The Expert's Guide to Running Apache Kafka on Kubernetes

portworx
by Pure Storage

WHO SHOULD READ THIS GUIDE?

Platform Architects building a Container-as-a-Service platform that offers Kafka as a stream-processing option to end users

Application Architects or Site Reliability Engineers (SRE) building or running a SaaS application on Kubernetes that requires a high-performance Kafka cluster

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

SUMMARY

Apache Kafka is a distributed messaging and stream-processing platform originally developed at LinkedIn in 2011 and donated to the Apache Software Foundation. The platform offers several broad capabilities, such as publishing, subscribing to and processing real-time streams of records and events, and durably storing them. These features enable a broad array of use cases, including but not limited to message queues, event processing, website activity tracking, log aggregation, etc.

Recently, with the rise of containers and microservices, IoT, and Big Data, Kafka has become one of the most popular open-source tools employed in many applications and data platforms. In particular, more people are interested in integrating Kafka in their containerized applications and container orchestration platforms.

In this guide, you'll learn best practices for running Kafka on Kubernetes. In particular, we discuss:

- Kafka's features and use cases
- Architecture and innovative solutions (brokers, topics, consumers, producers, etc.)
- Important configuration for Kafka's High Availability (HA), high performance, throughput, etc.
- Key prerequisites for the stable deployment of Kafka on Kubernetes
- Building a highly available Kafka cluster on Kubernetes using Kafka's built-in features and Portworx volume replication
- Storage requirements for Kafka in Kubernetes and how to meet them using Portworx
- How to make Kubernetes-based storage for Kafka more secure
- How to ensure Kafka's fault tolerance using cross-cluster migration
- Using the Portworx platform to enable efficient Kafka disaster recovery and backup

1. INTRODUCTION: WHY KAFKA?

Kafka had a great rise in popularity over the last couple of years, becoming the platform of first choice for many tech companies and applications. This trend is the result of a combination of several factors: the spread of data-driven architectures, the proliferation of data (IoT, Big Data), and the rise of microservices architecture.

1.1 Data-Driven Realities

Over the 2010s, we have witnessed the spectacular growth of data-driven and event-driven applications as a result of the reduced cost of cloud infrastructure, the proliferation of Big Data, and the development of new, more robust business analytics and machine intelligence methods. Real-time data has become a crucial component of applications' business logic, feature personalization, and dynamic adjustment of

pricing. In this context, the ability to instantly process data events becomes the core requirement of many applications (online games, e-commerce, social media platforms, etc.).

The event-driven model built into Kafka allows multiple actions to be triggered based on some metric. Also, Kafka enables sending messages to multiple event handlers in different subsystems, which provides scalability and resiliency required by many business applications.

1.2 New Sources of Data

Today, web applications are no longer the exclusive sources of valuable data. We see growing diversification of sources of data with IoT, sensors, self-driving vehicles, and literally any digital device turning into a source of real-time operational data and critical real-time insights that help manage commercial and enterprise digital systems. Growing streams of data should be processed more effectively to keep important systems operational and make customers happy. Kafka was designed to handle data and events processing in a distributed way with higher throughput and lower latency.

1.3 Microservices

Event-driven architectures fit well into cloud-native applications and microservices—an emerging approach for running distributed enterprise-grade applications at scale. Microservices are composed of multiple moving parts (micro-applications) that communicate over the network, sending and generating large volumes of messages and events.

Microservices run in distributed environments (e.g., computer clusters), and, correspondingly, event-processing pipelines that underlie microservices should also be distributed. Along with that, these stream-processing pipelines should have high I/O throughput, resilience, HA, and fault tolerance to keep microservices operational. Traditional message queue systems were not designed for emerging distributed use cases and microservices.

We need a stream-processing platform designed with new principles in mind. Kafka enables microservices to have architectural flexibility and can be used for applications that need pub-sub/streaming/event-driven models or any combination of the above.

1.4 What is Kafka?

Kafka is a message and stream processing platform that naturally addresses all of these innovative use cases. It is event-driven, distributed, multi-tenant, highly available, easily integrable with diverse data sources, and performant. It supports message queues, stream processing, and publish-subscribe in one uniform model.

In this article, you'll see how to make the most out of Kafka by integrating it with containers and container orchestration platforms. Once you have built a robust Kafka deployment on Kubernetes, the road to better data-driven and event-driven applications is open.

2. KAFKA FEATURES AND USE CASES

As we've already mentioned, Apache Kafka is a distributed message and event processing system that runs in a cluster of connected servers that can span multiple data centers.

Kafka can be used for the following purposes:

- **Publishing and subscribing to streams of records.** Multiple producers can produce messages to different categories, and multiple consumers can process them from these categories in parallel.
- **Storing messages and records.** Records published to Kafka can be stored in partitioned append-only logs distributed across the cluster for fault-tolerance. However, Kafka should not be used as a permanent storage location for records. Messages published to Kafka are usually retained until all customers have read them and then deleted or moved to another location (e.g., database or data warehouse).
- **Message stream filtering and transformation.** Published records can be processed using Kafka Streams API that enables message filtering, transformation, and analysis to inform real-time application logic and power business analytics platforms. For example, an e-commerce application might take a stream of sales and output a stream of price adjustments computed based on this data. After processing, messages may be sent to a data warehouse or passed to other consumers.
- **Consuming messages from diverse sources with the Connector API.** Kafka can consume messages in different formats from different sources. Producers can connect to any event and to message streams and pipe them into the Kafka cluster. Kafka supports Elasticsearch, Splunk, Cassandra, ActiveMQ, AWS Lambda, and many more.

The above-described basic functionality can be leveraged for a broad array of use cases. Here is a list of the most important ones:

- **Message brokers and message queues.** Kafka allows building a message processing pipeline that connects producers and consumers of messages. In this pipeline, Kafka acts as a middle layer distributing messages among consumers and ensuring that messages are not lost due to node failures. Message brokers are used for multiple reasons, such as decoupling data processing from data producers and/or buffering unprocessed messages. Kafka enables this model for microservices thanks to its distributed nature, built-in message partitioning, replication, and fault-tolerance.
- **Application activity tracking.** Modern mobile and web applications generate large streams of data, such as page views, clicks, messages, and other events that can be processed to get business insights, customize ads, personalize features, etc. Kafka can be used for efficient parallel processing of these events.
- **Metrics.** A common use case for Kafka is the monitoring of operational data. This can involve aggregating statistics from applications and producing centralized feeds of operational data.
- **Log aggregation.** In comparison to file-based log systems, Kafka provides a cleaner and more abstract view of logs as a stream of messages. It allows for lower latency processing of log data, saving on storage, enabling distributed data consumption, and providing support for multiple data sources.

- **Event sourcing.** Event sourcing is a style of application design where state changes are logged as a time-ordered sequence of records. Kafka's partitioned log architecture with strong ordering guarantees makes it a good fit for such types of applications.

3. KAFKA ARCHITECTURE

Kafka uses many conventional approaches for the design of distributed applications and message queue systems as well as some innovative solutions that make it a good fit for the above-described use cases.

Key architectural concepts of Kafka discussed in this section include topics, partitions, consumers, and brokers. You'll also learn how Kafka's approach differs from traditional message queues and publish-subscribe platforms.

3.1 Topics, Partitions, Segments

Kafka is built around a concept called a topic. A topic can be thought of as an abstract category to which records are published. Topics help abstract a message processing pipeline from how records persistence is implemented. Kafka topics are multi-subscriber and multi-consumer: each topic can have multiple consumer groups and multiple consumers within these groups.

At the lower abstraction level, topics consist of one or more partitions that represent commit logs stored in actual files. Partitions are atomic units of topics that store actual messages.

Partitions are designed as ordered and immutable sequences of records that are appended to a structured commit log. Each record in a partition has a sequential ID called an offset. This offset identifies each record within the partition and lets the message consumers attend to different points of time in a commit log.

