Portworx with Enterprise DB

Portworx with EDB delivers a resilient, high-performance platform for running PostgreSQL on Kubernetes. EDB manages database operations and distributed availability, while Portworx provides enterprise storage, data protection, and automated day-two operations.

Author: Ryan Wallner (Portworx)
Contributors: Temi Ogunrekun (Portworx), Greg Shultz (Portworx), José Moré (EDB), Gabriele Bartolini (EBD)

Table of Contents

Table of Figures

Table of Tables

Executive Summary

Modern database platforms require more than persistent volumes and basic CSI integration. They require an architecture that can deliver predictable performance, high availability, operational consistency, and data protection across Kubernetes environments. Portworx with EDB Postgres® AI addresses those requirements by combining enterprise-grade container-native storage with an operator-driven PostgreSQL platform that supports resilient database deployment, distributed architectures, and day-2 operational automation. When designed correctly, this approach helps organizations standardize database services, improve recovery outcomes, reduce manual storage and failover complexity, and give platform teams a more consistent way to run stateful workloads at scale.

About This Document

This document provides technical guidance for designing and deploying EDB with Portworx on Kubernetes. It is intended to help architects, platform engineers, and database teams understand the major design decisions that influence a successful deployment, including storage topology, cluster sizing, StorageClass design, security, data protection, networking, and the operational model for EDB. The goal is to present a practical white paper that can serve as foundational information while designing your Portworx and EDB architecture and a set of implementation considerations that can be used to plan a production-ready platform for stateful database workloads running on Kubernetes.

High-Level Value Proposition

Benefits of Portworx

Portworx cloud-native storage offers a transformative solution for managing stateful applications in Kubernetes environments, far surpassing traditional hardware storage solutions and their basic CSI (container storage interface) drivers. While CSI connectors often fall short in handling creates, updates, and deletes (CRUD) at the scale provided by Kubernetes and containers, Portworx excels in delivering a unified layer of enterprise-grade features that enhance both operational efficiency and platform engineering productivity for containerized environments whether you are on premises, at the edge or in the cloud.

Key benefits of Portworx cloud-native storage include:

- **Accelerated time to revenue:** By streamlining storage management and reducing complexities, Portworx helps organizations achieve faster deployment and time to market.
- **Data resiliency:** Portworx ensures high availability and disaster recovery with advanced data replication and backup capabilities, protecting against data loss and downtime.
- **Enterprise scalability:** Designed to scale with your needs, Portworx supports seamless growth, whether you're operating in a single data center or across multiple clouds.
- **Self-Service storage access:** Developers gain the flexibility to provision and manage storage independently through storage classes, empowering them to innovate without waiting for IT intervention.
- **Integrated storage management:** Portworx offers comprehensive management features, including rule-based automation, thin provisioning, and support for multi-cloud, hybrid-cloud, and on-premises environments agnostic to the hardware providers.
- **Boosted platform engineering productivity:** By providing a unified and efficient storage solution, Portworx enhances the productivity of platform engineers, allowing them to focus on building and maintaining robust applications in multiple environments.

With Portworx, organizations can achieve unmatched data agility and reliability, ensuring their Kubernetes storage and databases are always performant and available, regardless of the underlying infrastructure.

High-Level Design (Single Region)

The diagram below shows a high-level design of the Portworx architecture deployed with EDB within a single region, cluster or datacenter.

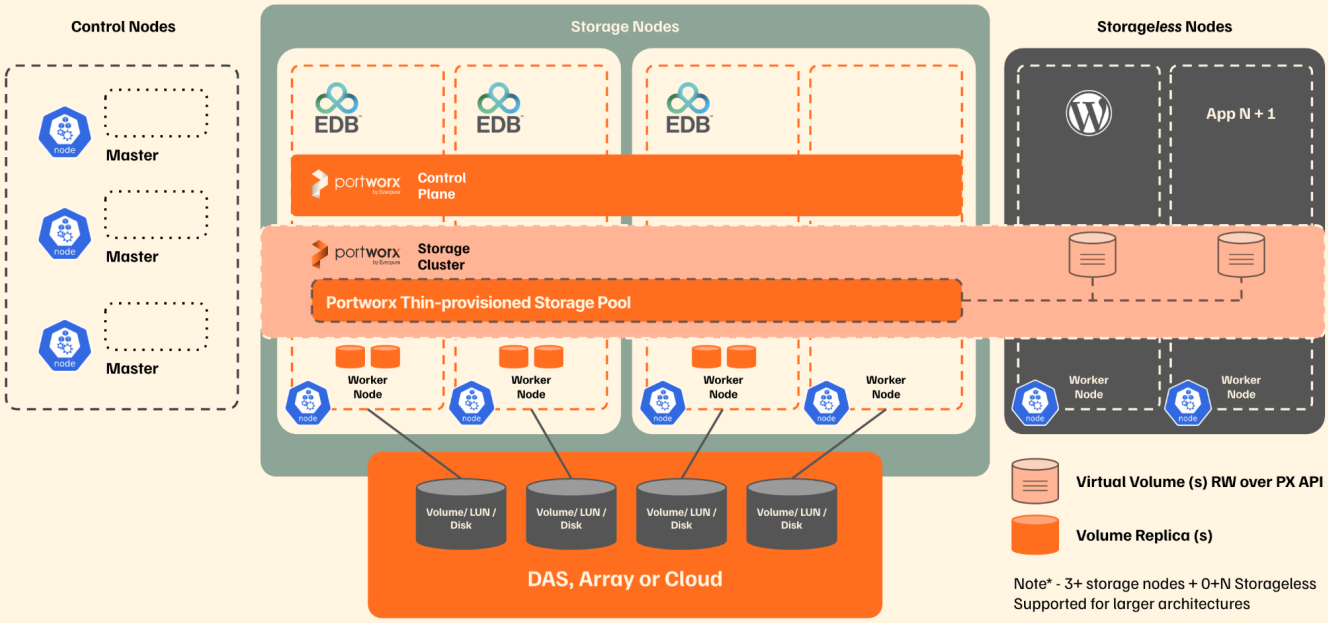

Figure 1. Single Cluster Portworx and EDB Design

This design includes a minimum of 3 storage nodes participating in a hyperconverged Kubernetes + Portworx Enterprise Cluster.

Additional Worker nodes may be part of the storage cluster or run as stateless for applications that do not need state. Portworx is agnostic to the type of backend storage which could be Direct Attached, Hardware Array or Cloud based.

For the purposes of this paper, it provides a high-level structure of how EDB and Portworx exist in the architecture and provides flexibility in configuration which will be described in detail below.

High-Level Design (Multi Region)

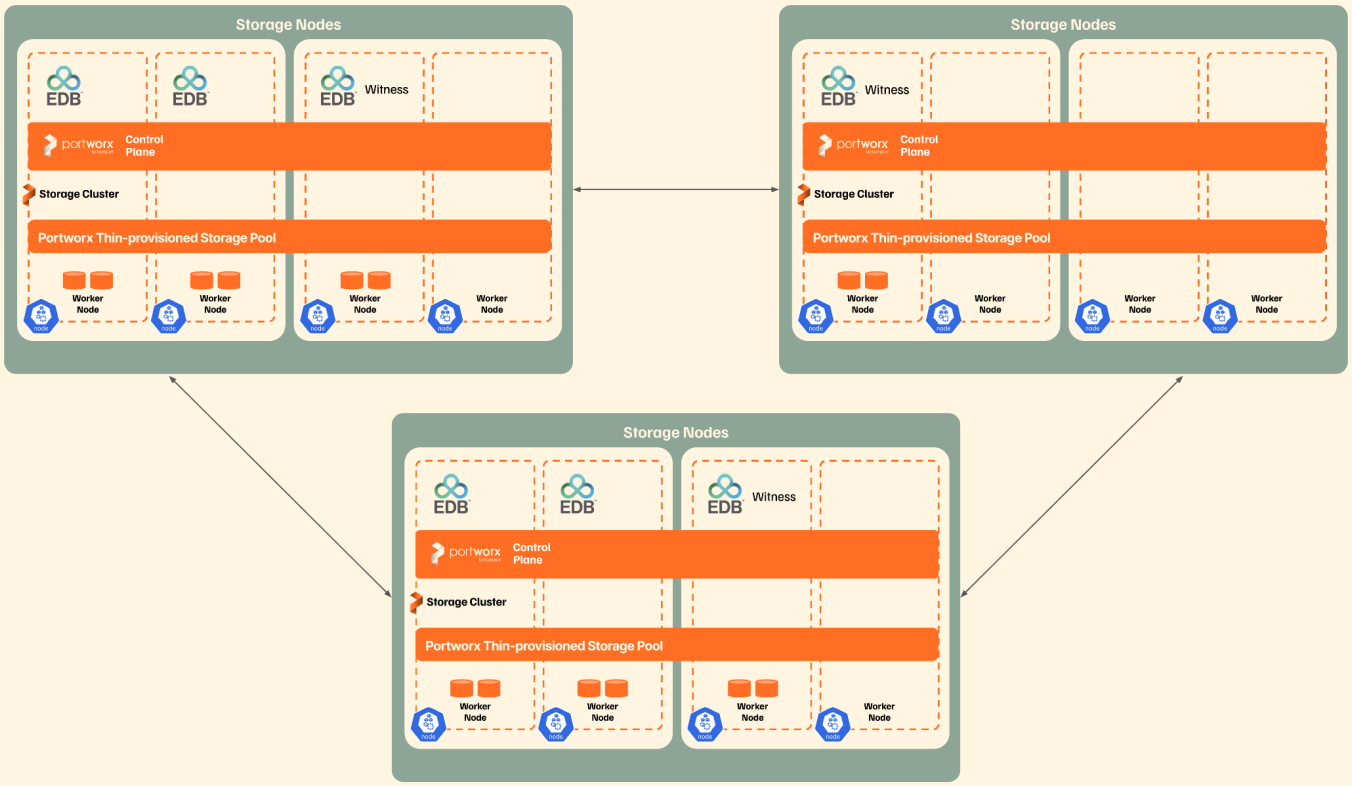

Figure 2. Multi-Region Portworx and EDB Design

EDB and Portworx can also be configured to run across clusters at the same time Portworx Enterprise provides multi-cluster features for the management of data across these boundaries with Replication, DR, Backups, Migrations and more.

These features will be described in detail further down in this paper and this architecture diagram is meant to provide a high-level overview of such a configuration for a high-level reference.

Portworx Cluster Configuration

Portworx requires a minimum of three storage nodes in the cluster to create a quorum and to prevent a split-brain scenario that can occur with two nodes.

***Note**: The exception is when using the supported Two-Node Arbiter architecture from Red Hat OpenShift which uses a witness node as the quorum tie-breaker.*

Three nodes are also needed to provide three redundant copies of data for persistent volumes. However, for a production environment, this reference architecture recommends starting with six storage nodes. Although a Portworx cluster with three storage nodes may work well in some environments, there are significant advantages to deploying with six storage nodes initially.

The advantages to using at least six nodes include:

- **Cluster capacity:**
  - In a cluster with three storage nodes, losing one node results in losing one-third of its capacity, increasing the load on the remaining two nodes until the lost node recovers, which could be days in the case of hardware failures.
  - In a six-node cluster, losing one node affects only one-sixth of its capacity, allowing the remaining five nodes to distribute the load more evenly, especially across fault domains like a rack.

