ACIT5900

MASTER THESIS

in

Applied Computer and Information Technology (ACIT)
May 2023

Cloud-Based Services and Operations

Evaluate Kubernetes for Stateful and highly available enterprise database solutions

Anurag Sharma

Department of Computer Science

Faculty of Technology, Art, and Design

OSLOMET

PREFACE

This master’s thesis represents the culmination of extensive research, rigorous analysis, and countless hours of dedication. I present this work to the academic community with great pleasure and a sense of fulfillment, as it signifies completing an important milestone in my educational journey.

This research was only possible with numerous individuals' invaluable support and guidance. First and foremost, I would like to express my deepest gratitude to my supervisor, Raju Shrestha, for his extraordinary supervision and support. His expertise, mentorship, insightful feedback, constructive criticism, and encouragement have undoubtedly shaped the outcome of this work. I also want to thank Oslo Metropolitan University and the Department of Computer Science Faculty of Technology, Art, and Design for granting me a chance to get higher studies.

I am deeply grateful to my family and friends for their unwavering support, patience, and encouragement throughout this arduous journey. Their love, understanding, and belief in my abilities have been a constant source of motivation, reminding me of the importance of perseverance and resilience in facing challenges.

Furthermore, I want to thank my parents, especially my father, who always kept pushing and encouraged me to do higher studies and my sister Preeti, my brother-in-law Sunil, and my two nephews, Kshitij and Arjun are always a part of me and a source of inspiration in many ways.

Finally, I would like to thank my wife Madhuri and son Samriddh for understanding, supporting me, and often bearing my frustration throughout these two years whenever I was too busy with my studies. It would not have been possible without their unconditional support and love.

Anurag Sharma

Oslo, May 2023

ABSTRACT

With the increasing adoption of containerization and microservices, Kubernetes has become a popular choice for managing complex distributed systems. This project evaluates the feasibility of setting up a high availability (HA) environment on Kubernetes and testing enterprise databases (PostgreSQL, MariaDB, TiDB, and MySQL), focusing on HA setups having multiple replicas for databases. The project uses Azure Kubernetes Service (AKS) and documents factors such as database deployment, performance, and scalability of the database pods to assess the behavior of the databases. Kubernetes provides an ideal platform for running stateful applications, such as databases, with its support for stateful sets and persistent volumes. However, there are many considerations when setting up HA databases on Kubernetes, such as ensuring data consistency, managing backups and failover, and handling network and storage configurations. This thesis will explore the steps in setting up HA databases on Kubernetes, including choosing a suitable database technology and creating stateful sets for databases.

The methodology involves a series of experiments on various popular databases running on Kubernetes clusters. The database throughput, latency, and resource utilization under different workloads and configurations were measured. The challenges in deploying and managing databases on Kubernetes, and comparing deployment models, such as Stateful Sets and Operators, were observed and then analyzed. Database performance benchmark tests were performed, and results indicate that Kubernetes can handle enterprise databases in HA, but each database has unique characteristics and requirements. MySQL and PostgreSQL performed better in all the categories of the benchmark tests compared to MariaDB and TiDB. The findings suggest that Kubernetes can be a viable platform for running enterprise databases but requires careful planning and configuration to achieve optimal results.

Overall, the project demonstrates the potential for using Kubernetes in enterprise database management and suggests areas for further research and improvements.

Table of Contents

1 List of Figures:

2 List of Tables:

3 Introduction

The advent of cloud computing has brought about a dramatic change in the way businesses set up and administer their infrastructure. The ability to provision and manage resources on demand is one of the most important aspects of cloud computing. This ability makes it much simpler for organizations to scale their operations to meet their specific requirements. As a result of this, Kubernetes has become widely recognized as the de facto standard for container orchestration. This has enabled developers to deploy and manage applications in a flexible and scalable manner. Kubernetes has also emerged as the platform of choice for deploying and managing highly available applications considering the growing demand for high availability (HA) and fault tolerance.

This research aims to assess the deployment process of a database in a Kubernetes high availability (HA) environment and test the performance of four enterprise databases (PostgreSQL, MySQL, MariaDB, and TiDB), and identify any challenges or limitations that may arise as a result of doing so for different databases and scenarios. The focus will primarily be HA configurations and multiple replicas for a stateful database scale set. To deploy the databases, the project will use Azure Kubernetes Service (AKS) to evaluate the behavior of the databases, and characteristics such as database performance and scalability will be documented.

Enterprise databases such as PostgreSQL, MariaDB, TiDB, and MySQL are widely used; each has advantages and disadvantages. PostgreSQL is a relational database that is dependable and adaptable, and it is frequently utilized in business environments. TiDB is an open-source and distributed database that supports HTAP (Hybrid Transactional and Analytical Processing). MySQL is a widely used open-source database well-known for its speed and user-friendliness. It is typically utilized in web applications (Tran et al., 2022). MariaDB is an open-source relational database system used in important websites like Wikipedia.

In recent years, there has been a growing interest in deploying databases on Kubernetes. This is because many organizations are developing helm charts and Kubernetes operators that boast a highly scalable and available database management platform.

Because it offers features such as automatic scaling, load balancing, and self-healing, Kubernetes is an excellent choice for deploying and managing databases because of its versatility. In addition, Kubernetes makes deploying and managing multiple database instances simple, simplifying the process of achieving high availability and fault tolerance.

Microsoft Azure offers Azure Kubernetes Service (AKS), a fully managed version of the Kubernetes container orchestration system. Because it simplifies the process of deploying and managing Kubernetes clusters, AKS is ideally suited for deploying and managing applications in a cloud-native environment. Because it offers functions such as automatic scaling, load balancing, and self-healing, AKS is an excellent choice for deploying and managing databases in an environment that must be highly available. In addition, Azure Container Service (AKS) enables integration with other Azure services like Azure Active Directory, Azure Monitor, and Azure DevOps, simplifying the process of integrating AKS with other cloud services.

According to the findings, Kubernetes can manage high availability for all four enterprise databases, but each has distinct characteristics and prerequisites (Nguyen & Kim, 2022). The project demonstrates the potential for using Kubernetes in enterprise database management. Additionally, it suggests areas for further research and improvements.

3.1 Problem Statement

Can Kubernetes handle enterprise databases with HA, precisely a set of HA setups? The project intends to observe and document various factors, such as database performance, deployment, scalability, I/O, transaction per second, latency, etc., to assess the behavior of the databases in this set of environments.

3.2 Research Questions

Some research questions that arise and are the motivation of this thesis are:

1. Which database performs the best among the alternatives on a high availability setup on Kubernetes on the benchmarking parameters?

2. In a HA configuration of Kubernetes, how does the consistency of enterprise databases compare to other database types?

3. In a setting where high availability is required, how does the scalability of corporate databases compare to that of Kubernetes?

4. What are the best methods for installing and maintaining enterprise databases on Kubernetes for high availability?

3.3 Scope

The scope of testing an enterprise database on Kubernetes for this research paper encompasses several vital aspects. The objective is to evaluate the database performance, scalability, reliability, and efficiency of deploying and managing a database system on the Kubernetes container orchestration platform. Many areas could have been tested in this research, but they needed to be excluded due to time constraints. A few of them are discussed here.

High Availability and Fault Tolerance: Kubernetes offers mechanisms for ensuring high availability and fault tolerance. Evaluate the database's resilience against container failures, node failures, and network disruptions. Assess Kubernetes features like pod rescheduling, node redundancy, and persistent storage solutions for data durability.

Security and Data Protection: Address security considerations related to running an enterprise database on Kubernetes that includes evaluating the security features provided by Kubernetes, such as network policies, secrets management, and access controls. Additionally, explore data backup and recovery mechanisms, ensuring the integrity and confidentiality of the database.

Operational Efficiency: Managing an enterprise database on Kubernetes involves administrative tasks such as upgrades, monitoring, and logging. How efficiently can these operations be performed using Kubernetes tools and integrations?

The following areas are intended to be covered within the scope of this research:

Database Deployment: The research aims to focus on deploying MySQL, PostgreSQL, TiDB, and MariaDB on a Kubernetes cluster on Azure. The deployment process, including containerization, configuration, and provisioning of the database will be thoroughly examined.

Performance Evaluation: The performance of the enterprise database on Kubernetes will be assessed under different scenarios. This includes benchmarking the database's read and write operations, assessing response times, and measuring throughput, latency, memory, and CPU utilization of the Kubernetes cluster.

Scalability: Scalable systems are highly in demand today. A highly scalable system should be able to scale up and down, manually, or automatically, depending upon specific criteria set. The scalability of pods will be tested for all four databases, and observations will be recorded.

3.4 Approach

Evaluating enterprise databases in a Kubernetes High Availability (HA) environment involves several considerations to ensure the chosen solution meets your organization's requirements. Here is the approach that is preferred to choose for evaluating MySQL, TiDB, MariaDB, and PostgreSQL enterprise databases in a Kubernetes HA environment:

Define Evaluation Criteria:

Database deployment: Measure and compare the installation process of various databases under the scope of this research.

Performance: Measure the database's speed and latency under different workloads.

Scalability: Observe how efficient the scale-up and scale-down process is for all the databases.

Select Four Enterprise Databases:

Choose popular enterprise databases known for their suitability in Kubernetes environments. Examples include Oracle, MSSQL, MySQL, PostgreSQL, TiDB, MongoDB, MariaDB, DB2, Sybase etc.

Set Up Kubernetes HA Environment:

Establish a Kubernetes HA environment using a tool like Kubernetes clusters or managed Kubernetes services such as Amazon EKS, Google Kubernetes Engine (GKE), or Azure Kubernetes Service (AKS). Ensure the environment is appropriately configured for high availability, with redundant nodes, network connectivity, and storage.

Prepare Test Workloads:

Define representative workloads that mimic your organization's database usage patterns. This may include read-heavy, write-heavy, or mixed workloads. Consider the size of the data set, the number of concurrent users, and specific operations required.

Deploy Databases:

Deploy each of the four selected databases within the Kubernetes HA environment using appropriate Helm charts, Kubernetes manifests, or operators. Ensure the databases are configured with recommended settings for high availability, replication, and fault tolerance.

Performance Testing:

Execute the predefined workloads against each database and measure performance metrics such as response time, throughput, and resource utilization. Monitor the database's behavior under different load scenarios, including peak usage periods.

Scaling:

Perform scale-up and scale-down operations for each database scale set. Increase the number of replicas to scale up and decrease the number of replicas to scale down and observe the efficiency, problems, and turnaround time for them.

Analyze Results:

Based on the evaluation criteria, compare each database's performance, high availability, scalability, and management capabilities. Finally, select the database that performs best on the benchmarking criteria.

Remember to document the evaluation process, including the test scenarios, results, and observations. This documentation can serve as a valuable resource for future reference and decision-making.

4 Background

4.1 Containerization

Containerization technology is a powerful approach that has revolutionized how software applications are developed, deployed, and managed. It provides a lightweight and efficient method to package software, and all its dependencies and libraries, into a portable and isolated unit called a container. These containers can be run consistently across different computing environments, making developing, testing, and deploying applications across diverse systems easier.

At the heart of containerization technology is the concept of a container, which encapsulates an application and its dependencies, including the operating system, into a single self-contained unit. This eliminates the need for complex and time-consuming installation and configuration processes, as containers can be deployed with all the required components already packaged within them. Containers are designed to be highly portable and can run on any host system that supports containerization, making them platform-independent and reducing compatibility issues (Docker, 2021).

One of the critical benefits of containerization is its ability to provide isolation between applications and their underlying infrastructure. Each container operates in its isolated environment, separate from other containers and the host system, ensuring that applications run consistently and predictably without interference from other processes. This isolation also enhances security, as vulnerabilities or breaches in one container are contained and isolated, minimizing the impact on other containers or the host system.

Despite the various advantages, containerization technology also presents challenges. Managing container lifecycles, orchestrating multiple containers, and ensuring security and compliance within containerized environments require specialized knowledge and tools. However, the vibrant ecosystem around containerization has led to the development of numerous management platforms, monitoring tools, and security solutions to address these challenges.

Container registries have also evolved, enabling efficient distribution and management of container images. Platforms like Docker Hub, Google Container Registry, and Amazon

Elastic Container Registry offer seamless image hosting, versioning, and integration with continuous integration/continuous deployment (CI/CD) pipelines.

Additionally, advancements in container networking have allowed for more flexible and scalable architectures. Technologies like overlay networks, service meshes, and container-native load balancing enable seamless communication between containers, both within and across different clusters, improving overall network performance and security.

The advancement of container technology has also facilitated the rise of serverless computing. Containers have become the underlying technology for serverless platforms, providing the agility and scalability required for event-driven, on-demand application execution.

Container technology, including orchestration platforms, runtimes, registries, networking, and their integration with serverless computing, have transformed the way applications are developed, deployed, and managed. These advancements have accelerated software delivery, improved resource utilization, and enhanced the overall efficiency and agility of modern software development practices.

Docker, an open-source project, introduced a user-friendly interface and toolset for creating, distributing, and running containers. It standardized the container format and introduced a layered approach to container images, enabling efficient sharing of common components and reducing the size of containerized applications. Docker made it significantly easier to package applications with their dependencies, ensuring consistency across different environments and eliminating the "it works on my machine" problem.

The success of Docker spurred the development of container orchestration platforms, aiming to simplify the management of containerized applications across clusters of machines. One such platform that emerged as a dominant force was Kubernetes, which was initially developed by Google.

4.2 Kubernetes

Kubernetes is a container orchestration platform developed by Google in 2014 and then made available to the public as an open source (Figueiredo & Subratie, 2020). Kubernetes is a platform that was created to automate the deployment, scaling, and management of containerized applications. Kubernetes offers a wide variety of capabilities that promote high availability, such as the following (Khalel et al., 2022):

1. **Self-healing:** Kubernetes is built to detect and recover from any failures that may occur automatically. If a container or node experiences an error, Kubernetes can automatically restart or migrate it to another node.

2. **Load balancing:** Kubernetes includes capabilities for built-in load balancing, which can distribute traffic across multiple containers or nodes.

3. **Scaling:** Kubernetes can automatically scale applications up or down depending on the demand. This can improve performance while simultaneously lowering the risk of overloading any individual node.

4. **Rolling updates:** Kubernetes can perform a rolling update, which means that applications can be updated without experiencing any downtime. Kubernetes performs updates on one node at a time during a rolling update while the other nodes continue to process traffic in the background.

5. **Replication:** Kubernetes' support for replicating containers and nodes can increase overall performance and ensure the system is always available.

It is intended for Kubernetes to be highly modular and extensible, and a wide variety of plugins and add-ons can be utilized to expand its capabilities. Kubernetes is becoming increasingly popular for managing containerized applications, and many organizations are adopting Kubernetes as their primary platform for deploying and managing applications due to this popularity.

4.2.1 Architecture

Figure 4.1 shows an architecture diagram of Kubernetes that mainly contains control plane and node components. The components (control plane and node) further consist of many small working units mentioned here.

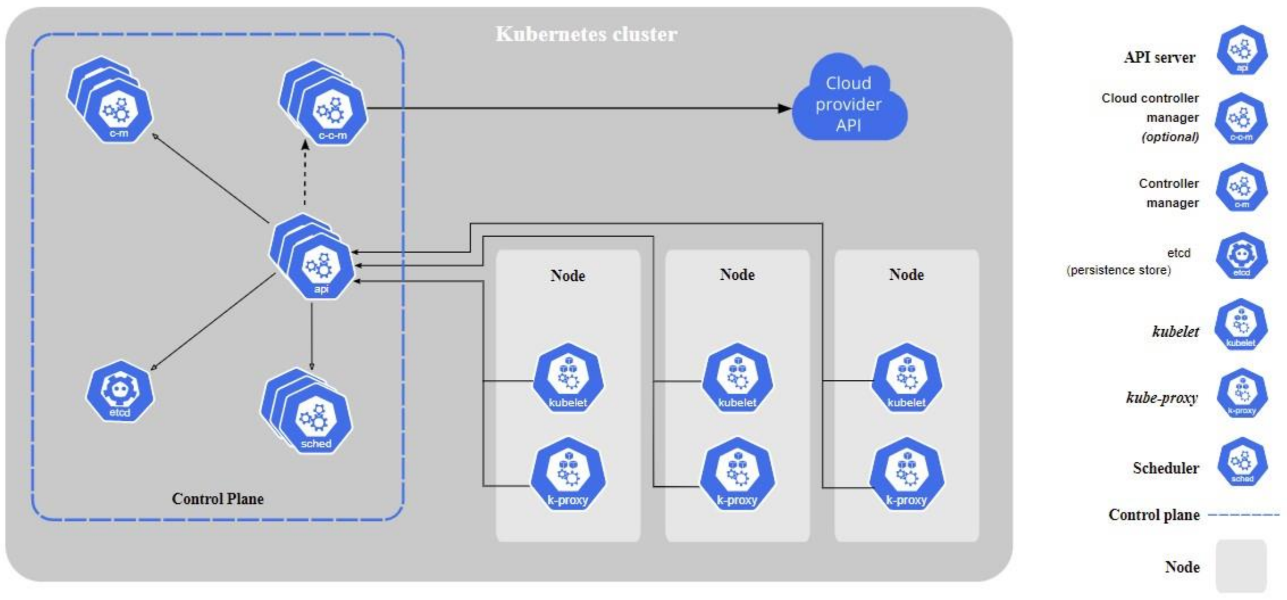


Figure 4.1: Kubernetes architecture diagram (Kubernetes, 2021a).

