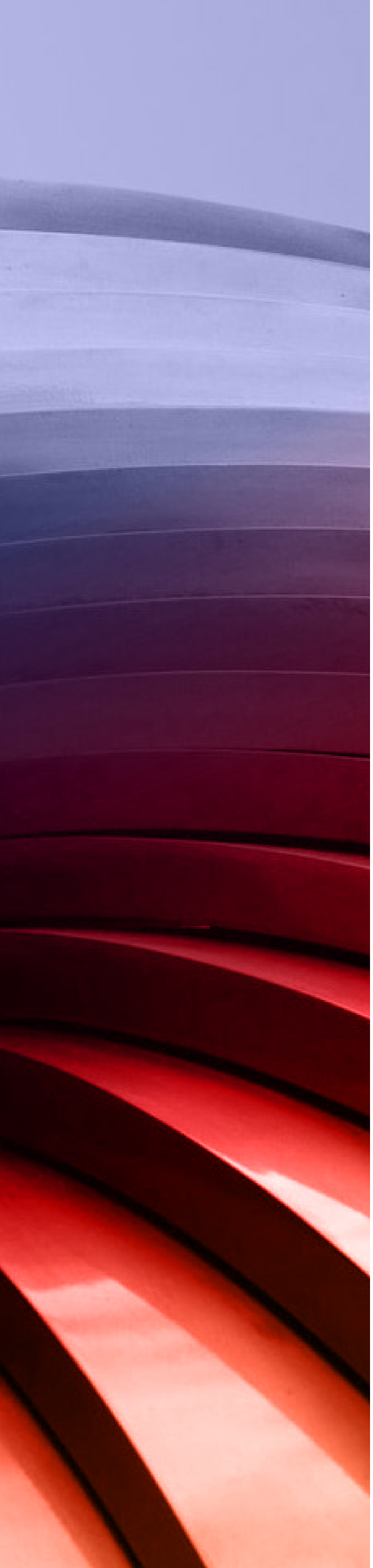


WHITE PAPER

RUNNING RESILIENT DATABASES ON

RED HAT OPENSHIFT

Benefits, Trade Offs, and Solutions for Deployment

yugabyteDB

Red Hat

Table of Contents

Introduction

Kubernetes, a popular open-source platform for managing containerized workloads and services, has become widely adopted in the Fortune 500. Many companies, such as Walmart, Target, and eBay, are now using the platform to run stateless and stateful applications on-premises or as hybrid cloud deployments in production. Of course, with any new technology, there are growing pains when running workloads on Kubernetes. But most executives and developers agree that the benefits far outweigh the challenges.

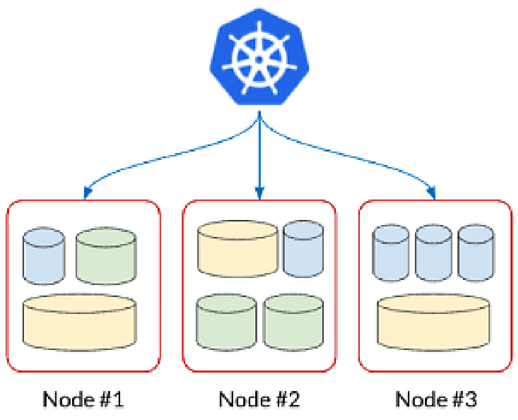
On the flip side, data on the Kubernetes ecosystem is evolving rapidly with the rise of stateful applications. However, stateful applications demand a new database architecture that takes into account the scale, latency, availability, and security needs of applications. But how do you know which database architecture is best equipped to handle these challenges?

In this white paper, we'll explore the benefits and potential trade offs of running a database in Kubernetes. We'll then discuss how to mitigate tradeoffs with distributed SQL, a new class of database that combines the best features of traditional RDBMSs and NoSQL databases for running transactional applications. Finally, we'll provide a real-world solution for ensuring Kubernetes deployments are resilient and continuously available.