- **I/O load distribution:**
  - A three-node cluster may experience more frequent I/O latencies during peak times.
  - A six-node cluster can distribute I/O requests more effectively, reducing the risk of I/O latencies.

Disaggregated Mode with EDB Postgres Distributed

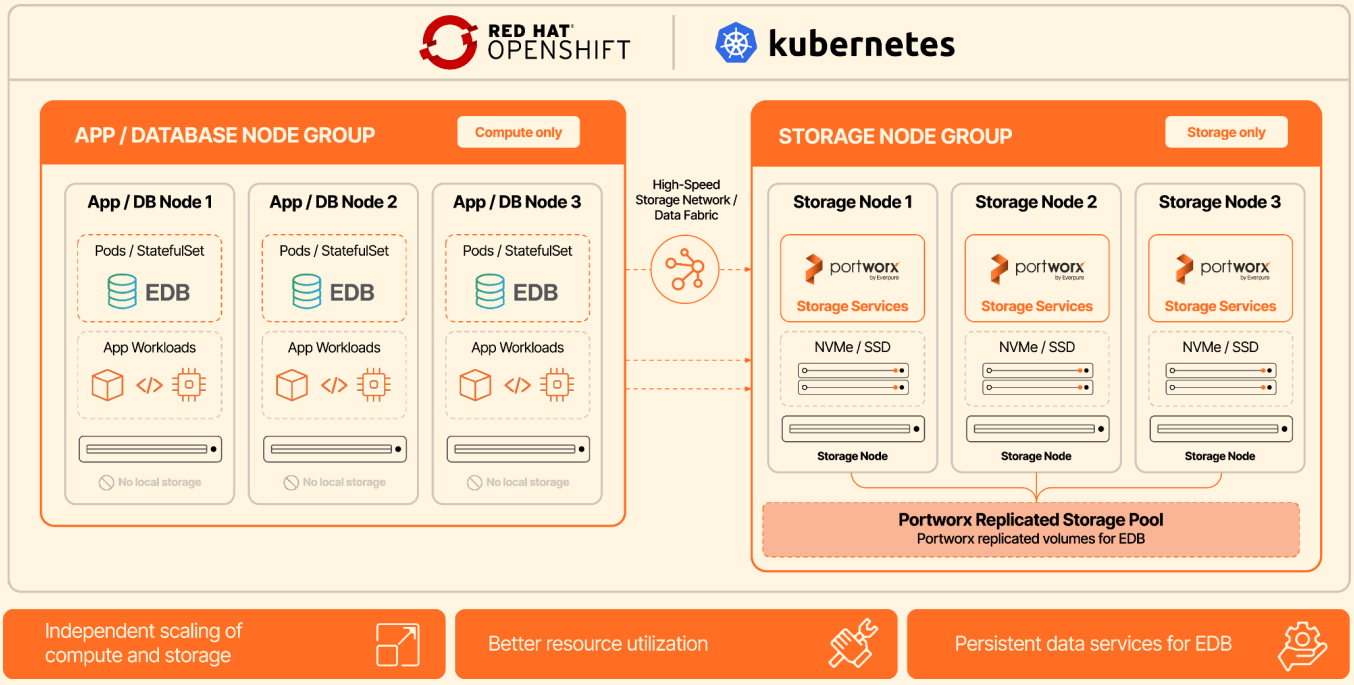

Figure 3. EDB on Portworx Disaggregated Mode

There is a non-hyperconverged mode called disaggregated mode, Portworx separates storage-bearing worker nodes from compute-only application nodes so storage and application capacity can scale independently. In this model, EDB database pods can run on compute-only nodes while consuming Portworx volumes over the network, which helps reduce stranded capacity, allows storage nodes to scale independently and gives platform teams a cleaner separation between storage operations and application operations. However it is most common, and provides the best performance to run in the previously mentioned hyperconverged mode where each worker node is storage and compute.

For disaggregated mode, the first step is to partition the Kubernetes worker nodes into those two groups and label them explicitly. Portworx uses `portworx.io/node-type=storage` for nodes that should host storage and `portworx.io/node-type=storageless` for nodes that should run as compute-only. In managed cloud environments, the cleanest pattern is to create separate node groups for each role so the labels persist automatically as nodes scale in and out. After generating the normal Portworx `StorageCluster` spec that uses Cloud Drives, disaggregated behavior is enabled by adding the environment variable `ENABLE_ASG_STORAGE_PARTITIONING=true`, which tells Portworx to honor the storage versus storageless partitioning defined by those node labels.

Design Considerations for Portworx Enterprise

Before deploying a database platform on Portworx, the topology must be defined. Portworx supports both converged and disaggregated deployments, and that decision must be made before installation because a disaggregated deployment requires nodes to be designated as storage or storageless in advance.

On supported platforms, PX-StoreV2 should be treated as the default storage engine baseline for new deployments.

From a cluster sizing perspective, Portworx must be treated as quorum infrastructure rather than as a simple CSI layer. Three storage nodes is the minimum viable starting point for a resilient deployment, because anything below that threshold weakens quorum and constrains replica placement. For production environments, six storage nodes is the stronger starting point where failure-domain balance, usable capacity, and sustained I/O distribution are priorities. In a three-node design, the loss of one node removes one-third of the available capacity; in a six-node design, the same event has a materially smaller operational impact.

Capacity planning must extend well beyond the initial database footprint. The Portworx design must account for primary data, WAL growth, replication overhead, snapshot retention, background rebuild headroom, and future expansion. Portworx supports both direct device consumption and cloud-backed storage pools. In a device-based deployment, the `storage.devices` section defines the media Portworx will consume. In a cloud-backed deployment, the `cloudStorage` section should define the provider, device specifications, metadata devices, and the initial number of storage nodes. When Portworx uses internal KVDB or PX-StoreV2, metadata capacity must be sized explicitly. It is recommended to allocate at least 64 GB for metadata, and larger or high-ingress environments should increase that allocation to 256 GB to extend degraded-volume tolerance.

***Note**: Portworx uses thin provisioning and Portworx AutoPilot can be used to dynamically and automatically scale individual PVCs or cluster wide storage pools. This provides a way for you to provision the capacity that you need at first and allow the Portworx tooling to help scale you into the future.*

For database workloads, storage class design is a primary design task rather than an implementation detail. Portworx StorageClasses support the key parameters required for stateful platforms, including replication factor, filesystem, volume group policy, snapshot intervals, encryption, backend selection, and I/O profile overrides. As a baseline, `repl: "2"` is the appropriate starting point for most stateful database deployments because it balances failure recovery with usable capacity. `repl: "3"` should be reserved for environments whose failure model justifies the additional replica cost. Portworx also provides database-oriented storage classes such as `px-csi-db, px-csi-db-encrypted`, and snapshot-capable variants, which should be treated as strong starting templates before workload-specific tuning is applied.

I/O tuning must be aligned to the database access pattern rather than applied generically. Portworx supports `io_profile` as a per-volume override and supports journal-backed behavior through the `journal: "true"` parameter when journal devices are present. For latency-sensitive database paths, `io_profile: journal` is the appropriate starting point.

Portworx also exposes runtime tuning through `runtimeOptions`, node-specific runtime overrides, and the `portworx.io/misc-args` annotation. Settings such as `num_io_threads`, CPU allocation, and memory allocation should therefore be treated as benchmark-driven tuning controls, not as arbitrary first-day changes. The documented `rt_opts` example of `small_conf=1` applies to two-node-with-arbiter configurations; for standard three-or-more-node production database clusters, `rt_opts` should remain conservative unless there is a validated performance requirement to change them.

Security and data protection must be designed before the first PVC is provisioned. PX-Security should be enabled where RBAC-driven authentication and authorization are required. A secrets provider must also be configured when encrypted volumes or cloud snapshots are in scope. Portworx supports Kubernetes Secrets as well as external providers such as Vault, and different providers can be assigned to different functions. For database platforms, that means defining the encryption model at the Portworx volume layer, deciding whether the underlying cloud drives must also be encrypted, and defining whether snapshot protection remains local, is copied to object storage, or is orchestrated through Portworx Backup.

Day-2 automation must be established up front rather than introduced only after pools begin to fill. Autopilot rules can monitor PVC and pool capacity and trigger automatic volume resize or storage-pool expansion based on Prometheus or Datadog metrics. For database platforms, that is critical for maintaining headroom on Postgres WAL and data volumes without relying on emergency manual intervention. Teams must decide whether pool growth should be fully automatic, constrained to approved maintenance windows, or subject to operator review. The action-window capability is especially useful in regulated environments where pool expansion must occur only during defined maintenance periods, while PVC resize actions can remain automatic.

Design Considerations for EDB Postgres

From the EDB side, preparation must begin with the understanding that EDB Postgres® AI for CloudNativePG™ Cluster and EDB Postgres® AI for CloudNativePG™ Global Cluster are operator-managed platforms, not a hand-built PostgreSQL deployment. For the sake of this paper, we will provide background on a few of the EDB options, however we will focus on distributed and single clusters. Please consult EDB Documentation for all product information.

- **CloudNativePG** is the **open-source** operator choice for standard PostgreSQL on Kubernetes.
- **EDB Postgres AI for CloudNativePG** is the **single-primary EDB-supported** operator path for PostgreSQL on Kubernetes.
- **EDB Postgres® AI for CloudNativePG™ Global Cluster** which uses **PGD**, is the advanced **distributed/global-cluster** option when the requirement moves beyond local HA into **multi-site and multi-master Postgres**.

Currently, the Global Cluster operator requires `cert-manager`, a registry pull secret in the operator namespace, and a certificate issuer before any PGD groups are deployed. The operator supports multiple replicas through leader election, and taints and tolerations should be used if operator pods must be kept off the same nodes that host PostgreSQL workloads. In larger environments, that separation should be treated as standard control-plane hygiene.

The next decision is the PGD deployment pattern. A flexible three-region pattern is an appropriate architectural baseline for multi-cluster: two data groups and one witness group, with logical or physical join selected according to the network and operational model. Where physical join is used, the first group creates the parent group, the subsequent data groups join physically, and the witness group remains logical. That pattern provides a strong general reference architecture for multi-site PGD deployments.

Security preparation must be explicit. EDB PGD for Kubernetes manages jobs, secrets, service accounts, and disruption budgets as part of the platform lifecycle. If users do not provide passwords and certificates, the operator will self-provision them. That is acceptable for non-production experimentation, but production systems should use a deliberate certificate and trust model. The `spec.connectivity.tls.mode` setting defaults to `verify-ca` and cannot be changed after the PGD group is created, so TLS mode must be selected correctly on day 0. Separate server and client CA secrets should be planned in advance, and cert-manager should be used to issue both the server TLS certificates and the client replication certificates used by the `streaming_replica` user.

PostgreSQL configuration must remain declarative. `ALTER SYSTEM` should not be used as the operational model for an operator-managed cluster because it introduces drift and is not replicated consistently across the platform. PostgreSQL settings should instead be defined in the cluster manifest so the operator can generate and maintain `custom.conf`. The operator-controlled baseline already sets critical replication- and security-relevant parameters such as `ssl=on`, `wal_level=logical`, TLS 1.3 minimum and maximum protocol versions, and replication-related timeout defaults. WAL retention must still be planned explicitly. Even when PGD is handling distributed replication, the platform team remains responsible for sizing WAL retention, replication slots, and the associated storage behavior to match the workload and recovery objectives.

