The Expert’s Guide to Running Elasticsearch on Kubernetes

portworx
by Pure Storage

WHO SHOULD READ THIS GUIDE?

Platform Architects building a Container-as-a-Service platform that offers Elasticsearch as a database option to end users

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

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

1. INTRO

Running stateful applications such as Elasticsearch as a microservice is not a trivial task because of how container orchestrators like Kubernetes treat the lifecycle of containers and pods. In essence, these resources are ephemeral entities with a short life span, depending on cluster state and application load. They move frequently across nodes to balance resources, but there is no automatic way to move data along with them.

Combined with the fact that Elasticsearch requires strong data availability guarantees, it is challenging to achieve high availability, consistent and predictable performance, and agile operations using the API of container platforms and container orchestrators alone.

Even as much progress is being made in exposing underlying storage and data management features such as persistent storage, snapshotting, and dynamic volume provisioning, in Kubernetes, many other features—including data high availability, automatic failover across fault domains, replication, data encryption, migration, backup and disaster recovery, and more—are required to confidently run Elasticsearch in Kubernetes in production.

Platform and SaaS architects are well aware of these requirements and choose to run Elasticsearch on Kubernetes because of the automation it provides. One of the most important areas of automation is around storage and data management.

In this guide, you'll learn how to meet these requirements for Elasticsearch clusters running on Kubernetes using a combination of native Elasticsearch tooling and the Portworx Enterprise Kubernetes storage platform.

Specifically, we'll discuss the following topics:

- Storage and data management requirements for stateful applications running as microservices
- Designing Elasticsearch clusters for High Availability (HA) and optimal performance using Elasticsearch best practices
- Implementing storage-aware scheduling and HA
- Ways to reduce Elasticsearch storage costs and enhance performance using a Class of Service (CoS) model
- Creating a multi-modal security model for Elasticsearch that encompasses application-, cluster-, and storage-security layers
- Efficient Elasticsearch data cross-cluster migration and replication

In the first part of this paper, we'll provide an overview of how to make Elasticsearch clusters more efficient by using built-in Elasticsearch features alone. Then, we'll add Kubernetes and Portworx into an equation to see how to combine three tools to achieve optimal performance in a microservices environment.

2. WHAT IS ELASTICSEARCH?

Elasticsearch is an open-source distributed analytics and search engine built on top of Apache Lucene. It provides a JSON-based REST API to access Lucene indexes and functions and adds a distributed system on top of Lucene to support cluster-based data storage and search functionality.

Elasticsearch supports numerous data aggregation and filtering tasks at search time in a single request, which makes it suitable for data analytics, monitoring, anomaly detection, data mining, text analysis, and even complex machine learning operations. Combining state-of-the-art data search with data analytics makes Elasticsearch great for the era of AI/ML and Big Data.

Elasticsearch is built for high performance with large pools of semi-structured and unstructured data. Among other things, this is achieved by efficient index design with dedicated, optimized data structures for each field type. For example, text fields use efficient inverted indexes, and numeric and geo fields are modeled as BKD trees.

Also, the platform provides such features as thread-pooling, queues, node/cluster monitoring API, data monitoring API, cluster management, etc.

In general, Elasticsearch supports the following use cases:

- *Free-text search,* enabling fast search of semi-structured and unstructured data for enterprise applications and e-commerce websites
- *Recommendation systems* based on data aggregation, proximity search, and other probabilistic methods
- Processing and analysis of *time-series* data for monitoring and log analytics
- Analysis of *spatial information* using geo-positional data types
- *Autocomplete functionality* and *contextual suggesters* for better search experience
- Basic operations on language, such as *tokenization* used in machine learning applications
- *Processing data from IoT devices* (for example, creating real-time maps for the positions of the fleet units)
- *Anomaly detection*

2.1 Elasticsearch Cluster Design Best Practices

While Elasticsearch ships with sensible defaults, it may require a lot of cluster design decisions and configurations before running it in a production cluster. Here we've compiled a list of the most important things you might need to consider when running Elasticsearch in production.

