting up configurations, or using GitHub Actions. These aspects were initially challenging for me. Through this project, I gained hands-on experience in cloud technologies and deepened my understanding of DevOps.

Shubh Anand

I'm Shubh Anand, a DevOps Engineer with two years of experience in cloud infrastructure, automation, and CI/CD pipelines. I have worked extensively with containerisation (Docker, Kubernetes) and cloud platforms like Azure, focusing on building scalable and reliable systems. For this DevOps group project, I contributed to designing and deploying an online bookstore using a micro-services architecture. My role involved setting up CI/CD pipelines, implementing zero-downtime deployment strategies, and integrating monitoring and logging to ensure smooth operations. I'm passionate about leveraging DevOps best practices to streamline deployments and enhance system reliability, and this project has been a great opportunity to apply my industry experience to a real-world scenario.

Sanjana Shahu

I possess beginner-level experience in cloud computing, DevOps implementation, and containerisation technologies with a little bit of hands-on experience in Azure Kubernetes Service (AKS) and Docker. I am AWS Well-Architected Proficient and AWS Cloud Essentials badge holder, and these have provided me with a sufficient background in cloud architecture and best practices. Implementation of Phase 3 (Resilience) provided me with detailed knowledge in terms of disaster recovery, Velero backup strategy, and monitoring using Kubernetes probes. It enhanced my problem-solving skills and provided me with knowledge regarding cloud-native development, which made me technically stronger and enabled the project to succeed.

1.2 Introduction on Project

This paper is about implementing a cloud-native, microservices-based bookstore application using modern-day DevOps practices to automatically scale, be reliable, and easy to deploy. The project aims to utilise Azure Kubernetes Service (AKS) to orchestrate containers in order to scale the system according to the needs of the users and provide high availability.

System architecture includes a Frontend UI developed in React, providing simple-to-use interfaces to customers to discover books and track their carts. Node.js-based Backend Micro-services enable Catalog and Cart functionality, enabling the feature to have hassle-free book list, metadata, stock, and session management by customers. Backend Micro-services consume information from a PostgreSQL database containing transactional data of books and carts.

In addition, the system is pre-installed with Nginx Ingress Controller and Load Balancer for efficient traffic routing and management of incoming requests. CI/CD pipeline, driven by GitHub Actions, automates deployment for continuous integration, testing, and deployment.

The dependability and the functionality of the application are enabled by containerisation with Docker, and its monitoring is handled by Prometheus and Grafana for acquiring real-time understanding of how the system functions. Design consideration, implementation issues at the time of development, and the ultimate assessment of how well the project functions to fulfil the role of being a scalable, fault-tolerant, and cloud-native application are covered in the report too.

To further optimise system performance and scalability, Kubernetes Horizontal Pod Autoscaler (HPA) automatically adjusts the number of running pods based on real-time CPU and memory consumption. This ensures that the system is responsive to varying loads of traffic without the need for human intervention. In addition, the AKS Cluster Autoscaler is set to create new infrastructure capacity when needed in order to avoid bottlenecks and deliver a seamless user experience.

The combination of orchestration, containerisation, automated deployment, and monitoring ensures high responsiveness and availability of the bookstore application. With modern DevOps, the system receives a robust, scalable, and fault-tolerant environment to facilitate seamless online transactions for end-users.

2. Project Specification

The cloud-based online bookstore system has been designed with scalability, high availability, and rapid deployment in mind using modern DevOps techniques. Microservices-based architecture, and this provides flexibility, scalability, and fault tolerance. The application is composed of a number of components including a React.js frontend as well as React.js backend micro-services for the Catalog and Cart services, and a PostgreSQL database to manage transactions.

With a micro-services architecture, the system is endowed with decoupled scaling, and each service can be independently scaled based on demand without affecting the whole application. This optimises resource utilisation, providing efficient performance and cost savings. AKS also provides automated scale, load balancing, and failover, reducing operational workload. The cloud-native approach enhances reliability with smooth failover, horizontal scaling, and self-healing features, which help achieve high availability. This architecture later supports system resilience, making it responsive to different traffic and workload.

1. Frontend: Frontend developed with React.js manages the user interface, enabling customers to browse books, manage their shopping cart, and accept payments. Scalability along with convenient maintenance is facilitated by React component-based architecture.

2. Backend Micro-services: Backend of the bookstore application is in Node.js that powers the Catalog and Cart Services core business. Catalog Service and Cart Service are built as distinct micro-services with their respective business logics for inventories and shopping carts.

    a. Catalog-Service:
    Catalog Service handles the books listings, whose metadata comprises authors, titles, prices, and inventories. This service communicates with the PostgreSQL database itself, where it writes and reads data pertaining to the books.

   b. Cart Service:
      Cart Service is responsible for the user shopping cart feature. It encompasses adding the books to cart, quantity update, and order processing. The Cart Service communicates with the database to store cart data and process transactions when clients decide to pay for their orders.

   Both services are run on Node.js and React for a responsive and seamless user interface, with ease of communication among the services and database. The React frontend takes care of the user interface, and Node.js backend serves the API endpoints for catalog and cart functionality. This decoupling allows modularity and scalability in the system.

3. Database: The PostgreSQL database is used as the central storage facility for the application to store book information, inventory, and user shopping cart activity. The database is run in a Kubernetes StatefulSet to enable persistence and data consistency.

   A multi-node database replication is in place for providing high availability and fault tolerance. This provisions for automatic failover upon failure of the nodes, minimising downtime and enhancing reliability of data. Additionally, Velero is used to implement automated backup policies, safeguarding data against accidental loss or corruption.

4. Container Orchestration: The application resides on Azure Kubernetes Service (AKS), which provides secure orchestration of containers, scalability, and resource management. Kubernetes allows the application to be scaled depending on demand and provides high availability.

   Kubernetes self-healing feature guarantees that in case a pod crashes or freezes, it is replaced automatically, ensuring continuity of service. Kubernetes-native logging and monitoring tools integrated into the system provide real-time visibility into system health and performance.

5. Traffic Management: A Load Balancer is configured to handle traffic efficiently on the backend services. The external API calls are processed by the Nginx Ingress Controller, TLS termination is done, and routing is properly forwarded to the respective service.

   For efficient handling of incoming traffic, traffic routing and load balancing in the Azure Kubernetes Service (AKS) cluster is done by Nginx Ingress. Rate limiting and throttling can be enabled to restrict heavy traffic and offer equality in terms of resources, but scalability and reliability become priorities. The system is designed to provide a seamless user experience with backend services being safeguarded and performance being upheld even with heavy traffic.

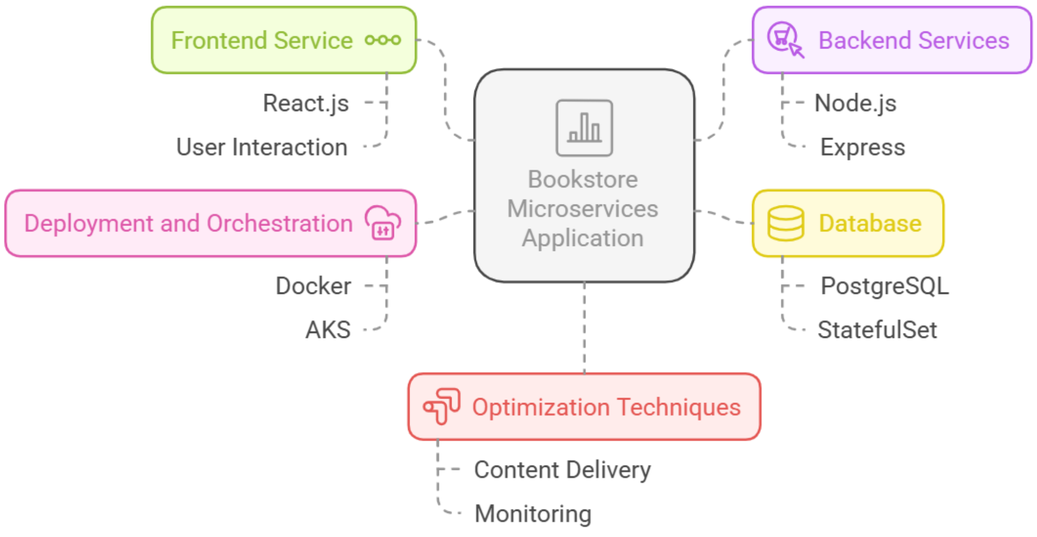


Fig 1: Bookstore Micro-services Architecture and Optimisation [6]

2.1 DevOps Objectives

1. **Scalability**:

   - The system also uses Horizontal Pod Autoscaling (HPA) for the Catalog Service so that automatically the service scales on utilisation (CPU and memory). This makes it easy for the system to handle heavy traffic without interference.
   - The AKS Cluster Autoscaler is also configured to dynamically change the infrastructure resources, which ensures the system under different loads as efficient as possible.
   - To enhance request distribution, the load balancer is configured with intelligent routing, ensuring even distribution of requests across pods and backend services.