Replication must be understood as a two-layer design. At the cluster level, EDB Postgres for Kubernetes manages physical streaming replicas, creates the `streaming_replica` user, and enforces TLS client certificate authentication for streaming replication. At the distributed layer, PGD extends replication and topology management across groups and regions. PGD defines the cross-group and cross-site database architecture, while the underlying PostgreSQL operator continues to own instance lifecycle, local replica orchestration, and parameter management. As a result, the EDB design must define the number of instances per group, witness placement, join method, WAL retention policy, and the relationship between local HA and distributed recovery.

Portworx sits below both Cluster and PGDGroup as the storage and data-services layer. It does not replace PostgreSQL replication or PGD replication; it complements them. Portworx `repl` protects the block volume, while PostgreSQL replication protects the database service and failover behavior. PGD handles the “how is the database distributed?” and Portworx answers “where does the data live, how is it protected at the storage layer, and how fast/local is access to it?”

Operator configuration must also be finalized before production rollout. The PGD operator can be customized through a ConfigMap or Secret named `pgd-operator-controller-manager-config` in the operator namespace.

That configuration can define additional pull secrets or override the default operand images for PGD and PGD Proxy. These changes are not applied dynamically; the operator deployment must be restarted, and only newly created PGD groups inherit the updated behavior. Registry design, image provenance, and private-repository access must therefore be established before the first production groups are created.

Finally, EDB preparation must include application-facing access policy, not just replication design. The PostgreSQL configuration model supports declarative `pg_hba` entries, and the operator can expand Kubernetes pod selectors into IP-based entries so ephemeral pod IPs do not have to be managed manually. TLS is enabled by default, with `ssl_min_protocol_version` and `ssl_max_protocol_version` set to TLSv1.3 unless deliberately changed. Combined with operator-managed client certificates, that provides a strong secure-by-default posture, but only if the application teams define their authentication model, certificate distribution model, and acceptable `pg_hba` policies before the platform is exposed to workloads.

Networking Considerations

Designing an efficient network is essential for the optimal performance and reliability of your Portworx deployment. This section outlines key considerations and best practices for configuring the network in a Portworx environment running on Kubernetes. By carefully planning the network architecture, you can ensure seamless communication, high availability, and enhanced security for your storage cluster. The following subsections cover critical aspects of network design, including the separation of management and data interfaces, as well as important firewall considerations to protect and optimize your infrastructure.

Management and Data Interfaces

When designing the networking configuration for a Portworx deployment, several key considerations must be taken into account to ensure optimal performance, reliability, and scalability. Portworx leverages the worker node's physical network interface card (NIC) for serving remote storage connections and for keeping persistent volume replicas synchronized across nodes. Special care should be taken when designing the network configuration for Portworx deployments in Kubernetes environments.

- **Default NIC usage:** By default, Portworx utilizes the same network interfaces used by the Kubernetes cluster for both management traffic and data replication. This configuration simplifies the initial setup of the Portworx storage cluster but may lead to potential network congestion as both types of traffic share the same physical network resources. Using the default NIC configuration can be acceptable for lab or development environments where simplicity is preferred, but it is generally not recommended for production environments or workloads where storage performance is critical.
- **Dedicated data networks:** To optimize network performance and reduce congestion, Portworx can be configured to use a dedicated NIC specifically for handling data replication traffic between nodes. This approach segregates storage replication traffic from Kubernetes management and application traffic, ensuring that the interfaces used by the cluster for orchestrating and servicing applications remain uncongested and performant. Implementing a dedicated data network can significantly improve the throughput, consistency, and reliability of the storage cluster, particularly in environments with high replication or heavy I/O demands.

***Note**: Bonded NICs would be preferred for the Portworx data network in order to prevent a disruption in the event that a NIC fails.*

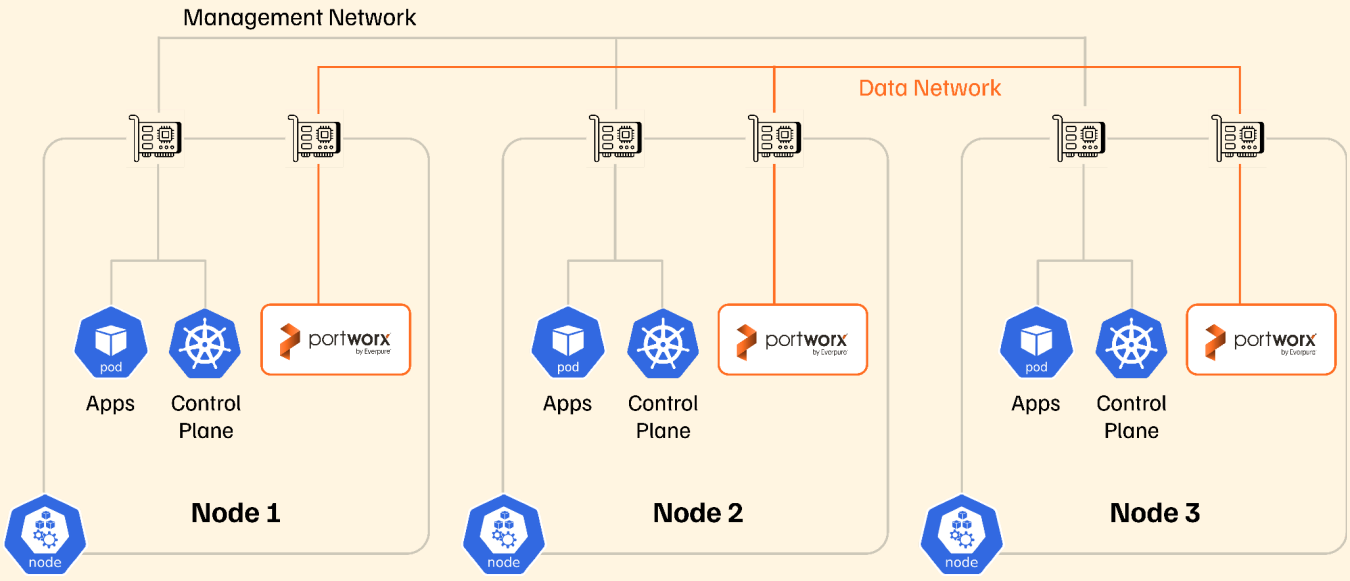

Figure 4. Management and Data Network Design

- **Network performance:** The physical network has a great deal to do with how fast data can be written to replicas across nodes in the Kubernetes cluster. Writes to a persistent volume must be replicated to a second node before committing the writes back to the application, meaning the network is involved in how fast applications can write data to disk when using Portworx for data availability. Portworx recommends a network bandwidth of 10Gbps, with a minimum requirement of 1Gbps with latency less than 5 ms. Ensure that all storage nodes participating in data replication for Portworx use the same speed NICs to provide consistent storage performance.

***Note**: The network bandwidth has an important effect on performance of the storage cluster to maintain replicas. Networks with 25Gbps or more would be preferred for high intensity data workloads.*

Firewall Considerations

When deploying Portworx in a Kubernetes environment, ensuring proper network communication is essential for the seamless operation of your storage infrastructure. A critical aspect of this setup involves configuring your firewall to allow the necessary ports that facilitate communication between Portworx nodes and management interfaces. Properly opening these ports ensures that data replication, monitoring, and management tasks can be performed without interruption, maintaining the reliability and performance of the storage cluster. Below, the essential firewall ports required for running Portworx are listed.

East-to-West (Internal Communication Ports)

Kubernetes	OpenShift	Description
9001	**17001**	Portworx management port [REST]
9002	**17002**	Portworx node-to-node port [gossip]/UDP | **Required** to open when using external KVDB
9003	**17003**	Portworx storage data port
9004	**17004**	Portworx namespace [RPC]
9012	**17009**	Portworx node-to-node communication port [gRPC]
9013	**17010**	Portworx namespace driver [gRPC]
9014	**17011**	Portworx diags server port [gRPC]
9018	**17015**	Portworx kvdb peer-to-peer port [gRPC]
9019	**17016**	Portworx kvdb client service [gRPC]
9020	**17017**	Portworx gRPC SDK server [REST]
9021	**17018**	Portworx gRPC SDK gateway [REST]
9022	**17019**	Portworx health monitor [REST]
9024	**17021**	Telemetry log uploader (Portworx version 2.13.8 and later) [gRPC]
9029	**17021**	Telemetry log uploader (Portworx version 2.13.8 and later) [gRPC]
12001	**20001**	Telemetry metrics collector [gRPC]
12002	**20002**	Telemetry phone home [HTTP]
2379	**2379**	External KVDB (etcd) port [gRPC] | **Open only if** you run an external etcd cluster

Table 1. Portworx East-West Ports Requirement

Inbound

Kubernetes	OpenShift	Description
9001	**17001**	Portworx management port [REST]
9021	**17018**	Portworx gRPC SDK gateway [REST]

Table 2. Portworx Inbound Ports Requirement

Outbound

Type	TCP Ports	Scope	Destination Hosts	Description
Install / Upgrade	443	PX install & version updates	install.portworx.com, mirrors.portworx.com	Retrieves install spec, helper scripts, and downloads PX kernel modules
License activation	443	PAYG licensing	rest.zuora.com	Usage reporting & license retrieval Not applicable for air-gapped deployments
	443	PAYG licensing	flex1327.compliance.flexnetoperations.com	For licensing to work Not applicable for air-gapped deployments
Telemetry	443	Cluster registration & metrics	pxessentials.portworx.com, register.cloud-support.purestorage.com, rest.cloud-support.purestorage.com	Registers cluster and uploads usage metrics
	443 / 80	Event log uploads	logs-01.loggly.com	Sends PX log events to Portworx Support
Snapshots / Backups	443	CloudSnaps, backups, restores	Your S3 or S3-compatible endpoint	Persist snapshots & object data

Table 3. Portworx Outbound Ports Requirement

Installation

Configure the PX Operator

The recommended starting point is PX Central, using the spec generator rather than hand-authoring Portworx manifests. The `StorageCluster` CRD is the authoritative definition of the Portworx deployment, and PX Central is the preferred method for generating that specification for the target Kubernetes environment. In practice, the workflow should begin in PX Central under Install and Run, where the operator-based deployment is selected and the cluster spec is generated with the required storage, networking, monitoring, telemetry, and security settings for the environment.

The first major design decision is the storage model. Portworx supports both local storage and cloud-backed storage, but the cluster should be designed around one primary model for a given storage node group. For local storage, the `spec.storage` section defines the block devices Portworx will consume. For cloud-backed deployments, the `spec.cloudStorage` section defines the provider and the backing disk specifications that Portworx will provision and manage automatically. When using cloud storage, the `spec.storage` fields should be left empty so the `cloudStorage` configuration remains authoritative.

Cloud Drive (a feature of Portworx Enterprise that automates the provisioning, attaching, and scaling of cloud block storage [like Everpure FlashArray, AWS EBS, Azure Disk, GCP Persistent Disk, and OCI Block Volumes]) configuration should be treated as foundational, not as day-2 tuning. The provider must be defined correctly, and the design should explicitly specify device sizing, metadata placement, and journal placement where applicable. These settings are immutable after initial provisioning, which means later changes normally require adding newly provisioned storage nodes with the revised configuration rather than expecting existing nodes to be reprovisioned in place.

