Expert's Guide to Running MongoDB on Kubernetes

portworx by Pure Storage

mongoDB.

1. INTRODUCTION 4

2. MONGODB ARCHITECTURE 5

2.1 Flexible data model 5
2.2 Replication 6
2.3 Horizontal scalability 7

3. MONGODB CONFIGURATION BEST PRACTICES 7

3.1 General system settings 7
3.1.1 CPU 7
3.1.2 RAM 8
3.1.3 Swap strategy 8
3.1.4 Storage 8
3.2 Linux configuration 8
3.2.1 Turn off Transparent Huge Pages (THP) 8
3.2.2 Number of open file descriptors 8
3.2.3 Number of connections 9
3.2.4 Other system settings 9
3.3 Replica set design considerations 9
3.3.1 Data-bearing nodes 9
3.3.2 Enable journaling 9
3.3.3 Configure write concern 9
3.3.4 Load balance on read-heavy deployments 9
3.3.5 Distribute replicas across datacenters 10
3.3.6 Prepare capacity ahead of demand 10
3.3.7 Oplog size 10
3.3.8 Control replication lag 10
3.4 Sharded cluster configuration 10
3.4.1 Select a proper shard key 10
3.4.2 Shard key cardinality 11
3.4.3 Shard key frequency 11
3.4.4 Shard key rate of change 12
3.4.5 Chunks size 12

4. BEST PRACTICES FOR RUNNING MONGODB ON KUBERNETES 12

4.1 MongoDB Kubernetes deployment options 12
4.1.1 Manual deployment 12
4.1.2 MongoDB deployment via Helm charts 13
4.1.3 Using MongoDB Operators 13
4.2 Managing MongoDB deployment on Kubernetes with Portworx 14
4.3 Enabling MongoDB local HA for Kubernetes 15
4.4 MongoDB backup options for Kubernetes 16
4.4.1 Backup with cp or rsync 16
4.4.2 Filesystem snapshots 16
4.4.3 Mongodump and mongorestore 17

4.5 Backing up MongoDB data with PX-Backup 17
4.5.1 Application-consistent backups 18
4.5.2 Kubernetes-aware backups 19
4.5.3 Kubernetes namespace-aware backups 19
4.5.4 Multi-cluster support 20
4.6 MongoDB disaster recovery with Portworx 20
4.6.1 Synchronous cross-datacenter DR 21
4.6.2 Asynchronous cross-datacenter disaster recovery with Portworx 22
4.7 MongoDB security 23
4.7.1 Authentication 23
4.7.2 Authorization 23
4.7.3 TLS/SSL encryption 24
4.7.4 Encryption at rest 24
4.7.5 Application-layer encryption 24
4.8 Hardening MongoDB security with Portworx 24
4.8.1 Authentication and RBAC with Kubernetes and Portworx 25
4.8.2 Encryption of Kubernetes resources 26
4.9 MongoDB monitoring 26
4.9.1 Monitoring MongoDB in Kubernetes 27
PX-Autopilot 29

5. CONCLUSION 30

WHO SHOULD READ THIS GUIDE?

Platform Architects building a Container as a Service (CaaS) platform that offers MongoDB to end users

Application Architects or **Site Reliability Engineers (SRE)** building or running a Software as a Service (SaaS) application on Kubernetes that requires a high-performance MongoDB database

Database as a Service (DBaaS) Architects offering multi-tenant MongoDB deployments as a service to end users

Developers building containerized applications for Kubernetes that use MongoDB to store data

1. INTRODUCTION

MongoDB is an open-source NoSQL database that provides many useful features—such as flexible schema design, embedded data models, cross-node replication, horizontal scaling, and even RDBMS-style ACID (Atomicity, Consistency, Isolation, Durability) transactions. According to the DB Engines web site, MongoDB is the most popular NoSQL database, trailing only SQL options PostgreSQL, Microsoft SQL Server, Oracle, and MySQL.

MongoDB's popularity among developers is due to its easy installation, flexible data structure, and comprehensive support for different production use cases. Some of the most valued MongoDB features are:

- **Flexible document data structure.** MongoDB's document data structure corresponds to object data structures common in many programming languages. For example, MongoDB documents are easily convertible to Javascript objects and/or JSON.
- **Dynamic and/or schema-less documents.** MongoDB dynamic schema provides fluid polymorphism that allows changing field data types dynamically and storing different data types (e.g., geo data, rich-text data, unstructured data) in different documents of the same collection.
- **Nested data models.** MongoDB supports nested data structures, which allows implementing parent-children relationships within the document. Data nesting enables atomicity at the document level, which reduces I/O activity on the database system. It also allows implementing "joins" and other traditional RDBMS concepts in the NoSQL environment.
- **Replica sets.** MongoDB can be deployed as a replica set with several data-bearing nodes. This enables data high availability, redundancy, and fault tolerance.
- **Horizontal scaling.** MongoDB collections can be scaled beyond the size of the individual server using sharding. MongoDB horizontal scaling allows distributing I/O load across nodes in the cluster.

Getting performance benefits from these features, however, requires careful configuration and design of the MongoDB cluster. Failure to follow MongoDB best practices may lead to poor performance and possible data loss and data security issues. For example, selecting a low cardinality shard key for a sharded cluster can hinder horizontal scaling and lead to deteriorated database performance (more on this later).

Similarly, MongoDB performance and maintainability are influenced by how and where the database is deployed. Many companies using MongoDB are looking at containers and container orchestration as a way to automate their MongoDB deployments, making CICD pipelines faster and time to market shorter. They also want to take advantage of immense cost savings brought by containers compared to virtual machines. As the most popular container orchestration platform, Kubernetes arises as a natural choice for companies undergoing a cloud-native transformation.

However, deploying and managing MongoDB in Kubernetes introduces many challenges that we try to address in this paper. The main goal of this paper is to show how companies adopting Kubernetes and running MongoDB can align these systems and what additional tools they can use to meet data protection and HA requirements of the MongoDB deployment. In this paper, we'll discuss the following topics:

- MongoDB architecture
- MongoDB configuration best practices
- MongoDB deployment options for Kubernetes
- Enabling local High Availability (HA) of MongoDB in Kubernetes using Portworx storage aggregation and replication
- Efficient MongoDB backup and recovery in Kubernetes with PX-Backup and PX-DR
- Hardening MongoDB security on Kubernetes with PX-secure storage-layer security
- Integrating MongoDB deployment with the Kubernetes monitoring pipeline and Portworx automation tools

2. MONGODB ARCHITECTURE

MongoDB combines the benefits of relational databases with their expressive query language and strong consistency guarantees with the flexibility and scalability offered by NoSQL databases. In what follows, we discuss the defining features of MongoDB architecture which make the database so popular.

2.1 Flexible data model

The MongoDB data modeling approach is based on the flexible schema design and polymorphic data types. In particular, documents within MongoDB collections do not need to have the same schema design. Each of them may have an individual set of fields with different data types. This schema flexibility facilitates direct mapping of MongoDB documents to entities (e.g., customers, assets) they represent.

Also, MongoDB documents can be nested to create sub-documents with parent-children relationships. These embedded data structures allow modeling the diverse type of relationships in data and implementing relational joins.

In recent releases, MongoDB developers added several SQL features. For example, with the MongoDB 4.0 release that came out in 2018, the database has introduced multi-document ACID transactions that further narrowed the gap between the relational database model and the NoSQL model. Also, the MongoDB 4.2 release introduced distributed ACID transactions across shared clusters. These features make MongoDB a good choice for any kind of application—including e-commerce, financial, and banking applications, which have been traditionally regarded as the domain of relational databases.

2.2 Replication

By design, MongoDB can be deployed as a distributed database cluster with strong High Availability (HA) guarantees. In particular, MongoDB supports multi-node replication through replica sets which provide HA, data redundancy, and automatic failover.

In general, a *replica set* is a group of MongoDB servers storing the same data set. A replica set can contain several data-bearing nodes and, optionally, one or more arbiter nodes. One of the data-bearing nodes is the primary node while the others are secondary nodes. By default, the primary node writes all changes to data to its operation log (oplog). In their turn, secondary nodes replicate the primary's oplog and apply all recorded operations to their data sets. If the primary node becomes unavailable, one of the data-bearing nodes becomes a primary.

To secure the elections quorum, the replica set can have one or more arbiter nodes that participate in elections but don't store replicas. Arbiter nodes are useful if you need additional voters but don't have enough storage or RAM to use them as data-bearing nodes.

In general, MongoDB replica sets provide the following benefits:

- **Faster read performance.** Keeping copies of data on different machines in a cluster means that more clients can access data faster without overloading the database with connections.
- **High Availability and data locality.** Keeping copies of data in different datacenters improves data locality and availability for distributed applications.
- **Multiple uses.** Data replicas can be used for backup, disaster recovery, audit and regulatory compliance.

