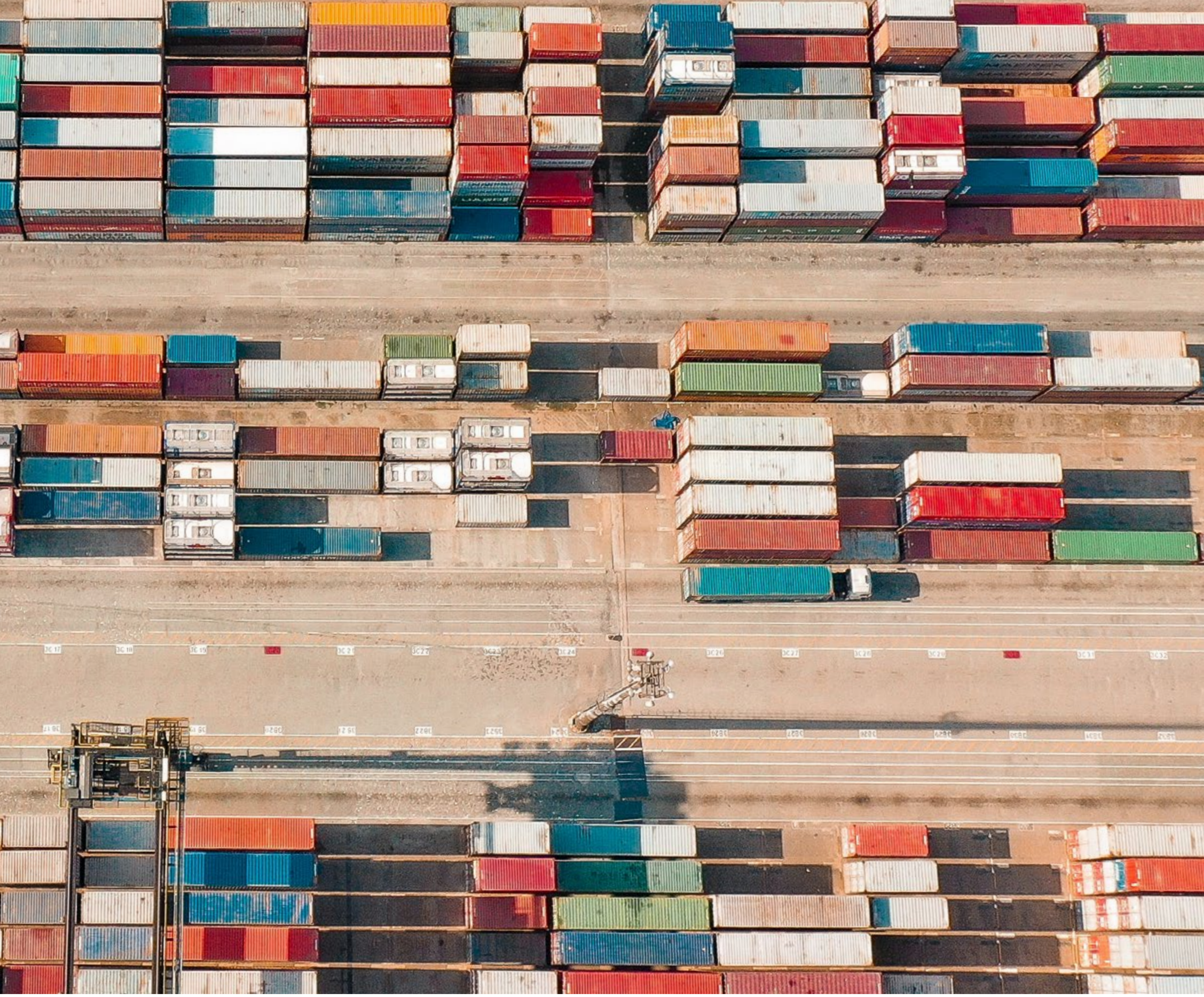


Expert's Guide to Running a Database-as-a-Service on Kubernetes

portworx
by Pure Storage

1. INTRODUCTION: WHAT IS DATABASE-AS-A-SERVICE (DBAAS) 4

2. DBAAS KEY BENEFITS AND FEATURES 4

2.1 Cost Efficiency 4
2.2 Scalability 5
2.3 Alleviating Database Administration Burden 5
2.4 Automation 5
2.5 High Availability 5
2.6 DBaaS as a Line of Business 5

3. BUILDING A DBAAS SOLUTION FOR KUBERNETES 6

3.1 How Containerization Enables DBaaS 6
3.1.1 Portability and Extensibility 6
3.1.2 Open Source and Free 7
3.1.3 Containers Are Compact and Fast to Launch 7
3.1.4 Microservices-Ready 7
3.2 Why Build DBaaS on Kubernetes? 7
3.2.1 Declarative Management 8
3.2.2 Support for Multiple Container Runtimes 8
3.2.3 Scalability 9
3.2.4 Support for Stateful Workloads 9
3.2.5 Cloud Provider Integration 9
3.2.6 Support for Rolling Updates 9
3.2.7 Ability to Turn your Database System into a Microservice 9
3.2.8 Built-in Resource Management and Namespacing 9
3.2.9 Power your DBaaS With Kubernetes Features for Free 10

4. MAKING DBAAS WORK ON KUBERNETES 10

4.1 Automate Database Lifecycle with DBaaS Operator 10
4.1.1 Custom Resources 10
4.1.2 Custom Controllers 11
4.1.3 Operator Pattern 11
4.2 Kubernetes Operator Example 12
4.3 Operator Ecosystem 13

5. STORAGE CONSIDERATIONS FOR DBAAS ON KUBERNETES 14

5.1 Basic Storage Requirements for DBaaS on Kubernetes 14
5.2 The Portworx Kubernetes Storage Platform 14
5.2.1 Replication 15
5.2.2 Storage Class of Service 16
5.2.3 Storage Policies 17

6. DBAAS SECURITY LAYER 17

6.1 DBaaS Security Considerations for Kubernetes 17

7. DBAAS BACKUP WITH PX-BACKUP 19

7.1 Other Backup and Snapshot Tools 20

8. DBAAS CLUSTER MIGRATION WITH PORTWORX 21

9. DBAAS DISASTER RECOVERY WITH PORTWORX 22

10. MONITORING 23

10.1 DbaaS Capacity Management with PX-Autopilot 23

11. CONCLUSION 24

WHO SHOULD READ THIS GUIDE?

Platform Architects offering a Database-as-a-Service (DBaaS) option for internal users running Kubernetes

Application Architects or Site Reliability Engineers (SRE) building or running a cloud-native solution for managing database clusters on Kubernetes

Database-as-Service Architects offering multi-tenant databases as a service to end users

1. INTRODUCTION: WHAT IS DATABASE-AS-A-SERVICE (DBAAS)

DbaaS is a special type of a SaaS (Software-as-a-Service) platform designed for managing databases. It allows automating many routine database management and administration tasks, such as installing, configuring, backing up and recovering, migrating, upgrading, and monitoring databases. With a DBaaS, a user selects some combination of a database version, a server type, number of database instances, availability zone, storage, database backup schedule, etc., and the DBaaS provider takes care of installing and configuring the database and provisioning all resources (CPU, RAM, storage) required by the database on the specified infrastructure. Once the database is installed, DBaaS users can log into and use the database and perform various database management tasks, such as scaling database resources, backing up, or migrating data, all via a centralized self-service portal.

With an effective DbaaS platform, small teams can support a large number of users, providing reliable operations and an easy, small-service user experience.