Five Benefits to Running a Database in Kubernetes

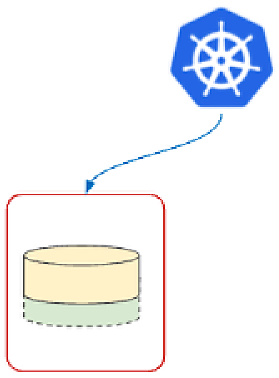
Should you run a database in Kubernetes? There are no definitive answers. However, there are tangible benefits to consider. Let's examine the reasons why it's worth it for your company.

1. **Better Resource Utilization**
   In a modern application, many companies are moving to adopt microservices architectures. As a result, this shift tends to propagate a lot of smaller databases. And companies often have a finite set of nodes on which to place those databases. So, when companies decide to manage these databases, they're left with a sub-optimal allocation of databases onto nodes. However, running Kubernetes allows the underlying system to determine the best places to put the databases while optimizing resource placement on those nodes.

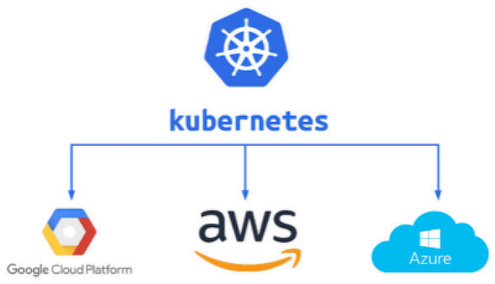


Kubernetes is best utilized when running a large number of databases in a multi-tenant environment. In this deployment scenario, not only do companies save on costs, they need fewer nodes to run the same sort of databases. These databases also have different footprints, CPU resources, memory, and disk requirements.

2. **Elastic Scaling of Pod Resources Dynamically**
The Kubernetes orchestration platform has the ability to resize pod resources dynamically. More specifically, memory, cpu, and disk can be modified to scale a database to meet demanding workloads. Kubernetes makes it easy to scale up automatically without incurring any downtime through its horizontal pod autoscaler (HPA) and vertical pod autoscaler (VPA) operators. However, for VPA, it's worth noting a database would need to have more than one instance to avoid downtime.



3. **Consistency and Portability Between Clouds, On-Premises, and Edge**
Companies want to be consistent with the way they build, deploy, and manage workloads at different locations. They also want the capability to be able to move workloads from one cloud to another, if needed. However, most organizations have a large amount of legacy code they still run on-premises and they're looking to move these installations up into the cloud.



Kubernetes allows you to deploy your infrastructure as code, in a consistent way, everywhere. This allows you to write a bit of code that describes the resource requirements deployed to the Kubernetes engine and the platform will take care of that. You now have the same sort of control in the cloud that you would have on bare metal servers in your data center, or edge.

4. **Out-of-the-Box Infrastructure Orchestration**
With Kubernetes, if a pod crashes, then it automatically restarts. Typically, pods can be started anywhere, as the platform has the capability to say, "I want to move this workload from this pod to this node onto another node." This allows it to do optimal resource allocation and utilization, but it's really good for stateless workloads.

A microservice can be deployed in a Kubernetes pod with 10 different instances of that pod serving traffic. The platform doesn't care if one goes down and moves to a different node since it has no state. However, for a database, this becomes a bigger challenge when dealing with stateful workloads. This means you need to set up specific policies in Kubernetes to ensure it addresses this challenge.

For example, you may want to set up anti-affinity that allows you to specify in code the rules Kubernetes should follow. These include not wanting two instances of the same database on the same node. This allows your system to suffer a hardware failure without taking down multiple copies of database instances. You don't want to lose multiple copies of the same piece of data.

5. **Automated Day 2 Operations**
   Kubernetes allows periodic backups and database software upgrades. You want these operations automated so they can stay up-to-date. Even better, doing these updates across a database cluster is easy. Therefore, if a security vulnerability is exposed, and you want to patch it across the cluster, Kubernetes makes this very simple.

   However, when using a traditional RDBMS with Kubernetes, you're going to have a couple copies of the data. As a result, if you lose a pod, there's another copy elsewhere. But you're still responsible for migrating the data between those two pods and resyncing it for the failed instance when it comes back online. This is done asynchronously in Kubernetes, which is why having automated Day 2 operations can be complicated for a traditional RDBMS.

