TECHNICAL WHITE PAPER

Apache Kafka on Kubernetes with Portworx

Operate and scale Apache Kafka seamlessly on Kubernetes with the Portworx® platform.

Contents

Introduction

Apache Kafka is an open-source distributed event platform used to stream real-time analytics, data pipelines, and data integration to internet of things (IoT), AI/ML, and numerous other applications. Every day, organizations are moving away from legacy systems toward adopting cloud-native microservices on a platform like Kubernetes for ease of use, agility, scalability, and productivity. However, organizations face the challenge of moving their mission-critical streaming applications like Apache Kafka into Kubernetes as Kafka requires seamless scalability, persistent storage with data protection, security, capacity management, and data mobility.

Portworx by Pure Storage

Portworx by Pure Storage® is the Kubernetes data services platform that businesses trust to run mission-critical production applications in containers. Only Portworx provides a fully integrated solution for persistent storage, data protection, disaster recovery, data security, cross-cloud and data migrations, and automated capacity management for applications running on Kubernetes. As a result, Portworx is the #1 most used Kubernetes data services platform by Global 2000 companies.

Running Apache Kafka on Kubernetes with Portworx enables you to:

- Gain operational efficiency in managing a scalable Kafka cluster through improved broker failover time and fully automated storage capacity expansion
- Enhance security with encryption-at-rest for Kafka broker data
- Migrate Kafka applications along with data between environments like prod, dev, test, and across cloud providers or data centers
- Gain additional storage savings through deduplication and compression on Pure Storage FlashArray™

Objectives

This paper will show you the advantages of running Apache Kafka on Kubernetes using Portworx andprovide best practices for this solution. This paper uses Pure Storage FlashArray as well as local non-volatile memory express (NVMe) drives as the on-premises storage destination for Portworx. The reference architecture presented is for Kafka, system, storage, and Kubernetes administrators who want to build scalable real-time event streaming applications with Kafka on-premises or in the cloud on Kubernetes with Portworx.

Technology Overview

Apache Kafka

Kafka is a message and streaming platform that is designed to handle data and events processing in a distributed way with higher throughput and lower latency with high availability. It supports message queues, stream processing, and publish-subscribe in one uniform model.

Kafka Cluster Architecture

Kafka brokers: A Kafka cluster is made up of one or more servers called *brokers*. Multiple brokers work together to form the Kafka cluster and achieve load balancing, high availability, and failover. Brokers use Apache Zookeeper for the management and coordination of the cluster. Each broker is capable of handling read and write requests reaching hundreds of thousands each second without any impact on performance. To achieve reliable failover, a minimum of three brokers should be used.

Kafka producers: Kafka producers are client applications that publish (write) events to one or more Kafka topics. They also serialize, compress, and load balance data among brokers through partitioning.

Kafka consumers: Kafka consumers are client applications that subscribe to (read and process) the events generated by producers. They read data by reading messages from the topics to which they subscribe. Consumers will belong to a consumer group and each consumer within a particular consumer group will have the responsibility for reading a subset of the partitions of each topic that it is subscribed to.

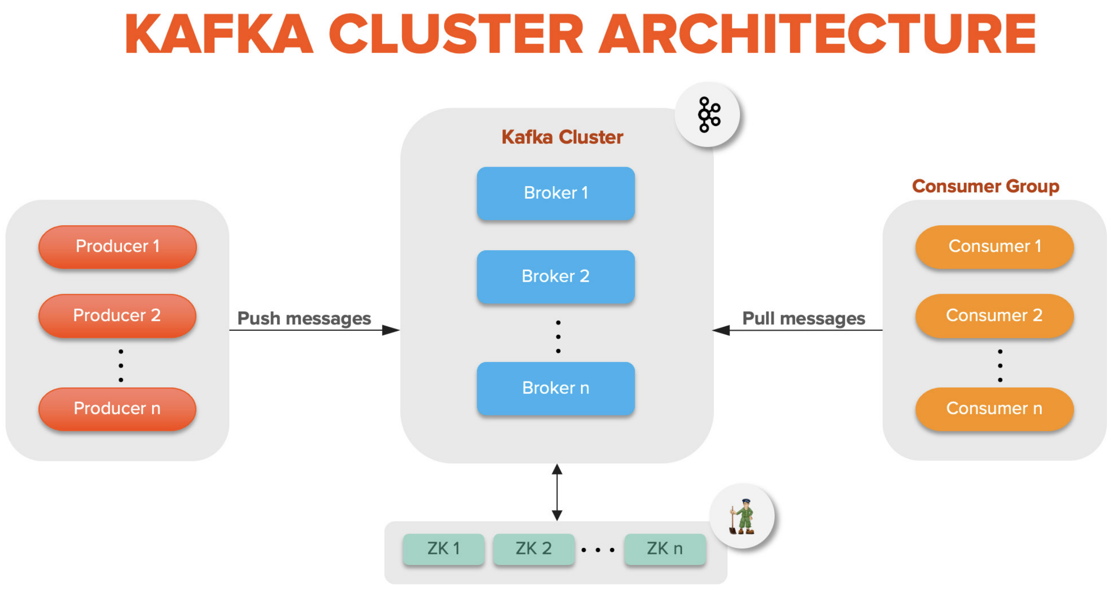


Figure 1. Kafka cluster architecture

Kafka Architecture Concepts

Kafka topics: Events or messages are logically organized and stored in topics. Producers publish messages to topics and consumers read messages from the topic they subscribed to. Topics are identified by unique names within a Kafka cluster, and there is no limit on the number of topics that can be created. Messages persist, so topics can be reprocessed for new purposes.

Kafka partitions: Topics are divided into partitions, and the partitions are spread across Kafka brokers. This distributed data placement is very important for scalability because it allows client applications to both read and write events from/to many brokers at the same time. In Kafka, the topic partition is the unit of parallelism. High availability of the Kafka data is achieved by replicating partitions across the brokers. Each partition has one server that acts as the *leader* and zero or more servers that act as *followers* based on the topic replication factor.

Topic replication factor: Topic replication is the key factor in designing resilient and highly available Kafka deployments. When a broker goes down, topic replicas on other surviving brokers will remain available to ensure that data is available as well as to avoid Kafka application failure and downtime. The replication factor is defined at the topic level, but it is effective at the partition level. For example, a replication factor of 2 will maintain two copies of a topic for every partition. Logically, the replication factor cannot be greater than the total number of brokers available in the cluster. A replica that is up to date with the leader of a partition is said to be an in-sync replica (ISR).

Consumer group: A Kafka consumer group is a set of consumers that work together to consume data from some topics. The partitions of all topics are divided among the consumers in the group.

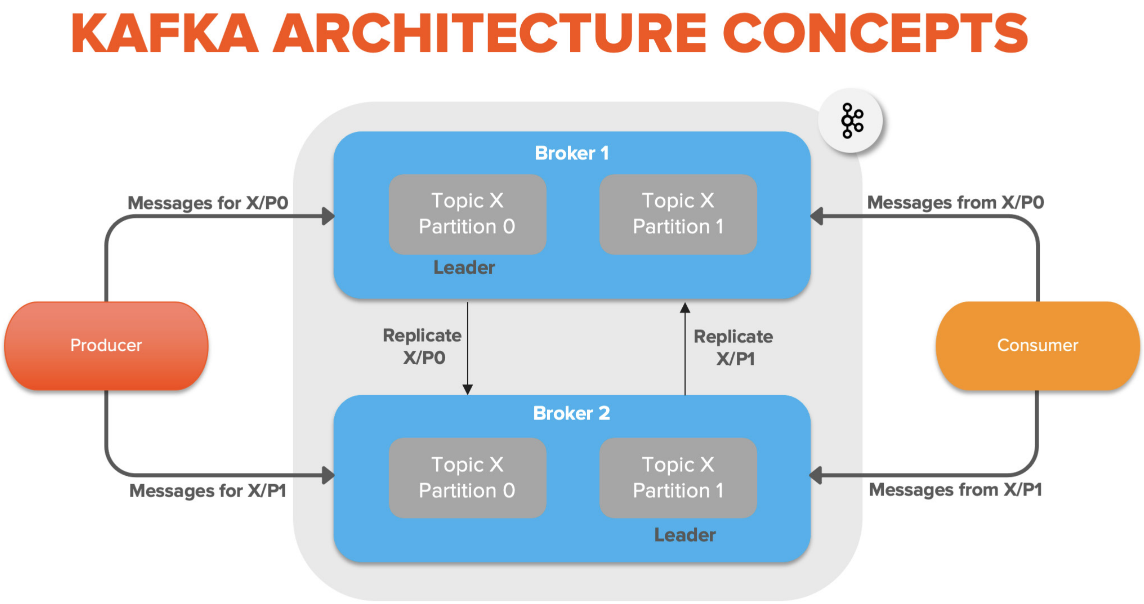


Figure 2. Kafka architecture concepts

In summary:

- A Kafka cluster stores events in categories called topics.
- A cluster consists of one or more servers called brokers.
- Each topic maintains a partitioned log.
- A topic may have many partitions that all act as the unit of parallelism.
- Brokers can host either one or zero replicas for each partition.
- Each partition includes one leader replica and zero or more follower replicas based on the topic replication factor.
- Each of a partition’s replicas must be on a different broker.

Portworx

Portworx is the container-native, software-defined storage (SDS) solution that enables higher data availability, data security, backup, and disaster recovery for container-based applications running on container orchestration platforms like Kubernetes running on-prem or across clouds.

What Is Software-defined Storage?

At a high level, a software-defined storage solution abstracts storage devices of various types and sizes that are attached to worker nodes in the Kubernetes cluster. All worker nodes with available storage on locally attached drives are added as nodes to the storage cluster. In this storage cluster, the physical storage is virtualized and presented as a virtual storage pool to the user and the SDS software manages this storage cluster..

An SDS offers a centralized storage platform like that of traditional storage appliances like storage area network (SAN) or network attached storage (NAS) by abstracting the storage resources from the underlying hardware platform for greater flexibility, scalability, and efficiency. SDS enables data service options such as thin provisioning, snapshots, backup, replication, and deduplication. Components like SDS and software-defined networking (SDN) form the basis for the containerization that powers container orchestration platforms such as Kubernetes.

In contrast, the legacy model of locally attached storage or direct-attached storage (DAS) offers control over the data and performance but lacks every other aspect, including data availability, accessibility, and storage expansion. Storage administrators have to configure RAID on top of the locally attached storage to get higher availability of the data in case of a drive failure. The lack of features, along with increased operational requirements, makes standalone DAS unsuitable for modern applications on containers because those require storage that can support scalability and availability.

How Does Portworx Work?

Portworx is an SDS system optimized for container environments. It brings all of the benefits of traditional SDS—such as storage virtualization and pooling—by aggregating all of the available storage that is attached to the worker nodes, and it creates a unified, persistent storage layer for stateful applications like Kafka. By using volume replication of each container-level volume across multiple worker nodes, Portworx ensures data persistence, data availability, and accessibility. Portworx also comes with integrated Storage Orchestrator Runtime for Kubernetes (STORK) that allows applications to take advantage of scheduler extenders through storage-aware scheduling via Kubernetes. Using STORK, you can ensure optimal placement of volumes in the cluster.

Portworx Enterprise Data Services Platform

Enterprise applications running on Kubernetes have non-negotiable business requirements like high availability, data security, backup and disaster recovery, strict performance service level agreements (SLAs), and hybrid/multi-cloud operations.

Portworx Enterprise was designed for exactly these applications. It includes the most scalable and robust capabilities (as seen in Figure 1) of the Portworx Storage Platform for Kubernetes with the ability to add on Portworx-DR and Portworx-Backup for even greater levels of data protection. You can scale up to 1000 nodes and 1 million volumes per cluster.

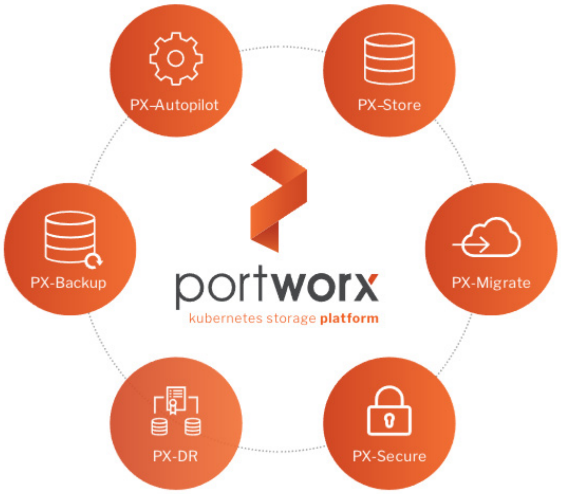


Figure 3 Portworx Enterprise capabilities

In this white paper, we'll be using PX-Store, PX-Autopilot, PX-Migrate, and PX-Secure.

PX-Store

PX-Store is the foundation of the Portworx Enterprise Data Services Platform for Kubernetes. Built from the ground up for containers, PX-Store provides cloud-native storage for applications running in the cloud, on-prem, and in hybrid/multi-cloud environments. PX-Store provides the reliability, performance, and data protection you'd expect from an enterprise storage company, but delivered as a container and managed 100% via Kubernetes and other leading container platforms. It takes your underlying hardware, even an existing SAN or NAS, and turns it into a cluster-wide storage pool for all your applications. PX-Store also provides built-in high availability (HA) for all stateful applications and allows failed pods to recover in seconds.

PX-Autopilot

PX-Autopilot for Capacity Management allows you to stop over-provisioning capacity in the cloud or on-prem so you can reduce your storage bill while offering automation to dynamically resize if required. It gives you:

- The ability to automatically resize individual container volumes or your entire storage pools
- A rules-based engine with customization capabilities so you can optimize your apps based on performance requirements
- Integration with Amazon EBS, Google PD, and Azure Block Storage

PX-Migrate

PX-Migrate delivers true multi-hybrid cloud Kubernetes with complete portability for applications, configurations, and data across clouds and data centers. Natively integrated with kubectl, you can easily move, upgrade, and migrate stateful applications with a single command. With PX-Migrate, you can:

- Back up your applications to another cloud with a single command. Ensure portability and access with a single unified snapshot format that is never hidden in proprietary formats.
- Ensure consistency and viable backups with application-aware backup. You can set up automatically coordinated distributed backups for multi-container applications, making application-consistent snapshots easy.
- Enable multi-cloud/multi-cluster application migrations.

PX-Secure

Your data is the lifeblood of your business and applications. PX-Secure offers container data security without compromise. PX-Secure ensures its security and integrity with built-in Kubernetes tooling. It also enables both cluster-wide and container-granular encryption. PX-Secure includes:

- Cluster-wide encryption and container-granular or storage-class based bring your own key (BYOK) encryption
- Role-based access control for authorization, authentication, and ownership
- Integrations with Active Directory and LDAP

Pure Storage FlashArray

FlashArray is the world's first 100% all-flash end-to-end NVMe and NVMe-oF array, ideal for the most demanding enterprise performance requirements. It's part of a Modern Data Experience™, delivering breakthroughs in speed, simplicity, flexibility, and consolidation. FlashArray is ideal for shared storage deployments from the departmental level to large enterprises, and for high performance and mission-critical applications. In a world of fast, pervasive networking, ubiquitous flash memory, and evolving scale-out application architecture, FlashArray provides customers with both networked and direct-attached storage in a single, shared architecture. With latency as low as 150μs, FlashArray brings new performance levels to mission-critical business applications and databases.

Figure 4. FlashArray //X90

From entry-level to enterprise workloads, FlashArray//X lets your organization accelerate the most critical applications. FlashArray//X delivers breakthroughs in performance, simplicity, and consolidation. It's ideal both for enterprise applications such as Microsoft SQL Server and for cloud-native, web-scale applications. FlashArray//X70 and //X90 support optional DirectMemory™ Cache, which uses Intel Optane storage class memory (SCM) to run database workloads at near-DRAM speeds. If extreme performance is a top priority, your organization can rely on FlashArray//X to deliver the low latency and high throughput that end users demand.