Partitions are designed with parallelism in mind. If a topic has several partitions, messages to them are saved in a round-robin fashion. This is to ensure that the load between partitions is evenly distributed. Consumers listen to message events in specific partitions. Once the partition gets a message, it is processed by its exclusive consumer. Other consumers can process messages arriving at other partitions in parallel. In a sense, partitions work as a message queue, but one that supports parallelism.

Also, partitions enable distributed message processing by allowing the message log to scale beyond the size of a single server. As a result, any given topic can encompass an arbitrary amount of data.

Last but not least, partitions make storing messages and records more efficient. Each partition in a Kafka topic is subdivided into segments, collections of partition messages. Splitting partition data into segments makes delete and read operations cheaper. If you have all logs in a single file, seek operation becomes computationally expensive. Kafka has index files that store the offsets and physical positions of messages in the log files, which makes navigation across logs easier.

3.2 Consumers

As we've already mentioned, records in Kafka topics are processed by consumers which can run as a separate process on separate machines. Consumers can be joined into consumer groups organized by a topic or several topics. A consumer group may be described as a 'logical subscriber' to a group of topics.

Each consumer in any consumer group reads messages from exactly one partition of the topic. Because messages are load-balanced between partitions, they are automatically load-balanced between consumer instances as well.

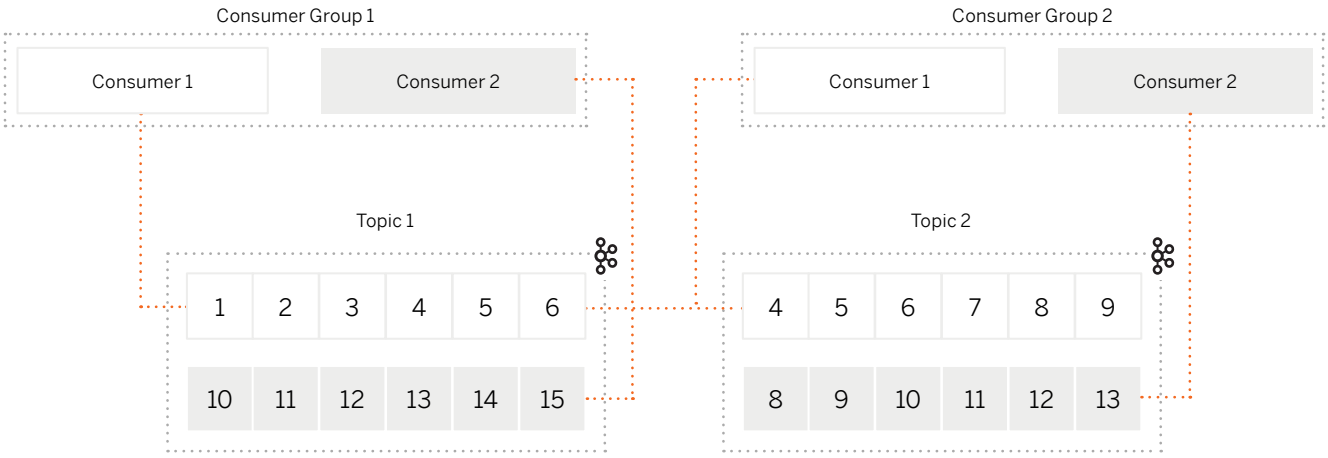

Image: Partition Distribution Among Kafka Consumers and Consumer Groups

Consumers have the liberty of attending to different parts of the commit log using message offsets. Kafka remembers the position of each consumer in the log, and the consumer can change it. Changing an offset may be useful when the customer wants to catch up with the previous messages or skip ahead to the most recent records. For example, if a consumer code has a bug, the consumer can re-consume certain messages once the bug is fixed.

At the same time, topics are broadcasted to all consumer groups. Thus, if all consumer instances belong to different consumer groups, each record is broadcasted to all the consumer processes.

You can see how this combination of consumers and consumer groups enables parallelism and a publish-subscribe pattern.

On the one hand, having multiple consumers read from specific partitions with load-balanced messages enables a message queue architecture. On the other hand, having multiple consumer groups enables a publish-subscribe pattern. There is no need to select between those two patterns—Kafka enables both message broadcasting and parallel processing simultaneously by design.

This provides an obvious benefit over traditional message brokers that usually implement either message queues or parallel processing. Similarly to a queue, the consumer group abstraction in Kafka allows dividing up processing over a group of processes. And, as with publish-subscribe, Kafka lets you broadcast

messages to multiple groups. Each topic and each broker in Apache Kafka has both of these properties.

Finally, it's worth mentioning that Kafka is a pull-based system by design. Consumers are free to pull messages whenever they want. This is important because consumers may have different consumption rates, and pushing large volumes of messages will make them too busy when the message volume is high. A pull-based system has a good property that the consumer simply falls behind and catches up when it's possible.

3.3 Brokers

A broker is a server that participates in the Kafka cluster. Each broker can be a 'leader' or 'follower' for a given partition. In Kafka, leadership and followership are distributed: each broker acts as a leader for some of the cluster partitions and a follower for the others, so the load is well-balanced within the cluster.

If the broker is a partition leader, it performs the following tasks:

- Handles all read and write requests and replicates them to followers
- Commits messages to the log when all in-sync replicas acknowledge that the message is received
- Mediates communication between consumers and producers
- Controls the overall state of the cluster

On the other hand, a 'follower' replicates partitions to their log. Any 'follower' can become a new leader if the current leader exits the cluster.

3.4 Other Innovative Features

Kafka incorporates several innovative features that enable high performance and throughput of messaging pipelines.

3.4.1 Message Set Abstraction and Batching

Along with topics, partitions, and segments, Kafka groups messages into an abstraction called a 'message set' that represents a batch of messages. This approach allows reducing the overhead of the network round trip when sending a single message at a time. Message batching results in larger network packets, bigger sequential disk operations, contiguous memory blocks, and other performance benefits.

3.4.2 Byte Copying

Kafka employs a standardized binary message format shared by producers, brokers, and consumers. This format allows data to be transferred without modification. Maintaining this common format helps optimize the most important operation: transferring persistent logs over the network.

3.4.3 Efficient Log Retention Policies

In a traditional logging system, logs are normally deleted after a certain amount of time. This works well for temporal data, but keyed data and mutable data—such as changes to a database table—require a different retention policy. For example, many applications would like to have a way to retain only the most recent changes to a database record—such as a user password—and delete all previous ones.

Kafka's log compaction feature allows implementing such a log retention policy. It ensures that Kafka retains at least the last known value of any given message within a log of data for a single topic partition.

Log compaction provides a more granular retention mechanism that allows reducing the log size while maintaining critical data. As a result, consumers can restore their state of the topic without Kafka having to retain a complete log. In Kafka, log compaction can be set per-topic, so you can have topics where retention policy is enforced by size or time as well as by the age of the record.