   For example, if you're migrating data manually, you would check to see that the cluster isn't under heavy load. You'll need to wait until the load mitigates before moving the data to another node. But if you're migrating data in an automated fashion, then you need to be careful to have those checks built in. Otherwise, if you take down a primary copy of data under heavy load, your replica may think it has the data when it really doesn't. This means there could be two different copies of the data. It also creates the potential for data loss and inconsistencies.

Five Trade Offs to Running a Database in Kubernetes

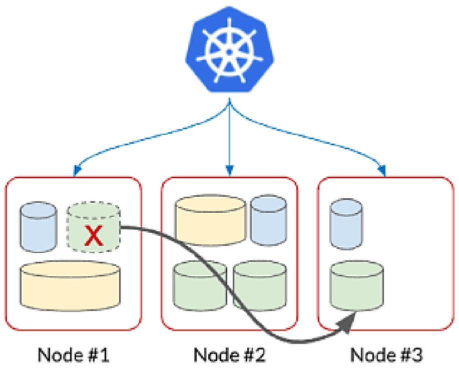
We've covered why you should run a database in Kubernetes. However, when making any new technology decisions, there are some potential trade offs to keep in mind.

1. **Possibility of Pod Crashes**
   Kubernetes is a wonderful orchestration tool. However, a Kubernetes pod may crash because it has process affinity. Therefore, if the process that starts a pod goes down, then the entire pod could go away. So, if you run a particular query, and that query overloads memory or a bad configuration, Kubernetes may take down the pods. This can lead to the entire pod crashing.

   Common reasons pods may crash are memory pressure and the OOM killer. Another factor is transparent rescheduling of pods. You can add a node, but you need to move those pods around to ensure optimal resource contention. But you may experience different issues when using different sorts of storage. Locally attached storage on databases provides fast performance. The only problem, though, is that locally attached storage doesn't exist on the new node.

   As an example, let's see what happens when you try to move a pod from node 1 to node 3, as illustrated in the below diagram.

If there is locally attached storage on node 1, when that pod moves to node 3, that locally attached storage isn't available. It looks like it's there but it's typically blank, which means there may be data loss. You can get around this issue by using a network-attached storage or external persistent storage and then attach the new pod.



Data loss likely if local storage used by pod

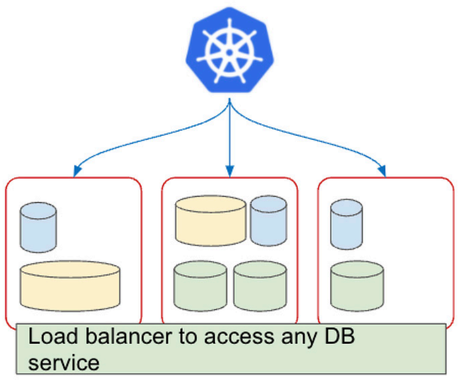
2. Local vs. External Persistent Storage

Locally attached disks are great. They are a single drive or set of drives that offer the fastest performance (e.g., NVME drives). For example, if latency is critical, locally attached storage is great. But as mentioned in the last section, if we move a pod around, the storage doesn't go with it. This means you start with a blank drive.

External persistent storage provides some form of network-attached storage and are easy to consume via Container Storage Interface(CSI) plugins. This is easy to set up with cloud providers, but not so easy on bare metal servers. The advantages of external persistent storage is it's a logical view of drives. As a result, instead of having a fixed drive, there's an arbitrary large view where a drive's storage capacity can be specified.

Another advantage is in moving a pod from one node to another. Pods can simply re-attach to the same storage—which is a virtual drive—and to that instance.

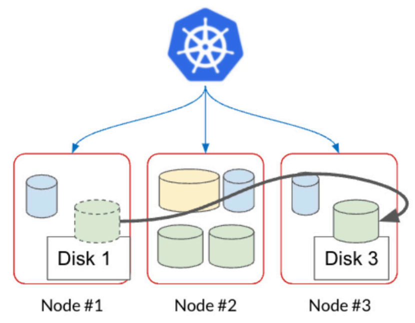
Various software solutions can handle on-premises deployments with ease. It all comes down to speed vs. the ease of restarting a new node.