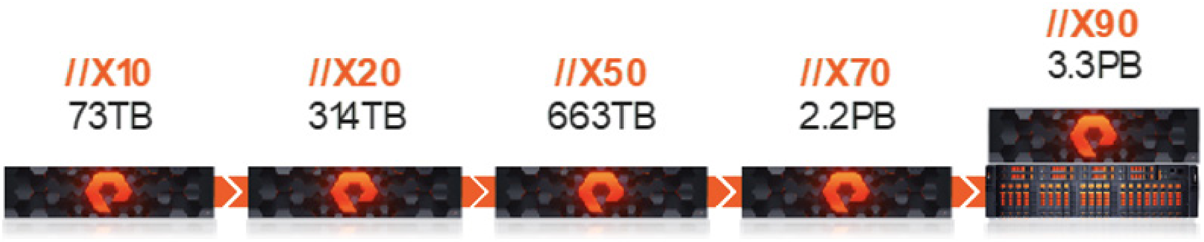


Figure 5. FlashArray delivers low latency and high throughput.

With Portworx Enterprise 2.8, it is easier to take advantage of the enterprise capabilities of FlashArray like all-flash performance, non-disruptive upgrades, inline encryption, and data reduction through deduplication and compression. FlashArray can be configured as a cloud storage provider within Portworx. This allows you to store your data on-premises with FlashArray while benefiting from Portworx cloud drive features, such as:

- Dynamic provisioning of volumes on FlashArray
- Cluster expansion by adding new drives or expanding existing ones
- Support for PX-Backup and Autopilot

Challenges with Kafka at Scale

Scalability

Data growth in organizations is enormous these days, and that growth will only accelerate as the data generated is not limited to residing within the data centers but on the edge, closer to physical assets. Hence, organizations constantly need to address their infrastructure scalability to support this massive growth. Supporting rapid data growth requires adding more servers, storage, and network bandwidth as well as scaling Kafka brokers. Scaling brokers requires automation to deploy the latest OS patches, required device drivers, and software libraries to run the application, network, and storage configuration, and so on. Many organizations manage hundreds of Kafka brokers within their clusters. An automated and orchestration platform that can eliminate most of these mundane tasks would be hugely beneficial to administrators.

Kafka Broker Failures

Kafka allows customizing the replication factor (RF) at a topic level to provide higher data availability in case of failures. The RF determines the Kafka topic's failure tolerance at a partition level. A topic-partition can tolerate a failure of RF-1 broker nodes. For example, a replication factor of 3 means a topic-partition can sustain up to two broker failures but three broker failures would result in the unavailability of that topic partition.

What happens when a Kafka broker goes down? The partition's availability now depends on the existence and status of other replicas. If the partition has no additional replicas, the partition becomes unavailable. If the partition has additional replicas that are in-sync, one of the in-sync replicas will become the interim partition leader. If the partition has additional replicas but if none of them are in sync, a choice has to be made: either wait for a partition leader to come back online while sacrificing availability or allow an out-of-sync replica to become the interim partition leader while sacrificing consistency. Once the broker comes back online, all its replicas will sync up with their partition leaders. Kafka's design principle assumes that a failed broker node will resume back in operation, and it doesn't create a new replica with the surviving nodes to meet the topic's replication factor. Instead, the partition stays under-replicated, which reduces data availability.

What happens when a Kafka broker fails permanently? The data hosted locally on that Kafka broker is not available anymore. The Kafka administrator can add a new broker and run the partition reassignment utility to rebalance the data across the available brokers while creating the missing replicas for the under-replicated partitions. This is a manual process, involves physical data movement over the network, and can be time-consuming, depending on the data sets. Consider the overhead this can cause for Kafka when running at scale.

Capacity Management

Rightsizing space usage in this ever-growing digital world is an ongoing challenge. Especially with a high-bandwidth data streaming platform like Kafka at scale, where the daily ingest can spike either seasonally or regularly, it is very critical to manage capacity. Most organizations have monitoring in place to keep space usage in check and take manual actions when the allocated space is exhausted. Any delay in addressing space usage can, unfortunately, cause an outage with the Kafka application. What if there is an option to automate the whole operational aspect of capacity management?

Portability

Maintaining the development, test, and staging environments of Kafka are common and standard operating procedures. Generally, these pre-production environments are created standalone and populated with synthetic data. What if the developers wanted to test the system with real data from production in an environment that has a similar application configuration to that of production? What if developers wanted this test system on the cloud with real data, while the production system remained on-premises? While the production data can be extracted through various means to be used in the development environment, getting it ported onto the cloud is not very straightforward and would involve various steps.

Security

A lot of organizations that deal with sensitive data want to encrypt the data that persists to disk (at-rest) due to compliance requirements. Kafka provides SSL/TLS encryption of data transferred between brokers and clients as well as consumers and producers. This enables encryption on the in-flight data passed along the network between the brokers and clients. This approach meets security requirements when data goes across security domains like internal networks to the public internet or partner networks, but it doesn't meet the physical storage security requirements like encryption-at-rest. What if the underlying data at the storage level is encrypted, which would meet the encryption-at-rest requirement and also offer additional storage authentication and authorization features?

The Solution: Kafka at Scale Using Kubernetes with Portworx

The solution to address the challenges with Kafka at scale is to use Kubernetes with Portworx by Pure Storage. Each subsection below addresses the challenges listed in earlier, in the same order.

Production-grade Container Orchestration with Portworx

Kubernetes is a portable, extensible, open-source container orchestration platform for managing containerized workloads such as Kafka. Containers enable applications to be broken into smaller, independent pieces that can be deployed and managed dynamically rather than as a monolithic stack. This allows for predictable application performance and resource utilization. Kubernetes provides a framework to run these containers on distributed systems resiliently and enables horizontal scaling.

Kubernetes offers portability and flexibility while working with virtually any type of container runtime. It is the market leader in the container orchestrator arena and the adoption has continuously grown in recent years. According to industry analyst firm Gartner, “By 2025, more than 85% of global organizations will be running containerized applications in production, which is a significant increase from fewer than 35% in 2019.”[1]

Most applications cannot run on Kubernetes alone because Kubernetes requires a container-native storage and data management solution to address the top barriers preventing wider Kubernetes adoption, including persistent storage, automated operations, and data mobility. Top of mind concerns also include backup, disaster recovery, and data security.

This is where Portworx comes in. Designed specifically for cloud-native applications, Portworx delivers the performance, reliability, and security you require from traditional enterprise storage, but it is built from the ground up for Kubernetes. Portworx is the most complete Kubernetes data services platform. GigaOm called Portworx “the gold standard for cloud-native Kubernetes storage for the enterprise.” With Portworx, you can run data services like Kafka in production at scale with all of the enterprise capabilities mentioned above.

Portworx is a software-defined storage solution for containers that provides container-granular storage, security, capacity management, backup, and disaster recovery features to containers running in distributed computing environments. Portworx can be tightly integrated with all major container orchestrators, such as Kubernetes or OpenShift, which makes it easy to use Portworx as a storage solution for containers in these environments.

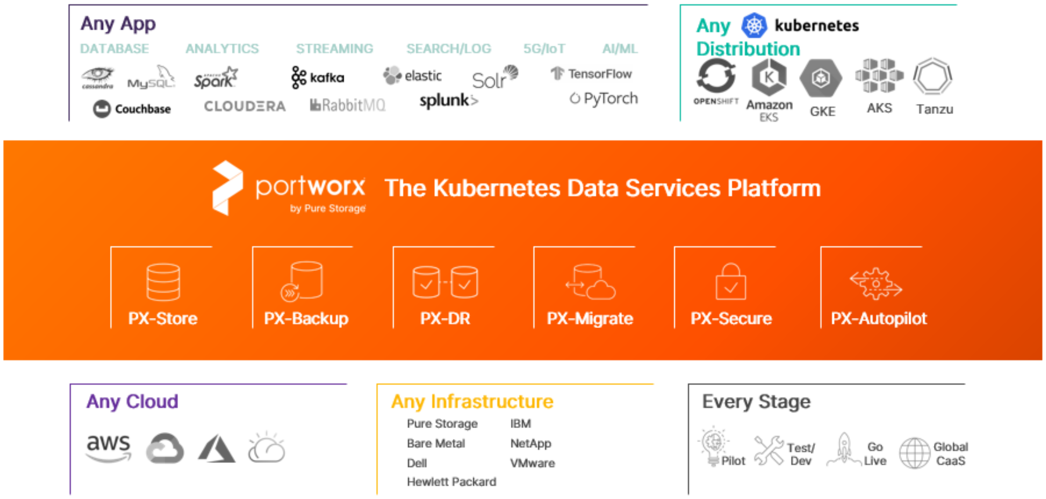


Figure 6. Portworx Enterprise Data Services Platform

StatefulSet with Portworx PX-Store

One of the major pain points expressed by Kafka administrators is the operational overhead with a *hard broker failure*. This requires them to add a new broker and run the partition reassignment script to rebuild the missing partition replicas while

[1] Gartner, Best Practices for Running Containers and Kubernetes in Production, Arun Chandrasekaran, 4 Aug 2020, ID G00730344

rebalancing the data across all the brokers. This activity involves physical data movement between the brokers, and it can be very time-consuming based on the total data set involved.

When a worker node that hosts the Kafka broker pod fails, it will be spawned on another worker node, but with a StatefulSet controllers, which have the sticky identity for each of their pods. If the new worker node doesn't have access to the same volume, it will fail to bring up the Kafka broker pod on a new worker node.

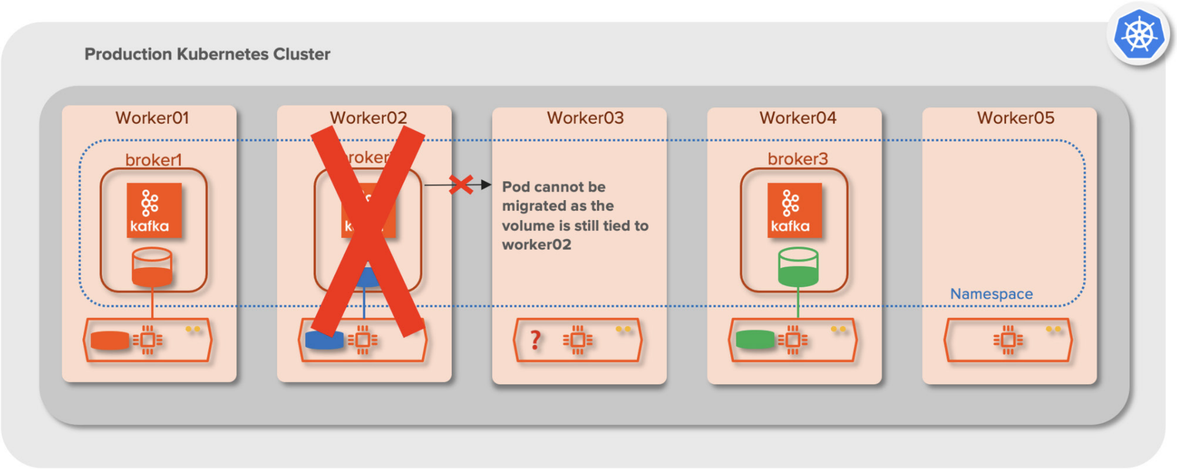

Production Kubernetes Cluster
Worker01
Worker02
Worker03
Worker04
Worker05
broker1
roker
broker3
kafka
kafka
Pod cannot be migrated as the volume is still tied to worker02
?
Namespace

Figure 7. Challenge with StatefulSet failure

This is where Portworx comes to the rescue. PX-Store is the foundation of Portworx Enterprise. PX-Store provides reliability, performance, and data protection on Kubernetes and other leading container platforms. PX-Store provides built-in HA for all stateful applications like Kafka, and it allows failed Kafka broker pods to recover in seconds.

Portworx can take local DAS (NVMe or HDD), SAN, or NAS and turn it into a cluster-wide storage pool for all container applications. Kubernetes users can allocate storage from this pool to create persistent volumes for applications running in the cluster. Portworx can ensure that this storage is highly available using replication. It can create volume replicas according to the user-defined replication setting and distribute those replicas evenly and dynamically across the Kubernetes cluster. For these replicas, Portworx uses synchronous replication, where each write is automatically synchronized with the replicas. This ensures data consistency among them. Volume replicas can be accessed by Portworx from any node where Portworx runs.

Now, when a worker node failure causes a Kafka broker to go down, we don't have to worry if the underlying server will ever come up so Kafka can synchronize the data to get the under-replicated partitions back to normal. With Portworx, any node-level failure will cause the Kafka broker to be spawned on the available worker nodes with the same volume, automatically. This eliminates any partition reassignment activity needed if the worker node fails for good.

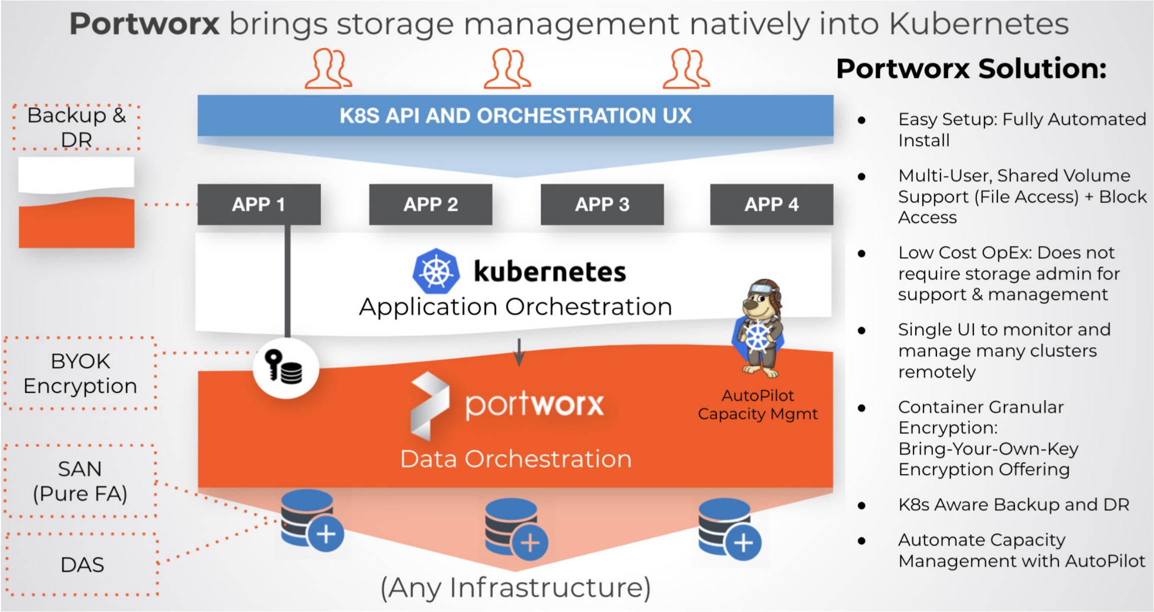

Portworx brings storage management natively into Kubernetes
Portworx Solution:
Easy Setup: Fully Automated Install
Multi-User, Shared Volume Support (File Access) + Block Access
Low Cost OpEx: Does not require storage admin for support & management
Single UI to monitor and manage many clusters remotely
Container Granular Encryption: Bring-Your-Own-Key Encryption Offering
K8s Aware Backup and DR
Automate Capacity Management with AutoPilot
K8S API AND ORCHESTRATION UX
APP 1
APP 2
APP 3
APP 4
Backup & DR
kubernetes
Application Orchestration
BYOK Encryption
portworx
Data Orchestration
AutoPilot Capacity Mgmt
SAN (Pure FA)
DAS
(Any Infrastructure)

Figure 8. Portworx Solution

PX-Autopilot for Capacity Management

Portworx Autopilot (PX-Autopilot) offers Kubernetes-aware data management and capacity automation. Autopilot is a rule-based engine that responds to the changes in the Kubernetes environment and acts on them based on a set of rules. For mission-critical stateful applications like Kafka, you can set up Autopilot to automatically grow the underlying storage for brokers if ingest data exceeds the original provisioned space. Autopilot includes the ability to automatically resize individual container volumes or your entire storage pools. It also enables Kubernetes admins to rebalance or expand entire backend storage pools on certain platforms, allowing them to right-size capacity, consumption, and spend, leading to a better overall total cost of ownership (TCO).

