# Expert's Guide to Running Apache Cassandra on Kubernetes

portworx

Cassandra

### WHO SHOULD READ THIS GUIDE?

**Platform Architects** building a Container as a Service (CaaS) platform that offers Cassandra 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 Cassandra cluster

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

# 1. INTRODUCTION

Apache Cassandra is a very powerful open-source, distributed, NoSQL database system that is particularly suitable for managing large volumes of data spread across multiple servers and data centers. Powered by simple and scalable NoSQL technology and taking advantage of peer-to-peer cluster design and in-memory data caching, Cassandra provides strong High Availability (HA) and data consistency guarantees and fast read and write performance combined with low latency. Also, Cassandra is one of the most scalable databases, powering some of the largest database clusters managed by such tech giants as Apple, eBay, Instagram, Netflix, Reddit, and many others. For example, Apple has over 75,000 nodes with 10 PB of data, and Netflix has over 2,500 nodes with 420TB and over 1 trillion requests per day. Cassandra's scalable architecture allows read and write throughput to increase linearly with the addition of new nodes.

Cassandra's distributed architecture works well with containers and container orchestration platforms like Kubernetes. However, running Cassandra on Kubernetes requires a deep understanding of the platform's internals and fine-grained configuration of Cassandra deployment to meet requirements of data HA, data consistency, storage scalability, backup, disaster recovery, and security. In this guide, we'll discuss how to make Cassandra ready for Kubernetes, focusing on the best production practices in the above-mentioned areas. The paper covers the following topics:

- Key features and use cases.
- Architecture and innovative solutions—such as consistent hashing, tunable consistency, in-memory data structures, Bloom filters, Merkle trees, etc.
- Important configuration for Cassandra's HA, performance, throughput, etc.
- Key options for deploying Cassandra on Kubernetes (manual deployment, Helm Charts, Cassandra operators).
- How to design a HA Cassandra cluster on Kubernetes using Cassandra's built-in features and Portworx volume replication.
- Storage requirements for Cassandra in Kubernetes and how to meet them using Portworx.
- How to make Kubernetes-based storage for Cassandra more secure.
- How to ensure Cassandra's fault tolerance using cross-cluster migration.
- Using the Portworx platform to enable efficient Cassandra disaster recovery and backup.
- Integrating Cassandra with the Kubernetes monitoring pipeline.

# 2. CASSANDRA ARCHITECTURE AND FEATURES

Cassandra architecture enables strong fault tolerance, HA, and write and read persistence combined with an efficient trade-off between latency and data consistency. In this section, we discuss key concepts and architectural decisions that make Cassandra so scalable and fast.

## 2.1 Cluster design

Cassandra is a distributed database platform that runs in a compute cluster where a single logical database is spread across all participating nodes. Also, Cassandra is a master-less database system in which all the nodes are identical peers and where any node can serve requests of database clients. This approach ensures no single point of failure and HA.

However, how does Cassandra manage to maintain the cluster state in the absence of master nodes? The solution is based on the peer-to-peer principles. A Cassandra cluster is bootstrapped using the configured list of nodes that participate in the cluster called a "seed list." Nodes that join the cluster spread information about their state using a gossip protocol. In this protocol, each node passes its status to a subset of other cluster nodes. This mechanism turns out to be as efficient in failure detection and intra-cluster communication as traditional quorum-based or master-based approaches used in other distributed database systems.

Also, as we've said, Cassandra spreads data across nodes participating in the cluster. This is done using a consistent hashing algorithm that maps Cassandra row keys to nodes. The range of tokens from the algorithm is a fixed circular space that can be visualized as a ring (see the image below). Consistent hashing does not only evenly distribute data across nodes but also minimizes key remapping after the addition/removal of nodes. On average, it requires k/n maps to be remapped in contrast to other hashing algorithms where a large portion of keys need to be remapped when the cluster structure changes.

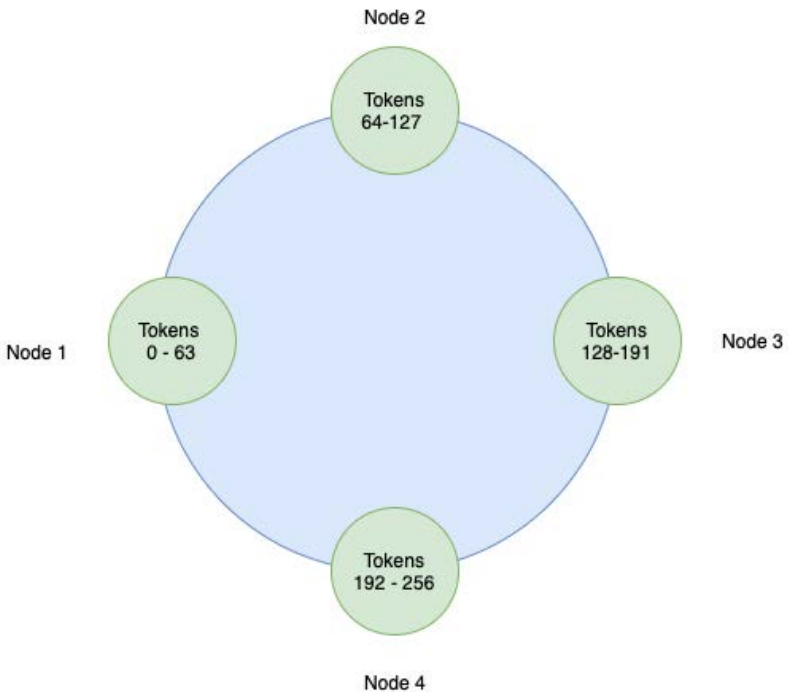


Image 1: Token distribution within a Cassandra ring

After the Cassandra cluster is set up, clients can send requests to any node. The communication can be done via the thrift protocol or Cassandra Query Language (CQL). Because Cassandra is a master-less system, any node to which a client has connected becomes a coordinator node. This node is responsible for managing the request and notifying the client of a success or failure. For example, if the client performs a write operation, Cassandra nodes should satisfy the consistency level that defines the number of replicas that should acknowledge the write operation before the coordinator notifies the client of a successful mutation.

## 2.2 High Availability (HA) approach

Cassandra ensures HA through automatic replication of data across nodes, automatic failed nodes replacement, and cross-datacenter replication. Cassandra supports three replication strategies: simple replication, network topology strategy, and transient replication.

In a *simple* replication strategy, all nodes are treated identically as if they were a part of one large datacenter. The logical subdivision of the cluster in racks and datacenters is ignored. Replicas are placed across nodes according to the replication factor (RF), and nodes are selected in a clockwise fashion.

In contrast, the `NetworkTopologyStrategy` is aware of cluster topology including registered data centers and their racks. It allows specifying a replication factor for each datacenter in the cluster and makes sure that replicas are not stored on the same rack. Using this strategy, Cassandra can be easily configured to work in a multi-datacenter environment to facilitate efficient failover and disaster recovery.