2.1.1 Cluster Design Considerations

Elasticsearch is a distributed system that needs to be Highly Available (HA) to prevent a single point of failure. A common requirement for the design of an HA Elasticsearch cluster is having at least 3 master-eligible nodes for master election quorum, and a set of dedicated data, client, and ingest nodes (see more details in Section 6).

As a general rule of thumb, it's recommended to run I/O-intensive operations on nodes with fast storage (SSD or NVMe) while maintaining cluster state and coordination on the dedicated **master**, **ingest**, and **client** nodes interfacing with the Elasticsearch clients. We'll discuss HA in more detail in Section 5.

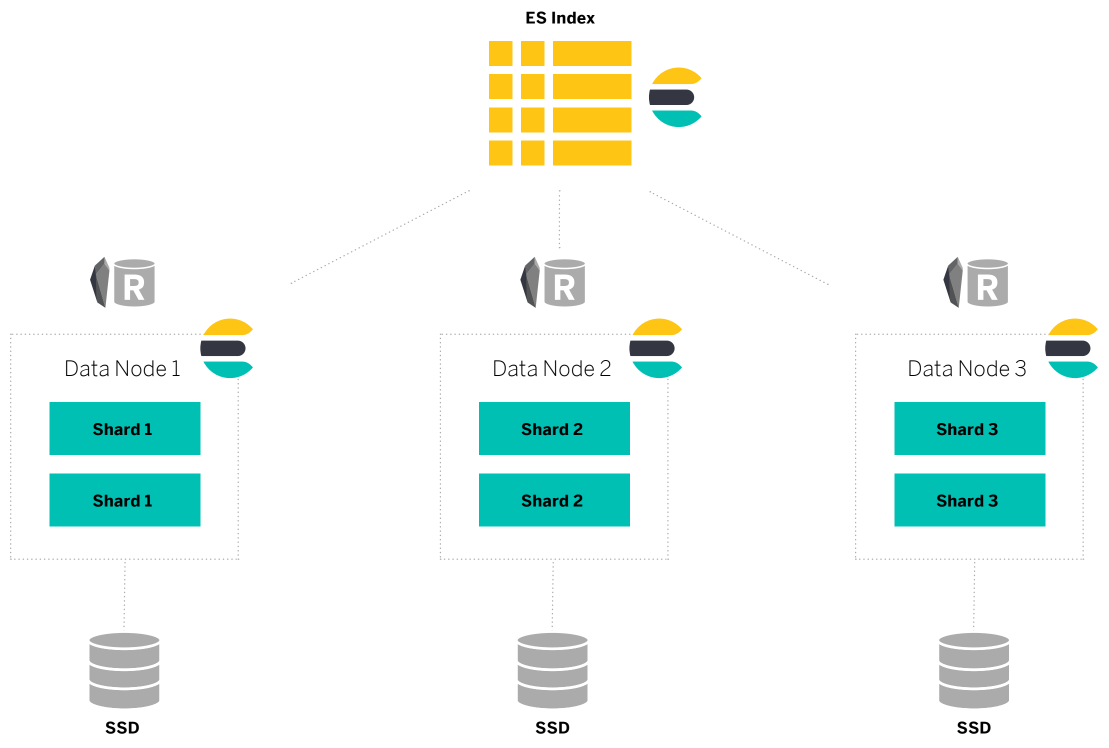


Image: Elasticsearch Index and Shards

2.1.2 Index and Shard Design

Elasticsearch data is stored in indexes—collections of documents similar to 'databases' in the relational database model. Each index can be partitioned into 'shards,' which may be thought of as index slices behaving like standalone indexes.

The motivation behind shards is to enable a distributed storage of data when an index is too big to run on a single node. With shards, one can split an index into multiple parts and distribute them evenly across the cluster.

A crucial cluster design decision is how many shards per index to have. The answer is not straightforward because several trade-offs between shard number, size, and performance of your cluster exist.

For example, the larger the shard size, the longer it takes to move them around when rebalancing the cluster. However, having too many small shards leads to more overhead because more queries to different shards are needed to collect information. The right choice usually lies in between these extremes. The best way to determine an optimal shard size is to run benchmarks using realistic data and queries one expects in production.