3. Potential Need for a Load Balancer

There's a possibility for networking restrictions in Kubernetes clusters. This means an application may need to be on the same cluster as the actual database. If not, then a load balancer is required.

A load balancer allows the exposure of services to external customers, be they in the cloud or bare metal servers. This isn't a major issue in public clouds or popular container platforms. This is because they have built-in network load balancers and few limitations outside of how many public IP addresses are acquired.



Pod sees a new, empty disk (Disk 3) after move with local storage

One potential solution is to host all database instances in the same cluster. However, this can be an anti-pattern. It's better to use smaller clusters—and more of them—to minimize the risk of localized failure. Or, to put it another way, if a node or set of nodes crash, what is the cost to an application across the distributed ecosystem you're building?

4. **Possible Networking Complexities**
   Let's say there's a cluster set up for geographical replication, with a cluster in the US-East region, and another in the US-West region. You want to replicate the data between them so there is geographical redundancy if a natural disaster occurs and an entire cluster crashes. You then have another full copy of the data you can automatically connect to and just keep running.

   Typically, these regional clusters reside in different data centers. And databases like to replicate over TCP at a very low level to provide efficient replication. However, some cloud providers, such as Google Cloud, can set up VPC pairing between the East and West regions to address these concerns. But not all cloud providers offer this, and if you're running on bare metal servers, you have to work out other solutions.

   One such solution is DNS chaining. This can be complex to do based on your environment, but it involves a set of technologies rather than a single technology. Another solution is to use a service mesh, such as Istio. A service mesh works well but could cause performance degradation since it runs over HTTP rather than TCP. The open source tool Submariner, built to connect overlay networks of different Kubernetes clusters using encrypted VPN tunnels, can solve this challenge.

5. **Operational "Gotchas"**
   If you're going to run a database on Kubernetes in production, there are several "gotchas" to watch out for:

   - Define anti-affinity and what constitutes a pod disruption
   - Understand the concept of side cars
   - Build in observability with a tool such as Prometheus
   - Create troubleshooting cookbooks (e.g., what happens in crash backups due to memory pressures over subscriptions?)
   - Define private image registries and pool secrets (e.g., who can upload, who can download, what permissions are in place)

   Many companies successfully run Kubernetes in production. But most organizations get started by running the platform in lower production environments such as dev, test, and staging. There's also a growing selection of companies who run distributed SQL databases on Kubernetes in production. We'll explore the ins and outs of this type of database in the next section.

Mitigating Trade Offs with Distributed SQL Databases

Up to now, we've discussed the many benefits and potential trade offs to running a database in Kubernetes. What's needed is a distributed SQL database that can mitigate trade offs while ensuring data is always available. In this section, we'll define what a distributed SQL database is, and show how it can mitigate trade offs when running in Kubernetes.

What is a Distributed SQL Database?

A distributed SQL database is a single logical relational database deployed on a cluster of servers. The database automatically replicates and distributes data across multiple servers. These databases are strongly consistent and support consistency across availability and geographic zones in the cloud.

At a minimum, a distributed SQL database has the following characteristics:

- A SQL API for accessing and manipulating data and objects
- Automatic distribution of data across nodes in a cluster
- Automatic replication of data in a strongly consistent manner
- Support for distributed query execution so clients do not need to know about the underlying distribution of data
- Support for distributed ACID transactions

Why Distributed SQL Databases?

