Vaasan yliopisto
UNIVERSITY OF VAASA

Heng Song

Customized openshift operator

An introduction to a tool for MongoDB automation and its application

School of Technology and Innovations

Major

Automation and information technology

Vaasa 2025

UNIVERSITY OF VAASA
School of Technology and Innovations

Author:	Heng Song
Title of the thesis:	Customized openshift operator: An introduction to a tool for MongoDB automation and its application
Degree:	Master of Science
Programme:	Automation and information technology
Supervisor:	
Year:	2025 **Pages:** 92

ABSTRACT:

MongoDB is a widely adopted NoSQL database in modern Platform-as-a-Service (PaaS) environments due to its flexibility, scalability, and ability to handle large volumes of transactional data. In large enterprise platforms processing thousands of transactions per minute, the need to create new databases dynamically is frequent and time critical. Manual provisioning in such cases is inefficient and prone to human error. Therefore, this research aims to design and implement an automated MongoDB database provisioning and deployment framework to enhance operational efficiency, consistency, and reliability across development, testing, and production environments.

The proposed solution leverages containerization and orchestration technologies, primarily Docker, Kubernetes (K8s), and OpenShift. To ensure continuous integration and delivery, Jenkins is incorporated as the CI/CD automation tool. This automation minimizes manual intervention and deployment risks while improving traceability and quality assurance. Building upon this foundation, the study explores OpenShift Operators, which extend Kubernetes functionality by embedding domain-specific operational intelligence into software controllers. A custom MongoDB Operator was developed using the Operator SDK and the Go (Golang) programming language. Go’s concurrency model and native support for Kubernetes APIs make it an ideal choice for implementing the Operator’s reconciliation loop, which continuously aligns the cluster’s actual state with the desired configuration defined in Custom Resources (CRs).

The thesis concludes with a prototype implementation demonstrating the end-to-end automation of MongoDB provisioning and deployment within an OpenShift environment. The results show that integrating Kubernetes, OpenShift Operators, and Jenkins CI/CD pipelines significantly improves the efficiency and reliability of database lifecycle management, effectively automate MongoDB management, reduce operational overhead, and serve as a scalable foundation for autonomous cloud-native database systems in large-scale enterprise platforms. The research also identifies key challenges—such as managing complex configurations and ensuring security compliance—and proposes directions for further enhancement toward building a robust, enterprise-ready automation framework.

KEYWORDS: Kubernetes, Operator, MongoDB, Golang, Jenkins

Contents

Figures

Tables

Codes

1 Introduction

1.1 Background and Motivation

In the world of cloud computing and digital transformation, data has become one of the most valuable assets of modern enterprises. Organizations operating large-scale online platforms must handle enormous amounts of data generated by millions of users and transactions every minute. Among the various database technologies available, MongoDB has emerged as one of the most popular NoSQL databases due to its flexibility, horizontal scalability, and schema-less design. These characteristics make MongoDB an ideal choice for microservices architectures and modern Platform-as-a-Service (PaaS) environments.

However, as platforms grow in complexity and scale, the demand for automated database provisioning has become increasingly critical. Traditional manual database creation and configuration processes are time-consuming and prone to human error. In large environments where new MongoDB instances must be created frequently—often as part of application onboarding, testing, or tenant provisioning—manual workflows can lead to delays, inconsistencies, and unnecessary operational costs. Automating these processes provides significant business benefits, including faster delivery times, reduced human intervention, and improved reliability and repeatability.

1.2 The Role of Containerization and Kubernetes

The evolution of container technologies has revolutionized how applications and databases are deployed and managed. Docker introduced lightweight, portable containers that encapsulate applications and their dependencies, enabling consistent behaviour across environments. Building on this foundation, Kubernetes (K8s) has become the standard for orchestrating and managing containers at scale.

Kubernetes offers powerful abstractions that simplify application lifecycle management. The use of Pods as the smallest deployable units enables flexible and isolated execution environments. Deployments manage the desired state of applications, ensuring fault tolerance, scalability, and zero-downtime updates. ConfigMaps and Secrets externalize configuration data and manage sensitive information securely. These core components together provide a robust and declarative framework that is particularly suitable for automating MongoDB deployments in cloud environments.

1.3 Continuous Integration and Deployment with Jenkins

Automation does not stop at the infrastructure level. The software development lifecycle also benefits significantly from Continuous Integration and Continuous Deployment (CI/CD) practices. In this thesis, Jenkins is utilized to orchestrate automated pipelines that manage the entire build-test-deploy process. Jenkins pipelines ensure that each modification to the MongoDB automation logic is continuously validated, tested, and deployed into Kubernetes or OpenShift environments in a consistent manner. This approach reduces human involvement in deployment processes, shortens delivery cycles, and increases the reliability of infrastructure changes.

1.4 OpenShift and the Operator Framework

While Kubernetes provides a powerful foundation, enterprise environments often require more sophisticated automation and governance features. Red Hat OpenShift extends Kubernetes with additional enterprise capabilities, including enhanced security, developer tools, and multi-tenant support. One of OpenShift's most powerful extensions is the Operator Framework, which allows developers to codify operational knowledge into custom controllers called Operators.

An OpenShift Operator can manage complex applications like MongoDB throughout their entire lifecycle—from provisioning and configuration to updates, scaling, and backups. By continuously observing the cluster state and reconciling it with the desired configuration defined in Custom Resources (CRs), Operators provide a self-healing, policy-driven automation model.

This thesis explores the design and implementation of a customized MongoDB Operator, developed using the Operator SDK and the Go (Golang) programming language. Go is chosen for its simplicity, performance, concurrency support, and native compatibility with Kubernetes APIs, making it ideal for building robust Operators that integrate seamlessly into cloud environments.

1.5 Objectives and Research Scope

The main objective of this thesis is to design and implement an automated solution for MongoDB database creation and lifecycle management across development, testing, and production environments. The specific goals include:

- Investigating the use of Kubernetes and OpenShift for deploying and managing MongoDB instances.
- Developing a proof-of-concept MongoDB Operator using Go and the Operator SDK.
- Implementing a CI/CD pipeline with Jenkins to automate deployment workflows.
- Evaluating the performance, reliability, and maintainability of the proposed solution.
- Identifying challenges and proposing improvements for future enterprise-grade implementations.

2 Introduction to MongoDB

Databases are the foundation of modern information systems, supporting everything from business applications to social media platforms and scientific research. Traditionally, relational databases such as Oracle, MySQL, and Microsoft SQL Server have dominated the field. However, in the past two decades, non-relational or “NoSQL” databases have emerged as a powerful alternative, particularly suited for handling massive amounts of unstructured or semi-structured data. MongoDB, one of the most widely adopted non-relational databases, has gained popularity for its flexibility, scalability, and ease of use.

MongoDB is an open-source, document-oriented NoSQL database that was first released in 2009 by MongoDB Inc. Unlike relational databases that store data in rows and tables, MongoDB uses a document model based on JSON-like structures called BSON (Binary JSON). This model allows developers to represent complex hierarchical data in a single record, eliminating the need for multiple joins or rigid schema definitions (what is MongoDB,n.d.).

Key characteristics of MongoDB include:

- **Schema flexibility:** Data structures can evolve without requiring costly schema migrations.
- **Horizontal scalability:** Sharding allows MongoDB to distribute data across multiple servers, making it suitable for high-volume workloads.
- **High availability:** Built-in replication provides fault tolerance and automatic failover.
- **Developer-friendly APIs:** MongoDB integrates seamlessly with modern programming languages and frameworks.

MongoDB is particularly attractive for projects requiring rapid development, frequent schema changes, and the ability to handle large-scale, real-time data. For example, network operator’s orchestration center, banking system, logistic system. etc.

2.1 Comparison between relational and non-relational databases

Before going deeply to MongoDB operations, it's beneficial to make a comparison between non-relational and relational databases to understand why non-relational databases become so popular today.

Data Model

Relational databases rely on structured schemas with predefined tables, rows, and columns. Data relationships are managed using primary and foreign keys, making relational type databases well-suited for applications where data consistency and integrity are paramount. In contrast, non-relational databases like MongoDB use more flexible models. Documents, key-value pairs, wide-column stores, or graph structures allow them to adapt to dynamic or complex data (What's the Difference Between Relational and Non-relational Databases, n.d.).

Scalability and Performance

Relational databases traditionally scale vertically, requiring more powerful hardware to handle larger workloads. While many relational systems now support some level of horizontal scaling, it is often complex to implement. Non-relational databases, by design, prioritize horizontal scaling, making them ideal for distributed systems, big data, and cloud-native applications (What's the Difference Between Relational and Non-relational Databases, n.d.).

Querying and Transactions

Relational systems excel in complex queries, joins, and ACID-compliant transactions. This makes them indispensable for financial applications, enterprise resource planning (ERP), and inventory systems. Non-relational databases, on the other hand, emphasize speed and flexibility. While MongoDB supports transactions, it is generally optimized for workloads where strict consistency is not always required (What's the Difference Between Relational and Non-relational Databases, n.d.).

Ecosystem and Use Cases

Relational databases remain dominant in industries where structured, consistent data is critical—such as banking, healthcare, and logistics. Non-relational databases thrive in modern web applications, content management, IoT, and analytics, where data is highly varied, rapidly changing, and requires scalable solutions (What's the Difference Between Relational and Non-relational Databases, n.d.).

The following table shows the comparison of key features between relational databases and non-relational databases:

Table 1. relational database vs non-relational database (Difference Between Relational and Non-Relational Database, 2022)

	Relational	Non-relational
Data Structure	Handles structured data	Handles unstructured data
Scaling	Vertically scalable	Horizontally scalable
Querying	Ideal for complex queries	Not ideal for complex queries
Performance	Can be slower	Very fast
Language	SQL	NoSQL

Typical Products:

Relational Databases

- **MySQL:** A widely used open-source database, often deployed in web applications such as WordPress and e-commerce platforms.
- **PostgreSQL:** Known for its standards compliance and advanced features, it is commonly used in scientific research, data warehousing, and government systems.
- **Oracle Database:** A commercial enterprise solution optimized for mission-critical applications, such as banking and large-scale enterprise management systems.

- **Microsoft SQL Server:** Popular in enterprise environments, particularly where integration with Microsoft technologies is required.

Non-Relational Databases

- **MongoDB (Document-oriented):** Used in content management systems, product catalogs, and mobile applications where flexible schemas are necessary.
- **Cassandra (Wide-column store):** Suited for large-scale logging, real-time analytics, and applications requiring high availability across multiple data centres.
- **Redis (Key-value store):** Commonly applied in caching, session management, and real-time leaderboard applications.
- **Neo4j (Graph database):** Specialized for social networks, recommendation engines, and fraud detection, where relationships between entities are the primary focus.

2.2 Managing MongoDB using Ops Manager

MongoDB provides Ops Manager, a management platform designed to simplify the operational aspects of running MongoDB at scale. Ops Manager allows enterprises to deploy, monitor, back up, and automate database clusters from a centralized interface. Its primary purpose is to streamline MongoDB administration by automating common tasks, enforcing best practices, and offering visibility into database health. Ops Manager transforms MongoDB management from a labour-intensive process into a streamlined and automated workflow (Ops Manager Overview, n.d.).

Followings will introduce key features of opsmanager.

2.2.1 Deployment Automation

One of the most powerful features of Ops Manager is its ability to automate the deployment and configuration of MongoDB clusters. Instead of manually installing software, editing configuration files, and managing replication or sharding, administrators can use Ops

Manager's UI or API to define cluster specifications. Ops Manager then provisions the servers, sets up replication, and ensures that all nodes are correctly configured (Install a Simple Test Ops Manager Installation, n.d.).

This reduces human error and accelerates the time required to spin up production-grade environments. Furthermore, Ops Manager supports version upgrades and rolling maintenance, minimizing downtime while keeping systems up to date.

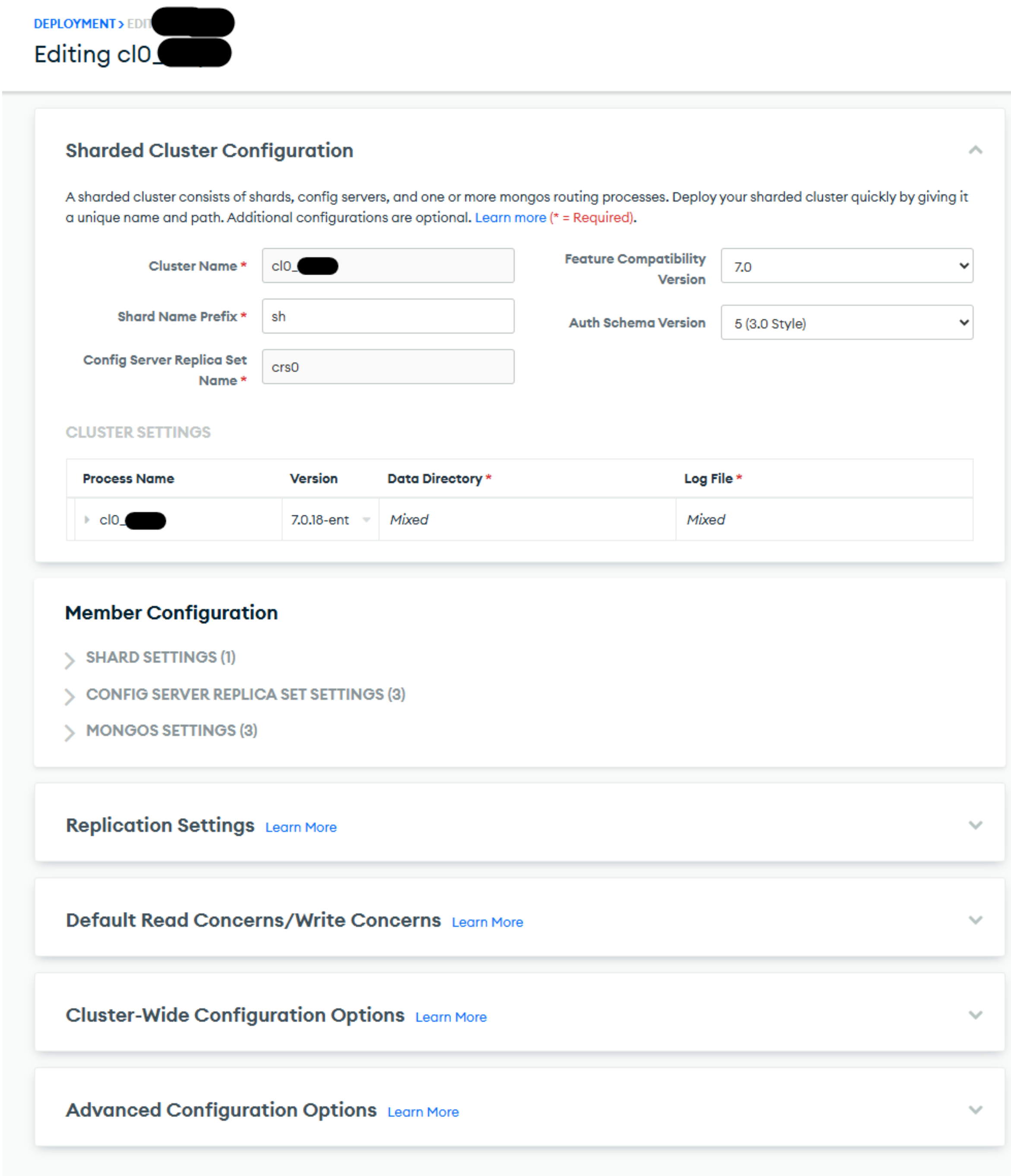

Figure 1. Deployment automation in Opsmanager

2.2.2 Monitoring and Metrics

Ops Manager provides real-time monitoring of MongoDB deployments, tracking over 100 system and database metrics. These include performance indicators such as query latency, replication lag, memory usage, and disk I/O. Metrics are visualized through interactive dashboards, allowing administrators to quickly identify bottlenecks or performance anomalies (Ops Manager Overview, n.d.).