2.1.3 How Many Shard Replicas

By default, Elasticsearch creates one replica for each primary shard. A cluster architect can arbitrarily increase this number to provide for storage redundancy and HA. Elasticsearch recommends setting the number of replicas according to the following formula:

max(max_failures, ceil(num_nodes / num_primaries) - 1)

Where:

- *max_failures* refers to how many node failures you tolerate at once.
- *Num_primaries* refers to the total number of primary shards.
- and *num_nodes* is the total number of nodes.

For example, if you have 5 nodes and 10 primary shards and can tolerate 2 node failures at most, the calculation yields **max (2, ceil(5/10) - 1)**, which is 2 replicas for each shard.

2.1.4 Disable Swapping

Swapping involves transferring memory pages from the running memory (RAM) into swap files stored on the hard drives. Swapping out running memory is appropriate for many applications, but is not a good default choice for Elasticsearch. Being much slower than RAM, swap memory can compromise Elasticsearch cluster performance in heavy I/O tasks and garbage collection. Therefore, it's recommended to disable swap memory for Elasticsearch.

2.1.5 File System Cache

A file system cache size is one of the main performance factors for Elasticsearch. File system cache enables faster access to data for frequent reading operations and allows chaining a large amount of I/O operations. For expert users, it's possible to preload the content of "hot" index files into the file system cache upon starting Elasticsearch thereby improving performance by a great margin. It's important to make sure that at least half the memory of the machine running Elasticsearch is dedicated to the file system cache.

2.1.6 Other File and Memory Settings

Some other useful file and memory tweaks for Elasticsearch are related to file descriptors, Java heap size, and number of threads.

File descriptors. It's a good practice to increase the limit on the number of open file descriptors for the user running Elasticsearch to 65,536 or higher.

Threads. Elasticsearch instances should be able to create at least 4096 threads for optimal performance on heavy tasks.

Java heap size. As Elasticsearch runs in Java, the Java heap size allows controlling how much RAM Java Virtual Machine (JVM) has access to. Although the larger heap size the better, usually it's not recommended to set it above 50% of the RAM because Elasticsearch may need memory for other tasks and a host's OS can become slow.

2.1.7 Optimizing Elasticsearch for Read and Write Performance

When going into production, it's critical to optimize your Elasticsearch cluster for read and/or write performance. In most cases, both read and write optimization can co-exist; however, there might be a few trade-offs when optimizing for one or the other. If you want to tune for read performance it's recommended to:

- Use Scroll API, which allows paginating the response data. Returning all matching documents in a single response can be computationally costly.
- Avoid indexing large documents like books or web pages in a single document. Loading big documents into memory and sending them over the network is slower and more computationally expensive.

- Create alerts for critical performance issues. For example, you can get alerts for query latency, the metric that directly impacts the user experience.
- Enable slow query logs to identify non-optimized slow queries.

If you want to tune for indexing speed, the following recommendations apply:

- Set a bigger refresh interval. The refresh interval determines how fast the data saved to an index is available for search. Setting a short refresh interval makes writing new data to Elasticsearch slower.
- Use bulk indexing requests which enable indexing a group of documents in a single call.
- Use faster storage hardware such as SSD or NVMe.

3. RUNNING ELASTICSEARCH IN THE CLOUD: MAJOR CHALLENGES

Major cloud providers like AWS and Google offer hosted Elasticsearch cloud solutions. Although most of them dramatically simplify the deployment of Elasticsearch in the cloud, it would be wrong to think that running Elasticsearch in the cloud is trivially simple. The reality is that many hosted Elasticsearch offerings provide a bare-minimum functionality not sufficient for running production-grade clusters.

There are several challenges associated with hosted Elasticsearch solutions. Here are some of the most important:

- Hosted Elasticsearch offerings may lack mainline ELK functionality like RBAC and security. This will make it harder to align cloud-hosted Elasticsearch management with your organization's security policies. To address this challenge, your company would have to add its own security solution on top of a hosted Elasticsearch service, which is the same burden if you were running Elasticsearch on-premises.
- Some popular hosted Elasticsearch solutions (for example, Amazon Elasticsearch Service) don't support some basic Elasticsearch functionality, such as as shard rebalancing, which is critical for large production clusters. Without shard rebalancing, ELK cluster administrators would need to do a lot of manual work if a node fails—for example, manually moving indexes to a new node and/or re-indexing data.
- Running hosted Elasticsearch may not be an option if you need full control over the Elasticsearch cluster. Cloud Elasticsearch providers usually hide many implementation details and settings that might be important for Elasticsearch users. The functionality they offer may be enough for small setups, but larger Elasticsearch clusters typically require more control over the settings described in the previous section.