In this guide, you'll learn best practices for building a DBaaS platform on Kubernetes. In particular, we'll discuss:

- Major DBaaS benefits and features.
- How containerization and Kubernetes enables a multi-cloud and distributed DBaaS solution.
- Key prerequisites and requirements for building a DBaaS platform on Kubernetes.
- How to use the Kubernetes Operator Pattern to create a fully automated DBaaS offering on Kubernetes.
- Additional Kubernetes resources and tools you can leverage for your DBaaS offering, including authentication mechanisms, monitoring, persistent volumes, etc.
- Storage requirements for a Kubernetes-based DBaaS platform and how they can be met by the Portworx Enterprise storage platform.
- How to configure DBaaS storage security with Portworx to protect against data loss and data theft.
- Setting up efficient DBaaS backup and disaster recovery with Portworx on Kubernetes.
- Plugging in a Kubernetes monitoring pipeline to provide database performance metrics.
- How to efficiently migrate DBaaS-backed databases using Portworx cluster migration functionality.

2. DBAAS KEY BENEFITS AND FEATURES

In this section, we review the major benefits of DBaaS for infrastructure teams, administrators, SREs, and application developers.

2.1 Cost Efficiency

At the core of DBaaS is a cost-effective, on-demand model that reduces overhead costs of pre-provisioning on-premises or cloud infrastructure (servers, memory, storage) for your databases. Leveraging the DBaaS model, IT teams can outsource infrastructure management to DBaaS providers and get the ability to scale databases per their dynamically changing needs. With a DBaaS platform, databases can be easily installed and managed from a central web console in one click.

2.2 Scalability

Scalability is a major requirement for businesses in the era of fast growth and the global consumer base. Cloud virtualization technology that powers DBaaS infrastructure enables on-demand provisioning of computing resources independently of each other. DBaaS users can easily add more database nodes, memory, and storage resources to meet the growing demand of the consumers or operational needs of developers and DevOps teams. This can drastically reduce the administration burden for IT departments and offer a frictionless experience for IT specialists, who can perform their tasks more efficiently under tight time constraints.

2.3 Alleviating Database Administration Burden

A good DBaaS service offers a web-based service console to simplify database administration. Database administrators can perform many common database management and administration tasks, such as back ups, migrations, and user management from this console. A DBaaS platform also usually provides a REST API for interacting with the databases from applications, which provides a useful feature for application developers working with databases.

2.4 Automation

A DBaaS platform can automatically manage databases at every stage of their lifecycle, from initial configuration and installation to software patching, upgrading, failure detection, and disaster recovery process. The automation provided by DBaaS frees up a developer's time for building actual applications. DBaaS users still have an opportunity to interact with the database instances directly, performing manual backups or accessing specific database tables.

2.5 High Availability

A DBaaS platform can provide built-in High Availability (HA) features such as replication, topology awareness, and automatic failover for databases. It also can allow users to evenly distribute multiple instances of their database across availability zones of the cloud to improve fault tolerance and lower database read/write latency.

2.6 DBaaS as a Line of Business

DBaaS is a viable business investment in the era of on-demand software services. In this model, a DBaaS provider offers a range of database services via a web-based console and charges users for consumed resources and support subscriptions.

From the overview of the key DBaaS benefits, we can conclude that a robust DBaaS solution should be able to provide at least some of the following features:

- Easy-to-use web-based database management console, CLI, and Rest API to enable a self-service experience for DBaaS users. A self-service portal for your DBaaS platform should be easily tied into existing operational and ticketing systems, security mechanisms, etc.
- Multi-cloud Support. Cloud-agnostic experience has recently become a crucial requirement for DevOps teams. A DBaaS architect can leverage containerization technology and a container orchestration ecosystem (e.g., Kubernetes), which dramatically reduces the development costs that come with being cloud-agnostic. Platforms such as Kubernetes enable fast and zero-pain deployment of databases on many popular cloud providers (Google Cloud, AWS, Azure, etc.) and on-prem.
- Elastic compute resources (compute, memory, storage) that can be easily scaled on-demand.
- Backup functionality, including backup schedules and policies, database-consistent snapshots, etc.
- Efficient disaster recovery mechanisms, including automatic host replacement, cross-cluster migration, asynchronous replication, and more.
- A built-in monitoring interface with metrics dashboards and performance indicators.
- An event notification system integrated with third-party messaging systems and services (Slack, GitHub, etc.).
- Support for rolling updates and upgrades.
- Built-in security, including authentication, authorization, storage encryption.
- Query editor and REST API. Many DBaaS, such as Amazon Relational Database Service (RDS), allow users to run database queries via a web-based console, which enables fast experimentation and debugging.

3. BUILDING A DBAAS SOLUTION FOR KUBERNETES

3.1 How Containerization Enables DBaaS

Containerization is an emerging standard for managing distributed applications at scale. In a nutshell, a container is an isolated 'application' with its autonomous view of the filesystem, dedicated access to server resources and OS calls, and environment-agnostic configuration. The container architecture leverages OS-level virtualization in Linux based on cgroups and kernel namespaces. These technologies enable various useful application deployment patterns and features that can be leveraged by your DBaaS platform. Let's discuss some of them.

3.1.1 Portability and Extensibility

Containerization allows packaging the entire database stack, including executables, configuration, and network settings, in a single container that can be easily ported, extended, and launched across different environments that support a container runtime used to build the container. As a result, your team spends less time configuring servers with dependencies, environmental variables, and system libraries. All required dependencies and configurations can be packaged inside the container at the development/build time rather than the deployment time.

3.1.2 Open Source and Free

Most popular container runtimes—like Docker and OCI—are completely free for anyone to use. They leverage open-source Linux distributions and technologies such as cgroups, user namespaces, systemd, and other kernel libraries which enable container isolation and OS-level virtualization.

3.1.3 Containers Are Compact and Fast to Launch

Containers take up less space than virtual machines, which means you can place more of them on a single server, dramatically saving on the infrastructure. Containers are also very fast to launch in comparison to virtual machines.

3.1.4 Microservices-Ready

The self-contained nature of containers makes them a natural choice for a microservices architecture. Microservices are loosely coupled applications with isolated namespaces and resources and built-in environmental consistency. A single microservice can be deployed, managed, and updated independently of other components of the application. By their design, containers enable faster implementation of a microservices architecture.

3.2 Why Build DBaaS on Kubernetes?

Setting up, configuring, and managing databases involves a lot of repetitive tasks and manual activities that are prone to errors and require significant efforts of your infrastructure and DevOps team.

Common database management tasks may include:

- Initial infrastructure provisioning, database configuration, and deployment
- Setting up user management including authentication and ACL (Access Control Lists)
- Decommissioning and/or adding new database instances
- Backing up data and performing disaster recovery
- Making database updates and upgrades
- Migrating databases between environments
- Adding storage and resizing a filesystem as allocated volumes become full

The job of database administrators can be further complicated because modern applications often run in distributed database clusters with multiple database instances. Managing a database cluster introduces new challenges, such as ensuring HA, implementing auto-scaling, configuring networking, and enabling an efficient disaster recovery policy, such as cross-cluster migration and cross-AZ failover.

All these tasks can lead to errors and prolonged downtime if done manually and are difficult to model declaratively using generalized infrastructure automation tools. So, is there a solution?

A traditional approach is to create scripts that automate various reproducible database management tasks. However, these scripts need to be regularly updated and maintained, which means more developer hours. Also, this approach requires deep expertise in specific database systems and dealing with possible conflicts with new database versions. Such an approach becomes even more problematic if you manage many database systems of many different database vendors.

