# NoSQL Databases in Kubernetes

Parth Sandip Mehta
*San Jose State University*

# NoSQL Databases in Kubernetes

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

by

Parth Sandip Mehta

May 2023

The Designated Project Committee Approves the Project Titled

NoSQL Databases in Kubernetes

by

Parth Sandip Mehta

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

MAY 2023

## ABSTRACT

With the increasing popularity of deploying applications in containers, Kubernetes (K8s) has become one of the most accepted container orchestration systems. Kubernetes helps maintain containers smoothly and simplifies DevOps with powerful automations. It was originally developed as a tool to manage stateless microservices that run seamlessly in containers. The ephemeral nature of pods, the smallest deployable unit, in Kubernetes was well-aligned with stateless applications since destroying and recreating pods didn't impact applications. There was a need to provision solutions around stateful workloads like databases so as to take advantage of K8s. This project explores this need, the challenges associated and the available solutions for running databases in Kubernetes. Most of the current research is focused towards SQL-like databases in K8s even though the DNA of NoSQL distributed databases is more aligned with K8s. With no research being done with NoSQL databases, this project outlines the process behind setting up two famous NoSQL databases in K8s: MongoDB and Cassandra. The project also shows a representative viewpoint of the performance comparison between them using the YCSB benchmark. The project lays a foundation around the setup of these databases using K8s Operators and their benchmarking. The goal of the project is to describe the advantages of having databases in K8s, provide developers a clear path for setup and provide insights on basic benchmark performance.

## ACKNOWLEDGEMENTS

I would like to thank my project advisor Dr. Robert Chun for his continuous support throughout this project. I wouldn't have been able to complete this project without his help and understanding. I would also like to thank my committee members, Dr. Ben Reed and Dr. Thomas Austin for taking interest in my project and providing valuable feedback. I would also like to thank Dr. Suneuy Kim for her continuous support, guidance and insight in this project since its inception.

Lastly, I would like to thank my family and friends who have been the most supportive throughout my Master's journey.

## TABLE OF CONTENTS

# LIST OF TABLES

# LIST OF FIGURES

# I. INTRODUCTION

Kubernetes, also called as K8s, is a powerful open-source platform for container orchestration. It was developed by Google to manage their numerous containerized applications running in their huge data centers. Post that, Kubernetes has become the de-facto tool in modern cloud computing and software development  for orchestrating and managing containers. The Cloud Native Computing Foundation (CNCF) now maintains the development of K8s. K8s focuses on automating and simplifying the process of deploying, scaling and managing applications that run in containers. These containers started with primarily being Docker based but as of the latest K8s release, Kubernetes can support multiple container runtimes like containerd and cri-o.

Containers package software applications in a light-weight fashion and hence provide portability and flexibility to run independently of the platform. As modern software becomes more and more complex, the number of containers to manage increases to an extent where managing them becomes time-intensive and complex.

Kubernetes helps developers define the desired state of their application using declarative YAML files. These files contain all the information required to manage the application like the number of replicas, network policies, container image source, etc. Based on this configuration,  Kubernetes ensures that the desired state is maintained and even in failure scenarios, where the containers might fail, Kubernetes redeploys them automatically. Apart from self-healing, it also provides tools for handling automatic load balancing, cluster management and rolling updates. Being platform-agnostic, developers can deploy clusters of containers on their choice of cloud service like AWS, GCP or Azure.

With the advent of microservices architecture, more and more applications are being developed using smaller, modular and loosely coupled services. Kubernetes has been increasingly used to decouple application services from infrastructure and hence making it independent of the physical machines. In Kubernetes, the pods that host these containerized services are prone to being destroyed and recreated, either to heal from a failure state or to replicate to handle more load. Modern applications are almost always stateful. They need to persist data externally and cannot be stateless like the service layer. Hence having stateful applications in Kubernetes pods becomes a challenge as there is a risk of losing data when a pod is recreated.

Traditionally databases have been hosted outside the Kubernetes ecosystem to safeguard them from the ephemeral nature of pods. This DevOps pattern doesn't enable developers to have a single way of managing their stack since most of their stateless workloads are deployed using Kubernetes. Many NoSQL databases have features in-built in them (like elections, replication) that make it more suitable to exist in Kubernetes since dying of database nodes is possible in NoSQL context. A recent survey by CNCF [19] showed that there was a 48% increase in the number of organizations that chose to deploy database workloads in Kubernetes.

This project aims to explore the following questions: What are the major challenges of having a database in Kubernetes? How do existing features of Kubernetes make it easy to run databases in pods? How easy is it to set up NoSQL database systems in Kubernetes? How do NoSQL database systems perform in a Kubernetes environment?

The project report is organized in the following manner: Chapter 2 discusses the Background of this topic which includes surveys of relevant literature, the architecture of Kubernetes and its supported database management tools. Chapter 3 outlines the design and implementation choices of this project. Chapter 4 describes the experiment run on the choice of NoSQL databases: MongoDB and Cassandra and its results. Chapter 5 concludes the report with findings of the project and future work.

## II. BACKGROUND AND RELATED WORK

Container orchestration is needed so that the developer that spins up numerous containers doesn’t need to manage hand-made scripts to keep check on the health of the container as well as the host. As software development transitioned to microservices (instead of monolith) more and more containers were needed and their management became tough for a DevOps engineer. Kubernetes is more popular than its competition like Docker Swarm and Mesos because Swarm doesn’t have advanced features like self-healing, automatic rollouts and rollbacks while Mesos is difficult to setup. In [10], the authors evaluated these container orchestration systems (COS) for their custom workload and found that Kubernetes provided better handling during node failure and better recovery time with network partitions.

Kubernetes architecture includes multiple terminologies, concepts and components that are discussed below:

1. Cluster: A cluster is made up of multiple nodes where each node can either be a physical machine or a VM (virtual machine). Each of these nodes will run K8s and be able to communicate.
2. Nodes: There are primarily 2 types of nodes, Master and Worker.

   The master node of K8s runs the control plane which is responsible for communicating with users and clients, managing the overall state of the cluster, resource management, workload scheduling and config storage. The worker node primarily hosts the workloads in pods. There can be multiple masters in a cluster to support high availability and a master node can also host pods.

3. Pods: They are the smallest deployable unit in K8s and each pod can have multiple containers running within it. Usually a pod consists of the main application container and a couple of sidecar containers like an init container for initial setup or a logging container. Pod is a K8s Resource type.
4. etcd: It is a consistent key value store that stores and acts as the single source of truth for the clusters' information, configurations, desired states of the pods and resides on the master node.
5. API server: This communication point helps the master communicate with workers and also allows developers to interact with the cluster using a command line utility: kubectl. Developers can inspect cluster status, create, update and delete resources like Pods via kubectl.
6. Scheduler: It resides on the master and is responsible for deploying pods in nodes. It tracks the resources of the cluster and schedules pods into nodes that have capacity. It also ensures optimal pod placement so that replicas of pods are running on different nodes for high availability.
7. Controllers: They run watch loops that keep track of configuration files and pushes the state of the cluster from the current state to the desired state outlined in the YAML config files of components.
8. YAML manifests: Kubernetes objects or resources need to be defined and given a desired state. All types of resources are defined using .yaml files that outline

the desired state like the number of replicas, compute resources, network policies, etc. They are also called config files or charts.

9. kubelet: It acts as an agent on each node that communicates node and pods telemetry to the control plane via the API server.
10. ReplicaSet: It is a type of resource definition and a higher-level abstraction which makes sure that a specified number of replicas of a pod are always up and running.
11. Deployments: A higher abstraction that manages the creation and updation of replicasets, pods and other resource types. It allows for rollback of pod versions to revert back easily and also rolling updates to ensure minimal disruption of the application service.
12. Service: A resource type that enables network access to a logical set of pods by providing a stable IP address, DNS name and exposes ways to interact with that set. There are multiple options like ClusterIP (static IP for intra-cluster communication), NodePort (fixed binding between host machine and the cluster) and LoadBalancer.

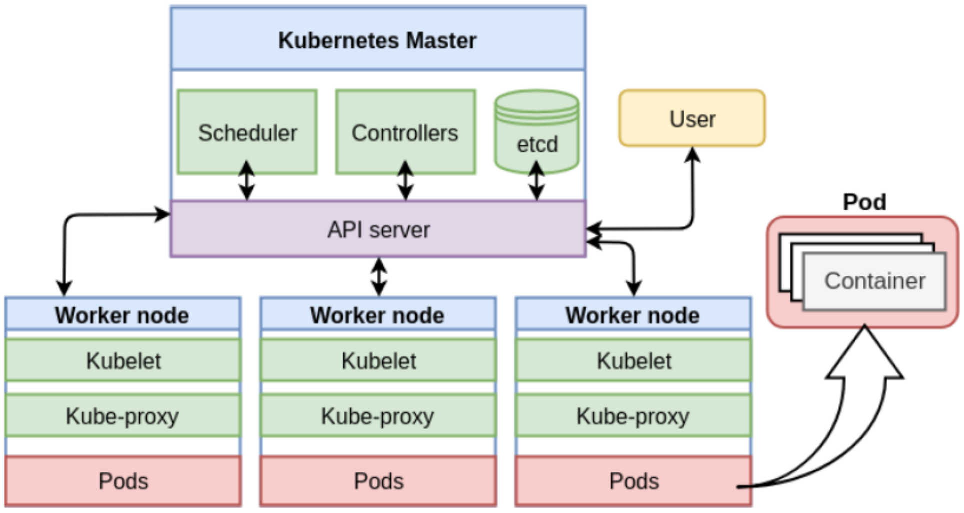


Figure 1: Kubernetes architecture [9]

As Kubernetes became the obvious choice for stateless workloads, a similar solution was needed to handle stateful workloads like databases. The ephemeral nature of pods made it extremely difficult to use pods as database instances. The prospect for databases in K8s was important since it not only enabled a single orchestration interface for DevOps but also gave an easy way of achieving replication and horizontal scalability even in databases that were designed towards single node monoliths. The later versions of K8s incorporated in-built resources that could support persistent storage and stateful workloads.