2. **Reliability and Recovery:**

   - Liveness and Readiness probes are configured to identify the health of the services to ensure that the failed pods will be automatically retried, thus making the system more resilient.
   - The PostgreSQL database has been defined as a StatefulSet to ensure continuous and persistent storage and data even during the restart of the pod.
   - Velero deployment is employed in disaster recovery for facilitating automatic backup as well as quick restore upon failure. This minimises downtime and preserves critical data.
   - Rolling Updates deployment strategies reduce downtime by gradually rolling out new versions of services while monitoring system performance.

3. **Deployment and Automation**:
   - Blue-Green Deployment pattern is applied to the Catalog Service, which supports seamless rollbacks and updates without service disruption. It allows the application to be updated with zero user impact.
   - GitHub Actions automate CI/CD by running continuous integration, testing, and deployment tasks. An automated pipeline provides the ability to test each change prior to deployment and also includes a provision for rollbacks to an earlier version in the event of failure.
   - A K6 script is used to simulate traffic and evaluate pod autoscaling in Azure Kubernetes Service (AKS). By generating controlled load scenarios, the script helps assess the system's ability to scale dynamically based on demand, ensuring efficient resource utilisation. However, K6 is not integrated into the CI/CD pipeline, and testing is conducted manually as needed.

2.2 Extensions Implemented

The bookshop micro-services application saw some significant additions to make it even more scalable, reliable, and performant. With these additions, the system is made more sustainable, resilient, and able to deal with increasing demands.

1. Blue-Green Deployment: Blue-Green Deployment was utilised to limit downtime during upgrade. By rolling out and validating the new release, it supports two environments of the application running simultaneously and routs traffic towards the stable build. To support seamless user experience, it reduces service interruptions once new features or patches are ready.
2. Observability Stack (Prometheus and Grafana): Both of them provided a solid monitoring. Grafana graphs the metrics scraped by Prometheus from the system in real time. With this observability stack, the team could observe system performance, detect bottlenecks, and monitor application health. It greatly helps real-time trouble shooting and preventative maintenance to create a smooth user experience.
3. Velero-based backup and restore: Backup and restore were performed using Velero for disaster recovery and protection of data from loss. This saves the important application data at regular intervals and makes sure that, in the event of system failure, the restoration can be performed as soon as possible. Using Velero enhances the fault tolerance of the system and maintains business continuity in the event of any unforeseen problem.
4. GitHub Actions for Automating CI/CD Pipeline: GitHub Actions were included to automate the CI/CD pipeline to build, test, and deploy the application. This automation offers updates continuously to be implemented and transferred to production rapidly and reliably with fewer chances for human error. CI/CD pipeline also facilitates testing at each phase of deployment such that the system is ensured to be rock-solid and flawless in development and production environments.

3. Background Research

DevOps emphasises automation, scalability, reliability, and continuous delivery, all of which are at the heart of this project. Through the inclusion of Infrastructure as Code (IaC), Continuous Integration/Continuous Deployment (CI/CD), and observability practices, the system enables seamless deployments, high availability, and quick recovery. Through the use of Azure Kubernetes Service (AKS) for container orchestration, GitHub Actions for CI/CD, and Velero for backup, we are adhering to DevOps best practices in creating a cloud-native, fault-tolerant application.

Observability is one of the core DevOps concepts that involves tracking the health of applications, real-time detection of issues, and resolving system crashes in advance. Prometheus and Grafana are part of the system to obtain real-time performance monitoring and identify anomalies. Prometheus gathers and accumulates time-series data from application components, whereas Grafana displays the aggregated data in interactive dashboards. This integration gives administrators an option to see key performance measures (KPIs) like CPU, memory consumption, request latency, and response time, making them have a highly responsive and available system.

Horizontal Pod Autoscaler (HPA) in Kubernetes scales the Pods within a workload (e.g., a Deployment or a StatefulSet) automatically based on CPU, memory, or custom metrics. It scales horizontally up and down by adding or removing Pods for demand matching. [1] The HPA script periodically checks resources and dynamically scales the desired replicas based on target metrics to get optimal utilisation. This control loop runs at a fixed rate and looks up the resource metrics API or custom metrics API in an effort to inform the scaling decisions.[1]

HPA accommodates scaling based on per-pod measurement of resources or user-specified metrics and computes the scaling factor as the average of all Pods' values. It is not applicable on non-scalable resources like Daemon Sets and maintains a history of allocated minimum and maximum number of Pods.[1] HPA controller is able to update the workload's scale sub resource to change replica counts as a reaction to the present load.[1]

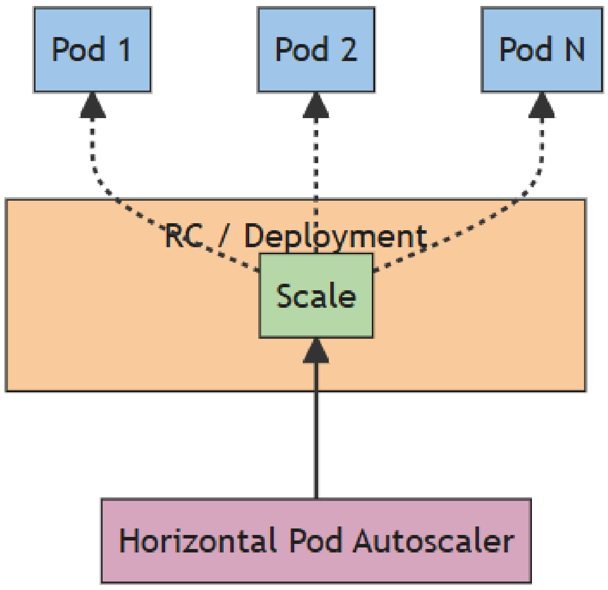


Fig 2: Horizontal Pod Autoscaler work [1]

Grafana k6 is an open-source, developer-focused load testing tool that assists engineering organisations with guaranteeing the performance and availability of applications and infrastructure. [2] k6 enables load and performance testing, browser performance testing, synthetic monitoring, and chaos testing. k6 is designed for low resource usage and can run high -load tests such as spike, stress and soak tests.

k6 can be integrated into CI/CD pipelines to drive performance tests in the development and release pipeline. [2] It also includes infrastructure testing and fault simulation capabilities for chaos experiments, so it is a purpose-built tool for performance validation and resilience testing. k6 also comes with extensions for testing various protocols and systems within the infrastructure.[2]

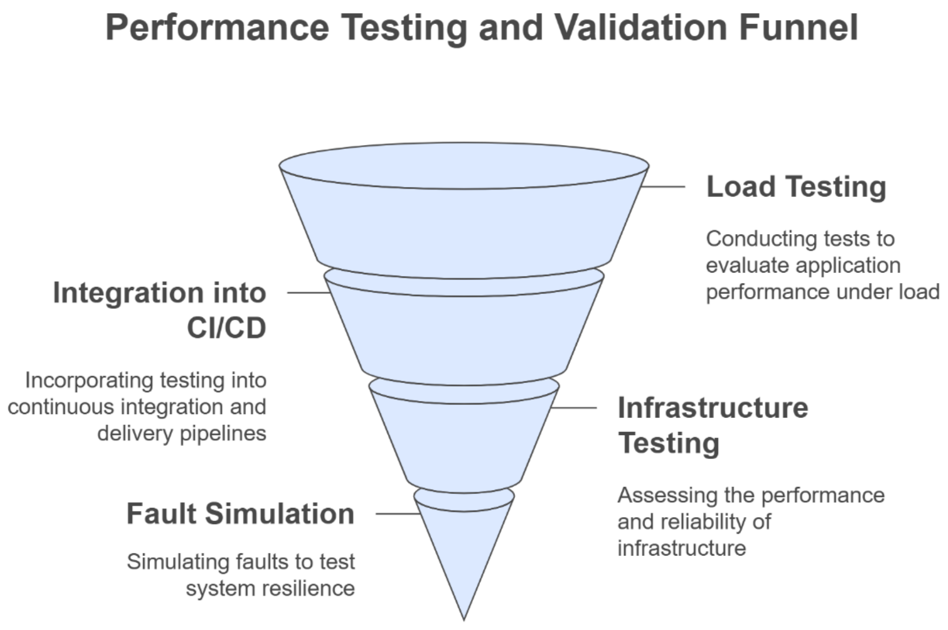

Performance Testing and Validation Funnel
Load Testing
Conducting tests to evaluate application performance under load
Integration into CI/CD
Incorporating testing into continuous integration and delivery pipelines
Infrastructure Testing
Assessing the performance and reliability of infrastructure
Fault Simulation
Simulating faults to test system resilience

Fig 3: Grafana performance testing

Velero is an open-source tool for backing up and restoring Kubernetes cluster resources and persistent volumes.[3] It gives users the capability to take backups, restore them in case of data loss, migrate resources to other clusters, and copy production environments to development or test clusters. Velero employs a server running on the cluster and a command-line client for local use.[3]

Backups can be scheduled with periodic times defined as Cron expressions. When a backup is initiated, Velero queries the Kubernetes API server, confirms the backup, and uploads the contents to object storage, in our case it was Azure blob storage.[3] To restore, Velero can restore objects into multiple namespaces, and it supports several restore configurations such as namespace remapping and optional restore hooks. Restore does not overwrite existing resources by default unless configured, providing non-destructive restores by default.[3]