Control Plane and Its Components (**Kubernetes, 2021a**):

The primary function of the control plane is to manage nodes and pods in the Kubernetes cluster. It handles scheduling resources and decisions when responding to events across the cluster. Some of the components are optional.

- **Episerver**: It is a central management utility that receives all the requests and acts as a front end for the Kubernetes cluster.
- **etc**: It is a key-value store to store configuration data such as namespace and configuration and is accessible only from the API server.

- **Controller-Manager**: It runs processes in the background, regulates the cluster's shared state, and performs routine tasks. If you make changes in the configuration files, the control manager instructs other components to adapt those changes.
- **Scheduler**: This component assigns resources to the nodes.

Node Components (Kubernetes, 2021a)

- **Kubelet**: It is the leading service that takes instructions from the API server so that all the containers are running in the desired state.
- **Kube-Proxy**: It deals with networking and talks to the outside world.
- **Pods**: The smallest entity in the cluster. Pods are a collection of containers with one or more containers to execute workloads.

The following components make up the Kubernetes architecture, which is based on a master-worker model:

1. **Master**: The Kubernetes master is the component of the system that is in charge of managing the cluster and all of its resources. It consists of a number of different components, including the API server, etc, the controller manager, and the scheduler.

2. **Nodes**: Nodes, also known as worker machines, are the computers that are responsible for running containerized applications. Each node is responsible for running a container runtime, such as Docker, and maintaining communication with the master in order to obtain instructions and receive updates.

3. **Containers**, also known as Pods, are the smallest units that can be deployed using Kubernetes. One or more containers that share the same storage volume and network namespace make up a pod. Pods can contain multiple containers.

4. **Services**: Services are what are used to give pods access to their respective networks. A pod may be made accessible to an internal network or the internet through a service, which may also offer load balancing and failover capabilities.

5. **Controllers**: Controllers are utilized to manage and monitor pods and services and are responsible for their overall operation. Controllers have the ability to carry out a variety of tasks, including scaling up or down pods, rolling out updates, and monitoring pod health.

Kubernetes orchestration is a powerful and widely used container orchestration platform that simplifies the deployment, management, and scaling of containerized applications. It provides a robust framework for automating various aspects of application lifecycle management, such as container scheduling, scaling, load balancing, and service discovery. At its core, Kubernetes is designed to manage and coordinate clusters of containerized applications. It leverages a declarative approach, where users define the desired state of the applications and Kubernetes takes care of maintaining that desired state. This means that instead of manually interacting with individual containers, developers can specify the requirements and dependencies of their applications in the form of Kubernetes objects, such as pods, services, deployments, and config maps.

One of the key features of Kubernetes orchestration is its ability to distribute containers across a cluster of machines. Kubernetes uses a scheduler component that intelligently assigns containers to nodes based on resource availability, workload constraints, and other user-defined policies. This ensures efficient utilization of resources and high availability of applications.

Kubernetes also provides a flexible and scalable networking model. Each container in a Kubernetes cluster is assigned a unique IP address, and containers within the cluster can communicate with each other directly using these IP addresses. Kubernetes automatically handles networking setup and routing, making it easier to build complex, microservices-based architectures.

Kubernetes is highly extensible and customizable. It offers a rich set of APIs, allowing developers to interact with and control the platform programmatically. Kubernetes also supports a wide range of extensions and plugins, enabling you to integrate it with various ecosystem tools and services.

Overall, Kubernetes orchestration provides a robust and scalable framework for managing containerized applications. By automating various aspects of application lifecycle management, it simplifies the deployment and operation of complex distributed systems. Whether you are running a small-scale application or managing a large-scale production environment, Kubernetes offers the tools and capabilities to effectively orchestrate your containers and ensure the smooth operation of your applications.

4.2.2 High availability (HA) environments and Kubernetes

The idea of having a system with high availability has been around since the beginning of computing, back when hardware malfunctions were an everyday occurrence. To reduce the likelihood of experiencing downtime, many of the earliest mainframe computers were built with redundant components. These included multiple processors, power supplies, and storage devices (Shamim et al., 2020). As the complexity of computer systems increased, so did the requirement for high availability; consequently, new technologies were developed to meet this demand. The following are some of the most critical advances in high availability (Hu & Wang, 2021):

1. **Clustering:** Clustering is the practice of connecting multiple servers so that they can operate as if they were part of a single system. If you have multiple servers working together in a cluster, they can share the load if one goes down. The high availability of individual components of both software and hardware can be provided by clustering.

2. **Failover:** The process of automatically switching to a backup system in the event that the primary system experiences an outage is referred to as failover. There are multiple levels at which failover can be implemented, including the hardware, software, and network levels.

3. **Load balancing:** Load balancing is the process of distributing workloads across multiple servers to improve performance and reduce the likelihood of any one server becoming overloaded. The ability to provide redundancy and failover is another benefit that can be gained from load balancing.

4. **Replication:** The process of creating copies of data or applications and then distributing those copies across a number of servers is known as replication. The availability of data and applications can be increased through replication, which also helps improve performance and scalability.

5. **Virtualization:** Running multiple virtual machines (VMs) on a single physical server is what virtualization entails. High availability can be achieved through virtualization by enabling the migration of virtual machines (VMs) to other servers in the event that the original server experiences a hardware failure.

High availability is an essential factor to consider for any information technology system, but it is of the utmost significance for mission-critical applications. Some examples of mission-critical applications include financial systems, healthcare applications, and emergency response systems. Downtime is cut to a minimum, and data is less likely to be lost when working in an environment with high availability. A system's performance, scalability, and dependability may all see improvements as a result of increased availability.

Therefore, high availability is a requirement that must be met by modern information technology systems, especially those responsible for supporting mission-critical applications. Even in the event of hardware or software failures, network outages, or other types of disruptions, users can still access the applications and data stored in high-availability environments. Kubernetes is a widely used platform for the management of containerized applications. Utilizing Kubernetes in conjunction with high-availability environments can deliver the required levels of dependability and scalability to organizations to support their mission-critical applications.

A company's success in today's fast-paced digital environment needs access to a wide variety of applications and services. A corporation risks suffering a loss of income, the confidence of its customers, and its brand whenever it has downtime. Therefore, ensuring high availability (HA) and fault-tolerance of applications and services is paramount.

A high-availability environment is one that is intended to guarantee that applications and services will be accessible at all times, regardless of whether the underlying hardware or software is functioning correctly (Todorov, 2022). In most cases, it includes using redundant hardware, software, and network components to eliminate the possibility of a single point of failure. In addition, it often contains automated failover methods that are able to identify and recover from failures in real-time, which ensures that the application or service will only have a limited number of low-availability scenarios (Gokhale et al., 2021).

Kubernetes is a free and open-source container orchestration technology that gives users the ability to build and manage applications on a platform that is both scalable and highly available. Kubernetes is a great platform for deploying and managing applications in a cloud-native environment because it offers capabilities such as autonomous scaling, load balancing, and self-healing. Kubernetes ensures that there is not a single point of failure by enabling several nodes to execute the same application simultaneously. This allows for high availability to be achieved. In addition, Kubernetes comes with automated failover features that can identify and recover from failures in real-time. This guarantees that the application will have the least amount of downtime possible.

Kubernetes is able to provide high availability because of its architecture and the design principles it adheres to. The design of Kubernetes is built on a concept known as "master-slave," in which the "master" node is responsible for managing the general state of the cluster, and "worker" nodes are responsible for carrying out the tasks. While the worker nodes are in charge of carrying out the workloads, the master node is in charge of managing the deployment of the application as well as scaling it and monitoring it.

Kubernetes eliminates the possibility of there being a single point of failure by allowing the deployment of numerous master nodes. This enables the system to achieve high availability. The master nodes interact with one another using a quorum-based method, which means that before any modifications can be performed, the majority of the nodes must first reach a consensus on the current state of the cluster (Liu et al., 2020). In addition, Kubernetes comes with automated failover features that can identify and recover from failures in real-time scenarios. This guarantees that the application will have the least amount of downtime possible (Kim et al., 2021).

Kubernetes offers additional features, including autonomous scaling, load balancing, and self-healing, all of which contribute to the platform's already impressive high availability capabilities. Kubernetes' automatic scaling feature enables the application to be dynamically scaled up or down depending on demand, ensuring that there are always sufficient resources available to manage the workload. The process of load balancing ensures that the burden is dispersed in an equitable manner among all the worker nodes, hence preventing any one node from getting overwhelmed with work. Kubernetes's ability to identify and recover from problems in real time is made possible by its self-healing features. This guarantees that the application will have high availability for real-time scenarios.

A system is said to have high availability if it can continue to function normally even in the event that one or more of its components become dysfunctional. Even in the event of hardware or software failures, network outages, or other types of disruptions, the users of the system should still be able to access the applications and data that they require. This is the goal of high availability (HA). Kubernetes is an increasingly popular platform for managing containerized applications. It offers a variety of features that support high availability, making it an attractive option. This report investigates the background of high-availability environments as well as Kubernetes, including their history, architecture, and key features, in this report.

4.3 StatefulSets

Kubernetes StatefulSets is a powerful feature within the Kubernetes ecosystem that allows you to manage stateful applications. While Kubernetes is well-suited for stateless workloads, StatefulSets provide a mechanism for running and scaling stateful applications with stable network identities and stable storage.

In Kubernetes, stateless applications can be easily scaled and managed using the deployment abstraction. However, stateful applications have additional requirements, such as stable network identities, consistent storage, and ordered deployment and scaling. StatefulSets address these requirements by providing a higher-level abstraction that ensures the reliable deployment and scaling of stateful applications.

The scaling of StatefulSets in Kubernetes is an interesting and important topic. StatefulSets are designed to manage stateful applications in a distributed system, where each instance of the application has a unique identity and maintains its own state. Scaling StatefulSets involves dynamically adjusting the number of instances based on workload demands while ensuring data consistency and maintaining application availability.

A StatefulSet is a workload API object in Kubernetes that is responsible for managing a set of stateful pods. Each pod within a StatefulSet has a unique ordinal index and a stable hostname that is based on the name of the StatefulSet and the ordinal index of the pod. This stable network identity enables applications to rely on consistent and predictable network addresses, which is crucial for stateful applications that may rely on peer-to-peer communication or data replication (**Kubernetes, 2021b**).

StatefulSets also provide a mechanism for managing the persistent storage associated with stateful applications. Each pod within a StatefulSet can have its own persistent volume claim, ensuring that data is persisted across pod restarts and rescheduling. This allows stateful applications to maintain their state even when the underlying pods are terminated or recreated (**Kubernetes, 2021b**).

In summary, Kubernetes StatefulSets are a valuable resource for running and managing stateful applications in a Kubernetes cluster. They provide stable network identities, persistent storage, ordered deployment and scaling, and automated updates. With StatefulSets, you can confidently run stateful workloads in Kubernetes, taking advantage of the scalability and resilience of the platform while maintaining the integrity of your data and application state.

4.4 Kubernetes Operators

Kubernetes operators are software extensions that automate the management and operation of applications and services within a Kubernetes cluster (Kubernetes, 2021c). They are designed to manage complex, stateful applications that require more than just basic deployment and scaling capabilities provided by Kubernetes.

Operators leverage the Kubernetes resource model and controller framework to define, deploy, and manage custom resources and controllers specific to a particular application or service. They encapsulate the operational knowledge and best practices for running a specific application, making it easier to manage and operate complex workloads.

Here are some key characteristics and capabilities of Kubernetes operators:

Custom Resources: Operators introduce custom resources that extend the Kubernetes API. These resources represent higher-level abstractions specific to an application, such as databases, message queues, or clusters.

Controllers: Operators include custom controllers that watch and reconcile the desired state of custom resources with the actual state of the cluster. They respond to changes in the resources and perform actions to ensure the desired state is achieved.

Automation: Operators automate various aspects of application management, including deployment, configuration, scaling, upgrades, backups, and more. They handle complex tasks that would otherwise require manual intervention or scripting.

Lifecycle Management: Operators manage the entire lifecycle of an application, from initial deployment to scaling, self-healing, and eventual decommissioning.

Domain-Specific Knowledge: Operators encapsulate domain-specific knowledge about an application or service. They understand the intricacies and requirements of the application and can make intelligent decisions based on that knowledge.

Ecosystem: Kubernetes operators have a growing ecosystem, with many operators available for popular applications and services, including databases (e.g., PostgreSQL, MySQL), message brokers (e.g., Kafka), monitoring systems, and more. These operators are typically developed and maintained by the application's creators or third-party vendors.

By leveraging operators, Kubernetes users can simplify the management of complex applications, reduce manual operations, and ensure consistent and reliable deployments. They promote standardization, automation, and best practices for running specific workloads within a Kubernetes environment.

4.5 Databases

A relational database stores data into tables, which consist of rows and columns. The RDBMS provides a set of tools and functions to store, retrieve, and manipulate data in these tables. RDBMS stands for Relational Database Management System. It is a software system that enables the creation, management, and maintenance of relational databases.

In an RDBMS, data is stored in a structured manner, and relationships between tables are defined using keys like primary and foreign keys. RDBMSs enforce data integrity by allowing the definition of constraints, such as unique values, referential integrity, and data types.

Some popular examples of RDBMSs include Oracle Database, MySQL, Microsoft SQL Server, PostgreSQL, and IBM DB2. These systems provide a wide range of features for managing large volumes of data, ensuring data consistency, supporting concurrent access, and optimizing query performance.

RDBMSs have been widely adopted in various industries and applications, including business systems, e-commerce websites, banking systems, human resources management, and more. They provide a reliable and efficient means of storing and retrieving structured data, making them a cornerstone of modern data management.

This section shows the databases used for performing the test in the report. Databases are MySQL, TiDB, MariaDB and PostgreSQL. All these four databases are relational database management systems. MySQL and PostgreSQL are quite old databases, while TiDB is fairly new as compared to them. In the relational rdbms, data is stored in the form of rows and columns. More about these databases is mentioned in the following sections.

4.5.1 MySQL

MySQL is an open-source relational database management system (RDBMS) and is used for managing and organizing structured data. It is owned by Oracle Corporation.
MySQL uses Structured Query Language (SQL) as its primary language for interacting with databases. SQL is a standard language for managing relational databases, and it allows users to create, modify, and query databases using a set of predefined commands.
Some key features of MySQL include:

1. Relational database management: MySQL allows you to create and manage relational databases, which organize data into tables with predefined schemas.
2. Data integrity and security: MySQL provides mechanisms to enforce data integrity rules, such as primary keys, foreign keys, and constraints. It also offers user authentication and access control to protect the data stored in the database.
3. Connectivity: MySQL is available for multiple platforms, including Windows, macOS, Linux, and various UNIX systems. This makes it widely accessible and compatible with different operating environments (MySQL, 2021a).

4. Extensibility: MySQL allows users to extend its functionality through the use of plugins and stored procedures. Java, PHP, and Python are a few multiple programming languages supported by it (MySQL, 2021a).

MySQL is most used in web applications, content management systems, data warehousing, and other scenarios where structured data storage and retrieval are required. It has a large and active community of users and developers, contributing to its ongoing development and support.

4.5.1.1 Kubernetes Operator for MySQL

There are several ways to deploy MySQL database to Kubernetes, but using the operator seems the best bit to ease out the complexity of installation and management. They automate tasks such as deployment, scaling, and failover of the application. Operators are Kubernetes extensions that provide custom resources and controllers to manage complex applications.

Many Kubernetes operators are available for managing MySQL databases, and a few popular operators are MySQL operator, Percona operator for MySQL, and Presslabs MySQL Operator.