1. StatefulSet is a special kind of resource that creates and manages a set of pods who stay in order with unique stable network IPs and a fixed mapping to persistent storage volumes. StatefulSet pods have persistent sticky identities and store their state data in a fixed PersistentVolume (PV). After recovering from a

failure, the restarted pod will receive the same identity and will be bound to the same volume as before, hence have access to its state data stored prior to its restart.

2. Persistent Volume: PV is a piece of storage in the cluster whose lifecycle is independent of those of the pods using it. The host filesystem, NFS, AWS EBS are examples of a PV.
3. Persistent Volume Claims (PVC): It is a request for storage made by a pod. A PVC binds the pod to a PV that matches the PVC's characteristics.

Deployment controllers are usually used for managing stateless applications. Kubernetes abstracts the details of storage solutions by providing PVs and PVCs that can be defined in a StatefulSet.

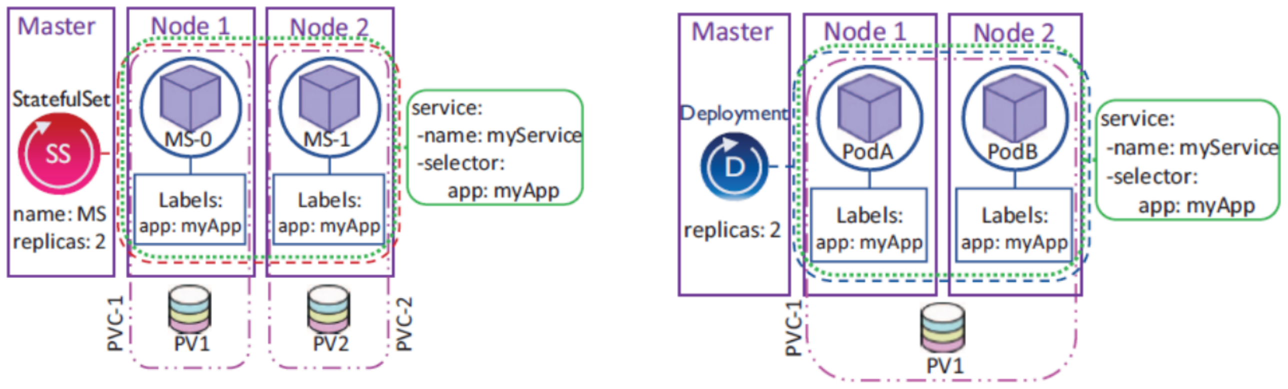


Figure 2: Difference between StatefulSet (left) and Deployment (right) [6]

| StatefulSet | Deployment |
|---|---|
| After restart, the pod retains identity and IP | After restart, the pod gets new identity and new IP |
| Each pod can have its own PV | All pods rely on same shared PV |
| When node shuts down, pod isn't recreated in a different node | When node shuts down, a new pod in a healthy node is created |

Table 1: Differences between pods in StatefulSet and Deployment

The official documentation of K8s [12] mentions that the process of cloning and keeping multiple replicas in sync isn't automatically done by StatefulSet since it is more purposed towards easing the complexity involved in deploying stateful workloads in K8s.

Another challenge that is being faced with databases in Kubernetes is that a lot of db admin tasks warrants manual steps from engineers. In [2], the author discusses the concept of Operators that was introduced by CoreOS. The Operator pattern combines domain-specific knowledge and the existing declarative style of defining state to manage software applications and infrastructure. The author notes that the Operator concept is not limited to Kubernetes or software. A train operator in a steam engine might be manual while one in a bullet train could be automatic. In K8s, it is difficult to manage failure, replication and upgrades in stateful applications but by using the built-in capabilities of K8s such as self-healing and combining those with application (DBMS)-specific complexities and domain knowledge, creates the idea behind the operator pattern. Databases require very specific steps to be performed in a fixed sequence for a conforming upgrade. While vanilla Kubernetes would rely on the simple restarting pods technique to fix issues, for databases the correct error-remediation way would be to have the Operator run specialized upgrade code with alerting.

*"An operator is a Kubernetes controller that understands 2 domains: Kubernetes and something else. By combining knowledge of both domains, it can automate tasks that usually require a human operator that understands both domains"* - Jimmy Zelinskie, CoreOS, company that created the first K8 operator.

A Kubernetes Operator is a software tool that handles complex application specific operations for us. More complex tasks and more number of environments to manage implies more benefits from a software operator over a manual engineer. Operators use custom control loop and Custom Resource Definitions (CRD), which act like custom components designed specifically for the database application.

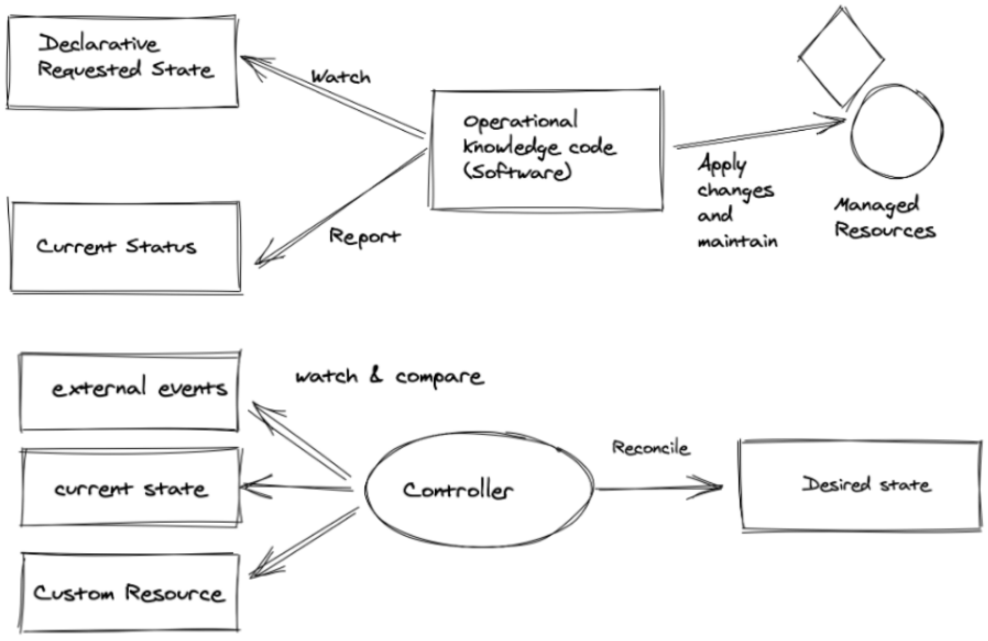


Figure 3: Operator and Custom Controller [13]

In the famous blogpost from Google [4], the author discussed the things to consider when deciding if a database should be run on Kubernetes or not. The author states that databases like ElasticSearch, MongoDB and Cassandra have features like leader election during failures, replication and sharding in-built in its DNA. These choices will be more compatible to run on K8s. The following flowchart guides a user in their decision.

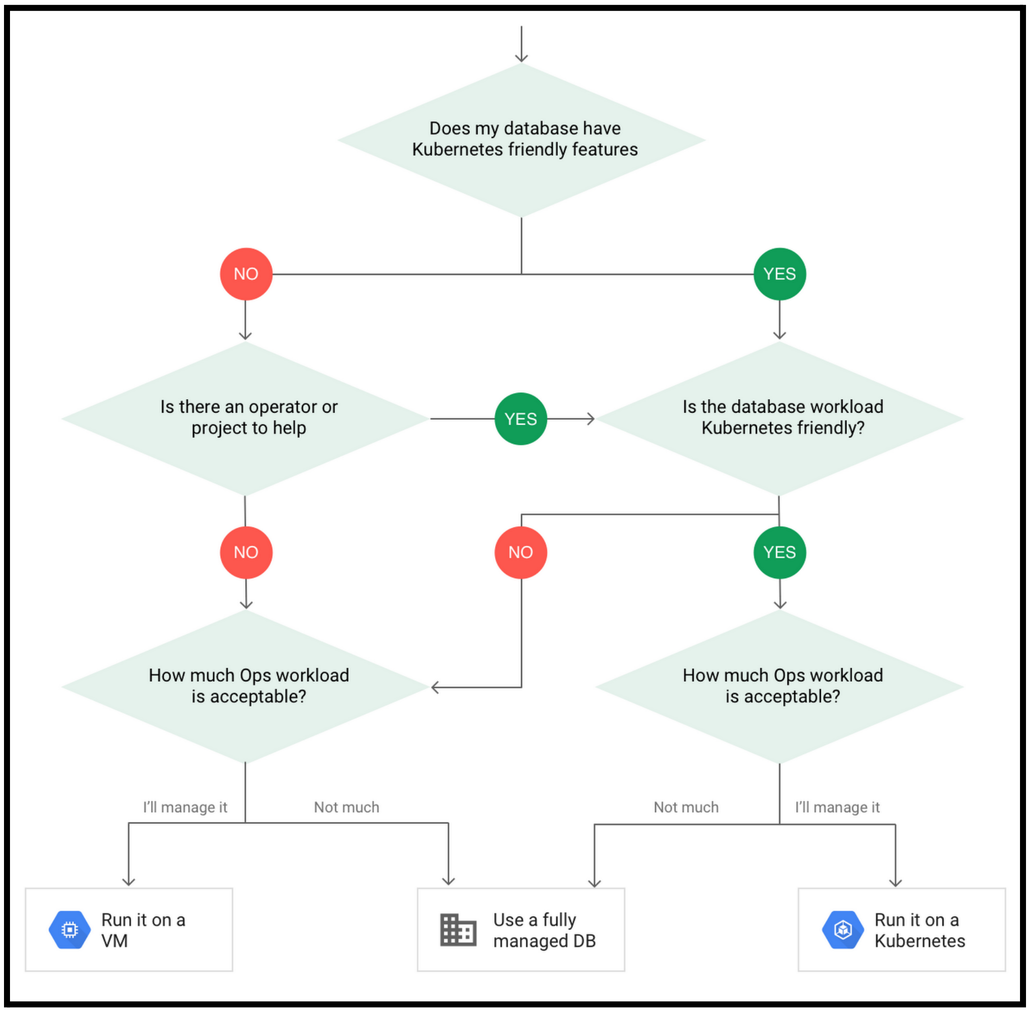

Does my database have Kubernetes friendly features
NO
YES
Is there an operator or project to help
YES
Is the database workload Kubernetes friendly?
NO
NO
YES
How much Ops workload is acceptable?
How much Ops workload is acceptable?
I'll manage it
Not much
Not much
I'll manage it
Run it on a VM
Use a fully managed DB
Run it on a Kubernetes

Figure 4: To run or not to run a database in K8s [4]

Having K8s operators for databases greatly assists in automating DevOps Day 0 tasks like initialization of the database pods, deployment, enabling replication and Day 1 operations like version upgrade, backups and recovery. Without operators, the data synchronization between instances would need to be handled manually. Database operators help save engineer time in an organization and also provide quick steps to setup multiple kinds of databases in an existing K8s cluster. A typical 3 replica set MongoDB setup with sharding, which takes hours when done manually via StatefulSets, is now automated into minutes by the operator. The CRD used by the operator is built on top of the primitive constructs of StatefulSet, PV and PVC.