In both strategies, to discover a network topology, Cassandra uses a mechanism called "snitches." Snitches are processes that determine the proximity of nodes within a ring. This proximity information is used to determine the locality of a particular copy.

Finally, Cassandra has a concept of transient replication. It allows replicating data that hasn't been incrementally repaired, which allows saving on storage resources.

## 2.3 Tunable consistency

Cassandra implements the "eventual data consistency model" that guarantees that, provided there are no new updates, all nodes will eventually return the last updated value. Users can specify various data consistency levels for read and write operations depending on their data consistency requirements. The highest level of consistency is `ALL`, in which all nodes should acknowledge the success of operation before the coordinator node sends the response (see more in the next sections). It's important to stress that the consistency level determines the number of acknowledgments, not the number of nodes that receive a replica. The latter is specified by the replication factor that is often tuned along with the consistency level.

## 2.4 Data storage and management systems

Cassandra has a multi-layered data storage system tailored for fast performance and fault tolerance. This is achieved by three building blocks: commit-logs, memtables, and SSTables.

### 2.4.1 Commitlogs

All writes to Cassandra are initially saved to commit-logs—append-only logs of all changes local to a node. Commit-logs are stored on disk to ensure data durability in case of unexpected shutdown of a node, which may cause the deletion of data kept in Cassandra in-process RAM.

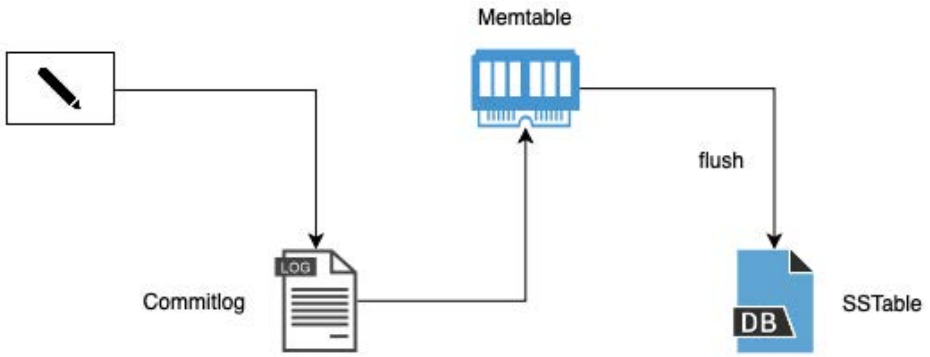


Image 2: Cassandra storage structures

### 2.4.2 Memtables

Memtables are in-memory structures to which data is written after it has been persisted in commit-logs. Memtables are stored in RAM to ensure fast access to the recently used data. Eventually, memtables are flushed onto disk, becoming immutable SSTables. This happens when their memory usage exceeds the configured threshold or the commit-log approaches its maximum size and triggers memtable flushes in order to free commit-log segments.

### 2.4.3 SSTables

SSTables are the final location of data on disk after it is passed through commit-logs and memtables. Cassandra triggers compactions, which combine multiple SSTables into one. Once the new SSTable has been written, the old ones can be removed.

## 2.5 Other innovative features

### 2.5.1 Bloom filters

Cassandra uses a probabilistic structure known as bloom filters to make reading operations faster. Bloom filters predict whether data exists in a given file. They cannot guarantee the correct answer, but it's possible to make Bloom filters more accurate by allocating more RAM to them (see more details in the next sections).

### 2.5.2 Merkle trees

Cassandra uses Merkle trees to ensure that nodes receive data blocks unaltered. Merkle trees are hash structures where the leaves represent the hashes of data blocks (e.g., SSTables) and subsequent nodes are hashes of their children. This structure allows creating a hash representation of complex data. Cassandra's Merkle trees compare data between neighboring nodes, and if there is a conflict, the nodes are repaired.

## 3. CASSANDRA REQUIREMENTS AND BEST PRACTICES

In this section, we discuss best practices for running Cassandra that are applicable for any environment—whether it's bare metal, cloud, or Kubernetes.

### 3.1 JAVA

Cassandra 3.x requires the latest version of Java 8, either Oracle Java Standard Edition 8 or OpenJDK 8. Since Cassandra runs within a Java VM, it uses a JVM heap pre-allocated with Java's `Xmx` system parameter. A sufficient heap amount is needed to serve memtables and other in-process RAM threads. A good rule of thumb is to allocate no less than 2GB and no more than 50% of the system RAM for Cassandra heap. Also, with heap sizes smaller than 12GB, users should consider `ParNew/ConcurrentMarkSweep` garbage collection, and with heaps larger than 12GB, they should use Garbage First Garbage Collector (G1GC) option.

### 3.2 RAM

Cassandra uses a significant amount of RAM off-heap for compression metadata, bloom filters, row, key, and counter caches, and an in-process page cache. It also takes advantage of the OS's page cache for storing recently accessed portions of data in RAM for re-use. With more RAM, memtables can also hold more recently written data, and fewer files can be scanned from disk during read operations. Thus, Cassandra will benefit from as much RAM as possible. This is also a reason why you should leave at least 50% of RAM off-heap. A minimal production server requires at least 8GB of RAM. Typical production servers have eight or more cores and at least 32GB of RAM (see "Operating Cassandra" guidelines). Cassandra documentation also recommends ECC RAM because Cassandra has few internal mechanisms to protect against bit-level corruption.

### 3.3 CPU

Since Cassandra is highly concurrent, it will benefit from parallel computation using multiple threads running on multiple cores. Adding more CPU cores increases the throughput of both reads and writes linearly and predictably. According to the official Cassandra documentation, a minimal production server requires at least two cores in non-loading testing environments. Typical production servers have eight or more cores.

### 3.4 Storage

Cassandra's SSTables work well both on hard disk drive (HDD) and solid-state disks (SSD). SSTables facilitate linear reads and writes, few seeks, and few overwrites, maximizing throughput for HDDs and the lifespan of SSD.

When using HDDs, it's recommended to have a commit-log directory on a physical disk rather than a simple partition. Also, data files (SSTables) should be stored separately from the commit-log directory on a dedicated physical disk. Separating the commit-log from the data directory allows writes to benefit from sequential appends to the commit-log without having to seek around the platter as reads request data from various SSTables on disk.

As far as network file system (NFS) and storage area networks (SANs) storage is concerned, using them with Cassandra is considered to be an antipattern and should be avoided. Servers with multiple disks are often better served with RAID0 or JBOD than RAID1 or RAID5.

## 3.5 Configure bloom filters for faster reads

Bloom filters can be made more accurate in their probabilistic estimation of data location if more RAM is allocated to them. RAM settings can be tuned per table by adjusting the `bloom_filter_fp_chance` to a float between zero and one (the default value is 0.1). Values closer to zero yield better accuracy. However, keep in mind that as the value is closer to zero, the Cassandra memory usage increases non-linearly. For example, the bloom filter with the value of 0.01 will require about three times as much memory as the same filter with the parameter set to 0.1.