These operators provide various features and capabilities for managing MySQL databases on Kubernetes. One can choose the one that best fits the requirements and integrate it into your Kubernetes cluster to simplify the management of your MySQL databases. For the experiment, Kubernetes Operator for MySQL InnoDB Cluster seemed like the perfect choice.

The MySQL operator for Kubernetes manages MySQL InnoDB cluster setup inside a Kubernetes cluster and manages operations like upgrades and backups (MySQL, 2021b). MySQL can be installed on Kubernetes with either kubectl commands or helm charts. Helm is the package manager for Kubernetes and bundles various Kubernetes resources into a single deployable file. A highly detailed architecture diagram is shown in figure 4.2 below.

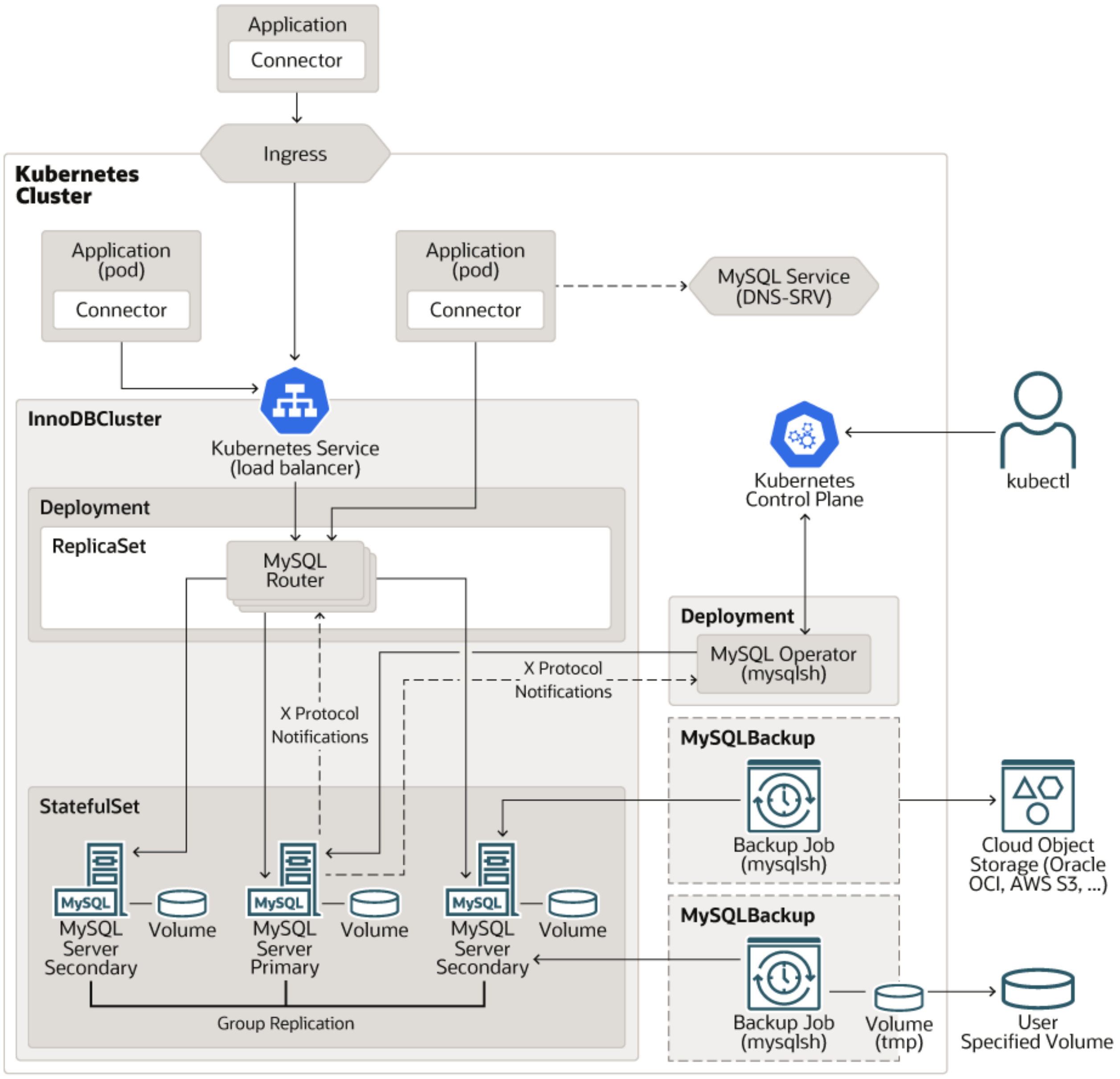


Figure 4.2: MySQL Operator for Kubernetes Architecture Diagram (MySQL, 2021c)

4.5.2 TiDB

TiDB is an open-source distributed database system that provides high availability, scalability, and strong consistency for managing large-scale online transaction processing (OLTP) and online analytical processing (OLAP) workloads. Developed by PingCAP, TiDB is designed to address the challenges faced by modern applications that require real-time data processing and analytics (TiDB, 2021a). Figure 3 shows the various components of TiDB architecture.

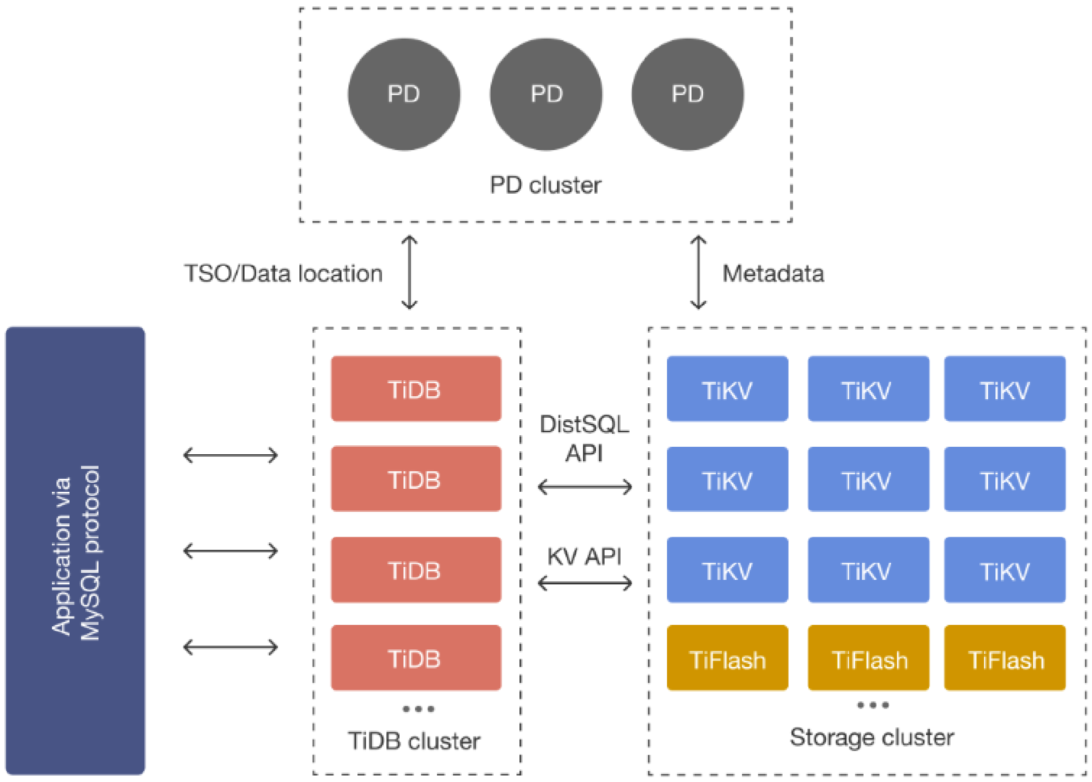


Figure 4.3: TiDB Architecture Diagram (PostgreSQL, 2021c)

It consists of four main components: TiDB Server, TiKV, and Placement Driver (PD) and TiFlash (TiDB, 2021a).

TiDB Server: TiDB Server acts as the SQL layer and handles client connections, SQL parsing, query optimization, and execution. It supports the MySQL protocol, which makes it compatible with existing MySQL applications and tools. TiDB Server provides a distributed execution engine to process queries and parallelize operations across multiple nodes in the cluster.

TiKV: TiKV is the distributed key-value storage engine used by TiDB. It is designed to provide horizontal scalability and fault tolerance. Each TiKV node stores a range of key-value pairs and supports transactions. Data is automatically sharded and distributed across multiple TiKV nodes based on the key range.

Placement Driver (PD): PD is responsible for cluster metadata management, including the distribution and scheduling of data across TiKV nodes. PD keeps track of the cluster topology, monitors node health, and handles data rebalancing and leader election. It provides information to TiDB Server and TiKV about the location and status of data.

TiFlash: TiFlash is a columnar storage engine that is integrated with TiDB. It is specifically optimized for analytical queries and provides significant performance improvements for analytical workloads.

Overall, TiDB's architecture combines distributed transaction processing, distributed data storage, and automatic data management to provide a scalable, highly available, and consistent database solution for modern applications.

One of the key features of TiDB is its ability to distribute data across multiple nodes, allowing for horizontal scalability. It adopts a shared-nothing architecture, where data is partitioned and stored on different nodes, ensuring high availability and fault tolerance.

TiDB is built to support SQL, making it compatible with a wide range of applications and developer tools. It provides support for various SQL features, including ACID transactions, joins, indexes, and aggregations. Additionally, TiDB supports distributed transactions, allowing users to perform transactions that involve multiple nodes and maintain consistency across them.

4.5.2.1 TiDB Operator for Kubernetes

TiDB Operator is developed by PingCAP for some public cloud environments such as AWS, Azure, GCP and Alibaba. This operator assumes that the following prerequisites must be met before deployment (**TiDB, 2021c**).

- Kubernetes >= v1.12
- DNS add-ons
- PersistentVolume
- RBAC enabled (optional)

- Helm 3

TiDB Operator uses Persistent Volumes to persist the data of TiDB cluster (including the database, monitoring data, and backup data), so the Kubernetes cluster must provide at least one kind of persistent volume. It is recommended to enable RBAC in the Kubernetes cluster (TiDB, 2021c).

4.5.3 PostgreSQL

PostgreSQL, often referred to as "Postgres," is also an open-source, relational database management system (RDBMS) that provides a robust and scalable platform for storing and managing data (PingCAP, 2021). PostgreSQL is a relational database system that organizes data into tables with rows and columns, allowing for efficient storage and retrieval of structured data. It follows the principles of ACID (Atomicity, Consistency, Isolation, and Durability) to ensure data integrity and reliability. It also provides transactional support to maintain consistency and reliability even in the presence of concurrent operations and system failures. Users can define their own data types, operators, and functions, which makes it highly extensible. This feature enables developers to create custom data structures and operations tailored to their specific application needs (PostgreSQL, 2021a).

PostgreSQL supports a wide range of SQL standards and offers advanced querying capabilities. It includes support for complex queries, subqueries, joins, window functions, and more. Additionally, PostgreSQL provides full-text search, geospatial querying, and support for JSON and XML data types. Through the multi-Version Concurrency Control (MVCC) feature, it can handle concurrent database access. This allows multiple transactions to work independently without blocking each other, resulting in better performance and scalability.

Having a vibrant and active community of developers and users worldwide attracts users to try PostgreSQL because the community-driven development model ensures frequent updates, bug fixes, and new features. PostgreSQL is widely used in various applications ranging from small-scale projects to large enterprise systems, including web applications, geospatial applications, data warehousing, scientific research, and more.

4.5.3.1 PostgreSQL on Kubernetes

Helm chart developed by bitnami sets up a PostgreSQL high-availability deployment on a Kubernetes cluster using the Helm package manager. The prerequisites for installation are the following (PostgreSQL, 2021b):

- Kubernetes 1.19+
- Helm 3.2.0+
- PV provisioner support in the underlying infrastructure

The image provided by bitnami does the non-root installation by default and follows the security best practices, which make it easier to set up on systems having very strict security constraints (PostgreSQL, 2021b).

4.5.4 MariaDB

MariaDB is an open-source relational database management system (RDBMS) that is a fork of MySQL (MariaDB, 2021c). The development of MariaDB is led by the original developers of MySQL. MariaDB retains many of the features and syntax of MySQL, making it compatible with MySQL applications and libraries. It offers high performance, scalability, and reliability for managing structured data.

One of the significant advantages of MariaDB over MySQL is its commitment to open-source development and community collaboration. It has a more rapid release cycle and is known for incorporating features and improvements faster than MySQL. MariaDB also includes additional features like Galera Cluster for synchronous multi-master replication and support for NoSQL databases.

Many popular websites, applications, and organizations use MariaDB, including Wikipedia, WordPress, and Google.

4.5.4.1 MariaDB Galera Cluster on Kubernetes

MariaDB Galera Cluster is a high-availability database clustering solution based on synchronous multi-master replication. It allows us to create a cluster of MariaDB database servers that are tightly interconnected and work together to provide high availability and fault tolerance.

Galera Cluster uses the Galera replication library, which enables synchronous replication of database transactions across all nodes in the cluster. This means that every write operation to one node is automatically replicated to all other nodes in the cluster, ensuring that all nodes have the same data at any given time.

To set up the MariaDB cluster on Kubernetes, a helm chart created by bitnami was used, and the prerequisites for installation are as follows (MariaDB, 2021b).

- Kubernetes 1.10+
- Helm 3.2.0+
- PV provisioner support in the underlying infrastructure

MariaDB cluster was created by executing the helm command when connected inside the AKS from the command line tool WSL.

4.6 Workload Generator

Sysbench is a popular open-source benchmarking tool used for evaluating the performance of computer systems. It provides a set of modular, configurable tests that measure various aspects of system performance, such as CPU, memory, file I/O, and database performance (TiDB, 2021d).

Originally designed for benchmarking MySQL database systems, over a period, sysbench has evolved to support a wider range of benchmarking scenarios, including CPU, memory, disk I/O, mutex contention, and multi-threaded workload simulation. It can be used to assess the scalability and stability of hardware components, operating systems, virtualization platforms, and database management systems.

Sysbench is written in the programming language Lua and supports multiple database backends, including MySQL, MariaDB, PostgreSQL, SQLite, and others. It utilizes a command-line interface (CLI) and offers a rich set of command-line options to customize the benchmarking parameters according to specific requirements.

The tool generates synthetic workloads that simulate real-world scenarios and measures various performance metrics, such as transactions per second, latency, throughput, and system resource utilization. By running different benchmarking tests, system administrators, developers, and performance engineers can gain insights into system performance, identify bottlenecks, and make informed decisions regarding system tuning and optimization.

Overall, Sysbench is a versatile benchmarking tool that provides a standardized and reproducible way to evaluate and compare the performance of computer systems across different hardware and software configurations (TiDB, 2021d).

4.7 Windows Subsystem for Linux (WSL)

WSL configuration is required to connect to the Kubernetes cluster from the command line. The benefit of using the Windows subsystem for Linux is that it allows running a Linux/Unix command-line environment on the Windows operating system without installing any other virtual machine on top of it. WSL can be used as a client to execute Kubernetes commands and generate the load from the sysbench.

5 Literature Review

5.1 Kubernetes in microservices:

The previous few decades have seen a gradual shift in programming language and framework evolution regarding distribution, a modular approach, and loose coupling with the aim of maximizing code reuse and robustness (Tran et al., 2022). This imperative has been driven by the requirement to raise software quality across the board, including less specialized off-the-shelf software and applications that are crucial for protection along with the economy. The use of cloud computing facilities has significantly increased interest in microservices architectural design in the software engineering community (Nguyen & Kim, 2022). Technically speaking, microservices should be functionally different parts that are implemented in isolation and furnished with specific memory preservation mechanisms (like databases). Although every component of the architecture of microservices is a microservice, the composition and coordination of its parts via communications are what give the design its distinctive behaviour (Todorov, 2022).

The development of service-oriented architectures (SOAs), which combine the requirements for interoperability between heterogeneous information systems that may be owned by different businesses with the requirement for code recycling and resilience, might be considered a step in this course of action (Gokhale et al., 2021). The conception of an offering as an entity that software engages with other software entities through message-passing interactions using established data standards and conventions (such XML, SOAP, and HTTP) and clearly described representations was raised as a result. A further evolution on this path is the concept of microservices, which emphasizes the use of compact services—they are, after all, termed microservices—and moves service-oriented methodologies from system integration to system conception, creation, including deployment (Tran et al., 2022).