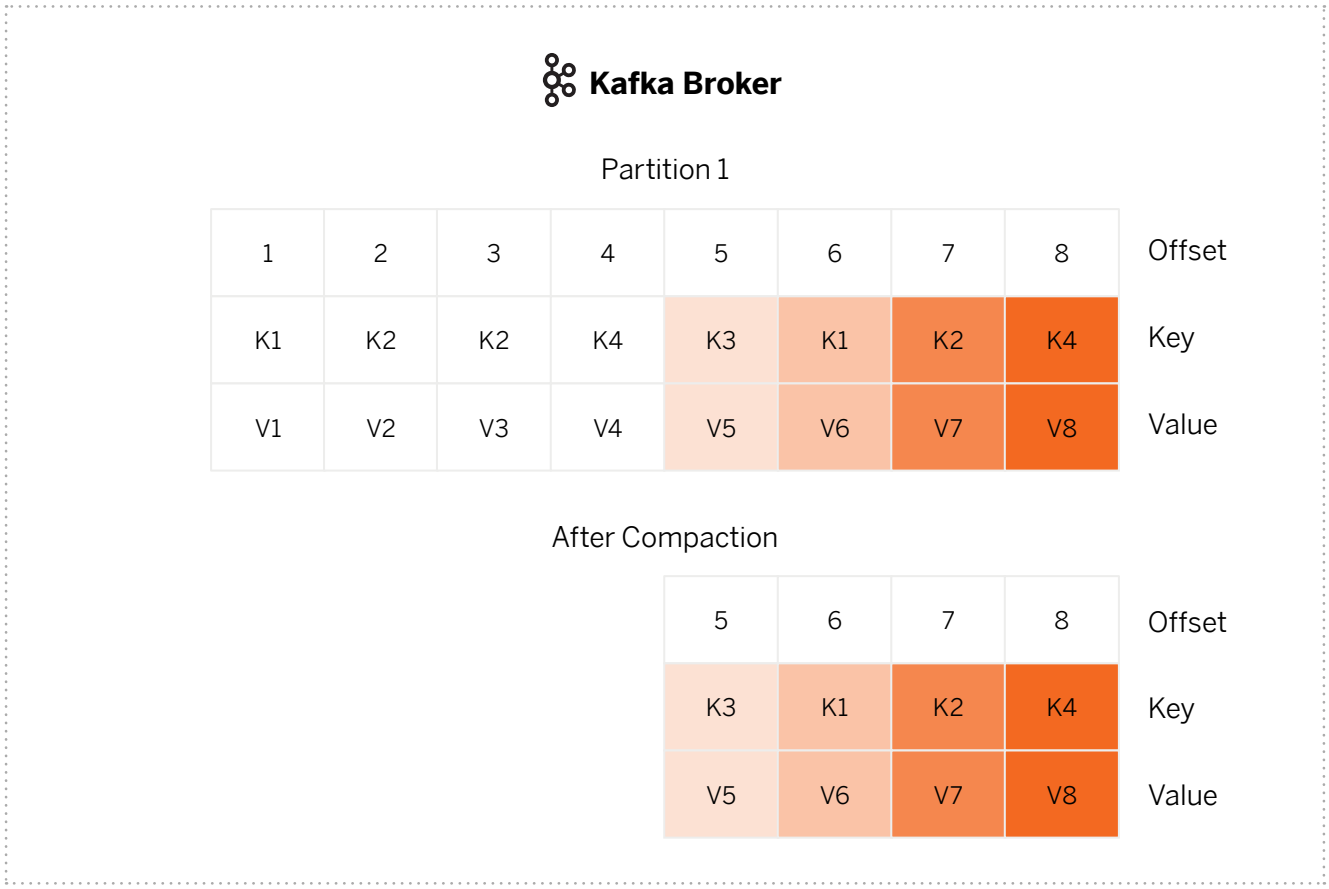

Image: Kafka Log Compaction

4. KAFKA PERFORMANCE AND CONFIGURATION BEST PRACTICES

As you now have a general idea of how Kafka works, let's discuss how to configure Kafka for high performance. In this section, we focus on general best practices applicable in any environment—on-premises bare metal, cloud, Kubernetes, etc.

4.1 Hardware, Runtime, and OS requirements

4.1.1 Java

Kafka is written in Scala and Java. To ensure optimal performance, you should run Kafka on the latest version of JDK (Java Development Kit). At the time of writing, JDK 12 (released on March 29, 2019) is the latest one. As far as JVM min/max heap settings are concerned, Kafka uses heap space very sparingly—you do not need more than 6GB on a 32 GB machine, for example.

4.1.2 RAM

In most cases, Kafka can perform optimally with 6GB of RAM for Java heap space. For large production loads, it is recommended to use machines with 32 GB or more. The LinkedIn setup described in the official documentation uses dual quad-core Intel Xeon machines with 24 GB of memory. You can estimate a required memory by assuming you want to buffer active readers and writers for 30 seconds and computing your memory needs as `write _ througput * 30`.

Also, enough memory should be provided to the main Kafka dependency, Zookeeper. The recommended minimum is 4GB, which can be specified in resource requests if running Kafka on Kubernetes.

4.1.3 OS Configuration

Kafka runs well on any Unix-based system and has been extensively tested on Linux and Solaris.

4.1.4 File Descriptor Limits

The number of open file descriptors directly depends on the number of open connections and partition count. By default, most Linux OS set a limit on this number, and it may be insufficient if your broker has many partitions and open connections. To determine the required limit, consider that a broker needs at least

```
(number _ of _ partitions)*(partition _ size/segment _ size)
```

to track all log segments in addition to the number of connections the broker makes. Kafka developers recommend at least 100,000 allowed file descriptors for the broker processes as a starting point.

When running Kafka on Linux, you can edit `/etc/sysctl.conf` and configure Ulimit to allow 100,000 or more open files.

4.1.5 Max Socket Buffer Size

This OS setting controls how much data a sender of the message can buffer into the network socket. The larger the buffer, the more throughput the network connection has.

4.1.6 Maximum Number of Memory Map Areas

The `vm.max_map_count` kernel setting on Linux defines the number of virtual memory pages that a process can use. Each log segment in any partition requires a pair of `index/timeindex` files, each of which consumes 2 map areas. Correspondingly, each partition with a single log segment requires 2 map areas. Thus, if you have 50,000 partitions on a broker, you'll need to allocate at least 100,000 map areas. By default, the value of `vm.max_map_count` that regulates this configuration is the region of 65,500. This number may be increased to avoid `OutOfMemoryError` when running Kafka.

4.2 Disks and Filesystems Configurations

Kafka immediately writes all data to the file system (OS page cache) and avoids storing it in the process's in-memory cache. This approach is preferred because storing data in process memory is associated with higher JVM memory overhead of objects and slow Java garbage collection when the used heap size is big. Instead, Kafka developers rely on OS-level memory caching system and flush settings.

Common recommendations for configuring a Kafka storage system and file system include the following:

- Use multiple drives to get higher writing and reading throughput. Disk throughput is always the performance bottleneck, and more disks are better.
- Do not share the same drives used for Kafka data with other applications and/or system processes to avoid high latency.
- Kafka's storage performance may be enhanced by RAIDing multiple drives together into a single volume or formatting each drive as its own directory.
- NAS (Network-Attached Storage) is not recommended to use for Kafka.
- SSDs can be used, but they don't deliver too many performance benefits due to Kafka's sequential disk I/O approach.
- Two filesystems that are used the most with Kafka are EXT4 and XFS. Recently, the XFS filesystem has demonstrated better performance than other filesystems with no deterioration in stability.
- Kafka documentation recommends using the default flush settings which disable application `fsync` entirely.

4.3 Topic Configuration

4.3.1 Replicate Partitions

For fault tolerance and HA, it is recommended to have 2 or more replicas for each partition in a topic. The precise choice of partition count depends on the throughput you want to achieve for your hardware. The more partitions, the greater parallelization of message processing you can achieve. For example, it is estimated that one partition on a single topic can produce 10 MB/s. You can use this estimate to assess how many partitions you'll need to secure jobs for each of the consumers in each of the topics. Alternatively, you can test by setting one partition per broker per topic and then increase partitions if more throughput is needed. A good rule of thumb is to keep total partitions for a topic below 10 and the total partitions in a cluster below 10,000.

4.3.2 Control Maximum Message Size

Messages fed into Kafka should not be too large because it increases seek time. If your producers generate large messages (~ 1 GB or more), you should consider splitting them into ordered pieces or using pointers to data (e.g., links to S3). In any case, messages should not exceed 1GB, which is the default segment size.

4.3.3 Calculate Data Rate of a Partition

The partition data rate is the rate at which data is produced to a partition. This metric is important for evaluating the minimum viable performance of consumers as well as the retention space and periods needed. The data rate can be calculated by the following formula: `average message size * the number of messages per second.`

4.3.4 Isolate Mission-Critical Topics

For a cluster with high-throughput SLOs, you can isolate some topics to the most performant subset of brokers. For example, if you have multiple transaction systems channeling data to your Kafka cluster, consider allocating a special subset of brokers for each of these systems. This may have an added benefit of decreasing the blast radius in case of a cluster incident.