## 3.6 Clock synchronization and health check

It's important to synchronize the clocks across all Cassandra nodes and all client machines (API Gateway hosts) to 1-millisecond precision. Failing to do so can result in failed data synchronization and failure to start and configure machines correctly.

## 3.7 Disable swap

Apache Cassandra recommends disabling swap at least for the Cassandra process. Preferably, swap should be disabled globally at the node running Cassandra.

## 3.8 OS and memory limits

When the write/read load is heavy, Cassandra may need to simultaneously access many SSTables files at once. By default, most operating systems set a limit on the number of open file descriptors (i.e., opened files), which may be insufficient if Cassandra has many open connections and read files. Thus, it is recommended to have at least 100,000 allowed file descriptors for the Cassandra node processes as a starting point. When running Cassandra on Linux, you can edit `/etc/sysctl.conf` and configure Ulimit to allow 100,000 or more open files.

## 3.9 Model data Cassandra way

Data modeling for Cassandra may be very counterintuitive for people coming from the RDBMS world. To create a good Cassandra data model, you should have a deep understanding of Cassandra data structures and query patterns for your data, which is hard when you don't have enough queries to analyze. However, there are several tools that can help you build better Cassandra schemas and data models from scratch:

- Hackolade. This is a tool for schema design for Cassandra (and other NoSQL databases) that supports Cassandra CQL concepts—such as partition keys and clustering columns—as well as data types including collections and UDTs.

- Kashlev Data Modeler. This tool automates data modeling methodology and best practices described in Cassandra documentation. It can be used to identify access patterns and implement conceptual, logical, and physical data modeling and schema design. It also includes model patterns that you can optionally leverage as a starting point for your designs.
- DataStax DevCenter. This tool allows managing Cassandra schema, executing queries, and managing multiple CQL scripts and connections to multiple clusters. The tool also ships with a query trace feature to gain insights into the performance of queries.

## 3.10 Configuring consistency levels

As we've already mentioned, Cassandra provides a tunable consistency feature that allows controlling the trade-off between latency and HA. With Cassandra, you can configure read and write consistency levels. The write consistency level defines the number of nodes to acknowledge the write before the client receives a success response. Similarly, the read consistency level defines the number of nodes to return the requested data before it is returned to a client.

A good practice is to make read and write consistency levels overlap so that at least one replica is guaranteed to participate in both the read and write request. The formula for such an overlap is W + R > RF where W is the write consistency, R is a read consistency, and RF is the replication factor. If RF = 3, a QUORUM consistency level will require responses from at least two of the three replicas. If QUORUM consistency level is used for both reads and writes, at least one of the replicas is guaranteed to participate in both the write and the read request, thus guaranteeing that the latest write is returned to the client.

## 3.11 Optimize Cassandra driver configuration

Cassandra drivers can make your cluster much more resilient if you take the time to configure them for your requirements. For example, a properly configured driver can route queries away from slow nodes or implement automatic fall back to lower consistency levels as needed to prevent issues.

# 4. BEST PRACTICES FOR RUNNING CASSANDRA ON KUBERNETES

Designed to operate in a distributed environment and handle large data volumes, Cassandra can directly benefit from containerization and container orchestration. Container orchestration can provide Cassandra with all the advantages of automation, operation, scaling, and monitoring.

## 4.1 Cassandra Kubernetes deployment options

Kubernetes is a mature platform with developed tooling for all major enterprise applications including Cassandra. Whether you are planning to run Cassandra in the self-hosted or bare metal Kubernetes clusters, there are several deployment options that may suit different levels of expertise and different tasks. Let's briefly discuss these methods.

### 4.1.2 Manual deployment

The manual deployment of Cassandra in Kubernetes requires a deep understanding of Cassandra architecture and the Kubernetes platform. Deploying Cassandra manually is hard and may not be an option for all companies.

However, if you are still planning to go with this option, two Kubernetes abstractions that help you deploy it are StatefulSets and Headless Services.

A StatefulSet is a Kubernetes primitive that allows maintaining a sticky identity and persistent network identifiers for database pods. These features make stateful sets a default option for deploying stateful services on Kubernetes. 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 Cassandra-specific configurations—such as seed lists, rack, data centers, JVM heap size, etc. For more information on deploying Cassandra as a stateful set, you can consult this tutorial from the Kubernetes blog.

Also, Cassandra nodes should be easily discovered by other nodes via the gossip protocol. Default stateless Kubernetes services abstract the identity of underlying pods, so they are not suitable for Cassandra'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 Cassandra nodes.

### 4.1.3 Cassandra 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 (Deployments, Services, service accounts, etc.) needed to deploy an application on Kubernetes. Charts embody best practices for configuring dependencies and Kubernetes resources for containerized applications. There are several Helm charts for Cassandra available from the Helm incubator repository and Bitnami.

Deploying Cassandra 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.

### 4.1.4 Using Cassandra Operators

An Operator is a method of packaging, deploying, and managing applications in Kubernetes. It may be thought of as a runtime that manages the full lifecycle of an application, including deployment, rolling upgrades, authentication, data backup, etc. It can be used by a person who knows Cassandra well without necessarily having a comparable level of Kubernetes expertise.

To give you an idea of how Cassandra Operators can benefit your deployment, let's discuss the node decommissioning scenario. An Operator would create and register custom Cassandra resources with the Kubernetes API and run a set of controllers managing these resources. For example, a controller could be watching the Cassandra 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 redistribution operation to remap the tokens managed by the deleted node to other nodes. When the controller confirms that this operation

has succeeded, it changes the stateful set definition to allow Kubernetes controllers to gracefully remove the node.

Cassandra Operators can be also used to configure Cassandra cluster topology, including the number of datacenters and racks in them. As you already know, Cassandra allows organizing the cluster nodes into multiple distributed layers. These layers are nodes, racks, datacenters, etc. For example, a rack is a logical grouping of nodes within a single ring. It is used to ensure that replicas are distributed among different logical groupings.

The cluster topology management feature discussed above is, for example, provided by Cassandra Operator CassKop (see the description of CassKop at the end of this section). CassKop Operator allows declaring the number of datacenters and racks within each datacenter, specifying a replication factor and other useful settings.

If you are convinced of the usefulness of the Operator pattern, below are some Cassandra operators to select from.

1. Instaclustr Operator. This Operator is developed by Instaclustr, a company that offers managed Cassandra clusters. This Operator has built-in support for monitoring, high-level cluster management via CRD, and detailed instructions on making backups of Cassandra clusters. The Operator is still in the alpha stage, and API changes can still be made preventing backward compatibility.
2. Jetstack Navigator. Jetstack navigator is a custom apiserver operating behind kube-aggregator, allowing you to manage custom Kubernetes resources. It can be used to implement custom resources. At this moment, Jetstack Navigator provides support for Elasticsearch and Cassandra.
3. CassKop. This Operator is based on the popular operator-sdk framework and provides a number of automation tasks—such as Cassandra deployment, scaling, adding/removing nodes, configuring the Cassandra and JVM, backups, data restores, and upgrades. CassKop provides monitoring out of the box using Instaclustr Prometheus exporter to Prometheus/Grafana.