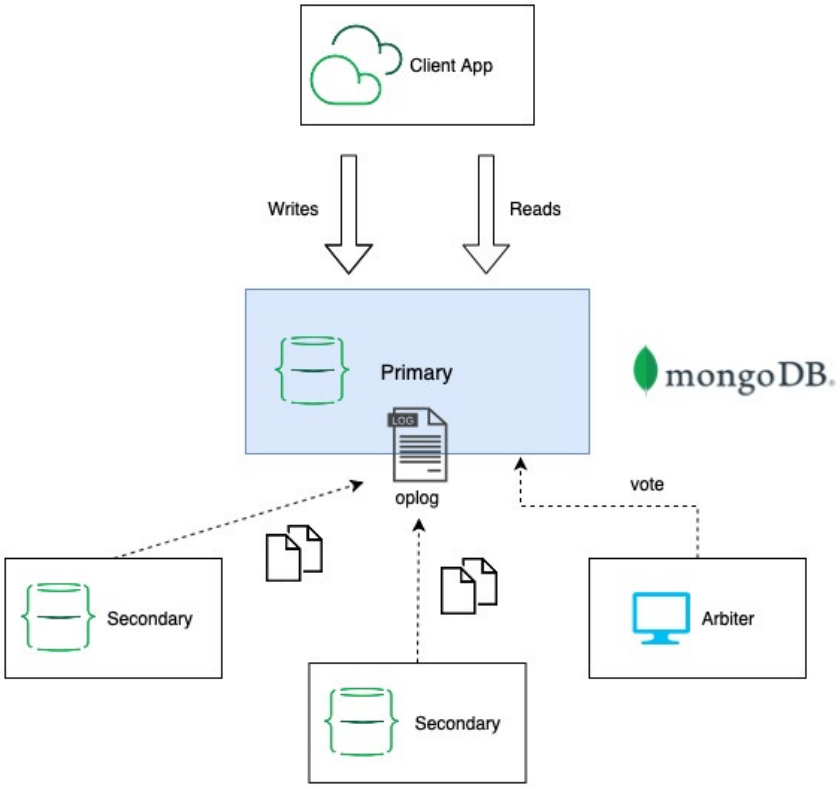

Image 1: MongoDB Replica Set

Also, MongoDB provides a way to tune fault tolerance and data consistency levels using write and read concerns. A write concern describes the level of acknowledgment required to consider the write operation successful. For example, the "majority" write concern requires data to be written by the majority of replicas before returning it to read requests. Through the combination of write and read concerns, one can adjust the consistency level and availability guarantees that reflect the business importance of the data.

2.3 Horizontal scalability

MongoDB supports horizontal scalability via sharding. The latter is the mechanism that involves dividing a dataset into chunks (shards) and distributing them across multiple servers, adding capacity as required.

The motivation behind sharding is simple. Shards allow the size of collections to grow beyond the size of individual servers. This makes it easy to scale collections horizontally by adding new capacity. As an added benefit, sharding improves HA of data and increases the total write throughput. With sharding, each machine handles just a subset of the overall workload, providing better efficiency than a single high-capacity and high-speed server.

A MongoDB sharded cluster consists of the following components:

- **A shard.** This is the data structure that contains a subset of the dataset. Shards can be deployed as replica sets for fault tolerance and HA.
- **Mongos.** This is a query router that navigates the sharded cluster.
- **Config servers.** These servers store shard metadata and configuration settings for the cluster.

MongoDB partitions collections into chunks using ranges of shard key values. The shard key determines the distribution of documents among the cluster's shards. Each chunk has an inclusive lower and exclusive upper range based on the shard key. MongoDB attempts to distribute chunks evenly among the shards using balancers. However, the resultant distribution is affected by the choice of shard key (more on this later).

Also, among other great distributed features of MongoDB one should mention zones. A zone is a logical entity that can be associated with one or more shards in a cluster, and a shard can associate with any number of zones. Some common deployment patterns with zones include isolating a specific subset of data on certain shards, distributing data on shards that are geographically closest to the app servers, and routing data to shards based on the hardware/performance of the nodes that store these shards.

3. MONGODB CONFIGURATION BEST PRACTICES

MongoDB has numerous requirements that should be met for efficient production deployment. These involve general system settings, replica set and shard cluster configuration, and write and read concerns among others. We'll discuss some of the most important configurations and best practices for MongoDB in this section.

3.1 General system settings

3.1.1 CPU

At a minimum, a MongoDB process must have access to two real cores or one multi-core physical CPU. The default WiredTiger storage engine is multithreaded so it can benefit from additional CPU cores.

3.1.2 RAM

For the WiredTiger storage engine (default in MongoDB 3.2), MongoDB utilizes both the WiredTiger internal cache and the filesystem cache.

The default WireTiger internal cache (starting in MongoDB 3.4) is computed as 50 percent of total RAM minus 1GB. For example, on a system with a total of 3 GB of RAM, the WiredTiger cache will use `0.5 * (3GB - 1GB) = 1GB`. If this formula yields a number less than 256MB, the MongoDB will use the default 256 MB for WiredTiger. In general, more RAM will benefit filesystem cache as more MongoDB data can be kept in RAM for faster access.

3.1.3 Swap strategy

Like many other database systems, MongoDB works better with the swapping disabled. According to the MongoDB documentation, swapping may be only permitted under high memory load; however, it is better to configure the kernel to disable swapping entirely.

3.1.4 Storage

SATA SSD drives are the recommended storage option for MongoDB because of their better performance in comparison to HDDs. Also, keep in mind that using remote file systems such as NFS may significantly degrade MongoDB performance.

If your company uses RAID, MongoDB docs recommend RAID-10 for optimal performance. RAID-5 and RAID-6 do not typically provide sufficient performance to support a MongoDB deployment.

Also, it's recommended to separate data, journals, and logs onto different storage devices, based on your app's access and write patterns. In practice, you can mount each component mentioned above in a separate filesystem and use symbolic links to map each component's path to the device storing this component.

3.2 Linux configuration

There are several important OS-level settings to consider when running MongoDB on Linux distributions.

3.2.1 Turn off Transparent Huge Pages (THP)

Database workloads often perform poorly with THP enabled, because they tend to have sparse rather than contiguous memory access patterns. When running MongoDB on Linux, THP should be disabled for better performance.

3.2.2 Number of open file descriptors

At peak traffic times, MongoDB reads and writes to multiple files opened simultaneously. To enhance performance, it's recommended to allow at least 64000 open file descriptors for the MongoDB process.

3.2.3 Number of connections

The maximum number of incoming connections supported by MongoDB is configured with the `maxIncomingConnections` setting. On Unix-based systems, system-wide limits can be modified using the `ulimit` command or by editing your system's `/etc/sysctl` file. Increasing the maximum number of incoming connections will only work if the node has enough resources.

3.2.4 Other system settings

Other recommended Linux system settings include the following:

- Set maximum threads per process (`kernel.threads-max`) to 64000.
- Maximum number of memory map areas per process (`vm.max _ map _ count`) should be at least `128000`.
- Allocate sufficient file handles by setting `fs.file-max` to 98000.
- TCP keepalive value of 300 often provides better performance for replica sets and sharded clusters.
- If possible, use XFS instead of EXT4, as it generally performs better with MongoDB.
- On Windows, use the NTFS file system. Do not use any FAT file system.

3.3 Replica set design considerations

3.3.1 Data-bearing nodes

At a minimum, a MongoDB replica set should consist of three data-bearing members: one primary node and two secondary nodes. If you can't allow having two secondary replicas, you may consider including an arbiter node instead of one data-bearing node. In any case, you should ensure that the replica set has an odd number of voting members (3,5,7, etc.) and up to seven voting members. In total, a replica set can have up to 50 members.

3.3.2 Enable journaling

MongoDB uses write-ahead logging to journal files to provide durability in the event of failure. Journaling ensures that MongoDB can quickly recover write operations that weren't written to data files. Journaling is enabled by default, and you shouldn't change this setting.

3.3.3 Configure write concern

Write concern level determines how quickly the write operation returns. With a stronger write concern, clients should wait before the specified number of replicas copy the write. The choice of write concern depends on the importance of your data. For example, you can select a lower write concern (e.g., "w:1") for data that is not critical. For business-critical and mission-critical data, the "`majority`" write concern is recommended.

3.3.4 Load balance on read-heavy deployments

By default, all read and write operations are managed by the primary replica. You can improve read throughput by distributing reads to secondary members using a "secondary" read preference mode. However, in this mode, MongoDB may return stale data if secondaries did not catch up with the primary. Thus, when using "secondary" read preference mode, you should ensure that your application can tolerate stale data.

3.3.5 Distribute replicas across datacenters

It is important to prepare for scenarios of the entire datacenter downtime. If this happens, it's good to have at least one member of a replica set in a standby datacenter. If possible, consider using an odd number of datacenters and choose a distribution of members that maximizes the likelihood of retaining at least some copies of your data if the "active" datacenter goes down.

Note: MongoDB team recommends distributing the replica set across three datacenters for higher fault tolerance.

3.3.6 Prepare capacity ahead of demand

If your application traffic has an upward trend, your replica set may experience storage and memory pressure. This can compromise the read and write performance of your databases. Therefore, set aside a spare capacity (storage, RAM) to be able to scale your replica set immediately when this happens. Alternatively, use autoscaling tools provided by Kubernetes to automatically add new capacity based on some usage metrics.

You should also ensure that non-hidden replica set members are identically provisioned in terms of their CPU, disk, network, RAM, etc.

3.3.7 Oplog size

If your database records a large number of operations in the oplog, the space left to record new operations may become very low. This may be the case if you have multiple updates to multiple documents at a time or have a significant number of in-place updates that do not increase the size of the documents but make the database record many operations. If you have these database patterns, consider increasing the oplog size.