4.3.5 Create a Clear Policy Regarding the Cleanup of Unused Topics

It may be useful to delete unused topics to free up some space. For instance, if no messages are received by a topic for N days, consider the topic to be dead and clear it from the cluster.

4.4 Consumer Configuration

4.4.1 Socket Buffer Size

Kafka batches messages both for consumers and producers to achieve a higher network throughput. To take advantage of this feature, Kafka users may need to increase the default TCP socket buffer sizes for brokers, consumers, and producers because the default value of 102,400 KB may not be enough. The TCP socket buffer size is controlled by the `socket.send.buffer.bytes` and `socket.receive.buffer.bytes` configurations on the consumer, broker, and producer. You can consider setting the socket buffers to 8 or 16MB for high-bandwidth networks (10 Gbps or higher). If your memory is scarce, 1MB would be a minimum viable configuration.

4.4.2 Avoid Over-Consumption of Data

Customers should get as much data as they are able to process at any given time. Therefore, it is recommended for them to digest fixed-sized buffers, preferably off-heap if running in a JVM. This will prevent consumers from pulling so much data that the JVM would spend all of its time performing garbage collection instead of processing messages.

4.4.3 Enforce Data Batching for Consumers

It may be desirable to let consumers digest messages only when a certain amount of data is accumulated. The amount of data to be accumulated before sending to consumers may be configured in `fetch.min.bytes` setting. If insufficient data is accumulated by the server, it will wait for more data to arrive before returning the fetch response. The default setting of 1 byte may be too small for your configuration. Setting this value higher can improve server throughput at the cost of some latency.

4.5 Producers Configuration

4.5.1 Control Durability and Retention of Producer Messages

Normally, producers keep their messages before the Kafka broker acknowledges the commit. You can decide how long producers should wait until removing the messages using the `acks` setting. It controls the number of acknowledgments the producer requires the leader to have received to consider a request complete.

If `acks` is set to `None`, the producer will behave in a 'fire and forget' fashion, deleting the messages once they are sent without waiting for the acknowledgement. It is recommended to set `acks` to some reasonable small value to ensure that messages are not lost but also not overloading the producer's retention space. Some trade-offs should be made here. The strongest available guarantee is `acks=all`. In this case, the leader will wait for the full set of in-sync replicas to acknowledge the record.

4.5.2 Configure Optimal Batch Size

There is a trade-off between different batch sizes for producers. Too small off a batch size can decrease throughput, whereas a very large size may result in the wasteful use of memory and higher latency. Ideally, you should select the batch size based on the producer's data rate, message size, number of partitions it produces, and available memory. Batch buffer sizes and batch sizes are controlled per partition by the `buffer.memory` and `batch.size` parameters.

4.6 Broker Configuration

4.6.1 Keep Your Broker Resources Consumption in Check

Kafka provides the ability to enforce quotas on requests to control the broker resources used by clients. You can enforce network bandwidth quotas that define byte-rate thresholds and request rate quotas that define CPU utilization thresholds as a percentage of network and I/O threads. These settings will help you to ensure that brokers are not out of CPU and memory.

4.6.2 Manage Log Compaction

Failed log compaction may lead to partitions that grow without bounds. Log compaction requires RAM and CPU cycles on the brokers to complete successfully. To provide more memory for log compaction threads, you can tune `log.cleaner.dedupe.buffer.size` (total memory used for log dedupe across all threads) and `log.cleaner.threads` on your brokers, but keep in mind that these settings affect heap usage on the brokers.

You should also consider log retention settings for standard retention logs:

- `log.retention.bytes`: the maximum size of the log before deletion
- `log.retention.hours`: the number of hours to keep a log file before deleting it
- `log.segment.bytes`: the maximum size of a single log file

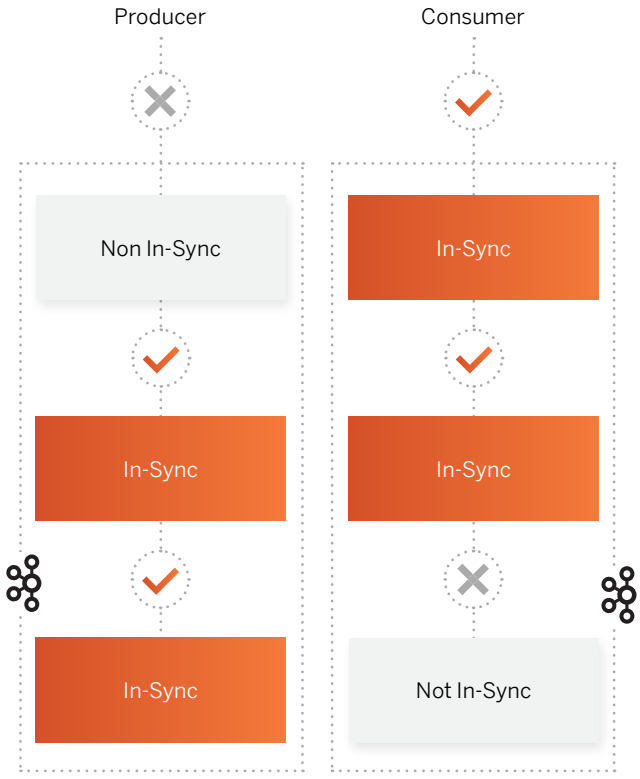

Image: In-Sync Replica Sets

4.6.3 Leader Rebalancing

Partition leadership must be evenly distributed among brokers in the cluster. Because leadership requires a lot of computing resources, leader imbalances in the cluster may result in low memory for some brokers. You should ensure that the setting `auto.leader.rebalance.enable` is set to True to let Kafka rebalance leaders if the leader imbalance exceeds `leader.imbalance.per.broker.percentage`.

5. BEST PRACTICES FOR RUNNING KAFKA ON KUBERNETES

5.1 Why Run Kafka on Kubernetes?

Setting up, configuring, and managing bare metal production Kafka clusters involves a lot of repetitive tasks and manual procedures that are prone to errors and require significant effort from your infrastructure and DevOps team. These tasks may include:

- Initial infrastructure provisioning, Kafka configuration, and deployment
- Configuring and managing ZooKeeper
- Decommissioning or adding new brokers
- Topic partition rebalancing
- Responding to critical events
- Performing manual backups
- Performing cluster upgrades

To get an idea of some difficulties involved in managing a Kafka cluster, let's take a look at the cluster upgrade scenario.

Normally, Kafka version upgrade to a new version (e.g from 2.2.x to 2.4.0) involves the following steps:

- Modify server properties such as `inter.broker.protocol.version` to match the currently installed version and `log.message.format.version` to match current message format version.
- Upgrade all Kafka brokers one at a time; shut down the broker, update the code, and restart.
- Change inter.broker.protocol version to a new version (e.g., 2.4).
- Restart brokers one by one again for the new protocol to take effect.
- Make one more rolling restart to upgrade `log.message.format.version` to a new version.
- Upgrade ZooKeeper if needed.

As you can see, Kafka's rolling upgrade involves three 'rolling restarts.' Each step of the process includes additional activities, such as secure login to brokers, configuration linting, graceful shutdown of brokers, broker initialization, etc.

All these operations are error-prone if done manually and are difficult to model declaratively using generalized infrastructure automation tools.

What is the solution? One option is to write some scripts to automate the process. However, this script should also be maintained, and possible conflicts with new Kafka versions might arise.

Another option is to use a deployment automation and orchestration tool that supports rolling upgrades and other administration tasks.

Kubernetes was designed precisely for this—to automate routine tasks of managing distributed applications reliably by providing a set of orchestration services needed to manage applications at scale.

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

Running Kafka in Kubernetes means that the cluster also leverages the power of containerization and developed infrastructure of tools. For example, you can easily connect Kafka to monitoring and logging pipelines running in Kubernetes and configure Kafka cluster communication with the outside world using Kubernetes Services, load balancers, Ingress, and other networking tools provided by Kubernetes.

If these advantages of Kubernetes seem compelling, you can now learn how to deploy and configure Kafka on Kubernetes in the most efficient way.