An alternative option is to use a mature deployment automation and orchestration platform that encompasses years of experience in managing applications at scale. Kubernetes was designed precisely for that. It automates routine tasks of managing distributed applications reliably by providing a set of orchestration services and abstractions.

Kubernetes can automate many common cluster administration tasks, such as rolling updates, upgrades, scaling up, down, and out, blue-green deployments, addition and removal of nodes, application health checks, etc. Your administration goals and the desired cluster state can be specified in a declarative way, leaving it up to Kubernetes Control Plane to make sure it is maintained.

When building a DBaaS platform on Kubernetes, you can also take advantage of containerization and a mature ecosystem of tools developed by the cloud-native community. For example, you can easily connect your DBaaS to monitoring and logging pipelines deployed on Kubernetes, configure database cluster networking using Kubernetes primitives such as Services, load balancers, and Ingress, and manage storage and data using container native storage.

In what follows, we summarize other benefits provided by Kubernetes for your future DBaaS platform.

3.2.1 Declarative Management

Most traditional database management systems are based on an imperative model. In this model, administrators interact with the databases using direct commands: ‘install database’, ‘launch instance’, ‘create snapshot.’ Such an approach lacks automation required by dynamic cluster environments where applications instances and nodes are launched and removed quite frequently.

In contrast, Kubernetes is based on the declarative model, in which database administrators simply tell how their database deployment should look at any point in time. For example, a database administrator could specify in a StatefulSet that the database should always run with three instances. The task of Kubernetes Control Plane is then to ensure that exactly three database replicas are running at any time and take action if one of them becomes unavailable. This feature is very useful in a distributed DBaaS environment where node downtimes are inevitable.

3.2.2 Support for Multiple Container Runtimes

Kubernetes implements Container Runtime Interface (CRI), a plugin-based interface that enables kubelet to use multiple container runtimes without the need to recompile. At this moment, Kubernetes supports such container runtimes as Docker, RKT, CRI-O, and more. Wide container support means you can package your database code in different container runtimes for greater portability and more use cases.

3.2.3 Scalability

Kubernetes makes it easy to scale database instances dynamically, based on a set of declarative rules. For example, Kubernetes ships with the Horizontal Pod Autoscaler (HPA), a tool that can automatically scale applications based on the observed traffic volume, CPU and RAM load, storage capacity, and any user-defined metrics. Similarly, one can leverage Kubernetes' cluster-wide auto-scaling to scale database clusters horizontally by dynamically adding new nodes.

3.2.4 Support for Stateful Workloads

Much work has been done recently to meet all major requirements of stateful applications in Kubernetes. StatefulSets, local persistent storage, Container Storage Interface (CSI), Headless services, and other Kubernetes API resources provide stateful applications what they need: stable network identifiers, persistent storage, and fault tolerance.

3.2.5 Cloud Provider Integration

Kubernetes provides good integrations with all major cloud providers, such as Microsoft Azure, AWS, Google Cloud, etc. You can easily deploy Kubernetes on these cloud providers with deployment automation tools, such as Kops, interact with the cloud API, and make use of cloud resources, such as storage, load balancers, reserved instances, and availability zones. This means you can offer databases in multiple cloud environments with mild changes in configuration and code. As a result, your developer team can have a multi-cloud experience for more efficient development and testing of databases and applications.

3.2.6 Support for Rolling Updates

Kubernetes Deployments and StatefulSets support rolling updates of stateful workloads. With Kubernetes rolling updates, you can smoothly upgrade to a new database version, one database instance at a time, ensuring that the database is not down, the data is safe, and users can continue reading and writing to the database during the update process. Also, via an integration with cloud native storage solutions like Portworx, Kubernetes makes it easy to implement other upgrade patterns, such as blue-green deployments—where you have two concurrent environments with new and old versions—and canary releases—where you provide a new upgraded version of the database only to some portion of end users. Kubernetes supports rollbacks to previous versions of your database, as well.

3.2.7 Ability to Turn your Database System into a Microservice

In Kubernetes, you can easily join individual instances of databases into a discoverable Service. A Kubernetes Service allows load balancing user requests between database instances so that users don't need to know the exact number and addresses of pods that run databases. This feature allows abstracting database access from the underlying deployment configuration, such as the number of database instances and nodes.

3.2.8 Built-in Resource Management and Namespacing

Kubernetes makes it easy to build a multi-tenant environment where resources are distributed across teams and applications. Using Kubernetes namespace quotas, a cluster administrator can allocate a fair share of resources to different developer and DevOps teams. These teams can then deploy as many database instances as allowed by their quota.

3.2.9 Power your DBaaS With Kubernetes Features for Free

Because Kubernetes is completely open-source and free, you can add new features provided by the platform as part of your DBaaS offering without writing a single line of code. You can easily leverage Kubernetes' built-in rolling updates, services, liveness and readiness probes, and monitoring as part of your DBaaS offering and focus more on database-specific features to further improve experience of DBaaS users.

4. MAKING DBAAS WORK ON KUBERNETES

Now that you understand the major benefits of containerization for your DBaaS solution, we are ready to discuss the basic tools and requirements that will help you build a robust DBaaS solution for Kubernetes.

4.1 Automate Database Lifecycle with DBaaS Operator

When building a DBaaS solution for Kubernetes, the first thing you need to consider is how to abstract the underlying primitives of the platform so that, for the DBaaS users, it looks like they are interacting directly with their DBaaS provider and their databases. DBaaS users are supposed to be experts in their specific databases (Cassandra, MySQL, Kafka), not necessarily skilled in Kubernetes Deployments, Pods, Services, and other native Kubernetes abstractions.

One way to achieve this is to use native Kubernetes primitives, such as StatefulSets, ConfigMaps, and Services pre-packaged in Helm charts for specific databases. These Helm charts can be automatically deployed with user configurations provided via the web console. When using native Kubernetes primitives such as Deployments and StatefulSets, databases deployed via the DBaaS will be managed directly by the default control loops implemented in the Kubernetes Control Plane.

This approach may suffice for a simple DBaaS system but is limited if you want to extend database deployments with custom behavior. To offer an authentic SaaS experience, a DBaaS solution should include a runtime that manages the full lifecycle of databases, from initial deployment to deletion. You may also need a way to define custom resources with behaviors and features specific to particular databases.

How can one implement such a runtime and custom resources on Kubernetes without reinventing the wheel? Luckily, Kubernetes is a very extensible platform that allows adding custom resources and controllers to the API server.

4.1.1 Custom Resources

A custom resource is an extension of the Kubernetes API with application-specific settings. For example, a custom resource for Elasticsearch may include additional settings for different types of Elasticsearch nodes: master, data, ingest, and client nodes. Similarly, it can expose general configurations for Elasticsearch cluster, such as JAVA settings and shard size, which are usually defined in ConfigMaps if deploying Elasticsearch as a standard Kubernetes StatefulSet.

4.1.2 Custom Controllers

Custom controllers implement behaviors for custom resources and allow them to be managed with the Kubernetes Declarative API. One can use custom controllers to encode domain-specific knowledge for databases into an extension of the Kubernetes API.

4.1.3 Operator Pattern

When combining custom Resources with custom Controllers, we can get a full-fledged runtime that encompasses a domain-specific operational knowledge for a specific database. This leads to the Operator pattern proposed by CoreOS to facilitate an extension of Kubernetes with custom resources.