Optional parameters should be set deliberately. Typical examples include CSI enablement, snapshot controller installation, monitoring, telemetry, HTTP or HTTPS proxy configuration, and runtime tuning. Portworx supports proxy settings through `PX_HTTP_PROXY` and `PX_HTTPS_PROXY`, and additional runtime behavior can be shaped with `runtimeOptions` or `portworx.io/misc-args` where the platform design requires it.

For design clarity, Cloud Drives should also be distinguished by infrastructure model. In public cloud, the provider values typically map to native cloud environments such as `aws`, `azure`, or `gce`. In enterprise-backed environments, the provider may instead be `pure` or `vsphere`, which reflects a different underlying infrastructure model even though Portworx still manages the backing capacity through the `cloudStorage` workflow.

Example: StorageCluster using cloud drives

````
### Example: StorageCluster using cloud drives

```yaml
apiVersion: core.libopenstorage.org/v1
kind: StorageCluster
metadata:
  name: px-cluster
  namespace: kube-system
spec:
  image: portworx/oci-monitor:3.6
  kvdb:
    internal: true
  cloudStorage:
    provider: aws
    deviceSpecs:
      - type=gp3,size=200
    journalDeviceSpec: type=gp3,size=20
    systemMetadataDeviceSpec: type=gp3,size=64
    kvdbDeviceSpec: type=gp3,size=32
    initialStorageNodes: 3
  csi:
    enabled: true
    installSnapshotController: true
  monitoring:
    prometheus:
      enabled: true
  autopilot:
    enabled: true
  env:
    - name: PX_HTTP_PROXY
      value: http://proxy.example.com:3128
    - name: PX_HTTPS_PROXY
      value: http://proxy.example.com:3128
  runtimeOptions:
    num_io_threads: "10"
````

For local-disk deployments, this same design would move away from `cloudStorage` and instead define the backing media under `spec.storage.devices`. The key architectural point is that Portworx should be told explicitly whether it is expected to consume pre-attached local devices or provision managed backing disks on behalf of the cluster.
````

An example of specifying local storage can be seen below.

```
kind: StorageCluster
apiVersion: core.libopenstorage.org/v1
metadata:
  name: px-cluster-ff29e069-22db-46df-906d-8bebe8599fb0
  namespace: portworx
  annotations:
    portworx.io/install-source:
"https://install.portworx.com/3.6?operator=true&mc=false&kbver=1.31.0&ns=portworx&b=true&kvdbtls=true&cert
mgr=true&dmthin=true&md=%2Fdev%2Fsdc&s=%22%2Fdev%2Fsdb%2CpoolLabels%3DPERF%3AMEDIUM%22&c=px-cluster-ff29e0
69-22db-46df-906d-8bebe8599fb0&stork=true&csi=true&mon=true&aut=false&tel=true&st=k8s&promop=true"
    portworx.io/misc-args: " -T px-storev2 "
spec:
  image: docker.io/portworx/oci-monitor:3.6.0.1
  imagePullPolicy: Always
  kvdb:
    internal: true
    enableTLS: true
  storage:
    devices:
      - /dev/sdb,poolLabels=PERF:MEDIUM
    systemMetadataDevice: /dev/sdc
  secretsProvider: k8s
  stork:
    enabled: true
    args:
      webhook-controller: "true"
  csi:
    enabled: true
  monitoring:
    telemetry:
      enabled: true
    prometheus:
      enabled: true
      exportMetrics: true
  certManager:
    enabled: true
```

Install the PX Operator and Cluster

Once the specification has been generated, installation becomes a two-step process: deploy the Portworx Operator, then deploy the `StorageCluster`. The operator is responsible for reconciling the `StorageCluster` object and creating the Portworx runtime components in the cluster, so the `StorageCluster` should be treated as the declarative definition of the deployment rather than as a one-time install file.

The commands are straightforward. First, apply the operator manifest generated by PX Central. Next, apply the `StorageCluster` manifest, or use the generated install URL directly if that is how the spec was produced. From that point, the operator converges the deployment and brings up the Portworx data plane, CSI components, and any optional services selected during spec generation.

Commands to deploy the operator and cluster

```
kubectl create -f px-operator.yaml
kubectl apply -f storage-cluster.yaml
```

```
kubectl apply -f
'https://install.portworx.com/<PXVER>?operator=true&mc=false&kbver=<KBVER>&b=true&c=<cluster-id>&stork=true&csi=true&mon=true&tel=true&st=k8s&promop=true'
```

After deployment begins, the first validation point is Kubernetes pod state. A healthy installation should show the operator pod running, a `px-cluster` pod on each storage node, and the `px-csi-ext` components running. If monitoring is enabled, the Prometheus-related components should also be present.

Commands to validate the deployment

```
kubectl -n kube-system get pods -o wide
kubectl -n kube-system get storagecluster
kubectl exec <px-pod> -n kube-system -- /opt/pwx/bin/pxctl status
kubectl exec <px-pod> -n kube-system -- /opt/pwx/bin/pxctl cluster provision-status
kubectl exec <px-pod> -n kube-system -- /opt/pwx/bin/pxctl service pool show
```

A healthy Portworx cluster should present a recognizable validation pattern. The `StorageCluster` should report `Online`, `pxctl status` should report `Status: PX is operational`, and the output should show storage pools and backing devices visible to Portworx. The `pxctl cluster provision-status` output should show the storage nodes as provisioned and online. Together, those checks confirm that the operator, the `StorageCluster`, and the Portworx runtime are aligned and functioning as intended.

Portworx Storage Classes

StorageClasses are a foundational design element for EDB on Portworx because they translate database requirements into enforceable storage behavior at provisioning time. Rather than relying on a generic cluster default, a Portworx StorageClass allows the platform team to define the exact policy for a given database path, including replication factor, filesystem, encryption, snapshot behavior, backend selection, and I/O tuning, while EDB manifests can bind explicitly to those classes for predictable and repeatable deployment outcomes.

This is particularly important for production database platforms because **PGDATA** and **WAL** often have different performance and protection requirements, and StorageClasses provide the mechanism to express those differences cleanly instead of forcing all volumes into the same policy model.

In practice, StorageClasses are what make Portworx operationally useful for EDB: they standardize provisioning, make resiliency and performance choices explicit, and ensure that every database volume is created with the intended availability and operational characteristics from day 0.

Example: Database-oriented Portworx StorageClass

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-db-ha
provisioner: pxd.portworx.com
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
parameters:
  repl: "2"
  io_profile: "db_remote"
  journal: "true"
  secure: "true"
  fs: "ext4"
  snap_interval: "60"
```

`repl: "2"` sets the Portworx replication factor to two, meaning the volume is stored on two nodes for storage-layer high availability. It is commonly used as the production baseline because it balances resilience and capacity efficiency better than `repl: "3"`, while providing HA that `repl: "1"` does not.

`io_profile: "db_remote"` tells Portworx to use the database-oriented remote-write optimization path for the volume. This profile coalesces multiple sync operations into fewer flushes and acknowledges them after data is copied into memory across replicas, which helps reduce sync overhead on HA database volumes with `repl >= 2`.

`journal: "true"` instructs Portworx to use the journal device configured on the node for that volume's data path. This is useful for short, bursty, sync-heavy write patterns because it absorbs frequent syncs more efficiently, but it only makes sense if journal devices or journal partitions were configured during the Portworx install.

`secure: "true"` creates the volume as an encrypted Portworx volume. In practice, this enables encryption at rest for the PVC and requires the cluster's key-management and secrets configuration to be set up correctly.

`fs: "ext4"` specifies the filesystem Portworx should lay down on the volume. Portworx supports `ext4` and `xfs`, but `ext4` is the preferred and default filesystem across all backends, which is why it is the usual choice in production examples.

`snap_interval: "60"` enables periodic snapshots every 60 minutes for that volume. The value is defined in minutes, and setting it to `0` disables periodic snapshots altogether.

Key profiles and StorageClass setting differences for EDB

For EDB on Kubernetes, Portworx replication settings and PostgreSQL replication settings solve different problems and should be treated as complementary layers rather than substitutes. Portworx `repl` defines how many storage replicas back a volume, while PostgreSQL replication determines how many database instances can continue serving data after an instance failure. In practice, the storage layer protects volume availability and accelerates restart and reattachment after node loss, while the database layer protects service continuity, role promotion, and transactional recovery.

For that reason, the most balanced starting point for production EDB deployments is typically Portworx `repl: "2"` combined with PostgreSQL replication of 2. That combination provides high availability at both layers without incurring the additional write-path and capacity overhead of `repl: "3"` for every volume.

Portworx StorageClass setting	Storage behavior	Considerations	EDB guidance
`repl: "1"`	Maintains a single storage replica for the volume.	Lowest storage overhead and the shortest write path, but no storage-layer HA. If the hosting node or backing device is lost, volume recovery depends entirely on the database replication layer and recovery workflow.	Generally appropriate only for non-production testing, temporary environments, or tightly benchmarked cases where the database layer is solely responsible for redundancy. Unless used with Portworx KDS where FA provides HA for Repl 1 or where database replication is providing redundancy.
`repl: "2"`	Maintains two synchronous storage replicas and is the minimum setting that enables the `db_remote` optimization path.	Best balance of write performance, storage efficiency, and storage-layer resiliency. It provides HA without the full write amplification and capacity cost of a third storage replica.	Recommended default for most production EDB deployments, especially when paired with PostgreSQL replication of 2.
`repl: "3"`	Maintains three storage replicas for the volume.	Highest resilience at the storage layer, but it consumes the most capacity and introduces the most replica coordination in the write path.	Reserve for workloads whose failure model or site policy explicitly requires a third storage replica.

Table 4. Storage Class Replication Behaviour

This distinction matters because PostgreSQL replication and Portworx replication operate at different layers. PostgreSQL replication protects the database service by maintaining replica instances that can be promoted or used for read scaling, while Portworx replication protects the block volume itself. Using only PostgreSQL replication with `repl: "1"` can preserve database redundancy, but it leaves restart, reattachment, and storage-level failure handling entirely to the database layer which can incur costly and lengthy rebuilds.

Using only higher Portworx replication without sufficient PostgreSQL replicas improves volume durability, but it does not replace database failover design. For most production EDB deployments, the configuration mentioned above of Portworx `repl: "2"` plus PostgreSQL replication of 2 is the most practical balance of performance, availability, and operational simplicity.

Portworx I/O profiles add a second layer of tuning at the StorageClass level. These settings do not change replica count; instead, they change how Portworx handles sync-heavy write behavior and flush activity for a volume. Portworx uses `auto` as the cluster default, and that default selects between `none` and `db_remote` based on the replication configuration of the volume. When a database team wants deterministic behavior for EDB data volumes or WAL volumes, the preferred pattern is to specify the `io_profile` explicitly in the Portworx StorageClass and then reference that StorageClass in the EDB manifest.

IO profile	What it does	Best fit for EDB
`auto`	Automatically selects the most appropriate behavior for the volume. In current Portworx releases, it chooses `db_remote` when the volume has `repl >= 2`; otherwise it falls back to `none`.	Strong default when a team wants Portworx to pick the profile automatically, especially for general-purpose database deployments.
`db_remote`	Uses a write-back flush coalescing algorithm that collapses multiple syncs within a 100 ms window into a single sync and acknowledges them after they are copied to memory on all replicas. It is designed for `repl: "2"` or higher volumes.	Best default for HA EDB data volumes on `repl: "2"` or `repl: "3"`, where reducing sync overhead across replicas is important.
`journal`	Writes stably to a dedicated journal device and commits to the backing pool in batches, reducing the cost of frequent sync operations. It requires a dedicated journal device or partition.	Best fit for latency-sensitive database paths when journal devices are present. It can be especially effective for WAL-heavy or sync-heavy workloads that are benchmarked to benefit from journal acceleration.
`auto_journal`	Monitors write behavior over time and switches between `none` and `journal` when journal mode is beneficial.	Useful where journal devices are available but workload behavior varies enough that dynamic selection is preferable.
`none`	Disables I/O profile optimizations and processes I/O without special handling.	Appropriate for simple or non-database workloads, or as a baseline when benchmarking the effect of the optimized profiles.
`use_cluster_default`	Inherits the cluster-wide default I/O profile.	Useful for teams that standardize tuning globally and do not want per-class overrides.

Table 5. Portworx IO Profile Behaviour

For EDB specifically, the clearest production pattern is to separate the Portworx classes by database path. A primary data class can use `repl: "2"` with `io_profile: "db_remote"` to balance HA and write efficiency across replicated volumes, while a WAL-oriented class can use `repl: "2"` with `journal: "true"` and, where supported by benchmark results, `io_profile: "journal"`. This makes the intent of each StorageClass explicit and allows EDB manifests to bind PGDATA and WAL to different Portworx policies instead of inheriting a generic cluster default. Portworx also supports explicit StorageClass references from Kubernetes PVC consumers, so production EDB manifests should name the desired Portworx StorageClass directly rather than relying on the cluster default behavior.

Note: *hostssl requires TLS, while scram-sha-256 uses password authentication. For more information see the documentation on pg_hba.*

Example: EDB Cluster using explicit Portworx StorageClasses for data and WAL

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: edb-cluster
spec:
  instances: 3
  storage:
    storageClass: px-db-ha
    size: 100Gi
  walStorage:
    storageClass: px-db-wal
    size: 50Gi
  postgresql:
    parameters:
      wal_keep_size: "1GB"
      max_replication_slots: "32"
    pg_hba:
      - hostssl app app ${podselector:app-clients} scram-sha-256
```