5.2 How to Deploy Kafka on Kubernetes

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

5.2.1 Using Kafka Helm Chart

Helm is the same to Kubernetes as apt to Linux and Brew to MacOs—a package manager of pre-configured Kubernetes resources called charts. A Helm chart contains configuration of all Kubernetes resources (Deployments, Services, service accounts, etc.) needed to deploy an application on Kubernetes. Charts embody best practices for configuring dependencies and Kubernetes resources for containerized applications. There are several Helm charts for Kafka available from the official Helm incubator repository, Bitnami, and Confluent.

5.2.2 Using Kafka Operator

An Operator is a method of packaging, deploying, and managing an application on Kubernetes. It is a more comprehensive deployment solution than Helm charts. A Kafka Operator may be thought of as runtime that manages the full lifecycle of Kafka, including deployment, rolling upgrades, authentication, data backup, etc. It can be used by a person who knows Kafka well without necessarily having a comparable level of Kubernetes expertise. There are several Kafka operators out there, including Kafka Strimzi and Confluent Operator, developed by the creators of Kafka.

5.2.3 Manual Deployment

Manual deployment gives you the most control over how Kafka runs on Kubernetes. If you plan to do a manual deployment of Kafka on Kubernetes, you need to ensure that Kafka brokers have persistent identity and storage.

Kafka is a stateful platform strongly dependent on data consistency and durability and persistent network identifiers. To run Kafka efficiently on Kubernetes, we need to deploy it using Kubernetes stateful components—StatefulSets and Headless services.

Unlike Deployments and Pods, StatefulSets provide stable identity (IPs) for each pod in a stateful set. If a pod from a StatefulSet is rescheduled, it gets the same IP as before. This is important because the address on which clients and consumers connect to brokers should not change. Also, StatefulSets guarantee ordered, graceful deployment and scaling and ordered rolling updates.

Another useful Kubernetes resource for manual deployment of Kafka is Headless services. By default, a Kubernetes Service load balances requests between Pods of the service; however, we need clients to access specific brokers where the partition resides. Headless services provide a single Service IP and allow interfacing with Kafka service discovery mechanisms without being tied to Kubernetes implementation.

5.2.4 Storage and Network Considerations

Kafka requires a low-latency network and storage with high throughput and fast linear writes. Dedicating fast media such as SSDs to brokers and enabling data locality can increase the overall performance of the system. We'll discuss Kubernetes storage for Kafka in the next sections in more detail. In what follows, we'll see how to meet all requirements for enterprise-grade Kubernetes deployment of Kafka using Kafka's built-in features, Kubernetes features, and Portworx Enterprise storage platform.

5.3 Kafka Fault Tolerance and High Availability

Fault tolerance and High Availability (HA) of a Kafka cluster are achieved through partition replication and quorum-based mechanisms of leader election which complement each other. Let's discuss these mechanisms in more detail.

The first mechanism is partition replication. By default, each partition has one replica that can be scaled up with a user-defined replication factor (2, 3). Users can set the replication factor on a topic-by-topic basis.

After the partition is created, Kafka distributes its replicas across the cluster. Replica distribution is done in a round-robin fashion to avoid stocking all partition replicas on a small number of nodes.

Also, for each partition, Kafka assigns a 'leader' of this partition from the list of running brokers. The partition 'leader' manages all data writes and reads to that partition and ensures that all partition replicas are up-to-date in the partition 'followers.' Followers consume messages for replication from the 'leader' much like a normal Kafka consumer and apply them to their replicated partition logs.

In Kafka, each broker can be a 'leader' of one or more partitions, and each 'leader' can be a 'follower' for other partitions at the same time. Obviously, such an approach helps avoid a single point of failure. Also, because the 'leader' role is more computationally intensive, it helps distribute the load between multiple brokers.

If a partition 'leader' fails for some reason, another leader is selected from brokers who have up-to-date replicas of the partitions—so-called in-sync replicas (ISR). This allows for automatic failover to new replicas. A leader failure event is managed by a 'controller,' who can be any broker elected for this role. The 'controller' is responsible for detecting 'leader' failures and replacing leaders for all the affected partitions. A new leader is elected based on the procedure discussed in the following sub-section.

5.3.1 Kafka's Approach to Leader Election and Quorum

Kafka's procedure for leader election deviates from the 'majority vote' approach used in many popular consensus algorithms, such as Raft, for example.

In this family of algorithms, the 'leader' failure is handled as follows. When the 'leader' exits the cluster, the first follower who receives an election timeout that indicates the absence of a 'leader' becomes a new

candidate and sends vote requests to other servers. If this candidate has a log that is more complete or as complete as the logs of other servers in the cluster, it gets a majority vote and is elected as a new leader.

In contrast, Kafka does not use a majority vote but dynamically maintains a set of ISR defined as replicas that are caught up to the leader. A partition leader regularly monitors the set of 'in sync' nodes and, if a follower exits or falls behind, removes it from the list. Any partition is guaranteed to have at least one in-sync follower because otherwise, no message will be committed. In fact, any write to a partition is considered committed only after all in-sync replicas have received the write.

This ensures high fault-tolerance for messages in a partition. Since messages need to be replicated by all in-sync replicas before commit, consumers only get those messages that will never be lost. This approach is better for the Kafka usage model where there are many partitions and ensuring leadership balance is important.

The discussed Kafka's HA model can be further strengthened with the built-in Rack Awareness feature. It allows spreading replicas of the same partition across different racks. This feature can be applied to different broker groupings, such as availability zones in EC2.

5.3.2 Extending Kafka High Availability for Storage on Kubernetes

As we showed, Kafka offers strong fault tolerance guarantees for topics including replication, efficient mechanism of in-sync followers, and leader election.

Additional HA guarantees can be provided by Kubernetes when you deploy Kafka on it. By default, Kubernetes knows how to recover failed pods and place them on new nodes. You can also configure liveness and readiness probes, Horizontal Pod Autoscaler (HPA), and implement a cluster auto-scaler to improve the durability of your Kubernetes-based Kafka cluster even further.

At the same time, however, running Kafka on Kubernetes means that you should take care of the Kubernetes HA. How to make Kubernetes highly available is beyond the scope of this article, but it's just worth mentioning that you should enable HA for Kubernetes cluster components (kube-proxy, kubelet, kube-apiserver, etc.) as well as nodes, load balancers, and other important components.

Also, keep in mind that although Kubernetes scheduler may ensure that the Kafka pod is relocated to another node, it does not ensure that it automatically has access to partition data. Therefore, it is important to consider what happens to the Kafka data when the broker moves to another node in the Kubernetes cluster.

> "When the new broker is started, Kafka will try to ensure that it is provided with the in-sync topic replicas from the cluster 'leader.' However, if the broker replicates data from scratch, this process may come at the cost of lower I/O and higher Kafka cluster latency during the rebuild. The recovery time of the failed node and the broker will depend on the volume of data to be synchronized and network latencies in the cluster during the rebuild phase."

Therefore, when running Kafka in Kubernetes, we need to devise a data replication strategy that enables faster failover to replicated partition data in case of broker failure. In the real-world production deployment, we need Kafka data to be instantly available to a new broker and the entire cluster to have high-throughput for service producers and consumers at any point in time.

5.3.3 Portworx Solution

The Portworx Enterprise container storage platform is a natural choice for addressing the challenges discussed above. In what follows, we'll discuss how Portworx works in general and how it can enable faster failover of Kafka brokers running in the Kubernetes cluster.

In a nutshell, 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.

At the most basic level, Portworx works as a software-defined storage virtualization solution that can aggregate all storage resources available in your cluster and create a unified storage layer to manage them and provide to containers on demand. The storage pool can aggregate different storage types and classes, including SANs, SSD, HDD, cloud block storage (EBS), etc.

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 factor and distribute those replicas evenly and dynamically across the Kubernetes cluster. Being tightly integrated into a Kubernetes cluster, Portworx has built-in cluster topology awareness that helps auto-detect availability zones, regions, or racks and provision replicas across them.