The capacity, minimalism, and cohesion call for the best use of microservice architecture. To benefit from the scalability, many businesses are switching from monolithic architectures to microservices today. A good instance of this is Netflix, which was a pioneer in the transition from monolith to micro services (Liu et al., 2020). Now, Netflix's microservice architecture allows them to scale efficiently and provide daily assistance to millions of consumers. Netflix employed universality not only to facilitate deploying and localization but also to automate the process: the installation tool understood how to install an enclosure and could do it regardless of what was contained therein. By introducing a service called Chaos Monkey to continuously check for errors in the system, Netflix was also able to increase resilience as well as accessibility thanks to the microservice architecture (Tran et al., 2022).

The difficulties of developing applications that use the cloud that make use of the chances that are provided by the public cloud environment are addressed by the microservices-based architecture (Kim et al., 2021). The customer-focused architectural style of creating software made up of little services that can be autonomously delivered and maintained is realized in microservices. Each microservice runs as an autonomous process, has a unique set of business functions, and uses simple protocols for communication (Tran et al., 2022). Utilizing technologies that are in line with the features of this architectural style is necessary to make use of the advantages of microservice-based architectures. A technology called containerization makes it possible to virtualize operating systems (Shamim et al., 2020).

An open-source framework for container orchestration called Kubernetes makes it possible to set up, scale, and supervise containerized software automatically (Hu & Wang, 2021). The most popular platform for orchestrating containerized applications is Kubernetes. Virtual or physical machines can be used to make up a Kubernetes cluster, which has a master-slave design. Kubernetes reduces the difficulty of establishing program resilience through its monitoring and auto-healing methods. The top orchestration platform for containerized microservice-based applications is now Kubernetes. Companies utilize microservices as a design approach to move their legacy systems to cloud-native architectures (Figueiredo & Subratie, 2020).

Kubernetes is primarily used to containerize along with orchestrate these microservice-based systems. Microservices and containers' attributes, such as their small size and lightweight, automatically improve the accessibility of services (Todorov, 2022). By rebooting the failing containers and substituting or postponing them when their hosts fail, Kubernetes fixes its administered microservices (Todorov, 2022). Additionally, until the dangerous instances are once again prepared to train, Kubernetes does not broadcast them in order to guarantee access to only healthy containers. However, carrier-grade providers of services may not find these precautions enough, and they continue to be concerned about service availability as a crucial non-functional need (Nguyen & Kim, 2022).

5.2 Kubernetes as a High Availability (HA) System:

High availability (HA) is the process of removing single points of failure from a program so that it may continue to run even if one of the servers or other IT components on which it needs fails (Khalel et al., 2022). In simple terms, when an entire system element is unsuccessful, it may not immediately result in the discontinuation of whatever service that component was providing. HA systems are fault-resistant structures with no single point of failure. Eliminating single points of failure allows IT professionals to guarantee uninterrupted operation and availability of at least 99 percent annually.

In excellent availability designs, identical software is installed on numerous machines in a redundant fashion so that each one can be used as a replacement in the event that one component fails. Absent this aggregation, a website or application's failure would prevent access to the service until it was fixed. A crucial aspect of a system that ensures it runs for a long period in relation to being "100% operational" or "never failing" is high availability (Khalel et al., 2022). The economic viability of a cloud computing platform, on which hundreds of firms have installed their business systems, is determined by its high availability. Numerous researchers concentrate on increasing system availability (He, 2020).

Whenever the system is accessible at least 99.999% of the time, a high level of availability is accomplished. Consequently, for highly available systems, the maximum amount of downtime permitted annually is about 5 minutes (Park et al., 2021). The fact that microservices and packaging are compact and lightweight will inevitably help to increase dependability (Todorov, 2022). For its controlled microservice-based applications, Kubernetes offers restorative [8]. When a host fails, Kubernetes has the option to replace or reschedule containers in addition to restarting any failed ones. While they are waiting to be ready again, the healing capacity is also in charge of propagating information about the unhealthy containers. The availability of the services offered by the applications implemented using Kubernetes would automatically increase as a result of these features (Moravcik et al., 2022).

The goal of "Kubernetes High Availability" is to configure "Kubernetes" and all of its auxiliary components so that there is no single point of failure. While a multi-master colony uses several parent nodes, every one of which has a connection to the same worker nodes, a single master cluster can easily fail. For monitoring the "High Availability (HA)" of the installed microservices, Kubernetes provides three layers of health checks and repair actions. First, Kubernetes checks the health of the software elements running inside a container at the program level using process health checks or preconfigured probes (Moravcik et al., 2022).

If the Kubelet detects the inability in either scenario, the container is restarted. Second, Kubernetes detects pod errors at the individual pod level and responds in accordance with the specified restart policy. Last but not least, Kubernetes monitors the cluster's nodes at the node level to detect node failures using its dispersed daemons. A pod is moved to another healthy node if the node housing it fails. Three sets of situations of failure have been identified by researchers for various levels of health checks. The VLC containers procedure failure in the first batch is what led to the application failure. The interruption of the pod container processes is to blame in the second set, and a malfunction of the node is to blame in the third set. Different redundancy models [14] and Kubernetes configurations with both the minimum and the best-performing settings have been tested for each set (Mubina et al., 2022).

5.3 Kubernetes for Stateful Databases:

A higher-level abstraction designed for (replicated) stateful applications like databases is called The StatefulSet, formerly known as PetSet. The StatefulSet's fundamental function is to control the installation, expansion, and upgrading of a collection of replica pods (Mubina et al., 2022). The chronological arrangement and individuality of pods and pod resources are guaranteed by the StatefulSet in a number of different ways. Each pod in a StatefulSet has a distinct, permanent identity and hostname that are preserved even in the case of restarts. Additionally, each pod is connected to reliable persistent storage.

Stateless microservices are now typically hosted using Kubernetes. Stateful programs save information on persistent disks for use by the website's server, clients, and numerous other programs.

A database or key-value store where information is kept and accessed by different applications is an illustration of a stateful application. It provides basic capabilities like cluster resource administration and the implementation and synchronization of microservices. Stateful microservices, like databases, are often housed outside of Kubernetes and maintained by subject-matter specialists, although unstructured modules are well suited towards being hosted in Kubernetes. Stateful services, such as databases, should be operated in Kubernetes to make use of its capabilities and ease of use and to standardize the environment throughout the entire application stack (Füstös et al., 2022).

Kubernetes, which can be basically described as a big, shared API for managing and orchestrating container-based applications, has become a well-established pillar of the contemporary technology environment. Nevertheless, Kubernetes was a lot simpler beast when it initially emerged from Google in 2014. The Kubernetes "Primitives"—the fundamental ideas or components of Kube that are stated in the API—lacked many of the functionalities that business users currently count on (Füstös et al., 2022). As Kubernetes progressed, various degrees of reliability and performance were introduced to the open-source undertaking, including the "Operator framework and Service Mesh". The notions of state, stateful applications, and persistent storage are perhaps foremost within these retrofits. Kubernetes was primarily designed to manage headless applications—ephemeral processes and functions that could be instantly spun up, started, relocated, including terminated without transferring the application's active state and user activity (Mubina et al., 2022).

"Pioneers of container-based development continued to foresee a dualistic world of applications as recently as 2015, with permanent applications and databases executing in virtual machines beside stateless services being deployed in containers. As statefulSets (petSets) were widely available in Kubernetes in 2016, the first trustworthy version of State was introduced. Presently, deploying stateful applications on Kubernetes is simple at first glance. All you need to do is download a Kubernetes Operator for your preferred database, key-value store, or streaming application, such as Redis or Kafka, or use a few lines of YAML to create persistent volumes from the platform's built-in block store"(Füstös et al., 2022).

5.4 Evaluating High Availability Solutions on Kubernetes:

Automatic placement and balancing of containerized workloads, as well as cluster scaling in response to customer growth, are all performed by Kubernetes. But in order to implement Kubernetes successfully in production at scale, IT operations teams must take measures to guarantee its high availability. Although Kubernetes maintains container clusters, some people might assume that it is redundancy by default. However, this isn't exactly true. The worker nodes, also known as the nodes that run vessels, are exchangeable. As a result, in a multi-node cluster, an outage of one or more nodes only causes workloads to be redistributed to one or more of the remaining worker nodes, causing no interruption to application users (Ghorab & St-Hilaire, 2022).

The "top-level nodes and etcd datastore of a Kubernetes cluster," in contrast to the worker nodes, are not necessarily identical. One typical style of design is exemplified by Kubernetes, a distributed framework with a non-redundant management oversight plane. Running applications won't be affected by the failure of a single comprehend, but it is going to stop the system from scheduling new workloads or renewing ones that are already in progress should those responsibilities or their entire node fail (Moravcik et al., 2022). Critical enterprise workloads might require extremely accessible Kubernetes clusters. Without modifying Kubernetes itself, a company can create a single-site dependable cluster; instead, Kubernetes's high availability is mostly a function of correct design, setup, and networking (Lemoine, 2022).

Since the main node elements are essential to the functionality of Kubernetes clusters, increasing redundancies to the master components is necessary to create a high availability environment. For both single data centre installations and geographically separated clusters, the devil is in the details (Park et al., 2021). The etcd data plane and its master node-based control plane components need to be protected from failure within a single data centre. Both of these components must be present in a highly reliable Kubernetes cluster and executed on different machines. The Kubernetes assistance group should additionally routinely back up these files to a separate archive system (Moravcik et al., 2022).

For HA topologies, Kubernetes suggests two different design strategies. Stacking control plane nodes, in which the control plane and etc components operate on the same nodes, is the first method for creating highly available Kubernetes clusters. In kubeadm, stacked clusters are the default configuration. A Kubernetes cluster can be set up with a fair set of defaults using the kubeadm tool. Kubernetes Operations, often known as kops—which is kubectl for clusters—is a competitor to kubeadm (Park et al., 2021). Kubeadm configures a cluster with all the control plane components on a single node in the simplest case, but there are ways to make deployments that are more complex. On each control plane node, Kubernetes automatically establishes a local etcd key-value store with Kubeadm, making setup and management simple. Considering every essential component for managing a cluster operates on the same server, this design poses a danger for "Kubernetes deployments" that must resist disruptions. Businesses using stacked clusters are advised to have at least three nodes of resilience (Lemoine, 2022).

5.5 Scalability of Databases on Kubernetes

It takes a lot of effort to maintain a database on Kubernetes and offer capabilities similar to those found in completely handled and cloud databases. If users are worried about maintaining high availability and staying away from single-point disappointment, flexibility is an important factor to take into account. The native features of Kubernetes are helpful when it comes to database flexibility. However, the main issues are data synchronization and determining the proper scaling demands. As a consequence, maintaining a scalable database strategy in a cloud-native context is difficult. According to the application needs, which may change over time, Kubernetes users have the ability to dynamically grow the total number of containers used. Via the command line, changing the number is simple (Sithiyopasakul et al., 2021).

When thinking about Kubernetes' horizontal Pod auto-scaling, Kubernetes gives you the ability to generate and distribute Pods (Chareonsuk & Vatanawood, 2021). A pod in Kubernetes is a grouping of one or a few containers that are closely connected via a shared IP address and port space (Kim et al., 2021). A pod comprises resources for storage, a specific network IP address, a program container (or, in certain cases, many containers), parameters dictating how the container(s) should function, and more. A network IP, storage facilities, a single application container (or several containers), and operating-system-specific characteristics are all wrapped up in one process. A Kubernetes installation unit known as a pod consists of one container or a small group of connected containers which collaborate on resources (Klos et al., 2021).

5.6 Enterprise Databases on Kubernetes:

Enterprises and major organizations handle their massive data collection using enterprise databases. A database like these aids businesses in increasing productivity.[20] An enterprise database is capable of supporting between 100 and 10,000 users at once and is strong enough to process their queries concurrently. Businesses frequently use corporate databases to plan, strategize, and regulate procedures. They are typically used to increase organizational efficiency. They aid in fostering organizational effectiveness by lowering costs (Kim et al., 2021).

An ideal company database is packed with a wide range of capabilities, all of which are intended to increase organizational productivity and efficiency. Because they have evolved into collectors and repositories of organizational data, enterprise database systems have become a crucial component of organizations in addition to serving as integrative services. This data was previously subjected to a rather rudimentary independent evaluation method, which might offer useful yet fundamental data for decision-making by management Data collecting and database storage have both become more complex and diverse as novel innovations have emerged. The information that management may use for decision-making and strategic planning has significantly improved in both professionalism and promptness as a result (Kim et al., 2021).

The apparent benefits of running Enterprise Postgres on Kubernetes include the ease of implementation, the capacity to manage the entire stack with a single orchestration tool, auto healing, and automatic reprovisioning of failed containers, all of which increase dependability. For instance, Kubernetes will automatically self-heal and reschedule the workload on a different node if one of the nodes hosting a database fails. It can immediately re-initialize the new node as a new replica and choose a new database primarily operating on an already-existing duplicate with the help of the database management software. However, there are other, more significant advantages why you should use Kubernetes to run databases. Most businesses prefer to use "Database-as-a-Service" to manage their databases. to automatically provision a self-healing database with monitoring and backups. Although the majority of cloud vendors deliver this, running it yourself with Kubernetes can save money and offer extra features like multi-cloud and cloud portability (Kim et al., 2021).

5.7 Consideration for running the Stateful Databases on Kubernetes:

The first route recommends using Kubernetes' Statefulsets method that will automatically supply persistent volumes for each pod (Klos et al., 2021), enabling state management for databases. However, employing external volumes for data persistence may cause disk I/O performance bottlenecks and hanging volume problems (Muthanna & Tselykh, 2022). Third-party Operators are used by the second path to enhance the Kubernetes APIs by developing unique resources along with controllers that include which address StatefulSets' limitations (Karypiadis et al., 2022). Nevertheless, the present Operators are inexperienced and unfit for mission-critical workloads, not to mention that the increased tech stack may make running DBs in Kubernetes even more challenging. In contrast to both study methods, some prior research suggests reducing the complexity of the tech stack by adopting a container-native persistence of information solution and then the simple. Run read-only databases using a deployment method in Kubernetes. In addition to enabling Kubernetes to manage read-only DB clusters comprehensively, the streamlined tech stack additionally motivates users to work using the Kubernetes-assisted At least for stateless programs under the present technological trends, stateful in production (Zerwas et al., 2022).

5.8 Comparative analysis:

5.8.1 Stateful Sets vs. Stateless Operation

This research paper has analyzed and proposed the enterprise database solution based on the stateful Kubernetes database solution that has High availability (HA) systems. There are also stateless Kubernetes database systems available for enterprises. There have been many research papers that proposed stateless Kubernetes systems. One of the most comprehensive research papers that analyzed the stateless Kubernetes database solution is the research conducted by Li et al. in their research work “Stateless Database Solutions on Kubernetes: An Experimental Study” (Zerwas et al., 2022). This article explains the stateless Kubernetes Database Solutions as a good and highly scalable system for enterprises with the presentation of an experimental solution.

A stateless application, as explained by Li et al. (Zerwas et al., 2022). is one that is independent of any dependent storage, whether it be in the context of Kubernetes or any other architecture. The service is stateful if the web server stores data in the backend and utilizes it to identify the user as an always-connected client. While Stateless, the server does save data, but it does so in a database for user/client authentication anytime a connection is required. This means that in a Kubernetes cluster, application data stays with the client, and the application state does not store on the cluster.

When a user enters into a stateful application (like Oracle EBS), which has one application server and one database server, the application first verifies its identity using the credentials they have provided. Following database-based authentication, it approves and pulls all of the user's rights. The status is false prior to the user logging into the program. When logging in successfully, the state changes to true. Since the state is already true, the user does not need to authenticate anew when they click on any responsibility or HTML page.

The benefit of a stateful database solution is that when it first accesses the database, the login state is saved. Since the status (some kind of flag) is already true on the second occasion the user clicks on any obligation; there is no need to access the database again. In a stateless application, if a user logs in via a load balancer and the load balancer then picks up a server on the left, the user may continue to make subsequent requests because his session variable is already set to true. Although the user is still not signed onto the server, the load balancer's connection to the one on the right will be denied.

Stateless database solutions are scalable since users do not need to store the state, but stateful systems are not very scalable. In general, container storage through a container's file system is transient and subject to destruction upon container creation. This transitory storage has a lifespan that is limited to the duration of the specific pod, and it cannot be distributed among pods. Persistent storage is a need for stateful applications. You can construct and use persistent volumes to store data outside of containers, giving it a long-lasting home and preventing data loss.