## 4.2 Cassandra High Availability on Kubernetes

Cassandra ships with several features enabling HA out of the box; however, the cluster needs to be properly configured to leverage them effectively.

First, Cassandra production deployments require at least three nodes for `QUORUM` consistency level. This level can only be created with an uneven number of nodes.

Also, it is recommended to use an appropriate replication factor. A replication factor of three is correct in most cases. It is also preferable to use a `NetworkTopologyStrategy` from the beginning because it allows for datacenter-granular replication. When using a network topology strategy, you may also choose to map Cassandra racks to distinct availability zones so that disruptions in one rack do not affect data in other racks.

Even more fault tolerance can be secured by spanning clusters across different cloud providers or using a hybrid cloud strategy (private network cluster combined with a public cloud). It might also be an option to split clusters by application function. For instance, one cluster may be used solely to take in data and another to store, process, or analyze it.

Another important consideration when configuring Cassandra for HA is finding the right balance between data consistency and latency. As we've already mentioned, `QUORUM` level configured for both read and writes with a proper replication factor can guarantee that the latest reads and writes are always returned to the users. However, for some use cases—such as scientific data like sensor readings—lighter consistency guarantees—such as consistency level ONE for all writes—can be enough.

Also, you should be careful when configuring `LOCAL_QUORUM` for a multi-datacenter strategy. When this setting is selected, a read or write request is acknowledged to the client once it has achieved quorum within the datacenter it is talking to. However, it is crucial that any read following the write queries the same datacenter. If the read request queries a different datacenter, it may turn out that the queried datacenter is not yet up to date with the latest data.

To secure even stronger multi-datacenter consistency, you can use `EACH_QUORUM` consistency level. With this configuration, a read or write request will be completed once it has achieved quorum across all the datacenter. However, this consistency level can significantly increase latency in the cluster, which is not a desired behavior.

## 4.3 Enabling Cassandra storage High Availability with Portworx

Cassandra replication and tunable consistency ensure HA and fault tolerance at the application level. However, additional steps should be taken to ensure Cassandra's HA in Kubernetes. In particular, Cassandra's deployment on Kubernetes can benefit from storage replication and storage health monitoring. The Portworx Enterprise storage platform can be used to enable these features for Cassandra on Kubernetes.

The Portworx Enterprise storage platform is an end-to-end storage and data management solution for containers and container orchestration platforms, such as Kubernetes. It supports intra-cluster data layer aggregation and container-granular storage provision, storage High Availability, security, backup, disaster recovery, fast failover, and other storage-related features in distributed container environments.

Portworx aggregates storage available in a Kubernetes cluster into a unified layer composed of different storage classes that can be provided as a service to containers via native Kubernetes abstractions and Portworx custom storage resources. Portworx storage layer is elastic and easily scalable. Instead of managing individual storage devices for each node in your cluster, you can leverage Portworx's storage virtualization to create a unified storage pool.

Storage provided by Portworx is HA by design, too. For each volume created with Portworx, you can create volume replicas and distribute them proportionally across all nodes in the cluster. These replicas are kept up to date using synchronous replication. Each database write to any of the volumes is synchronously replicated to all other replicas. Consequently, in case of an application failure or node shutdown, the new application instance spun up on a new node can have instant access to a copy of its data.

How does Portworx implement this behavior? Portworx ships with a built-in storage-aware scheduler called STORK (Storage Orchestration Runtime for Kubernetes). It is tightly integrated with kube-scheduler and other Kubernetes controllers. If a Kubernetes scheduler is configured to use an extender, it makes two API calls before making a pod placement decision: *Filter and Prioritize*.

A "filter" request is used by STORK to filter out nodes without the storage driver on them. Kubernetes does not provide this information by default. The "filter" request helps reduce the number of failed pod attempts, which results in faster and more efficient scheduling.

Similarly, a "prioritize" request allows STORK to select nodes that host a replica of the application data. Portworx checks which persistent volume claims (PVCs) are used by the pod and then queries the storage driver for the location of the pod's data. STORK uses this information to rank various nodes as to which node would provide the best performance when accessing the persistent storage from that node.

As a result, STORK makes sure that Cassandra has fast access to its data whenever the critical downtime or failure is detected. In this way, Portworx ensures HA and fast failover for Cassandra storage resources.

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.

In particular, node health monitoring is especially important for maintaining the HA of your Cassandra cluster. A common problem with stateful apps on Kubernetes is the "wear and tear" induced by the storage fabric. If the storage device fails, however, the app can still continue to run although writes are no longer persisted. These issues are hard to detect instantly. This can also result in the failure to reschedule apps to healthy pods, resulting in app unavailability.

Stork helps in these situations by failing over pods when the storage driver on a node goes into an error or unavailable state. This allows your applications to be truly highly available without any user intervention.

## 4.4 Cassandra backup and recovery

Even though Apache Cassandra is decentralized and has no single point of failure if properly configured, many scenarios—like accidental deletion due to operator error, misconfiguration, or global datacenter shutdown—can still compromise your data. These potential scenarios necessitate the development of an efficient backup and disaster recovery strategy.

Out of the box, Cassandra provides several backup and snapshot tools with varying degrees of efficiency and different levels of operator expertise required. These are snapshot-based backups, incremental backups, commitlog backups, and combinations thereof. Let's discuss these options in more detail.

### 4.4.1 Snapshot-based backups

Cassandra's command line interface called `nodetool` provides a snapshot command that allows taking node-level snapshots of Cassandra keyspaces. These one-time snapshots create a hard link to the current Cassandra data files, which can be accessed in the Cassandra backup directory.

To restore from a snapshot, Cassandra users must manually drain the node and copy the content of the snapshot to the data directory. The `nodetool` snapshot command is the simplest way to make a Cassandra database snapshot; however, its scope is limited. Most importantly, this tool creates a node-level snapshot that does not encompass a consistent state of the Cassandra cluster. To create a full snapshot of a Cassandra cluster, users should run this command on all nodes.

### 4.4.2 Incremental backups

Snapshots of the entire Cassandra cluster may be big in size and take considerable time to complete. Taking a full snapshot of the Cassandra cluster each time you want to backup data can result in higher latency and slower performance of the entire cluster. An alternative approach is to use an incremental backup strategy after the snapshot is taken. If this strategy is enabled, every time the `memtables` are flushed to SSTables, Cassandra creates a hard link to the new file in the incremental backup folder. That way, there is no need to make a full snapshot every time. This approach reduces the snapshot size after the first full snapshot and reduces the costs of extracting data from local servers and disks. They also allow saving on backup space because hard links are created only for new files and files created due to compaction are not hard-linked.