Threshold-based alerting ensures that teams are notified via email, Slack, PagerDuty, or other integrations whenever issues arise. This proactive monitoring helps organizations maintain service-level agreements (SLAs) and avoid unexpected downtime.

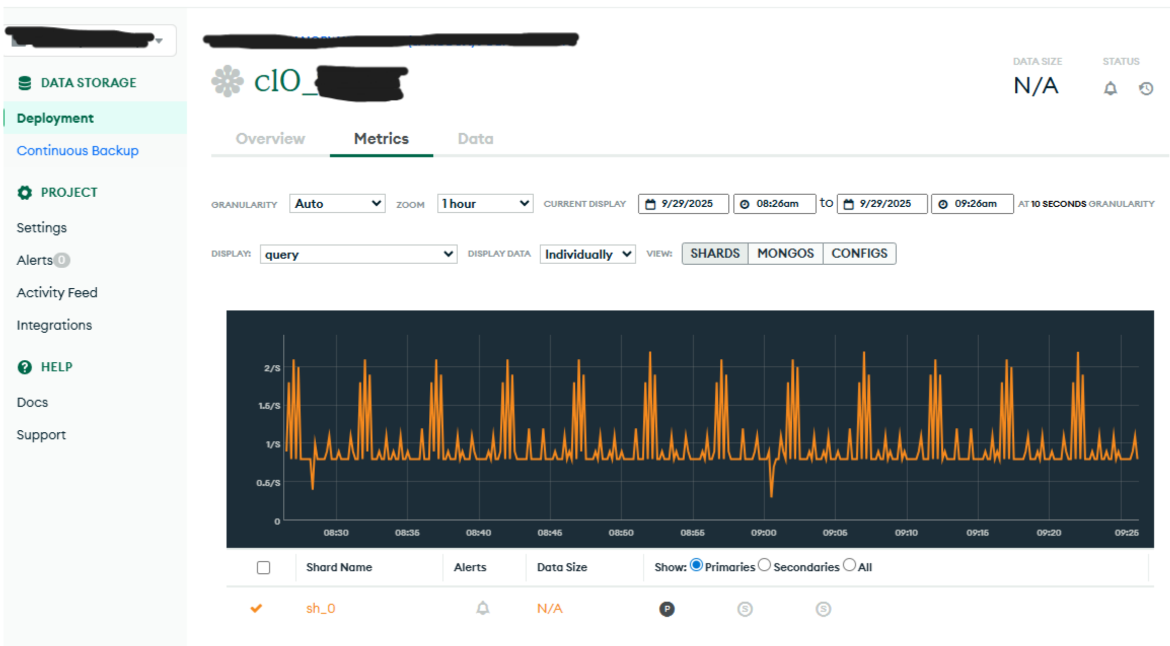

Figure 2. Cluster metrics in Opsmanager

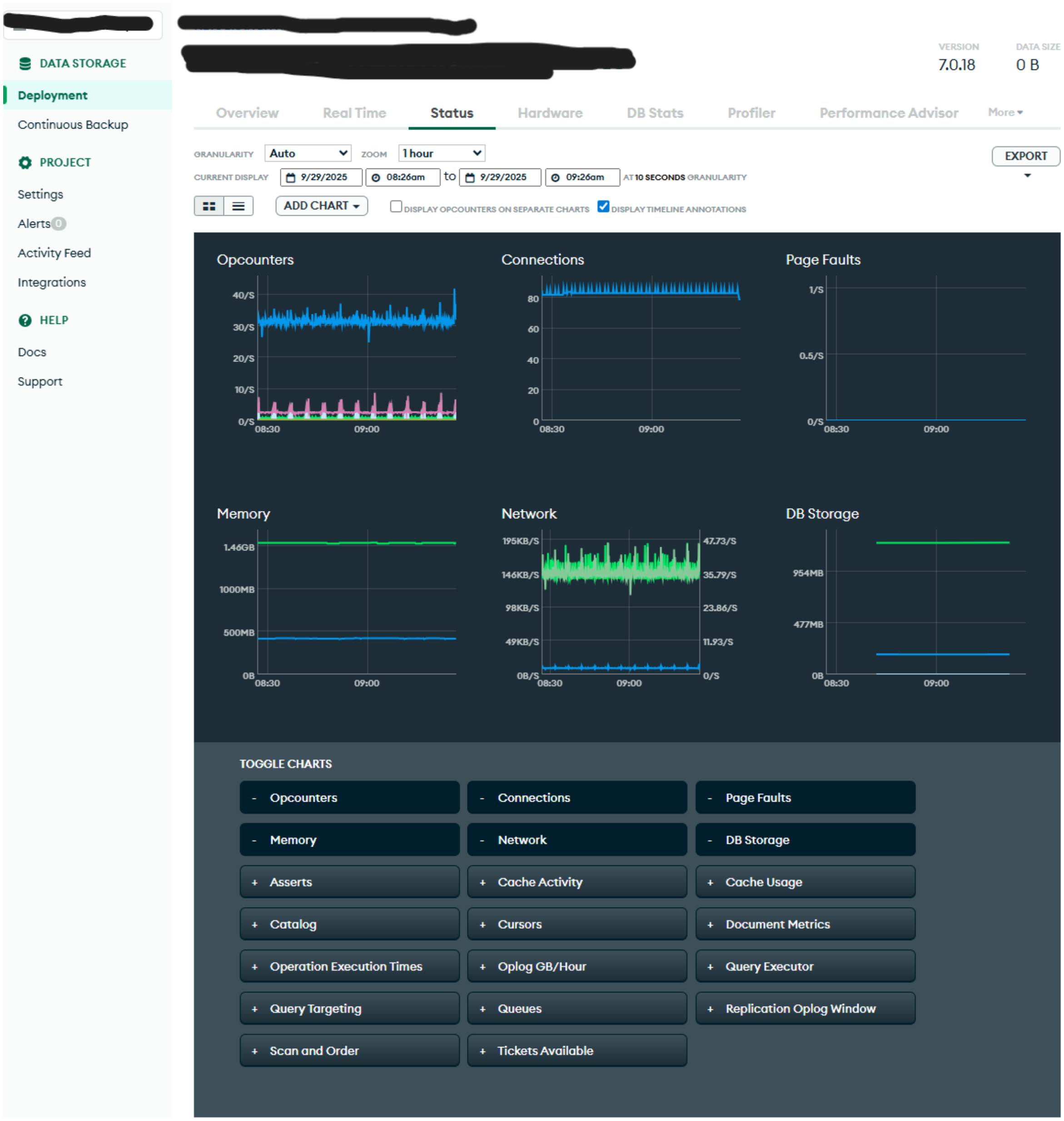

Figure 3. Single host's metrics in Opsmanger

2.2.3 Backup and Recovery

Data protection is essential for any production database. Ops Manager offers continuous, point-in-time backups of MongoDB clusters. Unlike periodic snapshots, which may leave gaps in data coverage, Ops Manager's continuous backup captures changes as they occur, ensuring minimal data loss in the event of a failure. Administrators can restore a cluster to a specific point in time, or clone backup data into a new environment for testing or analytics. These features make backup and recovery seamless, reliable, and compliant with enterprise disaster recovery policies (Ops Manager Overview, n.d.).

aption

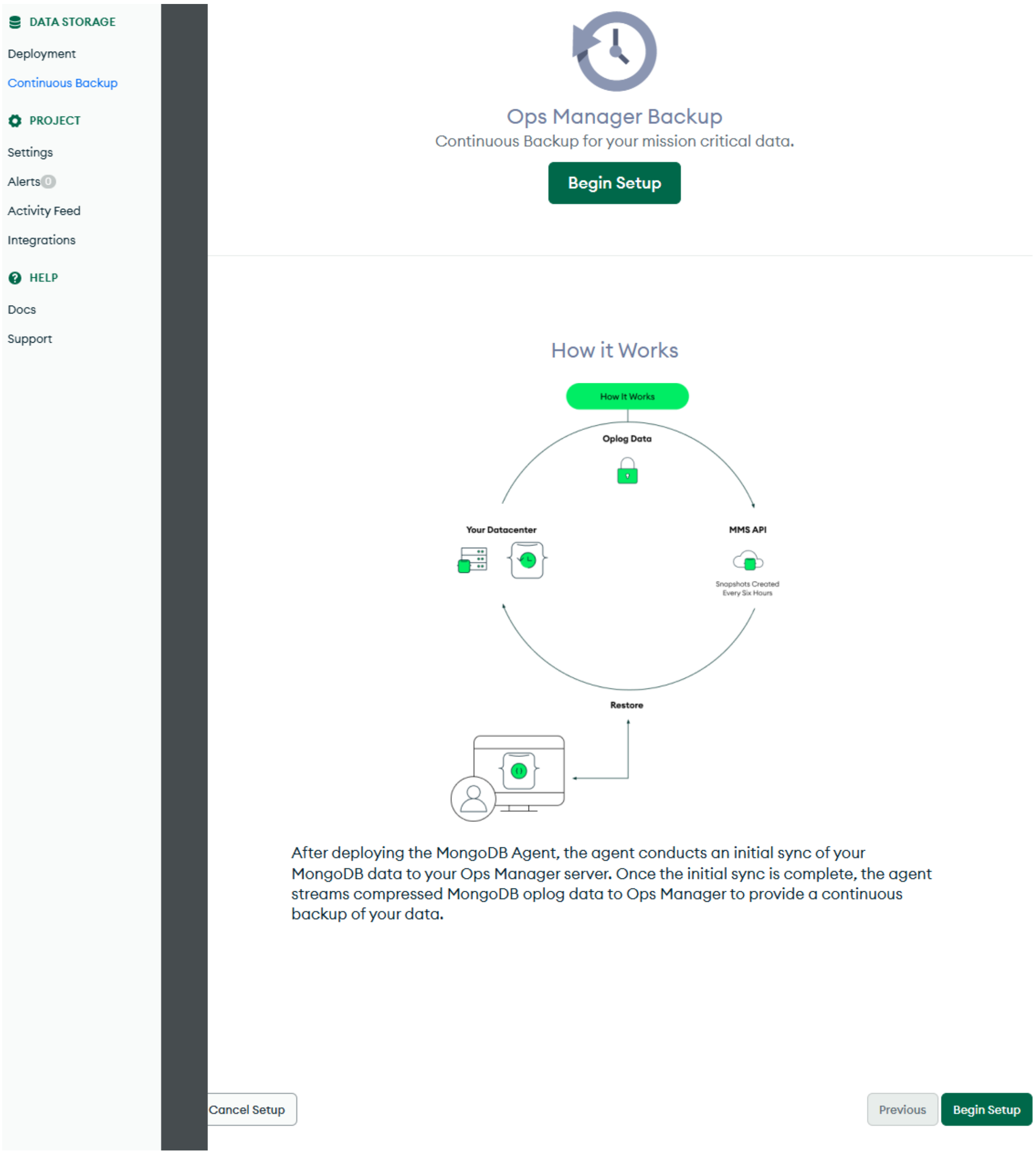

Figure 4. Backup and retore management in Opsmanger

2.2.4 Automation of Scaling and Maintenance

As application demands grow, MongoDB clusters often require additional resources. Ops Manager simplifies scaling by allowing administrators to add shards or replica set members with minimal effort. Similarly, storage and compute resources can be rebalanced across nodes to improve performance. Routine maintenance tasks, such as rolling restarts, configuration changes, and patch applications, are automated by Ops Manager, reducing manual intervention and ensuring consistent execution across large deployments (Ops Manager Overview, n.d.).