For these replicas, Portworx uses synchronous replication—each write is automatically synchronized for each replica. This secures data consistency among them. Also, volume replicas can be accessed from any node where Portworx runs.

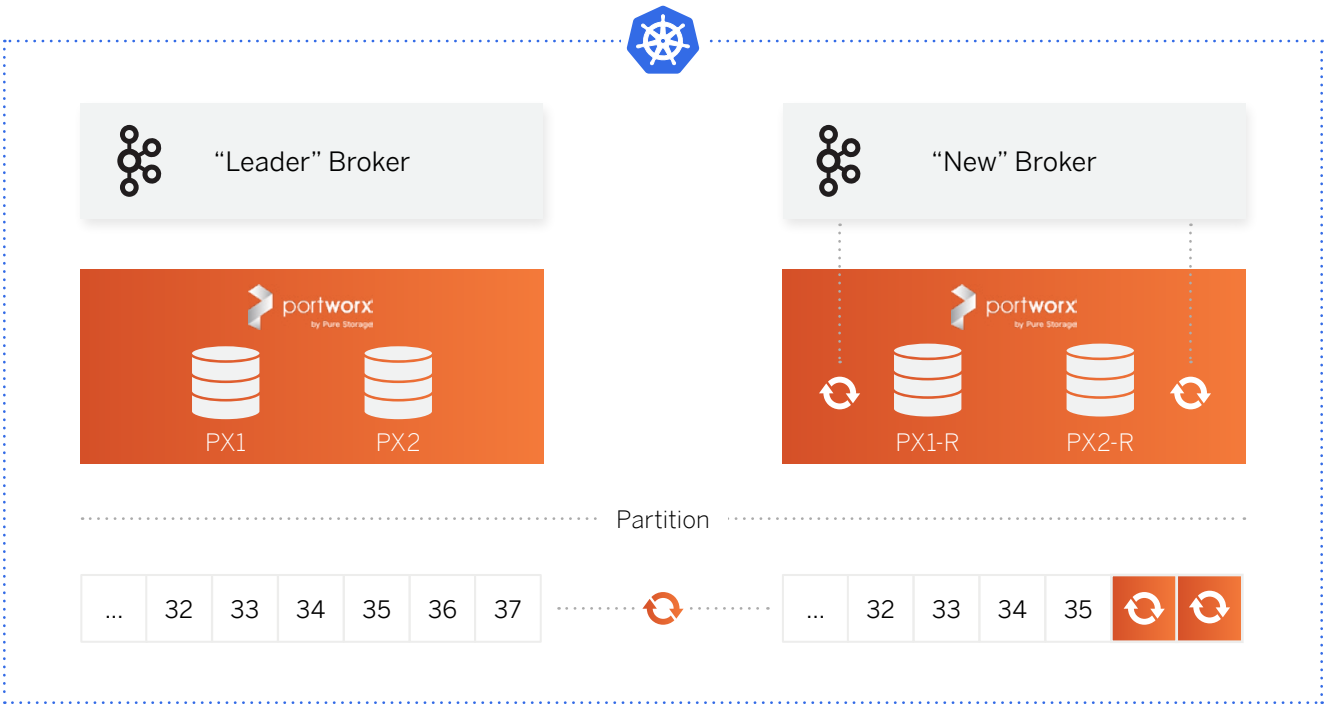

Image:Kafka Partition Replication with Portworx

Let's see how the Kafka broker can quickly rebuild with Portworx in the scenario discussed above.

Once Kubernetes node failure is detected by the Kubernetes API-server, Portworx will ensure that a new Kafka broker is placed on a node that has replicas of all Kafka topics. This is enabled by the Portworx Platform's storage-aware scheduling and hyperconvergence runtime called STORK (STorage Orchestrator Runtime for Kubernetes). STORK performs health monitoring of Portworx services, kubelet, and other components of the system, and when a broker failure is detected, it will react faster than kube-scheduler in rescheduling the broker to a healthy node. This in turn minimizes Kafka's ISR contractions/expansions.

More precisely, this means that when a failed broker is spun up to the new node, this node already has all topics and partitions for it to join the list of in-sync replicas. Correspondingly, the broker does not need to replicate large volumes of data from the 'leader' across the network. Thus, using a storage system for replication avoids the problem of network latencies. Once the data is rebuilt using the Portworx replica, the broker can quickly catch up with the leader, replicating only those offsets that were added when the broker was offline.

A special note should be made regarding Zookeeper replication strategies for Kubernetes. It is known that ZooKeeper is designed to store configuration and state information. It does not require large storage capacity because it keeps all state machines in-memory for high performance. ZooKeeper writes every change to the state to a durable Write Ahead Log (WAL) on storage media. If a server fails, ZooKeeper can quickly recover from the WAL by loading them directly into memory. Thus, because ZooKeeper keeps all data in memory and has a built-in recovery mechanism, we don't need to configure Portworx replication for it.

Portworx unified storage layer can also automatically create disks based on input disk templates whenever a new node spins up. This allows for automatic provisioning of storage when auto-scaling functionality is embedded.

In sum, Portworx Enterprise storage platform can be used to enable storage- or device-level HA, which complements Kafka application-level HA and Kubernetes cluster HA. With Portworx, you can further improve the fault tolerance and resilience of your Kafka cluster on Kubernetes.

> "Overall, this approach significantly reduces the recovery time. As a result, Kafka continues to run at high performance, I/O, and throughput during the rebuild process.
>
> Also, since Portworx provides storage High Availability for Kafka, users can run fewer brokers with the same level of reliability while dramatically reducing compute costs. For example, running 3 brokers instead of 5 brings a 40% cost savings."

5.4 Kafka Security in Kubernetes

Kafka provides basic security features needed to protect a cluster. The platform ships with the basic mechanism for authentication, in-flight data encryption, and authorization. Let's briefly discuss them.

5.4.1 Authentication

Authentication allows verifying the identity of consumers and brokers connecting to Kafka clusters. Kafka provides SSL and SASL authentication methods. Among the SASL mechanisms, Kafka supports the following:

- SASL/GSSAPI (Kerberos) - starting at version 0.9.0.0
- SASL/PLAIN - starting at version 0.10.0.0
- SASL/SCRAM-SHA-256 and SASL/SCRAM-SHA-512 - starting at version 0.10.2.0
- SASL/OAUTHBEARER - starting at version 2.0

Kafka also provides delegation token-based auth, a lightweight authentication mechanism to complement existing SASL/SSL methods.

5.4.2 Data Encryption (SSL/TLS)

Kafka provides SSL/TLS encryption of data transferred between brokers and clients as well as consumers and producers. You can create SSL Certificate Authority and issue certificates for clients and brokers to enable SSL. This encryption mechanism encrypts only in-flight data passed along the network.

5.4.3 Authorization Using Access Control Lists (ACLs)

Kafka ships with a pluggable Authorizer and an out-of-box authorizer implementation that uses Zookeeper to store all the ACLs. The Authorizer enables assigning different roles and rights to different clients and consumers in the cluster. Kafka will check the ACL list to see if a given consumer has the rights to read a particular topic or whether a certain producer can post messages to a topic. Also, Kafka can be integrated with external authorization services.

5.4.4 Extending Kafka Security with Portworx

When running Kafka on Kubernetes, there are additional layers of security to be addressed: physical storage security, Kubernetes authentication, and RBAC and storage management security.

First, we should mention that SSL encryption in Kafka works only on in-flight data passed along the network. Data sitting in Kafka volumes is not encrypted by default. Ideally, organizations should protect at the application level but also secure the data layer along with it for added security.

Thus, when running Kafka in a distributed compute environment like Kubernetes and using third-party storage, it's reasonable to add this additional layer of disk encryption. You can encrypt your Kafka volumes with Portworx.

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

As we've mentioned, Kafka supports SSL and SASL authentication mechanism and authorization. They are enough for enabling authentication in a Kafka cluster. However, we need an additional authentication mechanism for users, applications, and services interacting with Kafka as part of a Kubernetes cluster and Kubernetes API server. Kubernetes provides many useful authentication methods, including client certificates, bearer tokens, an authenticating proxy, HTTP basic auth, SSL, and more. Also, Kubernetes has a built-in RBAC model that allows assigning different roles to pods and users in different namespaces.