However, one should remember that the size of the incremental backup director can quickly grow out of bounds. To prevent this, the administrator is supposed to periodically clear the contents of the folder. Also, overusing incremental backups can create many small backup files that need to be compacted. This can result in a slow start from a new machine after a cluster outage.

### 4.4.3 Commit-log backups

Commit-log backups work similarly to incremental backups but target commit-logs instead of SSTables. This method is usually combined with periodic full snapshots and incremental backups. Using these three methods together can enable a strong backup policy for your Kubernetes cluster.

As we can see, Cassandra provides multiple backup options. However, the discussed options require a lot of manual work and are prone to errors. They may work for staging and test/dev stages but may need additional tooling (scripts, processes) and Operators if run in production at scale. Errors compromising data and causing downtimes can happen at any stage of the process.

Also, when managing Cassandra on Kubernetes, we should consider additional scenarios that can happen inside the Kubernetes cluster and tune our disaster recovery strategy accordingly. We'll discuss these additional measures in the next section.

## 4.5 Portworx backup tools for Cassandra

Cassandra enables data backup and recovery at the application level. However, it may be useful to introduce additional backup and recovery mechanisms at the level of physical disks and Kubernetes storage abstractions—such as Kubernetes config, secrets, service accounts, CRDs, etc.—as well as the entire Kubernetes cluster running Cassandra. Portworx provides several tools to implement the efficient backup strategy and cross-cluster disaster recovery for your Cassandra deployments in Kubernetes.

### 4.5.1 PX-Backup

PX-Backup is a critical component of the Portworx Enterprise that enables container-granular, application-aware, Kubernetes-aware, and multi-cloud backup options for your Cassandra deployment. PX-Backup is closely integrated with Kubernetes API and resources, which make Cassandra backups compatible with Kubernetes and fast to restore consistently and effectively.

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 relevant backup metadata.
- Auditing backup history.
- 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.

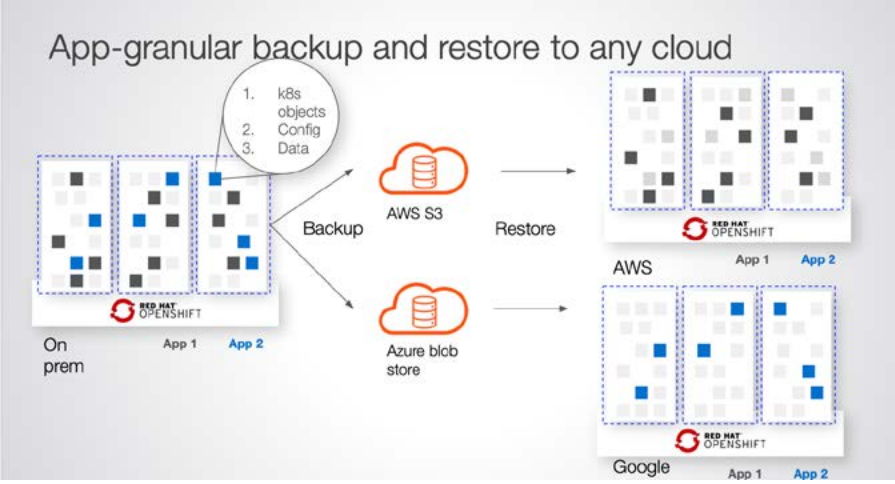


Image 3: App-Granular Backups with Portworx

Making application-consistent snapshots is very important for database systems such as Cassandra that store pending operations in memory. Taking a snapshot without clearing these operations may result in the inconsistent database state after restore. With PX-Backup, you can specify pre-snapshot and post-snapshot rules with the list of operations to run against Cassandra before and after the snapshot is taken. Portworx can pass these rules to Cassandra for execution. As a result, Cassandra failover will be faster and the database state consistent.

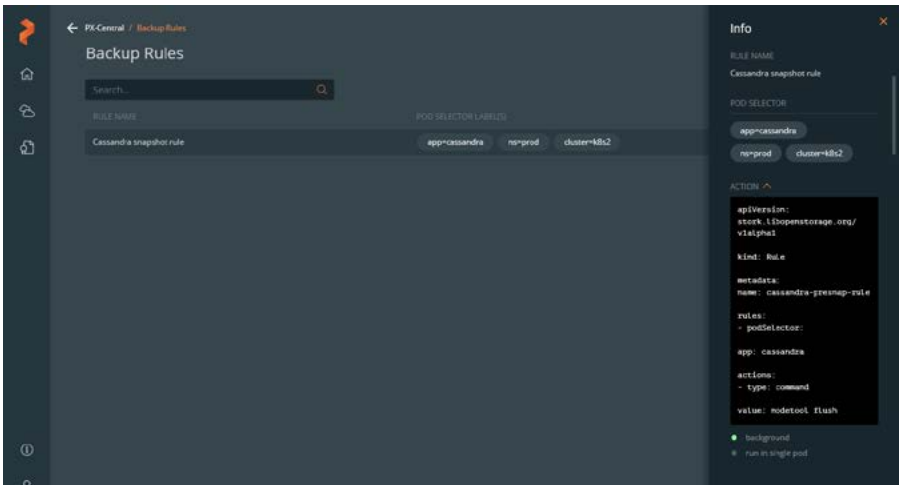


Image 4

Also, snapshots of Cassandra taken by PX-Backup may include not just data but also Kubernetes resources and objects—such as Secrets, ConfigMaps, and Persistent Volumes. Thus, the backup contains all the resources and configuration needed to restore Cassandra deployment "as is" without the need to re-create Kubernetes manifests and resources. This secures lower RPO and faster failover.

Thirdly, PX-Backup supports multi-pod and namespace-aware backups. For example, backing up the entire namespace may be useful if you want to back up not only Cassandra data but the entire stack associated with it (e.g., application servers, monitoring pipelines, etc.) If this stack is located within a single namespace, you can take a namespace-wide snapshot that can be easily restored to the same namespace. PX-Backup maintains the metadata about the backed-up pods so you can easily audit the process and restore the same pods to the same place.