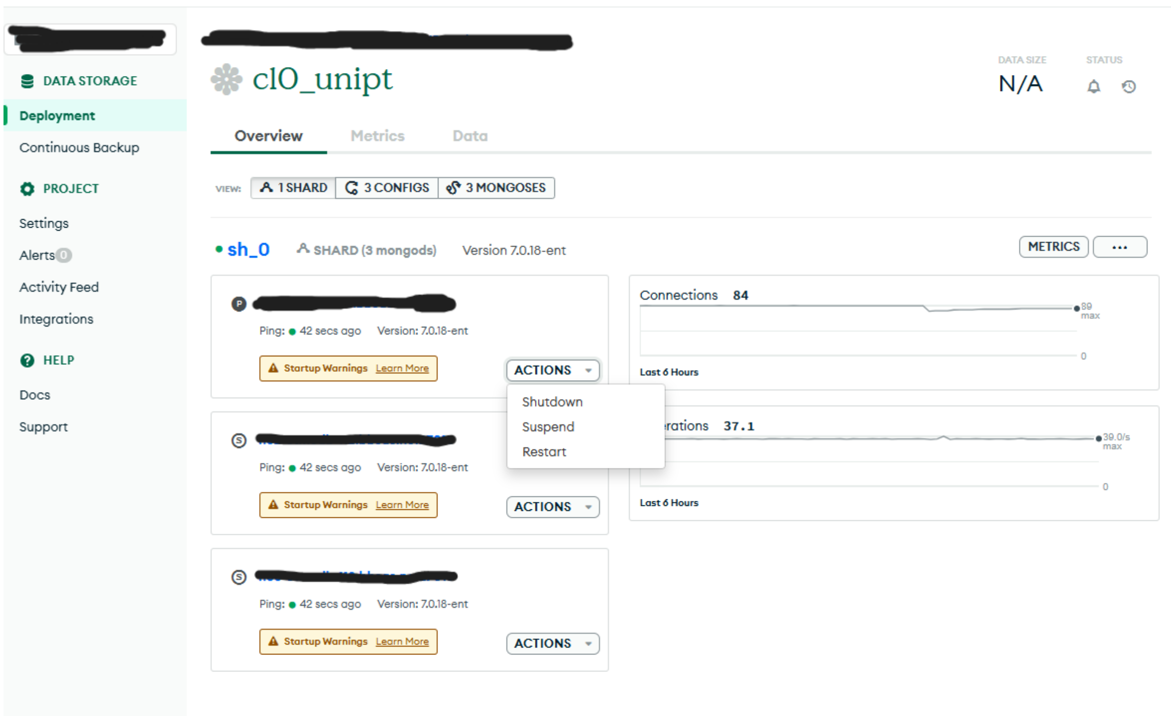

Figure 5. Scaling in Opsmanager

2.2.5 Security and Access Control

Security is another area where Ops Manager adds significant value. It centralizes the management of MongoDB authentication, role-based access control (RBAC), and encryption settings. Ops Manager integrates with corporate identity providers such as LDAP or Kerberos, making it easier for enterprises to enforce consistent access policies. Additionally, Ops Manager supports auditing capabilities that track user actions and configuration changes, which are essential for compliance with regulatory requirements in industries such as finance and healthcare (Ops Manager Overview, n.d.).

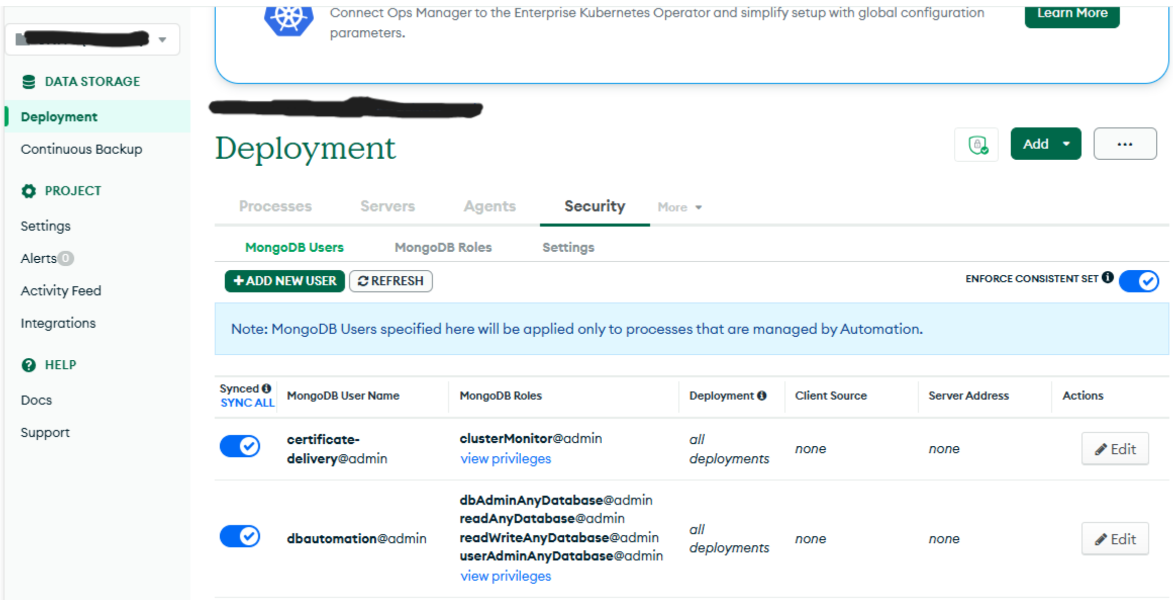

Figure 6. Access control in Opsmanager

3. Kubernetes

Kubernetes, commonly known as K8s, is an open-source platform for automating the deployment, scaling, and management of containerized applications. Originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF), Kubernetes has become the standard for container orchestration. Its core mission is to abstract infrastructure complexity and provide developers with a consistent, reliable environment for running workloads across clouds, data centers, and hybrid setups (Kubernetes overview, 2024).

At its foundation, Kubernetes provides primitives for describing application workloads, networking, storage, and security. Through its declarative model, developers and operators specify the desired state of a system, and Kubernetes ensures that the actual state matches this target. This architecture has reshaped how organizations build and operate modern applications, enabling resilience, scalability, and portability. In this chapter, we will compare Kubernetes based cloud with virtual machines (VMs)-based cloud, as well as introduce main concepts and components of a Kubernetes cluster.

3.1 VMs vs Kubernetes cloud

A cloud platform may consist of tens of micro services, and to ensure the stability and robust of each service, in VMs based cloud, the service will be deployed to at least 2 VMs, the amount of VMs can be even more than 10 for some core services. In such cases, load balancer and Virtual IP (VIP) needs to be manually configured for each service, additionally, tools which brings best configuration as a code practice like Ansible, Puppy are also needed for jobs like VM creation, components installation, service upgrade, certificates management, operating system maintenance. etc and lots of scripts/playbooks work are pending ahead for these jobs.

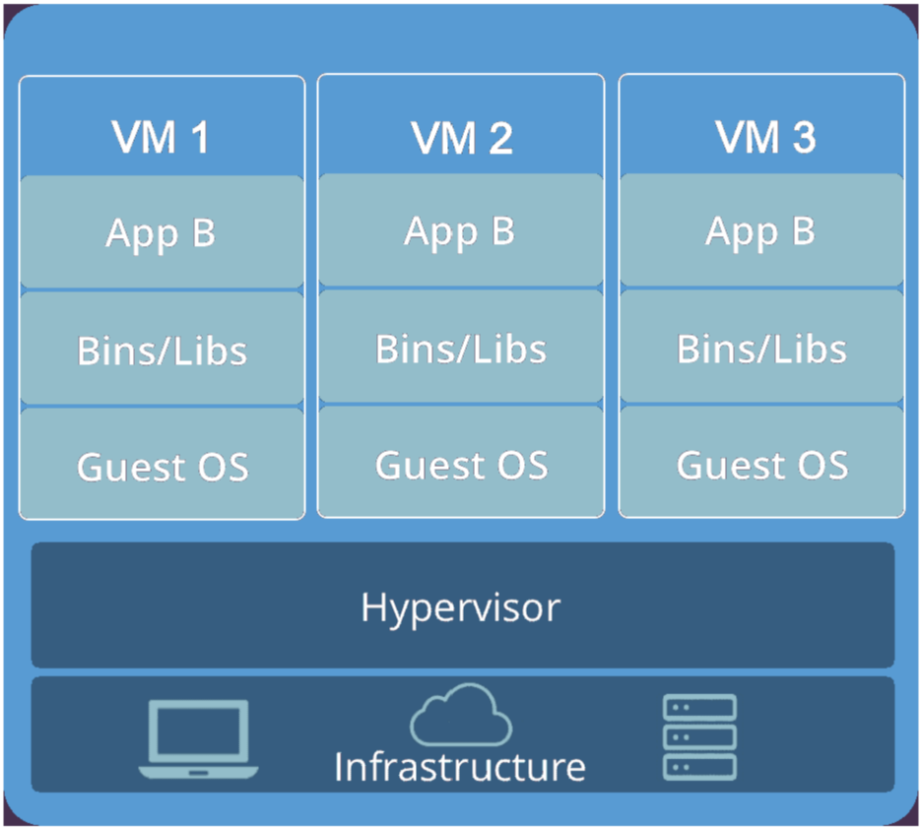

Figure 7. VM based cloud (Aniket Gawandar, 2024)

On the other hand, Kubernetes clouds focus on containerized workloads. Containers are lightweight, sharing the host OS kernel but isolating application processes. Kubernetes orchestrates deployment, scaling, and recovery of containers across clusters of machines. This model is optimized for microservices and cloud-native applications, enabling rapid scaling and efficient use of resources. Start-up times are measured in seconds, and horizontal scaling is automated. Kubernetes also abstracts infrastructure, making workloads portable across cloud providers or on-premises environments. While this flexibility is powerful, Kubernetes introduces operational complexity: teams must manage container images, networking, storage, and observability in a dynamic environment.

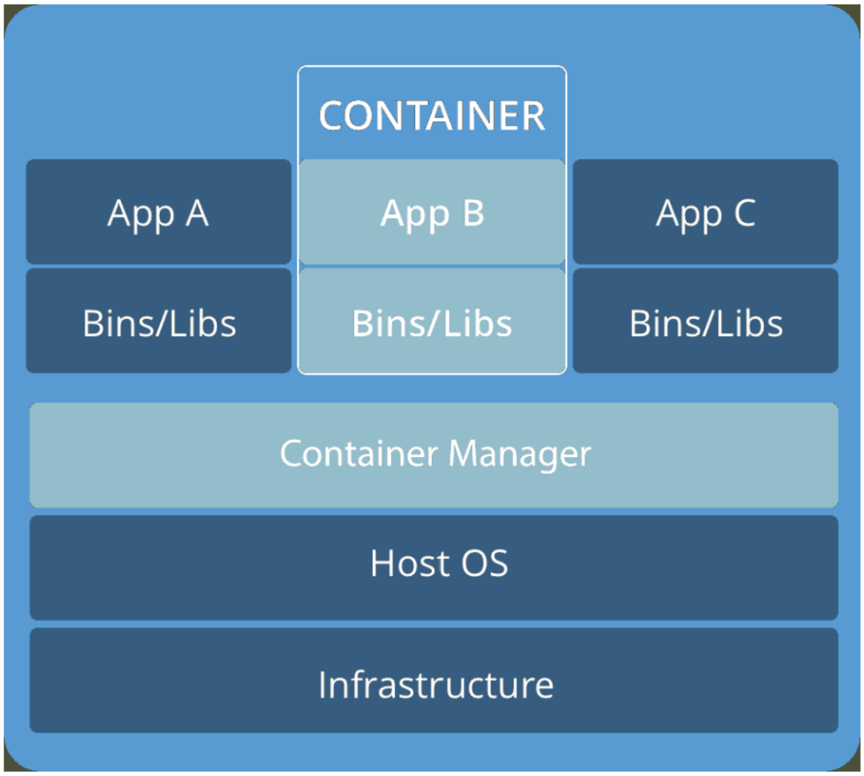

Figure 8. Kubernetes based cloud (Aniket Gawandar, 2024)

Comparison:

- **Efficiency**: VMs consume more resources; containers maximize density and performance.
- **Isolation**: VMs provide strong OS-level isolation; containers rely on process-level isolation.
- **Scalability**: VMs scale slower, while Kubernetes enables near real-time scaling.
- **Workload fit**: VMs suit legacy or monolithic apps; Kubernetes excels with microservices and modern CI/CD practices.

- **Management**: VM-based clouds are simpler for traditional IT teams; Kubernetes requires DevOps maturity.

In short, VM-based clouds are stable and familiar for running legacy workloads, while Kubernetes-based clouds offer agility and efficiency for modern, distributed applications. Organizations often use a hybrid model—VMs for traditional workloads and Kubernetes for cloud-native services (Aniket Gawandar, 2024) .

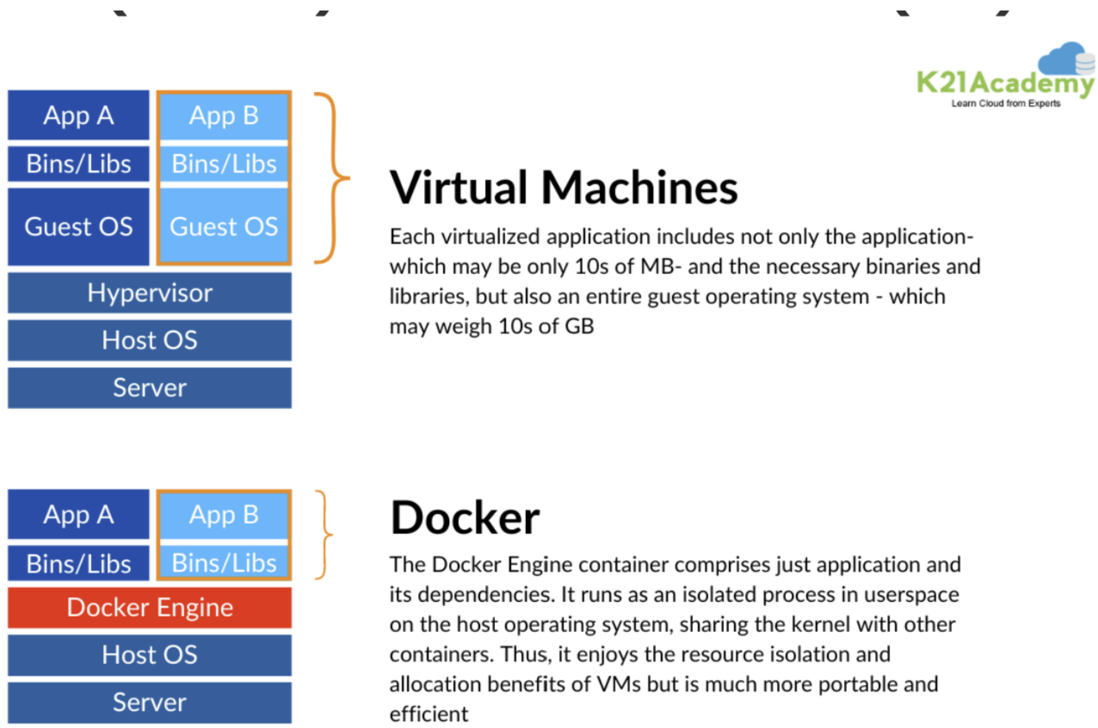

Figure 9. Virtual machines VS Kubernetes (Aniket Gawandar, 2024)

The concepts/definitions of Kubernetes' important components will be introduced in next several sections.

3.2 Pods

The Fundamental Execution Unit, a smallest deployable unit in Kubernetes is the Pod. A Pod encapsulates one or more containers that share storage, networking, and runtime context. This abstraction allows applications that need to work together—such as a main application container and a sidecar logging container—to run in the same environment (Pods, 2025).