In a nutshell, a Kubernetes Operator provides custom resources that expose application-specific knowledge and control loops integrated with the Kubernetes API and Control Plane to manage these resources. As defined in the CoreOS blog:

> *“An Operator is an application-specific controller that extends the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. It builds upon the basic Kubernetes resource and controller concepts but includes domain or application-specific knowledge to automate common tasks.”*

Encoding an application-specific knowledge is very important for stateful applications like databases. These systems require an application-domain knowledge to correctly scale, upgrade, and reconfigure while protecting against data loss or unavailability.

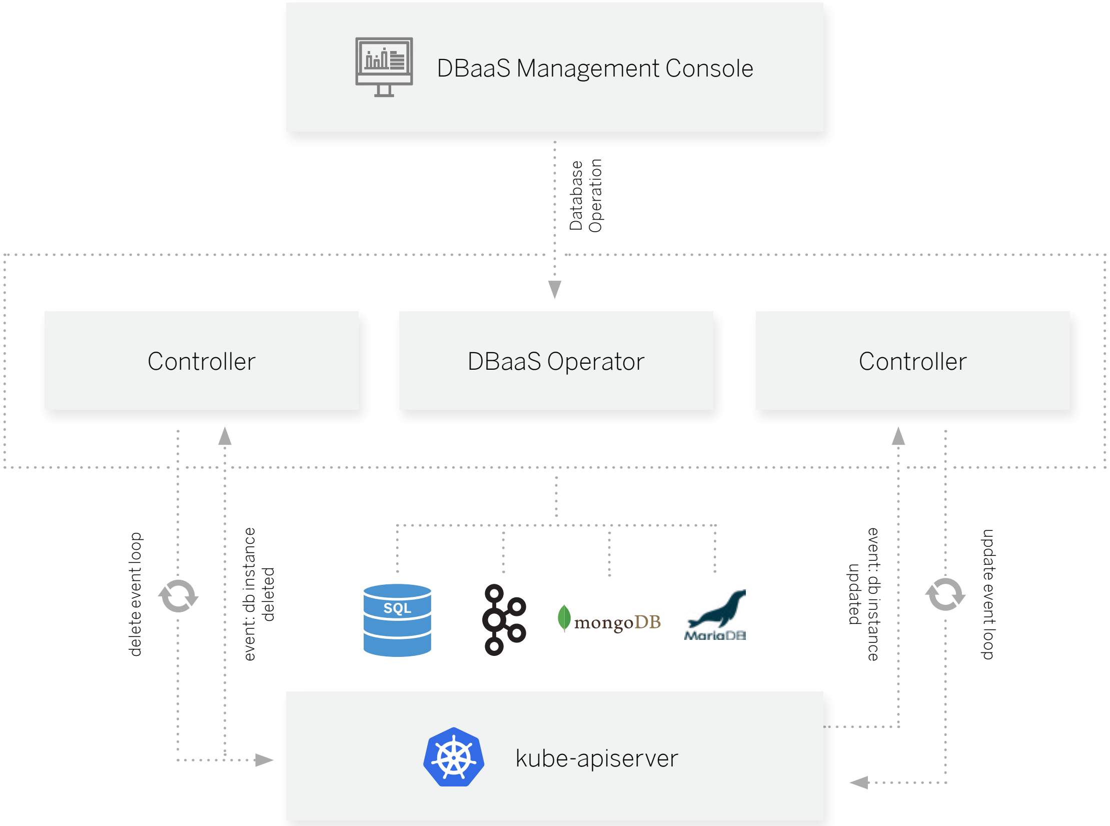


Image: Kubernetes Operator for DBaaS

For example, there is no straightforward way in Kubernetes to make an application-consistent snapshot of the MongoDB database. To make such a snapshot, we should use certain rules to ensure that the database state reflected in the snapshot is consistent. For example, when making a snapshot of a MongoDB database, we should use a pre-snapshot rule forcing the mongod to flush all pending write operations to disk and lock the entire mongod instance to prevent additional writes until the snapshot is done. A similar behavior should be implemented for other databases, as well.

Kubernetes' custom resources managed by the Operator can help us implement such a custom behavior. With Operators, you can extend Kubernetes with new primitives and abstractions that may be managed much like you manage regular Kubernetes resources, such as Deployments and StatefulSets. All this comes with the ability to add application-specific knowledge to the controller without introducing additional complexity to the Kubernetes code.

4.2 Kubernetes Operator Example

To get an idea of how Operators might be useful for the design of your DBaaS solution, let's take a look at one hypothetical example.

Let's say your goal is to build a simple DBaaS solution that enables users to quickly deploy Kafka on Kubernetes and you have decided to leverage the Kubernetes Operator pattern for this.

First, your Kafka Operator needs to create a set of Kafka custom resources to manage Kafka deployments. Kubernetes' Custom Resource Definition (CRD) is a Kubernetes primitive that lets you define such custom resources. It can contain an arbitrary number of custom fields and can be managed as a full-fledged Kubernetes API resource with standard Kubernetes command tools (kubectl).

In a custom resource definition, a DBaaS developer can encompass domain-specific configurations for Kafka, such as the number of brokers in a cluster, a minimum number of in-sync replicas, partition, broker, and topics settings, etc. One can still use ConfigMaps along with custom resources to provide all possible Kafka configuration settings to users.

Secondly, your Kafka Operator has to provide a set of controllers to maintain the desired state of the Kafka deployment. A controller code interacts with the Kubernetes Control Plane to check the state of a custom resource (Kafka deployment) and take action to match the desired state.

For example, one can imagine a custom controller that watches broker metrics for the production/consumption data rate in the Kafka cluster and scales the cluster horizontally if the data load increases. Or, when the user adds a new Kafka broker, a custom controller may set up a new PersistentVolumeClaim to provide durable storage and create a Job to handle initial configuration of the broker. Also, when the broker operates, the controller can run periodic checks, such as liveness and readiness probes, to make sure that everything works as expected. Controllers can be configured to watch for other important Kafka cluster parameters, such as leader imbalances, not responding brokers, and deterioration of network latencies and manage monitoring, logging, auditing, alerting sub-systems integrated with Kafka Finally, if a broker is deleted, the Operators may take a snapshot of a broker and clean up the cluster, making sure that the StatefulSet and Volumes are also deleted.

When speaking about database systems in general, Operators should be able to perform common database management tasks, such as regular database backups, database upgrades, disaster recovery, data transfer, etc. The scope of possible features supported by the Operator may be as broad as you choose. What an Operator can do is limited only by Kubernetes' built-in control functionality, which is, however, very comprehensive.

As these examples demonstrate, Kubernetes Operators can provide robust automation for DBaaS on Kubernetes. In fact, it can substantially augment a human database administrator, dramatically alleviating database administration burden. The human database administrator can use capacities of the controller to manage databases and be sure that the actual state of database clusters always matches the desired state because the Operator is tightly integrated with the Kubernetes Controllers sub-system.

It's now easy to see how the Kubernetes Operator pattern facilitates the development of a DBaaS platform on Kubernetes. In particular, the Operator pattern can enable the following functionality for your DBaaS system:

- **Smooth installation.** Operators can install databases using Kubernetes templates with the pre-configured settings that follow best production database practices.
- **Flexible configuration.** Users can specify the size of the database cluster, storage, node affinity rules, snapshot schedule, backup retention policy, and any other parameters you provide.
- **Database lifecycle management.** An Operator can manage the full lifecycle of database instances using domain-specific knowledge. This ensures that controller logic is compatible with the database requirements.
- **Kubernetes integrations.** Operators can be easily integrated with Kubernetes' built-in monitoring, node health assessment, and auditing systems to provide more features for DBaaS consumers.
- **Providing external access to the database cluster.** Operators can leverage Kubernetes Services, Ingress, and load balancers to build fine-grained network policies for the database cluster.
- **Database-granular resource allocation.** Your primary and replica databases can have different resource profiles within a cluster.