Using Portworx storage classes in PGD

Example: PGDGroup using an explicit Portworx StorageClass for MultiRegion

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: group-example
spec:
  instances: 2
  proxyInstances: 2
  witnessInstances: 1
  imagePullSecrets:
    - name: registry-pullsecret
  pgd:
    parentGroup:
      name: world
      create: true
  cnp:
    storage:
      storageClass: px-db-ha
      size: 100Gi
```

Operations

Integration of EDB operator

For a single EDB database cluster, the required control plane is the EDB Postgres for Kubernetes operator. For a PGD deployment, the required control plane is the EDB Postgres Distributed for Kubernetes operator, which also installs a supported EDB Postgres for Kubernetes operator because each PGD data node is managed internally as a `Cluster` resource. This distinction matters because the single-cluster path and the PGD path use different operator namespaces, different install manifests, and different readiness checks.

The first requirement in both cases is access to the EDB container registry. The operator images and operand images are pulled from `docker.enterprisedb.com`, so an EDB subscription token must be available and stored as a pull secret in the namespace where the operator will run. For a single EDB cluster deployment, the default operator namespace is `postgresql-operator-system`. For PGD, the default operator namespace is `pgd-operator-system`.

Single EDB cluster operator installation

The single-cluster path is the correct starting point when the goal is to deploy one EDB PostgreSQL cluster with local high availability inside a single Kubernetes cluster. The operator installs as a Kubernetes `Deployment`, defaults to a single replica, supports leader election, and can be scaled or constrained with taints and tolerations if operator-level high availability or workload separation is required.

The installation sequence is straightforward:

1. Create the operator namespace.
2. Create the registry pull secret.
3. Install the operator manifest.
4. Verify the operator deployment is ready.
5. Create a dedicated application namespace and deploy the first `Cluster`.

Commands: install the EDB Postgres for Kubernetes operator

```
export EDB_SUBSCRIPTION_TOKEN='<your-token>'

kubectl create namespace postgresql-operator-system

kubectl create secret -n postgresql-operator-system docker-registry edb-pull-secret \
  --docker-server=docker.enterprisedb.com \
  --docker-username=k8s \
  --docker-password="$EDB_SUBSCRIPTION_TOKEN"
```

***Note**: make sure you are using the most up to date version of pg4k. This was the version used at the time of writing this document.*

```
kubectl apply --server-side -f \
  https://get.enterprisedb.io/pg4k/pg4k-1.29.1.yaml

kubectl rollout status deployment \
  -n postgresql-operator-system postgresql-operator-controller-manager
```

For environments that require custom operator scope or manifest customization, the `cnp` plugin can generate a tailored installation manifest. That is the correct path when the operator must watch only selected namespaces rather than the full cluster.

Example: single EDB cluster namespace and Cluster manifest

```
apiVersion: v1
kind: Namespace
metadata:
  name: edb-single
---
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: edb-single-cluster
  namespace: edb-single
spec:
  instances: 3
  imageName: docker.enterprisedb.com/k8s/edb-postgres-advanced:18.1-standard-ubi9
  storage:
    storageClass: px-db-ha
    size: 100Gi
  walStorage:
    storageClass: px-db-wal
    size: 50Gi
```

This is the minimum operator-ready shape for a single EDB cluster. The key design rule is to pin the operand image explicitly and avoid floating tags such as `latest`, which undermine repeatability and version control in production environments.

Commands: deploy and verify the first single EDB cluster

```
kubectl apply -f edb-single-cluster.yaml
kubectl get pods -n edb-single
kubectl get pods -n edb-single -l k8s.enterprisedb.io/cluster=edb-single-cluster
```

PGD operator installation for distributed and multi-cluster deployments

The PGD path is the correct starting point when the target design requires distributed PostgreSQL across regions, sites, or multiple Kubernetes clusters. PGD can run in a single Kubernetes cluster or across multiple Kubernetes clusters, but the operational model remains the same: install cert-manager, create registry access, install the PGD operator, and then deploy PGD groups in dedicated namespaces rather than in the operator namespace.

PGD has one additional hard prerequisite: `cert-manager` 1.10 or higher. The PGD operator depends on certificate management for TLS and replication-related credentials. The manifest installation path is the most direct way to make a cluster ready for PGD. That manifest installs both the PGD operator and a tested EDB Postgres for Kubernetes operator in the same namespace.

Commands: install cert-manager and the PGD operator

```
export EDB_SUBSCRIPTION_TOKEN='<your-token>'

kubectl apply -f \
  https://github.com/cert-manager/cert-manager/releases/download/v1.16.2/cert-manager.yaml

kubectl create namespace pgd-operator-system

kubectl create secret -n pgd-operator-system docker-registry edb-pull-secret \
  --docker-server=docker.enterprisedb.com \
  --docker-username=k8s \
  --docker-password="$EDB_SUBSCRIPTION_TOKEN"

kubectl apply --server-side -f \
  https://get.enterprisedb.io/pg4k-pgd/pg4k-pgd-1.1.3.yaml

kubectl get deployment -n pgd-operator-system pgd-operator-controller-manager
```

For Helm-based deployments, the operator can be installed with explicit image credentials and customized default operand images. This is the preferred path when the platform team needs tighter control over default PGD image selection, proxy image selection, or operator packaging. The operator stores those defaults in `pgd-operator-controller-manager-config`, and new PGD groups inherit them unless overridden in the manifest.

Commands: Helm installation for a PGD-ready environment

```
helm upgrade --dependency-update \
  --install edb-pg4k-pgd \
  --namespace pgd-operator-system \
  --create-namespace \
  edb/edb-postgres-distributed-for-kubernetes \
  --set global.repository=docker.enterprisedb.com/k8s \
  --set image.imageCredentials.username=k8s \
  --set image.imageCredentials.password=${EDB_SUBSCRIPTION_TOKEN}
```

In a multi-cluster PGD design, this operator preparation must be repeated on each Kubernetes cluster that will host a PGD group. Each cluster must have the operator, cert-manager, registry access, and a deployment namespace ready before the PGD topology is created.

Architecting for resiliency

Multicluster PGD deployment patterns

PGD supports several multicluster deployment patterns, and the operator footprint should be aligned to the pattern selected for the target architecture. The correct pattern is determined by the number of locations, the number of active write locations, the required disaster-recovery outcome, and whether the design prioritizes active-active write locality, read scaling, lower-cost disaster recovery, or regional data-residency controls.

Pattern	Locations	Active write locations	Best fit	Disaster recovery posture
Single data group	1	1	Single-location HA and the simplest PGD starting point	Node failure only
Two data groups, active-active	2	2	Two-site active-active operation with local writes in both locations	Data center failure
Three data groups, active-active-active	3+	3+	True multiregion active-active deployment with local reads and writes in each region	Region failure
Single data group, global read scaling	1+	1	Central write group with subscriber-only groups for regional reads and analytics isolation	Node failure only
Primary active group, DR group	2	1	Lower-cost disaster recovery with a full primary site and reduced DR site	Data center failure, manual failover
Multiple locations, data residency	3+	3+	Multiregion deployment where sensitive data remains in-region and only global data is replicated	Region failure

Table 6. Postgres Deployment Patterns

For operator planning, the deployment implications are straightforward. A single data group is the foundational pattern for one-location high availability. Two data groups and three or more data groups are the main active-active multicluster patterns and require operator readiness in each participating cluster. The global read-scaling pattern adds subscriber-only groups that scale reads without participating in writes or consensus. The primary active group with DR is the lower-cost two-location pattern when full active-active is not required. The data-residency pattern is the correct model when legal or regulatory boundaries require selective replication and in-region control of sensitive data.

Namespace and certificate preparation for PGD groups

PGD groups should be deployed in dedicated namespaces. The default operator namespace is reserved for operator components, while the PGD groups themselves belong in application namespaces. Before a `PGDGroup` is created, the target namespace should have a certificate issuer and any required pull secrets in place. In multi-namespace and multi-cluster patterns, the CA material must be established consistently so the generated server and client certificates are valid across the topology.

Example: namespace, issuer, and image pull secret for a PGD target namespace

```
apiVersion: v1
kind: Namespace
metadata:
  name: region-a
---
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
  namespace: region-a
spec:
  selfSigned: {}
```

```
kubectl create secret docker-registry registry-pullsecret \
  -n region-a \
  --docker-server=docker.enterprisedb.com \
  --docker-username=k8s \
  --docker-password=${EDB_SUBSCRIPTION_TOKEN}
```

Example: PGDGroup ready for a distributed deployment

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
  namespace: region-a
spec:
  instances: 3
  witnessInstances: 1
  imagePullSecrets:
    - name: registry-pullsecret
  pgd:
    parentGroup:
      name: world
      create: true
  cnp:
    storage:
      storageClass: px-db-ha
      size: 100Gi
```

This is the minimum deployment-ready pattern for the first PGD group. Additional groups in the topology follow the same model, with discovery, join method, and certificate material adjusted for the target architecture. For a three-region design, a common baseline is two data groups and one witness group, with physical or logical join selected according to the network and operational model.

For a second region "B"

The key change is that it **joins** the existing topology instead of creating it. In the physical-join sample, region B sets:

- `spec.pgd.groupJoinMethod: physical`
- `spec.pgd.discovery` pointing at region A
- `spec.cnp.joinMethod: physical` for the intra-group node joins.