3.3.8 Control replication lag

Replication lag (RL) is a delay before data is copied from the primary to the secondary replica. Large lags can result in read requests returning stale data if the "read" preference is set to "secondary." To keep the RL in check, you can enable the "flow control" feature and configure the `flowControlTargetLagSeconds` time window. When the lag grows closer to this setting, each write to the primary must obtain tickets before taking locks to apply writes. You can limit the number of such tickets issued per second to limit the lag. However, the lag itself may indicate some other performance issues that should be also considered.

3.4 Sharded cluster configuration

3.4.1 Select a proper shard key

A shard key is responsible for splitting shards into chunks and distributing data proportionally among them. The poor choice of shard key can adversely affect the performance of your MongoDB cluster. When selecting the shard key for your collection you should consider shard key cardinality, frequency, and rate of change.

3.4.2 Shard key cardinality

Cardinality of a shard key determines the maximum number of chunks the balancer can create. Low cardinality shard keys result in fewer chunks and, as a result, less opportunity for horizontal scaling. For example, a "state" field (United States) would be an example of a shard key with low cardinality. Because it can take only 50 possible values but there may be over 10 million users in your collection, documents with the same state will reside on the same shard. This can lead to uneven distribution of data and problems such as unsplittable chunks and unbalanced shards. In turn, this can cause migration delays and I/O overheads.

A better choice would be the `post _ code` or `phone _ number` fields, which have a higher cardinality. This makes it easy to split chunks because they won't be made up of documents mapping to the same shard key.

In sum, a shard key with high cardinality improves the distribution of data across the sharded cluster and better facilitates horizontal scaling, but it should be considered along with other parameters such as frequency and the rate of change.

3.4.3 Shard key frequency

The shard key frequency determines how often a given value occurs in the data. For example, if the majority of documents have a "California" shard key value, then the chunk storing these documents is a bottleneck of the cluster. As these chunks become larger, they become indivisible. This reduces the effectiveness of horizontal scaling within your cluster. Therefore, consider a shard key with low frequency (evenly distributed values).

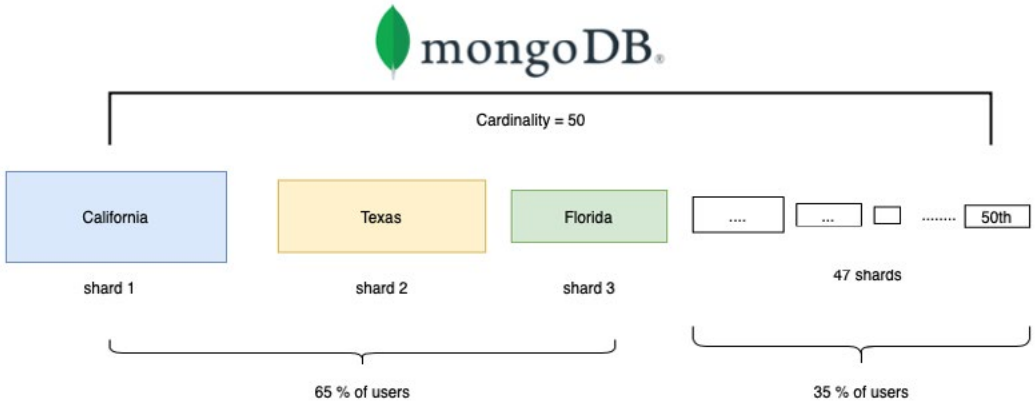

Image 2: Low cardinality and high frequency shard keys may result in uneven distribution of users in shards

3.4.4 Shard key rate of change

Shard keys that increase or decrease monotonically (e.g., document id) may also be a bottleneck. In this case, documents will be routed to the chunk with the `maxKey` as the upper bound. The shard that contains this chunk then becomes a bottleneck to write operations.

If you want to use a document id as a sharding key, consider using a shard key hashing option that will turn the shard key into hashes and distribute them evenly across shards.

3.4.5 Chunks size

Small chunks result in more even data distribution at the cost of more frequent migrations. In contrast, large chunks lead to fewer migrations, but these efficiencies have a side effect of potentially uneven distribution of data. For many deployments, it's reasonable to avoid frequent migrations that slow down the queries at the expense of slightly less evenly distributed data sets.

4. BEST PRACTICES FOR RUNNING MONGODB ON KUBERNETES

Container orchestration in Kubernetes provides a way to manage MongoDB deployments at scale while taking advantage of containerization benefits—such as the ease of deployment and maintenance and the ability to save on compute resources and storage, as more containers can fit on a single server. Container orchestration can provide MongoDB with all the advantages of automation, operation, scaling, and monitoring.

4.1 MongoDB Kubernetes deployment options

Kubernetes is a mature platform with developed tooling for all major enterprise applications, including MongoDB. There are several ways to deploy MongoDB on Kubernetes—manual deployment, as a Helm Chart, and using a MongoDB Kubernetes operator. Each of these options has certain advantages and limitations, which are described below.

4.1.1 Manual deployment

Manual deployment is the hardest option because it requires deep expertise both in MongoDB architecture and Kubernetes architecture. However, as Kubernetes is adding more support for stateful applications such as MongoDB, manual deployment of MongoDB has become less painful. If you are thinking about this option, there are several things to know about.

Normally, manual MongoDB deployment on Kubernetes requires two Kubernetes resources—StatefulSets and Headless Services.

A StatefulSet is a Kubernetes primitive that provides sticky and persistent identity for MongoDB pods. This is needed for stateful applications like MongoDB that should maintain their state after the rescheduling (have the same persistent volumes, configuration, network identifiers, DNS name, etc.).

Stateful sets are quite flexible. You can configure the number of replicas, networking (DNS, IPs), network policies, container security, and other Kubernetes parameters. You can also link MongoDB-specific configuration—such as commands to start mongod with, replica set and sharding settings, etc. This can be specified as container commands or provided via a ConfigMap. For more information on deploying MongoDB as a stateful set, you can consult this tutorial from the Kubernetes blog.

Next, you'll need a Kubernetes service to discover individual MongoDB nodes and components. Default stateless Kubernetes services abstract the identity of underlying pods, so they are not suitable for MongoDB's node communication. However, you can use a headless service to achieve the desired behavior. The headless service is a passthrough directly to all of the IPs behind it that can easily discover stable IPs of healthy running MongoDB nodes. When combined with StatefulSets, a headless service can provide unique DNS addresses to MongoDB pods, which are needed for clients to connect to shards and config servers if MongoDB is deployed as a sharded cluster.

4.1.2 MongoDB deployment via Helm charts

Helm is a package manager of pre-configured Kubernetes resources called charts. A Helm chart contains the configuration of all Kubernetes resources (StatefulSets, Services, service accounts, etc.) needed to deploy an application on Kubernetes. Charts embody best practices for configuring dependencies and Kubernetes resources for containerized applications. A good option is a MongoDB Helm Chart available from the official Helm chart repository.

Deploying MongoDB as a Helm chart is easier than the manual deployment, but it does not scale well for more complicated use cases—such as backups, restores, monitoring, etc.

4.1.3 Using MongoDB Operators

A Kubernetes Operator is a runtime that manages the full lifecycle of an application on Kubernetes, including deployment, rolling upgrades, authentication, data backup, etc. It may be used to create custom resources to declaratively define MongoDB replica sets and controllers to manage the MongoDB StatefulSet on Kubernetes.

To give you an idea of how MongoDB Operators can benefit your deployment, let's discuss the node decommissioning scenario. A MongoDB Operator would create and register custom MongoDB resources with the Kubernetes API and run a set of controllers managing these resources. For example, a controller could be watching the MongoDB custom resource for user-triggered changes in the node count and, if the user decided to remove one node, run the graceful node removal operation. This operation might involve gracefully stopping and draining the node and triggering the data re-balancing and re-syncing operation. When the controller confirms that this operation has succeeded, it changes the stateful set definition to allow Kubernetes controllers to gracefully remove the node.

One of the most popular MongoDB Kubernetes Operators is the MongoDB Enterprise Kubernetes Operator maintained by the MongoDB Support team. To run successfully on Kubernetes, this Operator requires MongoDB Ops Manager version >=4.0 or MongoDB Cloud Manager.

Using this Operator, you can deploy Ops Manager resources to your Kubernetes cluster and deploy MongoDB as a standalone instance, replica set, or sharded cluster. The Operator is closely integrated with the Ops Manager resources and services such as monitoring, alerting, back up, etc. However, some of its features may only be available to MongoDB subscribers.

4.2 Managing MongoDB deployment on Kubernetes with Portworx

Kubernetes provides many useful features to automate MongoDB management and maintenance. However, some additional tools and features are needed for efficient production deployment of MongoDB in Kubernetes. For example, we need an efficient way to manage MongoDB storage resources, backup and restore MongoDB data, and configure storage-layer security, disaster recovery, and monitoring. To provide these features, the solution that manages storage for MongoDB on Kubernetes should be closely integrated with Kubernetes API and kube-scheduler and understand Kubernetes abstractions and resources as well as provide interfaces to manage MongoDB using MongoDB domain-specific knowledge. Such a comprehensive solution for MongoDB in Kubernetes may be provided by the Portworx Enterprise storage platform.

The Portworx Enterprise storage platform is a storage and data management solution for containers and Kubernetes. Portworx tools provide many features for storage aggregation and on-demand provisioning, security, monitoring, backup, and disaster recovery, all via a centralized user interface and easy-to-use API.