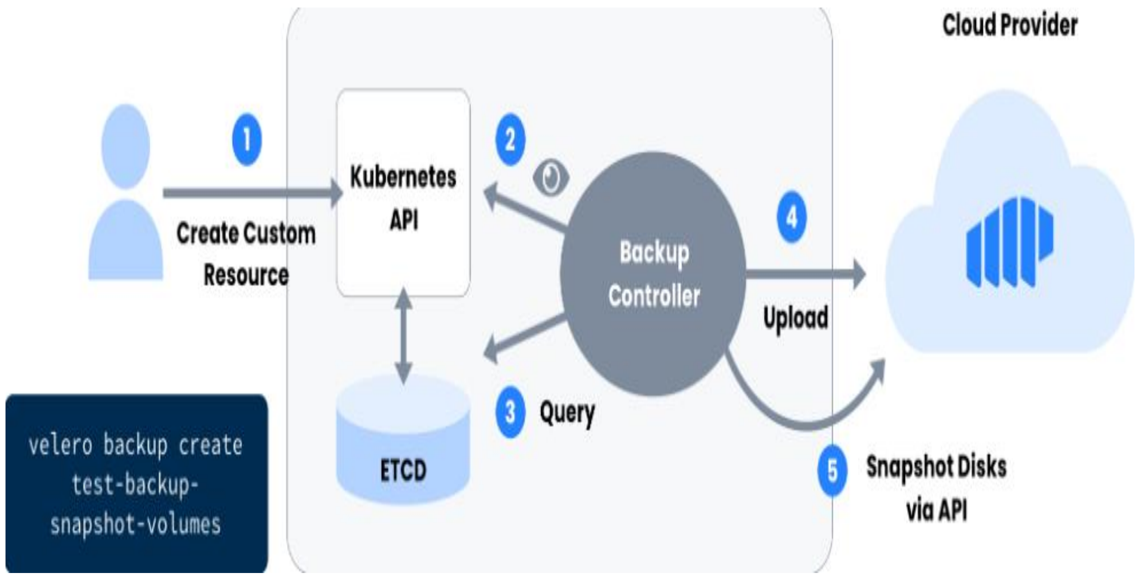


Fig 4: Kubernetes Backup via Velero [4]

3.1 DevOps Techniques Review

a. Scaling

To ensure the system dynamically scales based on traffic bursts and optimises resource use, we had Horizontal Pod Autoscaler (HPA) configured in Azure Kubernetes Service (AKS). It automatically scales application pods based on CPU and memory consumption to ensure there is smooth performance with high loads. Besides that, Cluster Autoscaler was configured, which introduces new worker nodes if the existing ones prove insufficient, so the system does not encounter any bottleneck and is extremely responsive. For verification of our scaling mechanism's efficiency, we had executed load testing with k6 and simulated thousands of simultaneous users for system testing under load.

These three HPA, Cluster Autoscaler, and load testing were also more in demand compared to the use of manual scaling because they provided instant flexibility, avoided incurring a large operational overhead, and checked on not running up excessive resources independently without operator input. In contrast to vertical scaling where resources assigned to an instance are merely added, our approach to horizontal scaling offers better fault tolerance, redundancy, and economics by spreading traffic into many small instances rather than the utilisation of a single large powerful node.

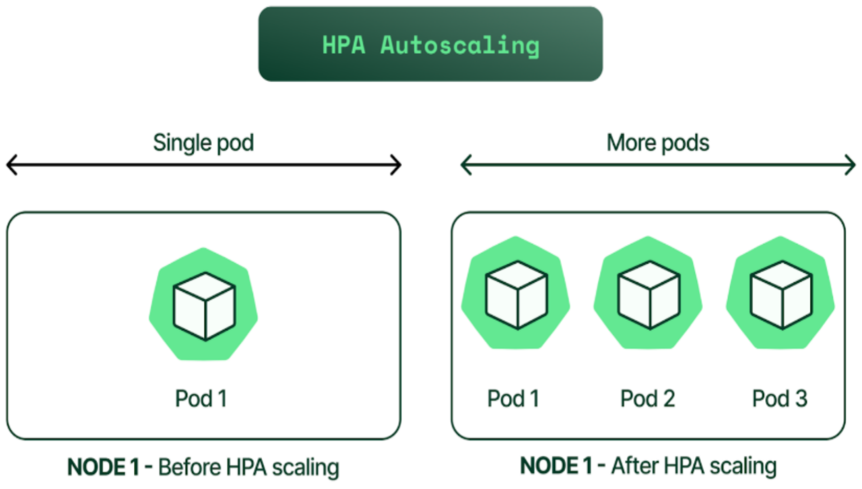


Fig 5: HPA Autoscaling [5]

b. Recovery

Data integrity and robustness of the system are also a very essential part of our design. In order to ensure this, we deployed PostgreSQL as a StatefulSet so that the database could maintain a uniform network identity and retain data in the event of pod restart or failure. Data loss is averted, and transaction consistency ensured. Further, we added Velero for automatic backup on Azure to provide scheduled backup and disaster recovery functionalities. This makes the recovery of data feasible even in the event of a complete failure, recovering it rapidly with reduced downtime.

We further added Liveness and Readiness probes to provide real-time monitoring of the health of application components. The probes assist Kubernetes to detect failure and automatically restart unresponsive services, thereby enhancing system self-healing capabilities. Our decision to use StatefulSets, Velero backups, and Kubernetes probes over regular backup scripts or manual failover came from the imperative to automate, scale, and disrupt operations as little as possible. These strategies encourage the system to be highly available in the face of unplanned failure by reducing human intervention and recovery by a long shot.

c. Deployment

A strong CI/CD pipeline was established leveraging GitHub Actions that easily automated build, test, and deployment tasks. The pipeline validates each change to the code base before going into production to ensure that it's less likely to introduce bugs. To achieve more reliable deployment, we incorporated a Blue-Green Deployment policy under which two iterations of the application run together. This allows the new release (Green) to be tested in production environment without routing traffic to it yet, as traffic is still routed to the existing stable release (Blue).
If the new deployment is stable, traffic is routed automatically with zero downtime and without any risk of service disruption. This approach was used over rolling updates since it possesses an immediate rollback capability if deployment fails, so it is a safer and more controlled process of deployment. Compared to the traditional approaches to deployment that involve service interruption, Blue-Green Deployment

provides zero-downtime updates, faster recovery, and a smoother user experience overall.

With the utilisation of automated scaling capabilities, robust recovery mechanisms, and advanced deployment methods, we have created an available, fault-resilient, and scalable online bookshop application in line with advanced DevOps practices. Together, the practices optimise the performance of the system, minimise downtime, and enhance operational efficiency to enable seamless interaction to the users while achieving cost savings and reliability in cloud-native environments.

4. Design Description

The bookstore application employs a modular micro-services architecture to enable flexibility, elasticity, and traceability.

High-Level System Design:

- Load Balancer distributes incoming user requests and sends the traffic effectively to the system.
- The requests are presented to the Nginx Ingress Controller and routed to the respective micro-service.
- Three main micro-services make up the system:
    - **Frontend Service (React):** Manages the customers' user interface.
    - **Catalog Service (Node.js/Express):** Controls book stock, metadata, and availability.
    - **Cart Service (Node.js/Express):** Handles shopping cart functionality and user sessions.

The backend services interact with the PostgreSQL database, which runs as a StatefulSet on Azure Kubernetes Service (AKS) to ensure data persistence and consistency.

Lower-Level Implementation:

The entire system is containerised using Docker in a way that makes it portable and deployable without any hassle.

- **Docker Hub** keeps the **container images**; thus, effortless environment integration is provided.
- Services are being executed on AKS, utilising Kubernetes capabilities for orchestration, scalability, and high availability.
- The **Nginx Ingress Controller**:
    - Handles incoming traffic and API routing.
    - Enforces **centralised routing rules** for optimised request forwarding.
- **PersistentVolumeClaims (PVCs)** guarantee **data preservation**, even during **container restart or scaling events**.

4.1 Justification for Design Choices

The bookstore application design was motivated by the requirement for maintainability, resilience, and scalability, and that was to provide an efficient and smooth system with little operational overhead.

- **Scalability with AKS and HPA**

  The use of Azure Kubernetes Service (AKS) makes cluster management easier with in-built scalability and monitoring. Horizontal Pod Autoscaler (HPA) was used to automatically scale different workloads. The system has the capability to automatically scale pods based on CPU and memory consumption. This helps the application to smoothly operate even in high traffic without any manual intervention. Besides, Cluster Autoscaler was also set to add new worker nodes when necessary to avoid bottlenecks and ensure optimal responsiveness.

  These methods provide a cost-effective and flexible substitute for manual scaling since they enable the system to react instantly to traffic bursts without pre-provisioning too many resources that may go idle. This practice, coupled with load testing with k6, validated the capacity of the system to support thousands of simultaneous users without any degradation in performance.

  Besides, AKS scaling policies were also tuned to optimise resource utilisation based on existing demand, so the system scales perfectly without extra overhead. Besides, integrating auto-scaling with proactive monitoring enables potential performance bottlenecks to be identified and removed before impacting users. In addition, AKS node pool management guarantees optimal assignment of heterogeneous workloads, maximising the resource utilisation of available nodes. Such enhancements are synergistic and maximise system resilience, cost savings, and automated scaling, with the application capable of coping with unforeseen traffic spikes and fluctuating workloads.