Problems might also arise when running Elasticsearch clusters on cloud storage infrastructure. Let's illustrate some shortcomings using the case of Amazon EBS volumes, a go-to storage solution for Elasticsearch deployments on AWS. A frequently cited issue with the EBS volumes is that they might get stuck in attaching or detaching state. In this case, somebody will have to manually detach volumes and reattach them to the new node when an Elasticsearch node is moved between hosts. Obviously, this defeats the purpose of automation. This is also not compatible with microservices architectures. When you run Elasticsearch in Kubernetes, you don't want our Elasticsearch pods to interact with EBS volumes directly. To address this issue, we'll need to use Kubernetes and a container-native storage solution. We discuss what's needed in the next section.

3.1 How To Make Elasticsearch Work in the Cloud

All the challenges mentioned above are also relevant if you choose to run your own Elasticsearch cluster in the cloud. Running a self-hosted Elasticsearch in the cloud requires careful consideration of storage options, security, disaster recovery, high availability, and much more.

Of top importance is the need to address the above-mentioned limitations with storage. What we actually need is a container-granular storage solution tightly integrated with Kubernetes and capable of addressing failures of the underlying physical infrastructure.

One way to do so is to have a unified storage layer that aggregates all storage resources available in your cluster, be they EBS drives of different types or even instance storage. Then, storage from different nodes can be used to provide on-demand virtual volumes without Elasticsearch being dependent on the specific physical volume of the cluster.

Along with solving the data availability and provisioning issues, such an approach can also help avoid some of the bottlenecks of virtual machines running in the cloud. Linux instances running in AWS have a limit of 40 block volumes per host. However, with microservices, we may have hundreds of containers running on a host, each requiring a dedicated volume. The block volume constraint can be avoided with the container-granular storage implemented as an abstraction of the underlying storage pool. Using this strategy, a large number of 'software-defined' volumes can be provided to containers circumventing the above-mentioned constraint.

Another problem that a container-granular storage solution can solve is storage overprovisioning that leads to higher costs when running large production-grade clusters. Because adding storage or an Elasticsearch node that is out of capacity requires taking the node offline for maintenance, many architects will over-provision storage capacity. While this minimizes the operational overhead of maintenance, it increases the cost significantly because you must pay for provisioned storage, even if it is unused. Thin provisioning of volumes from a pool of cloud storage and automatically resizing them on demand with no downtime with a software-defined storage solution solves both problems and is a preferred approach for container-as-a-service platforms and database-as-a-service offerings where cost reductions to improve profit margins are important.

When running stateful applications in the cloud, we are also interested in saving on compute by running the fewest number of database replicas possible in order to achieve a given level of reliability. While Elasticsearch can recover replicas in case of failover, large shards take a long time to recover and reduce cluster performance during the rebuild operation. This can be solved using smaller shards, but this requires more compute nodes to run the extra pods. A middle ground exists in which a copy of each replica is persisted on another host. In this case, we can recover quickly from failure and run fewer compute nodes while, at the same time, increasing write performance. Costs can be also reduced by storage tiering, i.e., distributing workloads across different storage classes depending on the workload priority. We will look at storage tiering in Section 5.

In what follows, we introduce Portworx Enterprise, a software-defined storage solution that can address all-important challenges with running Elasticsearch in the cloud or on-premises such as enabling HA storage, automatic failover across fault domains, storage redundancy, replication, migration, and security. We'll see how one can dramatically improve Elasticsearch performance and resilience in Kubernetes by adding Portworx container storage features.

4. ENSURING HA OF ELASTICSEARCH CLUSTERS

Creating HA Elasticsearch clusters in a microservices environment like Kubernetes is a challenging task because several layers of HA need to be considered. In this section, we'll discuss how to configure Elasticsearch for HA by using Elasticsearch native features, as well as Portworx storage-aware HA, and why both are important.