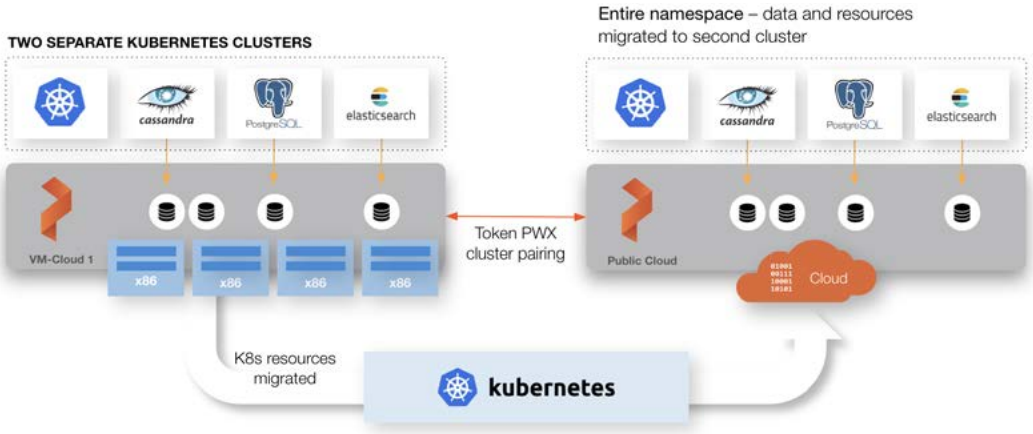


Image 5: Cross-Cluster Namespace Migration

All this comes with the multi-cluster support. PX-Backup allows managing application backups for multiple clusters from one central location. For example, you might have application backups from the AWS cluster stored in S3 and application backups from Azure or Google Cloud. Whenever you want to restore the app, you can select a cluster it was backed up from, and PX-Backup uses the backup metadata to perform a restore. It's very convenient when you run several Kubernetes clusters on different clouds or on-premises. PX-Backup is seamlessly integrated with cloud drives from GCP, AWS, and Azure, which allows making backups with cloud storage provided by these platforms and importing backups from them.

PX-Backup also provides a number of utilities for managing backups, including point-in-time restore of data and filtering backups on cluster(s), namespace(s), and labels. All these features allow automating backup management for your applications on Kubernetes.

## 4.6. Cross-datacenter disaster recovery of Cassandra with Portworx

Portworx replicated volumes provide a good starting point for disaster recovery; however, you should also consider scenarios when the entire cluster goes down. This scenario can be addressed with the cross-datacenter disaster recovery using Portworx.

Portworx Enterprise includes PX-DR, a package that implements cross-datacenter disaster recovery for containerized applications running in Kubernetes as well as other orchestration systems.

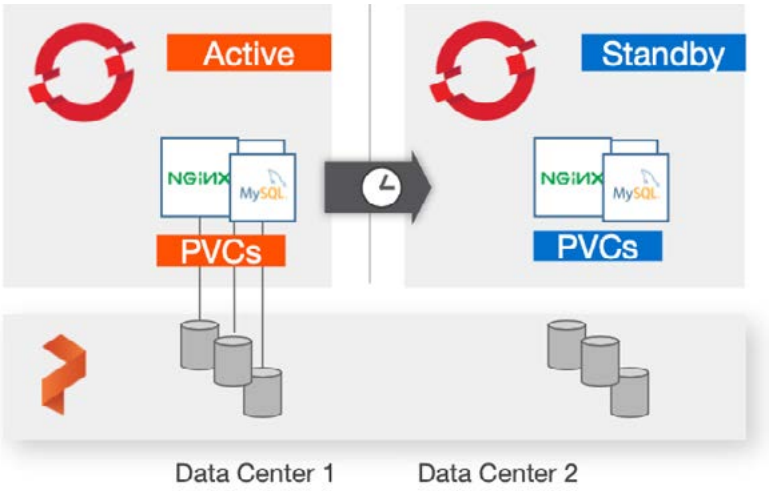


Image 6: PX Disaster Recovery with Active and Standby Clusters

PX-DR can operate in two modes—synchronous DR and asynchronous DR—suitable for different patterns of Kubernetes cluster deployment.

In both modes, you have at least one active and one standby cluster paired together. An active cluster copies data and Kubernetes resources to the standby cluster based on some schedule or incrementally. A standby cluster is not a passive storage for your Cassandra data and Kubernetes resources, though. It has all active controllers and processes running to be able to become an active cluster when the current active cluster fails. Portworx has a built-in failure detection loop that allows detecting such a failure once it occurs.

Synchronous and asynchronous modes substantially differ in terms of Kubernetes cluster and Portworx cluster design.

In a synchronous DR mode, Portworx is installed in a single stretch cluster across multiple Kubernetes clusters spanning a metropolitan area network with a maximum recommended latency of <=10ms. Low latency allows for synchronous replication of workloads to a standby and fast failover that meets low RPO and RTO targets. In particular, the synchronous DR mode has zero RPO and RTO of less than a minute. Also, the mode has a built-in fault-domain detection, so the backup replicas can be evenly distributed across the nodes of a standby cluster.

This mode is not suitable if your standby and active clusters live in different cloud regions.

In this case, an asynchronous DR mode is preferred.

In the asynchronous mode, Kubernetes clusters are located in different regions or datacenters and have high latency between them. To connect these datacenters, a separate Portworx cluster is installed in each Kubernetes cluster deployed in these datacenters. Incremental changes in Kubernetes applications and Portworx data are continuously sent to the standby cluster according to the migration schedule specified by the user. Even though active and standby clusters are eventually synchronized according to the schedule, due to the higher network latency, this mode will have an RPO of 15 minutes. The RTO of less than 60 seconds is ensured for this mode as well.

Users can also schedule synchronization between destination and source clusters if needed using migration schedule policies. They allow setting an interval for migration and specify objects and volumes affected 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.

Also, Portworx ships with the built-in migration monitoring tool accessible from the `storkctl` CLI. Using the tool, you can observe the state of migration, synchronization status, history of scheduled migrations, and other useful parameters that help you understand the state of your remote and source clusters.

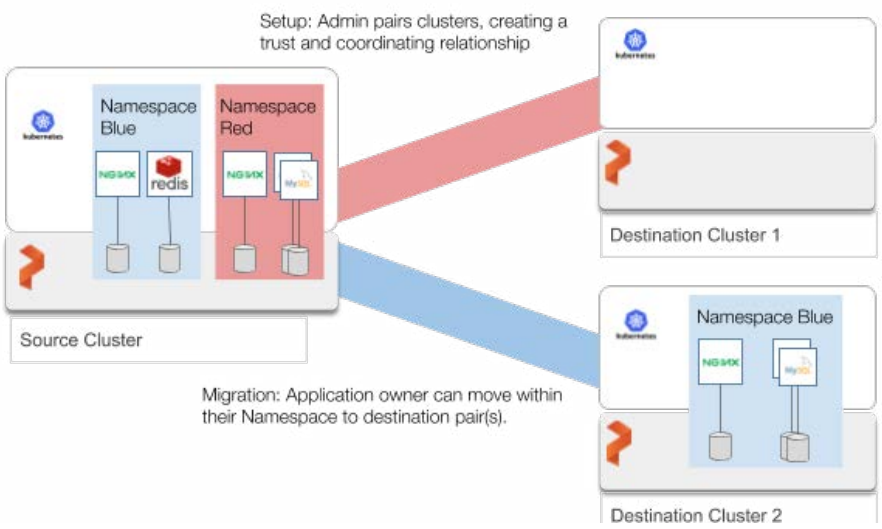


Image 7

Beyond disaster recovery, there are other helpful use cases for cluster migration with Portworx:

- Free storage on critical clusters by moving lower-priority data to remote clusters.
- Blue-green tests. Test new versions of Cassandra, Kubernetes, and Portworx with all configuration and data.
- Move workloads from dev/test environments to production without the need to manually provision the data. For example, developers may first use a region geographically close to them for the development and testing and then migrate applications to regions that are close to the users and customers.
- Move workloads from private on-premises clusters to cloud-hosted database clusters like Amazon EKS or Google GKE.
- Decommission a cluster in order to perform hardware-level upgrades.