Key characteristics of Pods:

Shared network namespace: All containers within a Pod can communicate via localhost.

Ephemeral lifecycle: Pods are not permanent; they are often replaced, scaled, or recreated by higher-level controllers.

Scalability: Instead of running a single Pod, Kubernetes encourages the use of controllers (like Deployments) to manage replicas and ensure availability.

Pods represent the atomic building block of applications in Kubernetes, but by themselves they are not sufficient for production-level workloads. That's why Kubernetes introduces higher-level abstractions such as Deployments.

3.3 Deployments

Manages Application Lifecycle. A Deployment is a controller that manages the lifecycle of Pods. It ensures that the desired number of replicas are running, provides rolling updates, and offers rollback functionality in case of failure (Deployments, 2025).

Advantages of Deployments include:

Declarative updates: Users specify the desired state in YAML, and Kubernetes reconciles the actual cluster state.

Scalability: Replica sets under a Deployment can be scaled up or down automatically.

Fault tolerance: If Pods fail, the Deployment controller replaces them.

For example, if an e-commerce application specifies 5 replicas in a Deployment manifest, Kubernetes ensures that 5 Pods are always running. If one node fails, new Pods are automatically scheduled on healthy nodes.

3.4 ConfigMaps

Manages Configuration Data. Modern applications often separate configuration from code. Kubernetes provides ConfigMaps to store non-sensitive configuration data in key-value pairs. ConfigMaps can be mounted into Pods as environment variables or files, allowing developers to modify configuration without rebuilding container images (Configmaps, 2024).

Common use cases:

Application configuration (URLs, file paths, log levels).

Feature toggles for enabling or disabling functionality.

Centralized storage of shared configuration for multiple services.

ConfigMaps promote portability and flexibility, enabling developers to adapt applications across environments (development, staging, production) without changing the application code.

3.5 Secrets

Securely Managing Sensitive Data. While ConfigMaps handle general configuration, Kubernetes Secrets store sensitive information such as passwords, API tokens, or TLS certificates. Secrets are base64-encoded by default and can be integrated with external security tools (e.g., HashiCorp Vault) for stronger encryption and lifecycle management (secrets, 2024).

Typical use cases:

Database connection strings.

Cloud service credentials.

Certificates and keys for secure communication.

By decoupling sensitive data from application code and container images, Secrets strengthen security and reduce the risk of accidental data leaks.

3.6 ServiceAccounts

Identity for Workloads. In Kubernetes, a ServiceAccount provides Pods with an identity to interact with the Kubernetes API or other system components. This is particularly important for workloads that need to access cluster resources programmatically (service accounts, 2024).

Features of ServiceAccounts:

Each Pod automatically receives a default ServiceAccount unless specified otherwise.

ServiceAccounts can be associated with specific Roles or ClusterRoles to control permissions.

Tokens linked to ServiceAccounts are mounted into Pods, enabling secure API access.

For example, a monitoring agent running in a Pod may use a ServiceAccount with read-only permissions to access cluster metrics. This follows the principle of least privilege.

3.7 Roles and RoleBindings

Fine-Grained Access Control Kubernetes implements Role-Based Access Control (RBAC) to manage permissions within a cluster. RBAC revolves around Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings (Using RBAC Authorization, 2025).

Role: Defines a set of permissions (e.g., “read Pods” or “create ConfigMaps”) within a specific namespace.

RoleBinding: Grants a Role’s permissions to a user, group, or ServiceAccount in that namespace.

ClusterRole: Like a Role, but cluster wide.

ClusterRoleBinding: Assigns cluster-wide permissions to subjects.

RBAC enables fine-grained control, ensuring that users and workloads only have access to the resources they need. This is essential for multi-tenant environments where different teams share the same cluster.

3.8 Namespace

A Namespace in Kubernetes is a logical partition within a cluster that allows teams or applications to isolate and organize resources. Instead of running everything in a single global scope, namespaces provide a way to separate environments, enforce resource quotas, and manage access control (Namespaces, 2024).

There are following key characteristics of namespaces:

Resource grouping: Objects like Pods, Deployments, ConfigMaps, and Services exist within a specific Namespace. This helps avoid name collisions since resources with the same name can exist in different Namespaces.

Multi-tenancy support: Different teams or projects can share a cluster without interfering with each other, as each team is allocated its own Namespace.

Access control: RBAC rules and Service Accounts can be scoped to specific Namespaces, ensuring fine-grained permissions.

Resource quotas and limits: Administrators can assign CPU, memory, and storage quotas per Namespace to prevent one workload from consuming all cluster resources.

Default Namespace: If no Namespace is specified, resources are created in the “default” Namespace. Kubernetes also provides system Namespaces such as kube-system (for internal components) and kube-public (for publicly accessible resources).

Example use cases:

Separating development, staging, and production environments within the same cluster.

Allocating distinct Namespaces for different microservices teams in a large organization.

Applying policies and quotas to ensure fair resource distribution.

Namespaces thus play a critical role in organizing workloads, enforcing governance, and supporting multi-tenant Kubernetes clusters.

4. Openshift and Customized Openshift Operator

OpenShift, developed by Red Hat, is an enterprise Kubernetes distribution that adds additional capabilities for security, developer productivity, and hybrid cloud management. While it builds on Kubernetes, OpenShift extends its functionality with features such as built-in CI/CD pipelines, enhanced security policies, and a web console for easier administration. OpenShift simplifies cluster setup, integrates enterprise authentication systems, and enforces stricter security defaults. This makes it particularly appealing to organizations that require Kubernetes but need enterprise support and compliance (Bob Reselman& Don Schenck, 2023).

According to red hat official document, one of OpenShift's most impactful innovations is the Operator Framework, which extends Kubernetes' capabilities for managing complex applications. Operators are Kubernetes-native applications that codify human operational knowledge into automated controllers (Understanding Operators, n.d.).

An Operator uses Custom Resource Definitions (CRDs) to define new types of Kubernetes resources beyond the built-in ones. The Operator then acts as a controller that watches these resources and performs tasks to keep the system in its desired state.

What makes the operators stands out is a fact that without operators, managing complex applications often requires manual steps—backups, schema migrations, scaling adjustments. Operators bring these operational tasks into Kubernetes' declarative model, enabling "Day 2 operations" (maintenance, upgrades, monitoring) to be automated just like initial deployments.

OpenShift and Operator Use Cases

- Database Operators: PostgreSQL, MongoDB, or MySQL Operators manage stateful databases inside Kubernetes environments.
- Middleware Operators**:** Automating Kafka or Redis deployments.

- Enterprise Applications**:** Automating SAP, ElasticSearch, or custom in-house workloads.

In enterprise contexts, Operators reduce human error, improve consistency, and provide DevOps teams with a repeatable, automated framework for application management. OpenShift makes Operators first-class citizens, providing tooling, certification programs, and a marketplace of prebuilt Operators.

5. MongoDB automation implementation

After we have researched what is MongoDB and its management tool Opsmanager, as well as Kubernetes and openshift operator, we may start to build an application utilizing these technologies.

5.1 Application Design

The demand of the application is that database users can create database requests as Kubernetes customize resource from their own managed namespaces, and the operator will be deployed to a certain namespace named e.g. platform-ci, and will be capable to watch all the database requests from its own namespace or other namespaces in the same openshift cluster, and then push new databases requested from customize resource to MongoDB. With the peer review of containerizing technologies (Carmen Carrion,2022), the big picture of the application design is as following:

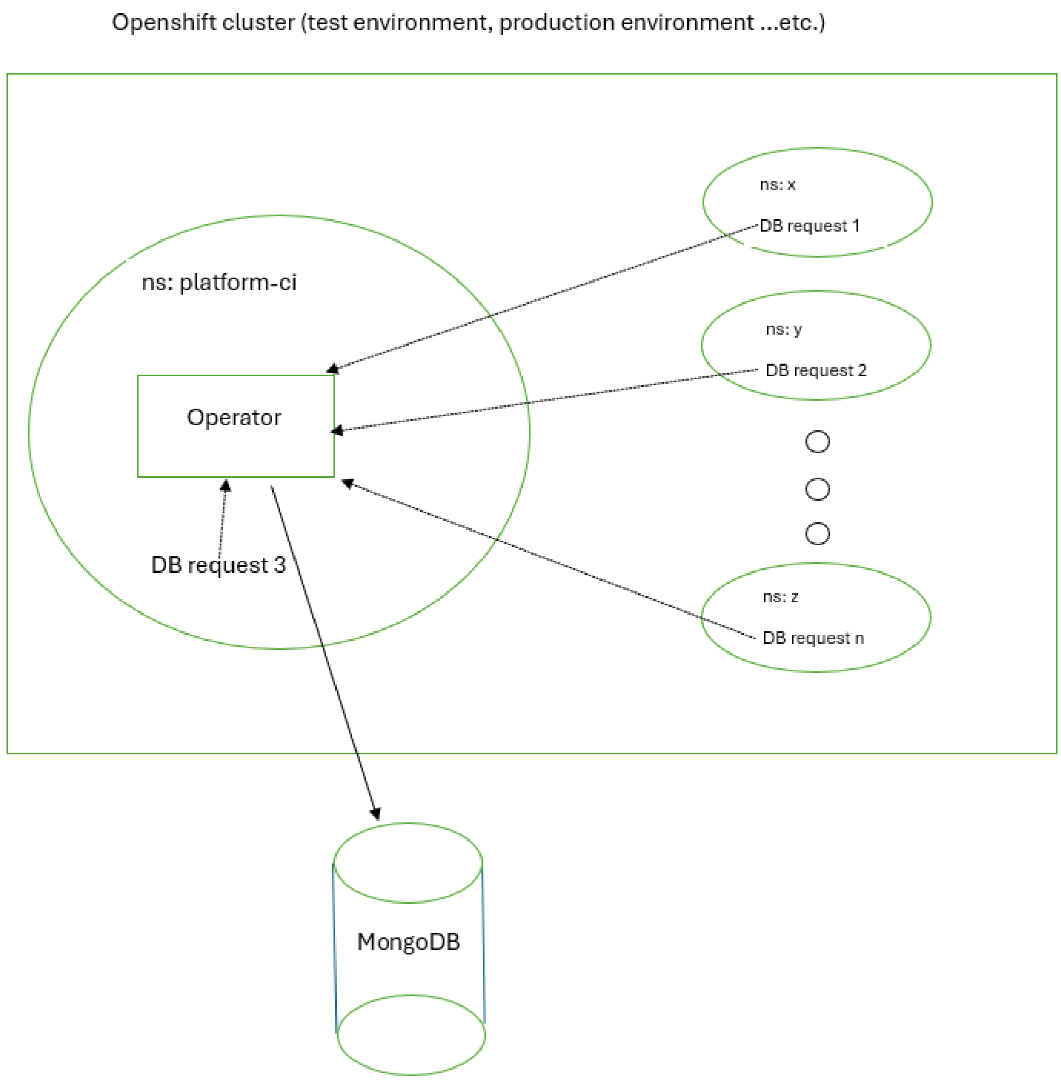

Figure 10. Operator use cases

How the operator will work in details:

There will be a config-map and secret deployed along with the operator, which contains the opsmanager connection information (opsmanager url, possible different cluster names for different usages) and the api keypair values for opsmanager operations. These values will then be passed to EnvLoader (environment loader) object, together with what user has defined in database request, will be passed into the controller object, create an opsmanager object, eventually get operated in MongoDB via opsmanager. The controller will also create new admin users for the database during database creation and save the user’s credentials into a secret to the namespace where database request is raised from.

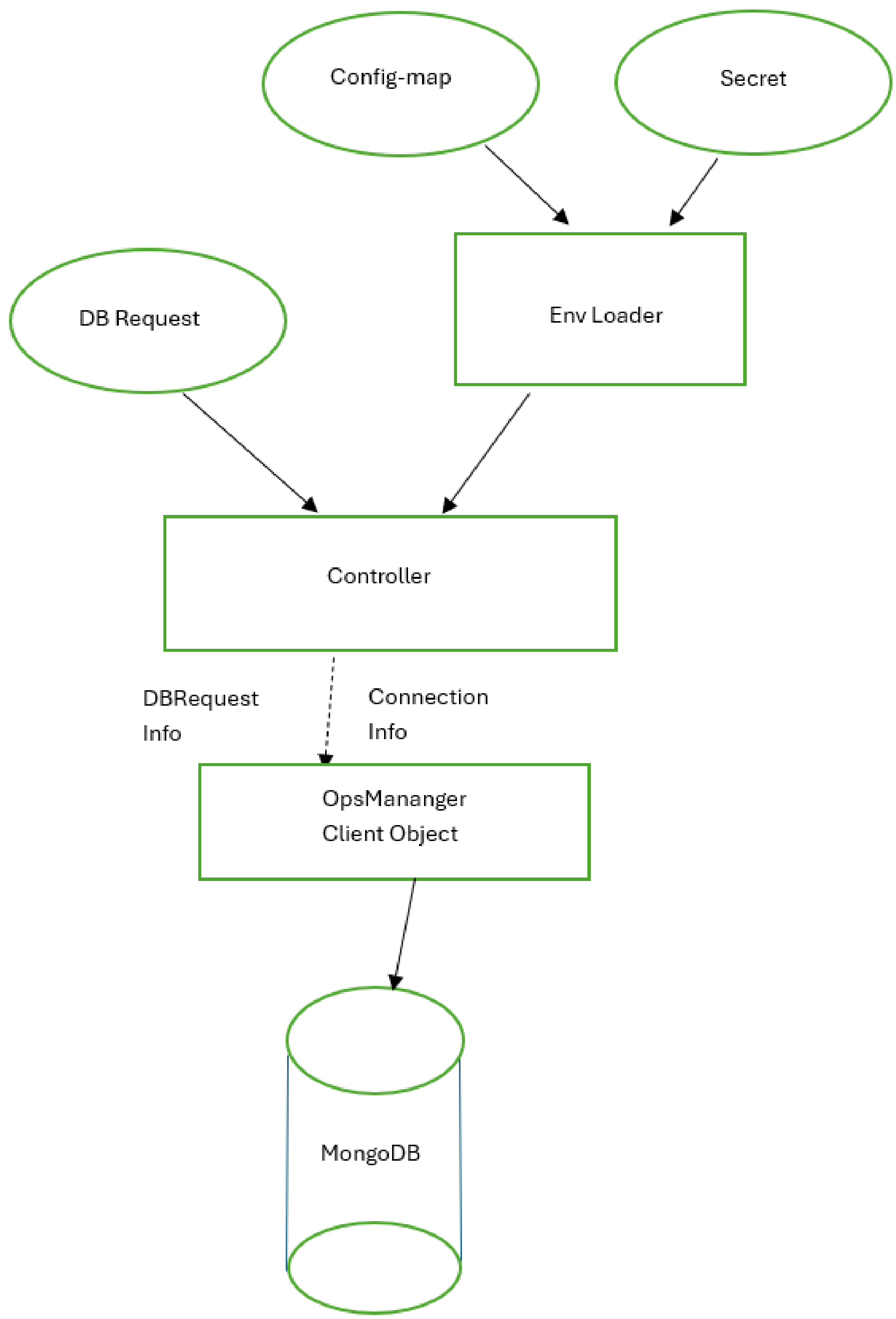

Figure 11. Workflow inside the operator

5.2 Code development

Before coding, it needs to be determined which programming language is preferred for the application and go lang is selected. The most compelling reason for using Go in Kubernetes-related development is that Kubernetes itself is written in Go. This means that all Kubernetes components, APIs, and client libraries are designed with Go in mind. Writing applications in the same language simplifies integration, leverages existing libraries, and ensures compatibility with the Kubernetes ecosystem. Next let us look of the application's core code.