At the basic level, Portworx works as a storage cluster integrated into a Kubernetes cluster and interacts with Kubernetes components and controllers. Portworx storage drivers aggregate the underlying storage resources (SANs, SSD, NVMe, etc.) available in the cluster and merge them into a unified storage pool. Storage resources are differentiated based on their I/O profile, Class of Service (CoS), location, etc. After aggregation, virtualized storage can be directly provided to containers using Kubernetes native storage abstractions—such as persistent volumes and storage classes. Various operations with volumes—such as taking snapshots, creating replicas, scheduling backups, and configuring volume access and ownership—can be performed via Portworx.

The Portworx storage layer is elastic and easily scalable. Instead of managing individual storage devices for each node in your cluster, you can leverage Portworx storage virtualization to create a unified storage pool.

On top of data aggregation, Portworx offers many data management services, including backup and disaster recovery features. These features are implemented in several Portworx tools—including PX-DR, PX-Backup, PX-Migrate, PX-Secure, and PX-Autopilot for Capacity Management among others. With the Portworx BCDR package, you can develop different strategies for your apps and data depending on their RTO and RPO requirements, criticality, and impact. With Portworx, you can ensure local HA and cross-cluster and multi-datacenter disaster recovery as well as effective container-granular and app-consistent backups for your MongoDB deployment.

4.3 Enabling MongoDB local HA for Kubernetes

MongoDB ensures HA of the data using replica sets, sharding, and efficient primary node election algorithms. MongoDB replica sets, however, ensure HA only at the database layer. The underlying storage layer is not replicated by default.

Portworx Enterprise can further strengthen data availability at the storage layer using its built-in persistent volume replication. Persistent volumes are Kubernetes resources that abstract the underlying storage available to the cluster and provide it on demand in specified amounts to applications running in pods. Portworx makes it easy to allocate storage to persistent volumes. When creating a persistent volume using the Portworx storage layer, users can specify a replication factor that determines how many copies of the volume should be made and how these copies should be distributed across the cluster. Also, Portworx provides additional volume features—such as classes of storage (CoS), I/O profile, multi-access options, shared volume settings, etc.

By default, the Portworx engine distributes replicas evenly across the fault domains (nodes, availability zones, etc.) so that each of them has a copy of MongoDB data. Having a data copy locally on each node ensures fast failover in the disaster scenario. All the replicas are kept up-to-date using synchronous replication. Portworx replicates each write to any of the volumes across all replicas in the cluster. Consequently, if the node or application fails, the new application instance spun up on a new node can have instant access to a copy of its data.

For example, let's first consider a MongoDB crash scenario without volume replication. When Kubernetes detects the failure of the node that runs a single MongoDB instance, it automatically reschedules the MongoDB instance to a new node. The Kubernetes scheduler's main task is to ensure that the number of MongoDB instances matches the number of replicas specified in the StatefulSet, but it does not care about whether the launched instance has a local copy of MongoDB data on a new node. This is why a copy of data should be manually provisioned to a new node and corresponding actions should be taken by the database administrator to resync the MongoDB replica set. This takes time and effort on the part of the infrastructure and database management team.

Now, let's look at what happens if the Portworx volume replication is enabled. In this case, the MongoDB failover is managed not just by the Kubernetes scheduler but also by Portworx STORK (Storage Orchestration Runtime for Kubernetes). STORK extends the scheduler behavior with the storage aware orchestration. If a Kubernetes scheduler is configured to use the STORK, it makes two API calls to Portworx before placing MongoDB on a new node: Filter and Prioritize. Using the "filter" request, STORK filters out nodes that do not participate in the Portworx cluster (that is, the Portworx storage layer is not installed on them). This reduces the number of failed scheduling attempts that may occur due to Portworx's inability to manage nodes that do not participate in the Portworx cluster.

Secondly, using a "prioritize" request, STORK selects nodes that have a replica of the MongoDB data. Portworx checks which persistent volume claims (PVCs) have been used by the MongoDB pod and queries the storage driver for the location of the pod's data. Using this information, STORK ranks various nodes based on the storage class (SSD, HDD) it provides to match the requirements specified in the MongoDB persistent volume and selects the best candidate. Finally, it communicates its choice to the `kube-scheduler`, and the latter places the MongoDB instance on that node. This mechanism ensures that the MongoDB has instant access to its data whenever the downtime occurs.

In addition to data-app hyperconvergence, STORK provides such features as failure-domain awareness, storage health monitoring, and snapshot-lifecycle features for stateful apps on Kubernetes.

4.4 MongoDB backup options for Kubernetes

Making regular backups of MongoDB data is the basic way to prevent data loss and ensure fast recovery after data corruption and cluster outages. At this moment, MongoDB data can be backed up using OS-level tools and built-in MongoDB backup tools:

- OS-level backups with `cp` or `rsync`
- Filesystem snapshots
- Mongodump and mongorestore

4.4.1 Backup with cp or rsync

This is the simplest and the least effective backup option. In essence, it relies on OS-level file system tools such as `cp` or `resync`. When using this option, all writes to the mongod should be stopped before copying files. Overall, when copying MongoDB data files manually, you should be aware of the following shortcomings. First of all, the backups done using `cp` and similar tools do not support point in time recovery for replica sets and are hard to manage for larger sharded clusters. Also, these backups are significantly larger because they include indexes and duplicate the underlying storage padding.

4.4.2 Filesystem snapshots

File system snapshots are a feature provided by the OS volume manager. If a volume hosting MongoDB supports OS-level point-in-time snapshots, this feature can be used as a baseline for data backup. There are, however, several prerequisites for this option:

- Journaling should be enabled. Otherwise, no data consistency guarantees are provided.
- The balancer of a sharded cluster should be disabled. Also, snapshots should be taken from every shard as well as a config server at approximately the same point in time.
- To create a coherent backup, the database must be locked and all writes to the database must be suspended during the backup process.

Also, keep in mind that filesystem snapshots create an image of an entire disk. To save on backup space, consider isolating your MongoDB data files, journal (if applicable), and configuration on one logical disk that doesn't contain any other data.

4.4.3 Mongodump and mongorestore

Mongodump and mongorestore are backup and restore tools provided by MongoDB installation.

Mongodump reads MongoDB data files and creates high-fidelity BSON files, which can be used by Mongorestore to populate the database during the restore operation. This tool is much more efficient than file snapshots and manual `cp` snapshots. In particular, snapshots created via MongoDB are space efficient because they include only the contents of the MongoDB data files. However, both mongodump and mongorestore are not ideal for larger systems and can adversely affect mongod performance. Also, mongorestore or mongod must rebuild the indexes after restoration.

Both tools cannot be used as part of a backup strategy for 4.2+ sharded clusters that have transactions in progress because their backups do not maintain the atomicity guarantees of transactions across shards.

For sharded clusters, it is recommended to use coordinated backup and restore tools provided by Ops manager; however, this suite is available only to MongoDB Enterprise Advanced.

4.5 Backing up MongoDB data with PX-Backup

Built-in MongoDB backup tools work well if you run MongoDB in a bare metal cluster. However, these tools have a limited scope if you run MongoDB in containers and Kubernetes simply because they don't understand Kubernetes abstractions, container runtime settings, and (generally) how a Kubernetes cluster operates. A MongoDB backup tool for Kubernetes should meet the following requirements:

- **Be container-granular.** A MongoDB backup solution for Kubernetes should know how to back up a database at the container and pod level because these are abstractions that wrap MongoDB instances in Kubernetes. With the container granularity, we can avoid costly and error-prone ETL procedures that would be required if we backed up all VMs in their entirety. By only backing up MongoDB containers, we also minimize storage costs and keep recovery time objectives (RTO) low.
- **Be Kubernetes-aware.** MongoDB deployment on Kubernetes consists of many abstractions and Kubernetes resources—such as persistent volumes, secrets, persistent volume claims, configuration objects, stateful set definitions, customer CRDs, etc. All these objects are important for MongoDB configuration, management of storage, networking, and interactions between individual containers. Thus, a consistent backup of MongoDB on Kubernetes should include not only data but these objects as well. A backup tool for Kubernetes should understand Kubernetes API to copy these objects and restore them in a single coherent deployment.
- **Be application-consistent.** A MongoDB backup tool for Kubernetes should incorporate database-specific knowledge used by MongoDB native tools like `mongodump` and `mongorestore` to create consistent backups of data. This knowledge may include operations such as flush, data repair, resync, and reindexing. The tool should know how to communicate these operations to MongoDB containers using Kubernetes API.

PX-Backup is a component of Portworx Enterprise that satisfies the mentioned requirements. It provides container-granular, application-consistent, and Kubernetes-aware snapshots of MongoDB on Kubernetes.

PX-Backup ships with the following features:

- Full application-consistent backups for Kubernetes resources
- Container data lifecycle management from the centralized and user-friendly interface
- Cataloging of relevant backup metadata
- Providing visibility into data access
- PX-Backup supports backups for applications storing their data on both Portworx Enterprise as well as directly on cloud block storage from Azure, AWS, and Google Cloud managed via the Kubernetes CSI plugin.

4.5.1 Application-consistent backups

PX-Backup ensures that MongoDB snapshots contain the most recent and application-consistent data. When creating a MongoDB snapshot with PX-Backup, you can specify pre- and post-backup rules to be executed before and after the backup is made. For example, you can tell MongoDB to flush all pending writes to disk before taking a snapshot and lock the database server to prevent additional writes. Correspondingly, you can specify database-specific actions such as compaction or data repair to ensure that the backup image is restored consistently.