6 Methodology

In this section, the methodologies used to achieve each of the objectives are discussed. A brief outline of these methodologies is as follows:

1. To design and implement a high-availability environment on Kubernetes.

- The methodology to achieve this objective involves the following steps:
    - Set up an Azure Kubernetes Service (AKS) cluster with multiple nodes for redundancy.
    - Create a Kubernetes deployment for each enterprise database to be tested.
    - Configure the deployment to use multiple replicas for the kubernetes statefulset for database.

2. To test the deployment and performance, and other factors of enterprise databases (PostgreSQL, TiDB, MariaDB, and MySQL) in a HA setup in a high availability environment on Kubernetes.

- The methodology to achieve this objective involves the following steps:
    - Deploy multiple instances of each database using Kubernetes stateful sets or replica sets to create a HA setup.
    - Use a workload generator such as sysbench to simulate a variety of read and write requests to the databases.

3. To test the scalability of the Kubernetes stateful sets for all four databases.

- The methodology to achieve this objective involves the following steps:
    - Scale up and scale down the number of replicas for the database in a stateful set and observe the efficiency of the process.

4. To document the findings and provide recommendations for deploying enterprise databases in a high-availability environment on Kubernetes.

- The methodology to achieve this objective involves the following steps:
    - Summarize the results of the testing and analysis conducted in objectives 1-3.
    - Provide recommendations for deploying enterprise databases in high-availability environments on Kubernetes based on the findings of the testing and analysis.
    - Document the methodology used for testing the databases and the results of each test, including any issues encountered and solutions implemented.

Description of the experimental setup, including details on Kubernetes and AKS

For achieving each of the objectives, the following setup was used,

1. To design and implement a high-availability environment on Kubernetes.

    - Azure Kubernetes Service (AKS) was used to set up a cluster with multiple nodes for redundancy. Kubernetes was configured to provide high availability by deploying resources such as replica sets and stateful sets. Created a Kubernetes deployment for each enterprise database to be tested and configured the deployment to use multiple replicas and ensure that the databases can handle failover and load balancing.

2. To test the four enterprise databases (PostgreSQL, MariaDB, TiDB and MySQL) in a high-availability environment on Kubernetes.

    - Deployed multiple instances of each database using Kubernetes stateful sets or replica sets to create a HA setup. A workload generator sysbench was used to simulate a variety of read and write requests to the databases. The workload generator against each database was executed to test their ability to handle multiple read and write requests simultaneously.

3. To evaluate the scalability of the enterprise databases in a high availability environment on Kubernetes.

    - Scale-up and scale-down operations were performed on each database’s stateful set, and observed how efficient and quick the scalable process was for all of them and compared.

4. To evaluate the performance of the enterprise databases in a high availability environment on Kubernetes.

    - Used a workload generator to generate a range of workloads and measured the response time and throughput of each database. The workload was increased by increasing the number of concurrent requests and measuring the performance of the databases under heavy load. Kubectl commands were used to track the availability of databases pods while the benchmark tests were being executed.

5. To document the findings and provide recommendations for deploying enterprise databases in a high-availability environment on Kubernetes.
    - Summarized the results of the testing and analysis conducted in objectives 2-4 and provided recommendations for deploying enterprise databases in high availability environments on Kubernetes based on the findings of the testing and analysis. Documented the methodology used for testing the databases and the results of each test, including any issues encountered and solutions implemented.

7 Implementation

The experiments were performed on Azure Kubernetes Service (AKS) to set up the HA environment for the four enterprise databases PostgreSQL, TiDB and MySQL and MariaDB. The deployment of a Kubernetes cluster in Azure is very simple. Using AKS for the tests, saved a lot of time because the configuration of the Kubernetes cluster from scratch requires effort and it is time-consuming. The following sections describe the implementation details for each step.

7.1 WSL (Ubuntu) Setup

The workload testing was performed through WSL (Ubuntu) client; therefore, it was important to install and configure the required packages on it to connect to the AKS cluster. To install the Windows subsystem for Linux, in the Windows search bar, type Microsoft Store and open it. Search for Ubuntu 20.04 (the latest version) in the Microsoft Store and install it. The Ubuntu app was required to execute Azure and other Linux commands. Installation of the required packages was completed in the next few steps.

To execute Azure commands from WSL, the Azure CLI package was installed with the "apt install azure-CLI" command executed via the root user. After executing the command, you'll be redirected to the Azure portal for authentication. Once authenticated, come back to the Ubuntu app to check your session. Once the Ubuntu app was installed, the latest Linux packages, Kubernetes packages and sysbench were installed.

7.2 Kubernetes Cluster Creation

Kubernetes cluster was created on AKS with the following resources:

- Two worker nodes with 8 CPU cores, 32 GB RAM, and 128 GB storage each.

AKS was used to create the cluster since it provides a fully managed Kubernetes service that can be quickly and easily set up, configured, and managed. In the case of TiDB, the Kubernetes cluster was created in Azure with the help of az CLI commands as instructed in their manual. Table 7.1 shows the configuration of AKS cluster.

Kubernetes Cluster	Configuration
Image	Standard D8s_v3
Memory	32
CPU	8

Table 7.1: Kubernetes Cluster Configuration in Azure

7.3 Database Deployments

To deploy databases on Kubernetes, the best possible approach was chosen. For MySQL and TiDB installation, Kubernetes operators were the choice, while for PostgreSQL, a helm chart from bitnami was used. Operators are Kubernetes extensions that provide a way to manage complex applications and services, including databases. The following operators were used:

MySQL:

The MySQL operator from Oracle was used to deploy and manage MySQL clusters in Azure Kubernetes Service **(MySQL, 2021b**). MySQL database deployment was straightforward, and no additional configuration was required. To connect to the MySQL database, a port forwarding command was executed in one terminal in Ubuntu WSL to create a connection from the local machine to the MySQL database on port 3306 and from the other terminal, sysbench tests were executed.

TiDB:

TiDB operator developed for Azure Kubernetes Service from PingCAP was used to deploy 2 TiDB and 3 TiKV pods (**TiDB, 2021b**). TiDB operator from PingCAP was used to install the TiDB cluster in Azure. It was configured out of AKS in Azure. To connect to the TiDB database, a port forwarding command was executed in one terminal in Ubuntu WSL to create a connection from the local machine to the TiDB database on port 4000 and from the other terminal, sysbench tests were executed.

There were some specific recommendations on how to run the sysbench test on TiDB were followed. For example, to set the global tidb_disable_txn_auto_retry parameter value to off. If the parameter value is ON, then TiDB can rollback a transaction when a concurrency conflict is detected in the database, and it can terminate the sysbench session (**PingCAP, 2021**).

PostgreSQL:

Helm chart developed by bitnami was used to install PostgreSQL HA database instances. All the manifests and Kubernetes resources are bundled in the helm chart (**PostgreSQL, 2021b**). To connect to the PostgreSQL database, a port forwarding command was executed in one terminal in Ubuntu WSL to create a connection from the local machine to the PostgreSQL database on port 5432 and from the other terminal, sysbench tests were executed.

MariaDB:

To deploy the MariaDB Galera cluster on AKS, a helm chart by bitnami was used. All the resources are bundled into the helm chart, and it's a multi-primary database cluster solution (**MariaDB, 2021b**).

A new database called "mariadb_sysbench" was created, and a user named "sysbench" was set up inside this database. A Sysbench user was used to perform the workload testing. As with other databases, similar port forwarding was done on port 3306 for localhost so that sysbench can make a client session can connect to the database.

7.4 Parameters for Workload Testing

The following table 7.2 shows the input parameter passed to sysbench for generating the workload.

Database	Tables	Table Size (No of rows)	Time (Sec)	Threads	Report Interval	db-driver
MySQL	10	1000	120	1,2,4,8	60	MySQL
PostgreSQL	10	1000	120	1,2,4,8	60	pgsql
TiDB	10	1000	120	1,2,4,8	60	MySQL

Table 7.2: The input parameter passed to sysbench for generating the workload.

For each database, 10 tables were created, and 1000 rows were inserted into the tables. For MySQL and TiDB, the db driver of mysql was used as TiDB is compatible with MySQL. Sysbench tests were executed on 1,2,4 and 8 CPU threads on Readonly, WriteOnly, ReadWrite, and Deleteonly operations for each database, respectively.

8 Results

In the era of big data, database performance is a crucial factor in any system architecture. Kubernetes has gained a lot of popularity in recent years as a container orchestration system. Designing a highly available Kubernetes setup for a workload depends on various factors, such as the specific requirements of the workload, the resources available, and the performance characteristics of each component in the system, including the databases.

Before designing a Kubernetes HA setup, it's important to understand the performance requirements of the workload, including the performance characteristics of the databases used in the system. This includes understanding factors such as the read-and-write throughput requirements, latency requirements, and resource utilization patterns of the databases.

Based on this information, one can then design a Kubernetes HA setup that meets the specific requirements of the workload. This might include deploying multiple replicas of the database to ensure high availability, using load balancing and auto-scaling to handle spikes in traffic, and implementing failover mechanisms to ensure that the system continues to function even in the event of a failure. Overall, understanding the performance characteristics of each component in the system is essential to designing a highly available and performant Kubernetes setup that meets the specific needs of the workload.

Database Deployments:

When it comes to the database deployment on Kubernetes, it was observed that it's better to use the Kubernetes operator and helm chart developed by the product vendor or the pioneer companies who are in the field. Companies keep coming up with new products and doing continuous development to make the product better. For MySQL database installation, Kubernetes operator from MySQL was used and deployment was smooth and straightforward. The MySQL Operator simplifies the deployment process by abstracting away the complexities of setting up a MySQL database. It automatically provisions and configures the necessary resources, such as PersistentVolumeClaims (PVCs) for data storage, Services for network access, and Deployments for managing the database pods.

Bitnami, a well-known provider of ready-to-use software packages, offers a PostgreSQL HA package that simplifies the deployment and management of a highly available PostgreSQL environment. No problems were observed when installing PostgreSQL on AKS via Bitnami's helm chart.

TiDB deployment was performed by installing the TiDB operator from PingCAP. Installation was a little bit cumbersome as there was no single command installation as done with MySQL and PostgreSQL. For Azure Kubernetes service, a set of az cli commands were required to be prepared to create node pools for the TiDB cluster, and then a separate database and user were created to perform the sysbench test.

MariaDB Galera cluster installation was also done by using Bitnami's helm chart, and installation was simple. Just like TiDB, here for MariaDB, a separate database and users were created to connect to the database to run sysbench tests.

Overall, the database deployment process was quite simple for all the four databases as the companies providing Kubernetes operators or helm charts have done a lot to make this process easier.

Performance Evaluation:

This report presents the results of a benchmark test conducted to compare the performance of four databases (MySQL, PostgreSQL, TiDB, and MariaDB) in a Kubernetes HA setup. The benchmark tests were performed using 4 different workloads (OLTP_READ_ONLY, OLTP_WRITE_ONLY, OLTP_READ_WRITE, and OLTP_DELETE) and were measured based on transactions per second (TPS), CPU utilization, memory utilization, and latency.

The benchmark tests were performed on AKS (Azure Kubernetes Service). Four benchmark tests were performed for each database and on multiple CPU threads (1,2,4,8). Sysbench is an open-source tool designed to benchmark CPU, memory, and I/O performance. It is widely used for database benchmarking, and it provides a set of

OLTP (Online Transaction Processing) workloads that can simulate real-world application scenarios, was used to perform benchmarking tests that measure transaction per second (TPS), latency (response time in MS). CPU and memory usage for Kubernetes service was collected from the Azure portal for the duration of each benchmark test (120 sec). The OLTP_READ_ONLY, OLTP_WRITE_ONLY, OLTP_READ_WRITE, and OLTP_DELETE workloads were used to simulate real-world application scenarios. By using Sysbench to simulate real-world application scenarios, we were able to accurately measure the performance of each database in a Kubernetes HA setup and draw meaningful conclusions about their performance.

Through the benchmark tests, the performance of MySQL, PostgreSQL, MariaDB, and TiDB was analyzed in terms of OLTP Read-Only, OLTP Write-Only, OLTP Read-Write, and OLTP Delete, and evaluated their performance based on the key metrics of TPS, Latency, CPU utilization, and Memory utilization.

OLTP Read-Only Test:

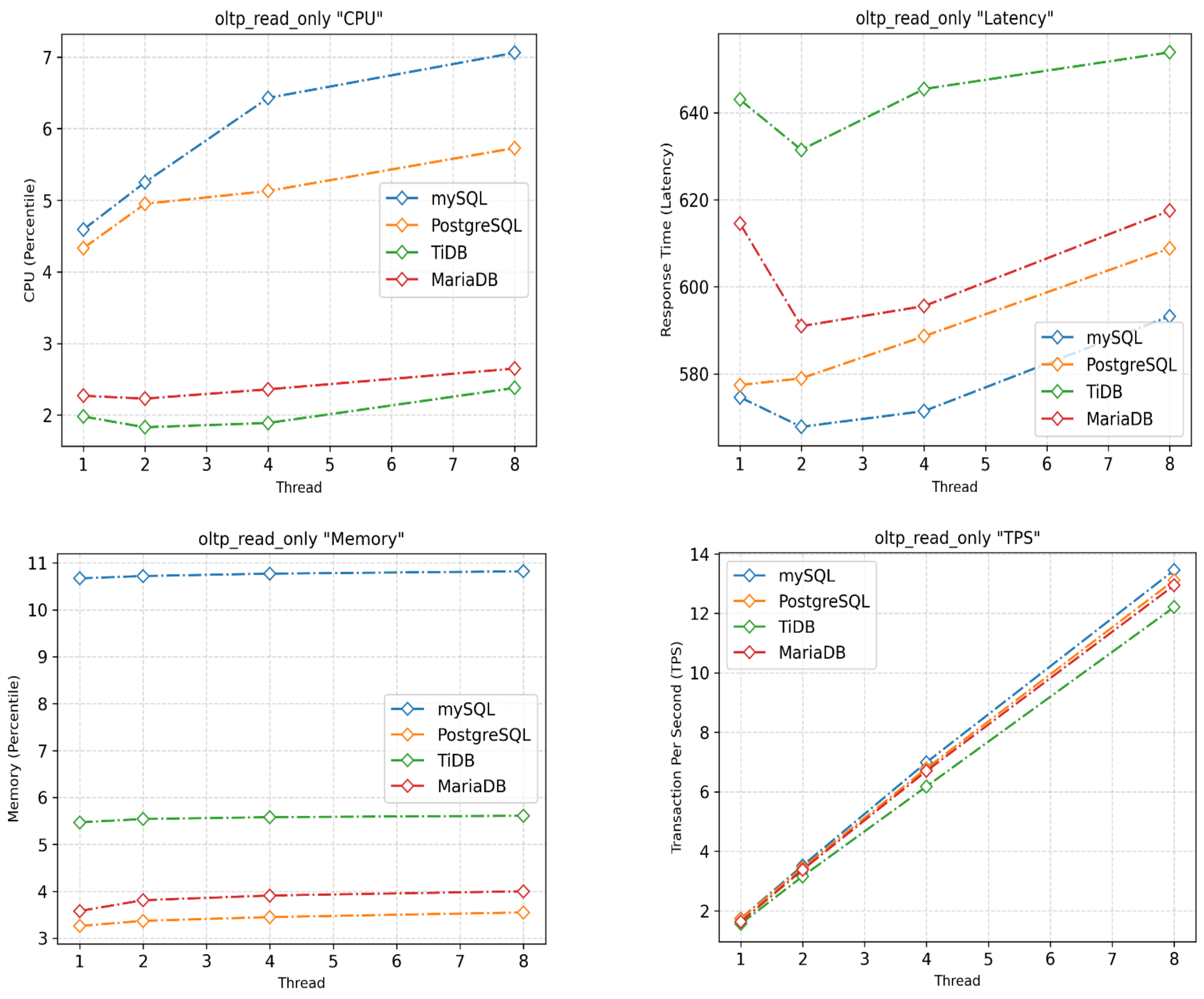


Figure 8.1. OLTP Read-Only Test

The OLTP read-only test shown in figure 8.1, suggest that MySQL came first, then PostgreSQL and MariaDB with a slight difference; TiDB was last in terms of TPS, with MySQL achieving a maximum TPS of 13.45, compared to PostgreSQL's 13.11, MariaDB's 12.94 and TiDB's 12.21. MySQL outperformed the remaining three databases in terms of latency; PostgreSQL and MariaDB were somehow close at 2 threads and forward, with TiDB having higher latency than the other three databases. MariaDB could be the chosen one considering CPU and Memory usage as it shows constancy, compared with TiDB and PostgreSQL, while MySQL uses more resources than the other three databases.