5.2.1 Main function

Main function is an entry of the application, it instantiates a controller to start watching resources pool and operating accordingly, for the code, please refer to Appendix 1.

5.2.2 Controller

The controller consists of core functions of the operator, reconciling defined namespaces for new database requests, handling database operations, creating new secrets for new databases, for the code, please refer to Appendix 2.

5.2.3 OpsManager client

This section shows how database operation is implemented, it gets automation config from opsmanager, append new database info from requests to the automation config, then push the updated automation config to opsmanager, and opsmanager will handle the database creation. For the code, please refer to Appendix 3.

5.2.4 Environment loader

Environment loader will load environment specific configuration and credentials for ops manager accordingly, e.g. test environment -> test opsmanager instance; pre-production environment -> pre-production opsmanager instance. For the code, please refer to Appendix 4.

5.3 Deploy and test

After the code is ready, it must be tested within the OpenShift environment. To deploy a customized Operator on OpenShift, a common approach is to package the compiled Operator project into a container image and reference this image in a deployment manifest. Among the available containerization tools, Docker is one of the most widely adopted in the cloud computing ecosystem.

5.3.1 Introduction to Docker and Code Segment

Docker is an open-source platform that fundamentally transformed the development, packaging, and deployment of software applications. Introduced in 2013, Docker popularized the concept of lightweight containers, which encapsulate an application together with its dependencies into a portable, self-contained unit. Unlike traditional virtual machines that emulate entire operating systems, Docker containers share the host operating system's kernel while isolating processes at the application level. This design results in significantly higher efficiency, faster startup times, and simplified management.

At its core, Docker enables developers to build container images—immutable blueprints containing application code, runtime libraries, and environment configurations. These images can be stored in container registries such as Docker Hub and deployed consistently across diverse environments, including development workstations, testing clusters, and production Kubernetes platforms. This ensures environment consistency and effectively eliminates the long-standing issue of *"it works on my machine"* (What is Docker, n.d.).

The key advantages of Docker include:

- **Portability:** Containers behave consistently across different operating systems and cloud infrastructures.
- **Resource efficiency:** Containers require fewer system resources than virtual machines, allowing higher application density on the same hardware.

- **Scalability:** Applications can be easily scaled by running multiple container instances as needed.
- **Developer productivity:** By encapsulating complex runtime environments, Docker simplifies setup and reduces onboarding time for developers.

The following section presents the Docker file used to build the MongoDB Operator image.

```
FROM registry.access.redhat.com/ubi8/go-toolset:1.21 as builder

# Switch to root user to have write permissions
USER root

WORKDIR /workspace

# Configure Go environment for corporate networks with certificate
bypass
ENV GOPROXY=https://nexus.xxxx.no:9006/repository/go-proxy/,direct
ENV GOSUMDB=off
ENV GO111MODULE=on
ENV GOTOOLCHAIN=local

# Copy the Go Modules manifests and vendor directory
COPY go.mod go.mod
COPY go.sum go.sum
COPY vendor/ vendor/

# Copy the go source
COPY main.go main.go
COPY api/ api/
COPY controllers/ controllers/
```

```
COPY pkg/ pkg/

# Build using vendored dependencies (no network access needed)
# Ensure the build directory is writable
RUN mkdir -p /workspace && chmod 755 /workspace && \
    CGO_ENABLED=0 GOOS=linux GOARCH=amd64 GO111MODULE=on go build
-a -mod=vendor -o manager main.go

# Use Red Hat UBI minimal as base image (more accessible in
corporate environments)
FROM registry.access.redhat.com/ubi8/ubi-minimal:latest
WORKDIR /
COPY --from=builder /workspace/manager .

# Create non-root user for security
RUN microdnf install -y shadow-utils && \
    groupadd -r -g 65532 nonroot && \
    useradd -r -u 65532 -g nonroot nonroot && \
    microdnf clean all

USER 65532:65532

ENTRYPOINT ["/manager"]
```

Code 1. Code of docker file

5.3.2 Openshift manifest files

In this section, the manifest files deployed to Openshift prior to running the operator will be introduced.

For the operator to recognize the customized resource, a customized resource definition (CRD) needs to be installed into openshift first, following is the CRD for the operator, it defines a customized resource named “MongoDB”:

```yaml
kind: CustomResourceDefinition
metadata:
  annotations:
    controller-gen.kubebuilder.io/version: v0.9.2
  creationTimestamp: null
  name: mongodbs.test.mongo
spec:
  group: [redacted]
  names:
    kind: MongoDB
    listKind: MongoDBList
    plural: mongodbs
    singular: mongodb
  scope: Namespaced
  versions:
  - additionalPrinterColumns:
    - jsonPath: .status.phase
      name: Status
      type: string
    - jsonPath: .metadata.creationTimestamp
      name: Age
      type: date
    name: v1
    schema:
      openAPIV3Schema:
        description: MongoDB is the Schema for the uniptmon
        properties:
          apiVersion:
            description: 'APIVersion defines the versioned
            type: string
          kind:
            description: 'Kind is a string value representi
            type: string
          metadata:
            type: object
          spec:
            description: MongoDBSpec defines the desired st
            properties:
              clusterType:
                description: ClusterType is the type of Mon
                type: string
              envSymbol:
                description: EnvSymbol is the environment s
                type: string
              mongodb:
                description: MongoDB contains the MongoDB d
                properties:
                  name:
                    description: Database name
```

Figure 12. Customized resource definition (CRD) for the operator

ServiceAccount, ClusterRole, ClusterRoleBinding enables the operator to own an account and with proper user rights assigned to perform operations:

ServiceAccount:

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mongodb-operator
```

Figure 13. ServiceAccount for the operator

ClusterRole

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: mongodb-operator-manager-role
rules:
- apiGroups:
  - test.mongo
  resources:
  - databaserequests
  verbs:
  - create
  - delete
  - get
  - list
  - patch
  - update
  - watch
- apiGroups:
  - ""
  resources:
  - configmaps
  - secrets
  verbs:
  - create
  - delete
  - get
  - list
  - patch
  - update
  - watch
```

Figure 14. Cluster role for the operator

ClusterRoleBinding:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: mongodb-operator-manager-rolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: mongodb-operator-manager-role
subjects:
- kind: ServiceAccount
  name: mongodb-operator
  namespace: testpt-ci
```

Figure 15. Cluster rolebinding for the operator

And deployment manifest is a core part to state application’s compute resources, image in use, specifications, parameters and all other important information:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongodb-operator
  labels:
    app: mongodb-operator
    group: test.devops
spec:
  replicas: 1
  revisionHistoryLimit: 2
  selector:
    matchLabels:
      app: mongodb-operator
      group: test.devops
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: mongodb-operator
        group: test.devops
    spec:
      serviceAccountName: mongodb-operator
      containers:
        - name: mongodb-operator
          image: mongodb-operator:@@VERSION@@
          imagePullPolicy: IfNotPresent
          command: ["/manager"]
          args:
            - "--leader-elect"
          env:
            - name: WATCH_NAMESPACE
              value: ""
            - name: OPERATOR_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
          # Note: Environment configuration is loaded dynamically from DatabaseRequest.spec.envSym
          ports:
            - containerPort: 8080
              name: metrics
              protocol: TCP
          resources:
            limits:
              cpu: "100m"
              memory: "1Gi"
            requests:
              cpu: "10m"
              memory: "512Mi"
          livenessProbe:
            httpGet:
              path: /healthz
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 15
            periodSeconds: 20
          readinessProbe:
            httpGet:
              path: /readyz
              port: 8080
              scheme: HTTP
            initialDelaySeconds: 5
            periodSeconds: 10
```

Figure 16. Deployment manifest file of the operator

Operator secret is not showing here, but it’s essential as it saves opsmanager connection info and credentials used for MongoDB authorization.

5.3.3 Introduction to Jenkins and practice deployment automation

There might be typo, bugs, misconfiguration, all kinds of errors found during the test, hence the application code will be updated and built into image then deployed to the openshift for multiple times. Summarized from above steps, doing these manually is terribly time consuming. Jenkins can help here to automate the build and deployment with one click after code changes.

Jenkins is an open-source automation server that enables Continuous Integration (CI) and Continuous Delivery (CD). While Jenkins can be used through a graphical interface, its real power comes from automation pipelines. Pipelines define the stages of software delivery—such as building, testing, and deploying—in a structured way. This is where Groovy plays a central role. Jenkins Pipelines are written using a Groovy-based Pipeline DSL. Developers write Jenkinsfile scripts, typically stored in version control, to describe how code should be built and deployed. For example, a Jenkinsfile might define stages like “Compile,” “Run Tests,” and “Deploy to Kubernetes.” (Creating your first Pipeline, n.d.)

Advantages of using Groovy in Jenkins include:

Declarative and Scripted Pipelines: Jenkins supports both declarative style (simpler syntax for common cases) and scripted style (full Groovy flexibility).

Reusability: Groovy functions and shared libraries allow teams to reuse pipeline logic across multiple projects.

Expressiveness: Groovy makes it easy to model complex workflows, conditionals, and error handling.

Integration: Since Groovy runs on the JVM, it integrates naturally with the Java-based Jenkins ecosystem.

With a peer review of CI/CD pipelines for cloud applications (Rafal Leszko,2022), Appendix 5. is the Jenkins file automating operator image building and the requested files deployment, it has two switches design that one controls deployment namespaces for different environments (e.g. test, pre-production, production etc.), while the other controls which openshift instance will be targeted when running the job from different Jenkins instances. And following is how the pipeline looks like based on the jenkins file:

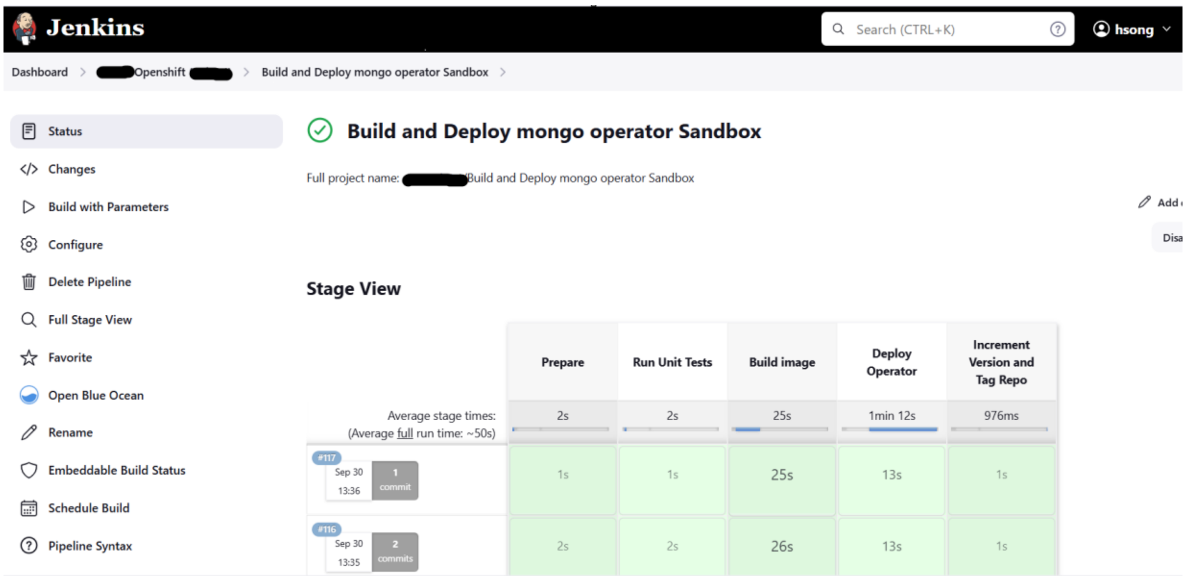

Figure 17. A jenkins job to compile and deploy the operator

5.3.4 Application test results

The expected outcome of the application is that there will be:

- a MongoDB operator pod running in the OpenShift
- once a custom resource feed to the operator, it will create database in MongoDB
- A secret created contains database admin user credentials info

After we deploy all the manifest files and the application image, we see an operator pod running in the OpenShift:

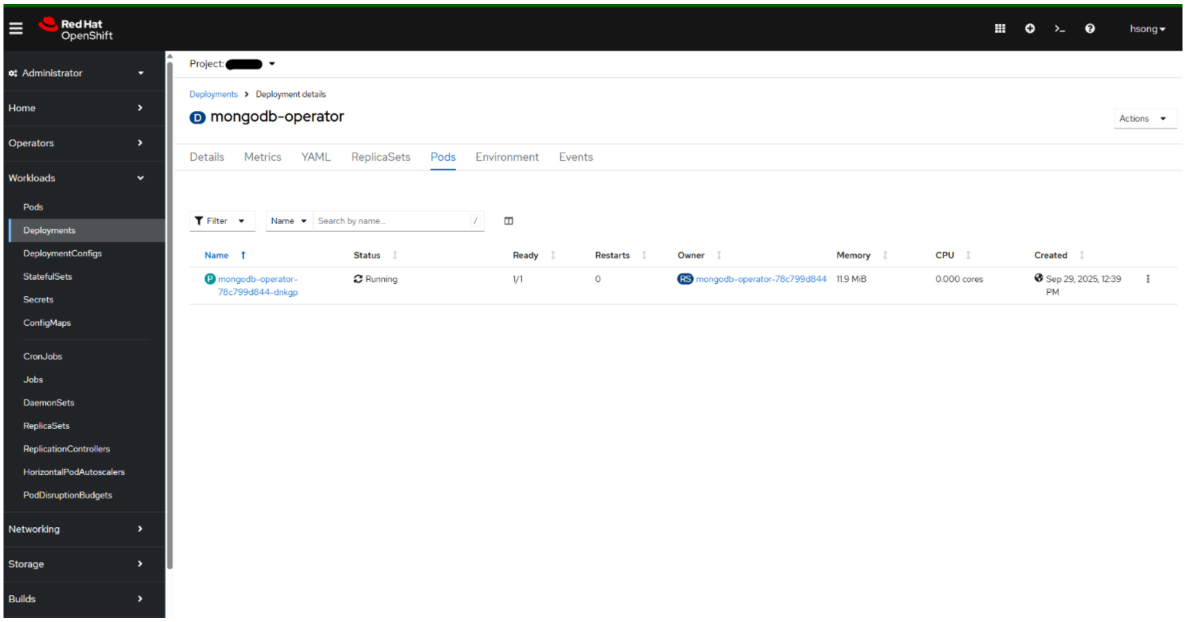

Figure 18. An operator pod running in the OpenShift

Then feed a database request to the operator, sample of a customized resource:

```yaml
apiVersion: mongo.test/v1
kind: MongoDB
metadata:
  name: heng-app-database
  namespace: testpt-t
spec:
  dbname: heng-application-db
  environment: test
  secretName: heng-app-db-credentials
  secretNamespace: testpt-t
```

Figure 19. Sample of a customized resource

In this sample, we have asked the operator to create a mongo database named “heng-application-db”, and from operator pod logs, when taking a look at red line marked information, we can see the database request has been handled and the database whose name specified as “heng-application-db” created successfully, additionally, a secret contains administrator credentials named “heng-app-db-credentials” is also created to wanted namespace “testpt-t”successfully:

```
2025-09-24T13:22:40Z INFO setup starting manager
2025-09-24T13:22:41Z INFO starting server {"kind": "health probe", "addr": "[::]:8081"}
2025-09-24T13:22:41Z INFO Starting EventSource {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "source": "kind source: *v1.mongoDB"}
2025-09-24T13:22:41Z INFO Starting Controller {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB"}
2025-09-24T13:22:41Z INFO Starting workers {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "worker count": 1}
2025-09-24T13:22:41Z INFO Reconciling mongoDB {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "4b0aaaa0-faff-4d93-9493-62594516cd8e", "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "1758720161311903076"}
2025-09-24T13:22:41Z INFO Resource details {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "4b0aaaa0-faff-4d93-9493-62594516cd8e", "generation": 1,
"resourceVersion": "2633666417", "databaseReady": false, "deletionTimestamp": "<nil>", "finalizers": []}
2025-09-24T13:22:41Z INFO Found OpsManager project {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "4b0aaaa0-faff-4d93-9493-62594516cd8e", "name": "test PT
(Sandbox)", "id": "66601ba0c3ecc34c5b90050b"}
2025-09-24T13:22:41Z INFO Adding finalizer to resource {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "4b0aaaa0-faff-4d93-9493-62594516cd8e"}
2025-09-24T13:22:41Z INFO Finalizer added, requeuing for next reconciliation {"controller": "mongoDB",
"controllerGroup": "mongo.test", "controllerKind": "mongoDB", "mongoDB":
{"name":"heng-app-database","namespace":"testpt-t"}, "namespace": "testpt-t", "name": "heng-app-database",
"reconcileID": "4b0aaaa0-faff-4d93-9493-62594516cd8e"}
2025-09-24T13:22:41Z INFO Reconciling mongoDB {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "1758720161425487442"}
2025-09-24T13:22:41Z INFO Resource details {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "generation": 1,
"resourceVersion": "2633670004", "databaseReady": false, "deletionTimestamp": "<nil>", "finalizers":
["mongo.test/finalizer"]}
2025-09-24T13:22:41Z INFO Found OpsManager project {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "name": "test PT
(Sandbox)", "id": "66601ba0c3ecc34c5b90050b"}
2025-09-24T13:22:41Z INFO Proceeding with database management {"controller": "mongoDB", "controllerGroup":
"mongo.test", "controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"},
"namespace": "testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b"}
2025-09-24T13:22:41Z INFO Creating database {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "name":
"heng-application-db"}
2025-09-24T13:22:41Z INFO Created database user in automation config {"controller": "mongoDB", "controllerGroup":
"mongo.test", "controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"},
"namespace": "testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b",
"username": "heng-application-db-user", "database": "heng-application-db"}
2025-09-24T13:22:41Z INFO Multiple MongoDB hosts found, using first one {"controller": "mongoDB",
"controllerGroup": "mongo.test", "controllerKind": "mongoDB", "mongoDB":
{"name":"heng-app-database","namespace":"testpt-t"}, "namespace": "testpt-t", "name": "heng-app-database",
"reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "hosts": ["no1-stmodb-112.bbsas.no",
"no2-stmodb-112.bbsas.no", "no3-stmodb-112.bbsas.no"], "selected": "no1-stmodb-112.bbsas.no"}
2025-09-24T13:22:41Z INFO Database setup completed {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "name":
"heng-application-db"}
2025-09-24T13:22:41Z INFO Creating new secret {"controller": "mongoDB", "controllerGroup": "mongo.test",
"controllerKind": "mongoDB", "mongoDB": {"name":"heng-app-database","namespace":"testpt-t"}, "namespace":
"testpt-t", "name": "heng-app-database", "reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "secret":
"heng-app-db-credentials", "namespace": "testpt-t"}
2025-09-24T13:22:41Z INFO Successfully created database and updated status {"controller": "mongoDB",
"controllerGroup": "mongo.test", "controllerKind": "mongoDB", "mongoDB":
{"name":"heng-app-database","namespace":"testpt-t"}, "namespace": "testpt-t", "name": "heng-app-database",
"reconcileID": "0f3bbed1-b5d6-4127-bc18-0befbe05a38b", "database": "heng-application-db"}
```

Figure 20. Logs from the operator pod

We can also log into opsmanager to verify if database is created:

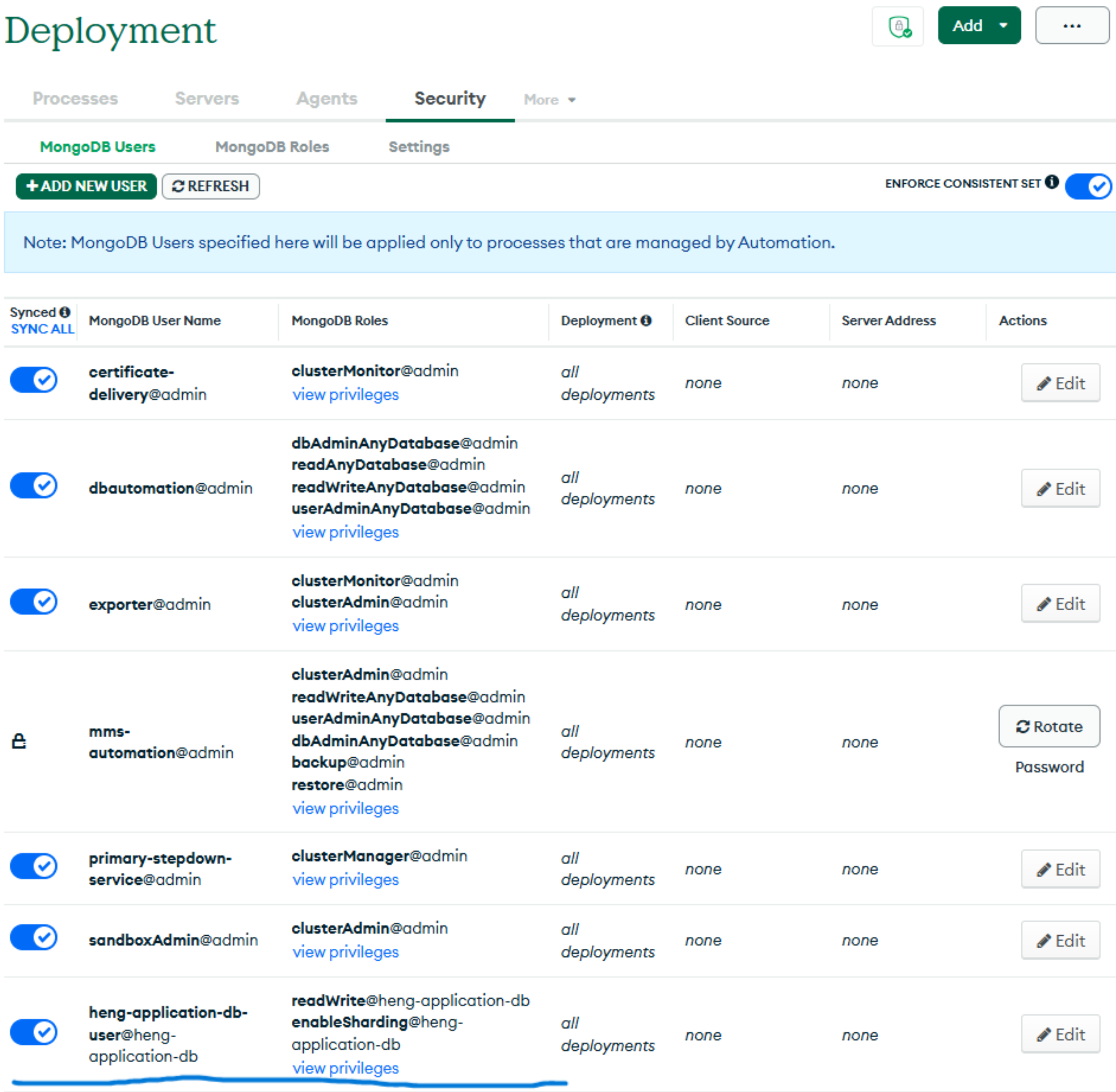

Figure 21. Verification from Opsmanager view

The new user in blue line is the database requested from customized resource, also the operator has reached out to the correct opsmanager instance. Based on the verification from logs and opsmanger, it comes a conclusion that operator has fetched the info from customized resource correctly and successfully created the database.

5.3.5 Problems met during the development/test

Despite business logic handling and go lang compiling errors (syntax errors mainly), there are also following problems when trying to integrate the operator with Mongo part:

- Invalid configmap/secret

The value in a Kubernetes configmap/secret must be base64 encoded and be very careful with any blank/line break handling, ensure the Opsmanager configurations/ related credentials have been correctly encoded.

- Connection failure

The application may complain about connection refused by the opsmanager; this is because the pod Ip address needs to be whitelisted from opsmanager api key user side, or the Openshift cluster Ips needs to be whitelisted from opsmanager web service network.

- Error while updating automation config

There will be errors in pod logs like following:

```
Invalid config: defaultRWConcern.defaultWriteConcern cannot be set
from
{"defaultReadConcern":{"level":"local"},"defaultWriteConcern":{"w"
:1,"wtimeout":0}} to null/empty
```

This is advising the Opsmanger version in go library file go.mod needs to be updated, e.g.

update “go.mongodb.org/ops-manager v0.25.0” to “go.mongodb.org/ops-manager v0.61.0”

- Secret creation failure

There will be errors in pod logs like “secret creation not permitted”, and this is because resources “secret” does not have “create” added into the verbs in cluster-role manifest file. Additionally, if any Kubernetes related permission denied errors found, it could be caused by the according resources/verbs are missing from the cluster-role definition.

- Pod crush

The operator pod may end up with crash loop after it starts to handle a database request, and out of memory exceptions can be found from the pod event, this is due to too little memory allocated to the pod to handle the business. To re-configure the memory allocation, the requests (minimum) and limits(maximum) memory need to be increased in the deployment manifest file then restart the deployment. And the value of memory to be adjusted is dependent on the practice observation.

6. Conclusions and Possible future improvements

In conclusion, the OpenShift Operator proves to be a powerful tool that enables database automation (such as MongoDB), significantly reducing manual effort and minimizing the risk of human errors. Building upon the capabilities provided by the OpenShift Operator framework, my project introduced several customizations, including deeper integration with MongoDB, support for reading environment variables, and logic for inserting new databases. Additionally, deployment automation was implemented to allow seamless rollout across multiple environments—such as test, pre-production, and production—with environment-specific configurations. The results demonstrate that this prototype successfully achieved the intended goals and validated the feasibility of the proposed design.

Although the prototype has demonstrated its effectiveness and achieved the intended design goals, several areas for improvement have been identified during practical use. This section discusses potential enhancements that could further strengthen the reliability, security, and maintainability of the Operator.

- Watch namespace to be fetched from a ConfigMap

In the current implementation, the Operator watches all namespaces within an OpenShift instance to detect new database requests. This approach introduces potential security risks, as unauthorized or unsafe requests might originate from less secure namespaces. To address this, the Operator should be modified to read a list of authorized namespaces from a ConfigMap. This ConfigMap would explicitly define which namespaces are permitted to be monitored and would be protected through appropriate access control, ensuring that only privileged users can modify it.

- Version control with Helm

The existing design lacks a formal mechanism for version management. As the Operator evolves, future updates, bug fixes, or feature enhancements will require structured upgrade and rollback procedures. Integrating Helm would provide a reliable solution for version control, allowing controlled deployment, upgrade, and downgrade processes across different environments.

- Database deletion and smoke test mechanism

To enhance system reliability and observability, a heartbeat or smoke testing mechanism should be introduced. With the addition of a database deletion feature (implemented via a finalizer), a scheduled Jenkins job could periodically create and delete test database requests. This process would validate that both the provisioning and deletion workflows are functioning correctly. Any failures detected during these tests could trigger alerts or notifications to administrators, enabling proactive issue detection.

- Error handling and notification

Improving error-handling mechanisms is another key priority. When exceptions occur, it is important that system administrators are notified promptly with sufficient diagnostic information. Comprehensive error-handling routines should be implemented across all critical code paths. Automated notifications—such as emails or ServiceNow incident generation—could be deployed, including detailed contextual information such as the failed step and relevant logs to facilitate rapid troubleshooting.

- Image building efficiency

During testing, it was found that building the Operator image using Docker takes approximately ten minutes. This long build time hinders rapid development and testing. OpenShift's Source-to-Image (S2I) feature offers a much faster alternative, capable of building new Operator images in roughly two minutes. Migrating to S2I would significantly reduce build time, streamline the development cycle, and improve productivity.