4.1 HA with Elasticsearch Alone

Previously, we mentioned that Elasticsearch has a distributed architecture that enables an index to be split into shards distributed across different nodes in a cluster. Elasticsearch can also create replicas of these shards evenly distributed across a cluster, which enables a high level of redundancy and HA if some of the nodes fail or are removed from the cluster via a network partition.

When a new node is added or removed from a cluster, the Elasticsearch master dynamically rebalances shards to match a new cluster state. Its goal is to achieve an optimal distribution of shards in the cluster.

This HA design is a great start, but additional things should be considered when running Elasticsearch in Kubernetes.

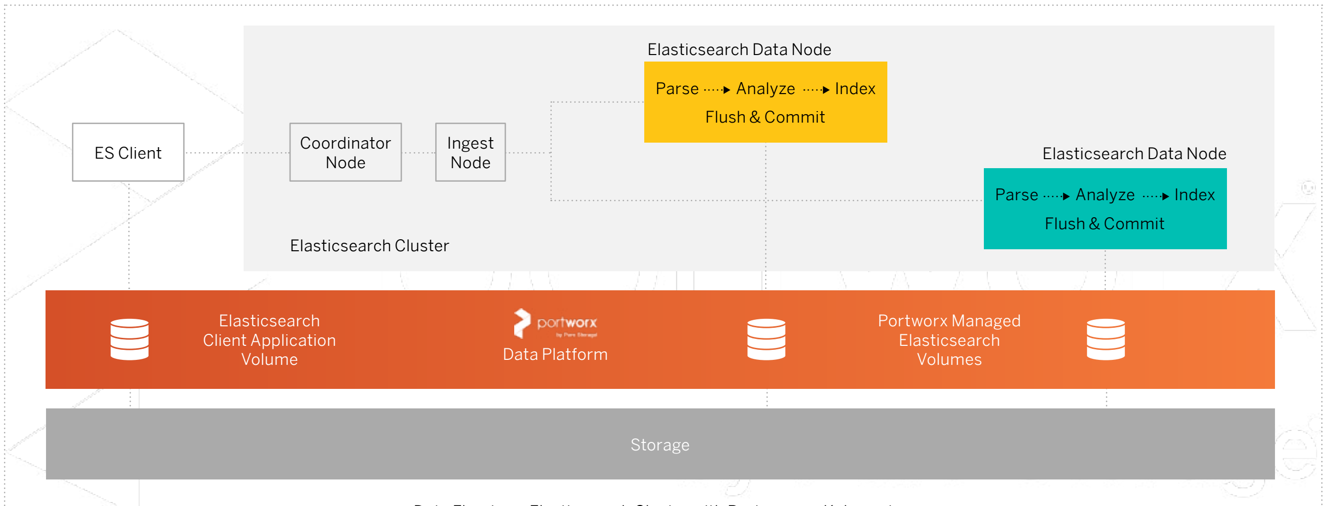
4.2 Elasticsearch HA with Portworx

In Kubernetes, pods and containers are frequently moved across the cluster when an application load changes and/or nodes hosting them die. Volumes provisioned for these applications need to be re-attached to new pods when this happens automatically. While Kubernetes provides many useful storage primitives and abstractions like Persistent Volumes and StorageClasses, an additional storage orchestration layer may be required to secure Elasticsearch HA.

To illustrate why shard rebalancing is not enough for agile Elasticsearch HA, let's assume that we are running Elasticsearch on an AWS-hosted Kubernetes cluster with three data pods. Each data pod has an AWS EBS volume attached as a Persistent Volume. What happens if one of these data pods dies and is rescheduled to some other node in the Kubernetes cluster? If this new node does not have a replica of the Persistent Volume with the Elasticsearch data, the entire Elasticsearch cluster will perform with diminished IOPS and throughput until the data is moved to the new node. While Kubernetes can theoretically detach the EBS volume from the initial host and reattach it to the new host, the volume re-attachment to a new node may fail. This can happen due to many reasons such as a failed AWS API call, failure to unmount and detach from the old node, etc. In the meantime, your Elasticsearch performance suffers, and you need to manually intervene.