OLTP Write-Only Test:

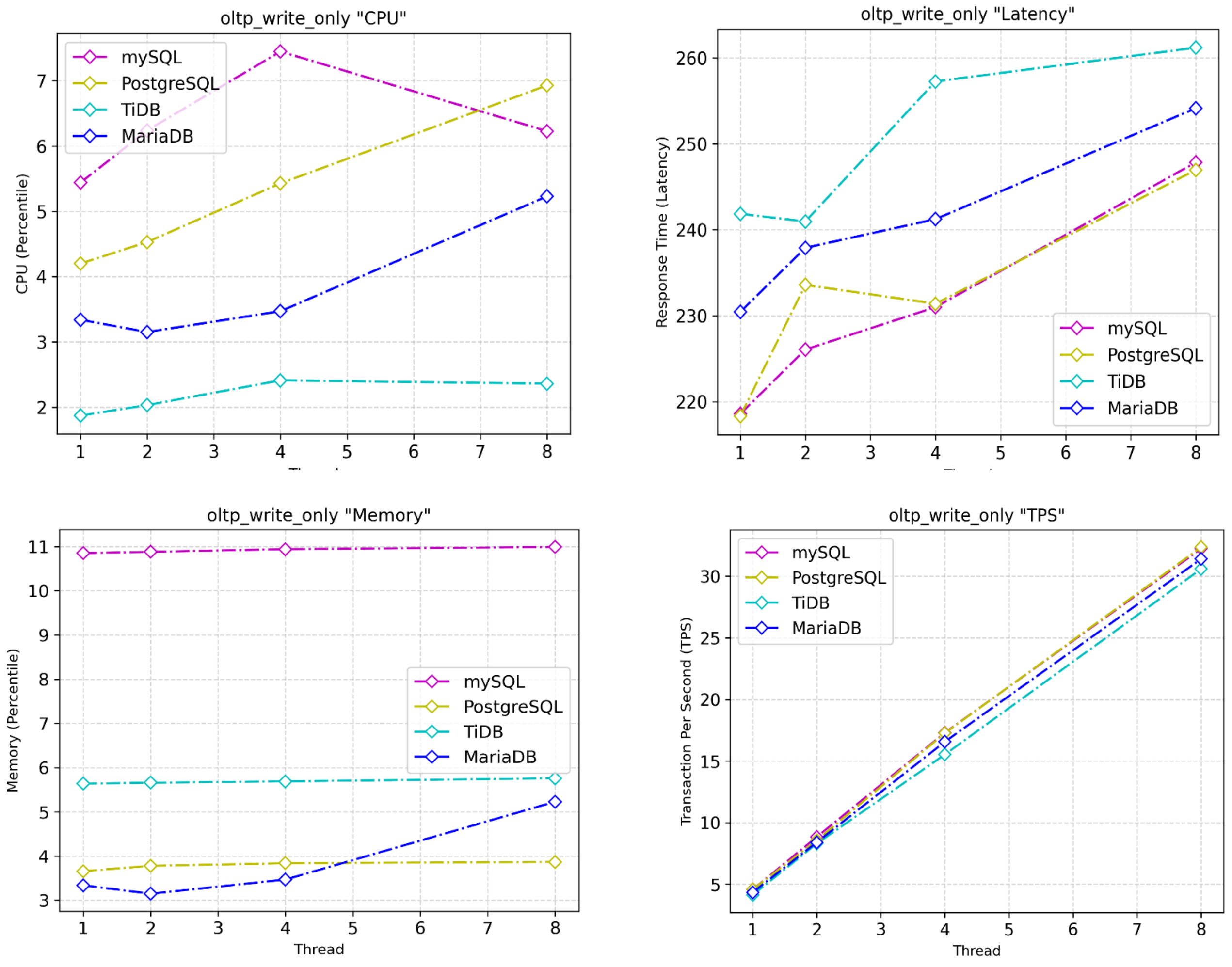


Figure 8.2: OLTP Write-Only Test

The results as for OLTP Write-Only test as shown in figure 8.2, suggest that MySQL performed slightly better under 4 threads, but PostgreSQL was slightly better at 4 to 8 threads, MariaDB came third, and TiDB was last in terms of TPS, with PostgreSQL achieving a maximum TPS of 32.36, and MySQL 32. 24 compared to MariaDB 31.41, and TiDB's 30.6. MySQL and PostgreSQL were the same at latency under 4 threads and above it; considering both factors, one might go with MySQL, but if the memory stats are added, then PostgreSQL, with no doubt, will be the chosen one. TiDB has higher latency and lower CPU usage than the other three databases. The CPU and Memory usage was different for all four databases, with MySQL using more resources than the other three databases.

OLTP Read-Write Test:

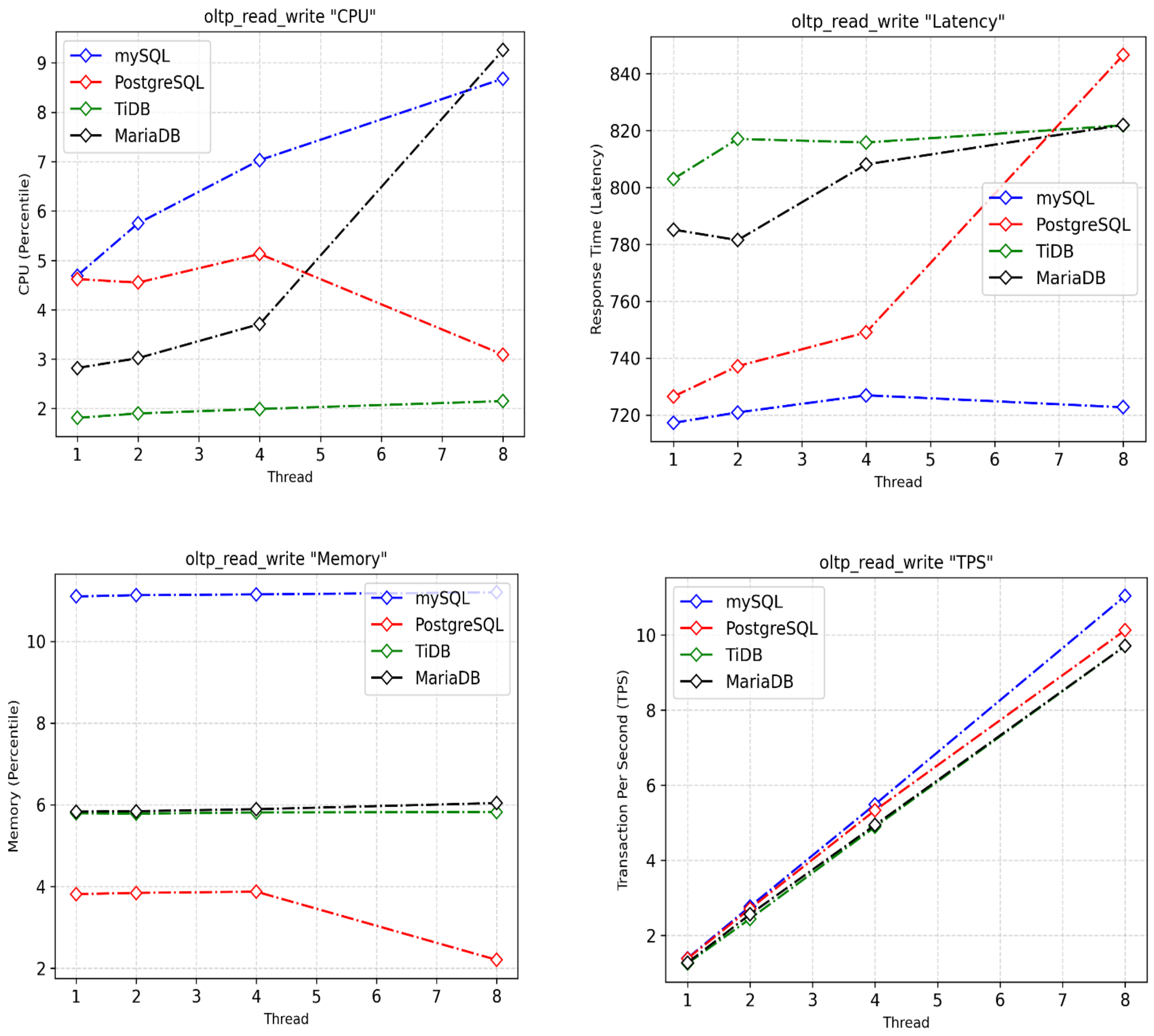


Figure 8.3: OLTP Read-Write Test

The results for OLTP read-write test as shown in Figure 8.3 suggests that MySQL was outperformed all other three databases at 4 threads and above, PostgreSQL second, MariaDB and TiDB were pretty much the same in terms of TPS, with MySQL achieving a maximum TPS of 11.04, compared to PostgreSQL's 10.13, MariaDB and TiDB's at 9.71.

MySQL outperformed all three databases in terms of latency. PostgreSQL showed a continuous increase at 4 threads and above with similar Latency for MariaDB and TiDB at 8 threads. CPU and memory, it was noticeable that PostgreSQL had lower usage of resources at 4 threads and above, while TiDB is a key competitor for PostgreSQL in terms of CPU utilization.

OLTP Delete Test:

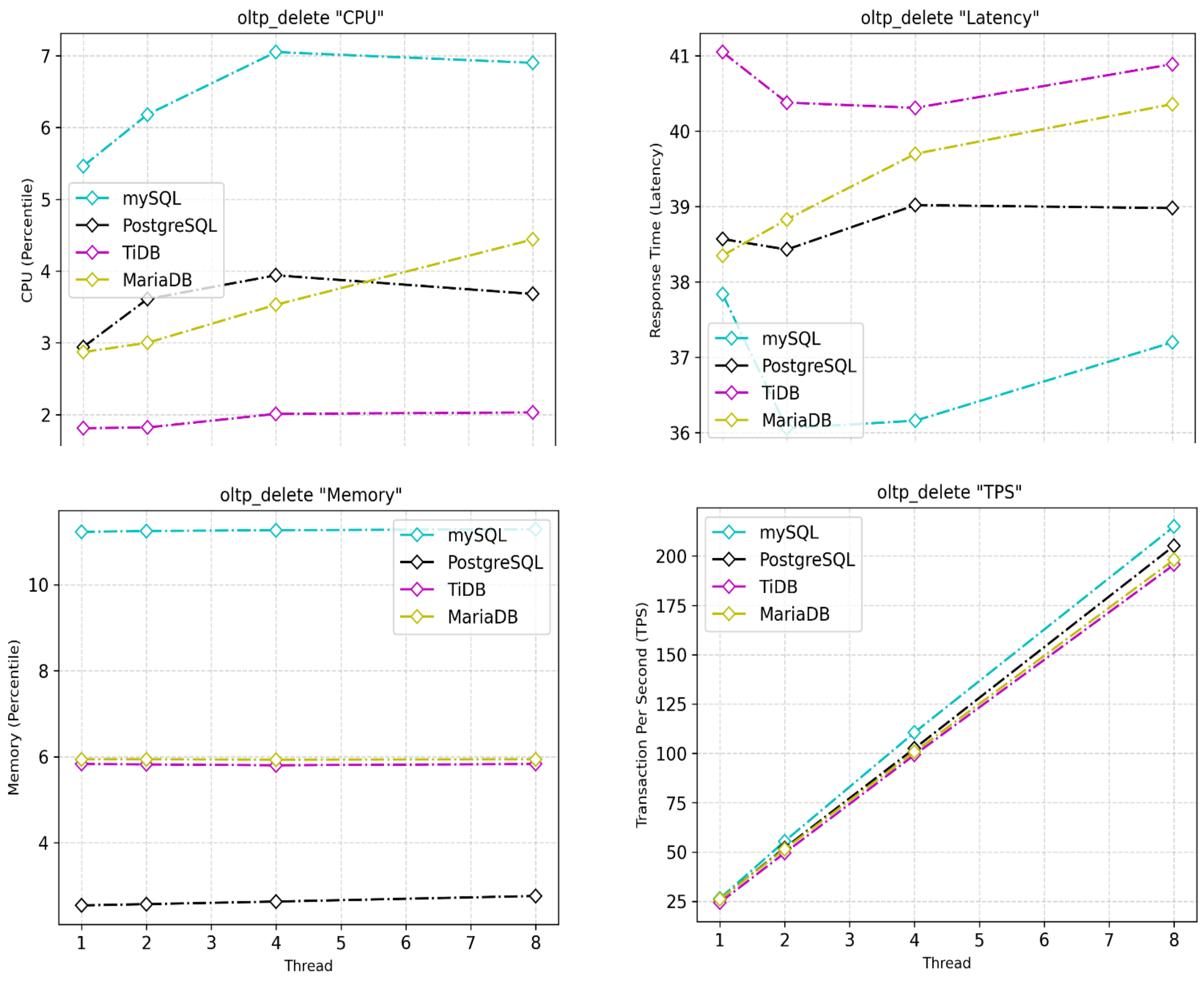


Figure 8.4: OLTP Delete Test

The OLTP delete test was performed to evaluate the performance of the databases under delete workloads. The results as shown in figure 8.4 suggest that MySQL outperformed PostgreSQL, MariaDB and TiDB in terms of TPS and Latency, with MySQL achieving a maximum TPS of 214.96 and latency of 37.2, compared to PostgreSQL TPS of 205.2 and latency of 39.02, MariaDB TPS of 198.19 and latency of 40.36, TiDB TPS of 195.62 and latency of 40.89. At CPU, TiDB had no competitor, same for PostgreSQL at Memory usage, with MySQL using more resources than the other three databases.

Scaling:

The scaling up and down of the number of replicas in a stateful set for each database was tested. For the TiDB database, the scaling up of the TiDB pod from 2 to 5 failed and was never completed, while the scale-down to an original number of the pods (2) was successful. Further, scale-down from 2 to 1 replica was tested, and it was a successful operation. Lastly, the scale-up of TiDB from 1 to 2 replicas was successful. However, the result of the scale-up and scale-down of TiKV, which is a key value store for TiDB, was positive. The resultResults showed that the scale-up from the existing 3 replicas to 5 and the scale-down operation to the original replicas both were successful.

For PostgreSQL, MySQL, and MariaDB, scale-up and scale-down operations for replicas were smooth.

9 Discussion

The benchmark results indicate that MySQL performed the best in terms of TPS and Latency for all tests, except for the OLTP Write-Only workload where PostgreSQL performed better. However, it's worth noting that MySQL had high CPU and memory utilization during the tests, indicating that it may require more resources to operate at peak performance.

In terms of memory usage, PostgreSQL performed the best, indicating that it may be a better option for workloads that require high memory usage.

OLTP Read-Only: MySQL showed the highest TPS (Transactions Per Second) and lowest Latency among the four databases. PostgreSQL and MariaDB followed closely behind MySQL in terms of TPS and Latency, while TiDB performed the worst in this workload. MySQL had high CPU and memory utilization during this test, indicating that it may require more resources to operate at peak performance.

OLTP Write-Only: PostgreSQL performed the best in terms of TPS and Latency in this workload, followed closely by MySQL. MariaDB and TiDB showed lower TPS, and higher latency compared to the other two databases. MySQL had high CPU and memory utilization in this test, similar to the results of the OLTP Read-Only workload.

OLTP Read-Write: MySQL once again outperformed the other three databases in terms of TPS and Latency. PostgreSQL showed weak performance at 4 threads and above and fell behind MySQL in terms of overall performance. MariaDB and TiDB had similar performance in this workload, with lower TPS and higher Latency compared to MySQL and PostgreSQL. MySQL had high CPU and memory utilization in this test, indicating that it may require more resources to operate at peak performance.

OLTP Delete: MySQL once again showed the highest TPS and lowest Latency in this workload, followed by PostgreSQL. MariaDB and TiDB showed lower TPS and higher latency compared to the other two databases. MySQL had high CPU and memory utilization in this test, like the results of the OLTP Read-Only and OLTP Read-Write workloads.

When it comes to measure memory utilization, PostgreSQL performed the best among the four databases in all four workloads, indicating that it may be a better option for workloads that require high memory usage.