- Loading configmap and secret too many times

Currently, for each new database request, the Operator retrieves configuration and secret data from Ops Manager via the event loader. This design increases both response latency and resource consumption (CPU and memory). A more efficient approach would be to embed the necessary Ops Manager configuration and secrets directly into the Operator's deployment manifest. This would allow the Operator Pod to load them once at startup, rather than fetching them repeatedly for each request. Such optimization would improve runtime efficiency and scalability in production deployments.

References

Aniket Gawandar. (2024, June 14). Containers (Docker) vs Virtual Machines (VM): Pros, Cons, and Use Cases. https://k21academy.com/docker-kubernetes/docker-vs-virtual-machine/

Bob Reselman& Don Schenck. (2023, June 27). OpenShift and the Developer Sandbox. https://developers.redhat.com/learning/learn:openshift:foundations-openshift/resource/resources:openshift-and-developer-sandbox

Carmen Carrion. (2022, December 06). Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis. https://link.springer.com/article/10.1007/s10723-022-09629-8

ConfigMaps. (2024, September 11). https://kubernetes.io/docs/concepts/configuration/configmap/

Creating your first Pipeline. (n.d.) https://www.jenkins.io/doc/pipeline/tour/hello-world/

Deployments. (2025, August 29). https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Difference Between Relational and Non-Relational Database. (2022, October 26). https://www.nebula-graph.io/posts/difference-between-relational-and-non-relational-database

Install a Simple Test Ops Manager Installation. (n.d.) https://www.mongodb.com/docs/ops-manager/current/tutorial/install-simple-test-deployment/

Kubernetes Overview. (2024, September 11). https://kubernetes.io/docs/concepts/overview/

Namespaces. (2024, September 03). https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

OpsManager Overview. (n.d.) https://www.mongodb.com/docs/ops-manager/current/application/

Pods. (2025, July 28). https://kubernetes.io/docs/concepts/workloads/pods/

Rafal Leszko. (2022, n.d.). Continuous Delivery with Docker and Jenkins: Create secure applications by building complete CI/CD pipelines. https://ieeexplore.ieee.org/document/10162439

Secrets. (2024, November 19). https://kubernetes.io/docs/concepts/configuration/secret/

Service accounts. (2024, November 19). https://kubernetes.io/docs/concepts/security/service-accounts/

Understanding Operators. (n.d.) https://docs.redhat.com/en/documentation/openshift_container_platform/4.19/html/operators/understanding-operators

Using RBAC Authorization. (2025, July 03). https://kubernetes.io/docs/reference/access-authn-authz/rbac/

What is MongoDB. (n.d.) https://www.mongodb.com/docs/manual/

What is docker. (n.d.) https://docs.docker.com/get-started/docker-overview/

What’s the Difference Between Relational and Non-relational Databases. (n.d.). Relational vs Nonrelational Databases - Difference Between Types of Databases - AWS

Appendices

Appendix 1. Main function of the operator

```
package main


import (
    "flag"
    "os"


    // Import all Kubernetes client auth plugins (e.g. Azure, GCP,
OIDC, etc.)
    _ "k8s.io/client-go/plugin/pkg/client/auth"


    "k8s.io/apimachinery/pkg/runtime"
    utilruntime "k8s.io/apimachinery/pkg/util/runtime"
    clientgoscheme "k8s.io/client-go/kubernetes/scheme"
    ctrl "sigs.k8s.io/controller-runtime"
    "sigs.k8s.io/controller-runtime/pkg/healthz"
    "sigs.k8s.io/controller-runtime/pkg/log/zap"


    mongodbv1 "mongodb-operator/api/v1"
    "mongodb-operator/controllers"
)


var (
    scheme   = runtime.NewScheme()
    setupLog = ctrl.Log.WithName("setup")
)
```

```
func init() {
    utilruntime.Must(clientgoscheme.AddToScheme(scheme))
    utilruntime.Must(mongodbv1.AddToScheme(scheme))
}

func main() {
    var metricsAddr string
    var enableLeaderElection bool
    var probeAddr string

    flag.StringVar(&metricsAddr, "metrics-bind-address", ":8080",
"The address the metric endpoint binds to.")
    flag.StringVar(&probeAddr, "health-probe-bind-address",
":8081", "The address the probe endpoint binds to.")
    flag.BoolVar(&enableLeaderElection, "leader-elect", false,
        "Enable leader election for controller manager. "+
            "Enabling this will ensure there is only one active
controller manager.")

    opts := zap.Options{
        Development: true,
    }
    opts.BindFlags(flag.CommandLine)
    flag.Parse()

    ctrl.SetLogger(zap.New(zap.UseFlagOptions(&opts)))

    // Set empty watchNamespace to watch all namespaces
    watchNamespace := ""
```

```
    // Setup manager with empty namespace to watch all namespaces
    mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(),
ctrl.Options{
        Scheme:                 scheme,
        MetricsBindAddress:     metricsAddr,
        HealthProbeBindAddress: probeAddr,
        Port:                   9443,
        LeaderElection:         enableLeaderElection,
        LeaderElectionID:       "mongodb-operator-leader-election-
helper",
        Namespace:              watchNamespace, // Empty namespace
watches all namespaces
    })
    if err != nil {
        setupLog.Error(err, "unable to start manager")
        os.Exit(1)
    }

    if err = (&controllers.testPTMongoDBReconciler{
        Client: mgr.GetClient(),
        Scheme: mgr.GetScheme(),
    }).SetupWithManager(mgr); err != nil {
        setupLog.Error(err, "unable to create controller",
"controller", "testPTMongoDB")
        os.Exit(1)
    }

    if err := mgr.AddHealthzCheck("healthz", healthz.Ping); err !=
nil {
        setupLog.Error(err, "unable to set up health check")
        os.Exit(1)
```

```
    }
    if err := mgr.AddReadyzCheck("readyz", healthz.Ping); err !=
nil {
        setupLog.Error(err, "unable to set up ready check")
        os.Exit(1)
    }

    setupLog.Info("starting manager")
    if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {
        setupLog.Error(err, "problem running manager")
        os.Exit(1)
    }
}
```

Appendix 2. Code of controller

```
// filepath: c:\VSWorkspace\openshift-database-
automation\controllers\databaserequest_controller.go

package controllers


import (
    "context"
    "fmt"
    "os"
    "strings"
    "time"

    corev1 "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/api/errors"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/runtime"
    ctrl "sigs.k8s.io/controller-runtime"
    "sigs.k8s.io/controller-runtime/pkg/client"
    "sigs.k8s.io/controller-runtime/pkg/log"

    mongodbv1 "mongodb-operator/api/v1"
    "mongodb-operator/pkg/opsmanager"
    "mongodb-operator/pkg/environment"
)

// MongoDBReconciler reconciles a MongoDB object
type MongoDBReconciler struct {
```

```
    client.Client
    Scheme *runtime.Scheme
}

// getOperatorNamespace returns the namespace where the operator
is deployed
// It reads from OPERATOR_NAMESPACE environment variable
func (r *MongoDBReconciler) getOperatorNamespace() string {
    return os.Getenv("OPERATOR_NAMESPACE")
}

func (r *MongoDBReconciler) Reconcile(ctx context.Context, req
ctrl.Request) (ctrl.Result, error) {
    logger := log.FromContext(ctx)
    logger.Info("Reconciling MongoDB", "request",
req.NamespacedName)

    // Fetch the MongoDB instance
    dbRequest := &mongodbv1.MongoDB{}
    err := r.Get(ctx, req.NamespacedName, dbRequest)
    if err != nil {
        if errors.IsNotFound(err) {
            // Request object not found, could have been deleted
after reconcile request
            return ctrl.Result{}, nil
        }
        // Error reading the object - requeue the request
        return ctrl.Result{}, err
    }
```

```
    // Load environment config from operator's deployment
namespace

        if err != nil {

        if errors.IsNotFound(err) {

            // Request object not found, could have been deleted
after reconcile request

            return ctrl.Result{}, nil

        }

        // Error reading the object - requeue the request

        return ctrl.Result{}, err

    }


    // ConfigMaps and secrets are created by Jenkins in the
operator's namespace, not the MongoDB's namespace

    operatorNamespace := r.getOperatorNamespace()

    logger.Info("Loading environment config", "envSymbol",
dbRequest.Spec.EnvSymbol, "operatorNamespace", operatorNamespace)



    //are we loading config too many times?

    envConfig, err := environment.LoadConfig(r.Client,
operatorNamespace, dbRequest.Spec.EnvSymbol)

    if err != nil {

        logger.Error(err, "Failed to load environment config",
"envSymbol", dbRequest.Spec.EnvSymbol, "operatorNamespace",
operatorNamespace)

        return ctrl.Result{}, err

    }
```

```
    logger.Info("Environment config loaded successfully",
"envSymbol", dbRequest.Spec.EnvSymbol)


    // Skip status checks and go directly to MongoDB operations
for testing

    logger.Info("Starting MongoDB database creation", "mongodb",
dbRequest.Spec.MongoDB, "clusterType", dbRequest.Spec.ClusterType)


    // Create MongoDB database

    logger.Info("Creating OpsManager client")

    logger.Info("Environment config details",

        "opsManagerURL", envConfig.OpsManager.URL,

        "hasPublicKey", envConfig.OpsManager.PublicKey != "",

        "hasPrivateKey", envConfig.OpsManager.PrivateKey != "",

        "envSymbol", envConfig.EnvSymbol)


    opsman := opsmanager.NewClient(envConfig)

    if opsman == nil {

        logger.Error(nil, "Failed to create OpsManager client -
check URL and credentials",

            "opsManagerURL", envConfig.OpsManager.URL,

            "hasPublicKey", envConfig.OpsManager.PublicKey != "",

            "hasPrivateKey", envConfig.OpsManager.PrivateKey !=
"")

        return ctrl.Result{RequeueAfter: time.Minute * 5}, nil

    }


    logger.Info("OpsManager client created successfully")

    logger.Info("Calling OpsManager CreateDatabase", "mongodb",
dbRequest.Spec.MongoDB, "clusterType", dbRequest.Spec.ClusterType)
```

```
    result, err := opsman.CreateDatabase(ctx,
dbRequest.Spec.MongoDB, dbRequest.Spec.ClusterType)

    if err != nil {
        logger.Error(err, "Failed to create MongoDB database")
        return ctrl.Result{RequeueAfter: time.Minute * 5}, nil
    }

    logger.Info("MongoDB database created successfully", "result",
result)

    // Create secret with credentials
    logger.Info("Creating credentials secret")
    err = r.createCredentialsSecret(ctx, dbRequest, result)
    if err != nil {
        logger.Error(err, "Failed to create credentials secret")
        return ctrl.Result{}, err
    }

    logger.Info("Credentials secret created successfully")
    logger.Info("MongoDB provisioning completed successfully")

    return ctrl.Result{}, nil
}

// createCredentialsSecret creates a secret with database
credentials
func (r *MongoDBReconciler) createCredentialsSecret(ctx
context.Context, dbRequest *mongodbv1.MongoDB, connInfo
*opsmanager.ConnectionInfo) error {
```

```
    // Create secret data
    secretData := make(map[string][]byte)

    // Add connection info
    secretData["host"] = []byte(connInfo.Host)
    secretData["port"] = []byte(fmt.Sprintf("%d", connInfo.Port))
    secretData["database"] = []byte(connInfo.Database)

    // Add user credentials
    for i, user := range connInfo.Users {
        secretData[fmt.Sprintf("username_%d", i)] =
[]byte(user.Username)
        secretData[fmt.Sprintf("password_%d", i)] =
[]byte(user.Password)
        secretData[fmt.Sprintf("roles_%d", i)] =
[]byte(strings.Join(user.Roles, ","))
    }

    // Add main user credentials for convenience
    if len(connInfo.Users) > 0 {
        secretData["username"] =
[]byte(connInfo.Users[0].Username)
        secretData["password"] =
[]byte(connInfo.Users[0].Password)
    }

    // Create the Secret object
    secret := &corev1.Secret{
        ObjectMeta: metav1.ObjectMeta{
```

```
            Name:      fmt.Sprintf("%s-mongodb-credentials",
dbRequest.Name),
            Namespace: dbRequest.Namespace,
            OwnerReferences: []metav1.OwnerReference{
                {
                    APIVersion: dbRequest.APIVersion,
                    Kind:       dbRequest.Kind,
                    Name:       dbRequest.Name,
                    UID:        dbRequest.UID,
                    Controller: &[]bool{true}[0],
                },
            },
        },
        Type: corev1.SecretTypeOpaque,
        Data: secretData,
    }

    // Create the secret in the cluster (or update if it already
exists)
    err := r.Create(ctx, secret)
    if err != nil {
        if errors.IsAlreadyExists(err) {
            // Secret already exists, update it instead
            return r.Update(ctx, secret)
        }
        return err
    }
    return nil
}
```

```
// SetupWithManager sets up the controller with the Manager.

func (r *MongoDBReconciler) SetupWithManager(mgr ctrl.Manager)
error {

    return ctrl.NewControllerManagedBy(mgr).

        For(&mongodbv1.MongoDB{}).

        Complete(r)

}
```