4.3 Operator Ecosystem

At this moment, there are many open-source Operators for specific databases and data processing systems, such as MongoDB, Cassandra, Kafka, and many others. You can leverage concepts and architectural solutions built into these operators in your own DBaaS solution. If you want to develop an Operator from scratch, there are also several tools that facilitate building a robust Kubernetes Operator:

- The Operator Framework. This is an open-source toolkit for creating and managing Kubernetes Operators. Key components of the framework are Operator SDK for building Operators, Operator Lifecycle Manager, and Operator Metering tool for collecting metrics.
- KUDO (Kubernetes Universal Declarative Operator). KUDO is an operator development toolkit and runtime that makes writing operators productive and simple.

- Kubebuilder. Kubebuilder is a framework for building Kubernetes APIs using custom resources.
- Metacontroller. This is an add-on that allows creating custom controllers for Kubernetes. One can use Metacontroller along with WebHooks you implement yourself.

If you are not sure which Operator development tool to select, you should first check the Operator Framework developed by CoreOs. This company (now part of the Red Hat) is an original creator of the Operator pattern.

5. STORAGE CONSIDERATIONS FOR DBAAS ON KUBERNETES

5.1 Basic Storage Requirements for DBaaS on Kubernetes

Storage is a central component of any database system. Although Kubernetes is now a mature container orchestration platform, it was originally designed for stateless microservices that do not require stable and persistent storage. Today, much progress has been made to close the gap between support for stateless and stateful apps in Kubernetes. At this moment, Kubernetes is a perfect choice for any stateful application; however, additional prerequisites for storage should be put in place. In this section, we discuss key requirements for storage on Kubernetes and how to meet them using the Portworx Kubernetes Storage platform. First, let's discuss general storage requirements for databases in a distributed environment.

- Databases require persistent and durable network identifiers and durable persistent storage. They cannot be managed as identity-less stateless services moved across nodes without notice.
- Databases failover in a distributed environment requires strong data locality guarantees. When a database instance moves to another node in the cluster, it should have instant access to the local copy of its data.
- Database snapshots and backups need to be consistent with the application state, such as pending write operations. This requires a set of application and operational rules for database snapshots.
- Database storage should be highly available. This requires synchronous replication of data and homogenous distribution of the replicas across the cluster.
- Different classes of storage for different database types should be available on demand. For example, Elasticsearch data nodes may require high IOPS SSD drives whereas master and ingest nodes may do with standard storage like HDD. To provide different classes of storage for your database instances, we need a software-defined storage aggregator stretching the Kubernetes cluster.

All these requirements for your DBaaS solution can be efficiently met by the Portworx Enterprise container storage platform.

5.2 The Portworx Kubernetes Storage Platform

Portworx is a software-defined storage platform that enables storage pool aggregation, container-granular storage provisioning, HA, data security, backup, disaster recovery, and automated capacity management for containerized applications in a distributed computing environment.

Portworx is a cloud-native solution integrated with several major container orchestration platforms, including Kubernetes and Mesos. Based upon cloud-native principles, Portworx is a great choice for storage orchestration of DBaaS running in Kubernetes.

At the lowest level, Portworx 'virtualizes' different storage types available in the cluster and aggregates them into a unified storage layer to manage them and provide storage to containerized applications on-demand. Portworx storage pools can aggregate various storage types, including SSD, SANs, HDD, cloud block storage, etc. With Portworx, Kubernetes users can easily allocate storage from this pool to create Persistent Volumes for applications in the cluster.

Thus, one can think of Portworx's storage aggregation layer as a kind of Storage-as-a-Service (SaaS) model where storage can be allocated across different nodes and applications on demand. Such a model provides the way to abstract cluster storage resources from the underlying storage media and make them available to nodes and applications in the Kubernetes cluster.

Let's discuss several other features of Portworx that allow your DBaaS platform to meet data consistency and HA guarantees discussed above.

5.2.1 Replication

Portworx can ensure that storage provided to databases via DBaaS is highly available using built-in replication. When creating `PersistentVolumes` with Portworx `StorageClass`, users can specify the number of replicas for the volume to have in the cluster. Portworx will create replicas according to this replication factor and evenly distribute them across the Kubernetes cluster. This is allowed by built-in topology awareness that helps auto-detect racks, availability zones, and regions and spread replicas across them.

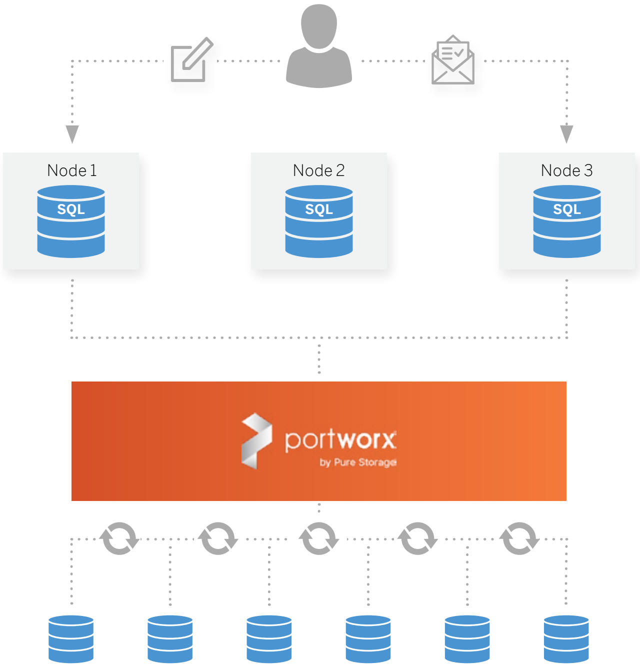


Image: Portworx Synchronous Replication

Portworx controllers ensure that these replicas are synchronized with each other. Whenever there is a new write to one replica, it is automatically synchronized with all other replicas. This secures data consistency among all database instances. By the same token, Portworx ensures parallelism of read operations. If the replication factor is greater than one, different data blocks will be read from different servers. Multi-sourcing the read operations across nodes ensures higher performance for databases.

Also, volume replicas created by Portworx can be easily accessed from any node where Portworx runs, which makes it easy to automatically transfer replicas to new nodes once they are added to the cluster. Portworx unified storage layer can also automatically create disks based on input disk templates whenever a new node spins up. This allows for automatic provisioning of storage when auto-scaling functionality is embedded.

Portworx storage replication ensures fast and efficient failover of database instances in case of a node outage. Let's say a DBaaS user deployed an instance of MongoDB database via the DBaaS console and the node where the instance runs suddenly became unavailable.

Once the Kubernetes API server detects the node failure, the DBaaS control loops are notified about this and the Operator uses its rules to reschedule a database replica to a new node. The role of the Portworx platform is to ensure that this node has a data replica already sitting on it. Because the Portworx's storage scheduler is integrated with a Kubernetes scheduler, it can help your DBaaS Operator select a node with a copy of data.

Overall, this approach significantly reduces the failover time. As a result, databases deployed with DBaaS continue to run at high performance, I/O, and throughput during the rebuild process. Also, thanks to replication, database users can run fewer database instances with the same level of reliability. This substantially reduces compute costs. Running three instances instead of five is a 40% cost savings for the same level of reliability.

5.2.2 Storage Class of Service

Most databases require fast storage with high IOPS and throughput. At the same time, some data solutions—like Elasticsearch—run in clusters where different nodes have different roles and, hence, different storage requirements. Portworx provides the possibility to use different storage classes of service (CoS) depending on the requirements of your data systems.