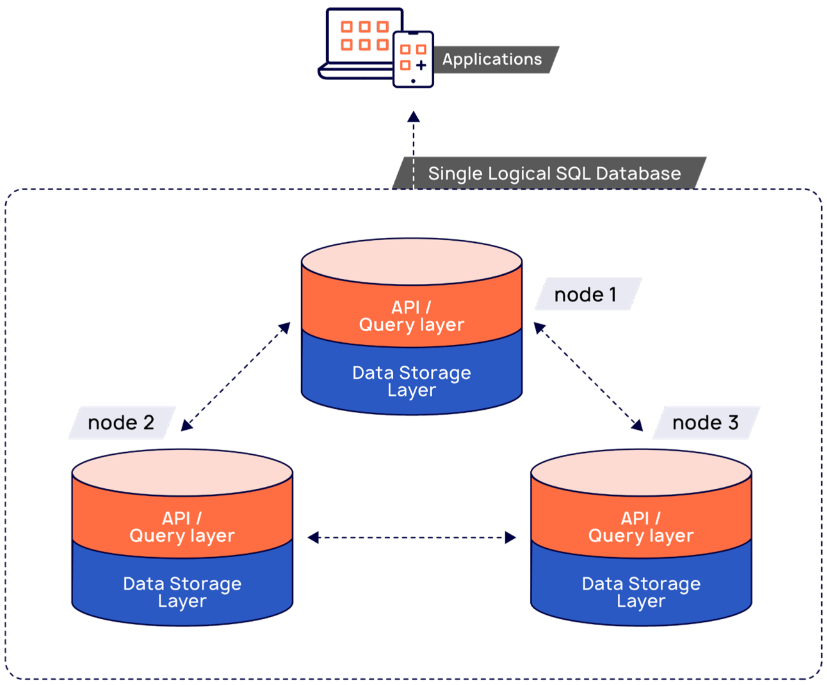
Business innovation is putting pressure on traditional systems of record. This is forcing companies to deliver high-value applications and services more quickly while lowering IT costs and reducing risk through compliance.

But these applications—in the form of microservices, born-in-the-cloud applications, and edge and IoT workloads—require a new class of database that is:

- **Resilient to failures and continuously available:** Critical services remain available during node, zone, region, and data center failures as well as system maintenance with fast failover
- **Horizontally scalable:** Operations teams can effortlessly scale out even under heavy load without downtime by simply adding nodes to a cluster, and scale back in when the load reduces
- **Geographically distributed:** Operators can make use of synchronous and asynchronous data replication and geo-partitioning to deploy databases in geo-distributed configurations
- **SQL and RDBMS feature compatible:** Developers no longer need to choose between the horizontal scalability of cloud native systems and the ACID guarantees and strong consistency of traditional RDBMSs
- **Hybrid and multi-cloud ready:** Organizations can deploy and run data infrastructure anywhere—and avoid being locked-in to any specific cloud provider.

Distributed SQL Database Architecture

A distributed SQL database provides the best of a traditional RDBMS with cloud native database capabilities. It has a two-layer architecture as part of a single logical SQL database:



SQL Query Layer

A distributed SQL database has a SQL API for applications to model relational data and also perform queries involving those relations. Queries are automatically distributed across multiple nodes of the database cluster.

Distributed Data Storage Layer

Data, including indexes, in a distributed SQL database are automatically distributed—or sharded—across multiple nodes of the cluster so that no single node becomes a bottleneck to high performance and availability.

Supporting a powerful SQL API layer requires the underlying storage layer to be built on strongly consistent replication across all nodes of the cluster. This means writes to the database are synchronously committed at multiple nodes in order to guarantee availability during failures.

And finally, the database storage layer supports distributed ACID transactions where transaction coordination is required across multiple rows located on multiple nodes.

How a Distributed SQL Database Mitigates Kubernetes Tradeoffs

A distributed SQL database functions as a single logical database deployed as a cluster of nodes. This means the database cluster takes care of sharding, replication, load balancing, and data distribution. Therefore, a distributed SQL database keeps your database up and running even if there's a pod, node, or underlying infrastructure failure. The database cluster is able to detect the failure, handle it, and recover without any loss of data or access by the application.

A distributed SQL database also provides a scalable and resilient data store for connecting applications. It takes care of migrating data between pods after a pod moves to a new node. It does this behind the scenes without any form of operator intervention.

YugabyteDB: Best-in-Class Distributed SQL for Resilient Kubernetes Workloads

YugabyteDB is a cloud native distributed SQL database for transactional applications. The database is 100% open source and built to solve availability and resiliency challenges when running application workloads on Kubernetes.