By default, Kubernetes does not solve this problem. However, in real-world production deployments, we need Elasticsearch data to always be available, no matter what node Elasticsearch pods end up on. Portworx is designed as a container-granular storage layer that addresses the issue mentioned above.

Portworx first aggregates all storage resources available in the cluster and creates a unified storage layer for them. This storage layer has a simple API that can be leveraged to dynamically provision persistent volumes for applications running in the cluster.



Data Flow in an Elasticsearch Cluster with Portworx on Kubernetes

Then Portworx creates replicas according to user-defined replication factors for these volumes distributed across the cluster. Dynamic distribution of replicas is achieved with the built-in topology awareness that allows Portworx to auto-detect availability zones, regions, or racks and provision replicas across them. Under the hood, Portworx uses synchronous replication across all replicas, which secures data consistency among them. Also, volume replicas can be accessible from any node where Portworx is installed.

For example, in the case we described above, the Elasticsearch data pod rescheduled to a new node in the cluster will have immediate access to the replica of the data it had before replacement. You don't have to wait until the data is migrated or a new volume with a data replica is provisioned.

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.

So, whereas Elasticsearch ensures HA at the application level, Portworx enables storage- or device-level HA. This adds another layer of HA to your cluster.

5. STORAGE CLASS OF SERVICE (COS) REQUIREMENTS FOR ELASTICSEARCH CLUSTERS

There are two motivations for using different storage types and I/O-priority classes for different entities in your Elasticsearch cluster: performance and cost reduction. In what follows, we describe basic storage CoS requirements for various node types and index types in Elasticsearch.

5.1 Elasticsearch Nodes

Normally, an Elasticsearch cluster consists of the following node types, each with their specific storage CoS requirements:

- **Master-eligible nodes (at least 3).** Master-eligible nodes normally need to perform cluster-wide operations such as creating and deleting indexes, tracking nodes, and doing shard allocation/rebalancing. These nodes are not supposed to perform heavy read/write operations, so they typically require storage with medium IOPS and throughput.

- **Data nodes.** These nodes perform I/O-intensive indexing and search operations. Thus, data nodes require high-priority storage CoS in terms of IOPS and throughput and also should run on SSD or NVMe storage infrastructure.
- **Coordinating nodes.** These nodes coordinate requests to data nodes. A typical scenario involving coordinating nodes is when a data request should return data from several data nodes. In this case, coordinating nodes send requests to these data nodes, receive and aggregate data, and return it to users. Working as client proxies, coordinating nodes do not need to have high storage throughput.
- **Ingest nodes.** These nodes are used to pre-process documents before indexing. Ingest nodes intercept index requests and apply certain transformations to them. If your cluster write/read ratio is very high, ingest nodes may also need to run on performant storage infrastructure.

5.2 Storage Requirements for Elasticsearch Indexes

One of the best features of Elasticsearch is index lifecycle management. It allows creating policies that govern index state at various stages of its lifecycle. This state reflects the relative value of write/update/read operations at various stages of the index lifecycle.

Index lifecycle of time-series indexes in Elasticsearch has four stages:

- **Hot**—active updating and queries
- **Warm**—the index is queried but no longer updated
- **Cold**—no longer updated and is seldom queried
- **Delete**—at this stage, the index is no longer needed

To optimize hardware usage, it's reasonable to provide different storage classes for indexes at different stages of their lifecycle. For example, as "hot" indexes process the majority of read/write operations, it's preferable to run them on fast storage such as SSD or NVMe. Likewise, "cold" indexes may be relocated to cheaper hardware such as HDD.

5.3 Guaranteeing CoS Requirements

Portworx Enterprise provides an automatic CoS model, which is useful for meeting Elasticsearch storage requirements mentioned above. When creating a unified storage layer, Portworx automatically pools different storage types available in the cluster into three storage CoS: high, medium, and low. The high CoS is optimized for IOPS; medium is optimized for throughput, and low CoS provides cheaper storage for low-priority data. In the cloud, this means that each VM could have 3 different cloud block devices that Portworx tiers across. On-prem, it could mean that each bare metal server is loaded with different storage drive types.