Appendix 3. Code of ospamanger client

```
package opsmanager


import (
	"context"
	"crypto/rand"
	"crypto/tls"
	"fmt"
	"log"
	"math/big"
	"net/http"
	"os"
	"strings"

	"github.com/mongodb-forks/digest"
	"go.mongodb.org/ops-manager/opsmngr"

	mongodbv1 "mongodb-operator/api/v1"
	"mongodb-operator/pkg/environment"
)

// Client is MongoDB Ops Manager client
type Client struct {
	client            *opsmngr.Client
	projectCache      map[string]string
	environmentPrefix string
	config            *environment.Config
	httpClient        *http.Client
```

```
}

// ConnectionInfo holds database connection information
type ConnectionInfo struct {
    Host     string
    Port     int32
    Database string
    Users    []UserInfo
    URI      string
}

// UserInfo holds user credential information
type UserInfo struct {
    Username string
    Password string
    Roles    []string
}

// NewClient creates a new MongoDB Ops Manager client
func NewClient(config *environment.Config) *Client {
    url := os.Getenv("OPSMANAGER_URL")
    if url == "" {
        url = config.OpsManager.URL
        if url == "" {
            return nil
        }
    }

    publicKey := os.Getenv("OPSMAN_PUBLIC_KEY")
```

```
    if publicKey == "" {
        publicKey = config.OpsManager.PublicKey
    }

    privateKey := os.Getenv("OPSMAN_PRIVATE_KEY")
    if privateKey == "" {
        privateKey = config.OpsManager.PrivateKey
    }

    tlsConfig := &tls.Config{InsecureSkipVerify: true}
    httpTransport := &http.Transport{TLSClientConfig: tlsConfig}

    t := digest.NewTransport(publicKey, privateKey)
    t.Transport = httpTransport
    tc := &http.Client{Transport: t}

    client, err := opsmngr.New(tc, opsmngr.SetBaseURL(url))
    if err != nil {
        log.Printf("Failed to create Ops Manager client: %v", err)
        return nil
    }

    projs, _, err := client.Projects.List(context.Background(),
nil)
    if err != nil {
        log.Printf("Error reading OpsManager project list: %v",
err)
        return nil
    }
```

```
    result := &Client{
        client:            client,
        projectCache:      make(map[string]string),
        environmentPrefix: strings.ToLower(config.EnvSymbol),
        config:            config,
        httpClient:        tc,
    }

    for _, o := range projs.Results {
        log.Printf("Found project: %s (ID: %s)", o.Name, o.ID)
        result.projectCache[o.Name] = o.ID
    }

    return result
}

// CreateDatabase provisions a MongoDB database and returns
connection information
func (c *Client) CreateDatabase(ctx context.Context, spec
mongodbv1.MongoDBSpec, clusterType string) (*ConnectionInfo,
error) {
    dbName := c.getExpectedName(spec.Name)

    // Get cluster ID from environment config using clusterType
    clusterID, exists := c.config.MongoDBClusters[clusterType]
    if !exists {
        return nil, fmt.Errorf("cluster type %s not found in
environment config", clusterType)
    }
```

```
    projectID, ok := c.projectCache[clusterID]
    if !ok {
        return nil, fmt.Errorf("cluster %s not found in project
cache", clusterID)
    }

    // Check if database already exists
    exists, err := c.checkDatabaseExists(ctx, projectID, dbName)
    if err != nil {
        return nil, err
    }

    if exists {
        return nil, fmt.Errorf("database %s already exists",
dbName)
    }

    // Get automation config
    config, _, err := c.client.Automation.GetConfig(ctx,
projectID)
    if err != nil {
        return nil, fmt.Errorf("failed to get automation
config: %w", err)
    }

    // Generate password for main user
    password, err := GenerateRandomString(24)
    if err != nil {
        return nil, fmt.Errorf("failed to generate password: %w",
err)
    }
```

```
    // Create database user using the working pattern from
reference code
    var newDB opsmngr.MongoDBUser
    newDB.Database = dbName
    newDB.InitPassword = password
    newDB.Roles = []*opsmngr.Role{
        {Role: "readWrite", Database: dbName},
        {Role: "enableSharding", Database: dbName},
    }
    newDB.Username = dbName
    newDB.Mechanisms = &[]string{}
    newDB.AuthenticationRestrictions =
[]opsmngr.AuthenticationRestriction{}

    // Add main user to automation config
    config.Auth.UsersWanted = append(config.Auth.UsersWanted,
&newDB)
    //list the config
    log.Printf("Automation config after adding main user: %+v",
config.Auth.UsersWanted)

    // Create users list with only the main database user
    users := []UserInfo{{
        Username: dbName,
        Password: password,
        Roles:    []string{"readWrite", "enableSharding"},
    }}

    // Get MongoDB server instances (mongos instances) before
attempting user creation
```

```
    mongosInstances := c.getMongosInstances(config)
    if mongosInstances == "" {
        return nil, fmt.Errorf("no mongos instances found in cluster")
    }

    // Try to update automation config with new users first
    log.Printf("Attempting to update automation config to add database users")
    _, err = c.client.Automation.UpdateConfig(ctx, projectID, config)

    if err != nil {
        log.Printf("Automation config update failed: %v", err)
    } else {
        log.Printf("Successfully updated automation config with database users")
    }

    // Parse the first instance for host info
    parts := strings.Split(mongosInstances, ",")
    firstInstance := parts[0]
    host := strings.Split(firstInstance, ":")[0]

    // Create connection info
    connectionInfo := &ConnectionInfo{
        Host:     host,
        Port:     27017,
        Database: dbName,
        Users:    users,
```

```
        URI:      fmt.Sprintf("mongodb://%s:%s@%s/%s?ssl=true&tls=
true&serverSelectionTimeoutMS=3000", dbName, password,
mongosInstances, dbName),
    }

    return connectionInfo, nil
}

// checkDatabaseExists checks if a database already exists in the
cluster
func (c *Client) checkDatabaseExists(ctx context.Context,
projectID, dbName string) (bool, error) {
    config, _, err := c.client.Automation.GetConfig(ctx,
projectID)
    if err != nil {
        return false, fmt.Errorf("failed to get automation
config: %w", err)
    }

    for _, user := range config.Auth.UsersWanted {
        if user.Database == dbName {
            return true, nil
        }
    }

    return false, nil
}

// getExpectedName returns the expected database name
func (c *Client) getExpectedName(appName string) string {
    return fmt.Sprintf("%s-%s", c.environmentPrefix, appName)
```

```
}

// getMongosInstances returns a comma-separated list of mongos
instances
func (c *Client) getMongosInstances(config
*opsmngr.AutomationConfig) string {
    var endpoints []string
    for _, instance := range config.Processes {
        if instance.ProcessType == "mongos" {
            endpoints = append(endpoints, fmt.Sprintf("%s:27017",
instance.Hostname))
        }
    }
    return strings.Join(endpoints, ",")
}

// cleanupMongoDBVersions removes "-ent" suffix from MongoDB
versions that cause API errors
func (c *Client) cleanupMongoDBVersions(config
*opsmngr.AutomationConfig) {
    // Update MongoDBVersions array - only clean "-ent" suffix,
don't force specific version
    if config.MongoDBVersions != nil {
        for _, versionMap := range config.MongoDBVersions {
            if versionMap != nil {
                if name, ok := (*versionMap)["name"].(string); ok
&& strings.Contains(name, "-ent") {
                    cleanVersion := strings.Replace(name, "-ent",
"", 1)
                    log.Printf("Cleaned MongoDB version: %s
-> %s", name, cleanVersion)
                    (*versionMap)["name"] = cleanVersion
```

```
                }
            }
        }
    }

    // Update Process versions - only clean "-ent" suffix, don't
force specific version
    for _, process := range config.Processes {
        if process != nil && strings.Contains(process.Version, "-
ent") {
            originalVersion := process.Version
            cleanVersion := strings.Replace(originalVersion, "-
ent", "", 1)
            process.Version = cleanVersion
            log.Printf("Cleaned process %s version: %s -> %s",
process.Name, originalVersion, cleanVersion)
        }
    }
}

// GenerateRandomString generates a random string of the specified
length
func GenerateRandomString(n int) (string, error) {
    const letters =
"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz-"
    ret := make([]byte, n)
    for i := 0; i < n; i++ {
        num, err := rand.Int(rand.Reader,
big.NewInt(int64(len(letters))))
        if err != nil {
            return "", err
        }
```

```
        ret[i] = letters[num.Int64()]
    }
    return string(ret), nil
}
```

Appendix 4. Code of environment loader

```
package environment


import (
    "context"
    "fmt"
    "strings"

    corev1 "k8s.io/api/core/v1"
    "sigs.k8s.io/controller-runtime/pkg/client"
)

// Config represents environment configuration
type Config struct {
    EnvSymbol       string
    OpsManager      OpsManagerConfig
    MongoDBClusters map[string]string
}

// OpsManagerConfig represents MongoDB Ops Manager configuration
type OpsManagerConfig struct {
    URL        string
    PublicKey  string
    PrivateKey string
}

// LoadConfig now takes explicit envSymbol parameter and loads
credentials from secret
```

```
func LoadConfig(k8sClient client.Client, namespace string,
envSymbol string) (*Config, error) {
    if envSymbol == "" {
        return nil, fmt.Errorf("environment symbol cannot be
empty")
    }

    // Load environment config from ConfigMap
    configMap := &corev1.ConfigMap{}
    err := k8sClient.Get(
        context.TODO(),
        client.ObjectKey{
            Namespace: namespace,
            Name:      "mongodb-operator-" + envSymbol,
        },
        configMap,
    )

    if err != nil {
        return nil, fmt.Errorf("failed to load config for
env %s: %w", envSymbol, err)
    }

    // Load credentials from Secret
    credsSecret := &corev1.Secret{}
    err = k8sClient.Get(
        context.TODO(),
        client.ObjectKey{
            Namespace: namespace,
```

```
            Name:      "mongodb-operator-credentials-" +
envSymbol,
        },
        credsSecret,
    )

    if err != nil {
        return nil, fmt.Errorf("failed to load operator
credentials: %w", err)
    }

    // Create config object
    cfg := &Config{
        EnvSymbol:       envSymbol,
        MongoDBClusters: make(map[string]string),
        OpsManager: OpsManagerConfig{
            URL: configMap.Data["OPSMANAGER_URL"],
            // Get credentials from secret
            PublicKey:  string(credsSecret.Data["OPSMAN_PUBLIC_KEY
"]),
            PrivateKey:
string(credsSecret.Data["OPSMAN_PRIVATE_KEY"]),
        },
    }

    // If OPSMANAGER_URL is also in the secret, use that instead
    if opsmanagerURL, ok := credsSecret.Data["OPSMANAGER_URL"]; ok
&& len(opsmanagerURL) > 0 {
        cfg.OpsManager.URL = string(opsmanagerURL)
    }
```

```
    // Parse cluster mappings
    for key, val := range configMap.Data {
        if strings.HasPrefix(key, "MONGODB_CLUSTER_") {
            clusterName := strings.TrimPrefix(key,
"MONGODB_CLUSTER_")
            cfg.MongoDBClusters[clusterName] = val
        }
    }

    return cfg, nil
}
```

Appendix 5. Code of jenkins file

```
@Library('testpt-pipeline-libs') _


withFolderProperties{
    environment = env.environment.toLowerCase()
    switch(environment) {

        case 'test-pp':
            namespace = "testpt-ci"

        break
        case 'test-p':
            namespace = "testpt-ci"

        break
        default:
            namespace = "testpt-ci"
    }
}

switch(testptEnvironment.getCurrent().cluster) {
     case "pci_prod":
        cluster_list = ["PCI_PROD"]
     break
     case "pci_no_prod":
        cluster_list = ["PCI"]
     break
    case "test":
```

```
        cluster_list = ["TEST"]
     break
     default:
        cluster_list = []
     break
}

properties([
    [$class: 'BuildDiscarderProperty', strategy: [$class:
'LogRotator', artifactDaysToKeepStr: '', artifactNumToKeepStr: '',
daysToKeepStr: '', numToKeepStr: '5']]
])

def version = "default"

testpt.timeBuild('golang') {
    stage('Prepare') {
        cleanWs notFailBuild: true
        checkout scm
        version = testpt.sh "git rev-parse --short HEAD"
    }

    stage('Compile Test') {
        sh """
            # Configure Go proxy and disable checksum verification
for corporate environment
            export
GOPROXY=https://nexus.xxx.no:9006/repository/go-proxy/,direct
            export GOSUMDB=off
            export GOPRIVATE=nets.eu/*
```

```
            export GOTOOLCHAIN=local

            export CGO_ENABLED=0

            export GOOS=linux

            export GO111MODULE=on


            # Install controller-gen compatible with Go 1.23 and
controller-runtime v0.13.0

            go install sigs.k8s.io/controller-
tools/cmd/controller-gen@v0.9.2


            # Add GOPATH/bin to PATH so controller-gen can be
found

            export PATH=\$PATH:\$(go env GOPATH)/bin


            # Ensure all dependencies are downloaded and go.sum is
updated

            go mod download

            go mod tidy


            # Vendor dependencies to avoid download issues in
Docker build

            go mod vendor


            # Try to generate DeepCopy methods, but continue even
if it fails

            controller-gen
object:headerFile="hack/boilerplate.go.txt" paths="./..." || echo
"Warning: controller-gen failed, continuing without generated
code"


            # Test compilation to catch errors early (after code
generation)

            go mod tidy
```

```
            go build -o /tmp/test-binary main.go
            rm -f /tmp/test-binary

            echo "Compilation test passed"
        """
    }

    stage('Build image') {
        if (cluster_list.isEmpty()) {
            echo "Warning: cluster_list is empty. Using default
test environment."
            cluster_list = ["TEST"]
        }

        cluster_list.each { cluster ->
            platform.login(cluster)
            sh """
                # Create Docker BuildConfig if it doesn't exist
                oc -n ${namespace} new-build --strategy=docker --
binary=true --name=mongodb-operator --to=image-registry.openshift-
image-registry.svc:5000/${namespace}/mongodb-operator:latest ||
true

                # Start Docker build from current directory
                oc -n ${namespace} start-build mongodb-operator --
from-dir=. --follow --wait
                oc -n ${namespace} tag mongodb-operator:latest
mongodb-operator:${version}
            """
        }
    }
```

```
    stage('Deploy Operator') {
        if (cluster_list.isEmpty()) {
            echo "Warning: cluster_list is empty. Using default test environment."
            cluster_list = ["TEST"]
        }

        cluster_list.each { cluster ->
            platform.login(cluster)
            sh """
                # Create operator namespace resources
                oc -n ${namespace} apply -f config/rbac/service_account.yaml
                oc -n ${namespace} apply -f config/rbac/role.yaml
                oc -n ${namespace} apply -f config/rbac/role_binding.yaml

                # Create config map for environment configurations
                oc -n ${namespace} apply -f config/env-configs/${environment}-config.yaml

                # Deploy operator with correct image reference (use image stream in same namespace)
                sed 's|mongodb-operator:@@VERSION@@|image-registry.openshift-image-registry.svc:5000/'${namespace}'/mongodb-operator:latest|g' config/deploy/operator.yml | oc -n ${namespace} apply -f -

                # Wait for deployment to be ready
                oc -n ${namespace} rollout status deployment/mongodb-operator --timeout=300s
```

```
                    # Verify deployment
                    oc -n ${namespace} get pods -l app=mongodb-
operator
                """
            }
        }
}
```

Acknowledgement

Sincere appreciation is extended to **Professor Timo Mantere** and **Professor Mohammed Elmusrati**, whose valuable guidance, insightful ideas, and constructive feedback greatly contributed to the successful completion of this thesis.