This database uses replicas for high availability and supports synchronization through the use of the Raft protocol. Additional features include partitions (called tablets) for scalability, and in case of a cross-tablet transaction, the two-phase commit protocol is also implemented.

YugabyteDB automatically partitions SQL tables into tablets without user intervention. It also automatically distributes tablet replicas to the configured failure domains ensuring, as much as possible, no data loss. This behavior can be influenced by the user configuring the failure domains, replication factor, and database affinity to failure domains.

Deployment Flexibility

YugabyteDB runs in public, private, and hybrid cloud environments, on VMs, containers or bare metal. Organizations can deploy the database in any Kubernetes environment. It is also available as a multi-cloud, fully managed database-as-a-service (DBaaS) for a frictionless experience. YugabyteDB offers the widest range of replication and geo-distribution options among distributed SQL databases.

High Performance

YugabyteDB can handle high throughput, low latency transactions on Kubernetes. It is proven in production to scale beyond 1 million transactions per second and thousands of concurrent connections.

Operational Simplicity

Organizations can use the self-managed or fully managed DBaaS offerings of YugabyteDB to simplify operations at the edge and in the cloud. The database also integrates with other data sources or sinks, allowing data engineers to build pipelines for machine learning, analytics, long term storage, and disaster recovery.

PostgreSQL Compatibility

YugabyteDB is not just wire compatible with PostgreSQL, it is code compatible. The database also offers a comprehensive set of advanced RDBMS features including triggers, functions, stored procedures, and strong secondary indexes. This allows developers to be immediately productive with the familiar interface and the rich ecosystem of PostgreSQL compatible frameworks, applications, drivers, and tools.

Security

YugabyteDB is built from the ground up with data security in mind, enabling organizations to maintain a robust security posture even with a more distributed footprint. YugabyteDB offers features such as data encryption at rest and in flight, multi-tenancy support at the database layer with per-tenant encryption, and regional locality of data to ensure compliance as well as manage geographic access controls.

YugabyteDB and Red Hat OpenShift: Cloud Native Resiliency at Enterprise Scale

YugabyteDB is available on Red Hat® OpenShift®, the leading enterprise Kubernetes platform for deploying and managing cloud native applications. It's a software product built on the Kubernetes container management project, but adds productivity and security features that are important to large-scale companies. For all that Kubernetes can do, users still need to integrate other components like networking, ingress and load balancing, storage, monitoring, logging, and more. Red Hat OpenShift offers these components with Kubernetes at their core because—by itself—Kubernetes is not enough.

Red Hat OpenShift provides a consistent application platform for the management of existing, modernized, and cloud native applications that run on any cloud and on-premises. It also offers a common abstraction layer across any infrastructure to give both developers and operations teams commonality in how applications are packaged, deployed, and managed. This means teams can deploy YugabyteDB on Red Hat OpenShift with confidence because the YugabyteDB container image and the Operator have been Red Hat certified. As a result, both are well-integrated to run on the platform and enable efficient day 1-2 operations.

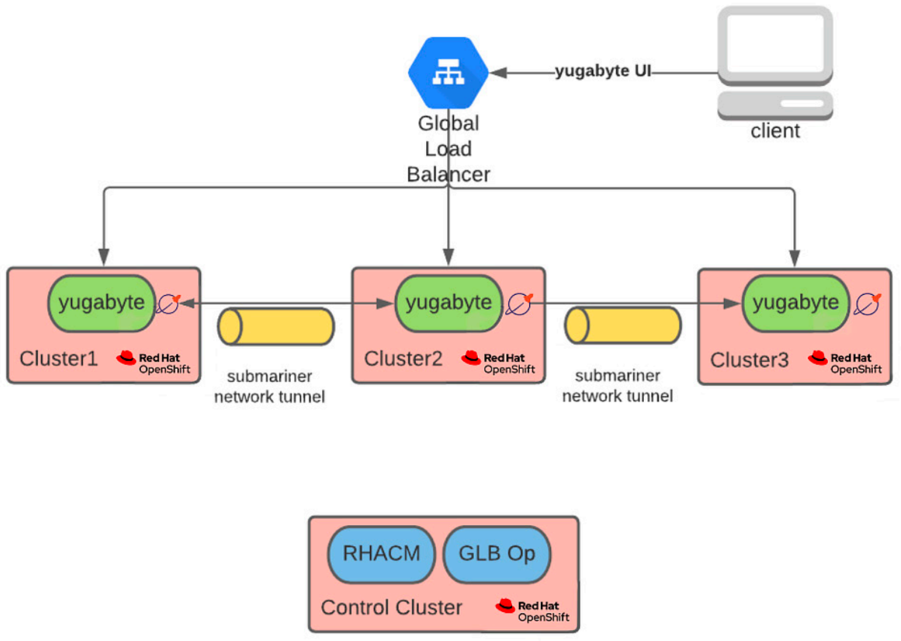
“Run Anywhere” Distributed Stateful Workloads