In both [1] and [3], the authors have considered experimenting on PostgreSQL in Kubernetes. Perera et al. [1] create a machine learning algorithm that helps scale PostgreSQL hosted on Kubernetes. The authors used the Zalando operator [14] to manage the setup. Unusually, the syncing logic between multiple PostgreSQL replicas was manually written even when the operator came packed with the Patroni package that could handle this. Similar to [9] they used Prometheus to gather metrics and Grafana for visualization. In [3], the authors analyzed the benchmarking of PostgreSQL/PostGIS geospatial databases operating on a clustered environment against non-clustered environments. They ran experiments on 3 environments: AWS EC2, AWS RDS and AWS EKS. They observed that the import time was quickest for databases operating in AWS EKS, because of its ability to scale up or down based on resource usage. They concluded that the performance in all 3 were comparable since geospatial queries operating upon indexed attributes involved low computation. While a computationally expensive geospatial query on a large dataset yielded better results in EKS. The research failed to explain their EKS setup which casted doubt on the experiments. Currently, a popular database like PostgreSQL has multiple available operators like Crunchy, Zalando and KubeDB. These also come in-built with the Prometheus operator that deploys pods for Grafana too making monitoring also automated.

A lot of research has been done around SQL-like systems on Kubernetes. In [5], [7] and [8], Spark has been analyzed on Kubernetes. Spark executors were set up in K8s and in bare-metal and benchmarked on WordCount and SQL Join operations in [5]. Surprisingly, the authors conclude that Spark on the bare metal outperforms Spark on Kubernetes because even if executors on Kubernetes are given the same CPU compute power, the bare metal executors are able to utilize more CPU cycles owing to their proximity to the kernel. On the other hand, Spark on Kubernetes had better disk I/O write performance in certain MapReduce stages. They used the open-source tool collectl to monitor usages of standard computing resources like CPU, memory and disk. In [7], authors analyzed SQL-on-Hadoop systems: Apache Hive, Apache Drill, Trino, Apache Spark-SQL on Kubernetes with the TPC-H benchmark. In contrast to [5], the HDFS data was stored outside K8s and the benchmark focused on the scalable query engines. They utilized  three different scale factors (10, 100, and 300 GB) and concluded that Trino was the fastest with Spark being one of the slowest. They voiced similar challenges regarding databases in K8s due to the complexity of deployments, lack of official guidelines and smaller community that has adopted K8s for databases. In [8], the authors have utilized GCP’s Dataproc, which are fully managed Spark and Hadoop clusters, to run analysis on popular Uber locations. Dataproc provides simple configuration to run the workload on Google Kubernetes Engine (GKE) versus on VMs. On contrary to [5], their results showed that in the Kubernetes environment the streaming process had a performance gain along with lower CPU utilization.  Nowadays, each cloud company provides a managed Kubernetes service (EKS in AWS, GKE in GCP and AKS in Azure) which not only ease the cluster setup but also provide monitoring dashboard and alerts.

Similar to [8], Olle Larsson [9] has compared 3 SQL databases: TiDB, MySQL and CockroachDB that are setup using GCP's GKE. In line with [1], they used operators to simplify their setup: TiDB-Operator, Presslabs MySQL Operator and the Couchbase Operator. The author has used SysBench as their synthetic benchmarking tool. The experiment evaluated impact on latency when database specific operations like scale up, scale down, scale out, scale in, version upgrade and backup were performed. They concluded that MySQL performed better since it kept complete copies of the dataset while TiDB and CockroachDB throttled when re-balancing of data between pods was being done. They align with the idea that for databases which have well implemented operators that perform most of the complex operations, the ease of adoption of the database in K8s is high. Sharing the same sentiment, in [6], the authors went a step ahead and created their own custom State controller that enabled high availability for stateful workloads which was lacking in Kubernetes.

Majority of the research related to databases in K8s was focused towards SQL-like systems even though NoSQL databases, being inherently distributed systems, are better tuned for K8s. In [11], a new workbench was created that helped compare different autoscalors. Though they utilized Cassandra on Kubernetes as an example, they didn't focus on the setup of the database or its performance in K8s. Similarly, in [10], MongoDB was compared on multiple COS like Docker Swarm, Mesos and Kubernetes but the focus was more towards the orchestration systems and its features than setup of the database. To answer the research question regarding the ease to set up NoSQL database systems in Kubernetes, this project outlines the process of setting up MongoDB and Cassandra using operators and provides a representative view of how these databases compare in performance when run in K8s.

# III. Design and Implementation

## Operator Choice

Most of the popular SQL, NoSQL and NewSQL databases now have multiple operators as outlined in Table 2.

| Database | Type | Design | # of Operators |
|---|---|---|---|
| Cassandra | NoSQL | P2P | 2 |
| CockroachDB | NewSQL | P2P | 1 |
| Couchbase | NoSQL | P2P | 1 |
| MariaDB | SQL | Leader/Follower | 2 |
| PostgreSQL | SQL | Leader/Follower | 5 |
| MongoDB | NoSQL | Leader/Follower | 3 |
| MySQL | SQL | Leader/Follower | 3 |
| TiDB | NewSQL | P2P | 1 |
| YugabyteDB | NewSQL | P2P | 1 |

Table 2: Different types of databases and their Operator availability

To explore NoSQL databases, this project covers the 2 most popular NoSQL databases: MongoDB and Cassandra. This will provide developers with an understanding of the available options, to deploy them in Kubernetes and an overview on their performance. For the choice of operator for MongoDB, the following operators were explored:

| Feature | MongoDB Community Operator | Percona MongoDB Operator | KubeDB Community operator | OpsTree operator |
|---|---|---|---|---|
| Replication | ✅ | ✅ | ✅ | ✅ |
| Sharding | ❌ | ✅ | ✅ | ❌ |
| Scaling | ✅ | ✅ | ❌ | ✅ |
| Automatic Database Upgrade | ❌ | ✅ | ❌ | ❌ |
| Monitoring | ✅ | ✅ | ✅ | ✅ |

Table 3: Feature analysis between different MongoDB Operators

Based on these, the Percona MongoDB [14] provides the most features for free and was chosen for this project. Even though the official MongoDB Operator would be more assuring, the Community version has limited features while the Percona operator provides all the features that the Enterprise edition of the official Operator provides.