Conceptually, the manifest should look like this:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-b
  namespace: region-b
spec:
  instances: 3
  witnessInstances: 1
  imagePullSecrets:
    - name: registry-pullsecret
  pgd:
    parentGroup:
      name: world
    groupJoinMethod: physical
    discovery:
      # point this at a reachable region-a PGD service / node endpoint
      # exact values depend on your service naming and connectivity model
  cnp:
    joinMethod: physical
    storage:
      storageClass: px-db-ha
      size: 100Gi
```

Apply it:

```
kubectl -n region-b apply -f region-b.yaml
```

Example witness group

Here's the kind of manifest you'd use for the **third-location witness**:

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-c
  namespace: region-c
spec:
  instances: 0
  witnessInstances: 1
  imagePullSecrets:
    - name: registry-pullsecret
  pgd:
    parentGroup:
      name: world
    groupJoinMethod: logical
    discovery:
      # point to a reachable PGD service in region-a or region-b
      # exact fields depend on your networking / service exposure model
  witness:
    # optional witness-specific overrides go here
```

Apply the witness group:

```
kubectl -n region-c apply -f region-c.yaml
```

Validation checks

The operator installation is not complete until the control-plane objects are verified. For a single EDB cluster deployment, the operator deployment in `postgresql-operator-system` must be ready before any `Cluster` manifests are applied. For PGD, the `pgd-operator-controller-manager` deployment must be ready, and the cluster should show both the PGD operator and the underlying PG4K dependency installed before any `PGDGroup` manifests are created.

Commands: validation for single-cluster and PGD paths

```
kubectl get deployments -n postgresql-operator-system
kubectl get deployments -n pgd-operator-system
kubectl get pods -n cert-manager
kubectl get ns
```

Commands: validation for multi-cluster with the PGD CLI:

```
pgd cluster show
pgd nodes list
pgd group pgd show
```

The operational guidance is straightforward. Use the standalone EDB Postgres for Kubernetes operator when the requirement is a single EDB database cluster in one Kubernetes environment. Use the PGD operator when the design requires distributed PostgreSQL across regions or clusters. In both cases, operator installation is the control-plane prerequisite that makes the rest of the platform possible, and it must be completed with the correct namespace, registry access, certificate handling, and deployment validation before the first database manifest is applied.

PX Security and Data Encryption

Security is a first-class design requirement for any EDB platform because the database layer carries the system of record, the replication layer carries application continuity, and the storage layer carries the persistent data that must remain confidential and recoverable under failure or attack. A production design therefore needs encryption in transit, encryption at rest, authenticated storage access, and a clear decision about where cryptographic control should live: in PostgreSQL, in Portworx, in the backing array, or across multiple layers at the same time.

Security requirements for EDB

For EDB, security begins at the database and operator layer. PostgreSQL server connections should use TLS, client authentication should be controlled explicitly, and any distributed PGD deployment must treat secure inter-node communication as mandatory rather than optional. EDB Postgres Distributed for Kubernetes requires secure communication between PGD resources and relies on PostgreSQL TLS for that communication, with cert-manager used to provision the required certificates. The default PGD TLS mode is `verify-ca`, and that mode must be chosen correctly at creation time because it cannot be changed after the PGD group is established.

At the database level, EDB also supports transparent data encryption for EPAS. That gives the platform team a way to encrypt the database contents themselves rather than relying only on storage encryption. In the Kubernetes model, EDB integrates TDE with external key management such as HashiCorp Vault through wrap and unwrap commands, which makes it possible to keep key material external to the database pods and enforce a cleaner security boundary around database encryption keys.

EDB encryption in practice

For a single EDB cluster, server and client certificates should be managed explicitly. EDB Postgres for Kubernetes supports cert-manager-managed certificates for TLS server authentication and client certificate authentication. That gives the platform a secure-by-default transport layer and allows `pg_hba` policy to enforce certificate-based access where appropriate.

For PGD, the requirement is broader. Each PGD node needs server certificate configuration and client certificate configuration. In practice, that means the platform must provide a server CA secret, a client CA secret, and cert-manager issuers capable of generating the server TLS certificates and client replication certificates used across the distributed topology.

Example: EDB PGD TLS configuration

```
apiVersion: pgd.k8s.enterprisedb.io/v1beta1
kind: PGDGroup
metadata:
  name: region-a
spec:
  connectivity:
    tls:
      mode: verify-ca
      clientCert:
        caCertSecret: client-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: client-ca-issuer
              kind: Issuer
              group: cert-manager.io
      serverCert:
        caCertSecret: server-ca-key-pair
        certManager:
          spec:
            issuerRef:
              name: server-ca-issuer
              kind: Issuer
              group: cert-manager.io
```

Example: EDB with external key management

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: hashicorp-vault-tde
spec:
  envFrom:
    - secretRef:
        name: env-var-secret
  projectedVolumeTemplate:
    sources:
      - secret:
          name: tls-vault-secret
          items:
            - key: ca
              path: certificate/vault-ca.pem
  instances: 3
  storage:
    size: 1Gi
  postgresql:
    epas:
      tde:
        enabled: true
        wrapCommand:
          name: vault-token
          key: wrap
        unwrapCommand:
          name: vault-token
          key: unwrap
```

Portworx encryption and PX-Security

Portworx adds the storage-side security controls that EDB alone does not provide. At the simplest level, Portworx can encrypt PVCs through the `secure: "true"` StorageClass parameter. When a cluster-wide encryption key is configured, every volume created from that secure StorageClass is encrypted at the Portworx layer. This is the cleanest pattern when the requirement is uniform storage-layer encryption for all database PVCs in the cluster or for a specific set of encrypted StorageClasses.

PX-Security adds the authorization layer on top of that model. With PX-Security enabled, Portworx validates storage operations using user tokens referenced by the StorageClass. For CSI, the StorageClass should include token references for the provision, node-publish, and controller-expand operations. This makes storage authorization explicit and lets the platform team enforce RBAC-backed control over who can provision, mount, or resize encrypted database volumes.

Example: set the cluster-wide Portworx encryption key

```
kubectl -n portworx create secret generic px-vol-encryption \
  --from-literal=cluster-wide-secret-key='<strong-passphrase>'

PX_POD=$(kubectl get pods -l name=portworx -n portworx -o jsonpath='{.items[0].metadata.name}')
kubectl exec $PX_POD -n portworx -- /opt/pwx/bin/pxctl secrets set-cluster-key \
  --secret cluster-wide-secret-key
```

Example: encrypted Portworx StorageClass with PX-Security token auth

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-edb-secure
provisioner: pxd.portworx.com
parameters:
  repl: "2"
  secure: "true"
  io_profile: "db_remote"
  csi.storage.k8s.io/provisioner-secret-name: px-user-token
  csi.storage.k8s.io/provisioner-secret-namespace: portworx
  csi.storage.k8s.io/node-publish-secret-name: px-user-token
  csi.storage.k8s.io/node-publish-secret-namespace: portworx
  csi.storage.k8s.io/controller-expand-secret-name: px-user-token
  csi.storage.k8s.io/controller-expand-secret-namespace: portworx
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
```

Namespace-scoped security is the next refinement. Instead of relying on one global token or one global secret everywhere, Portworx can resolve CSI secrets dynamically from the PVC namespace. That allows each namespace to own its own token or secret material, which is the right model when the platform needs tenant isolation between database teams, application namespaces, or environments such as dev, pre-production, and production. In operational terms, this is how namespace-level RBAC and namespace-level storage security align cleanly in Portworx.

Example: namespace-scoped secure StorageClass

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-edb-secure-ns
provisioner: pxd.portworx.com
parameters:
  repl: "2"
  secure: "true"
  io_profile: "db_remote"
  csi.storage.k8s.io/provisioner-secret-name: ${pvc.name}
  csi.storage.k8s.io/provisioner-secret-namespace: ${pvc.namespace}
  csi.storage.k8s.io/node-publish-secret-name: ${pvc.name}
  csi.storage.k8s.io/node-publish-secret-namespace: ${pvc.namespace}
allowVolumeExpansion: true
```

When the requirement is externalized key management instead of Kubernetes Secrets, Portworx can also use an external secrets provider and a `secret_key` reference directly in the StorageClass. That model is appropriate when the platform standardizes on Vault, cloud KMS, or another centralized key manager rather than keeping storage encryption secrets inside Kubernetes.

FlashArray encryption and backend LUN encryption

FlashArray changes the design discussion because encryption at the backend storage layer is already built in. FlashArray encrypts all data at rest with AES-256, maintains full data reduction, and does not require separate host-side or volume-by-volume key management to deliver array-level encryption. FlashArray uses built-in, always-on encryption at rest, with no separate per-volume enablement workflow for administrators. That makes FlashArray a strong default encryption boundary for Portworx cloud drives, direct-access volumes, and backend LUNs used by the Portworx storage pools.

This matters for EDB because Portworx volume encryption and array encryption are not interchangeable from an efficiency perspective. If the primary goal is to maximize FlashArray deduplication and compression, array-level encryption is the better first choice. In that design, the backend LUNs remain encrypted by FlashArray, while Portworx volume encryption can be left disabled unless there is a specific requirement for a second cryptographic boundary at the Portworx layer. This is the cleanest way to preserve array-level data reduction while still keeping the data encrypted at rest.

Verify FlashArray data-at-rest encryption