- **Resilience and Data Recovery with StatefulSets and Velero**

  Preservation of data integrity and resilience of the system was a major design point of consideration. PostgreSQL was implemented as a StatefulSet, which allows the database to be given a consistent network identity and preserve data during pod restart or crashes. This offers transactional consistency and prevents data loss, hence being a more reliable choice than stateless deployments. Besides, Velero was used for automated scheduled backups and disaster recovery, enabling quick data recovery in case of failure. This automated backup process is far more efficient and scalable compared to manual backup scripts because it reduces recovery time and minimises human intervention.

  Besides, Liveness and Readiness probes for Kubernetes were applied to monitor application component health permanently. The probes enable Kubernetes to recognise failures and auto-restart stuck services and so make the system self-heal more. Contrary to the traditional failover processes run by hand, such technique's advance fault tolerance with service interruption guarantees in high-availability scenarios.

  Additionally, Velero backup policies were further optimised so that incremental backups could be performed with optimal speed and with minimal overhead of storage but keeping full recovery functionality intact. Backup verification routines were also included to run data restoration tests periodically to ensure that not only are backups created but are also recoverable whenever required. Also, permanent storage capacities were allocated judiciously with fitting retention policies to avoid accidental data loss since system performance was kept up. All these self-utilising solutions enhance system performance resilience in that important information is preserved and easily accessible in the case of breakdown.

- **Efficient Deployment with CI/CD and Blue-Green Deployment**

  To automate build and deployment, a robust CI/CD pipeline using GitHub Actions was established. This makes all code changes go through automated builds, tests, and deploys, reducing the chances of putting bugs into production. Blue-Green Deployment strategy was used for deployments, allowing two versions of the applications to run side-by-side. This guarantees complete testing of new releases in the production environment before allowing traffic to be directed to them, minimising deployment risks. Blue-Green Deployment is not like rolling updates since it allows rollback instantly, a safer and controlled way with zero downtime.

  Utilising AKS for scaling, StatefulSets and Velero for high availability, and CI/CD using Blue-Green Deployment for worry-free updates, the solution has been made fault-tolerant, highly available, and cloud-native. Such configurations are based on contemporary DevOps best practices, which give the best possible performance, reduced downtime, and user experience with ease while ensuring operational expenses in check.

  Furthermore, GitHub Actions workflows were created to integrate staged testing and approval gates where only thoroughly validated changes reached the production environment. Automated unit, integration, and security scanning testing were part of the CI/CD pipeline where potential issues were caught early on in the software development process. Monitoring of the deployment status through real-time notification was also undertaken, allowing for developers to take quick action for failures or unusual occurrences. These enhancements also make the deployment process more stable so that the latest updates never destabilise the system.

To further improve deployments, feature flagging controls were implemented to allow selective enablement of new features without complete deployments. This facilitates controlled rollouts whereby functionalities can be enabled for particular users before general release, minimising risks in massive updates. Secondly, post-deployment performance monitoring was included through Prometheus and Grafana so that teams can monitor the effect of new releases on system performance. Through a combination of Blue-Green Deployment, automation of CI/CD, feature flagging, and active monitoring, the system is able to deliver continuously with high availability and low risk and provide a smooth and controlled release of software.

5. Implementation Overview

The bookstore micro-services application was developed using React.js for Frontend and Node.js with Express for the Catalog and Cart services, and PostgreSQL as the central database, all deployed on Azure Kubernetes Service (AKS). The system was made scalable, fault-tolerant, and highly available with modern DevOps practices such as containerisation, CI/CD automation, and monitoring. The architecture is characterised by peak performance, unbroken communication between services, and failure-resistant infrastructure to support dynamic workloads and bursts of traffic.

Frontend Service (React) is the web presentation and provides users with access to browsing and purchasing books. It communicates with the Catalog Service (Express/Node.js), which serves book catalogs, metadata, and inventory. Cart Service (Express/Node.js) is responsible for shopping cart transactions, user session state management, and order processing. Backend services communicate with the PostgreSQL database, which is executed as a StatefulSet on AKS for persistent storage and data consistency.

All the micro-services were containerised using Docker, and images were stored in Docker Hub for portability and deployment convenience. Kubernetes manifests were created to declare deployments, services, and ingress so that orchestration can be easily done in AKS. Nginx Ingress Controller controlled traffic routing and can manage API requests efficiently and do TLS termination.

To enable automated deployment and testing, a CI/CD pipeline with GitHub Actions was put in place to ensure continuous integration and delivery. The system uses a Blue-Green Deployment pattern, enabling zero-downtime updates by running two instances of a service concurrently and directing traffic only to the stable release. Monitoring and observability were added using Prometheus and Grafana, giving real-time visibility into system performance. Velero was also used for automated backups, enabling disaster recovery and data protection.

Using Azure Kubernetes Service, Docker, CI/CD pipelines, and monitoring tools, the application was successfully implemented as a cloud-native, scalable, and fault-tolerant online bookstore to provide a seamless and efficient experience for users.

Along with this, API Gateway mechanisms were integrated to process incoming requests in an efficient manner and implement rate limiting for increased security and traffic management. This prevents server overload by providing equal utilisation of resources by multiple consumers. Circuit breaker patterns were also implemented in the system within micro-services to prevent cascading failures by separating rogue services and routing traffic to maintain application stability. These techniques introduce a fault tolerance and ruggedness element into the application so that it still operates even under high traffic or lost partial service.

To further enhance database interaction, connection pooling was activated in the PostgreSQL database to reduce run time for queries and save resources. Data indexing methods were implemented to speed up lookup and retrieving book listings and cart transactions. Database load was equalised with imperceptible delay with the aid of read replicas to establish an interruption-free purchase process for clients. Such attributes yield scalability together with high availability to eradicate traffic congestion as it tends to gather through users.

Besides, excessive logging and monitoring were necessary with Prometheus and Grafana to facilitate real-time monitoring of system performance and resource usage. Custom dashboards were developed to graphically display critical metrics such that developers can easily identify potential bottlenecks and performance issues. Log storage configurations were also set to ensure that key logs are stored for debugging purposes while keeping storage costs unnecessary to a minimum. By using efficient logging, active monitoring, and automated notifications, the system ensures high availability, efficient diagnosis, and seamless operation under fluctuating workloads.

Apart from optimisation for performance and user experience, optimisation techniques in content delivery were utilised. The frontend assets were compressed and minified for purposes of saving the initial page load time, and mechanisms for cache were implemented for optimising API response time. Lazy loading techniques were also used within the React frontend for purposes of preventing the loading of only required components on demand and saving the initial load time as well as providing responsiveness. These additions make the bookstore app more scalable, efficient, and user-friendly, with a seamless experience even during peak traffic.

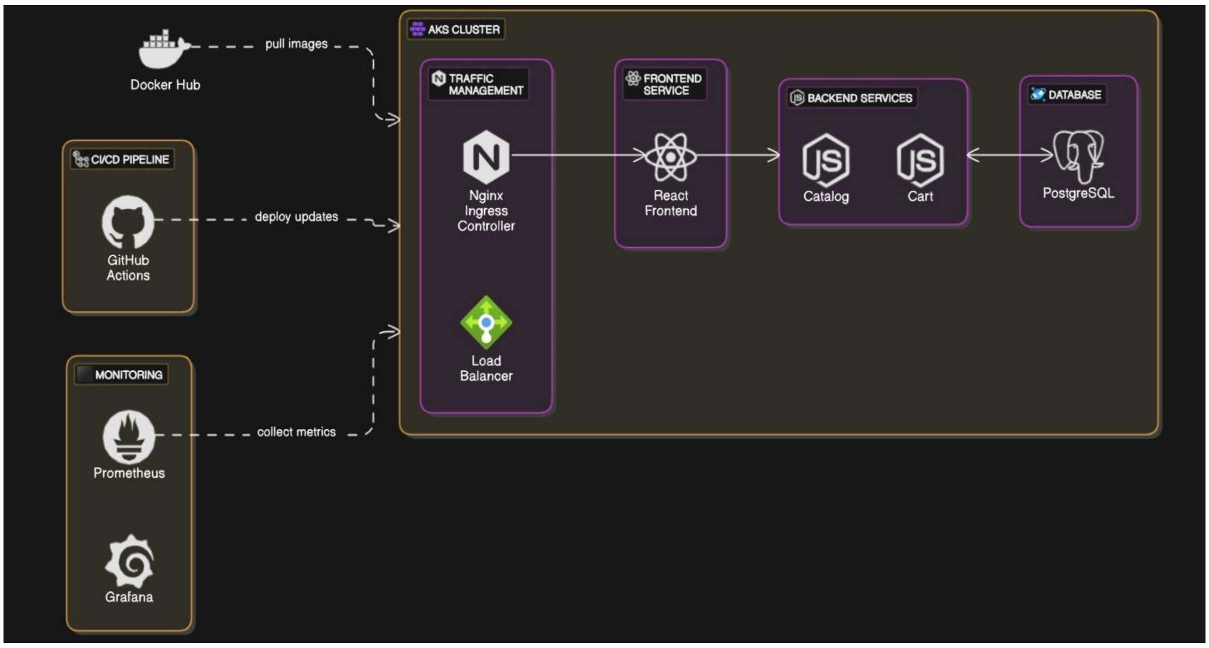


Fig 6: Architecture diagram [7]

5.1 Challenges and Solutions

A couple of problems were encountered while deploying the application to the bookstore micro-services, and certain strategic steps had to be undertaken in trying to attain scalability, reliability, and seamless deployments.