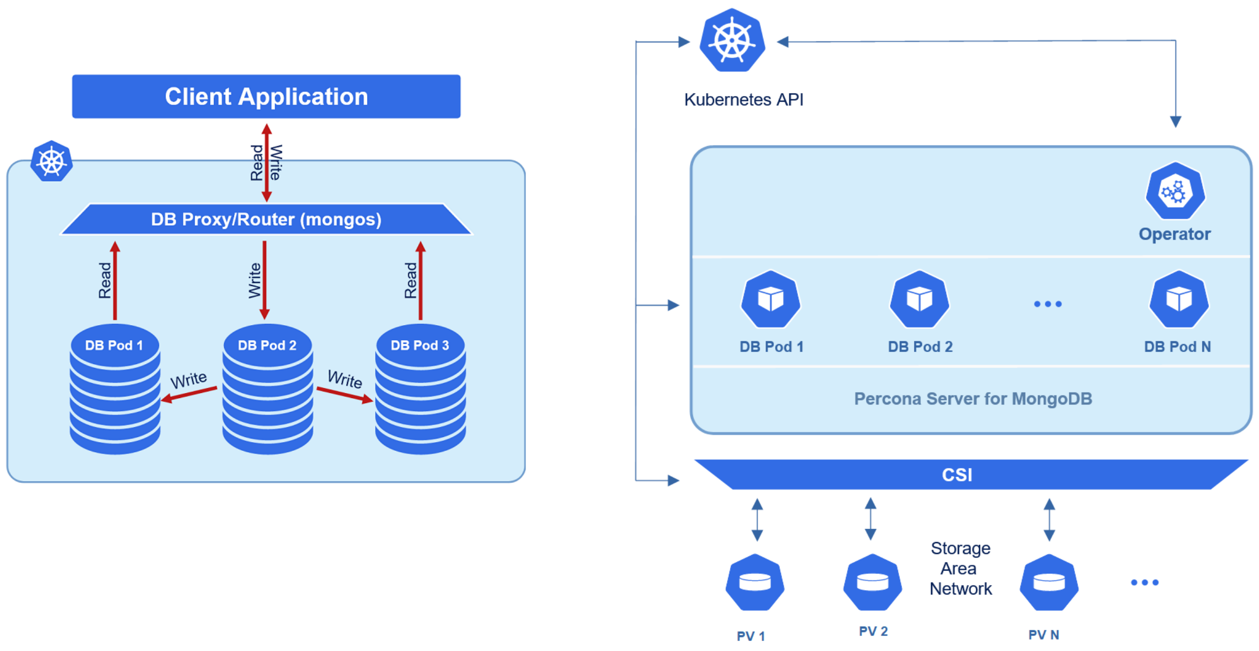
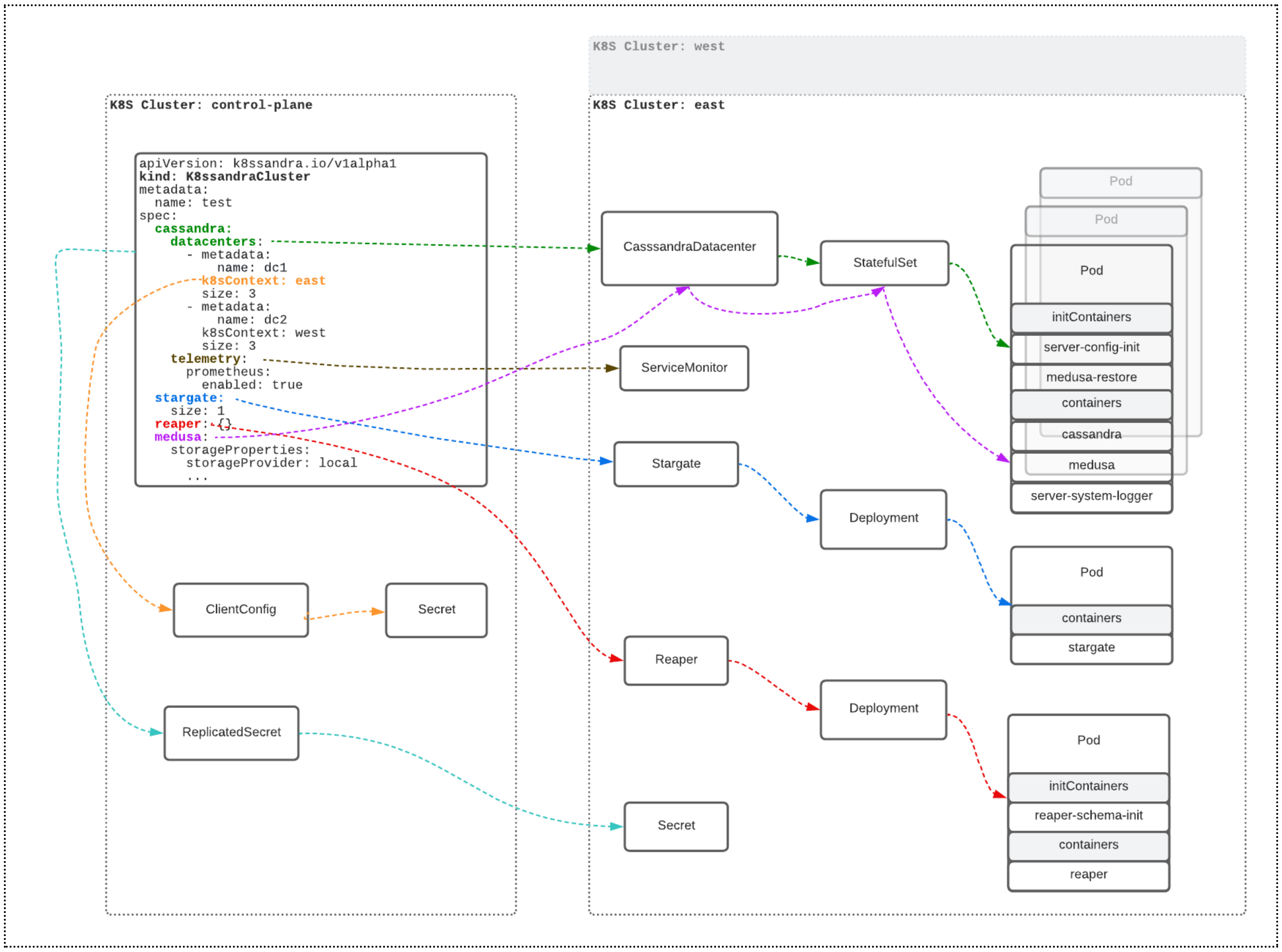


Figure 5: A sharded MongoDB cluster (left) and Kubernetes architecture with Percona Operator [14]

For the choice of operator for Cassandra, 2 operators were compared: K8ssandra-operator (also called simply K8ssandra) and CassKop Operator. Both of these operators provide the most commonly needed features for deployment and even advanced options like live backups and upgrades. The CassKop operator lacks in terms of scalability where one instance of the operator can run only in one namespace while a single K8ssandra can run multiple clusters in multiple namespaces. K8ssandra is also the official open-sourced project of the Apache Cassandra community and offers better community support. K8ssandra also provides an ecosystem of common Cassandra related tools bundled together: Stargate (data gateway that supports REST, CQL, GraphQL), Prometheus, (metrics monitoring) and Medusa (backup and recovery). Hence the official K8ssandra operator was chosen for this project [15].



```
apiVersion: k8ssandra.io/v1alpha1
kind: K8ssandraCluster
metadata:
  name: test
spec:
  cassandra:
    datacenters:
      - metadata:
          name: dc1
        k8sContext: east
        size: 3
      - metadata:
          name: dc2
        k8sContext: west
        size: 3
    telemetry:
      prometheus:
        enabled: true
  stargate:
    size: 1
  reaper: {}
  medusa:
    storageProperties:
      storageProvider: local
      ...
```

Figure 6: Generic K8ssandra Architecture [15]

## Benchmark

Existing literature has used various benchmarks like SysBench [9], TPC-H [7] and MapReduce WordCount [5]. TPC-H is an OLAP workload that is a successor of the TPC-C benchmark. The TPC benchmarks, being SQL and row-like, have been modified to work on NoSQL databases like MongoDB but that wouldn't be the idle scenario. For this project, the Yahoo! Cloud Serving Benchmark (YCSB) was used. YCSB [18] creates synthetic workloads that simulate real-world usage patterns, such as read and write operations, and measures database performance under various levels of concurrency and dataset load. YCSB has been used on MongoDB and Cassandra in [11], [16], [17], [20] and [21]. YCSB provides a simple command line utility with multiple parameters that can be tweaked to generate new benchmarking scenarios. YCSB [18] includes the following Core workloads:

- Workload A: Update heavy workload which has a mix of 50/50 reads and writes.
- Workload B: Read mostly workload which has a mix of 95/5 reads and writes.
- Workload C: Read only
- Workload D: Read latest workload; new records are inserted, and then they are read
- Workload E: Short ranges of records are queried, instead of individual records.
- Workload F: Read-modify-write workload will read a record, modify it, and write back the changes.

The benchmark also offers the following configuration parameters to generate a flexible multi-dimensional workload:

- executiontime: Runtime of the workload (in minutes)
- threadcount: Number of parallel threads
- recordcount: Number of initial records
- operationcount: number of operations (default = 1000)
- readproportion: read portion of the workload (0 - 1)
- writeproportion: write portion of the workload (0 - 1)

- updateproportion: Update portion of the workload (0 - 1)
- requestdistribution: uniform vs zipfian (some rows have more probability to be targeted by reads so as to generate a hotspot)

YCSB benchmarking is divided into 2 phases: the load phase and the run phase. The load phase is common for all the Core Workloads where records are added into the database using the Java driver for the target database. The database needs to be prepared beforehand in terms of empty table creation or keyspace creation or empty collection creation before the load phase can start. A single table `*usertable*` is used for the INSERT operations in the load phase. The schema of this table consists of a `*user_id*` which acts as a primary key or index key and 10 other string fields that are filled with random characters.

## Infrastructure

AWS was used as the cloud provider of choice. There are 2 main ways to run K8s experiments on AWS: EKS cluster (fully managed service) and running one's own K8s cluster on EC2. Since this project aims to provide a representative viewpoint of databases in Kubernetes, the cheaper option of EC2 was chosen. The EC2 instance that runs YCSB is kept separate, though within the same availability zone of the EC2 that hosts the K8s cluster. Many lightweight Kuberentes distributions are available that make setting up clusters on lower compute resources VMs easier. These distributions abstract out the complexity of setting up and maintaining K8s clusters. As part of this project, microk8s, k3s, kind and minikube were explored. kind stands for Kubernetes in Docker and along with minikube, these distributions spin up a cluster inside a Docker container. Both the operators of choice have compatibility with these 2 but weren't used because of the extra nested layer created by the Docker container. The external YCSB query will have to go from one EC2 to another, tunnel into the Docker container and then connect into the open port of the pod. Among k3s and microK8s, the latter was chosen since it provided a single command installation and also provided addons (plugins) for additional packages like dns, storage provisioning, helm and prometheus that could be installed easily.

## Monitoring and Metric Collection

As part of the experiments, Prometheus and Grafana were used to monitor metrics like CPU, memory and disk utilization. Prometheus is an open-source systems monitoring tool that collects time-series data. PromQL, its own query language, can be used to manipulate and retrieve time-series data of metrics. Prometheus is based on a pull mechanism where it scrapes data from multiple sources rather than relying on the source to push data. Prometheus has built-in support for service discovery and can easily detect new pods, nodes, namespaces, PVs in Kubernetes.

Grafana is a powerful visualization tool used to generate custom dashboards and alerting. Grafana can have multiple data sources with the most prominent being Prometheus. It allows for loading pre-existing templates of dashboards that can be used to monitor time series based graphs of metrics. The in-built plugins in microk8s made Prometheus and Grafana stack the obvious choice for the monitoring solution.

## Architecture

K8ssandra operator was used to set up a single Cassandra datacenter with 3 Cassandra instances (3 replicas). Separate pods were created to run Prometheus and Grafana. Their endpoints were exposed publicly so that the dashboards can be viewed from any browser. The benchmark operated on a single keyspace *`ycsb`* in the Cassandra cluster and ran operations on the *`usertable`*.