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

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

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

6. STORAGE SECURITY

Running Elasticsearch in a microservices environment like Kubernetes introduces new security risks not covered by the default Elasticsearch security model. To ensure the security of your Elasticsearch deployment, you should address various layers of security: application-level security, Kubernetes security, and storage security.

As far as Elasticsearch security is concerned, the platform supports a number of useful security features such as password-protection of data, encrypting communications, RBAC, IP filtering, and auditing. These features, however, are applicable only to the Elasticsearch cluster level.

The next layer (Kubernetes) has cluster security that controls how the cluster is accessed, what applications and users running in it are allowed to do, how secrets and configuration are exposed to applications, and much more. The Kubernetes security model works at the level of abstractions such as Pods and Persistent Volumes. So, it does not affect access to the actual storage underneath these abstractions.

Thus, we need the third specialized storage security layer for volumes storing Elasticsearch data. This layer can be provided by Portworx volume encryption, authentication, and RBAC that can control access to underlying storage infrastructure storing mission-critical Elasticsearch data.

For volume encryption, Portworx uses dm-crypt, a transparent disk encryption subsystem in the Linux kernel to create, access, and manage encrypted devices. Portworx volumes may be encrypted with cluster-wide secrets that default to all volumes or per-volume secrets—unique secrets for each volume. Data can be encrypted at rest as well as in transit.