Kubernetes Database Operators are specifically designed to automate the deployment and management of databases on Kubernetes. A Kubernetes Operator is a method of packaging, deploying, and managing stateful applications on Kubernetes and it extends the Kubernetes API to create, configure, and manage complex applications as native Kubernetes objects. Installation of MySQL InnoDB Cluster and TiDB cluster was performed using Kubernetes operators for MySQL and TiDB operator respectively. These operators can handle tasks such as scaling, backups, upgrades, and failovers, making it easier to manage the lifecycle of a database in a Kubernetes environment. Operators automate the complex deployment process of databases by encapsulating best practices and domain-specific knowledge into the operator's code. These operators use CRDs to define custom resources that represent the desired state of a database.

Installation of PostgreSQL and MariaDB were done by helm charts from Bitnami, and process appeared to be pretty simple. Helm Chart is a package manager for Kubernetes that simplifies the deployment and management of applications. Helm charts are the packages used to define, install, and configure applications on Kubernetes clusters. A Helm chart contains templates, values, and dependencies required to deploy a specific application or stack.

The Helm chart for PostgreSQL contains the replication manager, to manage replication and failover on PostgreSQL clusters. However, replication and failover capabilities of the product were not evaluated in the research. The chart does exactly what it is intended for and installs PostgreSQL database with 3 replicas set within a few minutes. MariaDB Galera cluster installation was performed by helm chart from Bitnami, and MariaDB database with 3 replicas set was up and running after few minutes of executing the commands. From a research perspective, studying Kubernetes database operators and Helm charts in the context of database deployment on Kubernetes can contribute to understanding and improving managing stateful applications in containerized environments. It can address various aspects such as performance, scalability, resiliency, security, and developer experience, ultimately enhancing the adoption and effectiveness of deploying databases on Kubernetes.

Scaling capabilities were also testing for these database solutions on kubernetes. MariaDB Galera cluster, PostgreSQL, MySQL InnoDB, and TiDB Cluster solutions able to perform scale-up and scale-down operations as expected.

There were quite a few obvious features like data consistency, database availability when pods are being scaled-up or scaled-down, security, upgrades could not be evaluated in this research. It would have been really nice if more and more features could be included in the research so that readers can make an informed choice based upon the results. Future work section covers features that must be considered when evaluating stateful databases on Kubernetes.

10 Conclusion

Stateful database evolution on Kubernetes is an intriguing area of research that focuses on managing and evolving stateful databases within Kubernetes clusters. Kubernetes, as a powerful container orchestration platform, has primarily been designed for stateless applications. However, with the growing adoption of Kubernetes for various workloads, there has been a demand for efficiently managing stateful applications, including databases. Traditionally, stateful databases have been deployed on dedicated servers or virtual machines with fixed resources and specific configurations. However, migrating these databases to Kubernetes brings a set of challenges due to the dynamic and distributed nature of the platform.

The database deployments process was smooth for all the databases and one can say that there is nothing comparable among them. Kubernetes operators for database and helm charts are a great way to set up the database as they are simple to execute and because of the open-source community support, lots of enhancements and bug fixes are happening rather quickly. Kubernetes has pioneered the containerization orchestration for stateless applications and similar expertise is seemed to be achieved for the stateful application like database. Scaling operations were tested and apart from TiDB's TiDB Server pod scaling, rest scale-up and scale-down operations were successful and turnaround time to provision or delete a pod was within few seconds.

Overall, the benchmark results indicate that MySQL is a strong performer in most OLTP workloads, with high TPS and low latency. However, it also showed high CPU and memory utilization, indicating that it may require more resources to operate at peak performance. PostgreSQL showed strong performance in the OLTP Write-Only workload and consumed less memory compared to the other databases, making it a good choice for workloads that require high memory usage.

Future work and improvements

Project's primary emphasis was on determining how cumbersome or easy is to setup stateful services like database on a high availability environment using Kubernetes and putting business databases through their paces in this context. And to evaluate their performances based on the metrics. Also, to evaluate how scalable these systems are. The findings do give some useful insights, nonetheless, there are several areas that may benefit from more study and development. Following are the few areas that should be considered in the future work:

1. More research into the various configurations of Kubernetes: The investigation concentrated on a particular configuration of Kubernetes; however, there is the possibility for more research into the many settings of Kubernetes to assess the influence on database performance, scalability, consistency, and availability. Project focused on the straightforward deployment of the databases using helm chart or the kubernetes operator but there are many settings which can be looked upon to achieve an optimal performance for a database system.

2. A more in-depth analysis of the features of the database: Even though the project evaluated the performance, scalability, and deployment of the four enterprise databases, there is the potential for a more in-depth analysis of specific features of these databases that may impact how they behave in an environment managed by Kubernetes.

3. Investigation of more databases: The study investigated and assessed a total of four databases; however, there are a great number of additional databases that are accessible and may be appropriate for use in a Kubernetes context. In a further study, the behavior of additional databases might be investigated when placed in this setting.

4. Investigation of potential risks: Security is a vital concern in production situations; thus, future study might investigate the potential risks associated with running databases on Kubernetes.

5. Varied Workload Testing: Experiments were conducted on a small sample size, for instance, duration of each test, no. of tables, no of records and CPU threads. A more

comprehensive test that involves large data and more stressful load test can be performed in the future work.

6. Database consistency: How consistent a database is in maintaining its ACID properties when some pods/nodes are unavailable in the configuration and gives a consistent view of data to application or users.

7. Database replication: Database replication in Kubernetes refers to the process of creating and maintaining multiple copies of a database across different nodes or clusters within a Kubernetes environment. Various synchronization methods, such as synchronous or asynchronous replication, depending on the specific requirements of the application can be tested on specific databases.

8. Database failover: Database failover in Kubernetes involves the process of ensuring high availability and continuous operation of a database in the event of a failure. With the help of Kubernetes features such as replica sets, stateful sets, and services, it is possible to implement a reliable failover mechanism for databases and then can be evaluated.

9. Backup and Recovery: Database backup and recovery in Kubernetes involves implementing strategies and techniques to ensure the protection and availability of data stored within databases running on Kubernetes clusters. Kubernetes provides a scalable and resilient platform for running applications, including databases, and offers various tools and features to facilitate backup and recovery processes. This area must be explored in future research work.

11 Reference List

- Bestari, M. F., Kistijantoro, A. I., & Sasmita, A. B. (2020). Dynamic Resource Scheduler for Distributed Deep Learning Training in Kubernetes. In 2020 7th International Conference on Advance Informatics: Concepts, Theory and Applications (ICAICTA) (pp. 1-6). Tokoname, Japan. https://doi.org/10.1109/ICAICTA49861.2020.9429033
- Bose, D. B., Rahman, A., & Shamim, S. I. (2021). 'Under-reported' Security Defects in Kubernetes Manifests. In 2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS) (pp. 9-12). doi: 10.1109/EnCyCriS52570.2021.00009.
- Chareonsuk, W., & Vatanawood, W. (2021). Translating TOSCA Model to Kubernetes Objects. In 2021 18th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON) (pp. 311-314). Chiang Mai, Thailand. https://doi.org/10.1109/ECTI-CON51831.2021.9454890
- Docker and Container. (2021). Docker. https://www.docker.com/resources/what-container/
- Duan, R., Zhang, F., & Khan, S. U. (2021). A Case Study on Five Maturity Levels of A Kubernetes Operator. In 2021 IEEE Cloud Summit (Cloud Summit) (pp. 1-6). doi: 10.1109/IEEECloudSummit52029.2021.00008.
- Figueiredo, R., & Subratie, K. (2020). Demo: EdgeVPN.io: Open-source Virtual Private Network for Seamless Edge Computing with Kubernetes. In 2020 IEEE/ACM Symposium on Edge Computing (SEC) (pp. 190-192). San Jose, CA, USA, 2020, pp. 190-192, doi: 10.1109/SEC50012.2020.00032.
- Füstös, F., Péter, K., László, N. -P., Mátis, S. -G., Szabó, Z., & Sulyok, C. (2022). Managing a Kubernetes Cluster on Raspberry Pi Devices. In 2022 IEEE 20th Jubilee International Symposium on Intelligent Systems and Informatics (SISY) (pp. 133-138). doi: 10.1109/SISY56759.2022.10036262.
- Ghorab, A., & St-Hilaire, M. (2022). SDN-Based Service Function Chaining Framework for Kubernetes Cluster Using OvS. In 2022 32nd International

Telecommunication Networks and Applications Conference (ITNAC) (pp. 347-352). doi: 10.1109/ITNAC55475.2022.9998380.

- Gokhale, S., et al. (2021). Creating Helm Charts to ease deployment of Enterprise Application and its related Services in Kubernetes. In 2021 International Conference on Computing, Communication and Green Engineering (CCGE) (pp. 1-5). Pune, India. https://doi.org/10.1109/CCGE50943.2021.9776450
- Gunawardena, T. M., & Jayasena, K. P. N. (2020). Real-Time Uber Data Analysis of Popular Uber Locations in Kubernetes Environment. In 2020 5th International Conference on Information Technology Research (ICITR) (pp. 1-6). Moratuwa, Sri Lanka. https://doi.org/10.1109/ICITR51448.2020.9310851
- Gupta, M., Sanjana, K., Akhilesh, K., & Chowdary, M. N. (2021). Deployment of Multi-Tier Application on Cloud and Continuous Monitoring using Kubernetes. In 2021 5th International Conference on Electrical, Electronics, Communication, Computer Technologies and Optimization Techniques (ICEECCOT) (pp. 602-607). Mysuru, India. https://doi.org/10.1109/ICEECCOT52851.2021.9707957
- Hasan, B. T., & Abdullah, D. B. (2022). Real-Time Resource Monitoring Framework in a Heterogeneous Kubernetes Cluster. In 2022 Muthanna International Conference on Engineering Science and Technology (MICEST) (pp. 184-189). doi: 10.1109/MICEST54286.2022.9790264.
- He, Z. (2020). Novel Container Cloud Elastic Scaling Strategy based on Kubernetes. In 2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC) (pp. 1400-1404). doi: 10.1109/ITOEC49072.2020.9141552.
- Hu, T., & Wang, Y. (2021). A Kubernetes Autoscaler Based on Pod Replicas Prediction. In 2021 Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS) (pp. 238-241). Shenyang, China. https://doi.org/10.1109/ACCTCS52002.2021.00053
- Islam Shamim, M. S., Ahamed Bhuiyan, F., & Rahman, A. (2020). XI Commandments of Kubernetes Security: A Systematization of Knowledge Related to Kubernetes Security Practices. In 2020 IEEE Secure Development (SecDev) (pp. 58-64). Atlanta, GA, USA. https://doi.org/10.1109/SecDev45635.2020.00025

- Kang, R., Zhu, M., He, F., & Oki, E. (2021). Implementation of Virtual Network Function Allocation with Diversity and Redundancy in Kubernetes. In 2021 IFIP Networking Conference (IFIP Networking) (pp. 1-2). Espoo and Helsinki, Finland. https://doi.org/10.23919/IFIPNetworking52078.2021.9472200
- Kang, R., Zhu, M., He, F., Sato, T., & Oki, E. (2021). Design of Scheduler Plugins for Reliable Function Allocation in Kubernetes. In 2021 17th International Conference on the Design of Reliable Communication Networks (DRCN) (pp. 1-3). Milano, Italy. doi: 10.1109/DRCN51631.2021.9477366.
- Karypiadis, E., Nikolakopoulos, A., Marinakis, A., Moulos, V., & Varvarigou, T. (2022). SCAL-E: An Auto Scaling Agent for Optimum Big Data Load Balancing in Kubernetes Environments. In 2022 International Conference on Computer, Information and Telecommunication Systems (CITS) (pp. 1-5). Piraeus, Greece. https://doi.org/10.1109/CITS55221.2022.9832990
- Kayal, P. (2020). Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope : Invited Paper. In 2020 IEEE 6th World Forum on Internet of Things (WF-IoT), New Orleans, LA, USA, 2020, pp. 1-6, doi: 10.1109/WF-IoT48130.2020.9221340.
- Khalel, M. M., Pugazhendhi, M. A., & Raj, G. R. (2022). Enhanced Load Balancing in Kubernetes Cluster By Minikube. In 2022 International Conference on Smart Technologies and Systems for Next Generation Computing (ICSTSN) (pp. 1-5). doi: 10.1109/ICSTSN53084.2022.9761317.
- Kim, C., Kim, R., Kim, G.-Y., & Kim, S. (2021). Kubernetes-based DL Offloading Framework for Optimizing GPU Utilization in Edge Computing. In 2021 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 143-146). Jeju Island, Korea, Republic of. https://doi.org/10.1109/ICTC52510.2021.9621002
- Kim, J., Kim, D., & Lee, J. (2021). Design and Implementation of Kubernetes enabled Federated Learning Platform. In 2021 International Conference on Information and Communication Technology Convergence (ICTC) (pp. 410-412). Jeju Island, Korea, Republic of. https://doi.org/10.1109/ICTC52510.2021.9620986

- Kjorveziroski, V., Mishev, A., & Filiposka, S. (2021). Evaluating IPv6 Support in Kubernetes. In 2021 29th Telecommunications Forum (TELFOR) (pp. 1-4). Belgrade, Serbia. doi: 10.1109/TELFOR52709.2021.9653276.
- Klos, A., Rosenbaum, M., & Schiffmann, W. (2021). Scalable and Highly Available Multi-Objective Neural Architecture Search in Bare Metal Kubernetes Cluster. In 2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) (pp. 605-610). Portland, OR, USA. https://doi.org/10.1109/IPDPSW52791.2021.00094
- Kubernetes Components. (2021a). Kubernetes. https://kubernetes.io/docs/concepts/overview/components/#cluster-level-logging
- Kubernetes Operators. (2021c). Kubernetes. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/
- Kubernetes. StatefulSets. (2021b). Kubernetes. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/
- Leiter, Á., Gago, I., Fernández, A., Vázquez-Poletti, J. L., & Moltó, G. (2022). Intent-based 5G UPF configuration via Kubernetes Operators in the Edge. In 2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN) (pp. 186-189). Barcelona, Spain. doi: 10.1109/ICUFN55119.2022.9829576.
- Lemoine, B. (2022). Energy Efficiency of N:1 Protection Setups with Kubernetes Horizontal Pod Autoscaler. In 2022 25th Conference on Innovation in Clouds, Internet and Networks (ICIN) (pp. 126-130). doi: 10.1109/ICIN53892.2022.9758129.
- Liu, C., Cai, Z., Wang, B., Tang, Z., & Liu, J. (2020). A protocol-independent container network observability analysis system based on eBPF. In 2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS) (pp. 697-702). Hong Kong. https://doi.org/10.1109/ICPADS51040.2020.00099
- Liu, Q., Haihong, E., & Song, M. (2020). The Design of Multi-Metric Load Balancer for Kubernetes. In 2020 International Conference on Inventive Computation Technologies (ICICT) (pp. 1114-1117). Coimbatore, India. doi: 10.1109/ICICT48043.2020.9112373.