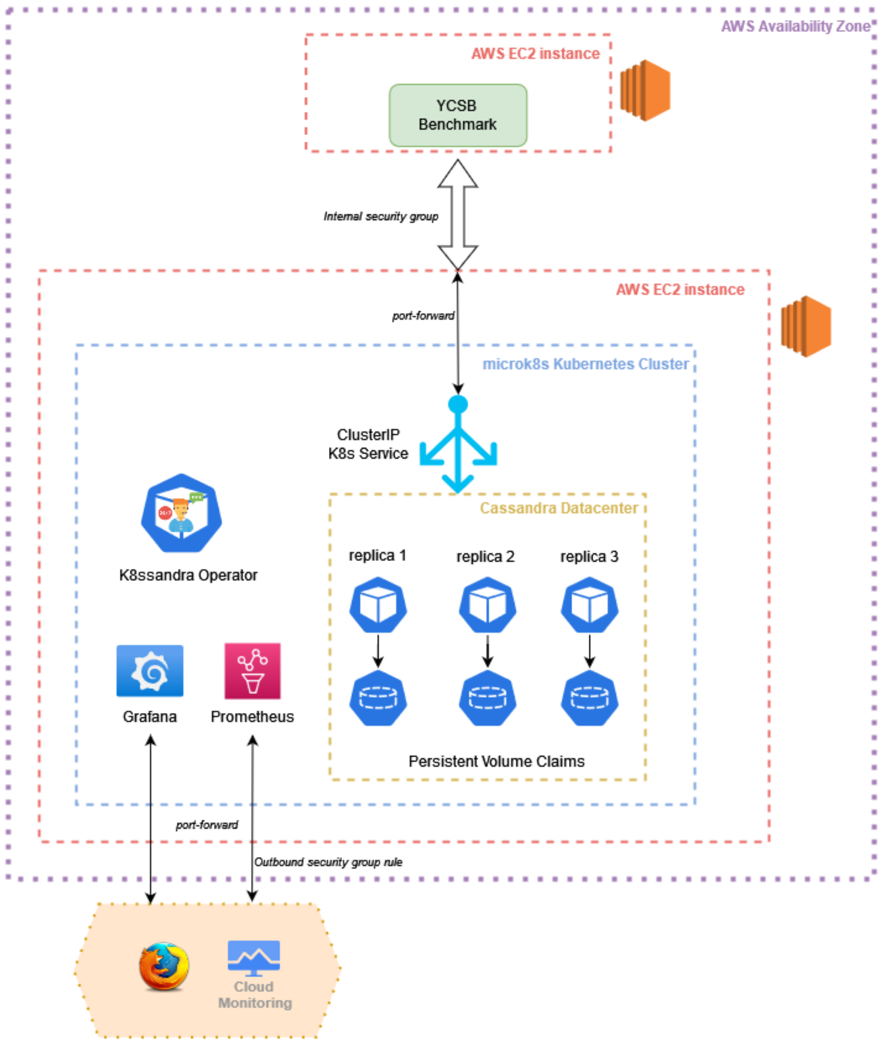


Figure 7:  Cassandra Architecture

The Percona MongoDB operator was used to set up a MongoDB cluster with 3 replicas of each of the 3 shards. A single mongos instance ran with a replica-set of 3 config servers. Each of the components ran in its separate pod. A database named `*ycsb*` and an empty collection named `*usertable*` was needed for the benchmark.

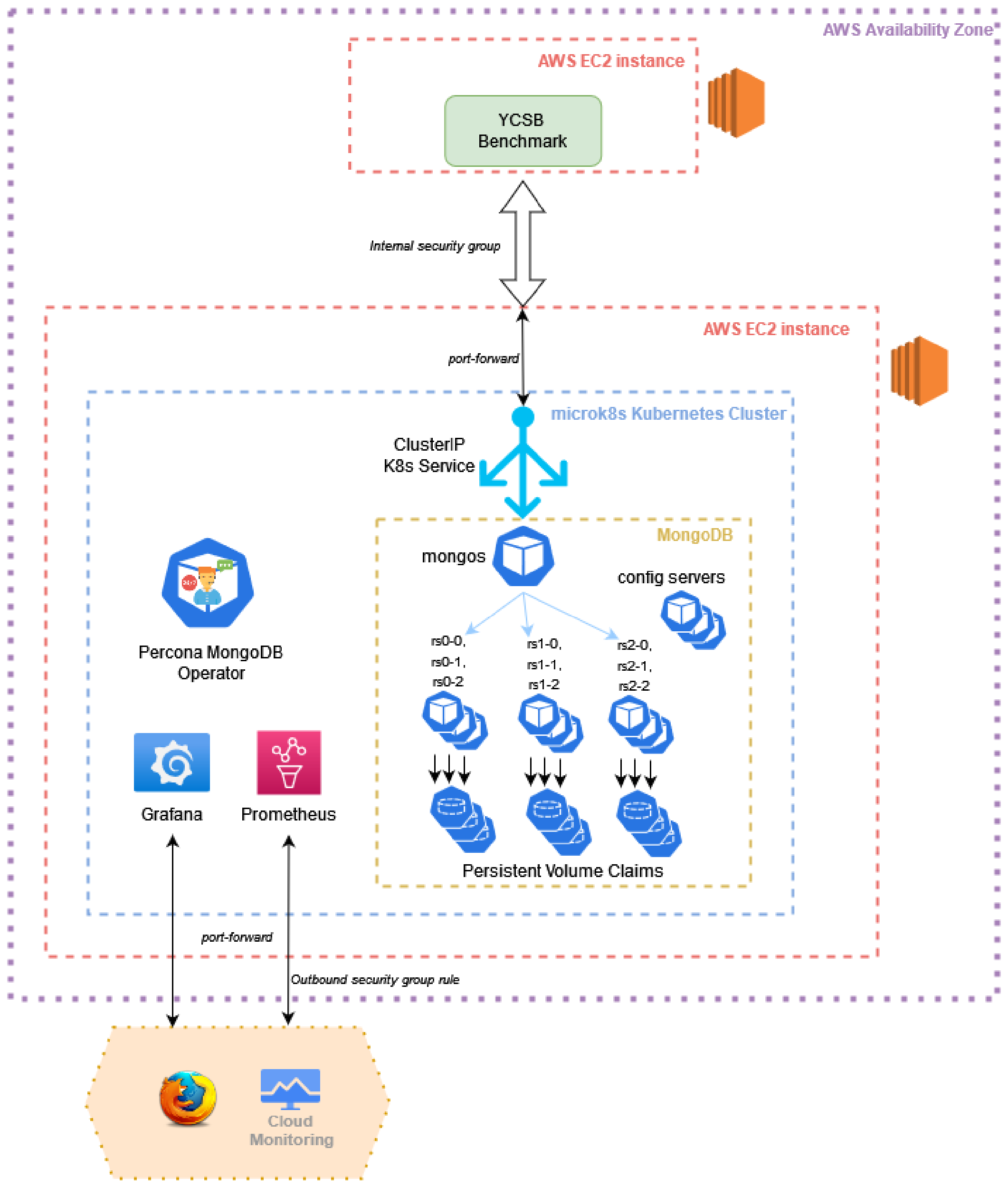


Figure 8: MongoDB Architecture

# IV. Experiments and Results

The experiments are run in AWS EC2 virtual machines where 1 EC2 will host all the database pods inside a single node Kubernetes cluster. Since the focus is not on high availability, the K8s cluster has only 1 node but each replica of the database is housed in a separate pod. Another EC2 instance in the same availability zone runs the YCSB client benchmark. To increase the load on the database system, the client needs to spawn multiple threads, each performing operations on the database.

| EC2 | type | vCPU | RAM | Storage disk (EBS) |
|---|---|---|---|---|
| Kubernetes + DB | t2.xlarge | 4 | 16 GB | 64 GB |
| YCSB Client | t2.2xlarge | 8 | 32 GB | 32GB |

Table 4: Hardware configurations of EC2 instances

Since the client needs to increase the load by spawning more threads, an even more powerful machine with 8 vCPU was used. WorkloadA and WorkloadC of the Core YCSB workloads were utilized in this project. Before each of the workload Run phase, the database was populated with 100,000 records in the Load phase. In each experiment run, a fixed number of operations were performed: 50,000 as a standard. Each workload benchmark consisted of multiple experiments in which the database throughput was measured against an increasing number of YCSB threads. The experiments ended when increasing the number of threads on the client side did not yield a gain in throughput. This indicates the saturation point of the database in terms of handling the workload. All experiments utilized the zipfian distribution so as to emulate a real world application that is prone to hotspots. The experiments were run each time with a different *`-threads`* parameter to increase the simultaneous load on the database.

## Data Load

The load phase for each of the workloads involved populating MongoDB and Cassandra with 100,000 records. Additional steps were performed on the MongoDB setup to enable sharding prior to loading the data. It was observed that MongoDB took 3.7 times more time to finish loading the data as compared to Cassandra.

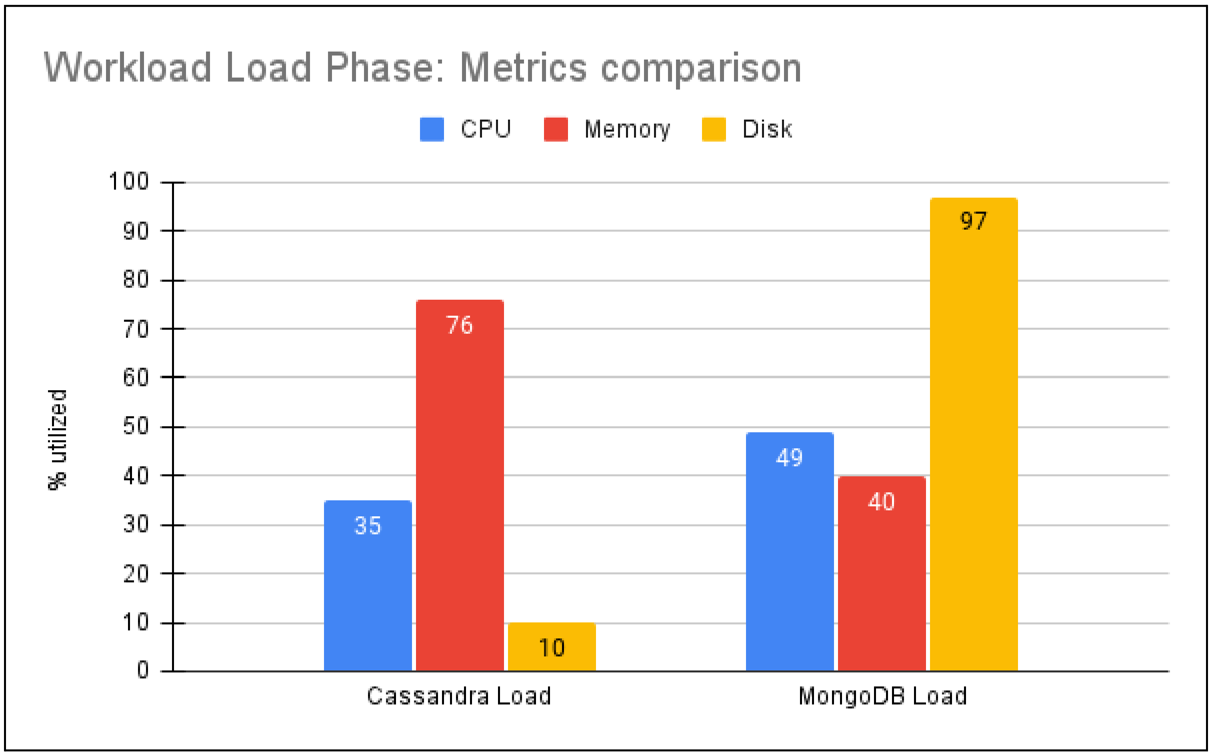


Figure 9: Metrics comparison during load phase

The average throughput during the load phase was 493 ops/s for Cassandra with an average latency of 1.9 ms per INSERT operation and 203 ops/s for MongoDB with 7.2 ms average latency. It was observed that MongoDB's disk utilization spiked to 97% while handling insert which gave it a lower max throughput value. Cassandra used less CPU resources than MongoDB but consumed 1.9 times more memory. Better write performance by Cassandra can be attributed to its peer to peer architecture where each node can handle writes in contrast to MongoDB's single master that handles write operation like INSERT. Cassandra's memory based storage model that utilizes SSTables, makes writes quicker than MongoDB which had to perform more writes to the disk which in turn gave a higher disk utilization percentage.