One major benefit to running YugabyteDB on Red Hat OpenShift is the guarantee of geo-distributed stateful workloads. The below diagram depicts what such an architecture looks like in practice.

Starting from the top, we have a global load balancer directing connections to the YugabyteDB UI. Then, there are three Red Hat OpenShift clusters with YugabyteDB instances deployed to each cluster. These instances can communicate with each other via a network tunnel implemented with Submariner.

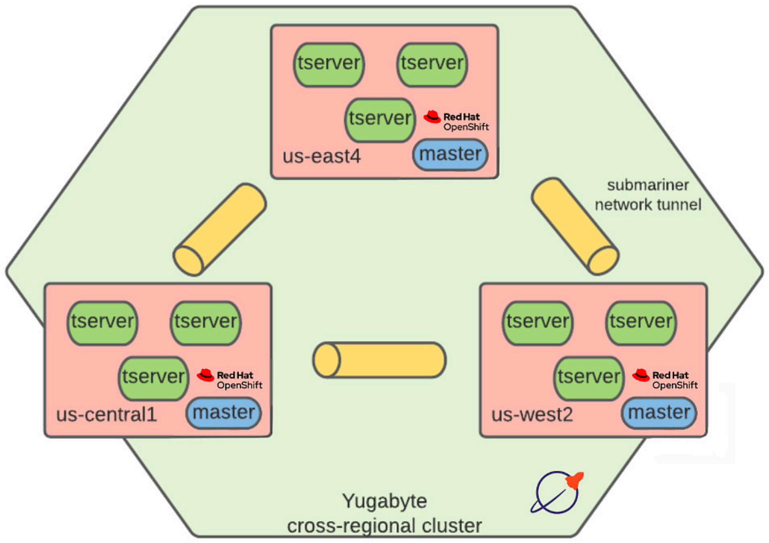
Finally, at the bottom of the diagram, Red Hat Advanced Cluster Manager (RHACM) has been installed within a control cluster. This is used to create the other clusters along with the global load balancer operator, which facilitates configuring the global load balancer at the top of the diagram.

Each cluster is in a different region of a public cloud provider.



Zooming in on the YugabyteDB deployment, we have three tablet servers and a master (metadata server) in each cluster. Together, they form a logical YugabyteDB instance.

Load test results reveal this kind of deployment is usable in production, as is shown by the integration work done between Yugabyte and Red Hat in this Geographically Distributed Stateful documentation.