PX-Migrate for Portability

Portworx creates a globally available virtual storage pool for Kubernetes applications. As a result, Portworx enables both applications like Kafka and their *data* to be portable across any infrastructure or cloud. This means that you can create an application-aware copy of your Kafka services in real-time from your production and move them to another environment either for test or development purposes. The PX-Migrate service enables these capabilities for stateful applications like Kafka on Kubernetes.

PX-Secure

PX-Secure, a container data security service, enables cluster-wide encryption or container-granular or storage-class based encryption. With PX-Secure, the volume that hosts the Kafka broker can be encrypted at the storage volume (PVC) level, providing protection for the data-at-rest as well as in transit. Portworx uses dm-crypt, a disk encryption subsystem of the Linux kernel that can create, access, and manage encrypted devices. Volumes provisioned with Portworx can be encrypted with cluster-wide secrets shared by other volumes or per-volume secrets unique to each volume.

Portworx storage authentication and authorization can extend the security layer of Kafka and Kubernetes that allow you to control how volumes are accessed and managed by Kafka users. For authentication, Portworx accepts OpenID Connect

(OIDC) tokens and self-generated JSON web tokens (JWT), which makes it easy to use Portworx for enterprise-grade authentication systems, such as LDAP, AD, or SAML 2.0. Portworx also supports role-based access control (RBAC) for authorization, and you have the flexibility to specify ownership rights to control types of access (read, write, administrator) for specific volumes.

Apache Kafka on Kubernetes with Portworx

Kubernetes has become the standard across organizations for running cloud-native applications. Complemented by Portworx, you can run stateful applications like Kafka without worrying about the operational overhead of Kafka broker failure or managing the storage space to accommodate massive data growth. The solution allows scalability of the Kafka applications without compromising performance or availability.

For this solution, we also included Pure Storage FlashArray to host the Portworx storage pools. We performed a couple of tests that showed the value of both Portworx and FlashArray, which are documented in the Solution Validation and Testing section.

Solution Architecture

The figure below shows the overall technology stack for running Kafka on Kubernetes with Portworx. You can use various backing stores, from directly attached NVMe or SSD storage, to on-prem SANs such as FlashArray, to cloud-based block storage such as Amazon Elastic Block Store (EBS) or Azure Disk. Portworx can provide the same abstraction and benefits for your Kafka on Kubernetes architecture irrespective of the cloud or the infrastructure. Once Portworx is deployed on the Kubernetes environment, it can provide specialized data management capabilities such as high availability, placement strategies, encryption, migrations, and fast-failover.

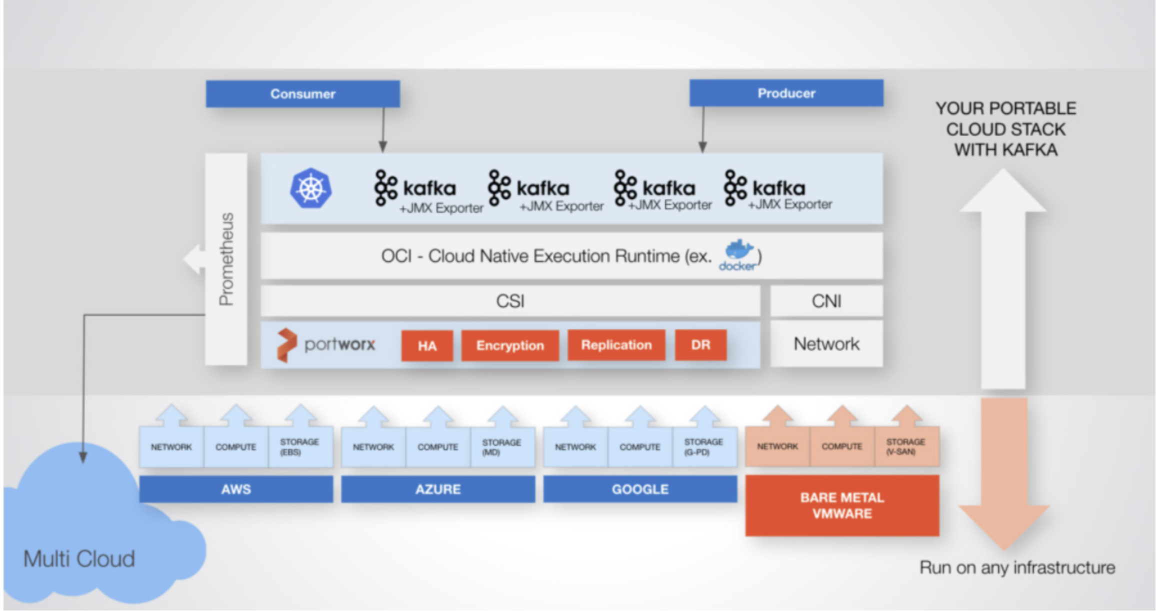


Figure 9. Kafka on Kubernetes with Portworx

The following diagram illustrates an example of the Kafka deployment on Kubernetes with Portworx. This is a four-broker Kafka cluster with Portworx that uses the local direct-attached NVMe drives to create the storage pool on every server.

NOTE: This is purely an illustration of a sample deployment and not a limit on any of the underlying product's or platform's capabilities. You can certainly scale well beyond this number of brokers.

For higher application and data availability, the Portworx replication factor should be configured at 2. With this, Portworx replicates the volume at the storage layer and, as a result, it can quickly recover from pod-, node-, network-, and disk-level failures. It can quickly bring up the failed broker pod on another Kubernetes worker node for maximum uptime.

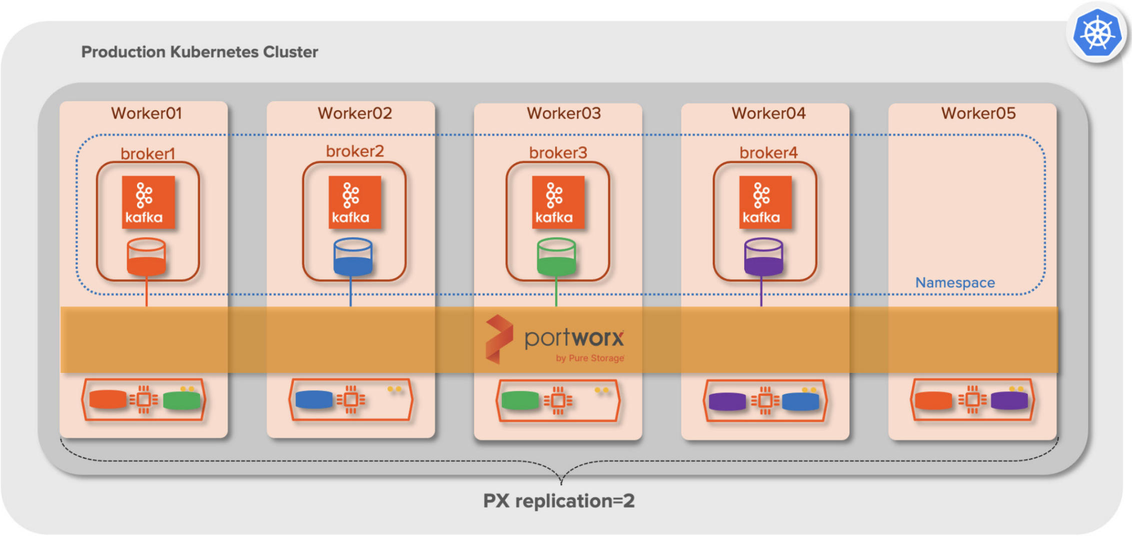


Figure 10. Kafka solution architecture with Portworx

The following diagram shows a similar logical architecture of a Kafka cluster on Kubernetes with Portworx by using FlashArray as the underlying storage. Portworx 2.8 automatically creates a separate volume out of FlashArray for each worker node and attaches to the node over iSCSI, where Portworx creates the storage pool on every node.

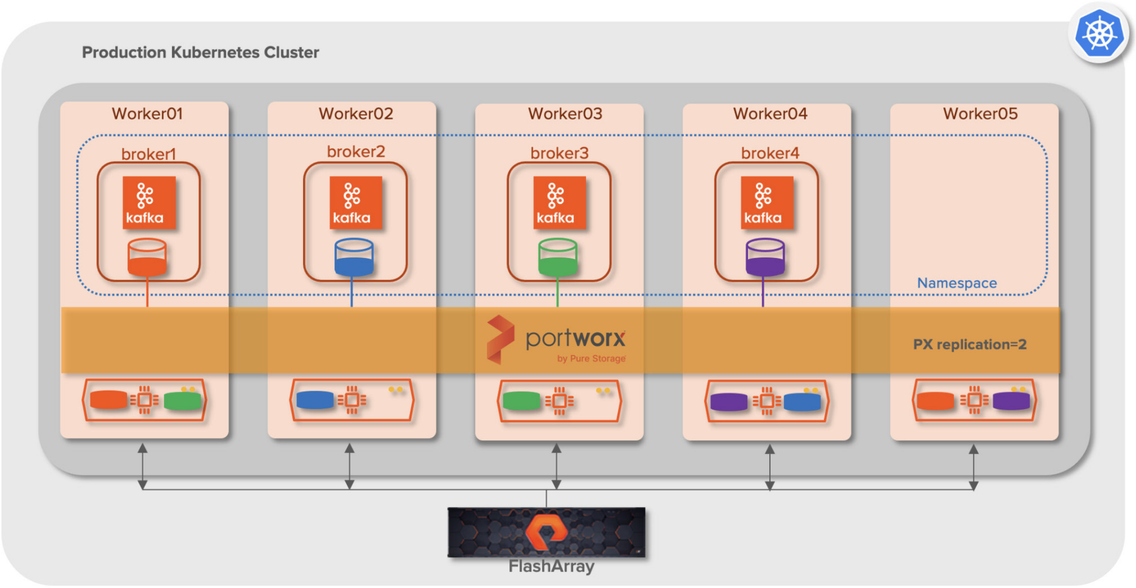


Figure 11. Kafka solution architecture with Portworx and FlashArray

Test Environment Overview

For this test environment, Portworx 2.8 was deployed on Kubernetes 1.20.6 in an environment that was made up of 10 physical and 10 virtual machines, all running Ubuntu 18.04.5. Each physical server included 8 x 1.5TB of NVMe drives and 8 x 2TB HDD drives. Based on the tests, the Portworx backend store was either created out of the NVMe drives or the Pure Storage FlashArray SAN volumes.

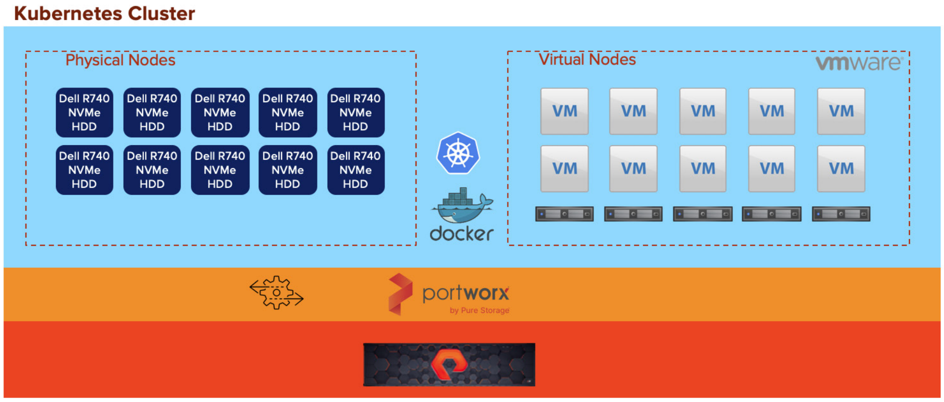


Figure 12. Test environment overview

The physical servers were equipped with high performant compute resources (2x Intel Xeon Platinum 8260 @2.4GHz with 24 cores, 48 cores total), and the Kafka broker pods were hosted on the physical servers throughout the tests.

The virtual servers were mostly used for hosting the test client pods that were either producing data or consuming data out of the Kafka cluster. It also hosted the internal key-value database (KVDB) for Portworx and the Zookeeper pods.

For the majority of the tests, a single Kubernetes cluster with all 20 servers was used, with three additional virtual machines that hosted the master nodes. For the portability/migration test, two Kubernetes clusters with 10 servers (five physical servers and five virtual machines) were used, with an additional virtual machine for the master node on each cluster.

Kubernetes can be deployed in multiple ways, such as bootstrapping clusters with kubeadm or through deployment tools like kubespray, kops, etc. We used kubespray to deploy the Kubernetes cluster, but feel free to follow your organization's preferred choice for deploying Kubernetes. Note that this document will not show how to deploy the Kubernetes cluster, but it does include relevant links in the References section.

System Configuration Details

Software Configuration

Component	Physical Servers
Operating System	Ubuntu 18.04.5
Kubernetes	1.20.6
Portworx	2.8
Kafka	2.8
Kubespray	2.16.0

Table 1. Software configuration

Server Configuration

Component	Physical Server	Virtual Server
CPU	48 Cores (Intel Xeon 8260)	24 vCPUs
Memory	256GB	128GB
Network	2 x 10GbE	2 x 10GbE
Portworx Backing Disk	Local NVMe FlashArray attached iSCSI	FlashArray attached iSCSI

Table 2. Physical and virtual server configuration

Kafka Metrics Monitoring

Kafka publishes various metrics in real-time, including bytes ingested, bytes read, count of produced/consumed messages, message size, fetch requests, and broker resource usage. It publishes these metrics through JMX (Java Management Extensions). JMX exporter is a collector that can run along with Kafka and expose JMX metrics over an HTTP endpoint. The metrics can be consumed by systems like Prometheus.

We configured Kafka to run with the Java Agent option pointing to the JMX to Prometheus exporter and exposed the metrics on port 7071 within each Kafka broker:

```
- name: KAFKA_OPTS
   value: "-javaagent:/opt/kafka/prometheus/jmx_prometheus_javaagent-
0.13.0.jar=7071:/opt/kafka/prometheus/kafka-2_0_0.yml"
```

Prometheus was deployed on the same Kubernetes cluster, configured to scrape the data from the brokers on port 7071. Grafana was deployed as the visualization and analytics tool connecting to Prometheus.

Apart from Kafka, Portworx storage and network metrics can also be monitored through Portworx's integration with Prometheus. Portworx has had the option to enable monitoring through Prometheus for quite some time, but as of Portworx Enterprise 2.8, the **Enable Monitoring** option under Advanced Settings is enabled by default. This option deploys Prometheus automatically through an Operator. For more details, check this Portworx documentation.