The biggest problem that had to be resolved was Ingress misconfiguration, where user requests were being routed to backend services in a wrong manner. This is because rules had not been specified properly within the Nginx Ingress Controller. To correct this issue, Ingress was tuned, with path-based routing and TLS termination, so that traffic could be sent to the correct services properly and securely.

The other major concern was the downtime during deployment, where new updates would at times be released with shutdowns of the service. This was addressed with the implementation of a Blue-Green Deployment strategy. This allowed both the original and new versions of the application to be run in parallel, such that only stable releases of the application were given traffic. This not only did away with downtime but also gave an immediate rollback system in case of failures.

Another challenge was database performance optimisation with increasing workloads. With increasing numbers of concurrent users, database queries became longer, affecting catalog search and cart responsiveness of performance. Query optimisation techniques were employed to reduce this, such as indexing most accessed fields and SQL statement optimisation to reduce execution time. Connection pooling also served to cancel out database connections in an attempt to avoid utilisation of resources. These enhancements significantly enhanced database performance, such as response time and also a responsive user interface under heavy loads.

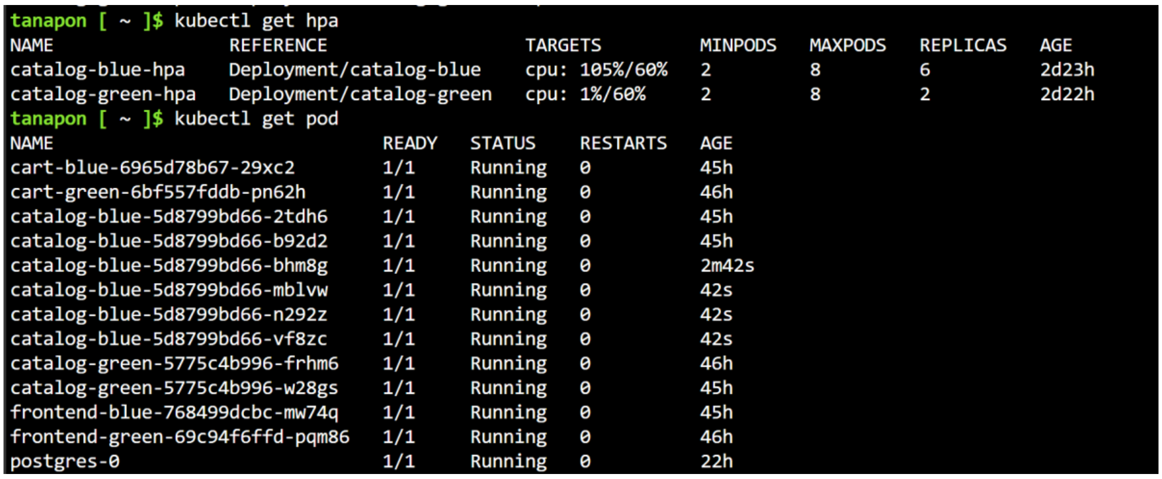
Besides, the system had its initial challenges in dealing with traffic spikes, resulting in decreased performance under heavy load. This was addressed by setting up Kubernetes Horizontal Pod Autoscaler (HPA) to automatically scale the number of active pods depending on CPU and memory consumption. This guaranteed scaling to demand automatically without human interference, greatly enhancing responsiveness and reliability of the system.

By implementing these solutions, the system was able to overcome its problems effectively, delivering efficient traffic management, flawless deployments, and dynamic scalability.

6. Test Plan

For the bookstore micro-services application to provide stability, reliability, and performance, we performed the following to test:

1. **Testing Horizontal Pod Autoscaling:** We tested the horizontal pod autoscaling by running a k6 script, which would simulate traffic and would go up to 500 users and would last for around 5 minutes. We configured horizontal pod autoscaling to increase pods replicas from 2 to 8, whenever the CPU utilization is more than 60 percent.

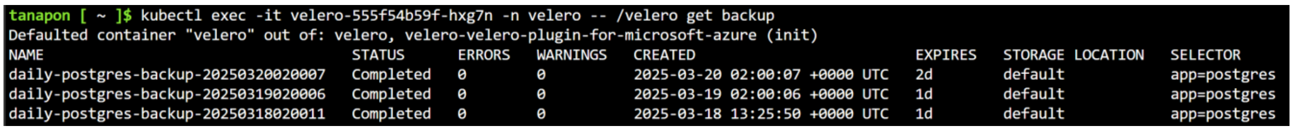


```
tanapon [ ~ ]$ kubectl get hpa
NAME                REFERENCE                   TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
catalog-blue-hpa    Deployment/catalog-blue     cpu: 105%/60%   2         8         6          2d23h
catalog-green-hpa   Deployment/catalog-green    cpu: 1%/60%     2         8         2          2d22h
tanapon [ ~ ]$ kubectl get pod
NAME                               READY   STATUS    RESTARTS   AGE
cart-blue-6965d78b67-29xc2         1/1     Running   0          45h
cart-green-6bf557fddb-pn62h        1/1     Running   0          46h
catalog-blue-5d8799bd66-2tdh6      1/1     Running   0          45h
catalog-blue-5d8799bd66-b92d2      1/1     Running   0          45h
catalog-blue-5d8799bd66-bhm8g      1/1     Running   0          2m42s
catalog-blue-5d8799bd66-mblvw      1/1     Running   0          42s
catalog-blue-5d8799bd66-n292z      1/1     Running   0          42s
catalog-blue-5d8799bd66-vf8zc      1/1     Running   0          42s
catalog-green-5775c4b996-frhm6     1/1     Running   0          46h
catalog-green-5775c4b996-w28gs     1/1     Running   0          45h
frontend-blue-768499dcbc-mw74q     1/1     Running   0          45h
frontend-green-69c94f6ffd-pqm86    1/1     Running   0          46h
postgres-0                         1/1     Running   0          22h
```

Fig 7: Showing increase in pods under heavy load

From Fig 7, it can be concluded that whenever the CPU utilization was more than 60 percent, the pods scaled. In our case when the CPU utilization was 105% for catalog-blue deployment, the pods scale from 2 to 6. When we ran the command *'Kubectl get pod',* it can be stated that there were 6 po replicas of catalog-blue deployment.

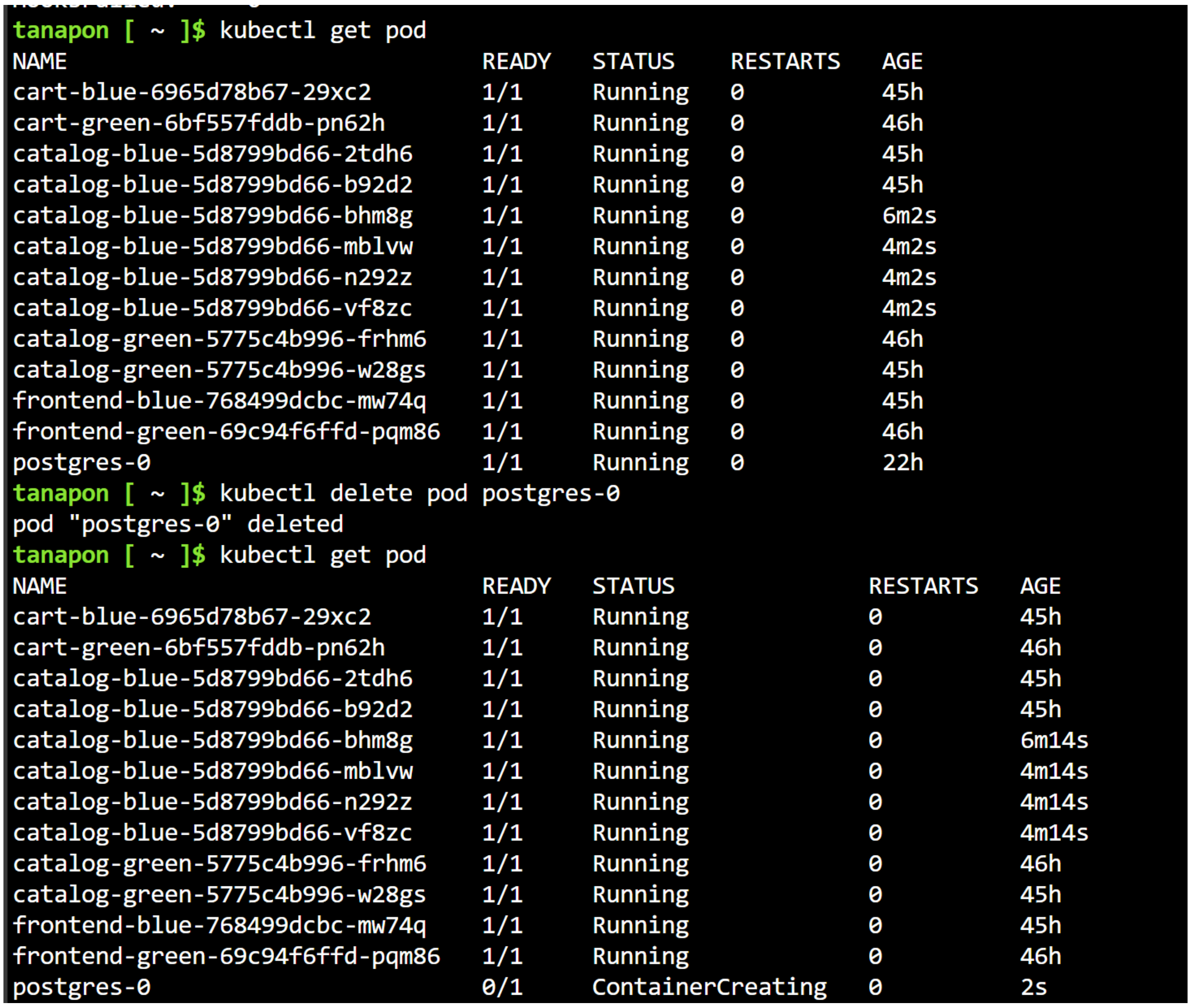
2. **Testing restoration of data with velero:** We configured velero to back up the bookstore app's data at 2 am UTC in every 2 hours, Fig 8 shows the daily backups from the last three days.



```
tanapon [ ~ ]$ kubectl exec -it velero-555f54b59f-hxg7n -n velero -- /velero get backup
Defaulted container "velero" out of: velero, velero-velero-plugin-for-microsoft-azure (init)
NAME                                STATUS      ERRORS   WARNINGS   CREATED                         EXPIRES   STORAGE LOCATION   SELECTOR
daily-postgres-backup-20250320020007   Completed   0        0          2025-03-20 02:00:07 +0000 UTC   2d        default            app=postgres
daily-postgres-backup-20250319020006   Completed   0        0          2025-03-19 02:00:06 +0000 UTC   1d        default            app=postgres
daily-postgres-backup-20250318020011   Completed   0        0          2025-03-18 13:25:50 +0000 UTC   1d        default            app=postgres
```