When creating a unified storage layer, Portworx automatically labels different storage types available in the cluster into high, medium, and low classes of service (CoS) and pools them together.

The high CoS is optimized for IOPS; medium is optimized for throughput, and low CoS provides cheaper storage for low priority data. In the cloud, this means that each VM could have three different cloud block devices that Portworx tiers across. On-prem, it could mean that each bare metal server is loaded with different storage drive types.

Thereafter, you can allocate Portworx volumes with a specific I/O priority or CoS for your database instances. For example, using this feature, you can easily run Elasticsearch data nodes on dedicated SSD volumes with high IOPS and throughput profiles and allocate medium-class storage to master, coordinating, and ingest nodes. You can also control the storage type of your indexes when they transit to a 'colder stage'.

This behavior can be achieved automatically with Portworx volume placement strategies. These strategies allow for establishing the gravity/anti-gravity (affinity or anti-affinity) between specific volumes or replicas and different CoS and storage infrastructures. For example, you can use various affinity rules to spread Elasticsearch volume replicas across different storage pools based on CoS (High, Medium) or media type (SSD, HDD). You may choose to put replicas of data nodes to SSD pools and replicas of other node types on pools with fewer requirements. Similarly, you can select failure domains (availability zones or regions) for placing volumes and their replicas.

In summary, the automatic pooling of volumes based on their CoS is very convenient and greatly reduces costs. You don't have to manually assign volumes to specific categories. Portworx takes care of matching a needed CoS to underlying storage devices in your cluster. Moreover, containers operating at different classes of service can co-exist in the same node/cluster.

5.2.3 Storage Policies

Portworx allows creating storage policies that describe default parameters of storage. For example, a storage policy can define default replication level, snapshot schedule, encryption type, volume stickiness, and other parameters. Having different storage policies for different volume types provides much-needed flexibility and automation for your DBaaS platform.

6. DBAAS SECURITY LAYER

Security is a top concern for databases, especially in a distributed compute environment with complex networking structures and multiple potential attack vectors. Databases are common targets of attacks because they contain valuable user data. Although threat protection is a responsibility of end users, DBaaS providers can include tools to make it easier to implement the most basic protection.

Most databases provide basic functionality such as authentication and authorization that works at the database instance or cluster level. Your DBaaS offering should provide additional security features for database clusters and instances, including the following:

- **Authentication and Authorization.** A DBaaS platform should be integrated with cloud access management roles and policies.
- **Data encryption.** It's useful to provide real-time encryption and decryption of the database, backups, and transaction log files at rest. One option is to use the transparent data encryption to encrypt the storage of an entire database with a symmetric key.
- **Network isolation.** A DBaaS solution can leverage the underlying cloud provider network isolation tools, such as VPC in AWS and firewall rules, to control access to database clusters and instances deployed via a DBaaS.
- **Auditing.** Database auditing involves tracking events and activities in the databases. Auditing is very important for maintaining regulatory compliance, understanding database activity, and finding anomalies that point to security violations.
- **A security analytics solution.** Such a solution can provide security alerts and recommendations on how to investigate and mitigate threats. Using such a system, database administrators can get alerts about suspicious activities, injection attacks, and other anomalous access and query patterns.
- **Vulnerability assessment service.** Such a service scans databases to find deviations from best practices, misconfigurations, wrong permissions, and unprotected sensitive data.

6.1 DBaaS Security Considerations for Kubernetes

When building a DBaaS on Kubernetes, there are several additional layers of security to be addressed. These layers are physical storage security, Kubernetes cluster security, and storage access security.

The first concern is, of course, storage security. Data sitting in database volumes is not encrypted by default. Ideally, organizations should protect at the application level but also secure the data layer along with it for added security.

Thus, when running databases in a distributed computer environment like Kubernetes, it's reasonable to encrypt your storage. You can do this for all storage resources in your cluster pool using Portworx.

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.

The second important concern is the 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 a DBaaS 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 an additional authentication and authorization protection for the storage layer. It will allow your DBaaS 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.

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

In summary, adding Portworx to your DBaaS platform 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 databases deployed via DBaaS.

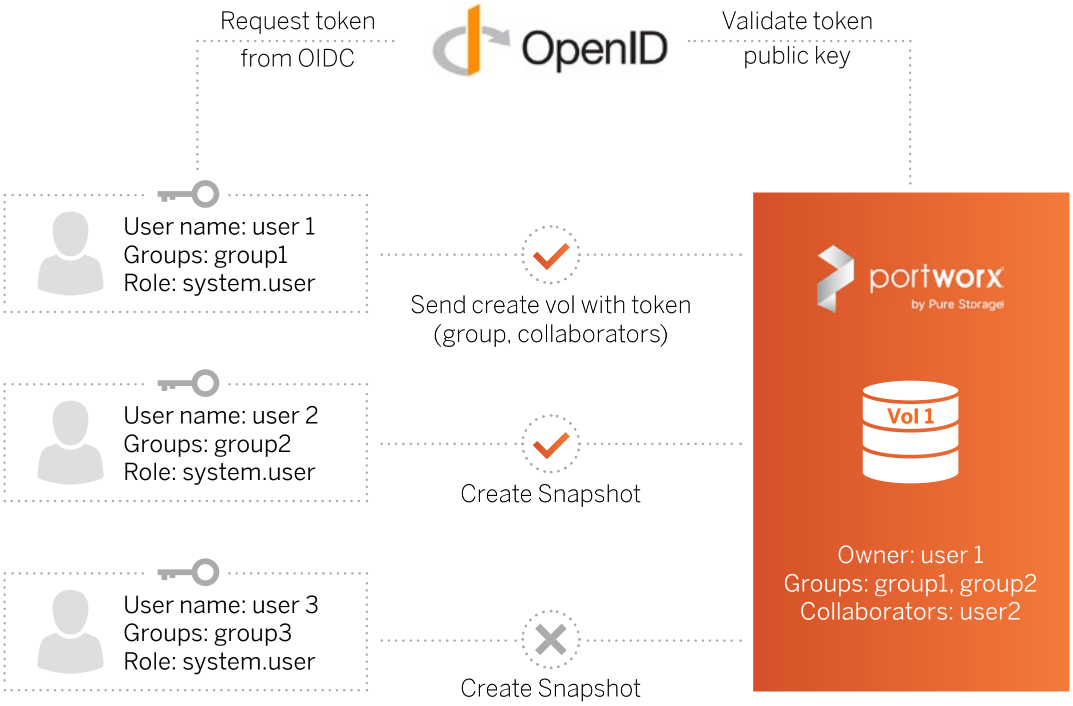


Image: Portworx Security Model

7. DBAAS BACKUP WITH PX-BACKUP

Most modern databases ship with built-in backup and snapshot tools. These tools, however, can only backup data that is not sufficient for Kubernetes where the application state consists of many resources, such as app configuration, persistent volumes, Kubernetes deployment objects (StatefulSets, Deployments), metadata such as namespaces, resource request, and limits, etc. Therefore, it's advisable to have a DBaaS backup tool customized specifically for Kubernetes.

PX-Backup is a toolkit that satisfies all major requirements of the efficient database backup policy on Kubernetes. It enables a *full* backup and restore of Kubernetes applications—rather than merely data—with a single click. Along with the application data, PX-Backup can backup configuration, Kubernetes resources, and cluster metadata. Also, with PX-Backup it's possible to create snapshots of multiple groups of Pods or an entire Kubernetes namespace. PX-Backup maintains the metadata about the namespace from where the backup was taken, so it's easy to restore multiple apps to the same namespace.