Both Kafka and Kubernetes layers of security can be further extended with Portworx storage authentication and authorization that allow you to control how volumes are accessed and managed by Kafka users.

For authentication, Portworx accepts OIDC and self-generated JWT tokens, which makes it easy to use Portworx for enterprise-grade authentication systems, such as SAML 2.0, LDAP or Active Directory.

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

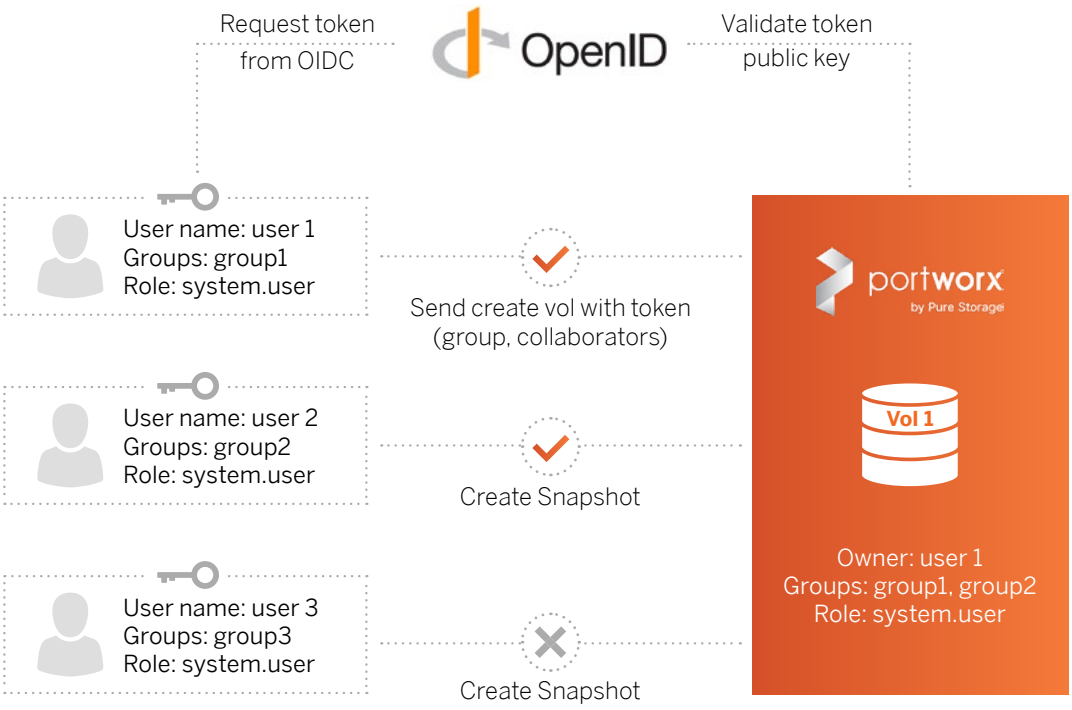

Image: Portworx Security Model

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

Kubernetes and Portworx RBAC models meet when the user aims to create a Kubernetes resource using a Portworx volume. Here, the user does not only need RBAC authorization from Kubernetes but also provides a token generated for Portworx that contains the roles and groups of the user trying to create a volume.

This completes our security model for Kafka in Kubernetes. By adding a storage security layer, we can have multi-modal security that covers Kafka application-level security, cluster-level security, and storage-level security, dramatically decreasing the potential vectors of attacks against your deployments.

5.5 Kafka Metrics Pipeline on Kubernetes

Real-time monitoring of Kafka consumption and production rates, network latency, request and response times, number of in-sync replicas, and other metrics helps in keeping your Kafka cluster performant and resilient. A stable monitoring pipeline thus becomes an intrinsic component of Kafka's daily administration.

Important monitoring targets for your Kafka deployment are the following:

- The number of produced messages, average message size, and number of consumed messages are important metrics for understanding production and consumption rate and calculating required resources for the Kafka cluster.
- Broker's network throughput, disk I/O, storage space, and CPU usage—monitoring these metrics will help you prevent out of memory and low disk issues beforehand.
- In-sync replica (ISR) shrinks and under-replicated partitions—for example, in-sync shrinks are good indicators that the data for that partition is in excess of the leader's resources.
- Partitions without an active leader and leader imbalances.
- Latency of end-to-end message delivery between Kafka producers and consumers.

Apache Kafka publishes these and other metrics via Java Management Extensions (JMX) by default. To collect Kafka metrics in Kubernetes, you'll thus need a solution that understands JMX metrics format.

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

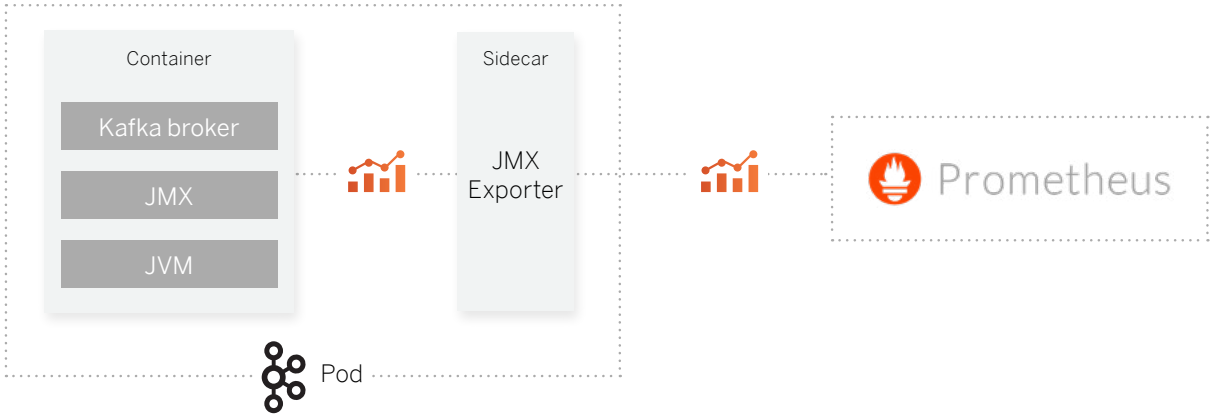

Image: Sending Kafka Metrics to Prometheus

To scrape Kafka metrics to Prometheus, you can deploy JMX Exporter as a sidecar container in Kafka pods and then configure the JMX agent in the Kafka container to report its metrics to the JMX Exporter (this can be set in the KAFKA_OPTS environment variable). Eventually, your Prometheus deployment can be configured to receive and process the JMX-exported metrics.

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

6. KAFKA CLUSTER BACKUP AND MIGRATION

Kafka replication is good at handling issues such as broker failure. However, there are many more disaster scenarios that necessitate a responsible backup policy, which can be implemented using Kafka built-in tools as well as third-party solutions. For example, what if your entire datacenter goes down? Or some topics are deleted due to Kafka's bug or the error of Kafka administrator? These scenarios require a clear back up policy for your Kafka cluster.

Overall, there are two broad options to back up your Kafka data using Kafka built-in tools:

- Real-time cross-cluster replication/migration with Kafka's MirrorMaker
- Using Kafka Connector to source Kafka topics to a remote database or backup store

6.1 Kafka Connect

Kafka Connect is a built-in component that allows users to connect to outside data sources to import and export data from/to them. To back up your Kafka data to an outside data store like Amazon S3, you'll need a 'sink' collector designed for this purpose. The Kafka community has a developed ecosystem of open source connectors to the most popular data services, such as Elasticsearch, Splunk, ActiveMQ, Amazon

Lambda, and more. You can find most of them in the Confluent Connector Hub. Overall, Kafka Connect is useful if you want to back up the most critical topics, but it does not provide a replica for your entire cluster. If you want to have a consistent and regularly updated replica of the entire cluster, you should consider using another option—Kafka MirrorMaker.