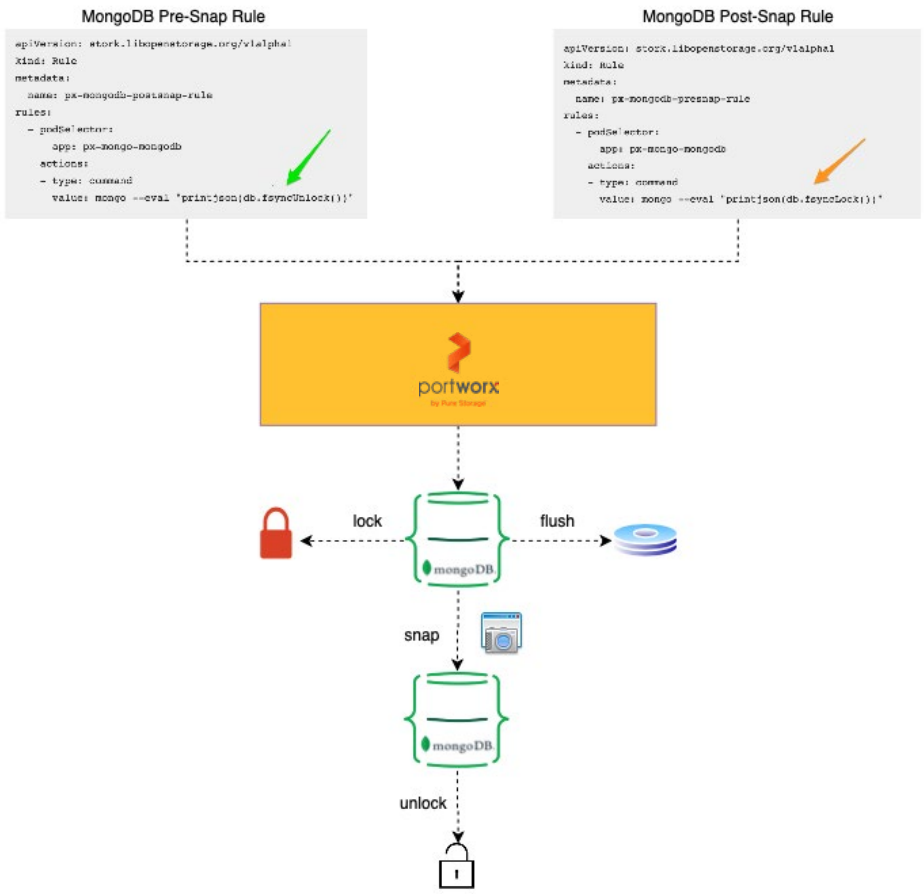

Image 3: MongoDB Application-Consistent Backups with Portworx

PX-Backup provides UI and backup management features. Each backup has a metadata, including the timestamp, original data location, backup destination folder, cluster, and database from which the backup was taken. This information allows for a better audit of backups long after the snapshot was taken. Also, with PX-Backup you can configure a regular backup schedule (e.g., daily) and backup retention policy that allows cleaning old backups to save on storage.

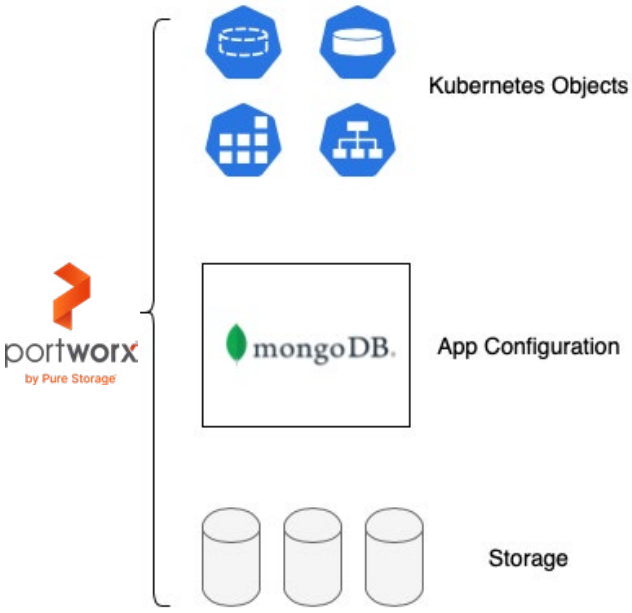

Image 4: Kubernetes-aware backups with Portworx

4.5.2 Kubernetes-aware backups

PX-Backup understands Kubernetes APIs and Kubernetes native and custom resources. When making a backup, it also copies all configuration objects—such as CRD, secrets, service accounts, etc.—along with the data. This backup can be easily restored on Kubernetes without the need to re-create multiple Kubernetes objects and configurations. This leads to a much faster failover of your apps on Kubernetes that is unattainable with most backup solutions. Your RTO objectives can be met with this approach with no pain.

4.5.3 Kubernetes namespace-aware backups

Kubernetes namespaces allow cluster administrators to distribute Kubernetes resources among different teams. With namespaces, they can set container resource request and limit constraints and a per-namespace quota for different teams using a single Kubernetes cluster. For example, a software engineering team may run several MongoDB instances in a "dev" namespace, and the "prod" namespace can host the production database. Additionally, these namespaces could host a stack of applications interacting with MongoDB—such as business analytic engines, various jobs, and application servers. Given this configuration, it would be convenient if a given team could back up the entire namespace instead of taking snapshots of individual applications by hand—which would be costly, error-prone, and inefficient. Instead, the entire namespace could be backed up with all of its resources and configuration.

PX-Backup provides such an option with the Kubernetes namespace-aware backups. Such a backup contains all application data along with configuration and metadata that facilitates efficient management of backup long after it was taken. It also enables audit and regulatory compliance. PX-backup can easily restore the namespace backup to the same namespace—or even a different namespace—in a single click. It will ensure that all Kubernetes resources are properly launched and that the database state is consistent before and after the snapshot was taken.

4.5.4 Multi-cluster support

Many enterprises have different MongoDB deployments in several Kubernetes clusters. For example, this is the case when MongoDB is used as a database for several applications and several Kubernetes clusters run these deployments in different datacenters. It would be convenient to manage MongoDB backups from different Kubernetes clusters from the single console.

PX-Backup supports multi-cloud backup management from one central location. It maintains the information about the cluster and the cloud the backup was taken from so you can easily restore the MongoDB cluster to this particular cluster or another cluster.

Having a centralized interface for managing backups for these clusters provides visibility into the source environments and allows management of the full lifecycle of your backups.

PX-Backup works with all major cloud drives—GCP, AWS, Azure—which makes it easy to create backups using the cloud storage provided by these platforms and import backups from them.

Among other features supported by PX-Backup, there are the following:

- Point-in-time restore of data
- Filtering backups on cluster(s), namespace(s), and labels
- Backup audit

4.6 MongoDB disaster recovery with Portworx

Local HA with volume replication ensures fast intra-cluster failover. However, we should also address the scenarios when the entire datacenter hosting the Kubernetes cluster running MongoDB goes down. To protect data against such scenarios, a sound cross-cluster data protection strategy for MongoDB should be enabled.

Cross-datacenter disaster recovery can be implemented using the PX-DR component of the Portworx Enterprise. The PX-DR is built on the same container granularity, Kubernetes awareness, and application consistency principles as PX-Backup extended to the scale of multi-cloud.

The PX-DR can operate in two modes—synchronous cross-datacenter DR and asynchronous cross-datacenter DR. The choice of a mode depends on your MongoDB cluster deployment architecture.

In both modes, you have an active-standby and/or active-active cluster pair. An active cluster migrates workloads synchronously or asynchronously to the standby cluster so that all resources and data are eventually synchronized based on your desired RPO. The standby cluster is not just a passive storage location. In essence, it's a perfect "mirror" of the active cluster with all controllers, processes, and configuration prepared to supersede the active cluster once its failure is detected. To enable this, PX-DR migrates not just data but full applications with their Kubernetes manifest, configuration, current state, etc.

Migrations can be done in a granular way. Users can migrate specific pods, groups of pods, or entire namespaces similarly to what they can do with the PX-Backup.

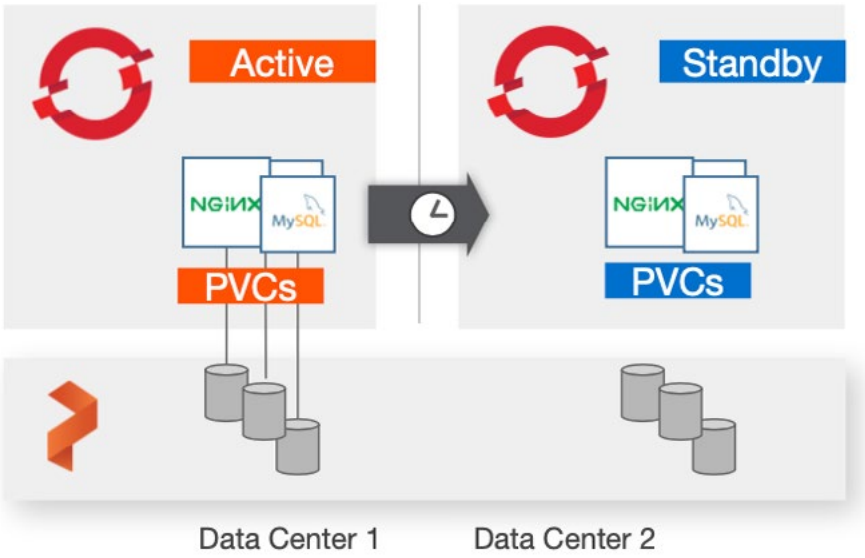

Image 5: PX disaster recovery with Active and Standby Clusters

4.6.1 Synchronous cross-datacenter DR

In the synchronous cross-datacenter DR mode, Portworx is installed as a single stretch cluster covering multiple Kubernetes clusters located within a single metropolitan area network (MAN). Datacenters within this network should be located within a 50-mile distance and be part of the same cloud region (possible different availability zones). For synchronous replication to work efficiently, the network latency between nodes is expected to be less than 10 milliseconds.