Fig 8: Daily Velero back ups

To test the restoration from velero we deleted the Postgres pod.

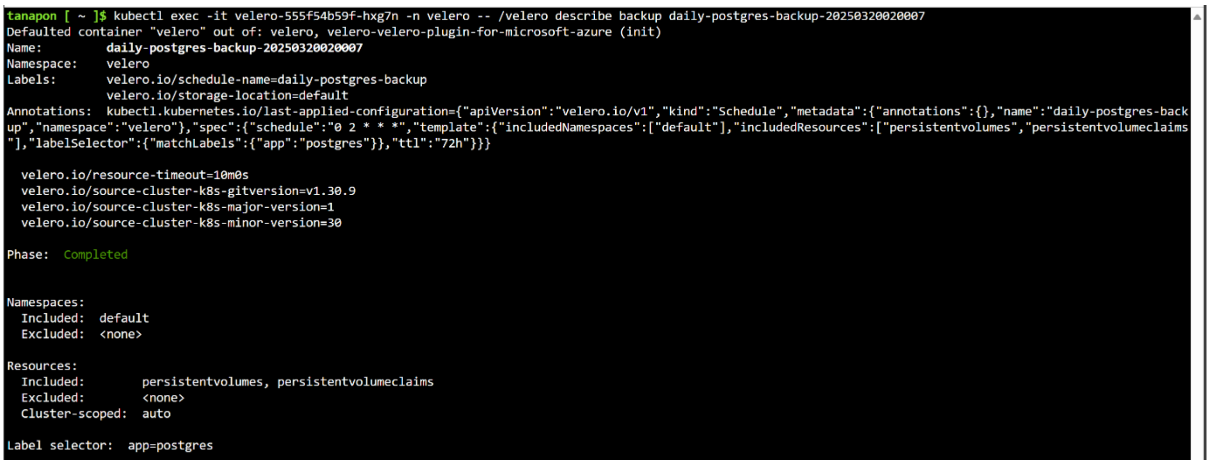


```
tanapon [ ~ ]$ kubectl get pod
NAME                               READY   STATUS    RESTARTS   AGE
cart-blue-6965d78b67-29xc2         1/1     Running   0          45h
cart-green-6bf557fddb-pn62h        1/1     Running   0          46h
catalog-blue-5d8799bd66-2tdh6      1/1     Running   0          45h
catalog-blue-5d8799bd66-b92d2      1/1     Running   0          45h
catalog-blue-5d8799bd66-bhm8g      1/1     Running   0          6m2s
catalog-blue-5d8799bd66-mblvw      1/1     Running   0          4m2s
catalog-blue-5d8799bd66-n292z      1/1     Running   0          4m2s
catalog-blue-5d8799bd66-vf8zc      1/1     Running   0          4m2s
catalog-green-5775c4b996-frhm6     1/1     Running   0          46h
catalog-green-5775c4b996-w28gs     1/1     Running   0          45h
frontend-blue-768499dcbc-mw74q     1/1     Running   0          45h
frontend-green-69c94f6ffd-pqm86    1/1     Running   0          46h
postgres-0                         1/1     Running   0          22h
tanapon [ ~ ]$ kubectl delete pod postgres-0
pod "postgres-0" deleted
tanapon [ ~ ]$ kubectl get pod
NAME                               READY   STATUS              RESTARTS   AGE
cart-blue-6965d78b67-29xc2         1/1     Running             0          45h
cart-green-6bf557fddb-pn62h        1/1     Running             0          46h
catalog-blue-5d8799bd66-2tdh6      1/1     Running             0          45h
catalog-blue-5d8799bd66-b92d2      1/1     Running             0          45h
catalog-blue-5d8799bd66-bhm8g      1/1     Running             0          6m14s
catalog-blue-5d8799bd66-mblvw      1/1     Running             0          4m14s
catalog-blue-5d8799bd66-n292z      1/1     Running             0          4m14s
catalog-blue-5d8799bd66-vf8zc      1/1     Running             0          4m14s
catalog-green-5775c4b996-frhm6     1/1     Running             0          46h
catalog-green-5775c4b996-w28gs     1/1     Running             0          45h
frontend-blue-768499dcbc-mw74q     1/1     Running             0          45h
frontend-green-69c94f6ffd-pqm86    1/1     Running             0          46h
postgres-0                         0/1     ContainerCreating   0          2s
```

Fig 9: Deleting 'postgres-0' pod

Fig 9 shows we deleted the Postgres pod and upon deletion the pod would respawn with status showing 'ContainerCrearting'

To retrieve the backup, we used one of the Velero generated backups to restore the data which is shown in Fig 10.

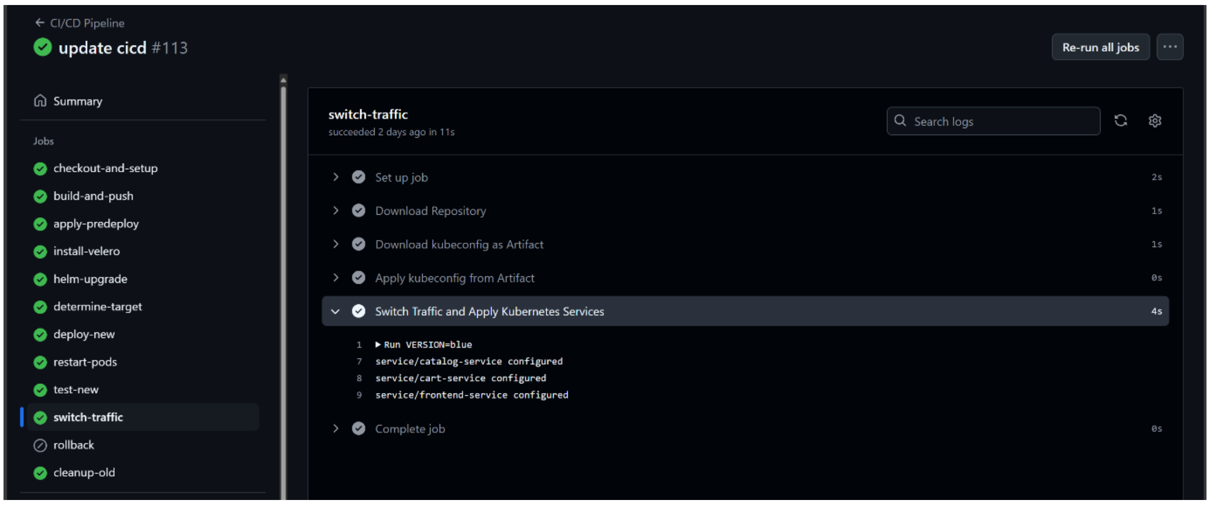


```
tanapon [ ~ ]$ kubectl exec -it velero-555f54b59f-hxg7n -n velero -- /velero describe backup daily-postgres-backup-20250320020007
Defaulted container "velero" out of: velero, velero-velero-plugin-for-microsoft-azure (init)
Name:         daily-postgres-backup-20250320020007
Namespace:    velero
Labels:       velero.io/schedule-name=daily-postgres-backup
              velero.io/storage-location=default
Annotations:  kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"velero.io/v1","kind":"Schedule","metadata":{"annotations":{},"name":"daily-postgres-back
up","namespace":"velero"},"spec":{"schedule":"0 2 * * *","template":{"includedNamespaces":["default"],"includedResources":["persistentvolumes","persistentvolumeclaims
"],"labelSelector":{"matchLabels":{"app":"postgres"}},"ttl":"72h"}}}

  velero.io/resource-timeout=10m0s
  velero.io/source-cluster-k8s-gitversion=v1.30.9
  velero.io/source-cluster-k8s-major-version=1
  velero.io/source-cluster-k8s-minor-version=30

Phase:  Completed


Namespaces:
  Included:  default
  Excluded:  <none>

Resources:
  Included:        persistentvolumes, persistentvolumeclaims
  Excluded:        <none>
  Cluster-scoped:  auto

Label selector:  app=postgres
```