Test Data Set

Kafka offers its performance testing tools, named `kafka-producer-perf-test` and `kafka-consumer-perf-test`, to measure the bandwidth and latency. The synthetic data created by kafka-producer-perf-test is very similar across all records, so the data is highly compressible when using producer/broker-level or storage-level (FlashArray) compression. Unfortunately, this level of compression is not common in the real world, so it wouldn't help in measuring the actual throughput as well as the storage usage. Hence, we opted to use the following options to generate data as the source for data ingestion into topics in our solution testing:

- Flog data: A fake Apache log. An open-source utility that can generate synthetic Apache log data with the standard record size between 100 and 120 bytes.
- FIO data: We used a flexible I/O (FIO) utility to generate binary data files that can be deduplicated at a ratio of 2:1 (50% reduction) across various record sizes like 128 bytes, 256 bytes, 512k bytes, 1K, 4K, and 10K. We used those files as the payload file into the kafka-producer-perf-test utility.

Portworx Enterprise Configuration

Portworx Enterprise 2.8 was deployed on-prem. The Portworx cluster used the spec generator tool located at https://central.portworx.com to generate the installation resources.

Portworx with FlashArray

Pure Storage FlashArray is the world's first 100% all-flash end-to-end NVMe and NVMe-oF array, ideal for the most demanding enterprise performance requirements.

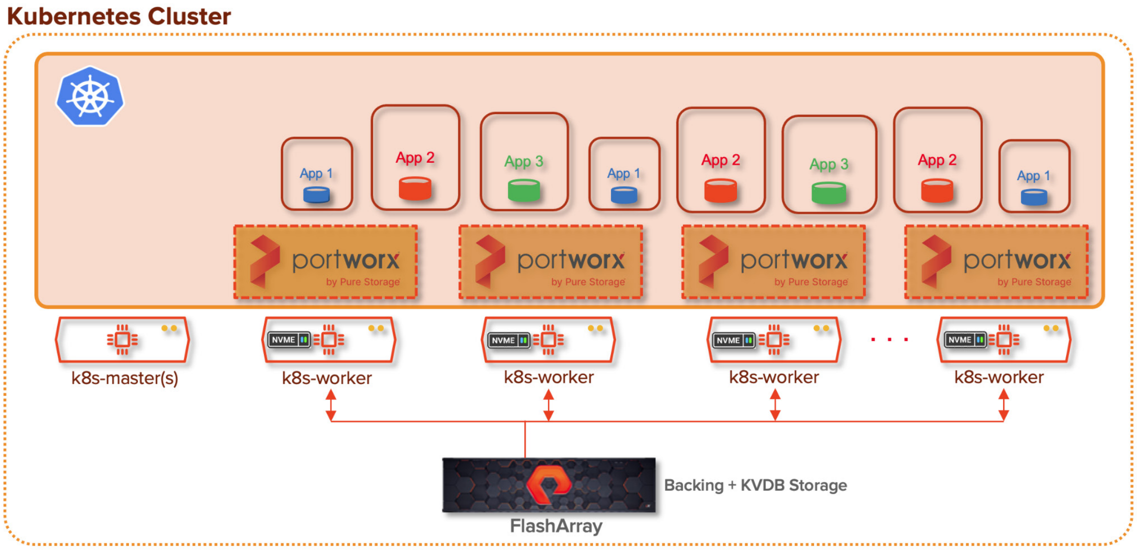
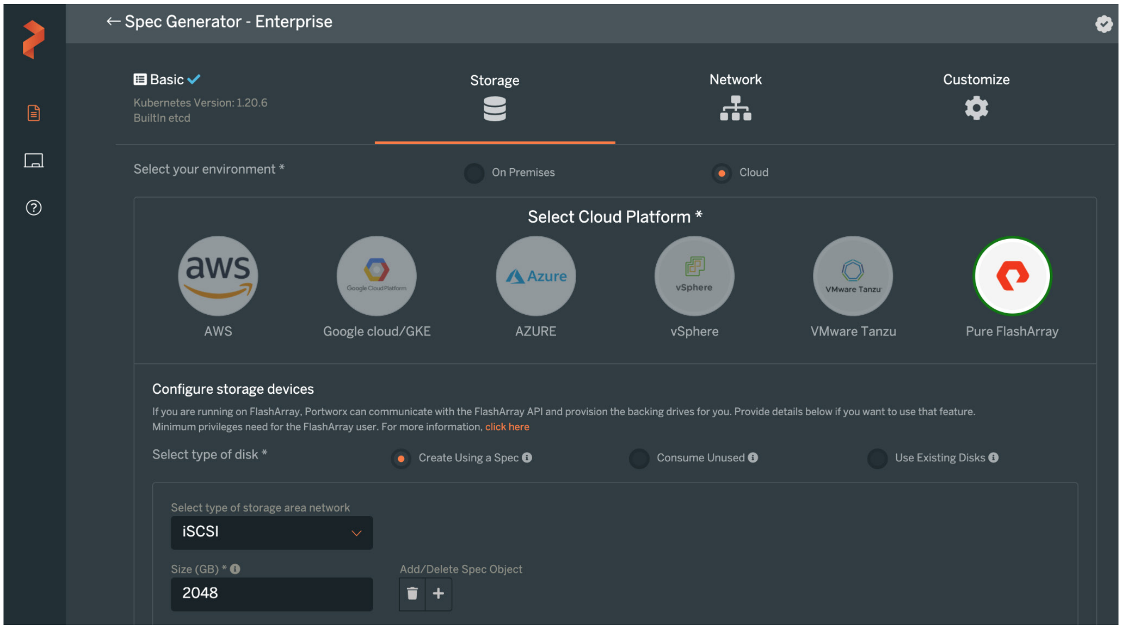


Figure 13. Portworx configuration with FlashArray

Portworx Enterprise makes it even easier to take advantage of the enterprise capabilities of FlashArray, including all-flash performance, non-disruptive upgrades, and data reduction, alongside the container-granular capabilities of Portworx.

Portworx 2.8 automatically creates the FlashArray volume for every worker node based on the size provided (we opted for 2TB per node). It also creates the volumes required for the KVDB metadata (32GB each) and attaches them over iSCSI to the worker nodes. You need at least three worker nodes in your Portworx cluster so that Portworx can replicate data across nodes. By creating the volumes out of FlashArray, Portworx can now support all cloud-native features like replication, snapshots, migration, and security in a container-granular and application-consistent manner.

Within the Portworx Spec Generator, we selected the Portworx Operator option with built-in etcd data stores. We chose Pure FlashArray under the Cloud environment option with a size of 2TB per node. All other options were left at default, so Portworx was set up with Monitoring, Telemetry, CSI, and STORK features.



Select your environment * ○ On Premises ● Cloud

Select Cloud Platform *

Configure storage devices

If you are running on FlashArray, Portworx can communicate with the FlashArray API and provision the backing drives for you. Provide details below if you want to use that feature. Minimum privileges need for the FlashArray user. For more information, click here

Select type of disk * ● Create Using a Spec ○ Consume Unused ○ Use Existing Disks

Select type of storage area network: iSCSI

Size (GB) *: 2048

Add/Delete Spec Object

Figure 14. Portworx Enterprise Spec Generator

Once the details were entered on the Portworx Spec Generator UI, the spec file was downloaded. Before applying the specs on the Kubernetes environment, a pure.json file and the secret named px-pure-secret are required so Portworx can provision the backend storage on FlashArray.

```
root@kpx-master01:~# more pure.json
{
  "FlashArrays": [
    {
      "MgmtEndPoint": "10.21.214.13",
      "APIToken": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx"
    }
  ],
  "FlashBlades": [
    {
      "MgmtEndPoint": "10.21.213.200",
      "APIToken": "xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx",
      "NFSEndPoint": "10.21.214.205"
    }
  ]
}
```

We ran the following command to create the px-pure-secret from the JSON file that was created in the previous step, which includes the FlashArray details:

```
kubectl create secret generic px-pure-secret --namespace kube-system --from-file=pure.json
```

In addition to the above, to identify the servers that should host the internal key-value data store, we ran the following command:

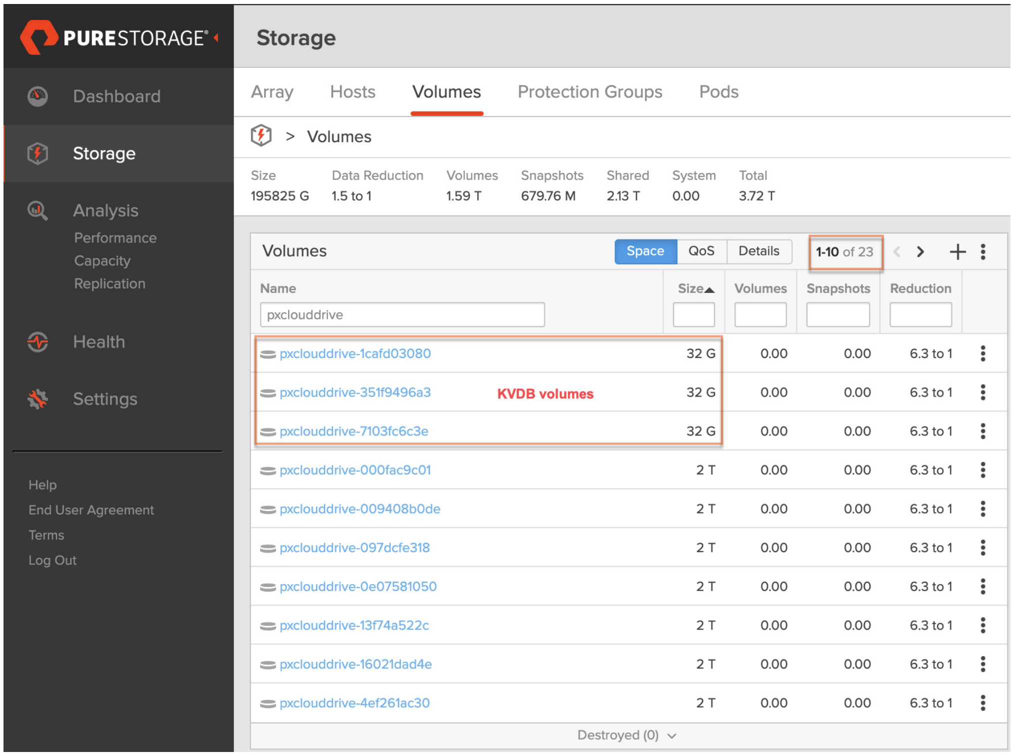
```
kubectl label nodes k8s-worker16 k8s-worker18 k8s-worker20 px/metadata-node=true
```

Only the nodes with the label **px/metadata-node=true** will participate in the KVDB cluster.

The downloaded spec file was applied to the Kubernetes cluster. Portworx created a resource type **storagecluster** under the kube-system namespace, which was in the "Initializing" state until Portworx was installed and configured on all the worker nodes. Once it completed, the status showed as "Online."

```
root@kpx-master01:~# kubectl get storagecluster -n kube-system
NAME                                             CLUSTER UUID                           STATUS  VERSION AGE
px-cluster-72343af1-38eb-4b97-b1d6-48fa3c680b74  1ebfbfce-72e9-4806-a473-c03da4864612 Online  2.8.0   5d2h
```

The volumes created on FlashArray will have **pxclouddrive** as the prefix. You can see from the below screenshot that 23 volumes start with "pxclouddrive", of which 20 are the backing storage volume of 2TB each and three of them are 32GB to host the KVDB metadata.



Size	Data Reduction	Volumes	Snapshots	Shared	System	Total
195825 G	1.5 to 1	1.59 T	679.76 M	2.13 T	0.00	3.72 T

Volumes

Name	Size	Volumes	Snapshots	Reduction
pxclouddrive				
pxclouddrive-1cafd03080	32 G	0.00	0.00	6.3 to 1
pxclouddrive-351f9496a3	32 G	0.00	0.00	6.3 to 1
pxclouddrive-7103fc6c3e	32 G	0.00	0.00	6.3 to 1
pxclouddrive-000fac9c01	2 T	0.00	0.00	6.3 to 1
pxclouddrive-009408b0de	2 T	0.00	0.00	6.3 to 1
pxclouddrive-097dcfe318	2 T	0.00	0.00	6.3 to 1
pxclouddrive-0e07581050	2 T	0.00	0.00	6.3 to 1
pxclouddrive-13f74a522c	2 T	0.00	0.00	6.3 to 1
pxclouddrive-16021dad4e	2 T	0.00	0.00	6.3 to 1
pxclouddrive-4ef261ac30	2 T	0.00	0.00	6.3 to 1

Figure 15. Portworx cloud drive volumes on FlashArray

You can check the status of Portworx at the worker node level by running the `pxctl status` command. It will show the operational state of Portworx on the local worker node, the local storage pool details, the cluster summary, including the details of all the storage nodes and the space used, and the total space.

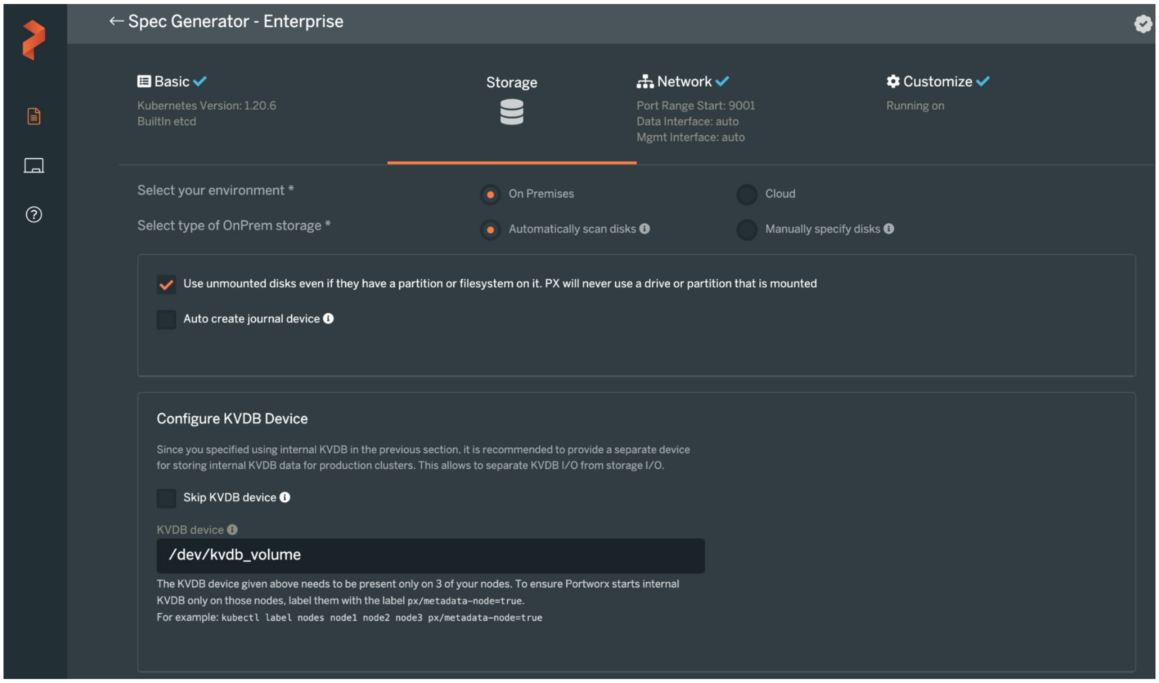
Portworx with Local Storage

The process to set up Portworx Enterprise 2.8 is similar when using locally attached storage like NVMe drives or HDD drives. In the Portworx Spec Generator, select **Portworx Operator** along with **Built-in ETCD**. Under the environment section, select **On-Premises** and choose **Automatically scan disks**. If you have already created a software RAID volume out of these drives, you can instead choose **Manually specify disks** and provide the path. Note that this means all the nodes should have the same path for Portworx to configure the backend store. Also, you have to specify the volume for the internal key-value data store on three of the nodes that are labeled with `px/metadata-node=true`.

> NOTE: Make sure to have the volume attached to the nodes that will host the KVDB store but not mounted. All other options can be left at default, so Portworx is set up with Monitoring, Telemetry, CSI, and STORK features.