Low network latency allows an active cluster to continuously back up MongoDB data to standby clusters and enables the fast failover. Also, the mode takes advantage of the Portworx topology awareness and fault domain detection, so the backup replicas can be evenly distributed across the nodes in a standby cluster. If properly configured, the synchronous cross-datacenter DR mode yields zero RPO and RTO of less than a minute, which perfectly matches the requirements of business-critical and mission-critical applications working with MongoDB.

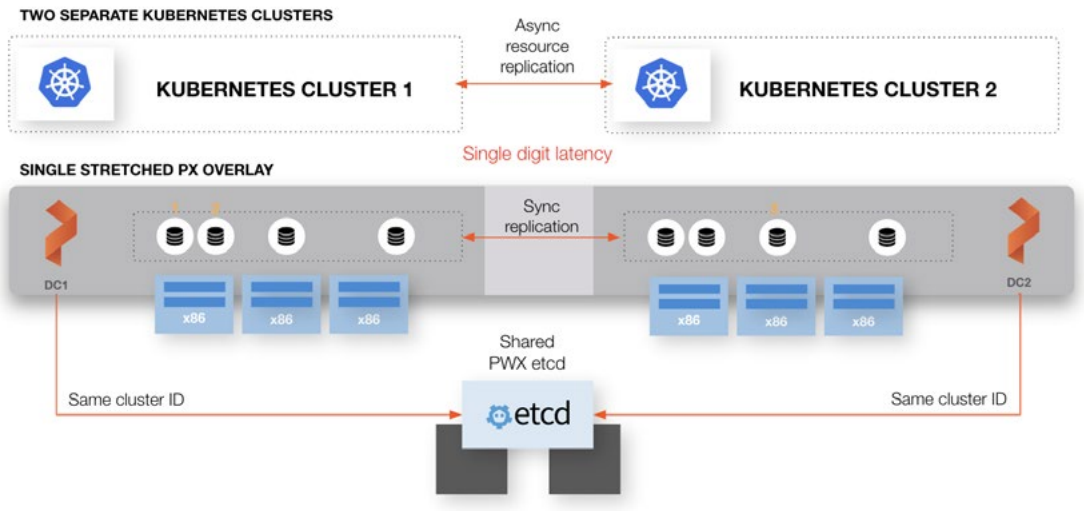

Image 6: Synchronous DR with Portworx

4.6.2 Asynchronous cross-datacenter disaster recovery with Portworx

In the asynchronous cross-datacenter DR mode, Kubernetes clusters are located in different cloud regions (wide area networks) and have higher latency between them (more than 10 milliseconds). For this setup, Portworx needs to be installed as a separate cluster in each Kubernetes cluster deployed in these datacenters. As a result of higher latency, the continuous backups are done asynchronously according to the migration schedule specified by the user. Active and standby clusters are eventually synchronized, but due to the higher latency, the failover may take more time. Normally, this mode yields an RPO of 15 minutes and an RTO of less than 60 seconds, which meets the requirements of many applications and data systems.

Similarly to the synchronous mode, in the asynchronous mode the standby Kubernetes cluster has running controllers, configuration, and PVCs that map to local volumes, so it's ready to become active any time.

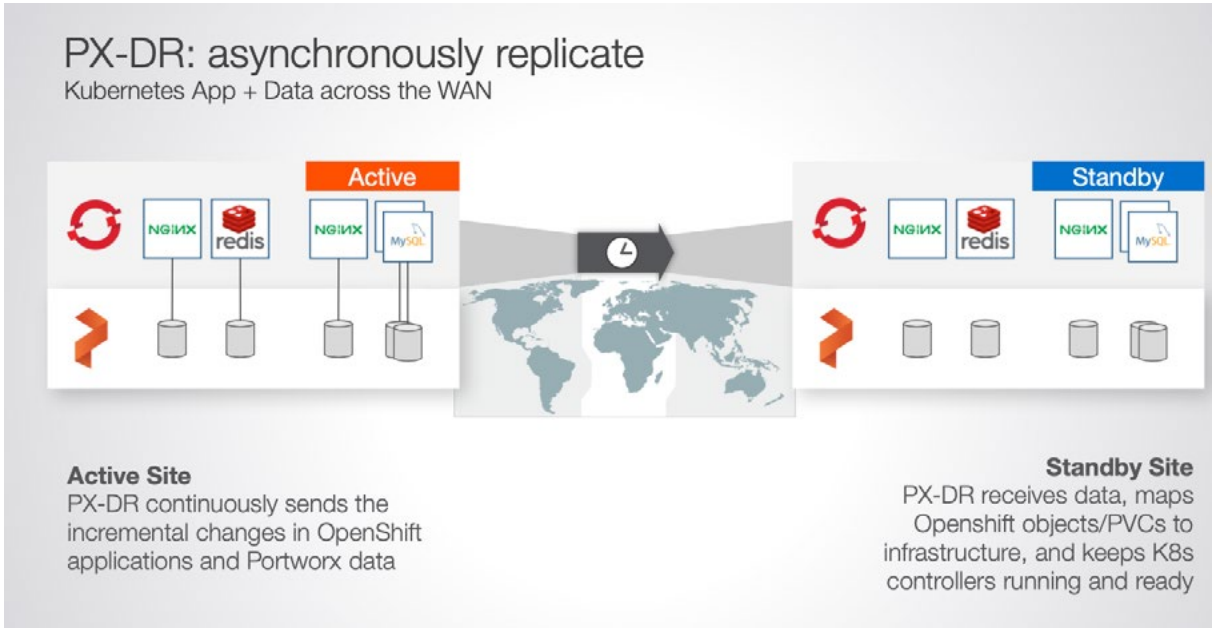

Image 7: Asynchronous DR with Portworx

Also, users can schedule synchronization between the active and the standby clusters using migration schedule policies. They allow setting an interval for migration and specify objects and volumes covered by the schedule. You can also decide whether an application should start once it's migrated using the `startApplications` flag.

As with snapshots, users can specify `preexec` and `postexec` rules for migrated volumes. This may be useful when you migrate databases that require all pending write operations to be flushed to disk before making a migration. As part of the Asynchronous DR feature, Portworx can migrate PV, PVCs, Deployments, StatefulSets, ConfigMaps, Services, Secrets, and other important Kubernetes resources.

4.7 MongoDB security

MongoDB provides several basic security mechanisms to protect your data against corruption, theft, and loss. Most of them are disabled by default for faster testing; however, they should be configured for production deployments.

4.7.1 Authentication

Authentication allows verifying the identity of a user before granting access to the database. MongoDB provides SCRAM or x.509 mechanisms for authentication. You can also integrate MongoDB authentication with your existing Kerberos/LDAP authentication systems.

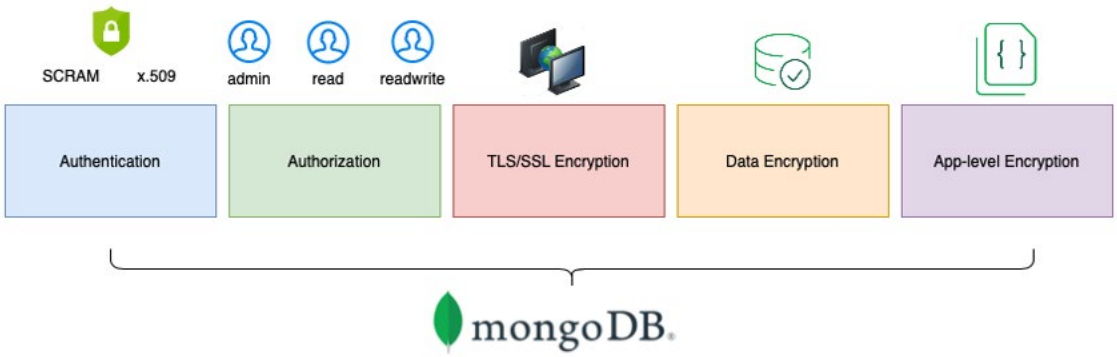

Image 8: MongoDB Security

4.7.2 Authorization

MongoDB implements a traditional Role-Based Access Control (RBAC) system to manage user access privileges. The MongoDB administrator can grant one or more roles to users which determine resources and operations he/she has access to. The RBAC is disabled by default.