## Workload C: 100% Read

The workload C of YCSB performs 100% READ operations on the databases. Unlike [17], the results of this experiment showed that Cassandra gave a higher throughput than MongoDB and was also able to handle more threads.

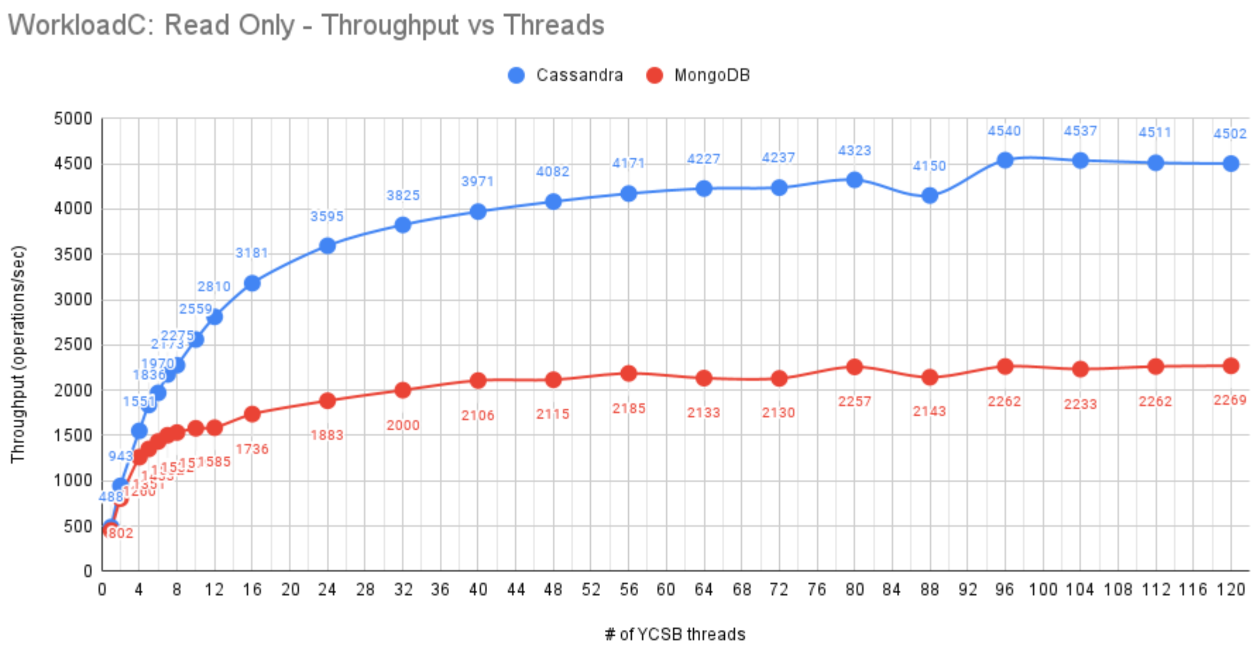


Figure 10: WorkloadC: Throughput vs Threads

The considerably higher throughput of 4540 ops/s of Cassandra could be because of the database design which is suited for higher concurrency READ operations. In [16] and [21], the authors observe a similar trend when YCSB WorkloadC was run on MongoDB and Cassandra setup on bare-metal. By having comparable results, the knowledge gap of performance comparison between databases set up in bare-metal and in Kubernetes was bridged in this project.

## Workload A: 50% Read and 50% Write

The workload A of YCSB has an equal split between READ and UPDATE operations. Similar to the results of the load phase, Cassandra showed better throughput and lower latency than MongoDB. Even this could be attributed to Cassandra's architecture that supports multiple masters that can perform writes and its ability to perform in-memory writes. Figure 11 shows that while MongoDB saturated after 120 concurrent threads of YCSB, Cassandra was able to handle more load with an increase in throughput up till 160 threads. The throughput of both the databases fell when compared with results of workload C. This is because the 50% of write operations took more time than the read operations. In contrast to their findings with workload C, the authors of [17] also found that Cassandra outperformed MongoDB in workload A. The results of Figure 11 corroborate with the results in [16], [20] and [21] where Cassandra gave lower latency with the YCSB workload A.

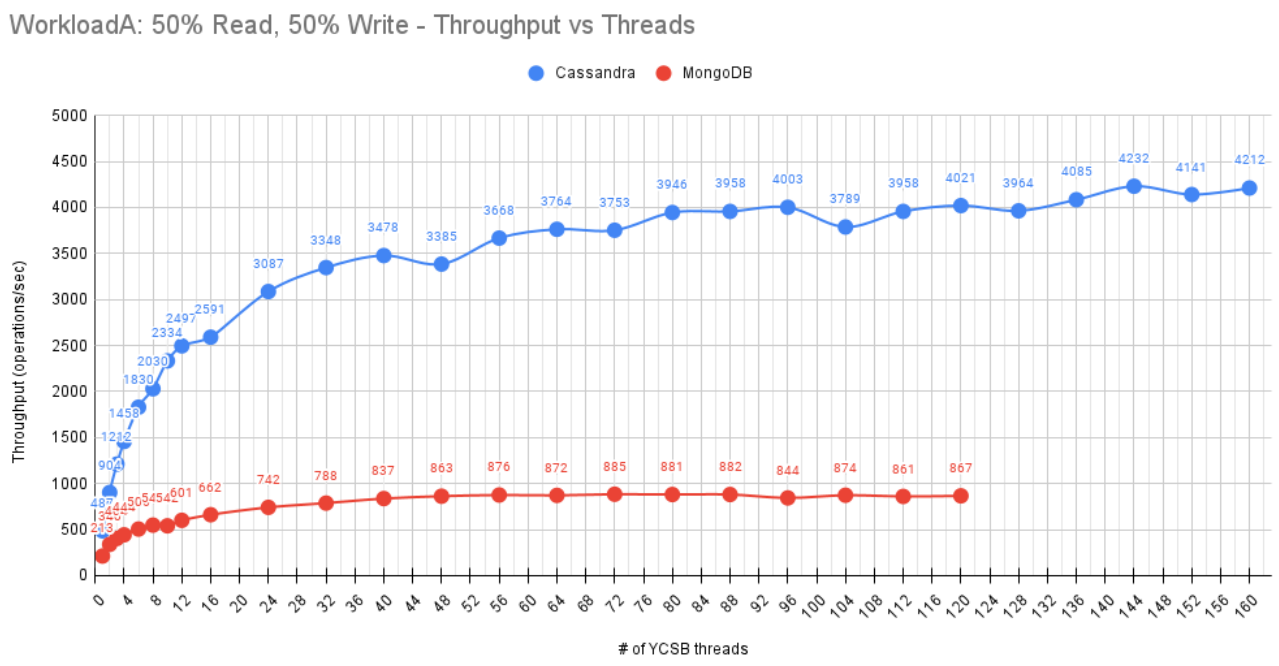


Figure 11: WorkloadA: Throughput vs Threads

## Metrics Results

It was observed that MongoDB metrics steadily increased as the number of threads increased unlike with workload C where the max CPU utilization was 30%. Figure 12 shows that due to the write-intensive nature of this workload, the system resources consumed by MongoDB increased steadily until it reached a saturation point. At that point, the max throughput was observed for MongoDB. Also, the disk utilization was over 93% for the majority of the run, similar to the load phase, due to the frequent flushes of the oplog of MongoDB to the disk. MongoDB does this to survive unexpected crashes but this limits the performance of write operations.