For DBaaS users managing multiple Kubernetes clusters, PX-Backup provides multi-cluster and multi-cloud backup support for each individual database running on your service. It's very convenient when you run several Kubernetes clusters on different clouds or on premises. Having a centralized interface for managing backups for these clusters provides visibility into the source environments and allows managing the full lifecycle of your backups.

PX-Backup is seamlessly integrated with cloud drives from GCP, AWS, and Azure, which allows for 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, filtering backups on cluster(s), namespace(s), and labels. All these features allow automating backup management for your DBaaS offering on Kubernetes.

7.1 Other Backup and Snapshot Tools

Along with PX-Backup, Portworx provides several other tools for creating snapshots of volumes and applications on Kubernetes. These are volume snapshots and 3DSnaps.

Volume snapshots allow you to create copies of `PersistentVolumes` used in database pods. With Portworx, you can create on-demand (one-time) and scheduled volume snapshots.

With on-demand volume snapshotting, users can take a one-time snapshot of the persistent volume. This snapshot can be then referenced in the `PersistentVolumeClaims` if a database data needs to be restored.

Also, for each volume in your DBaaS-deployed database cluster, you can specify a snapshotting schedule. Portworx will periodically take snapshots of database volumes and store the most recent ones according to the retention policy.

One can store snapshots of DaaS volumes locally or in the cloud using STORK (Storage Orchestration Runtime for Kubernetes). Cloud snapshots can be automatically uploaded to the configured S3-compliant endpoint (e.g., AWS S3).

Taking snapshots of application data at the storage level is a useful feature, but you also need to ensure that snapshots are consistent with the database state. In Portworx, you can create such application-consistent snapshots using 3DSnaps. For each 3DSnap, users can specify pre- and post-snapshot rules that are run on the application pods using the volumes. This feature allows users to pause the database writes before the snapshot is taken and resume I/O after the process is over.

8. DBAAS CLUSTER MIGRATION WITH PORTWORX

Portworx ships with the PX-Migrate, a tool that lets users migrate data, application configuration, and Kubernetes objects (ConfigMaps, Secrets, etc.) across Kubernetes clusters seamlessly and with almost no effort.

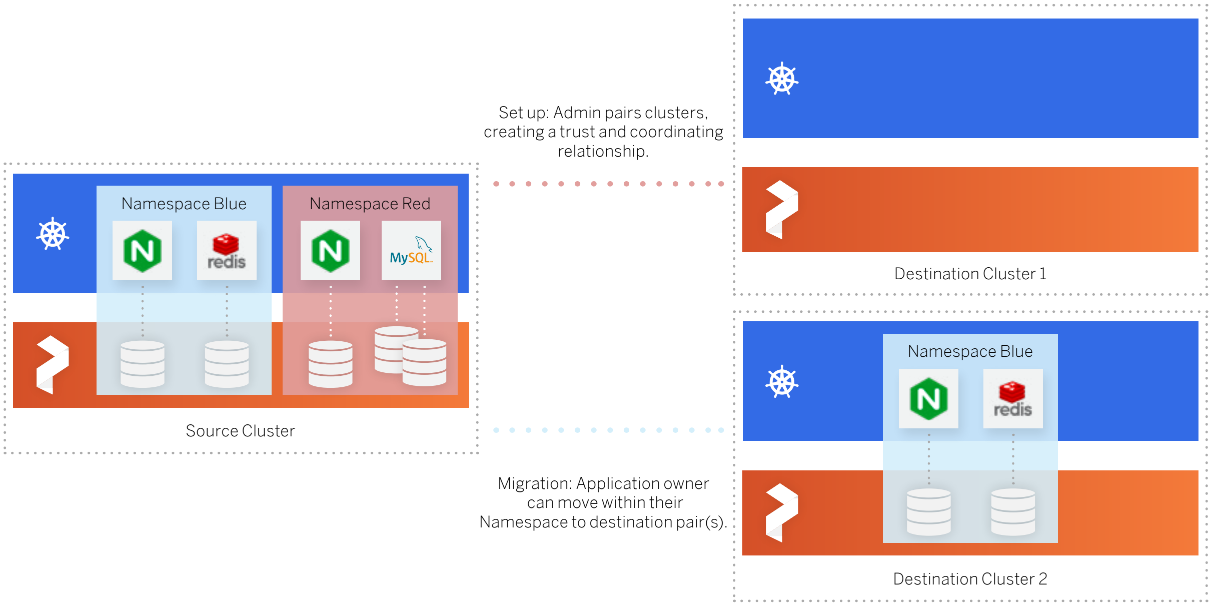

Set up: Admin pairs clusters, creating a trust and coordinating relationship.
Namespace Blue
Namespace Red
redis
MySQL
Source Cluster
Destination Cluster 1
Namespace Blue
redis
Destination Cluster 2
Migration: Application owner can move within their Namespace to destination pair(s).

Image: Cluster migration with PX-Migrate

There are many scenarios where your DBaaS can benefit from the cluster migration with Portworx:

- Free storage on critical clusters by moving lower priority data to remote clusters.
- Use blue-green deployments to test new versions of your DBaaS, databases, Kubernetes, and Portworx with all configurations 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.

To enable the PX-Migrate tool, two clusters should be first paired using a cluster pair key. After this is done, Portworx can start migrating the data using in-flight compression for higher data transfer speed.

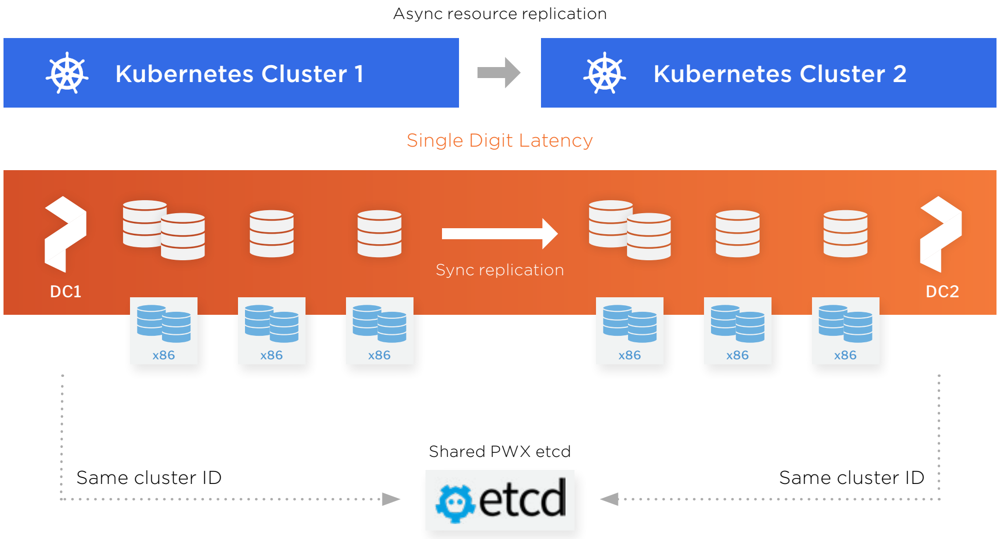
Also, Portworx ships with the built-in migration monitoring tool accessible from the `storkctl` CLI. Using the tool, you can observe the state of migration, synchronization status, history of scheduled migrations, and other useful parameters that help you understand the state of your remote and source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

In sum, a cluster migration feature makes it easy to implement failover and failback of databases 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.