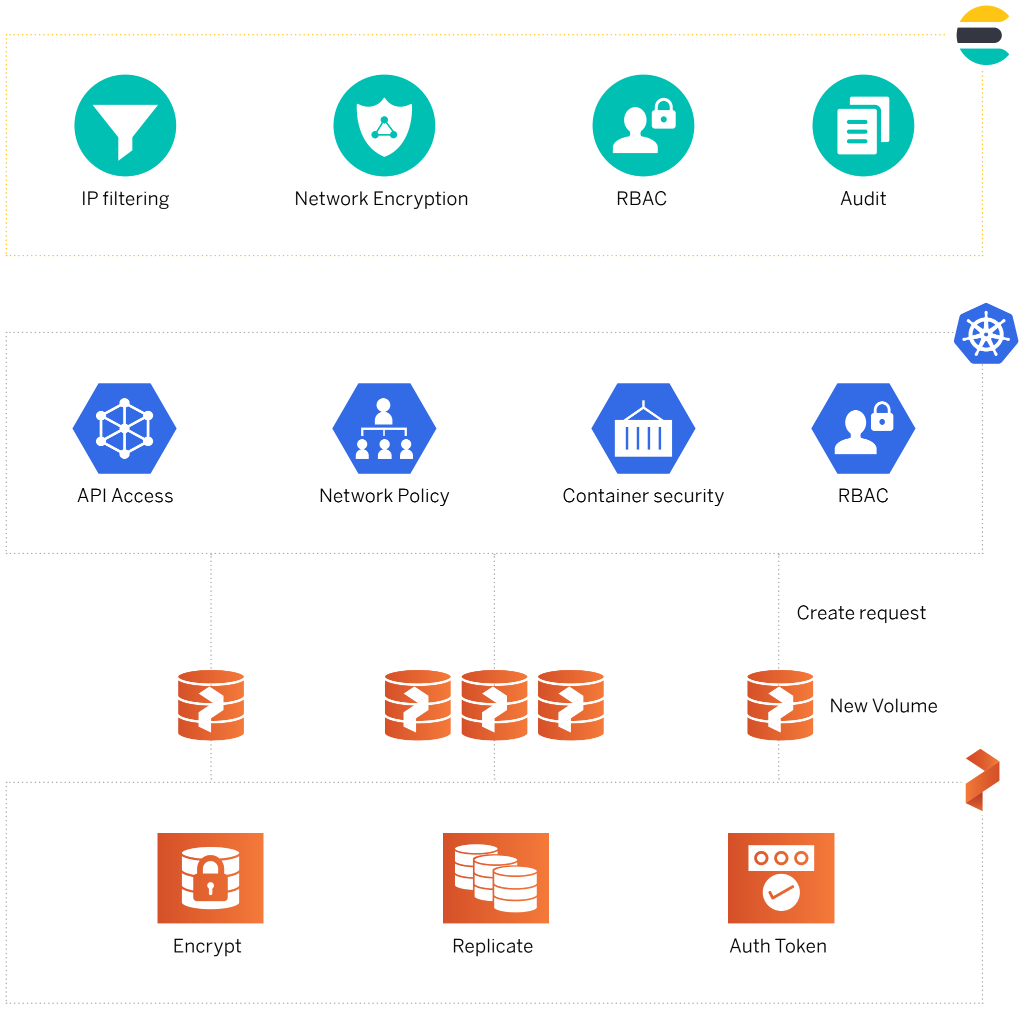


Image: Three Layers of Security

Also, Portworx supports user authentication and authorization for volume management. 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. Portworx supports RBAC for authorization. You can also specify ownership rights to control types of access (read, write) 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 Elasticsearch in Kubernetes. By adding a storage security layer we can have multi-modal security that covers both Elasticsearch application-level security, cluster-level security, and storage-level security, dramatically decreasing the potential vectors of attacks against your deployments.

7. ELASTICSEARCH CROSS-CLUSTER MIGRATION

To make your Elasticsearch cluster even more resilient, you may need another remote cluster. PX-Migrate is a Portworx capability that lets you migrate data, application configuration, and Kubernetes objects (ConfigMaps, Secrets, etc.) across your Kubernetes clusters seamlessly and with almost no effort.

Common use cases for migrating Elasticsearch may include:

- Moving low-priority indexes, such as 'cold' indexes to secondary clusters.
- Testing new versions of Elasticsearch, Kubernetes, or Portworx using the same application and data.
- Moving workloads from dev/test environments to production without the need to manually provision the data.
- Moving workloads from private on-premises clusters to cloud-hosted Elasticsearch clusters like Amazon EKS or Google GKE.

8. DISASTER RECOVERY AND BACKUP

Disaster recovery and backup are two major concerns when managing data stores and stateful applications like Elasticsearch. Fortunately, Elasticsearch has good built-in support for backup and recovery with the Snapshot and Restore module. Also, Elasticsearch allows automating the snapshot backup and restore process with the snapshot lifecycle management system.

However, quite reasonably, the Elasticsearch snapshot restore API works at the application-level. It ensures that snapshots reflect the consistent state of all pending in-memory operations and data, which is impossible to achieve by manually copying Elasticsearch data files.

At the same time, we need to ensure that physical volumes are also backed up and can be easily restored. We've already mentioned how you can ensure HA and data safety with Portworx volume replication. Having volumes configured with the replication factor of three is enough for ensuring the safety of your data.

In addition to volume replication, Portworx supports container-granular backups of data volumes and application configuration to make restoring Elasticsearch as simple as running `kubectl -f apply`. When creating a volume, users can specify a snapshot schedule to periodically back it up.

Also, Portworx supports backup to any cloud or on-prem object store, which allows the user to export/import volume snapshots to a specified backup location. This may be useful in case of cluster failure, as snapshots can be easily imported to any Portworx cluster.

For an additional layer of protection, you can use PX-DR to have an up-to-date second Elasticsearch cluster always up and running in case of a complete data center outage. At the end of the day, Portworx gives you complete data protection for your Elasticsearch environment.

9. CONCLUSION

Elasticsearch has special requirements when running in containers on the Kubernetes platform. In particular, it requires a resilient storage layer tuned for HA and performance. Portworx provides such a layer and guarantees HA and storage agility via tight integration with the Kubernetes scheduler.

As we've demonstrated in this paper, users can leverage Portworx's container-granular storage to provide the following benefits for their Elasticsearch deployments:

- High Availability
- Topology-aware data replication
- Automatic failover
- Different storage Classes of Service (CoS)
- Thin provisioning
- Security
- Cross-cluster data migration

All these features are ultimately designed to significantly reduce costs associated with storage and compute while making the user experience much better due to lower latency, and faster reads and writes. In this way, we can combine the benefits of Elasticsearch, Kubernetes, and Portworx to generate synergies and enhance performance.

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