4.7.3 TLS/SSL encryption

MongoDB provides TLS/SSL (Transport Layer Security/Security Sockets Layer) mechanism for the encryption of MongoDB's network traffic. To ensure stronger protection, MongoDB allows only TLS/SSL ciphers with a minimum of 128-bit key length for all connections. Also, MongoDB supports TLS/SSL authentication using certificates, both for client and internal authentication of members of replica sets and sharded clusters.

4.7.4 Encryption at rest

"Encryption at rest" is a mechanism for encrypting MongoDB data files stored on disk. It is supported in MongoDB Enterprise 3.2 for the WiredTiger storage engine. When "encryption at rest" is enabled, data is presented in the unencrypted form only when in memory. The default encryption mode is AES256-CBC (or 256-bit Advanced Encryption Standard in Cipher Block Chaining mode) via OpenSSL). Along with protecting data from unauthorized access, encryption at rest can help ensure compliance with such privacy and security standards as PCI-DSS, FERPA, and HIPAA.

4.7.5 Application-layer encryption

Application-layer encryption allows encrypting fields in documents prior to their transmission over the network. Only users and applications with the correct encryption keys can decrypt the protected data in documents. The application-layer encryption provides additional guarantees for the safety of sensitive data in case network encryption layers or data encryption layers fail or are compromised.

Also, MongoDB supports field-level redaction, which restricts the content of the documents based on the information stored in them. For example, the document may include fields with tags specifying certain access groups (e.g., "EXCLUSIVE"). Fields with these tags will then be accessible only by the users who belong to these groups.

4.8 Hardening MongoDB security with Portworx

Running MongoDB in Kubernetes introduces new attack vectors and security challenges that need to be addressed. In particular, when running MongoDB in Kubernetes, one should consider storage media and storage access security along with Kubernetes cluster security.

The storage layer security can be configured with PX-Security, a component of the Portworx platform that provides important security features, such as:

- Storage encryption
- Role-based access control (RBAC) for volume access and management
- Volume ownership configuration
- Support for integration with AD and LDAP

4.8.1 Authentication and RBAC with Kubernetes and Portworx

The Kubernetes security layer enables authentication and authorization at the cluster and namespace level. Kubernetes provides many useful authentication methods, including client certificates, bearer tokens, an authenticating proxy, HTTP basic auth, SSL, and more. Also, Kubernetes has a built-in RBAC model that allows assigning different roles to pods and users in different namespaces.

However, Kubernetes does not provide a specialized RBAC mechanism at the storage and persistent volume layer. As a result, any user who has access to the Kubernetes cluster can affect the volumes allocated to MongoDB pods. This may introduce unnecessary risks and security issues.

PX-Security closes this gap by providing additional authentication and authorization mechanisms that extend Kubernetes RBAC at the storage layer. If the Kubernetes storage resources are managed by Portworx and Portworx RBAC is enabled, users' interactions with the storage engine are subject to Portworx authentication and authorization mechanisms.

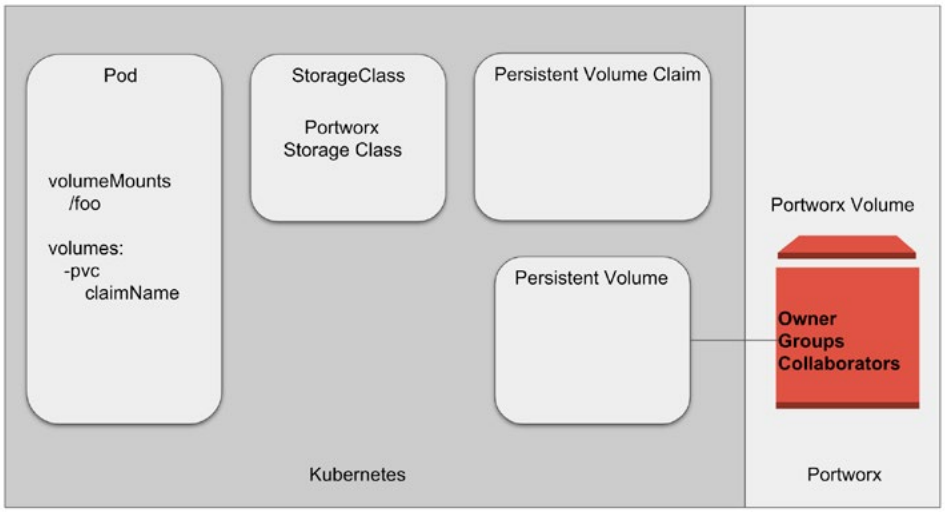

Image 9: Relationship between Portworx and Kubernetes Security Layer

First, in order to access Portworx services, the user should be authenticated to the cluster. For authentication, Portworx uses OIDC and self-generated JWT tokens, which makes it easy to use Portworx with enterprise-grade authentication systems such as SAML 2.0, LDAP, or Active Directory.

Secondly, in order to perform corresponding operations on storage—such as taking snapshots or creating/deleting volumes—users should provide a valid JWT token that contains the roles and groups of the user trying to create a volume (see the image below). For example, the user will be able to take a snapshot of the volume only if he/she has the corresponding role assigned by the Portworx cluster administrator. Portworx provides the ability to create custom roles for interacting with persistent volumes not available in the Kubernetes RBAC model by default.

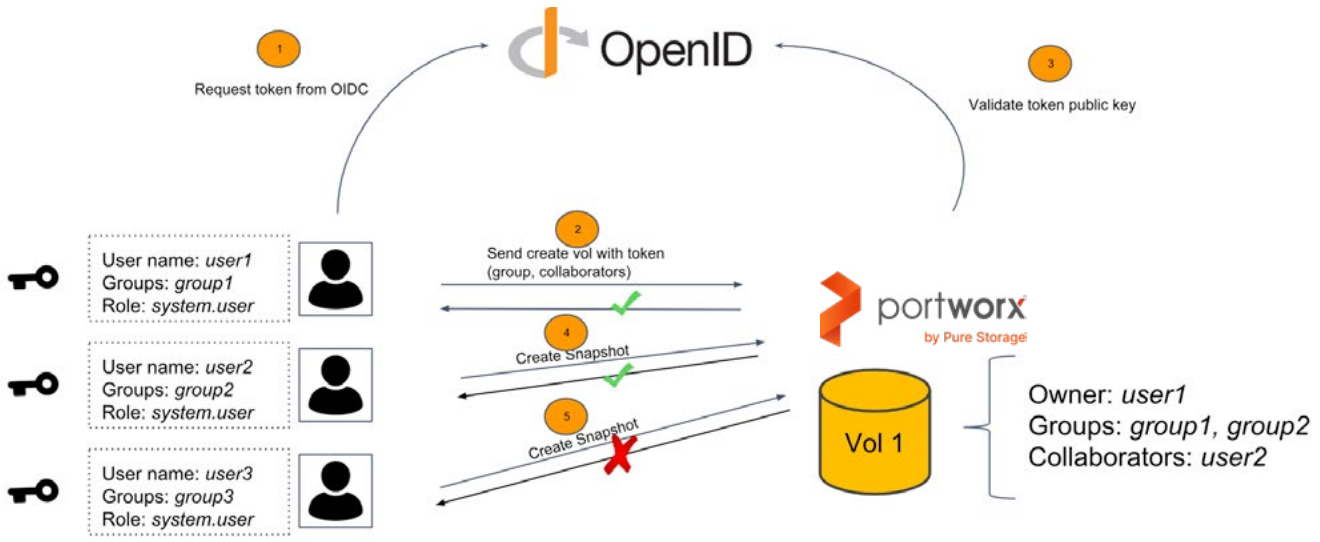

Image 10: Portworx Security Model

In conjunction with RBAC, PX-Security implements an ownership model for PX-managed volumes. The model includes the volume owner and the list of groups and collaborators with different volume access privileges. A volume owner can provide write, read, and admin access to the volume.

4.8.2 Encryption of Kubernetes resources

Securing data is essential for production applications. PX-Secure can help with container- or cluster-granular encryption.

The Portworx implementation of volume encryption is based on dm-crypt, a disk encryption subsystem of the Linux kernel that can create, access, and manage encrypted devices. Volumes provisioned by Portworx can be encrypted with cluster-wide secrets shared by other volumes or per-volume secrets unique to each volume. Portworx provides an opportunity to encrypt data at rest as well as in transit.

In summary, adding Portworx to your MongoDB deployment enables a multi-modal security that covers both database application-level security, cluster-level security, and storage-level security, dramatically decreasing the potential vectors of attacks against your MongoDB cluster on Kubernetes.

4.9 MongoDB monitoring

Monitoring is a critical component of any database administration. It can allow you to continuously evaluate the state of the MongoDB deployment and maintain it without problems. Monitoring can also help to diagnose problems before they lead to serious failures.

MongoDB provides several useful built-in tools and commands every administrator should know in order to diagnose problems quickly. Below are some of the most important of them:

- `mongostat`. This tool provides a quick overview of the status of a currently running mongod instance. In particular, it allows monitoring database operations by type (e.g., insert, query, update, delete, etc.).
- `mongotop`. This tool reports the current read and write activity of a MongoDB instance.
- Node watchdog (available in MongoDB 4.2). This tool monitors MongoDB directories—such as `dbpath`, journal, log, and audit directories—to detect filesystem unresponsiveness.
- Database Profiler. Using this tool, you can collect detailed information about operations run against a mongod instance. It's useful for identifying long-running operations and slow queries.