In conclusion, a cross-datacenter disaster recovery feature makes it easy to implement failover and failback of a Cassandra database running on Kubernetes. You can easily activate applications on a destination cluster when the source cluster experiences outages and re-activate them when the origin cluster becomes healthy.

## 4.7 Cassandra security layer

Cassandra provides four security features that allow implementing baseline protection of your database(s):

- TLS/SSL encryption for client and inter-node communication
- Client authentication
- Authorization
- JMX Access

All these features are disabled by default, which means that an out-of-the-box Cassandra installation presents a large attack surface for malicious users. To provide a baseline level of security for your Cassandra cluster, you should consider enabling and configuring the following features.

### 4.7.1 User authentication

Cassandra provides a role-based authentication mechanism stored internally in Cassandra system tables. It regulates access to Cassandra keyspace and tables. Administrators can create users with specific roles using CQL commands. These roles can have superuser, non-superuser, and login privileges.

### 4.7.2 SSL/TLS

Cassandra encrypts in-flight networking communication between client machines and cluster nodes with the standard SSL/TLS mechanism. Cassandra manages client-to-node and node-to-node encryption separately. Keep in mind that Cassandra only encrypts in-flight connections, not physical storage (see below).

### 4.7.3 Authorization

Cassandra uses a whitelist approach to permissions, which assumes that a given role has no access to any database resource by default. If authorization is enabled on a node, all requests without required permissions are disabled.

### 4.7.4 JMX access

JMX (Java Management Extension) is a technology that allows managing and monitoring resources related to instances of a Java Virtual Machine (JVM). This is achieved by instrumenting resources with Java objects known as Managed Beans (MBeans) that are registered with an MBean server. JMX authentication controls access to JVM and JMX by third-party applications such as nodetool and monitoring pipelines collecting Cassandra metrics.

## 4.8 Cassandra security considerations for Kubernetes

Kubernetes has a large surface area that includes networking, container runtime, and internal components (kube-apiserver, kube-proxy) which should be protected against bad actors. Although Cassandra can ensure application-level security for your databases, you'll need additional protection. Portworx allows configurations to lock down the data layer and its APIs from malicious actors, which is especially important for distributed databases like Cassandra.

Portworx can provide storage-layer security, authentication, and authorization for physical storage media underlying your Cassandra clusters in Kubernetes.

First, Portworx can extend the default Cassandra TLS/SSL encryption that works on the in-flight data with the storage-level data encryption to protect files on disk. This can provide an additional layer of security for your Cassandra deployments.

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.

The second important concern is Kubernetes cluster security. Most databases have default authentication mechanisms and authorization. They are enough for enabling authentication in a database cluster. However, we need an additional authentication mechanism for users, applications, and services interacting with Cassandra as part of a Kubernetes cluster and Kubernetes API server. 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.

Portworx can enable additional authentication and authorization protection for the storage layer. It will allow your Cassandra users to control how database storage is accessed and managed. Kubernetes and Portworx security models meet when the user aims to create a Kubernetes resource using a Portworx volume. Here, the user does not only need authorization from Kubernetes but also needs to provide a token generated for Portworx that contains the roles and groups of the user trying to create a volume.

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.

Also, Portworx supports RBAC for authorization. Once the user is authenticated, Portworx will read the user roles from the JWT token to determine what actions the user can perform with volumes.

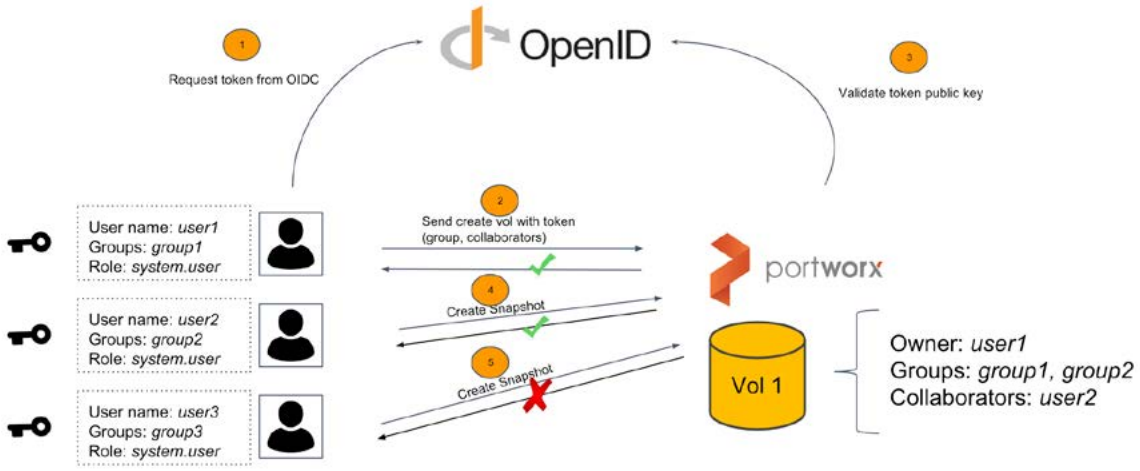


Image 8: Portworx Security Model

You can also specify ownership rights to control types of access (read, write, administrator) for specific volumes.

In summary, adding Portworx to your Cassandra 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 Cassandra cluster.

## 4.9. Embedding a monitoring pipeline

A monitoring pipeline is an indispensable requirement for managing a distributed system like Cassandra. Kubernetes introduces many layers of complexity—such as container runtime, custom resources, controllers, and networking structures—that should be understood to control resource allocation and performance of the Cassandra cluster. Moreover, due to the volatile nature of Kubernetes abstractions and nodes, critical events that can compromise the security and performance of your Kubernetes cluster are happening quite frequently. Repair processes, traffic spikes, taking snapshots, adding/removing nodes, and other activities can cause issues in your cluster that should be addressed immediately. Monitoring provides visibility into events happening in your cluster in real time.

Cassandra provides a number of useful metrics that can be scraped to build a comprehensive monitoring solution. These metrics are managed with the Dropwizard Metrics library, a popular Java library for collecting metrics. Cassandra metrics can be queried via JMX or pushed to external monitoring systems using a number of built-in and third-party monitoring plugins. Metrics are provided for a single node, so metrics aggregation should be implemented as well.