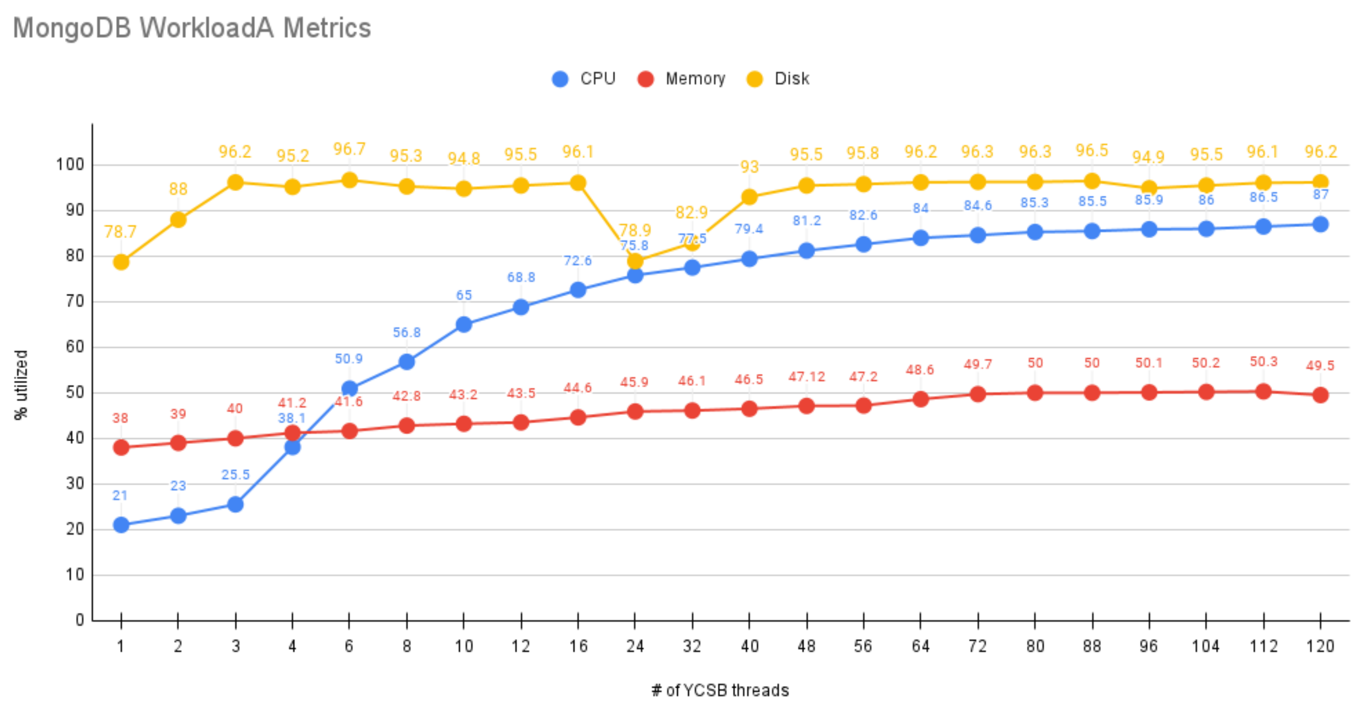


Figure 12: MongoDB WorkloadA Metrics

The average latency observed by the READ and UPDATE operations is shown in Figure 13. It is observed that the MongoDB UPDATE operation has the highest latency while Cassandra READs in workload C have the lowest latency. Interestingly, both READ and UPDATE operations in workload A for Cassandra have similar latency vs threads trajectory. Due to this, their line plot overlaps in Figure 13. It is observed that all Cassandra operations are faster than MongoDB operations which furthers the hypothesis of better performance of Cassandra over MongoDB in Kubernetes.

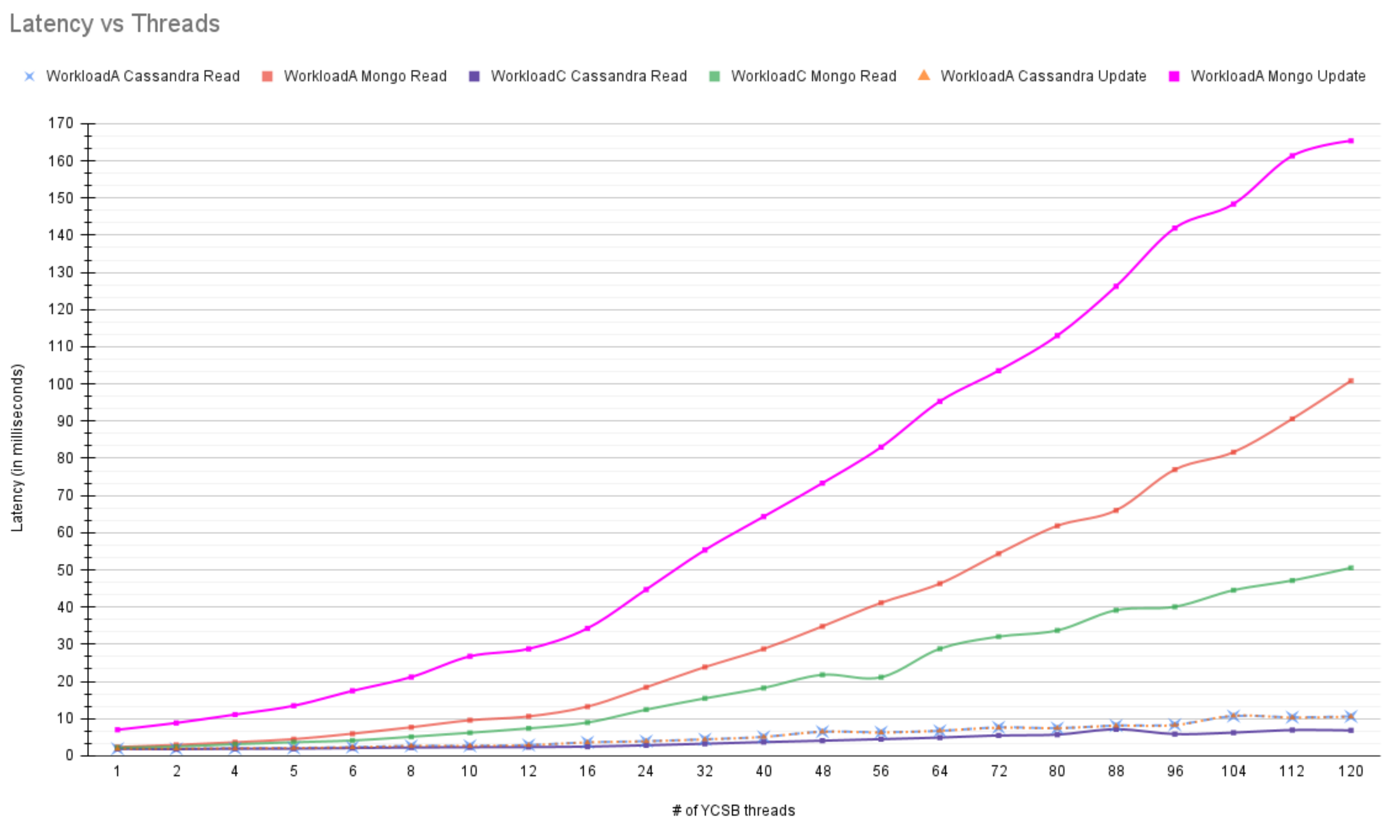


Figure 13: Latency vs Threads

The metrics CPU, memory and disk are compared across workloads in Figure 14. It is seen that there is marginal increase in resource utilization in Cassandra when moving from a read-intensive workload to a write-intensive workload. Whereas, MongoDB sees a jump from 30% CPU to 87% CPU at its max throughput point. MongoDB's disk usage almost maxed off at 96% during write operations. Cassandra used similar % of memory throughout the experiments which could be because of its constant reliance on memory for its operations. MongoDB was able to run at almost half the RAM that was needed by Cassandra.

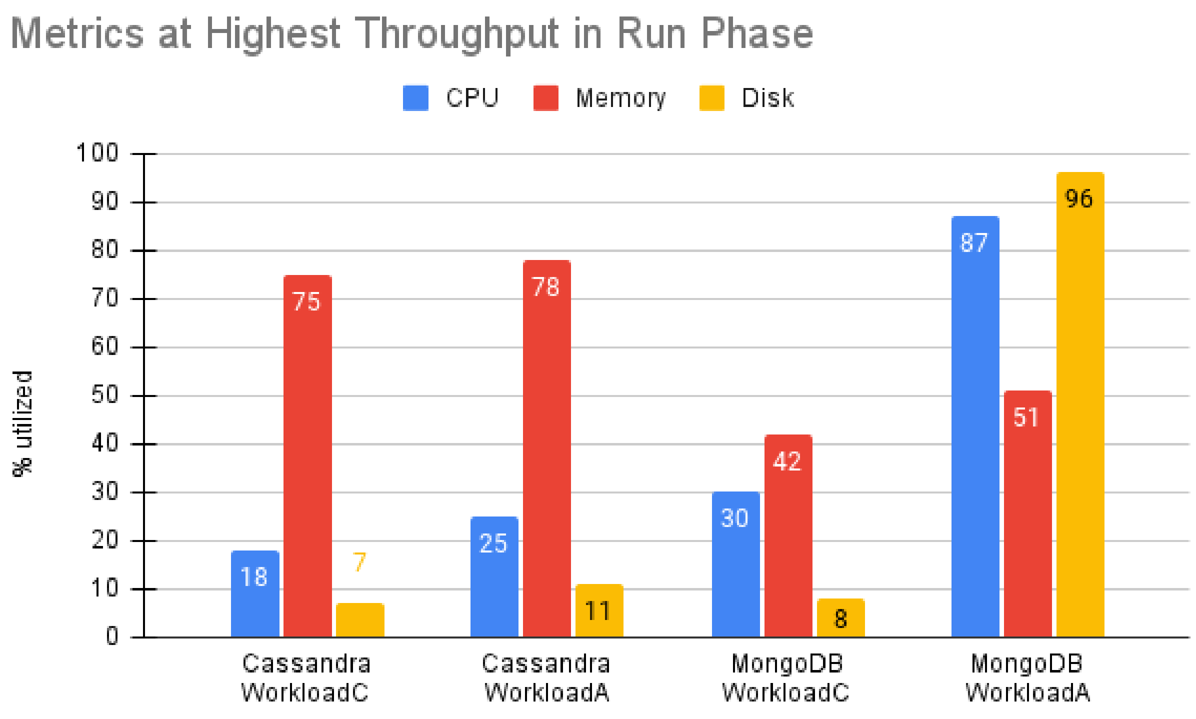


Figure 14: Metrics at highest throughput in run phase

Figure 15 shows a screenshot of disk utilization over time from the Grafana dashboard. It shows occasional spikes in Cassandra Kubernetes cluster's disk utilization. This spike every 5 minutes can be attributed to the fact that Cassandra performs the majority of its operations in-memory and writes them to disk in batches to increase performance.