- MariaDB Galera packaged by Bitnami. (2021c). Artifact Hub. https://artifacthub.io/packages/helm/bitnami/mariadb-galera
- MariaDB in brief. (2021b). MariaDB Foundation. https://mariadb.org/en/
- Mart, O., Negru, C., Pop, F., & Castiglione, A. (2020). Observability in Kubernetes Cluster: Automatic Anomalies Detection using Prometheus. In 2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS) (pp. 565-570). Yanuca Island, Cuvu, Fiji. doi: 10.1109/HPCC-SmartCity-DSS50907.2020.00071.
- Moravcik, M., Kontsek, M., Segec, P., & Cymbalak, D. (2022). Kubernetes - evolution of virtualization. In 2022 20th International Conference on Emerging eLearning Technologies and Applications (ICETA) (pp. 454-459). doi: 10.1109/ICETA57911.2022.9974681.
- Mubina, B. F., Dewanta, F., & Aditya, B. (2022). Performance Analysis of High Availability Video Conference Service with Kubernetes Cluster Across Data Center. In 2022 2nd International Conference on Electronic and Electrical Engineering and Intelligent System (ICE3IS) (pp. 164-168). doi: 10.1109/ICE3IS56585.2022.10010300.
- Muthanna, M. S. A., & Tselykh, A. (2022). Development of Docker and Kubernetes Orchestration Platforms for Industrial Internet of Things Service Migration. In 2022 International Conference on Modern Network Technologies (MoNeTec) (pp. 1-6). Moscow, Russian Federation. https://doi.org/10.1109/MoNeTec55448.2022.9960769
- MySQL Features. (2021a). MySQL :: MySQL 8.0 Reference Manual :: 1.2.2 The Main Features of MySQL. https://dev.mysql.com/doc/refman/8.0/en/features.html
- MySQL Kubernetes Operator. (2021b). GitHub. https://github.com/mysql/mysql-operator
- MySQL Kubernetes Operator. (2021c). MySQL :: MySQL Operator for Kubernetes Manual :: 1 Introduction. https://dev.mysql.com/doc/mysql-operator/en/
- Nguyen, N. T., & Kim, Y. (2022). A Design of Resource Allocation Structure for Multi-Tenant Services in Kubernetes Cluster. In 2022 27th Asia Pacific Conference

on Communications (APCC) (pp. 651-654). Jeju Island, Korea, Republic of. https://doi.org/10.1109/APCC55198.2022.9943782

- Ning, A. (2022). A Platform of Containers Based on High Availability Kubernetes Cluster. In 2022 International Conference on Applied Physics and Computing (ICAPC) (pp. 278-281). Ottawa, ON, Canada. https://doi.org/10.1109/ICAPC57304.2022.00059
- Park, J., Choi, U., Kum, S., Moon, J., & Lee, K. (2021). Accelerator-Aware Kubernetes Scheduler for DNN Tasks on Edge Computing Environment. In 2021 IEEE/ACM Symposium on Edge Computing (SEC) (pp. 438-440). doi: 10.1145/3453142.3491411.
- PingCAP, (2021a) How to Test TiDB Using Sysbench. (2021). PingCAP Docs. https://docs.pingcap.com/tidb/dev/benchmark-tidb-using-sysbench
- PostgreSQL (2021a) What is PostgreSQL. PostgreSQL. https://www.postgresql.org/about/
- PostgreSQL Architecture. (2021b). educba. https://www.educba.com/what-is-postgresql/
- PostgreSQL HA on Kubernetes. (2021c). Artifact Hub. https://artifacthub.io/packages/helm/bitnami/postgresql-ha
- Reddy, Y. S. D., Reddy, P. S., Ganesan, N., & Thangaraju, B. (2022). Performance Study of Kubernetes Cluster Deployed on Openstack,VMs and BareMetal. In 2022 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT) (pp. 1-5). Bangalore, India. https://doi.org/10.1109/CONECCT55679.2022.9865718
- Reile, C., Chadha, M., Hauner, V., Jindal, A., Hofmann, B., & Gerndt, M. (2022). Bunk8s: Enabling Easy Integration Testing of Microservices in Kubernetes. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp. 459-463). Honolulu, HI, USA. doi: 10.1109/SANER53432.2022.00062.
- Sithiyopasakul, J., Archevapanich, T., Purahong, B., Sithiyopasakul, P., & Benjangkaprasert, C. (2021). Automated Resource Management System based on Kubernetes Technology. In 2021 18th International Conference on Electrical

Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON) (pp. 1146-1149). Chiang Mai, Thailand. https://doi.org/10.1109/ECTI-CON51831.2021.9454911

- Sysbench. (2021). Severalnines. https://severalnines.com/database-blog/how-benchmark-performance-mysql-mariadb-using-sysbench
- TiDB Architecture. (2021d). PingCAP Docs. https://docs.pingcap.com/tidb/stable/tidb-architecture
- TiDB Database. (2021a). DB-Engines. https://dbdb.io/db/tidb
- TiDB on Azure. (2022c). PingCAP Docs. https://docs.pingcap.com/tidb-in-kubernetes/stable/deploy-on-azure-aks
- TiDB Operator on Kubernetes. (2021b). PingCAP Docs. https://docs.pingcap.com/tidb-in-kubernetes/stable/deploy-tidb-operator
- Todorov, M. H. (2022). Deploying Different Lightweight Kubernetes on Raspberry Pi Cluster. In 2022 30th National Conference with International Participation (TELECOM) (pp. 1-4). Sofia, Bulgaria. https://doi.org/10.1109/TELECOM56127.2022.10017262
- Tran, M.-N., Vu, D.-D., & Kim, Y. (2022). A Survey of Autoscaling in Kubernetes. In 2022 Thirteenth International Conference on Ubiquitous and Future Networks (ICUFN) (pp. 263-265). Barcelona, Spain. https://doi.org/10.1109/ICUFN55119.2022.9829572
- Viody, Y., & Kistijantoro, A. I. (2022). Container Migration for Distributed Deep Learning Training Scheduling in Kubernetes. In 2022 9th International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA) (pp. 1-6). Tokoname, Japan. doi: 10.1109/ICAICTA56449.2022.9932961.
- Wang, M., Zhang, D., & Wu, B. (2020). A Cluster Autoscaler Based on Multiple Node Types in Kubernetes. In 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC) (pp. 575-579). Chongqing, China. doi: 10.1109/ITNEC48623.2020.9084706.
- Y.-T. Wang, S.-P. Ma, Y.-J. Lai, & Y.-C. Liang. (2022). Analyzing and Monitoring Kubernetes Microservices based on Distributed Tracing and Service Mesh. In 2022

29th Asia-Pacific Software Engineering Conference (APSEC) (pp. 477-481). doi: 10.1109/APSEC57359.2022.00066.

- Yin, J., Liu, B., & Liu, H. (2022). A user-space virtual device driver framework for Kubernetes. In 2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA) (pp. 202-206). Shenyang, China. https://doi.org/10.1109/ICPECA53709.2022.9718896
- Zerwas, J., Tsoukalas, L., Spiliopoulos, V., Gouglidis, A., & Katos, V. (2022). KAPETÁNIOS: Automated Kubernetes Adaptation through a Digital Twin. In 2022 13th International Conference on Network of the Future (NoF) (pp. 1-3). Ghent, Belgium. https://doi.org/10.1109/NoF55974.2022.9942649
- Zhu, M., Kang, R., He, F., & Oki, E. (2021). Implementation of Backup Resource Management Controller for Reliable Function Allocation in Kubernetes. In 2021 IEEE 7th International Conference on Network Softwarization (NetSoft) (pp. 360-362). Tokyo, Japan. https://doi.org/10.1109/NetSoft51509.2021.9492724

12 Appendices

Appendix A

Sysbench commands to create tables in the database.

TiDB

```
sysbench /usr/share/sysbench/oltp_read_write.lua --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --tables=10 --table-size=1000  prepare
```

MariaDB

```
sysbench /usr/share/sysbench/oltp_read_write.lua --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --tables=10 --table-size=1000  prepare
```

PostgreSQL

```
sysbench /usr/share/sysbench/oltp_read_write.lua --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --tables=10 --table-size=1000  prepare
```

MySQL

```
sysbench /usr/share/sysbench/oltp_read_write.lua --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --db-driver=mysql --tables=10 --table-size=1000  prepare
```

Appendix B

Workload Simulation Using Sysbench on 1,2,4 and 8 CPU threads.

Read only Test.

MySQL

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=1 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=2 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=4 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=8 --report-interval=60 run
```

PostgreSQL

```
sysbench oltp_read_only --tables=10 --table_size=1000  --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=1 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=2 --report-interval=60 run
sysbench oltp_read_only --tables=10 --table_size=1000  --pgsql-host=127.0.0.1 --pgsql-
port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --
db-driver=pgsql --time=120 --threads=4 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --pgsql-host=127.0.0.1 --pgsql-
port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --
db-driver=pgsql --time=120 --threads=8 --report-interval=60 run
```

MariaDB

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-
db=mariadb_sysbench --db-driver=mysql --time=120 --threads=1 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-
db=mariadb_sysbench --db-driver=mysql --time=120 --threads=2 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-
db=mariadb_sysbench --db-driver=mysql --time=120 --threads=4 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-
db=mariadb_sysbench --db-driver=mysql --time=120 --threads=8 --report-interval=60 run
```

TiDB

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-
driver=mysql --time=120 --threads=1 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-
driver=mysql --time=120 --threads=2 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-
driver=mysql --time=120 --threads=4 --report-interval=60 run
```

```
sysbench oltp_read_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-
driver=mysql --time=120 --threads=8 --report-interval=60 run
```

Write only test

MySQL

```
sysbench oltp_write_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900
--threads=1 --report-interval=60 run
```

```
sysbench oltp_write_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900
--threads=2 --report-interval=60 run
```

```
sysbench oltp_write_only --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900
--threads=4 --report-interval=60 run
```

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=8 --report-interval=60 run

PostgreSQL

sysbench oltp_write_only --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=1 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=2 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=4 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=8 --report-interval=60 run

MariaDB

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=1 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=2 --report-interval=60 run sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=4 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=8 --report-interval=60 run

TiDB

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=1 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=2 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=4 --report-interval=60 run

sysbench oltp_write_only --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=8 --report-interval=60 run

Read Write Test

MySQL

```
sysbench oltp_read_write --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=1 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=2 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=4 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000  --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=8 --report-interval=60 run
```

PostgreSQL

```
sysbench oltp_read_write --tables=10 --table_size=1000  --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=1 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000  --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=2 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000  --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=4 --report-interval=60 run
```

sysbench oltp_read_write --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --db-driver=pgsql --time=120 --threads=8 --report-interval=60 run

MariaDB

sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=1 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=2 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=4 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=8 --report-interval=60 run

TiDB

sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=1 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=2 --report-interval=60 run sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=4 --report-interval=60 run

sysbench oltp_read_write --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=8 --report-interval=60 run

Delete Only Test

MySQL

sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=1 --report-interval=60 run

sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=2 --report-interval=60 run

sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=4 --report-interval=60 run

sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=root --mysql-password='password' --mysql-db=mysql --time=900 --threads=8 --report-interval=60 run

PostgreSQL

```
sysbench oltp_delete --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-
port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --
db-driver=pgsql --time=120 --threads=1 --report-interval=60 run
```

```
sysbench oltp_delete --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-
port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --
db-driver=pgsql --time=120 --threads=2 --report-interval=60 run
```

```
sysbench oltp_delete --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-
port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --
db-driver=pgsql --time=120 --threads=4 --report-interval=60 run
```

```
sysbench oltp_delete --tables=10 --table_size=1000 --pgsql-host=127.0.0.1 --pgsql-
port=5432 --pgsql-user=postgres --pgsql-password='uolamCfcVp' --pgsql-db=postgres --
db-driver=pgsql --time=120 --threads=8 --report-interval=60 run
```

MariaDB

```
sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-
db=mariadb_sysbench --db-driver=mysql --time=120 --threads=1 --report-interval=60 run
```

```
sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-
db=mariadb_sysbench --db-driver=mysql --time=120 --threads=2 --report-interval=60 run
```

```
sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-
port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-
db=mariadb_sysbench --db-driver=mysql --time=120 --threads=4 --report-interval=60 run
```

```
sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=3306 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=mariadb_sysbench --db-driver=mysql --time=120 --threads=8 --report-interval=60 run
```

TiDB

```
sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=1 --report-interval=60 run

sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=2 --report-interval=60 run

sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=4 --report-interval=60 run

sysbench oltp_delete --tables=10 --table_size=1000 --mysql-host=127.0.0.1 --mysql-port=4000 --mysql-user=sysbench --mysql-password='sysbench' --mysql-db=sbtest --db-driver=mysql --time=120 --threads=8 --report-interval=60 run
```

Appendix C

Database and User Creation

MariaDB

```
create database mariadb_sysbench;
USE mariadb_sysbench;
CREATE USER 'sysbench' IDENTIFIED BY 'sysbench';
GRANT ALL ON *.* TO 'sysbench';
FLUSH PRIVILEGES;
```

TiDB

```
set global tidb_disable_txn_auto_retry = off;
create database sbtest;
USE sbtest;
CREATE USER 'sysbench'@'localhost' IDENTIFIED BY 'sysbench';
GRANT ALL ON *.* TO 'sysbench'@'localhost';
FLUSH PRIVILEGES;
```

Appendix D

Port Forwarding for Database Connectivity

PostgreSQL

```
kubectl port-forward --namespace default svc/my-release-PostgreSQL-ha-pgpool 5432:5432 &
```

MySQL

```
kubectl port-forward service/mycluster mysql
```

TiDB

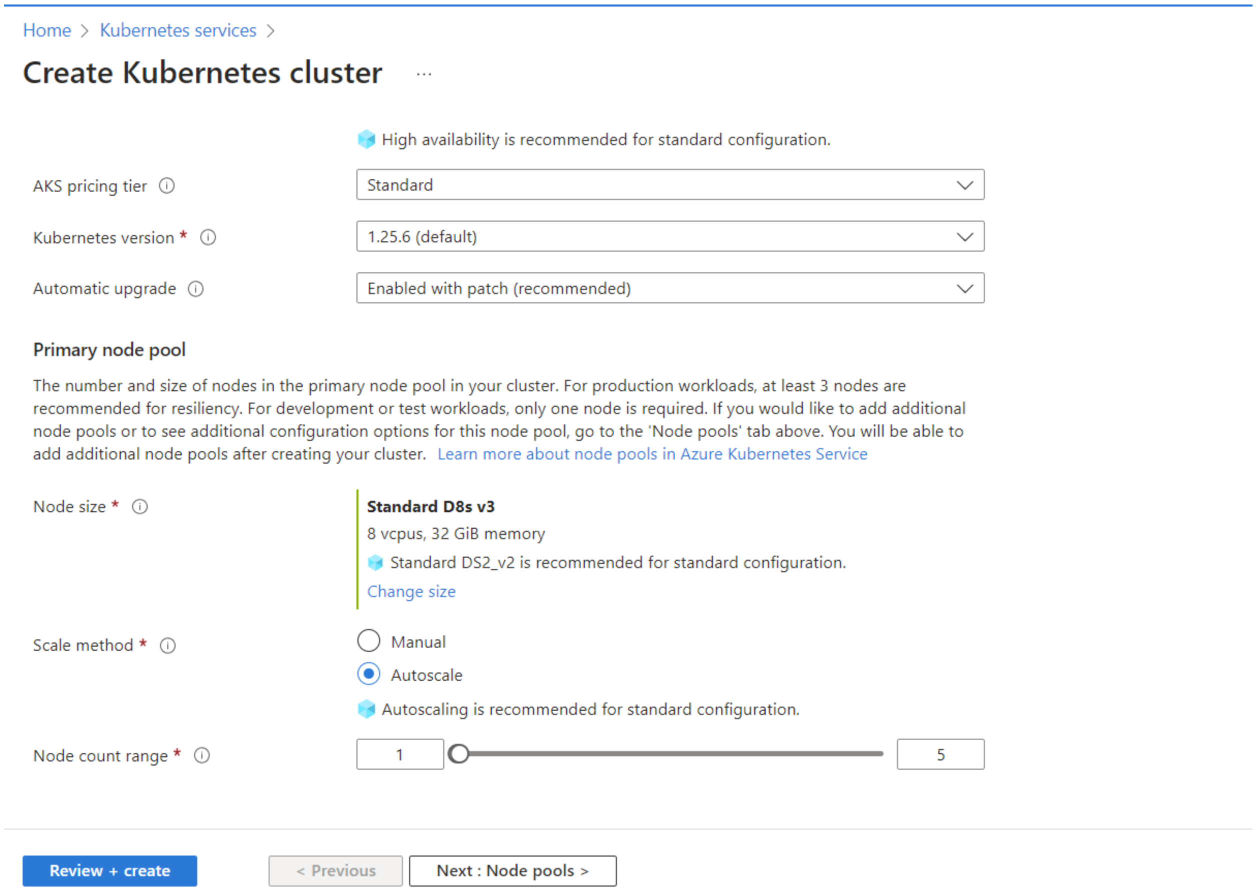
```
kubectl port-forward svc/basic-tidb -n tidb-cluster 4000
```

MariaDB

```
kubectl port-forward --namespace default svc/my-release-mariadb-galera 3306:3306
```

Appendix E

Create Kubernetes Cluster in Azure


Home > Kubernetes services >

Create Kubernetes cluster ...

High availability is recommended for standard configuration.

AKS pricing tier	Standard
Kubernetes version *	1.25.6 (default)
Automatic upgrade	Enabled with patch (recommended)

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. If you would like to add additional node pools or to see additional configuration options for this node pool, go to the 'Node pools' tab above. You will be able to add additional node pools after creating your cluster. Learn more about node pools in Azure Kubernetes Service

Node size * — **Standard D8s v3**
8 vcpus, 32 GiB memory
Standard DS2_v2 is recommended for standard configuration.
Change size

Scale method * — Manual [✓ Autoscale]

Autoscaling is recommended for standard configuration.

Node count range * — 1 to 5

Appendix F

Errors and Issues

segmentation fault for postgres when using 8 threads.

Unable to connect to the server: dial tcp: lookup k8stidb-dns-88fclejb.hcp.westeurope.azmk8s.io on 172.26.0.1:53: no such host.