Fig 10: Restoring data by using one of velero backups

3. Testing rollback functionality in case of a failed Deployment:

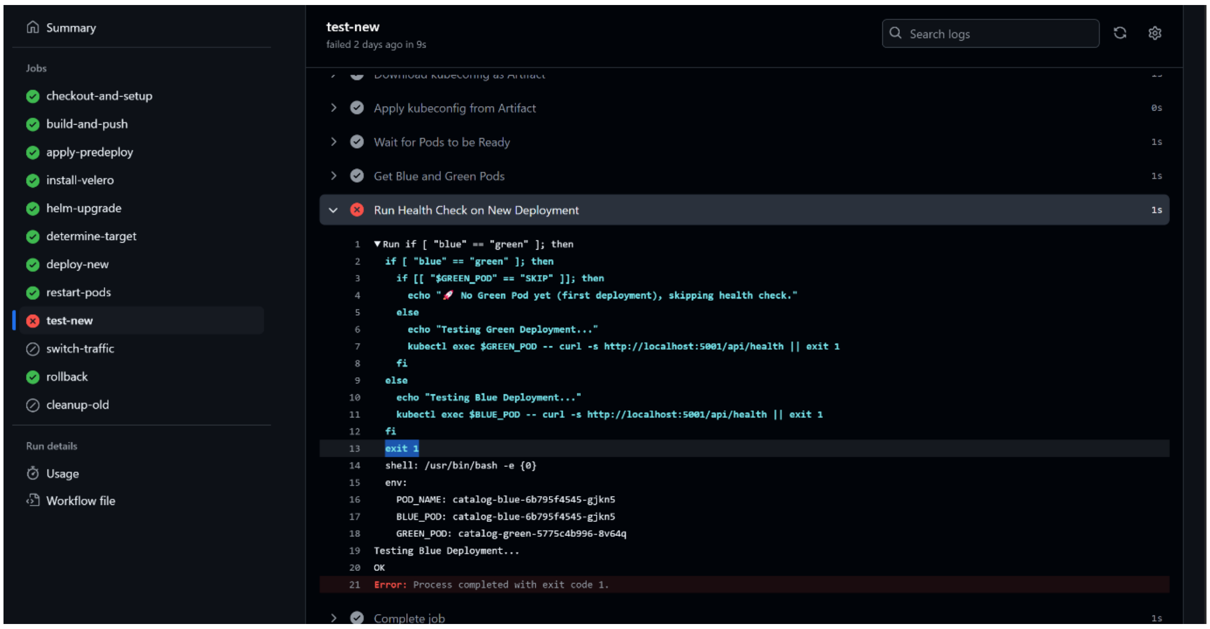
We implemented blue-green deployment for this project, which is evident in Fig 11.



```
▶ Run VERSION=blue
service/catalog-service configured
service/cart-service configured
service/frontend-service configured
```

Fig 11: *Successful green-blue deployment*

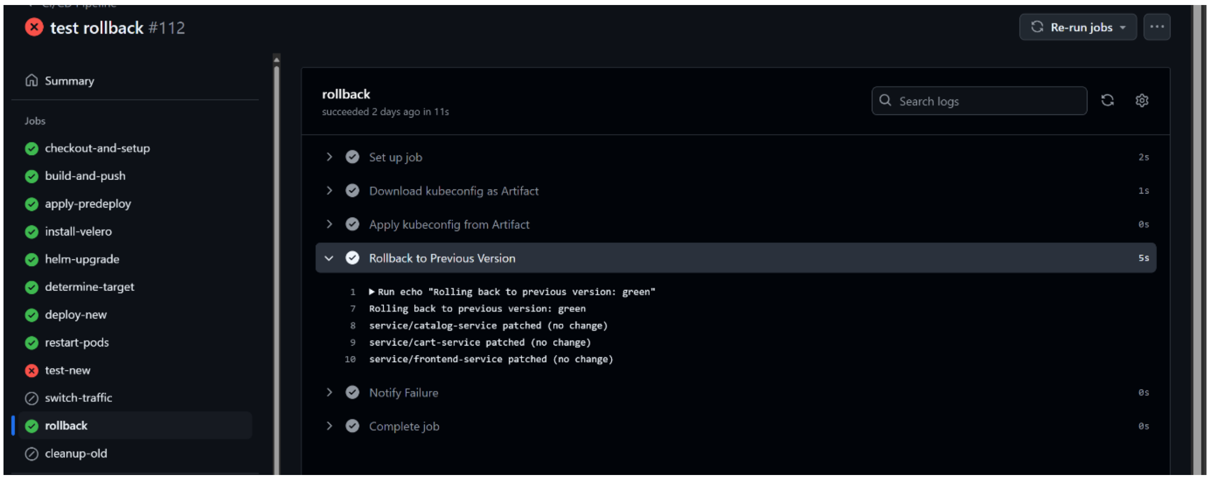
To test the rollback functionality, we on purpose started deployment which would fail, which is shown in Fig 12.



```
▼Run if [ "blue" == "green" ]; then
  if [ "blue" == "green" ]; then
    if [[ "$GREEN_POD" == "SKIP" ]]; then
      echo "🚀 No Green Pod yet (first deployment), skipping health check."
    else
      echo "Testing Green Deployment..."
      kubectl exec $GREEN_POD -- curl -s http://localhost:5001/api/health || exit 1
    fi
  else
    echo "Testing Blue Deployment..."
    kubectl exec $BLUE_POD -- curl -s http://localhost:5001/api/health || exit 1
  fi
  exit 1
  shell: /usr/bin/bash -e {0}
  env:
    POD_NAME: catalog-blue-6b795f4545-gjkn5
    BLUE_POD: catalog-blue-6b795f4545-gjkn5
    GREEN_POD: catalog-green-5775c4b996-8v64q
Testing Blue Deployment...
OK
Error: Process completed with exit code 1.
```

Fig 12: *Failing a deployment on purpose*

In case of a failed deployment, our rollback functionality will roll back to the previous version, which can be seen in Fig 13.



test rollback #112

Re-run jobs

Summary

Jobs

- ✓ checkout-and-setup
- ✓ build-and-push
- ✓ apply-predeploy
- ✓ install-velero
- ✓ helm-upgrade
- ✓ determine-target
- ✓ deploy-new
- ✓ restart-pods
- ✗ test-new
- ⊘ switch-traffic
- ✓ rollback
- ⊘ cleanup-old

rollback
succeeded 2 days ago in 11s

Search logs

Set up job	2s
Download kubeconfig as Artifact	1s
Apply kubeconfig from Artifact	0s
Rollback to Previous Version	5s

```
▶ Run echo "Rolling back to previous version: green"
Rolling back to previous version: green
service/catalog-service patched (no change)
service/cart-service patched (no change)
service/frontend-service patched (no change)
```

Notify Failure	0s
Complete job	0s

Fig 13: Rollback functionality working well.

7. Critical Evaluation

7.1 Self

Our design decisions had a significant impact on the scalability and functionality of the product. For example, the decision to containerise all micro-services using Docker and deploy on Azure Kubernetes Service (AKS) facilitated easy scalability and resilience. This decision allowed the application to effectively handle variable loads, such that it was able to scale both horizontally and vertically as needed. Yet this option also required an in-depth understanding of container orchestration and entailed greater initial setup complexity.

The other critical design choice was the use of micro-services architecture. While it brought advantages such as modularity and independent scalability for services such as frontend, catalog, and cart services, it made the provision of transparent communication between services challenging. The use of Postman for integration testing and JMeter for load testing helped to mitigate integration issues and performance bottlenecks and thus render the system efficient in traffic management. But, maintaining the communication synchronised and the services aligned added overhead and complexity during development.

Use of Kubernetes health checks (readiness and liveness probes) was one option that helped to build system resilience. It helped in automatically detecting pods that failed and restarting them, hence minimising downtime. To ensure these health checks were properly set up, however, close monitoring of the startup times of the micro-services and interdependencies was required, which could be a cause of surprise issues on first deployment.

Furthermore, the focus on security with the implementation of RBAC and private networking also highly improved the integrity and security of the application. Nevertheless, there were some concerns in trying to balance security and accessibility, particularly in exposing services to be easily accessible but still secure against attacks from outside.

Overall, the design decisions made were instrumental in the stability, scalability, and security of the system. But they brought along with them complexities and trade-offs that had to be skill fully balanced to make the deployment of the product a success.