6.2 MirrorMaker

MirrorMaker is a built-in tool in Kafka that allows replicating partitions and messages across multiple datacenters and cloud regions. This feature can be used for cross-cluster backup and recovery or placing data closer to its consumer (a data locality requirement).

When using MirrorMaker, you mark your current cluster as a source cluster and the cluster you want to replicate your data to as a destination cluster. Later, source and destination clusters interact as producers and consumers according to the Kafka built-in paradigm. Data is read by the destination cluster and written to a topic with the same name in a destination cluster.

It's worth mentioning that source and destination clusters may be completely independent entities. In other words, a destination cluster may have its own topics and partitions along with those replicated from the source cluster.

So, cluster mirroring is not just for fault-tolerance. Its use cases include the following:

- Aggregation—Kafka deployment may consist of a single regional cluster in each datacenter and one aggregate cluster for the data warehouse. Regional clusters can collect data and mirror it to the data warehousing cluster using MirrorMaker.
- Cross-cluster replication
- Testing new Kafka versions

6.3 Kafka Backup and Recovery with Portworx

When running on Kubernetes, you also need a way to back up Kubernetes PersistentVolumes Kafka uses and underlying storage (e.g., SSD) that provide the physical capacity to these Kubernetes abstractions. Portworx enables Kubernetes-level backup policy and data backup policy for Kafka with the following tools:

- `PersistentVolume` snapshots
- 3DSnaps
- PX-migrate

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

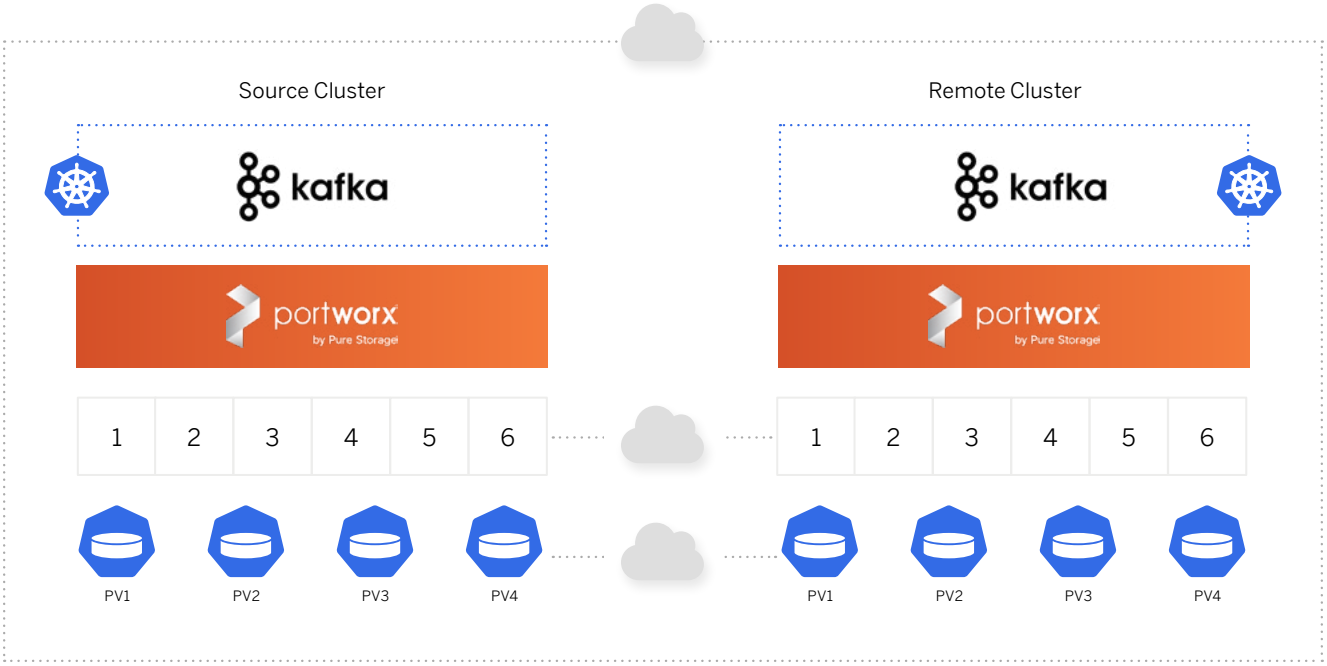

Image: Cross-Cluster Migration of Kafka Data with Portworx

The first method allows taking per-volume snapshots of Persistent Volumes. These snapshots can be used in `PersistentVolumeClaims` if Kafka data needs to be restored.

Also, using the scheduled volume snapshots, you can specify a snapshotting schedule for each volume in your Kafka cluster. Portworx will periodically take snapshots of Kafka volumes and store the most recent snapshots in your Kubernetes cluster according to the retention policy.

Snapshots created with Portworx can be stored locally or in the cloud using STORK (STorage Orchestration Runtime for Kubernetes). Cloud snapshots can be automatically uploaded to the configured S3-compliant endpoint (e.g., AWS S3).

Taking snapshots of application data at the storage level, however, may not be enough in a production setting. With such platforms as Kafka, you need to ensure that snapshots are consistent with the application's state. With Portworx, you can ensure snapshot consistency using 3DSnaps.

3DSnaps are application-consistent snapshots. For each 3DSnap, users can specify pre and post rules that are run on the application pods using the volumes. This feature allows users to pause the applications before the snapshot is taken and resume I/O after the snapshot is taken. In this way, the snapshots reflect the application's state.

7. KAFKA CLUSTER MIGRATION ON KUBERNETES WITH PORTWORX

What Kubernetes MirrorMaker does for the Kafka cluster running on bare metal, Portworx PX-migrate can provide for the Kubernetes-based Kafka cluster. PX-Migrate is a Portworx tool that lets you migrate data, configuration, and Kubernetes objects (ConfigMas, Secrets, etc.) across your Kubernetes clusters seamlessly and with almost no effort. The tool is tightly integrated with the Kubernetes Control Plane and is aware of all Kubernetes abstractions that should be migrated.

Common use cases for migrating a Kubernetes-backed Kafka cluster to a remote cluster may include:

- Maintaining a full up-to-date replica of the Kubernetes-based Kafka cluster
- Moving low-priority topics to secondary clusters
- Testing new versions of Kafka, Kubernetes, or Portworx using the same application and data
- Moving workloads from development/test environments to production without the need to manually provision the data
- Moving workloads from private on-premises clusters to cloud-hosted Kafka clusters like Amazon EKS or Google GKE

8. CONCLUSION

Deploying Kafka on Kubernetes is the first step towards transforming your message-processing pipeline into a distributed container-based service. Building a performant Kafka cluster is, in and of itself, a rather complex task that requires a lot of planning, configuration, making correct architectural decisions, and clear assessment of infrastructure needs. Deploying Kafka on Kubernetes introduces additional challenges and new layers of complexity associated with storage, microservices, fault tolerance, and HA.

Since Kubernetes' open sourcing in 2014, the Kubernetes community has done a lot to make the platform suitable for stateful applications such as Kafka. Much effort has been made recently to integrate Container Storage Interface (CSI) and provide stateful features like local persistent volumes, StatefulSets, etc. As we've found in this paper, Portworx can further extend this built-in functionality with container-granular storage orchestration.

To sum it up, key benefits provided by Portworx for Kafka deployment on Kubernetes include:

- Efficient Kafka volumes replication across Kubernetes cluster to enable storage HA and fault tolerance
- Data-aware scheduling. Portworx is tightly integrated with the Kubernetes Scheduler to enforce right broker placement decisions, which reduces time of the new broker syncing with the cluster. This means lower latency and higher throughput
- Extending Kafka built-in security with storage-level authentication, authorization, and ownership
- Efficient backup and cross-cluster migration

All these features enable seamless and smooth deployment of Kafka on Kubernetes, which meets all data persistence and performance guarantees defined by Kafka. We hope that the real-world Kafka deployment tips discussed in this article make your Kubernetes journey more comfortable and productive.

portworx
by Pure Storage

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