To collect Cassandra metrics in Kubernetes, we need a solution that understands JMX metrics format.

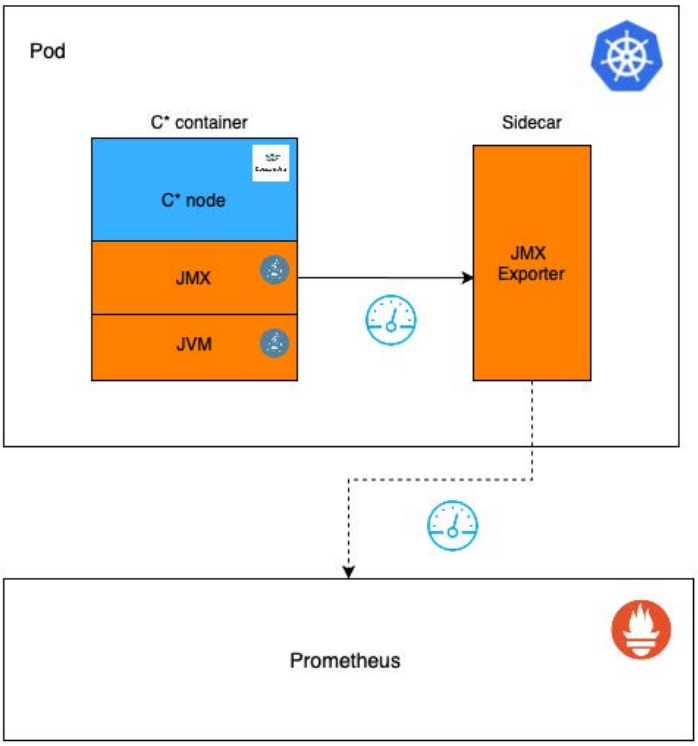


Image 9: Exporting Cassandra Metrics to Prometheus

Prometheus is one of the most popular cloud-native metrics solutions for Kubernetes that understands JMX metrics. It ships with the JMX exporter—a collector that can scrape and expose the MBeans of a JMX target. This exporter runs as a Java Agent that exposes an HTTP server and serving metrics of the local JVM.

To scrape Cassandra metrics to Prometheus, you can deploy JMX Exporter as a sidecar container in Cassandra pods and then configure the JMX agent in the Cassandra container to report its metrics to the JMX Exporter. Eventually, your Prometheus deployment can be configured to receive and process the JMX-exported metrics.

To visualize and analyze the metrics, you can connect Grafana to Prometheus or ship metrics to Elasticsearch to visualize and process them in Kibana.

Another option is the `cassandra_exporter` that was originally a fork of jmx-exporter aimed at easier integration with Apache Cassandra. The project supports a number of features—such as filterings on MBean's attributes, comprehensive metrics configuration, and a neat dashboard.

Also, the `cassandra-exporter` has several performance improvements. It restricts the Prometheus scrape frequency because metrics collection operations are expensive. Also, the `cassandra-exporter` supports caching metrics to improve query performance.

## 5. PX-AUTOPILOT

Monitoring storage resources is a special area of interest when running Cassandra in Kubernetes. It is important for understanding storage current capacity and preventing low disk problems. To add storage-aware monitoring for your Cassandra cluster on Kubernetes, you can use the PX-Autopilot solution. With PX-Autopilot you can turn your monitoring pipeline into a source of action and automation in your database clusters.

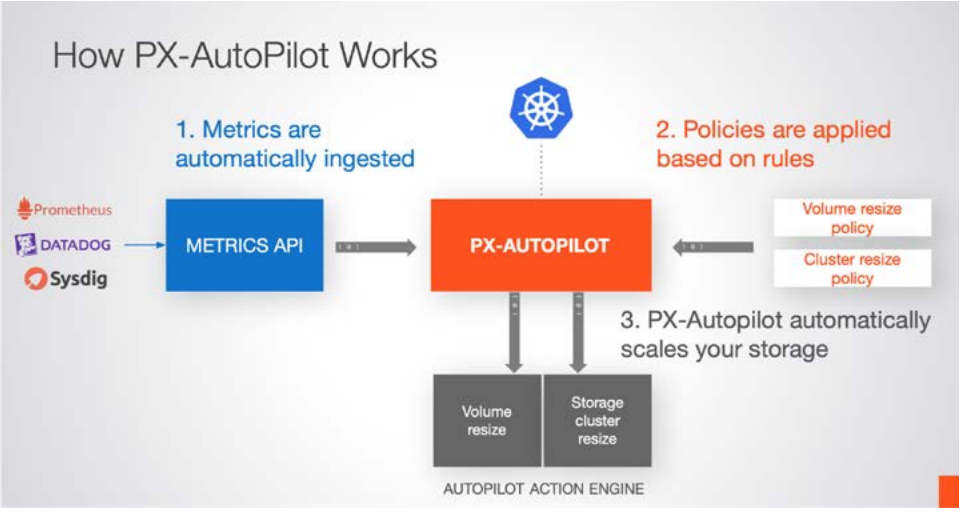


Image 10: PX-Autopilot

PX-Autopilot is a rule-based engine connected to a monitoring target and responding to changes in it. It allows users to specify monitoring conditions and actions needed to be taken when the condition occurs.

One of the use cases for PX-Autopilot is dynamic volume resizing. For example, you can use Prometheus to monitor Portworx volumes for capacity changes and trigger volume resizes 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 the high storage usage is detected, it communicates with Portworx to resize the pool. Currently, the storage pool resizing feature supports AWS, Microsoft Azure, and VMware vSphere volumes.

## 6. CONCLUSION

Apache Cassandra is a powerful database system that enables fast performance and scalability in multi-node distributed environments. However, adjusting Cassandra to your business case takes a lot of expertise and appropriate cluster design decisions. Fortunately, Cassandra architecture is compatible with containers and Kubernetes, so your team can benefit from automation and orchestration services offered by this platform.

In this paper, we discussed major requirements and best practices for running Apache Cassandra on Kubernetes. In particular, we learned how to use the Portworx Enterprise storage platform to implement the following features:

- Storage-layer replication of Cassandra Kubernetes volumes
- Configuring efficient failover of Cassandra using STORK on Kubernetes
- Implementing Cassandra node and storage health monitoring
- Extending the Cassandra security model with Kubernetes-level security and storage layer security, including volume encryption and RBAC
- Integrating a monitoring pipeline based on JMX metrics
- Adding efficient backup and disaster recovery mechanisms for Cassandra on Kubernetes

Using Cassandra Operators for Kubernetes together with Portworx seems to be the most reasonable approach to implementing these features. Cassandra Operators can provide many common Cassandra management tools and procedures out of the box so you don't have to reinvent the wheel. All in all, however, there are no one-size-fits-all solutions for all use cases. Putting Cassandra to production on Kubernetes will still require a lot of planning and cluster design decisions that satisfy your specific use case.

portworx

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