Make sure that the three nodes that are identified to host the internal key value store are labeled as below:

```
kubectl label nodes node1 node2 node3 px/metadata-node=true.
```



← Spec Generator - Enterprise

Basic ✓
Kubernetes Version: 1.20.6
BuiltIn etcd

Storage

Network ✓
Port Range Start: 9001
Data Interface: auto
Mgmt Interface: auto

Customize ✓
Running on

Select your environment * ● On Premises ○ Cloud

Select type of OnPrem storage * ● Automatically scan disks ○ Manually specify disks

☑ Use unmounted disks even if they have a partition or filesystem on it. PX will never use a drive or partition that is mounted

□ Auto create journal device

Configure KVDB Device

Since you specified using internal KVDB in the previous section, it is recommended to provide a separate device for storing internal KVDB data for production clusters. This allows to separate KVDB I/O from storage I/O.

□ Skip KVDB device

KVDB device: /dev/kvdb_volume

The KVDB device given above needs to be present only on 3 of your nodes. To ensure Portworx starts internal KVDB only on those nodes, label them with the label `px/metadata-node=true`.
For example: `kubectl label nodes node1 node2 node3 px/metadata-node=true`

Figure 16. Portworx Enterprise Spec Generator for local storage

Now apply the spec file that will set up and configure Portworx across all worker nodes. You can check the status of the installation by checking the status of the new resource type **storagecluster** from the kube-system namespace, which should show **Online** once Portworx is set up and configured.

```
root@kpx-master01:~/px# kubectl get storagecluster -n kube-system
NAME                                              CLUSTER UUID                                  STATUS   VERSION
px-cluster-61cdb582-0225-4fbc-b6b2-c6c7586abc8f   4674252d-d23d-43c2-b735-110752e772e3   Online   2.8.0
AGE
3h7m
```

Apache Kafka Configuration

There are various ways to deploy Kafka on Kubernetes. Some of the most common deployment approaches are:

- Helm chart (Bitnami, Confluent)
- Kafka Operators like Strimzi and Confluent
- Manual deployment

We opted for manual deployment in our testing because it gives the most control over how Kafka runs on Kubernetes. Kafka is a stateful application and it requires stable network identity (Ips) and persistent storage. On Kubernetes, Kafka requires the two stateful components: StatefulSet resources and headless services.

StatefulSet resources provide stable network identity (Ips) for each pod in a stateful set. The pod gets the same IP when it is rescheduled, and this is important because the address on which the Kafka clients and consumers connect to the brokers should not change. An additional benefit of using StatefulSet resources is that they guarantee ordered, graceful deployment, scaling, and ordered rolling updates.

Headless services provide a single service IP and allow interfacing with Kafka service discovery without being tied to the Kubernetes implementation. Headless services can be used in-the case where you don't need load-balancing but a single service IP.

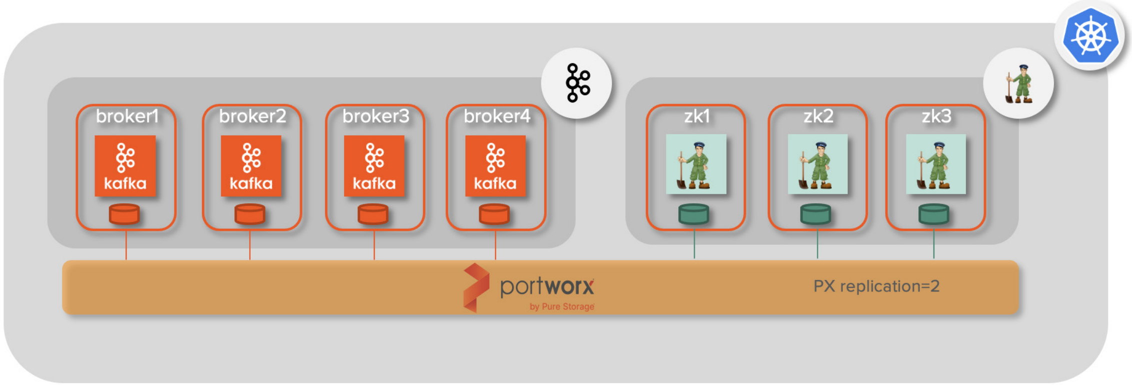


Figure 17. Apache Kafka configuration with Portworx

Kafka was configured with a 3-node Zookeeper and a set of Kafka brokers all using persistent storage from Portworx. Kafka can consume Portworx volumes through the use of a StorageClass object. A StorageClass object allows Kubernetes operators to define the type of storage offerings available within a Kubernetes cluster. For Portworx and Kafka, the StorageClass object would look like the following:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: px-storageclass-kafka
provisioner: kubernetes.io/portworx-volume
allowVolumeExpansion: true
parameters:
  repl: “2”
  priority_io: “high”
  io_profile: “db_remote”
```

Portworx offers a variety of parameters, including labels, journals, encryption, snapshot scheduling, snapshot types, and replications. We will focus on the principal parameters to use for Kafka deployments:

- **repl:** This setting provides the number of full replicas of broker data that Portworx will distribute across the cluster. It allows values between 1 and 3. A replica setting of 3 for a Portworx broker PVC means that there will be three copies of the broker data across failure domains in the Portworx cluster. The Portworx cluster can sustain up to two storage node failures that hold the replica. With a replica setting of 1, node failure of the replica-owning node can cause application downtime.
- **priority_io:** This setting instructs Portworx to place the replicas for your broker on the fastest backend disks it has available within its storage pools. Ultimately, when designing Kafka on Portworx, you should provide the Kafka-recommended type of fast backing storage per node to Portworx. Portworx will then create a virtual pool and provide Kafka brokers with a virtual block device from this high priority storage pool.
- **io_profile:** Portworx offers various I/O profiles to allow operators to tailor the I/O interaction on the Portworx storage pools. I/O profiles help with the overall performance of the volumes used by the applications. For Kafka, *db_remote* should be used to implement write-back flush coalescing for optimal performance of Kafka brokers.

The StorageClass resource is referred to in the volumeClaimTemplates section of the ZooKeeper and Kafka application configuration files, which provide stable storage using PersistentVolumes provisioned by the PersistentVolumes provisioner.

The application configuration files, ConfigMaps, StorageClass, and the scripts to start and shut down the Kafka cluster can be found in this GitHub repository.

Solution Validation and Testing

The following tests were performed as part of the solution validation to ensure that it addresses the challenges that were discussed in the earlier section:

- Operational efficiency through broker failover improvements with Portworx
- Kubernetes-aware Data Management for Kafka through PX-Autopilot
- Demonstrate the portability of Kafka application along with the data to another Kubernetes cluster using PX-Migrate
- Secure Kafka brokers by encrypting the data at the PVC level using PX-Secure
- Scalability tests to ensure Kafka on Kubernetes with Portworx scales linearly

Kafka Broker Failover Improvements with Portworx

When a Kafka broker with direct-attached storage hosting its data fails, all partition replicas on that broker become unavailable, making the topic-partition under replicated. (This assumes that it has more than one replica, which you should in a production setup.) The Kafka cluster will be operating with a lowered availability. If the broker fails permanently, Kafka doesn’t offer any automatic recovery and the Kafka administrators have to manually perform the partition rebalancing activity among the brokers using the kafka-reassign-partitions utility. This is a manual effort that is proportional to the size of the data. If it takes two hours to rebalance a 4TB cluster, then it will take four hours for an 8TB cluster and the cluster will remain under-replicated until this completes.

Portworx can significantly improve the availability of Kafka data by providing storage-level replication, where a broker pod failure results in the broker being scheduled onto a healthy node and reattached to the original Portworx virtual volume it was using. As you can see from Figure 16, broker2 was hosted on the node worker02 with the storage hosted on the Portworx virtual volume that is serving out of the Portworx replica on worker02. When the worker02 node failed, the corresponding broker2 pod was impacted as well. Portworx helps in rescheduling the pod to another worker node (worker03) and attaches the original virtual volume, which is now served out of the Portworx replica in worker03.

This allows the broker to come online with the original data and participate in the Kafka cluster right away. The broker then synchronizes any new messages after the failure has occurred instead of rebuilding all data from the other in-sync replicas that weren't affected by the failure. There is no data movement required between the brokers, which is the biggest advantage of Portworx. Compared to the traditional setup, the Portworx approach doesn't take longer as the Kafka cluster grows, but rather it stays constant at the time to respawn the pod.

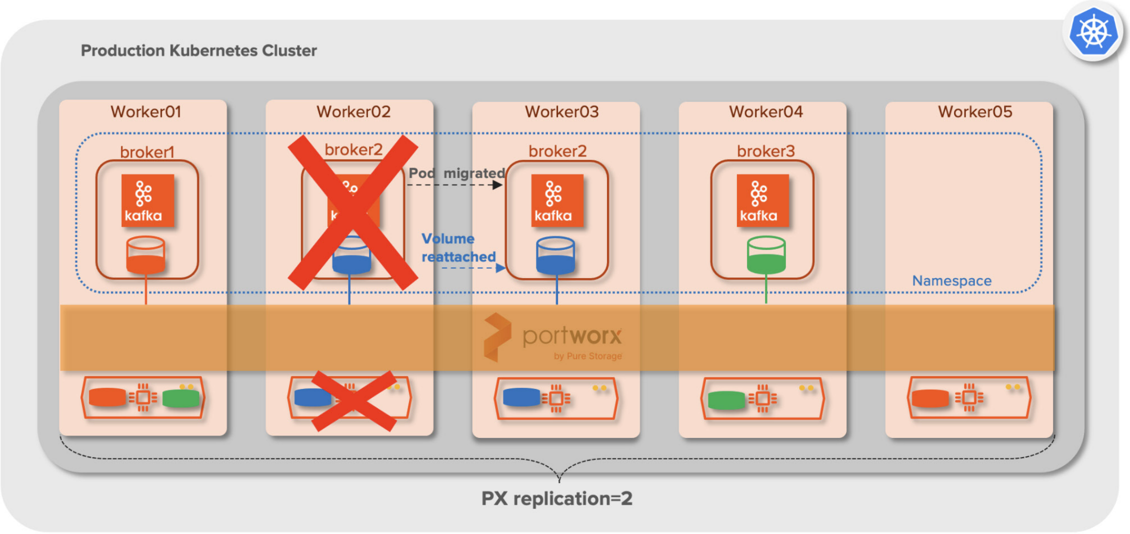


Figure 18. Kafka broker failover with Portworx

Test Setup

Number of Kafka Brokers	4	4
Storage	Local NVMe drives	Local NVMe drives
Topic replication factor	2	2
Ingested data	1TB	2TB
Kafka cluster storage usage	2TB	4TB
Per broker data usage	511GB	1021GB

Table 3. Test setup.

To measure the baseline of partition reassignment timing on a Kafka cluster that was not using Kubernetes, we:

1. Failed a broker (1004).
2. Added a new broker (1005).
3. Performed a partition reassignment with the available four brokers (1001,1002,1003,1005).
4. Measured the time taken to rearrange 511GB of data.

To measure the broker failover time with Kafka on Kubernetes with Portworx, we:

1. Failed a broker pod.
2. Measured the time that the pod takes to be rescheduled on a new worker node with the original data.

Based on our testing, rebalancing 511GB of data took over 18 minutes on the traditional Kafka cluster without Kubernetes, whereas Portworx reduced the respawning time to around three minutes. Similarly, rebalancing 1021GB took over 34 minutes on the traditional Kafka cluster while Kafka pod rescheduling took under two and half minutes.

The interesting point here is that the Portworx failover time includes various configurable failover thresholds within Portworx to find any nodes that are down and see if they will come back online. The recovery or failback of the broker with Portworx doesn’t involve any data movement, and hence would be the same irrespective of the size of the Kafka data on the broker. This time can be further reduced. See Broker Pod Availability within the Best Practices section for more details.

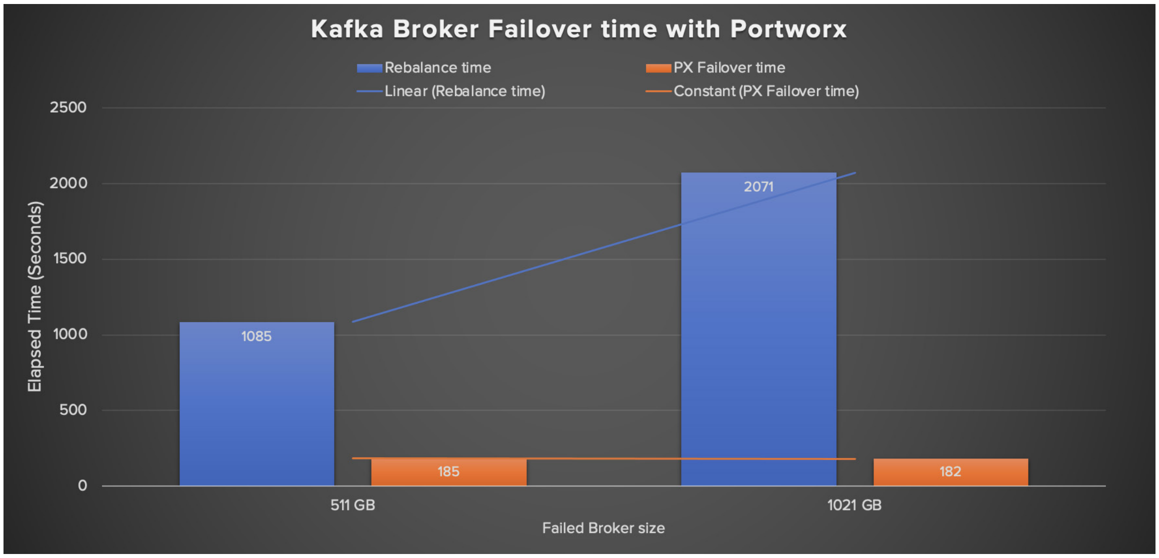


Figure 19. Kafka Broker failover improvement with Portworx

As you can see, even in the Kubernetes environment with StatefulSet controllers, a hard failure of a worker node that has the Kafka broker running doesn’t reschedule the broker pod to another healthy node. The only option is to create the broker on the surviving nodes and rebalance the data between the broker pods, but it can be very time consuming based on the size of the data hosted by the failed broker pod.

With Kafka on Kubernetes with Portworx, any worker node failure causes the broker pod to be respawned to another healthy worker node with access to the original Portworx volume automatically, without any manual intervention. The respawning time will be similar every time irrespective of the size of the Kafka data on the pod, thus reducing the failover time significantly.

Kubernetes-aware Data Management for Kafka Through PX-Autopilot

To test capacity automation, we performed the following sequence of activities:

1. Set up Kafka cluster with four broker nodes each with PVC of 400GB
2. Enabled Autopilot rule on the Portworx cluster with a rule to increase the Portworx volume by 100% when the space usage reaches 80%, with a maxsize value of 1TB.
3. Created a topic and ingested data.
4. As the individual Kafka broker usage reached 80% of 400GB, or 320GB, we expected Autopilot to resize the PVC by 100%, to 800GB.
5. Continued ingesting data into the topic.
6. As the broker usage reached 80% of 800GB (the new PVC size) or 640GB, we expected Autopilot to resize the PVC to 1TB and not to 1200GB (100% of the original 400GB) due to the maxsize settings.
7. Continued ingesting data into the topic.
8. As the broker usage reaches 80% of 1TB or 800GB, we expected Autopilot not to resize the PVC due to the `maxsize` settings.

Here is the Autopilot rule that we applied to the cluster to perform the test.