```
purearray encryption list --data-at-rest
purearray encryption list --module-version
```

The expected result is that data-at-rest encryption reports as enabled and the encryption module version is visible to the operator. This is the correct operational check when the design relies on FlashArray encryption as the primary at-rest control for the backend storage layer.

Hardening FlashArray with KMIP and Rapid Data Locking

For environments that need stronger physical security than built-in encryption alone, FlashArray supports Rapid Data Locking through KMIP or smart cards. KMIP-backed RDL introduces a secondary, user-controlled key that must be available to unlock the array's flash modules on power-on. RDL can be enabled during installation or later, but once it is enabled it is permanent. This is the correct hardening model when the security requirement includes protection against whole-array theft or loss of physical custody.

The operational workflow is straightforward. Register the FlashArray with the KMIP appliance, exchange certificates between the array and the KMIP server, enable Rapid Data Locking, and then verify the state under `Settings -> System -> Rapid Data Locking`. Once configured, the array depends on the external KMIP key to unlock the flash modules after restart. If that key is revoked and the array is powered off, the data cannot be unlocked.

Recommended design guidance

For EDB on Portworx, the security model should be layered and intentional:

- Use EDB TLS for all client and inter-node database communication, and use explicit certificate management for PGD.
- Use EDB TDE (Transparent Data Encryption) when database-level encryption is a compliance or threat-model requirement, not as a substitute for transport security.
- Use PX-Security and secure StorageClasses when storage-layer RBAC and Portworx-managed volume encryption are required.
- Use namespace-scoped secrets and tokens when the platform must isolate database teams or tenants operationally.
- Use FlashArray data-at-rest encryption as the preferred backend encryption boundary when preserving array-level deduplication is the priority.
- Add KMIP-backed Rapid Data Locking on FlashArray when the design must also address physical loss of the array itself.

The correct production design is therefore not "encrypt everything in every layer by default." The correct design is to apply encryption at the layer that satisfies the requirement with the least operational and performance penalty, then add higher-layer controls only where the threat model or compliance posture actually requires them.

Optimize EDB for Better Performance

Optimizing EDB on Portworx requires treating PostgreSQL configuration, Kubernetes resource sizing, and storage policy as one performance design problem. PostgreSQL pods must be provisioned with sufficient CPU and memory so they do not become CPU-throttled or memory constrained under load, and EDB guidance is explicit that PostgreSQL resource settings should be aligned with the pod resource model. EDB also recommends benchmarking the chosen storage class before production rather than assuming the default class is appropriate for database workloads.

At the Portworx layer, the StorageClass is the first major control point. For most production EDB deployments, `repl: "2"` remains the best starting point because it provides storage-layer high availability without the additional write-path coordination and capacity overhead of `repl: "3"` on every database volume. When paired with PostgreSQL replication of 2, this creates a balanced design in which Portworx protects the volume and PostgreSQL protects service continuity at the database layer. For the primary data path, `io_profile: "db_remote"` is typically the correct default for replicated EDB volumes because it is designed for `repl >= 2` and reduces sync overhead by coalescing flush-heavy database writes across replicas.

One of the most important design choices is to separate PGDATA from WAL. EDB supports a dedicated WAL volume through `.spec.walStorage`, and the public EDB documentation calls out three concrete benefits: improved I/O parallelism, finer control over storage sizing and tuning, and stronger reliability because exhaustion of PGDATA space does not interfere with WAL writing. In a Portworx design, that should normally translate into two explicit StorageClasses: one for PGDATA and one for WAL. A common production pattern is to place PGDATA on a Portworx class using `repl: "2"` and `io_profile: "db_remote"`, while WAL uses a dedicated Portworx class with `repl: "2"`, `journal: "true"`, and, where validated by testing, `io_profile: "journal"` to optimize low-latency, sync-heavy log writes.

Storage pool selection also influences database behavior. Portworx supports `priority_io`, which allows volumes to be provisioned on higher-priority storage pools. This is useful when database volumes must land on the best-performing backing media rather than sharing generic pools with less latency-sensitive workloads. In the same way, volume placement strategies can be used to influence where PGDATA and WAL PVCs are created so that they align with the intended performance tiers, failure domains, and storage-node topology.

Noisy-neighbor control should also be part of the design. Portworx Application I/O Control allows IOPS and bandwidth limits to be applied through a StorageClass or directly through `pxctl`, making it possible to cap less important workloads that would otherwise compete with database traffic. In most EDB environments, the preferred use of Application I/O Control is to constrain adjacent non-database workloads rather than throttle PostgreSQL itself, thereby protecting database headroom during periods of shared-cluster contention.

Pod placement matters just as much as storage policy. Portworx uses Stork to improve hyperconvergence by prioritizing nodes that already host a replica of the volume being mounted. That reduces remote access overhead and improves restart efficiency after pod rescheduling. Where strict locality is required, the annotation `stork.libopenstorage.org/preferLocalNodeOnly: "true"` can be used so that PostgreSQL pods run only on nodes with a local replica, assuming adequate CPU and memory are available on those nodes.

The resulting performance model is straightforward. Size PostgreSQL pods correctly, separate WAL from PGDATA, bind each path to an explicit Portworx StorageClass, place database volumes on the highest-value storage pools, use Application I/O Control to suppress noisy neighbors, and rely on Stork plus placement strategy to keep PostgreSQL close to its data. That combination improves throughput, reduces latency variability, and preserves high availability without pushing the design into an inefficient over-replicated storage model.

Example: Portworx StorageClasses for PGDATA and WAL

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-edb-data
provisioner: pxd.portworx.com
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
parameters:
  repl: "2"
  io_profile: "db_remote"
  priority_io: "high"
  secure: "true"
  fs: "ext4"
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-edb-wal
provisioner: pxd.portworx.com
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer
parameters:
  repl: "2"
  io_profile: "journal"
  journal: "true"
  priority_io: "high"
  secure: "true"
  fs: "ext4"
```

Example: EDB Cluster using dedicated Portworx classes for data and WAL

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: edb-performance-cluster
spec:
  instances: 3
  resources:
    requests:
      cpu: "4"
      memory: 16Gi
    limits:
      cpu: "4"
      memory: 16Gi
  storage:
    storageClass: px-edb-data
    size: 500Gi
  walStorage:
    storageClass: px-edb-wal
    size: 100Gi
  postgresql:
    parameters:
      wal_keep_size: "4GB"
      max_replication_slots: "32"
      shared_buffers: "4GB"
      ssl_min_protocol_version: "TLSv1.3"
      ssl_max_protocol_version: "TLSv1.3"
```

Stork hyperconverges PostgreSQL pods with Portworx volume replicas so data access remains local whenever possible. That reduces remote I/O, lowers steady-state latency, and improves restart behavior after rescheduling because the database pod is preferentially placed on a node that already hosts a replica of the attached volume.

Example: Volume placement strategies for PGDATA and WAL

```
apiVersion: portworx.io/v1beta2
kind: VolumePlacementStrategy
metadata:
  name: edb-pgdata-placement
spec:
  replicaAffinity:
    - matchExpressions:
        - key: pool-type
          operator: In
          values:
            - database
  replicaAntiAffinity:
    - topologyKey: topology.kubernetes.io/zone
---
apiVersion: portworx.io/v1beta2
kind: VolumePlacementStrategy
metadata:
  name: edb-wal-placement
spec:
  replicaAffinity:
    - matchExpressions:
        - key: pool-type
          operator: In
          values:
            - wal
  replicaAntiAffinity:
    - topologyKey: topology.kubernetes.io/zone
```

Example: StorageClasses linked to placement strategies

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-edb-data
provisioner: pxd.portworx.com
parameters:
  repl: "2"
  io_profile: "db_remote"
  priority_io: "high"
  placement_strategy: "edb-pgdata-placement"
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-edb-wal
provisioner: pxd.portworx.com
parameters:
  repl: "2"
  io_profile: "journal"
  journal: "true"
  priority_io: "high"
  placement_strategy: "edb-wal-placement"
```

This pattern makes the intent explicit. PGDATA can be directed to database-oriented pools, while WAL can be directed to lower-latency pools intended for log-heavy write behavior. Because the strategy is attached to the StorageClass, every EDB-managed PVC created from that class inherits the same placement policy automatically.

Example: Application I/O Control for neighboring workloads

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: px-general-throttled
provisioner: pxd.portworx.com
parameters:
  repl: "2"
  io_profile: "auto"
  io_throttle_rd_iops: "1024"
  io_throttle_wr_bw: "50"
```

Validation commands

```
kubectl get pvc
kubectl describe pvc <pgdata-pvc>
kubectl exec -it <px-pod> -n kube-system -- /opt/pwx/bin/pxctl volume list
kubectl exec -it <px-pod> -n kube-system -- /opt/pwx/bin/pxctl volume inspect <volume-id>
kubectl exec -it <postgres-pod> -- df -h /var/lib/postgresql/data /var/lib/postgresql/wal
```

Day 2 Operations Guidance

For EDB on Kubernetes, Day 2 operations include the ongoing work required to keep the platform performant, recoverable, secure, and cost-efficient as data volume, workload intensity, and service-level expectations increase. That includes managing PGDATA and WAL growth, tuning WAL retention and replication behavior, maintaining backup and recovery posture, adjusting storage policy as workload patterns evolve, and ensuring the platform continues to meet both operational and business requirements.

These operations matter because the database and storage layers do not grow in the same way or at the same pace. WAL and PGDATA behave differently, consume capacity differently, and impose different performance demands on the underlying platform. Separating them gives the platform team precise control over sizing, monitoring, and tuning. StorageClass selection must also be validated against the actual workload before production, because the default class is rarely the correct long-term policy for a high-value database platform.

Portworx simplifies Day 2 operations by turning storage lifecycle management into policy-driven automation. Autopilot can monitor PVC consumption and automatically resize volumes when utilization crosses defined thresholds. That allows the platform to maintain safe operating headroom for EDB volumes without relying on reactive manual intervention. In practice, this is one of the most important operational controls in the environment, because database growth is continuous while manual storage management is inconsistent, slow, and error-prone.

Autopilot rules should be scoped with precise labels rather than generic selectors. EDB environments typically include multiple storage consumers with different growth patterns and operational importance, including PGDATA, WAL, backup-related volumes, and non-database workloads. A broad selector such as `app: postgres` is not sufficiently precise for production automation. Labels such as `app.kubernetes.io/name: edb-postgres` combined with a storage-role label such as `portworx.io/workload-class: pgdata` or `portworx.io/workload-class: wal` ensure that each rule applies only to the intended volume class. This allows PGDATA and WAL to scale independently and keeps automation aligned with how the database actually consumes storage.

Example: Autopilot rule for EDB PGDATA PVC growth

```
apiVersion: autopilot.libopenstorage.org/v1alpha1
kind: AutopilotRule
metadata:
  name: edb-pgdata-grow
spec:
  namespaceSelector:
    matchLabels:
      data-platform: edb
  selector:
    matchLabels:
      app.kubernetes.io/name: edb-postgres
      portworx.io/workload-class: pgdata
  conditions:
    expressions:
      - key: "100 * (px_volume_usage_bytes / px_volume_capacity_bytes)"
        operator: Gt
        values:
          - "75"
  actions:
    - name: openstorage.io.action.volume/resize
      params:
        scalepercentage: "50"
        maxsize: "4Ti"
```

A separate rule should be created for WAL volumes using a distinct label such as `portworx.io/workload-class: wal`. That separation is operationally important because WAL growth is often burst-driven and tied directly to checkpointing, replication, and recovery behavior. PGDATA growth is usually steadier and capacity-driven. Treating them as separate storage classes and separate automation targets produces a more stable and predictable operating model.

Portworx also improves Day 2 operations below the PVC layer. When the cluster uses cloud drives, Autopilot can expand the underlying storage pools as free capacity drops below the defined threshold. This enables the platform team to provision the initial storage footprint for current demand, then grow the storage estate incrementally as database usage increases. That is the correct financial and operational model for most EDB environments. It avoids tying up capital or cloud spend in unused storage on day 0, while still preserving the ability to scale without service disruption.

This model directly improves cost efficiency. Instead of provisioning the full projected storage requirement at the start of the project, the platform can begin with a smaller, right-sized storage pool on the EDB storage nodes and let Portworx add capacity only when the cluster actually needs it. The result is a storage footprint that grows with real demand rather than with forecast uncertainty. Over time, that reduces overprovisioning, improves infrastructure utilization, and aligns storage investment with actual database consumption.

Example: Autopilot rule for cloud-drive storage-pool expansion

```
apiVersion: autopilot.libopenstorage.org/v1alpha1
kind: AutopilotRule
metadata:
  name: edb-pool-expand
spec:
  enforcement: required
  conditions:
    expressions:
      - key: "100 * (px_pool_stats_available_bytes / px_pool_stats_total_bytes)"
        operator: Lt
        values:
          - "50"
      - key: "px_pool_stats_total_bytes / (1024*1024*1024)"
        operator: Lt
        values:
          - "4000"
  actions:
    - name: openstorage.io.action.storagepool/expand
      params:
        scalepercentage: "50"
        scaletype: "add-disk"
```

In cloud-drive deployments, this rule allows Portworx to add capacity to the storage pools when available capacity falls below the defined threshold while still enforcing an upper growth boundary. This creates a controlled, policy-based expansion model instead of forcing the platform team to overbuild the cluster or repeatedly perform manual storage interventions as the database grows.

EDB owns the database-layer concerns such as topology, replication, and recovery behavior. Portworx automates the storage-layer concerns that otherwise become recurring operational bottlenecks: volume growth, pool expansion, and ongoing capacity control. That division of responsibility creates a stronger operating model for the platform and allows the team to scale the environment with less risk, less manual toil, and better financial discipline.

Example: Validation commands