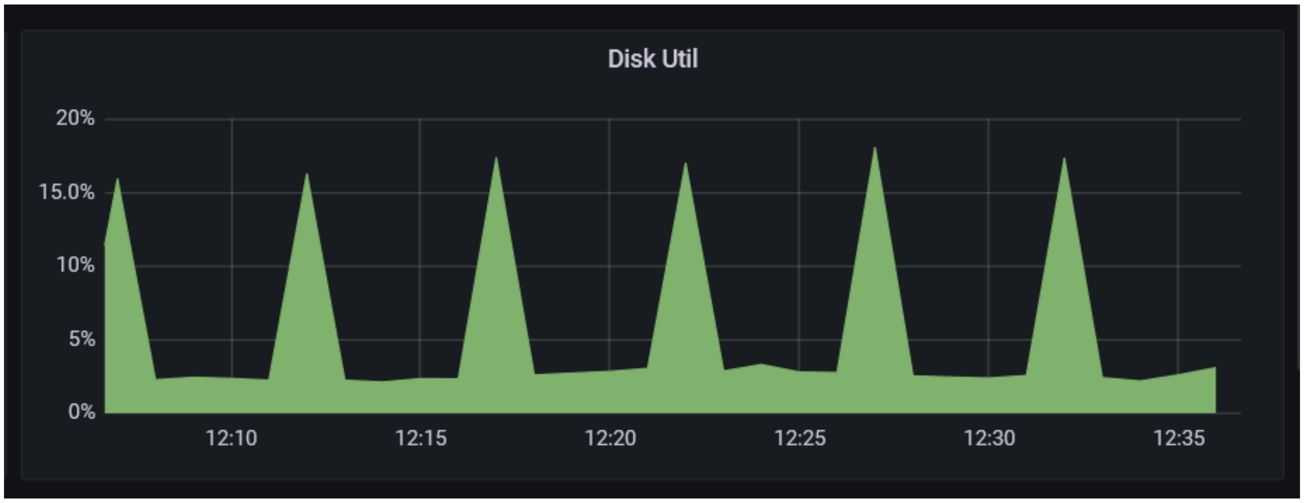


Figure 15: Disk utilization spikes in Cassandra

Even though Cassandra in Kubernetes outperformed MongoDB in throughput, latency, CPU % and disk utilization %, MongoDB was able to run at a lower memory footprint. During the database setup, it was observed that the K8ssandra operator was not able to deploy pods in hardware where the number of vCPUs were less than 3. This suggests that MongoDB is a better option when deploying in a Kubernetes cluster with less resources since it uses less RAM as well as CPU for setup. The benchmark results suggest that master-master database architecture like Cassandra are well suited to handle heavier loads of both reads and writes while master-follower systems like MongoDB are prone to saturate due to a bottleneck at the master. The experiments successfully provided a representational overview of how these NoSQL databases perform in Kubernetes and lays down a foundation for future heavier workloads on multi-node Kubernetes clusters.

## V. Conclusion and Future Work

The biggest challenge for having stateful workloads in Kubernetes is the ephemeral nature of pods. To handle this, Kubernetes provides a host of in-built components to handle stateful applications like databases. By using StatefulSets, PersistentVolumes and PersistentVolumeClaim, database pods can receive a sticky identity that helps them bound back to their original data volume and survive pod crashes and restarts. Even though these provisions help deploy databases in Kubernetes, it doesn't help operate it automatically. The setup and maintenance of the database in Kubernetes is still manual since these components do not have any domain specific knowledge. Kubernetes Operators help bridge this gap by automating Day 0 and Day 1 DevOps tasks related to databases. Operators act like extensions that automatically manage the database and abstracts out the operational complexities that would otherwise be handled by a database administrator with expertise in Kubernetes.

By having databases in Kubernetes, one can also benefit from having optimal allocation of hardware resources for each of the databases deployed. Managing multiple databases manually on multiple nodes leaves out resources that could otherwise be optimally scheduled by Kubernetes. Databases in pods can enable efficient moving of pods to nodes that have capacity which gives maximum utilization of resources. Kubernetes manages and fine tunes resource requests and limits of each pod and node. Kubernetes also provides elasticity and autoscaling capability via its vertical and horizontal pod autoscaler. Organizations also have the option of moving their databases from on-premise to public cloud on the fly using Kubernetes to adjust cloud computing bills. Kubernetes acts like a standardized infrastructure as a code layer that can help move workloads around multiple clouds. Database deployments in Kubernetes using operators help DevOps set policies for auto-scaling and resiliency and have the operator automate it rather than manually reacting to failure in production. By using helm-charts, pre-packaged K8s yaml files, customization based on environment, region and policies can be done that automates the recreation of the entire stack (database, query layer, monitoring tools, backup solutions, etc) in any Kubernetes cluster.

Existing literature covers performance of various SQL-like databases in Kubernetes but doesn't explore NoSQL databases that are better aligned with Kubernetes ideology. The project was able to explore the available operators for MongoDB and Cassandra and demonstrate the simplification in terms of setup that is provided by Percona MongoDB operator and K8ssandra operator. Benchmarking these database setups in Kubernetes with YCSB provided a foundation for comparisons between NoSQL systems in Kubernetes. The experiment results on throughput and latency align with existing results of comparing Cassandra and MongoDB on bare-metal. Metrics like CPU, memory and disk utilization were captured by Prometheus and visualized in Grafana to compare the 2 databases.

The project aimed to provide a base on which future complex experiments can be run to benchmark MongoDB and Cassandra in different setups. Further research can be done in terms of running different Core workloads of YCSB, varying the amount of data by having more records and running for a larger operation count. The base Kubernetes cluster can also be varied to get results in multi-node Kubernetes clusters. The hardware configuration of EC2s can be increased and the performance of the databases can be plotted against memory and CPU. Future experiments could also benchmark and compare performance of NoSQL systems in different managed Kubernetes services from AWS, GCP and Azure.

As more and more developers explore container technologies, Kubernetes will become more prominent as an orchestration tool. With DevOps moving towards automation, more stateful workloads like databases will be deployed alongside stateless microservices to provide a unified source for application management. With more awareness of database operators, a larger community is expected to form around developing better tooling around databases in K8s. This will lead to more performance benchmarking research and a gradual shift from traditional bare-metal database setups to automated Kubernetes setup when comparable or even better performance is consistently observed between the two set ups.

## References

[1] H. C. S. Perera et al., "Database Scaling on Kubernetes," *2021 3rd International Conference on Advancements in Computing (ICAC)*, 2021, pp. 258-263, doi: 10.1109/ICAC54203.2021.9671185.

[2] CNCF Operator White Paper
https://github.com/cncf/tag-app-delivery/tree/main/operator-whitepaper/v1

[3] Sharma, Bansal, P., Chugh, M., Chauhan, A., Anand, P., Hua, Q., & Jain, A. (2021). Benchmarking geospatial database on Kubernetes cluster. *EURASIP Journal on Advances in Signal Processing*, 2021(1), 1–29.

[4] To run or not to run a database on Kubernetes: What to consider - Benjamin Good
https://cloud.google.com/blog/products/databases/to-run-or-not-to-run-a-database-on-kubernetes-what-to-consider

[5] C. Zhu, B. Han and Y. Zhao, "A Comparative Study of Spark on the bare metal and Kubernetes," *2020 6th International Conference on Big Data and Information Analytics (BigDIA)*, 2020, pp. 117-124, doi: 10.1109/BigDIA51454.2020.00027.

[6]  L. Abdollahi Vayghan, M. A. Saied, M. Toeroe and F. Khendek, "Microservice Based Architecture: Towards High-Availability for Stateful Applications with Kubernetes," *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS)*, 2019, pp. 176-185, doi: 10.1109/QRS.2019.00034

[7] Cardas, Aldana-Martín, J. F., Burgueño-Romero, A. M., Nebro, A. J., Mateos, J. M., & Sánchez, J. J. (2022). On the performance of SQL scalable systems on Kubernetes: a comparative study. *Cluster Computing*. https://doi.org/10.1007/s10586-022-03718-9

[8] T. M. Gunawardena and K. P. N. Jayasena, "Real-Time Uber Data Analysis of Popular Uber Locations in Kubernetes Environment," *2020 5th International Conference on Information Technology Research (ICITR)*, 2020, pp. 1-6, doi: 10.1109/ICITR51448.2020.9310851.

[9] Running databases in a Kubernetes Cluster - An Evaluation - Olle Larsson - Master's Thesis.

[10] E. Truyen, M. Bruzek, D. Van Landuyt, B. Lagaisse and W. Joosen, "Evaluation of Container Orchestration Systems for Deploying and Managing NoSQL Database Clusters," *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, 2018, pp. 468-475, doi: 10.1109/CLOUD.2018.00066.

[11] W. Delnat, E. Truyen, A. Rafique, D. Van Landuyt and W. Joosen, "K8-Scalar: A Workbench to Compare Autoscalers for Container-Orchestrated Database Clusters," *2018 IEEE/ACM 13th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2018, pp. 33-39.

[12] Kubernetes official documentation https://kubernetes.io/docs/home/

[13] Zalando Operator for PostgreSQL documentation https://github.com/zalando/postgres-operator

[14] Percona Operator for MongoDB documentation https://docs.percona.com/percona-operator-for-mongodb/

[15] K8ssandra Operator documentation https://docs.k8ssandra.io/

[16] E. Tang and Y. Fan, "Performance Comparison between Five NoSQL Databases," *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, Macau, China, 2016, pp. 105-109, doi: 10.1109/CCBD.2016.030.

[17] S. N. Swaminathan and R. Elmasri, "Quantitative Analysis of Scalable NoSQL Databases," *2016 IEEE International Congress on Big Data (BigData Congress)*, San Francisco, CA, USA, 2016, pp. 323-326, doi: 10.1109/BigDataCongress.2016.49.

[18] Yahoo! Cloud Serving Benchmark (YCSB) https://github.com/brianfrankcooper/YCSB/

[19] CNCF Annual Survey 2021 https://www.cncf.io/reports/cncf-annual-survey-2022/

[20] J. M. A. Araujo, A. C. E. de Moura, S. L. B. da Silva, M. Holanda, E. d. O. Ribeiro and G. L. da Silva, "Comparative Performance Analysis of NoSQL Cassandra and MongoDB Databases," *2021 16th Iberian Conference on Information Systems and Technologies (CISTI)*, Chaves, Portugal, 2021, pp. 1-6, doi: 10.23919/CISTI52073.2021.9476319.

[21] Abramova, & Bernardino, J. (2013). NoSQL databases: MongoDB vs cassandra. *Proceedings of the International C Conference on Computer Science and Software Engineering*, 14–22. https://doi.org/10.1145/2494444.2494447