```
apiVersion: autopilot.libopenstorage.org/v1alpha1
kind: AutopilotRule
metadata:
 name: kafka-resize
spec:
  ##### selector filters the objects affected by this rule given labels
  selector:
    matchLabels:
      app: kafka
  pollInterval: 2
  ##### conditions are the symptoms to evaluate. All conditions are AND’ed
  conditions:
    # volume usage should be greater than 80%
    expressions:
    - key: “100 * (px_volume_usage_bytes / px_volume_capacity_bytes)”
      operator: Gt
      values:
        - “80”
  ##### action to perform when condition is true
  actions:
  - name: openstorage.io.action.volume/resize
    params:
      # resize volume by scalepercentage of current size
      scalepercentage: “100”
      maxsize: “1Ti”
```

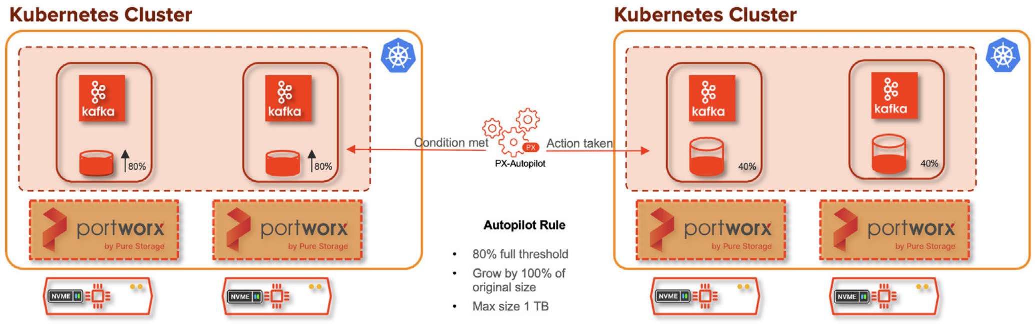


Figure 20. Capacity management through Autopilot

The following table shows the outcome of the test based on the Autopilot rule listed above.

Number	PVC Size	Scale %	Max Size	Threshold %	Threshold Value	New PVC Size	Remarks
1	400GB	100%	1TB	80%	320GB	800GB	Resized 100%
2	400GB	100%	1TB	80%	640GB	1TB	Resized but to maxsize
3	400GB	100%	1TB	80%	800GB	1TB	Resize rejected

Table 4. Autopilot test results

As you can see, Autopilot detects when the Kafka broker PVCs meet the threshold and automatically performs a resize action based on the usage metrics from the PVCs. In the traditional setup that does not use Portworx, when the Kafka broker runs out of space, the following alternative options might be used:

- Limit or reduce the retention of topics.
- Extend the space on the existing filesystem if the underlying storage has additional space. Restart the Kafka broker for the new space to take effect.
- Add new disks to the server, mount them, and append the `log.dirs` parameter with the new location. Restart Kafka brokers for it to take effect.
- Add a new broker with space to the Kafka cluster and rebalance the data across all brokers to make room in the existing brokers.

All the above scenarios don't allow dynamic changes and generally involve downtime. In contrast, the Autopilot rule is very dynamic and allows you to specify various monitoring conditions, along with the actions it should take when those conditions are met. Autopilot also enables Kubernetes admins to rebalance or expand entire backend storage pools on certain platforms.

Kafka Application Portability with PX-Migrate

In this test, we attempted to migrate the Kafka application with its data from one Kubernetes cluster to another using PX-Migrate. PX-Migrate is the essential technology behind PX-DR. For this test, we focused on migrating Kafka and its data from the production environment to a test environment.

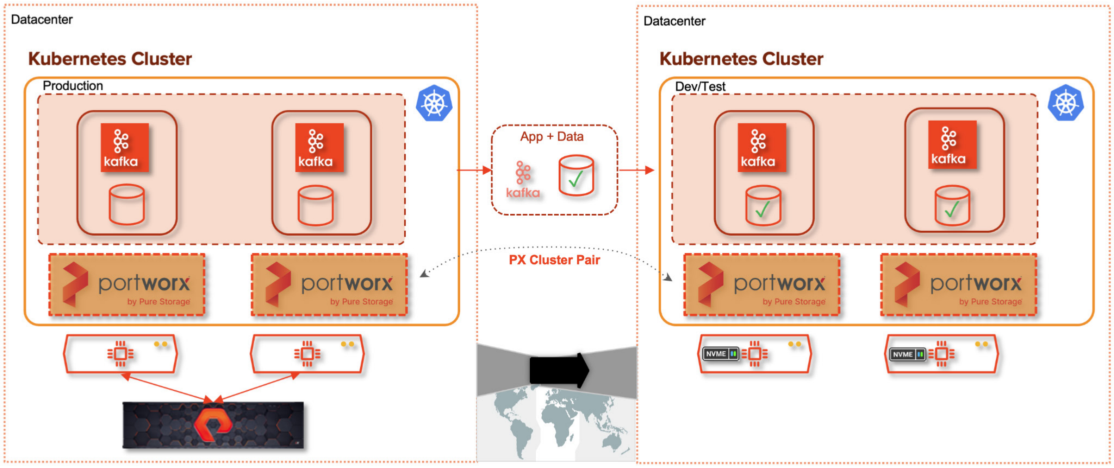


Figure 21. Migration of Kafka app and data through PX-Migrate

The above illustration shows how PX-Migrate can pair two independent clusters and allow Kafka metadata and the volume data to be moved as a single unit.

The migration process is documented here: https://docs.portworx.com/portworx-install-with-kubernetes/migration/.

PX-Migrate uses an interim S3 object store as a placeholder to back up the application, data, and the metadata from the source cluster and restores them from that object store onto the target cluster. For this test, we used Pure Storage FlashBlade®, all-flash scale-out file, and object storage, as the interim placeholder because FlashBlade supports S3. You can use any S3-compatible object store that you deem appropriate.

In our test, PX-Migrate successfully migrated the Kafka application, along with the data, from the source cluster to the target cluster rapidly through FlashBlade. On the target cluster, the PX volumes were created in the backend with the same properties as the source and the similar Kafka broker pods were brought up.

```
root@kpx-master01:~# storkctl get migration
NAME               CLUSTERPAIR     STAGE   STATUS       VOLUMES   RESOURCES   CREATED              ELAPSED
kafka-migration    remotecluster   Final   Successful   7/7       22/22       15 Jul 21 22:07 PDT   6m39s
```

PX-Migrate enables migrations that can provide application-aware rules for consistency. It allows you to move the Kafka-application metadata (deployment and StatefulSet resources) along with its data to another Kubernetes environment.

Kafka Data Encryption at PVC Level with PX-Security

PX-Secure allows the volume that hosts the Kafka broker to be encrypted at the PVC level, fulfilling the encryption-at-rest requirement. Apart from validating the functionality of PX-Secure, the test measured the overhead of enabling encryption at the Portworx volume level.

To compare the performance overhead of PX-Secure, we performed the following steps:

1. Ingested flog (fake Apache log) data for 10 minutes
2. Measured the records per second generated by the ingest at Kafka level
3. Repeated the above test for multiple producer clients like 32, 64, 96, and 128

4. Performed a query to read 75 million messages by varying the clients and measured the response time

The above tests were performed on a four-broker Kafka cluster without encryption and again with encryption through PX-Security, and then we compared the difference between the two. The backend store used the locally attached NVMe drives.

To enable encryption at the PVC level, the following steps should be followed.

1. Check if Portworx is configured to use K8s secrets by inspecting the /etc/pwx/config.json file. The secret_type value should show “k8s.”

```
more /etc/pwx/config.json
{
  "alertingurl": "",
  "clusterid": "kpx-cluster-f1f7e504-9b23-4344-9542-4adfd7798765",
  "dataiface": "",
  "bootstrap": true,
  "mgtiface": "",
  "scheduler": "kubernetes",
  "secret": {
    "secret_type": "k8s",
    "cluster_secret_key": ""
  },
  "storage": {
    "devices": [
      "/dev/md0"
    ],
    "cache": [],
    "rt_opts": {},
    "max_storage_nodes_per_zone": 0,
    "journal_dev": "",
    "system_metadata_dev": "",
    "kvdb_dev": ""
  },
  "version": "1.0"
}
```

2. Create a cluster-wide key, secret key pointing to a secret value/passphrase that can be used to encrypt all volumes.

```
Kubectl -n kube-system create secret generic px-vol-encryption --from-literal=cluster-wide-secret-key=K@fkA
secret/px-vol-encryption created
```

3. Set the cluster-wide secret key in Portworx from a worker node.

```
[root@k8s-worker06 /]# /opt/pwx/bin/pxctl secrets set-cluster-key --secret cluster-wide-secret-key
Successfully set cluster secret key
```

4. Create a new StorageClass object, including the key secure parameter set to the value of “true”.

```
Kind: StorageClass
apiVersion: storage.k8s.io/v1beta1
metadata:
  name: px-storageclass-kafka
provisioner: kubernetes.io/portworx-volume
allowVolumeExpansion: true
parameters:
  repl: "2"
  secure: “true”
```