```
kubectl get autopilotrules
kubectl get events --field-selector involvedObject.kind=AutopilotRule,involvedObject.name=edb-pgdata-grow
--all-namespaces --sort-by .lastTimestamp
kubectl get events --field-selector involvedObject.kind=AutopilotRule,involvedObject.name=edb-pool-expand
--all-namespaces --sort-by .lastTimestamp
kubectl exec -it <px-pod> -n kube-system -- /opt/pwx/bin/pxctl service pool show
```

Data Protection

Disaster recovery and backup serve different purposes and must be designed as separate layers in the platform. Disaster recovery is focused on restoring service after a site, cluster, or infrastructure failure. Backup is focused on recovering data after deletion, corruption, operator error, or ransomware. A production EDB design requires both. One layer preserves application continuity and recovery time. The other preserves durable recovery points and long-term data retention.

EDB Postgres Distributed and the role of PGD

EDB Postgres Distributed improves database availability and geographic resilience, but it is not a backup system. PGD provides distributed replication, failover domains, and multi-site data movement across PostgreSQL groups. That makes it a strong HA and DR technology for keeping a database service running across locations, but it does not replace retained backup copies for corruption, ransomware, or operator mistakes.

That distinction is critical in a production architecture. A surviving PGD node or surviving PGD group can often be used to recover from a node or site event more quickly than restoring from backup, but backup is still required for point-in-time recovery, retained recovery points, and recovery from logical corruption that has already been replicated through the PGD topology.

PGD backup design also differs from ordinary single-node PostgreSQL backup design. In a distributed system, data changes can originate from multiple nodes, which means backup and PITR planning must account for multi-origin recovery semantics rather than assuming a single WAL origin. In practice, that means PGD should be treated as the database-layer DR capability, while backup remains a separate control for retention and recovery.

Portworx Disaster Recovery

Portworx DR provides the Kubernetes and volume mobility layer that sits below the database. It protects Kubernetes objects and persistent volumes across clusters and complements PGD rather than competing with it. PGD protects the distributed PostgreSQL service. Portworx DR protects the Kubernetes application footprint and the Portworx-backed volumes that support it.

Portworx supports two primary DR patterns:

Mode	Architecture	Recovery objective	Best fit
Asynchronous DR	Two independent Portworx clusters paired with `ClusterPair` and scheduled migrations	**Non-zero RPO** based on schedule interval	Cross-cluster DR, regional recovery, migration between independent clusters
Synchronous DR	One stretched Portworx cluster across sites with synchronous replication	**RPO=0** when latency requirements are met	Metro-style active/passive site protection with zero data loss target

Table 7. Portworx Disaster Recovery Types

In asynchronous DR, each site runs its own Portworx cluster and Portworx moves application resources and volume data between clusters according to a migration schedule. This is the correct model when the recovery sites are independent Kubernetes clusters and the business can tolerate a schedule-based recovery point objective. Portworx guidance recommends a migration interval of at least 15 minutes and requires `startApplications: false` so the destination volumes are not brought online prematurely during recurring DR synchronization.

In synchronous DR, Portworx uses a stretched cluster across sites and replicates storage synchronously. This is the correct model when the environment can meet the network and latency requirements of less than 10ms RTT for metro replication and the design target is zero data loss. In this model, the storage layer is already mirrored, so scheduled migration is typically used to move Kubernetes resources while volume replication is handled underneath by the stretched storage fabric.

Note: *RPO 0 does not by itself establish end-to-end database consistency or application failover behavior. Portworx RPO is for acknowledging writes / data in the recovery site. Application failover will need to be handled separately through the use of storkctl and other automation.*

For EDB, the guidance is straightforward: use PGD for database-layer geographic resilience and database failover semantics, and use Portworx DR when the platform must also recover the Kubernetes application, PVC, and namespace layer in a controlled cross-cluster manner. These are complementary controls, not mutually exclusive ones.

Configure ClusterPair for asynchronous DR

A `ClusterPair` defines the trust and data-movement relationship between the source and destination clusters. For asynchronous DR, it also defines the object-store or NFS path used to transfer persistent volume data between clusters. Portworx supports multiple back ends for this transfer path, including S3-compatible object storage and NFS.

Example: create a unidirectional ClusterPair for async DR with S3

```
storkctl create clusterpair edb-dr-pair \
  --namespace kube-system \
  --src-kube-file /tmp/src-kubeconfig \
  --dest-kube-file /tmp/dest-kubeconfig \
  --mode async-dr \
  --provider s3 \
  --s3-endpoint s3.amazonaws.com \
  --s3-access-key <s3-access-key> \
  --s3-secret-key <s3-secret-key> \
  --s3-region us-east-1 \
  --bucket edb-dr-bucket \
  --unidirectional
```

If the Portworx API is exposed externally, the cluster pair can also include explicit source and destination endpoints. If an existing object-store location is already defined, that can be reused rather than creating a new one during the pairing workflow.

Example: verify ClusterPair status

```
storkctl get clusterpair -n kube-system
kubectl get clusterpair -n kube-system
```

A healthy asynchronous DR pair must show the cluster relationship as ready before migration schedules are created. If the cluster pair is not ready, scheduled migration will fail regardless of the schedule policy configuration.

Create migration schedules

A migration schedule defines how often Portworx moves Kubernetes resources and Portworx-backed volume data from the source cluster to the recovery cluster. For DR use cases, scheduled migration should be treated as a standing replication policy rather than as a one-time migration task.

Example: 15-minute DR schedule policy

```
apiVersion: stork.libopenstorage.org/v1alpha1
kind: SchedulePolicy
metadata:
  name: edb-15min-dr
  namespace: kube-system
policy:
  interval:
    intervalMinutes: 15
```

Example: MigrationSchedule for an EDB namespace

```
apiVersion: stork.libopenstorage.org/v1alpha1
kind: MigrationSchedule
metadata:
  name: edb-prod-dr
  namespace: kube-system
spec:
  schedulePolicyName: edb-15min-dr
  template:
    spec:
      clusterPair: edb-dr-pair
      includeResources: true
      includeVolumes: true
      startApplications: false
      namespaces:
        - edb-prod
      autoSuspend: true
```

This is the correct asynchronous DR pattern for an EDB namespace. Kubernetes resources and Portworx volumes are moved on a recurring schedule, but the applications are not started on the target side until an actual disaster event occurs and determined to be the method of recovery. That prevents accidental split-brain behavior and preserves the destination cluster as a recovery target rather than an actively running second copy of the application.

Example: create MigrationSchedule from the CLI

```
storkctl create migrationschedule edb-prod-dr \
  --cluster-pair edb-dr-pair \
  --namespaces edb-prod \
  --schedule-policy-name edb-15min-dr \
  --start-applications=false
```

For synchronous DR, the scheduling pattern changes. The stretched Portworx cluster already replicates the storage layer synchronously, so migration schedules are typically used for Kubernetes resources only, with volume migration excluded. That is why sync DR workflows

commonly use `includeVolumes: false` or the equivalent CLI behavior.

PX-Backup and retained backup protection

PX-Backup is the backup layer in this architecture. It is the control that provides retained recovery points, cross-cluster restore capability, and the off-cluster copy required for a 3-2-1 style protection model. PGD is not a backup platform. Portworx DR is not a backup platform. PX-Backup is the backup platform.

In practice, PX-Backup protects the Kubernetes application, its configuration, and its associated persistent data by writing backup data to an external backup location. That backup copy is what allows recovery after destructive events such as deletion, logical corruption, or ransomware, even when the active PGD topology or the active DR topology is no longer trustworthy.

This is where the architectural distinction matters:

- **PGD** keeps the database service available across sites.
- **Portworx DR** moves Kubernetes resources and volumes between clusters for DR recovery.
- **PX-Backup** provides retained, restorable recovery points and immutable backup protection.

Object Lock and ransomware protection

PX-Backup supports S3 Object Lock for all S3-compliant object stores. Object Lock uses a bucket-level immutability model to prevent deletion or modification of protected backup objects during the retention period. This is the mechanism that turns a standard backup repository into a ransomware-resistant backup target.

Portworx supports both **Governance** and **Compliance** retention modes. Governance mode prevents deletion or overwriting unless the caller has special bypass permissions. Compliance mode is stricter and prevents deletion of protected objects even by highly privileged users during the retention period.

For object-lock-enabled backups, the protection period defines the backup immutability window, and Portworx calculates the retention period as **protection period + 6 days of buffer**. Backup locations used for object lock should be configured with a minimum retention period of **7 days** or greater.

Example: create an object-lock-enabled S3 bucket

```
aws s3api create-bucket \
  --bucket edb-pxbackup-immutable \
  --region us-east-1 \
  --object-lock-enabled-for-bucket

aws s3api put-object-lock-configuration \
  --bucket edb-pxbackup-immutable \
  --object-lock-configuration '{
    "ObjectLockEnabled": "Enabled",
    "Rule": {
      "DefaultRetention": {
        "Mode": "COMPLIANCE",
        "Days": 30
      }
    }
  }'
```

The bucket must be created with Object Lock enabled from the start. The retention mode and retention period must then be defined before the bucket is used as an immutable backup target.

Required IAM permissions for Object Lock

```
s3:GetBucketObjectLockConfiguration
s3:GetObjectLegalHold
s3:GetObjectRetention
s3:BypassGovernanceRetention
s3:PutBucketObjectLockConfiguration
s3:PutObjectLegalHold
s3:PutObjectRetention
```

Example: create an immutable backup in PX-Backup

Once the object-lock-enabled backup location has been defined in PX-Backup, manual or scheduled backups can be created against that location. Immutable scheduled backups require an object-lock-enabled schedule policy, while immutable manual backups can be created directly against the protected backup location.

The operational outcome is significant. A protected PX-Backup copy cannot be deleted until the retention period expires. That creates a recovery layer that is independent from the live Kubernetes cluster and independent from the active PGD topology. It is the control that allows the platform team to recover after destructive events that replication alone cannot solve.

Recommended protection models for EDB on Portworx

The correct production design is layered:

1. **PGD** for database-layer replication, regional availability, and distributed failover.
2. **Portworx DR** for cross-cluster Kubernetes and volume recovery, using async or sync DR according to the RPO target.
3. **PX-Backup** for retained backup copies, cross-cluster restore, and immutable ransomware protection through Object Lock.

That architecture separates service continuity from retained data protection. It gives EDB the distributed database behavior it needs, gives Kubernetes the cross-cluster DR controls it needs, and gives the platform an immutable backup layer that remains recoverable even after corruption, deletion, or ransomware.

Summary

EDB with Portworx creates a database platform for Kubernetes that is designed for production. EDB supplies the operator-driven PostgreSQL control plane, including single-cluster lifecycle management and multi-region distribution through **PGD**, while Portworx supplies the storage layer that makes those database services practical at scale through persistent volume management, storage-layer high availability, performance tuning, and data mobility.

The result is a platform that standardizes how PostgreSQL is deployed, protected, and operated across Kubernetes environments, with explicit control over PGDATA, WAL, replication, encryption, performance policy, and recovery behavior.

In practice, EDB with Portworx gives platform teams a repeatable way to run PostgreSQL on Kubernetes with stronger performance, clearer operational boundaries, and more resilient storage and data-protection outcomes than a database deployment built on basic persistent volumes alone.