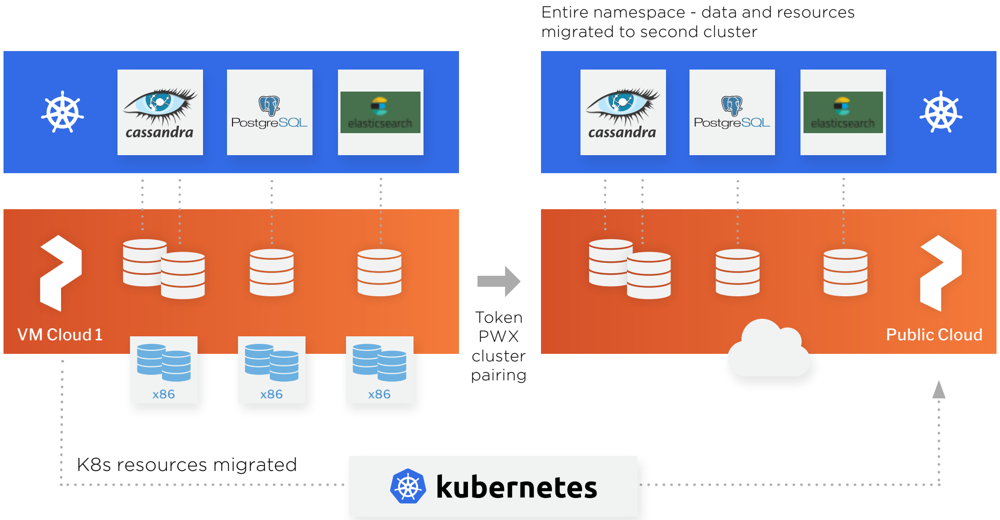
9. DBAAS DISASTER RECOVERY WITH PORTWORX

Portworx replicated volumes provide a good starting point for disaster recovery; however, your DBaaS solution should also consider scenarios when the entire cluster or data center becomes unavailable. This scenario can be addressed with the cross datacenter migration and cross datacenter synchronization that allow creating a hot standby cluster.

Portworx supports synchronous and asynchronous modes of cluster disaster recovery. Synchronous DR works when Portworx is installed as a single stretch cluster across multiple Kubernetes clusters spanning a metropolitan area network with a recommended maximum roundtrip latency of 10ms.



In asynchronous mode, Portworx asynchronously replicates application, configuration, and data to remote clusters located beyond the scope of the metropolitan area network. Incremental changes in Kubernetes applications and Portworx data are continuously sent to the standby cluster.



Users can also schedule synchronization between destination and source clusters, if needed, using migration schedule policies. They allow you to set 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.

10. MONITORING

Many mature DBaaS solutions have built-in performance assessment and monitoring features combined with the visualization engine to produce visual reports, dashboards, and other insight-rich materials.

When building your DBaaS solution for Kubernetes, you can take advantage of existing cloud-native monitoring and visualization tools tightly integrated with Kubernetes. Some of the popular tools are Prometheus for monitoring and Grafana visualization.

For example, you can configure Prometheus to scrape metrics from your database containers and export it to the visualization dashboard in your DBaaS web console. You can use database built-in metrics, such as compute/memory/storage capacity utilization, I/O activity, and instance connections, traffic, etc. along with Kubernetes-wide cluster metrics internally.

10.1 DBaaS Capacity Management with PX-Autopilot

With PX-Autopilot you can turn your monitoring pipeline into a source of action and automation in your database clusters, allowing you to easily manage multiple DBaaS instances.

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:

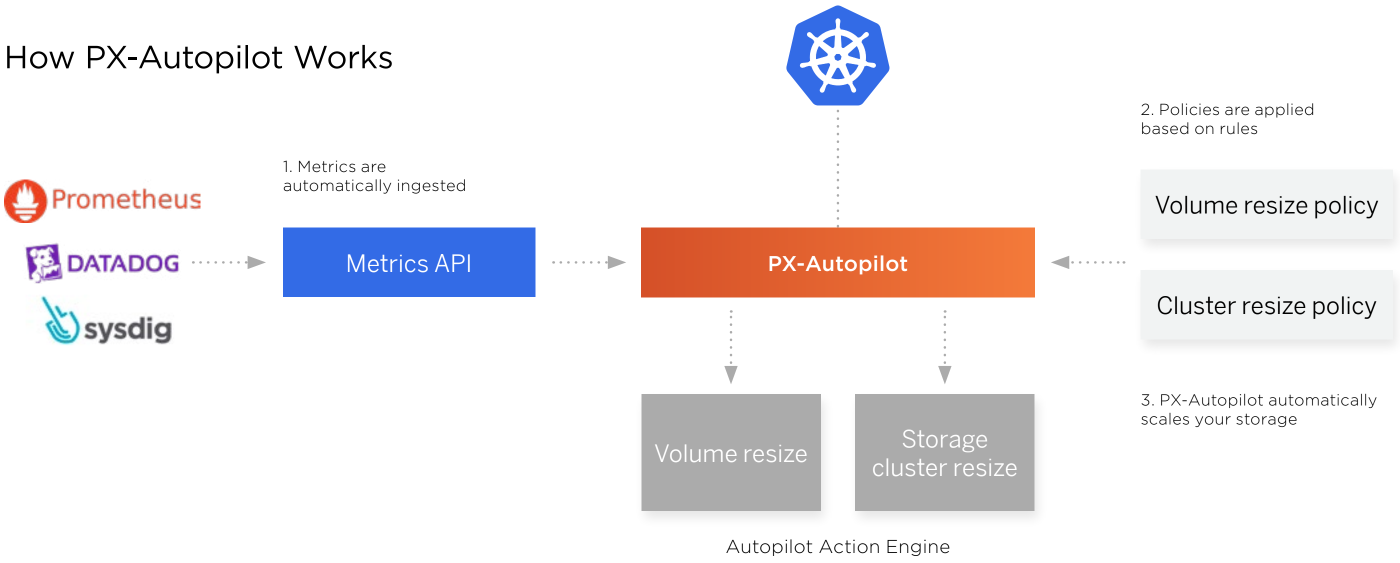
First, PX-Autopilot automatically consumes metrics from the tools you are already using to monitor your Kubernetes environment like Prometheus—for example, metrics like storage utilization at the individual volume, or even cluster level.

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 Portworx 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, 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.

How PX-Autopilot Works



11. CONCLUSION

As this paper demonstrates, Kubernetes provides a lot of useful features that make it a good choice for your DBaaS platform. Your DBaaS offering can leverage Kubernetes' built-in declarative API, stateful resources, and automation tools to enable an authentic DBaaS experience for your users. All these features can be extended with custom domain-specific functionality for your databases using the Operator pattern.

Combining Kubernetes-native features with custom controllers implemented via the Kubernetes Operators will let you create a cloud-agnostic, fully automated, extendable, and feature-rich DBaaS solution.

At the same time, to run DBaaS efficiently on Kubernetes you should also address such aspects as storage HA, security, backup, disaster recovery, capacity management, etc. As we've found in this paper, Portworx can provide many useful features to implement a container-granular, secure, and highly-available storage for your DBaaS.

To sum it up, key benefits provided by Portworx for DBaaS on Kubernetes include the following:

- Efficient database volumes replication across Kubernetes cluster to enable storage HA and fault tolerance
- Data-aware scheduling through deep integration with the Kubernetes scheduler to enforce right database placement decisions and enable data locality
- Extending database built-in security with storage-level authentication, authorization, and ownership
- Application-consistent database backups and snapshots
- Asynchronous and synchronous cluster recovery and cluster migration

All these features enable seamless and smooth deployment of DBaaS on Kubernetes that meets all data persistence and performance guarantees defined by database systems.

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