There are also several important commands that provide information on the MongoDB instances and clusters:

- `serverStatus`. This command provides a general overview of the status of the database, including disk and memory usage, number of connections, and index access.
- `dbStats`. This command reflects the amount of storage used. You can use `collStats` command to reflect the same information for individual MongoDB collections.
- `replSetGetStatus`. Data obtained from this command can be used to ensure that replication is properly configured and to check the connections between members of the replica set. In particular, it's important to watch the `optimeDate` parameter and check the time difference between the primary and the secondary members. It's crucial that the times of all members are synchronized.
- `db.locks.find()` (use config database). This command allows monitoring the status of stale locks that may negatively affect the write/read throughput.

4.9.1 Monitoring MongoDB in Kubernetes

In Kubernetes, MongoDB processes run in containers wrapped by such abstractions as pods and stateful sets. To monitor MongoDB in Kubernetes, you'll need a container-granular monitoring solution integrated with Kubernetes. Fortunately, Kubernetes has a developed ecosystem of monitoring tools capable of scraping container metrics.

One of the best monitoring tools for MongoDB is—without doubt—Prometheus, originally developed by the team at SoundCloud. This tool can collect and aggregate metrics from specified monitoring targets and expose it to applications via HTTP endpoints and provide to analytical and monitoring backends (e.g., Grafana, Elasticsearch). Also, Prometheus ships with a powerful PromQL query language that allows filtering and searching through metrics and supports alerting and processing time series data.

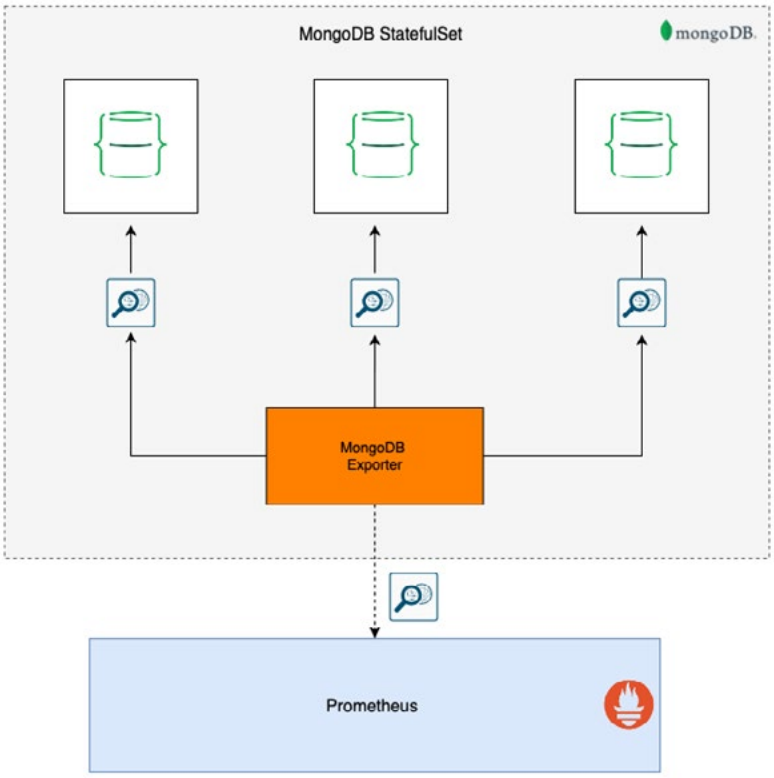

Image 11: MongoDB Monitoring with Prometheus

Prometheus can be easily configured to collect MongoDB metrics. To export MongoDB metrics, one should first install one of the community-supported MongoDB metrics exporters and configure it to send metrics to Prometheus.

Monitoring MongoDB is very important; however, there are many other components of the Kubernetes system that may affect MongoDB performance, and they should be also monitored. For example, important insights on the performance of your MongoDB deployment can be gained by monitoring MongoDB storage. When using Portworx, you can monitor Portworx volumes associated with MongoDB stateful sets through Prometheus. After Prometheus is configured to monitor the Portworx cluster, you can see the status of your MongoDB volumes (capacity, I/O load, disk errors, etc.) You can also configure the Prometheus AlertManager to handle alerts sent by the Prometheus server. Portworx has a predefined set of alerts related to the cluster, nodes, disks, volumes, and pools. They can be sent to the correct receiver via email, Slack, and PagerDuty.

PX-Autopilot

You can turn MongoDB metrics into a source of action using PX-Autopilot, a capacity management automation component of Portworx Enterprise.

PX-Autopilot is a rule-based engine connected to a monitoring target that responds to changes in it. It allows users to specify monitoring conditions and actions needed to be taken when the condition occurs. The PX-Autopilot workflow looks as follows:

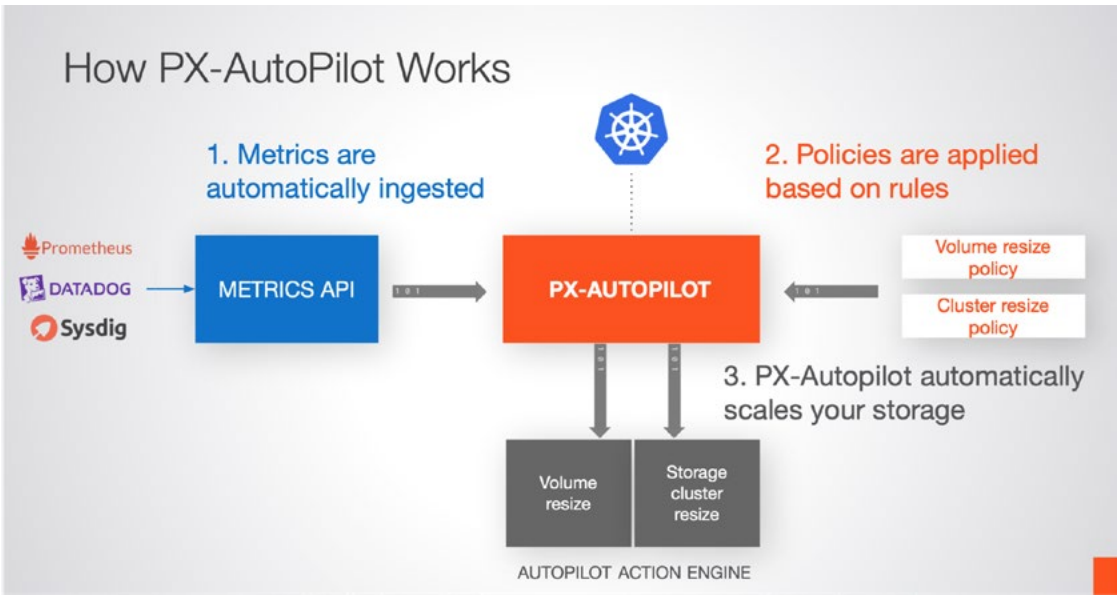

Image 12: PX-Autopilot

First, PX-Autopilot automatically consumes metrics from the tools you are already using to monitor your Kubernetes environment like Prometheus—for example, metrics like storage utilization at the individual volume or even cluster level.

Second, it compares these metrics against policies that you have defined.

Finally, PX-Autopilot takes action on your cluster to make its state match the policy.

One of the use cases for PX-Autopilot is dynamic volume resizing. For example, you can use Prometheus to monitor MongoDB volumes for capacity changes and trigger volume resizes without any application downtime when the volumes run out of memory.

You can also use PX-Autopilot to dynamically scale the entire Portworx storage pool. In this case, PX-Autopilot also monitors the cluster metrics, and when it detects high storage usage, it communicates with Portworx to resize the pool. Currently, the storage pool resizing feature supports AWS, Microsoft Azure, and VMware vSphere volumes.

5. CONCLUSION

MongoDB is a powerful database system that combines many features of NoSQL and SQL approaches. Having the best of two worlds makes MongoDB suitable for diverse use cases, including finance and e-commerce. Thanks to its built-in replication and horizontal scaling features, MongoDB is also a good choice for large distributed deployments spanning multiple nodes and datacenters.

As we demonstrated in this paper, running MongoDB on Kubernetes can provide many additional benefits. In particular, with Kubernetes, companies can automate a MongoDB-dependent stack's release/update cycle, bridge the gap between development, testing, and deployment of their applications, and benefit from integrating MongoDB with monitoring pipelines provided by the Kubernetes ecosystem. They can also take advantage of large cost savings associated with containerization of their stack.

However, running MongoDB on Kubernetes involves a lot of challenges. First, there is still a gap between the support for stateless and stateful services in Kubernetes. To run MongoDB efficiently, one needs to guarantee application-consistent MongoDB backups, efficient cross-node failover, storage security, and HA. Kubernetes alone does not provide enough tools to run MongoDB in production. Fortunately, solutions such as Portworx Enterprise provide all that is needed to enable MongoDB data HA and data protection mechanisms in place when running MongoDB in Kubernetes. The benefits of Portworx extend beyond data protection and security, though. As we demonstrated in this paper, the Potworx platform provides efficient storage, security, RBAC, monitoring, backup, disaster recovery, and numerous other features that make running MongoDB on Kubernetes fairly simple. We hope that with the help of Portworx your Kubernetes transition can be much smoother and faster.

portworx
by Pure Storage

Portworx, Inc.
4940 El Camino Real, Suite 200
Los Altos, CA 94022
Tel: 650-241-3222 | info@portworx.com | www.portworx.com