```
priority_io: "high"
io_profile: "db_remote"
```

Test Setup

Component	Value
Number of Kafka Brokers	4
Storage	Local NVMe drives
Topic Replication Factor	2
Portworx Replication	2
Producer Batch settings	linger.ms=20, batch.size=64K, compression.type=lz4, acks=all
Producers/Consumers	32, 64, 96, 128
Data ingested	Flog data for 10 minutes
Data consumed	75M messages per consumer

Table 5. Test setup.

Producer Performance

During the ingestion of data through producers, we observed that encryption added overhead in the records or messages generated per second, especially when there was more load on the system. As you can see from the chart below, the impact of encryption on ingest performance was relatively small for 32 and 64 producer clients, but encryption had much more impact when more users were added to the four-broker system.

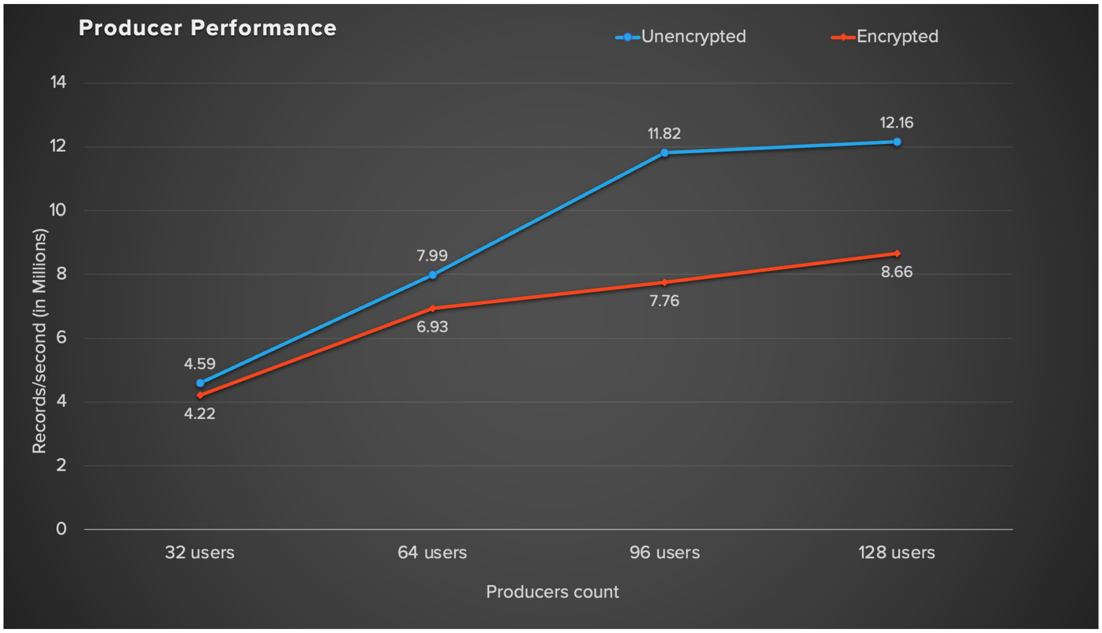


Figure 22. Producer performance with and without encryption

Consumer Performance

Like the producer performance, we noticed a similar overhead with consumer performance. The graph below shows the metric records or messages consumed per second with and without encryption.

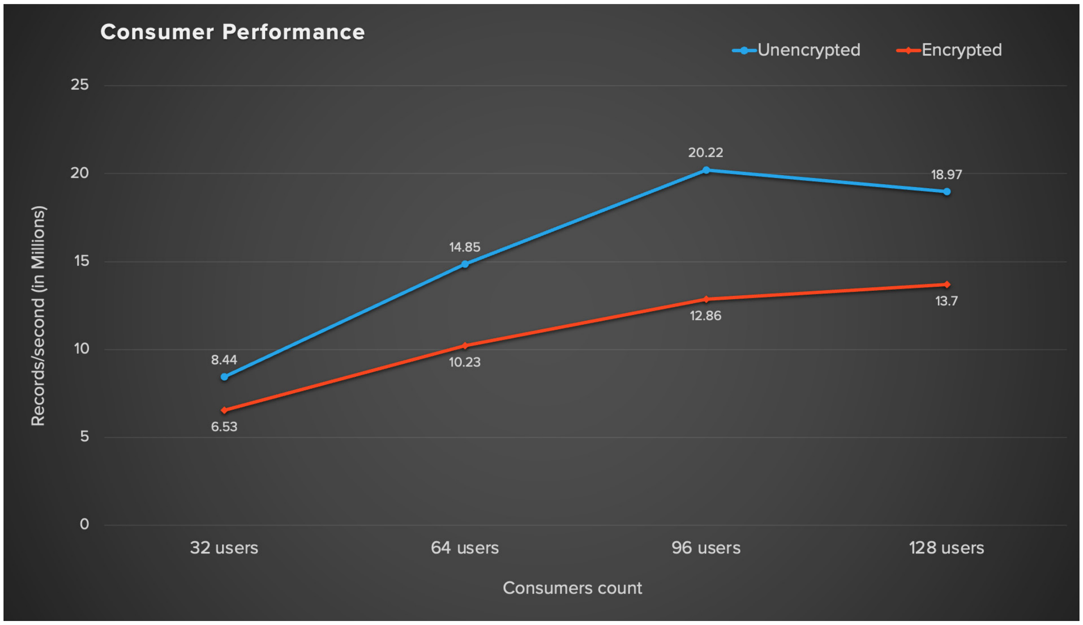


Figure 23. Consumer performance with and without encryption

Like the producer performance graph, we noticed that the increase in the load on the brokers can have an impact on the consumer performance because the data must be decrypted before sending it to the consumer clients. We noticed that the performance was clearly impacted by CPU usage and not due to any I/O bottlenecks.

In general, any software-level encryption will incur additional CPU cycles to encrypt/decrypt the data, but additional overhead depends on factors like load on the system, the type of data, and CPU type. As we noticed in the producer and consumer performance graphs, adding more load on the system impacts the performance when the volume is encrypted at the Portworx volume level because PX-Secure uses the Linux kernel-level dm-crypt module to encrypt/decrypt the data. This requires additional CPU cycles, sowhen the system is loaded, there is a heavy demand for the processors, which increases the overhead. The same is true when consumers are consuming data out of the brokers because that data has to be decrypted.

PX-Secure offers encryption-at-rest functionality through encryption at volume level that is not available out of the box for Kafka applications running on direct-attached storage.

If encryption-at-rest functionality is required, but you are not willing to incur the overhead of the encryption, an alternate option is to use FlashArray, which secures the data at rest with AES-256-bit encryption. The added advantage of FlashArray is that the data encryption occurs without impacting performance and while maintaining full data reduction capabilities.

Scalability Tests

We performed the user scalability, broker scalability, and replication performance tests to ensure Kafka on Kubernetes with Portworx scales linearly.

Users and Brokers Scalability

Component	Value
Number of Kafka Brokers	2, 4, and 8
Storage	Local NVMe drives
Topic Replication Factor	2
Portworx Replication	2
Producer Batch Settings	linger.ms=30, batch.size=64K, compression.type=lz4, acks=all
Users (Producers)	2x, 4x, 8x, 16x of brokers
Ingested Data	Flog data for 10 minutes per test

Table 6. Users and brokers scalability

Users and brokers scalability was validated through a series of tests where the users were either configured with two, four, or eight brokers, and four different tests were performed at 2x, 4x, 8x, and 16x users per broker. So, in the case of two brokers, the user scalability tests were performed with four, eight, 16, and 32 users. Similarly, with four brokers, the user scalability tests were performed with eight, 16, 32, and 64 users. Across all the tests, the records ingested per second were graphed for 2x, 4x, 8x, and 16x users against two, four, and eight brokers.

The following graph shows the users' scalability across two, four, and eight brokers. The tests with two brokers were included to show the results when it was scaled to four brokers, but this is not a common configuration.

As you can see, the performance scales almost linearly across the three different broker settings. The producer was configured with acks=all for higher durability of data.

Remember, the results shown here are not the maximum possible performance numbers based on the configuration. Instead, the results simply reflect the producer. broker, and Portworx settings. For example, the ingest performance with encryption for 128 users with eight brokers as illustrated in the producer performance graph in the previous section (Figure 22) is 12.16 million/sec with the producer settings of batch.size=64k and linger.ms=20, while we observed 13.8 million/sec with this test, where linger.ms was at 30. You can still fine-tune various parameters at the application, Portworx, and storage level to get higher numbers.

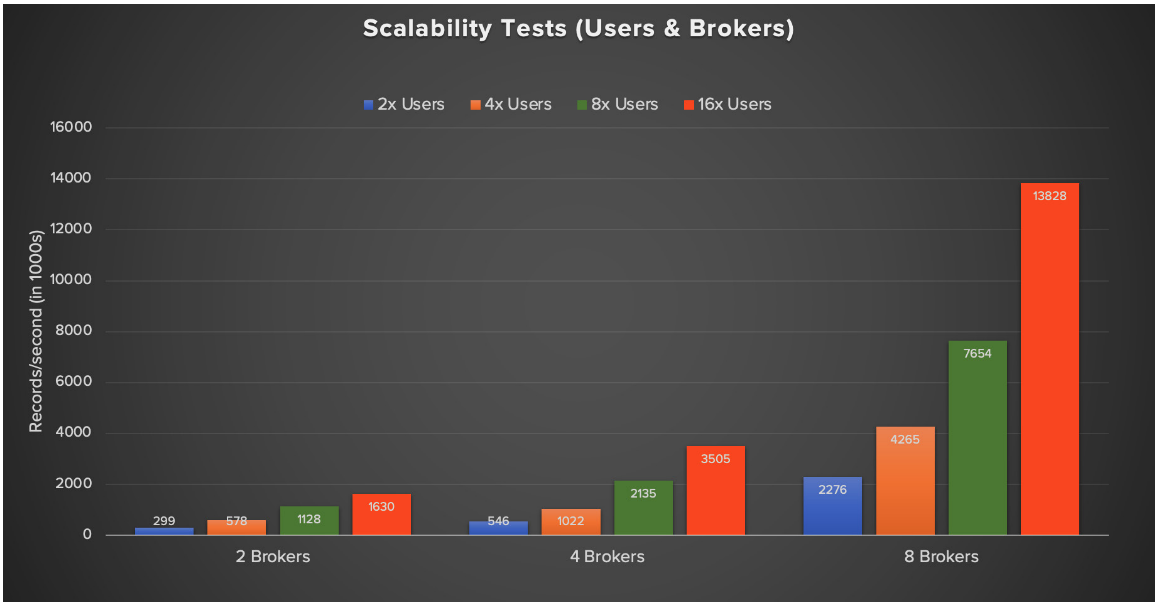


Figure 24. Users and brokers scalability tests

Replication Performance

Replication is the key requirement at both the application and storage level to achieve higher availability of data. To test the performance overhead of replication, we performed the following two tests:

- Comparing Kafka replication factors of 1, 2, and 3, combined with Portworx replication settings of 1 and 2.
- Comparing and contrasting the replication performance for the configurations 3x1 and 2x2.

Replication Performance Test #1

We tested ingest performance for all the six different scenarios with the Kafka replication factor set between 1 and 3, as well as Portworx replication between 1 and 2.

Component	Value
Number of Kafka Brokers	8
Storage	Local NVMe drives
Kafka Replication Factor (RF)	1, 2, 3
Portworx Replication (PX)	1, 2
Producer Batch Settings	linger.ms=30, batch.size=64K, compression.type=lz4, acks=all
Users (Producers)	64, 128, 192
Ingested Data	Flog data for 10 minutes per test

Table 7. Replication performance test #1.

The following graph illustrates the test results. The ingest performance was at the highest when both Kafka and Portworx replication factor was set at 1 (RF=1/PX=1). This test was included to show the range of throughput, but as suggested above, we do not recommend that users run Kafka with the topic replication factor of 1.

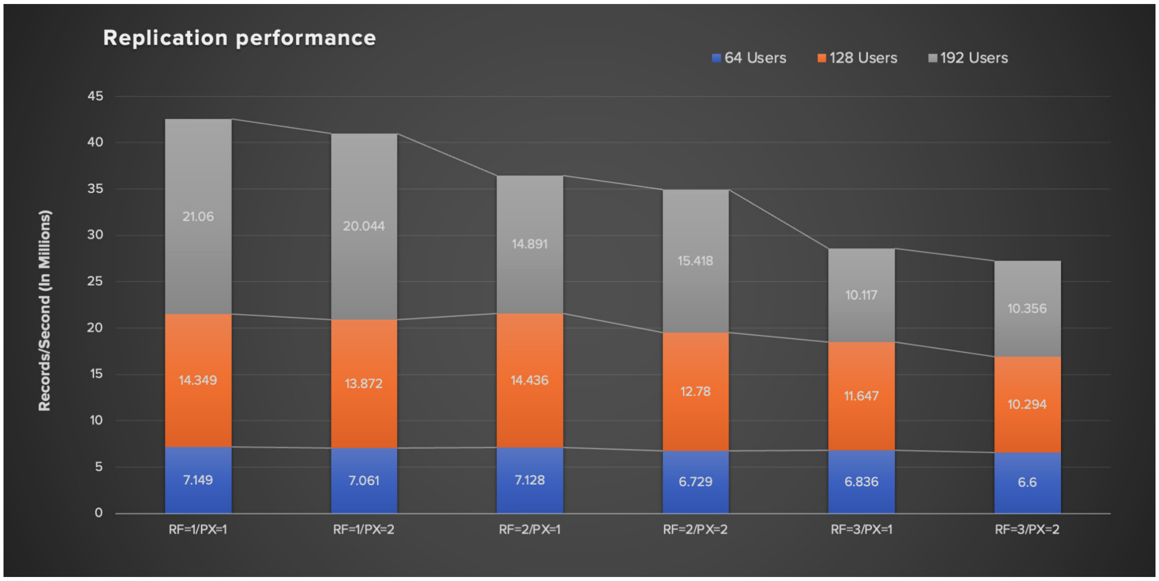


Figure 25. Kafka and Portworx replication performance

The results show that the performance with the replication factor for Kafka and Portworx set at 2 (RF=2/PX=2 or2x2) was similar to the performance when Kafka's replication factor was set to 3 and Portworx was set to 1 (RF=3/PX=1 or 3x1), when the load on the system (64 users) was within the resource limits. Once the load increased (128 and 192 users), clearly 2x2 performed better than 3x1. Also, it was very clear from the test results that increasing the Portworx replication factor from 1 to 2 did not cause much of a decrease in throughput in comparison to the increase in Kafka replication. This is clearly visible from the 192 users' throughput numbers for Portworx replication at 1, which went from 21.06 million/sec with RF=1 to 14.891 million/sec with RF=2 (29% reduction) to 10.117 million/sec with RF=3 (32% reduction). That is a reduction of about 52% in the throughput from Kafka RF=1 to RF=3. In contrast, changing Portworx replication from 1 to 2 while the Kafka replication factor is 1, 2, or 3 is less than 10%. This confirms that replication at the application level has more latency than replication at the storage level. Alternatively, it can be understood that Portworx replication does not add much of an overhead, but it offers better availability for the stateful application.

Replication Performance Test #2

We also carried out a test to see if there was any performance overhead when Portworx replication was set to 2 along with Kafka replication at 2, and the compared results when there was no Portworx replication but Kafka replication was set at 3. The second scenario can be seen as the equivalent of running Kafka in a non-Kubernetes environment. We performed similar scalability tests as we performed in the above section, but we varied the Portworx replication settings of "repl" between 1 and 2 by using the corresponding StorageClass object when starting the Kafka cluster and setting the topic replication factor to either 2 or 3.

Component	Value
Number of Kafka Brokers	4 and 8
Storage	Local NVMe drives
Kafka Replication Factor (RF)	2 or 3
Portworx Replication (PX)	1 or 2
Producer Batch Settings	linger.ms=30, batch.size=64K, compression.type=lz4, acks=all
Users (Producers)	2x, 4x, 8x, 16x of brokers
Ingested Data	Flog data for 10 minutes per test

Table 8. Replication Performance Test #2

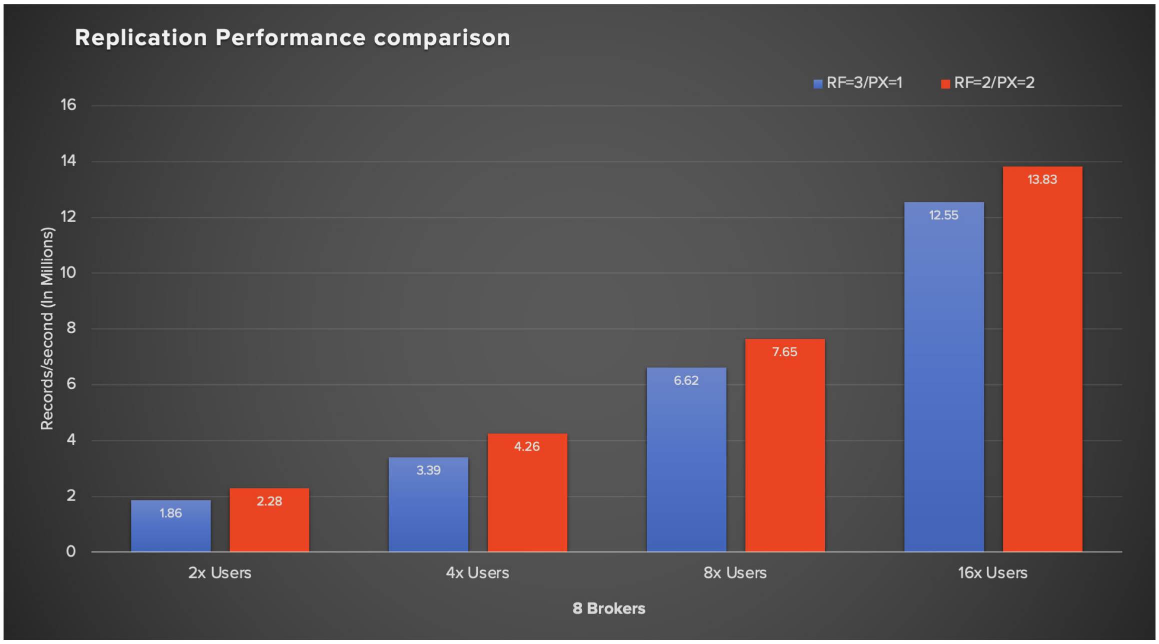


Figure 26. Replication performance tests with 8 brokers

As you can see from the above graph, changing the Portworx replication factor from 1 to 2 doesn't negatively impact the overall performance of the application. We see higher performance when the replication factor is set to 2 for both Kafka and Portworx in comparison with the scenario where Kafka replication factor is at 3 and Portworx replication is at 1.

We observed that replication at the application level has a higher impact than replication for storage. The following graph has the same data as the above graph for both four-broker and eight-broker Kafka clusters, but it is depicted in a line graph for better understanding.

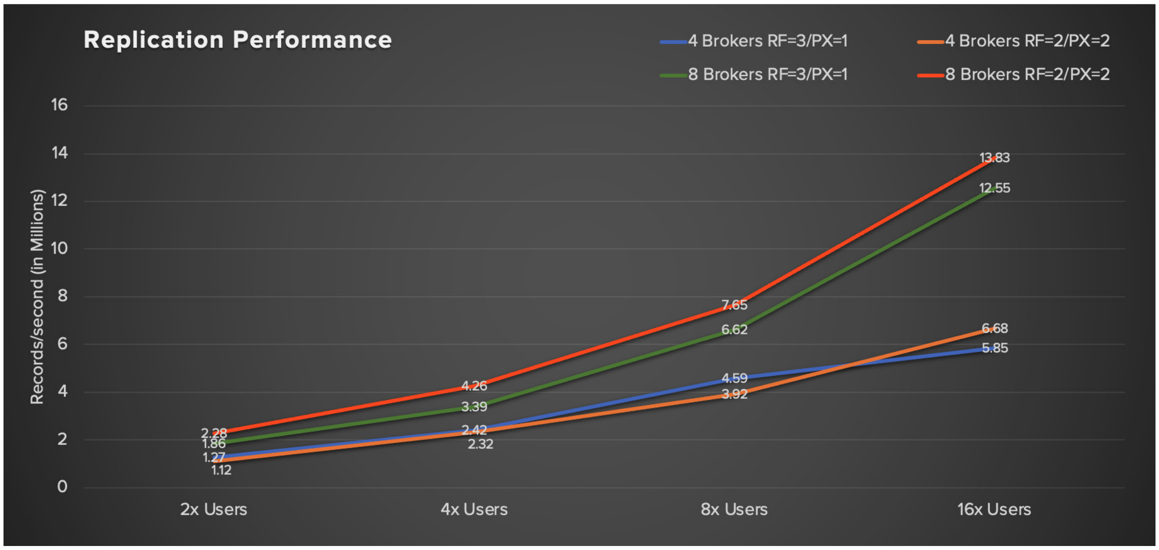


Figure 27. Replication performance comparison between four and eight brokers

Replication is one of the key elements that enables higher availability and durability of data. The above test illustrates that Portworx replication did not add any overhead to the overall performance. Enabling Portworx replication bridges the gap between the application and Kubernetes by providing better availability with lower failover time and improved operational efficiency. There is no need to perform any partition rebalancing because the broker pod always comes back up with the same volume with the data.

Added Benefit: Space Savings of Kafka on Pure Storage FlashArray

Aside from the standard benefits of FlashArray, including all-flash performance, simplicity, availability, durability, and non-disruptive upgrades, this storage array solution offers additional space saving to applications like Kafka due to its data reduction benefits through inline deduplication and compression. Saving space gives you the flexibility to expand data sources or increase data retention, as applicable.

FlashArray deduplication operates on a 512-byte alignment, with variable block size ranging from 4K to 32K. FlashArray eliminates duplicate copies of data within a storage volume or across the entire storage system, and it uses pattern recognition to identify redundant data and replace them with references to a single saved copy. FlashArray also has inline compression that reduces data and lets it use less capacity than the original format. Combined with deep reduction, compression yields a better data reduction.

With Kafka's topic replication factor, which replicates topic-partitions across brokers for higher availability, the replicas between the brokers are the same copy. When placed on FlashArray, those replicas can be deduplicated, resulting in space savings.

If no compression is enabled at the producer or the broker level, you will experience better data reduction through compression with FlashArray. Even if compression is enabled at the producer or broker level, depending on the type of compression codec, such as lz4 or snappy, you will get further data reduction benefits through compression.

The data reduction rate can vary anywhere from 30% to 400%, depending on the replication factor settings for Kafka and Portworx, as well as the compression settings. As a rule of thumb, the data reduction through deduplication is almost equivalent to that of the replication settings. For example, a Kafka replication factor of 2 and a Portworx replication factor of 2 can yield up to 4:1 data reduction through deduplication, while a Kafka replication factor of 3 and a Portworx replication factor of 1 can yield up to 3:1 data reduction through deduplication.

> NOTE: The above data reduction numbers are a suggestion based on replication settings and not a guarantee because the actual data reduction depends on factors such as data type, FlashArrau Purity version (data reduction improvement is an ongoing item), and system busyness.

Best Practices

The following sections offer the best practices or guidelines to follow when running Kafka on Kubernetes with Portworx. This is not an extensive list because these are based on our internal testing of the application and reflect a subjective view of what offers the best experience, considering performance, stability, or availability when running Kafka on Kubernetes.

Back-end Storage

Because Portworx is a software-defined storage solution that provides container-granular storage, you can use any backing stores, including directly attached NVMe/SSD or HDD storage, on-prem SAN like FlashArray, or cloud-based block storage such as Amazon EBS or Azure Disk. The Kafka application is all about high performance, higher throughput, and lower latency, so any storage based on flash would be the preferred choice.

Evaluate the flash storage based on your throughput needs. There are pros and cons to consider when deciding between locally attached NVMe/SSD drives and using enterprise all-flash storage. The direct-attached storage model offers higher bandwidth when a similar server with locally attached NVMe drives is added to the cluster, but for higher availability of the data you have to mirror the data between the drives within the server. In addition, direct attached storage doesn’t offer any additional data services like encryption or compression.

Meanwhile, FlashArray//X is the first mainstream, 100% NVMe, enterprise-class, all-flash array that offers high performance with in-built data services like encryption, data reduction through deduplication and compression, data protection through RAID-HA, and non-disruptive upgrades. Select the FlashArray model that’s right for your organization based on your application bandwidth requirements, data retention needs, and workload profile.

Broker Pod Availability

The advantage of running applications like Kafka on containers is the app-level abstraction. This enables container orchestration platforms like Kubernetes to scale the containers as needed and manage container failures seamlessly. If a Kafka broker pod fails in Kubernetes due to node failure, Kubernetes with the help of Portworx can respawn the broker pod on a node that has the Portworx replica. This is enabled by STORK (Storage orchestrator runtime for Kubernetes). STORK performs health monitoring of Portworx services, kubelet, and other components of the system. When a broker failure is detected, it will react faster than kube-scheduler in rescheduling the broker pod to a healthy node.

When a failed broker is respawned on a new node with access to the original volume, the broker has all of the topics and partitions that it needs for it to join the list of in-sync replicas. Hence, the broker does not need to replicate huge amounts of data from the “leader” across the network. Instead, it just needs to catch up with the offsets that were added since the broker went offline. This feature allows the Kafka broker pod to be available within a short period, the time that STORK takes to reschedule the pod, as opposed to hours or days in a non-container solution, especially when the Kafka broker fails permanently.

The broker pod's availability time after a failure is generally around three minutes or more because, by default, STORK scans every two minutes for any node failure and, once it finds a node failure, it waits for a minute before it attaches the volume to another node. STORK scan intervals are configurable; to reduce this time by half, you can change the parameter **–health-monitor-interval** from the default of 120 seconds to 30 seconds:

```
kubectl edit deploy stork -n kube-system
deployment.apps/stork edited
```

Unless the broker respawns time of three minutes or more is not good enough, you can consider changing the settings.

> NOTE: In the absence of STORK, the default kube-scheduler timeout to reschedule a pod after a node failure is over five minutes because the pod-eviction-timeout within kube-controller-manager is five minutes.

Kafka Replication with Portworx

Kafka, having been designed as a distributed and horizontally scalable streaming system, relies on the replication factor of topic-partitions to achieve higher availability of the data if a broker has to fail. As such, for production readiness you should never run Kafka with the default replication factor of 1 even if you are running the application on Kubernetes with Portworx. The general recommendation is to use either 2 or 3 for the replication factor, which allows either 1 or 2 broker failures and still gives you access to the data.

Portworx enables high availability of the application data through replication at the storage level. Portworx allows you to set the replication between 1 and 3, where 3 offers the highest level of data protection and availability.

Given that both Kafka and Portworx provide replication, what is the ideal replication setting to run Kafka on Kubernetes with Portworx?

Kafka RF*	PX Repl	Kafka Availability		Broker Recovery & Rebalance Time		Space Usage
		Pod Failure and Rescheduling	Storage Node Failure	Pod Failure and Rescheduling	Storage Node Failure	
3	1	Can sustain two failures	Can sustain two failures	Seconds	Minutes/Hours	High
2	2	Can sustain one failure	Can sustain one failure	Seconds	Seconds	Higher
3	NA**	NA**	Can sustain two failures	NA**	Hours/Days	High

RF* = Replication factor

** Portworx and Kubernetes were not used in this scenario, but Kafka was set up with direct-attached storage.

Table 9. Kafka on Kubernetes settings

Based on our testing, all replication factor combinations across Kafka and Portworx show that the main decisions when architecting Kafka on Kubernetes are centered around availability, recovery times, and space usage. Our testing revealed that two options were ideal for running Kafka, depending on your business requirements around tolerance to failures.

Option 1: Fast Recoverability

With this option, the Portworx and Kafka replications are set at 2. With replication set to two, Portworx replicates the volume at the storage layer, which enables quick recovery from failures of pods, nodes, the network, and disks by quickly respawning the failed broker on another Kubernetes node with the original volume attached. During that short period, when the pod is rescheduled, Kafka's replication factor of 2 can handle the reassignment of leadership. Portworx reduces the overall broker pod failure time through STORK and brings back the broker with the original data, which significantly improves the data synchronization by the failed broker with its surviving replica. This eliminates the need for Kafka admins to perform any partition rebuild over the network after hard node failures. This option offers quicker recoverability, better availability, resilience, and operational efficiency gains at the cost of some additional storage.

Option 2: Kafka Standard Configuration
With this option, the Kafka replication was set at 3 and Portworx replication was set at 1.

This is the standard configuration that most Kafka users are used to. This is equivalent to running Kafka on a non-Kubernetes environment with Kafka replication at 3, but with some additional benefits. With this option, during a pod failure or a node failure where the node doesn't have the Portworx replica, Portworx can still reattach its virtual volumes from the surviving nodes to the respawned broker pod on another node. However, if the node that fails owns the Portworx replica, the broker will be unavailable until recovery occurs. During this time, Kafka is still operational as it has two other replicas. When the failed node comes back, the respawned broker can synchronize the data from the other Kafka brokers.

It is still operationally better to have Portworx with a single replica than to have storage directly on the hosts as in the traditional setup because Portworx can move and reattach its virtual volumes to other Portworx nodes. This is possible as long as it has an available replica, even if the failed node running the broker is unavailable. This is not possible with directly attached storage.

Producer Settings

The producer configurations in Kafka can play a major role in achieving certain key characteristics like message ordering, message durability, overall throughput, and performance. While there are various producer configurations, we will focus on the following configurations based on our internal testing.

Message Durability
The message durability in Kafka is controlled through the acks setting, which supports three values – 0, 1, and all (-1). An *ack* is an acknowledgment that the producer gets from a Kafka broker to ensure that the message has been successfully committed to that broker. The config acks are the number of acknowledgments the producer needs to receive before considering a successful commit.

Settings for acks include:

- acks=0: You should never set acks to 0 in a production environment because the producer does not wait for a response and assumes the write is successful once the request is sent out.
- acks=1: This is the default setting, and the producer will wait for the ack and will consider a successful commit when it receives an ack from the lead broker, but the leader doesn't wait for confirmation from all replicas. The trade-off with this setting is to tolerate lower durability for better performance.
- acks=all: This setting also makes the producer wait for the ack and, in this case, the leader will wait for the full set of in-sync replicas to acknowledge the message and to consider it committed. This gives the best available guarantee that the

record will not be lost as long as at least one in-sync replica remains alive. The trade-off with this setting is to tolerate lower throughput/higher latency for better durability beause the lead broker has to wait for acknowledgments from replicas before responding to the producer. Our recommendation is to use acks=all if you want the highest available guarantee of data.

Message Throughput

To optimize throughput, the producers need to send as much data as they can at a faster rate. If the producers are sending several messages to the same partition in a broker, they can be sent as a batch. Generally, a smaller batch leads to more requests and queuing, which can result in higher latency. Increasing the batch size, along with time spent waiting for the batch to fill up with messages, can yield higher throughput and reduce the load on producers. It also reduces the broker's CPU overhead to process each request if those were all sent at the time. Here are some of the key configurations that play a vital role in achieving higher throughput:

- batch.size: This value allows the producer to batch the messages to the size set before sending the messages to the broker. The default batch.size value is 16384. If your message size is higher than the batch.size value, then you should increase your batch.size value to have better throughput with improved latency.
- linger.ms: This value allows the producer to wait until a specific number of milliseconds before sending the data to the broker. The default setting is 0, meaning no delay. The trade-off with this parameter is the added latency.
- compression.type: This value is the compression type for the data generated by the producer. The default value is none (or no compression). Valid values are none, gzip, snappy, lz4, or zstd. If compression is enabled, the producer compresses the completed batch. Hence, more data can be sent to the producer. Compression helps to send more payloads for the same batch size, which again can improve the throughput.

The use of compression comes with additional CPU cycles at the producer level and the compression ratio. Gzip offers the best compression, but it comes with very high CPU usage. On the other hand, lz4 has the lowest CPU usage, but the compression ratio is lower than every other compression type. Snappy and zstd use a moderate level of CPU and provide a medium level of compression ratio compared to the others. For performance reasons, we do not recommend using gzip. You can choose between lz4 and snappy based on your requirements.

To get better throughput, update both batch.size and linger.ms parameters. Increasing the batch.size value without changing linger.ms from 0 might not yield the throughput you expect and, at the same time, you might be adding more load to both producers and brokers by sending way too many smaller requests. If you want higher throughput with some added latency, we recommend that you increase the values for batch.size and linger.ms, and enable compression at the producer level.

There is no one specific setting for batch.size and linger.ms that anyone can recommend. The batching depends on various factors like the type of data, its compressibility, available CPU, and network resources. Hence, we recommend testing your data to identify the right settings for these parameters to achieve higher throughput.

Other important settings are:

- socket.send.buffer.bytes: The size of the socket send buffer. The default is 100KB. For higher throughput environments, this value might not be sufficient, and we recommend increasing this value. You can consider increasing this value to 8MB or 16MB if you have high-bandwidth networks and enough memory. If not, you can set this as low as 1MB.
- buffer.memory: The total memory (in bytes) that the producer can use to buffer records that are waiting to be sent to the broker. The default value is 33,554,432 or 32MB. If you have a lot of partitions, you might want to increase this setting.

Consumer Settings

Like the batching of messages with producers, Kafka batches messages for consumers to achieve higher throughput. The following settings customize message batching:

- socket.receive.buffer.size: The size of the socket receive buffer. The default is 100KB or 102400 bytes, which might not be sufficient if you have a higher volume of data to be published and read. You can consider increasing this to 8MB or 16MB if you have high-bandwidth networks and enough memory. If not, you can go down to 1MB.
- fetch.min.bytes: This parameter sets the minimum number of bytes expected for a fetch response from a consumer. Increasing this value will reduce the number of fetch requests made to the broker, reducing the broker CPU overhead to process each fetch.

Broker Settings

Here are some useful broker settings:

- compression.type: The compression type for the data to be performed by the broker. Valid values are none, gzip, snappy, lz4, zstd, and producer. A value set to “producer” retains the original compression codec set by the producer. Our recommendation is to set this value to “producer” so that the producer can take the responsibility of compressing the data and allow broker CPUs to be used for other critical functions.
- Partitions: The topic partition is the unit of parallelism in Kafka. Both partitions write (through producer/broker) and read (through consumer) can be done fully in parallel. In general, the more the partitions, the higher the throughput you can achieve. There are some disadvantages with too many partitions; for example, more partitions may increase unavailability and also require more open file handles. There is no perfect number for partition settings. You can determine the value based on your throughput requirements.

You can find more information about settings for the producers, consumers, and brokers in the Apache Kafka documentation.

Portworx Best Practices

To review the prerequisites and best practices for running Portworx, see the Production Readiness topic.

Volume Management

Portworx volumes are thin provisioned by default. Ensure that the volume capacity is monitored and necessary actions are taken through PX-Autopilot to avoid any outages for stateful applications such as Kafka.

High Availability of Stateful Applications

Applications like Kafka that need to be highly available and resistant to any failures in the worker node, including CPUs, memory, drives, or power, should use Portworx replicated volumes.

Portworx allows up to three replicated copies of the volumes, and the replication setting of two is generally recommended. Portworx also offers the rack parameter that can be used to place replicated volumes across failure domains.

StorageClass

Portworx allows administrators to pick and choose or create “classes” of storage. The below example creates a StorageClass object for a PVC with a replication factor of two, with higher I/O priority.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: px-storageclass-kafka
provisioner: kubernetes.io/portworx-volume
allowVolumeExpansion: true
parameters:
  repl: "2"
  priority_io: "high"
```

For all the parameters supported by Portworx, please check this table in the "Dynamic Provisioning of PVCs" topic.

Conclusion

The idea of switching mission-critical streaming applications like Apache Kafka into a container orchestration platform such as Kubernetes is getting increased attention because container orchestration platforms offer numerous benefits, including agility, scalability, automation, and simplified operations such as upgrades, scaling, restarts, and monitoring.

Despite the benefits, deploying Kafka on Kubernetes comes with its own set of challenges, especially at the storage layer, due to the statefulness of Kubernetes and its high availability requirements.

As explained in this paper, Portworx by Pure Storage bridges the gap between the application and the container orchestration platform, and it offers the following advantages::

- Gain operational efficiency in managing a scalable Kafka cluster through improved broker failover time and fully automated storage capacity expansion
- Enhance security with encryption-at-rest for Kafka broker data
- Enable migration of Kafka applications and data between environments like prod, dev, test, and across cloud providers or data centers
- Gain additional storage savings through deduplication and compression on Pure Storage FlashArray

The combination of Kubernetes and Portworx enables clear architectural benefits that are critical to the mission-critical Kafka support team.

Supporting Documentation

- Portworx documentation: https://docs.portworx.com/
- Apache Kafka documentation: https://kafka.apache.org/documentation/
- Kubernetes documentation: https://kubernetes.io/docs/home/

About the Author

Somu Rajarathinam is the Pure Storage solutions architect responsible for defining application solutions based on the company's products, performing benchmarks, and developing reference architectures for applications on Pure. Somu has over 25 years of database experience, including as a member of Oracle Corporation's Systems Performance and Oracle Applications Performance Groups. His career has also included assignments with Logitech, Inspirage, and Autodesk, ranging from providing database and performance solutions to managing infrastructure to delivering database and analytical application support both in-house and in the cloud.

The Pure Storage products and programs described in this documentation are distributed under a license agreement restricting the use, copying, distribution, and decompilation/reverse engineering of the products. No part of this documentation may be reproduced in any form by any means without prior written authorization from Pure Storage, Inc. and its licensors, if any. Pure Storage may make improvements and/or changes in the Pure Storage products and/or the programs described in this documentation at any time without notice.

THIS DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID. PURE STORAGE SHALL NOT BE LIABLE FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS DOCUMENTATION. THE INFORMATION CONTAINED IN THIS DOCUMENTATION IS SUBJECT TO CHANGE WITHOUT NOTICE.

Pure Storage, Inc.
650 Castro Street, #400
Mountain View, CA 94041