7.2 Relative

- **Comparison with group-5**

  1. Choice of Google Cloud Platform (GCP)

     Group 5 had initially faced challenges with AWS, i.e., limited resources and complex Kubernetes setup, and therefore shifted to GCP based on lower prices, easy setup, and better automation.
     The ease of use in GCP reduced the hassle of the deployment management of Kubernetes in comparison to AWS.

     Dollar savings were significant, since GCP's minute-level billing gave tighter control over costs than Azure's hour-based billing method.

  2. Backup and Disaster Recovery

     Velero was used for PostgreSQL backup, but Group 5 augmented it with Google Cloud Storage (GCS) to offer cloud storage and retrieval at ease.

     Their structured backup retention policy (daily backups for 7 days) ensured that data was protected and could be restored in case of failures.

     Automated backup timestamps on Persistent Volume Claims (PVCs) ensured that database snapshots remained current.

  3. Scaling and Load Testing

     Horizontal Pod Autoscaler (HPA) was employed to dynamically scale the Catalog Service based on the existing traffic patterns.

     Cluster Autoscaler in GCP automatically adjusted the worker nodes, ensuring optimum resource utilisation and cost savings.

     Load testing with k6 was performed to validate system performance at peak loads to ensure that the application could handle sudden spikes of traffic.

- **Critique and Key Differences**

  1 Cloud Provider Choice:

  Group 5 launched on Google Kubernetes Engine (GKE) due to cost savings and simplicity of management, while our Group 7 launched on Azure Kubernetes Service (AKS) for better integration with Microsoft cloud infrastructures.

  GCP's pricing model (minute-based billing) gave Group 5 more cost control, while Azure's hourly rate billing can result in higher operational costs.

  2 Backup and Disaster Recovery:

Both groups used Velero for automated backups, but Group 5 supported a 7-day retention policy for backups, while Group 7 used standard backups without a planned retention policy.

Group 5 similarly used Persistent Volume Claims (PVC) annotations to provide homogeneous database snapshot backups, which might be an extension for our system.

3 Scaling and Performance Optimisation:

Both groups used HPA for autoscaling pods, but Group 5's GCP Cluster Autoscaler had stronger multi-region failover support, whereas Azure's Cluster Autoscaler is essentially region-focused.

Group 5 optimised costs with effective autoscaling of worker nodes, whereas Azure autoscaling is slightly less flexible in cost optimisation.

4 Monitoring and Observability:

Group 5 employed Google Cloud native monitoring features, whereas our Group 7 utilised Prometheus and Grafana for customised observability.

GCP's in-built monitoring systems provide tighter integration with their cloud services, but Prometheus/Grafana provide more flexible metrics.

- **Improvements we can implement from Group 5**

1) Structured Backup Retention Policy

Implementing a structured backup policy like Group 5's 7-day retention would enhance our disaster recovery process.

Adding PVC annotations for enhanced database snapshot control would also increase backup consistency.

2) Cost-Effective Autoscaling

Group 5's GCP Cluster Autoscaler automatically dynamically scales worker nodes and reduces unnecessary use of cloud resources.

We can further explore more advanced auto-scaling features in Azure to attain cloud cost optimisation with high availability.

3) Improving Multi-Region Deployment

GCP supports multi-region autoscaling, which adds system reliability against failure.

Implementation of Azure Multi-Region Deployment can improve our system's fault tolerance and availability.

- **Comparison with group-6**

  To quantify the comparison between the design alternatives and deployment strategy, we compared our team's bookstore micro-services application with Group 6's work. Both projects employed a micro-services architecture and used Kubernetes for container orchestration but with differences in deployment, monitoring, and scalability methods.

- **Strength of Group 6's Design**

  1 Effective Use of AWS Services

     Group 6 implemented their system on AWS Elastic Kubernetes Service (EKS) with Cluster Auto Scaling and Auto Scaling Groups (ASG) for resource management efficiently.

     Using AWS CloudWatch for performance monitoring gave real-time visibility to enable proactive system tuning.

  2 Strong Backup Strategy

     Like our solution with Velero, Group 6 utilised automated database backup on AWS S3 and EBS. This guaranteed data resiliency and disaster recovery.

  3 Scalability and Performance Optimisation

     They were able to leverage HPA for their Catalog Service, dynamically scaling replicas of pods based on real-time usage of resources. K6 load testing was implemented to simulate traffic bursts and capture system performance.

- **Critique and Comparison**

  1 Cloud Provider Choice:

     Group 6 used AWS (EKS, CloudWatch, ASG) while we used Azure Kubernetes Service (AKS), Prometheus/Grafana for monitoring and Azure Velero for backup.

     AWS CloudWatch provides deeper cloud-native monitoring while Prometheus and Grafana provide more configurable observability in our system.

  2 Scaling Mechanisms:

     Both squads used Horizontal Pod Autoscaler (HPA) for dynamic scaling. Group 6's Cluster Auto Scaling with ASG had better workload balancing compared to our AKS Cluster Autoscaler, which was regionally based.

  3 Deployment Strategies:

     Blue-Green Deployment was used by the two teams to execute zero-downtime deployments with simple rollback. Group 6's pipeline through GitHub Actions was similar to ours, with automated builds, tests, and deployments.

- **Improvements We Could Adopt from group 6**

  1 Better Cloud Monitoring

  While Prometheus and Grafana are excellent for monitoring, the use of Cloud-native observability tools (e.g., Azure Monitor similar to AWS CloudWatch) could provide more informative performance metrics.

  2 Automated Load Balancing Across Availability Zones

  Group 6's Auto Scaling across ASG clusters distributes workloads across multiple AWS Availability Zones, attaining fault tolerance. We can consider the same strategy with Azure's Multi-Region Deployment for increased resiliency.

8. Group working

Team Member	Task/Responsibility	% Contribution
1. Aryan Katyayan	Configured the **Horizontal Pod Autoscaler (HPA)** for the Catalog Service. Conducted **load testing with k6** to evaluate scalability under peak traffic conditions. Analysed test results and optimised system performance. (Phase 2) Contributed to the **presentation slides** as well create presentation slides.	20%
2. Bansi K. Dobariya	Developed the **Frontend Service** using **React** and the **Backend (API layer) with Node.js**. Integrated APIs for seamless communication between services. Contributed to **report writing**.	20%
3. Tanapon Suwankesawong	Integrated **Prometheus and Grafana** for monitoring and system observability. Documented the project setup, configuration, and testing procedures. Provided insights into monitoring metrics and system health. Played a key role in **report writing**.	20%

4. Shubh Anand	Designed and implemented the **CI/CD pipeline** using **GitHub Actions** for automated testing and deployment. Configured **Blue-Green Deployment** for zero-downtime releases and rollback in case of failures. (Phase 4) Contributed to the **presentation slides** as well create presentation slides.	20%
5. Sanjana T. Shahu	Configured **Liveness and Readiness probes** for monitoring. Deployed **PostgreSQL as a StatefulSet** for data consistency. Set up **Velero backups** on Azure and simulated database crash recovery. (Phase 3) Played a key role in **report writing**.	20%

9. Conclusion

If we were redoing the project, we would focus on the improvement of some key areas in order to ensure the overall infrastructure and security of the system to be improved. Infrastructure as Code (IaC) would be used in provisioning AKS so that standard and automated process of deployment would be attained. Security improvements would also be prioritised, i.e., better Role-Based Access Control (RBAC) and private networking in order to keep sensitive resources and data secure. Also, we would add distributed tracing tools like Jaeger to introduce more observability and enable improved diagnosis among micro-services.

On the other hand, there are some in which we would not be duplicating old patterns. For instance, we would avoid using manual installation of the Kubernetes cluster, opting instead for complete automation by using Terraform to optimise efficiency and reduce human operation-related errors. We would also ensure that database services aren't openly exposed, using private networking as a way of not exposing ourselves unnecessarily to external vulnerability. These changes would allow us to create a more secure, stable, and scalable solution in the future.

10. References:

[1] Kubernetes. (n.d.). Horizontal Pod Autoscaler. [online] Available at: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[2] Grafana Labs. (n.d.). Grafana k6 documentation | Grafana k6 documentation. [online] Available at: https://grafana.com/docs/k6/latest/.

[3] Velero.io. (2025). *Velero Docs - Overview*. [online] Available at: https://velero.io/docs/v1.9/ [Accessed 20 Mar. 2025].

[4] Storj.dev. (2025). *Velero-based Kubernetes Backup Guide - Storj Docs*. [online] Available at: https://storj.dev/dcs/third-party-tools/velero [Accessed 20 Mar. 2025].

[5] Kubecost.com. (2021). *The Guide To Kubernetes HPA by Example*. [online] Available at: https://www.kubecost.com/kubernetes-autoscaling/kubernetes-hpa/ [Accessed 20 Mar. 2025].

[6] Napkin AI (2024). *Napkin AI - The visual AI for business storytelling*. [online] Napkin AI. Available at: https://www.napkin.ai/.

[7] Eraser.io. (2025). *Untitled File*. [online] Available at: https://app.eraser.io/workspace/LMPBuj6UiRpKTW6B1BU8?origin=share [Accessed 20 Mar. 2025].