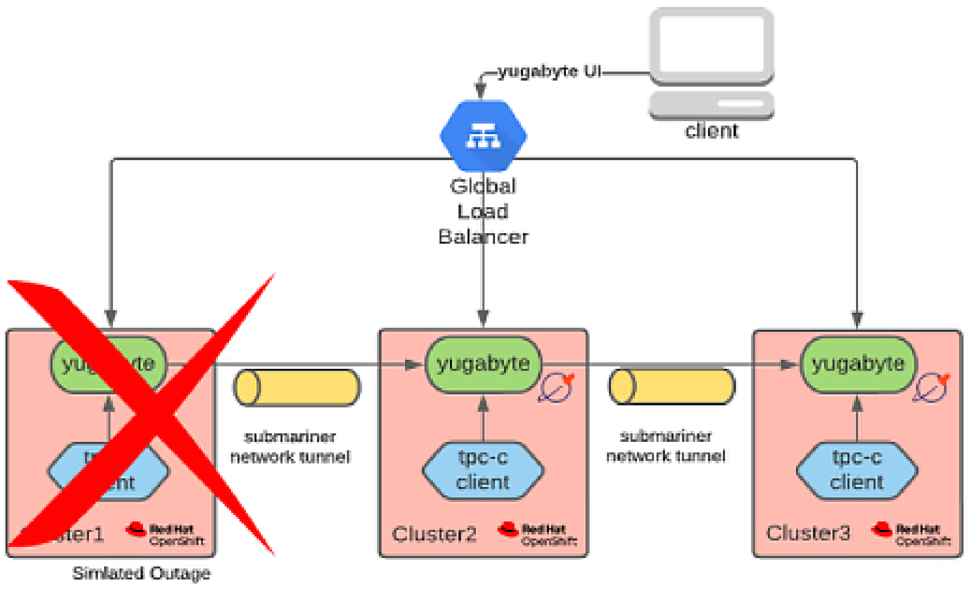


Zero Data Loss and Continuous Availability

Another benefit to running YugabyteDB on Red Hat OpenShift is zero data loss and continuous availability during a major system outage or natural disaster.

For example, in the diagram on the following page, the network of one region is isolated by preventing any inbound or outbound traffic while running a TPC-C test.

When simulating this disaster, there were a few errors in the surviving TPC-C clients; essentially, some in-flight transactions were rejected or failed to complete. But YugabyteDB moved all of the tablet leaders to the healthy instances.



The system managed the disaster without the need for any human intervention. When connectivity to the isolated region was restored, there were no issues within the ongoing TPC-C clients. YugabyteDB rebalanced the database by moving the tablet leaders back to the newly-available tablet servers. Again, no human intervention was needed.

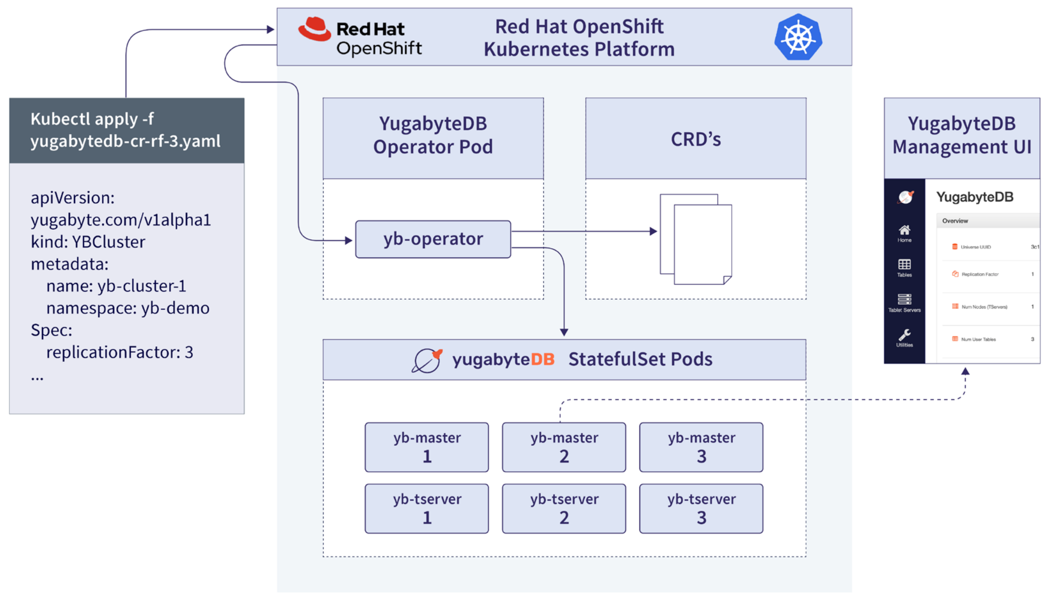
During this simulation, the system experienced zero data loss (RPO 0and very little unavailability (RTO measured in seconds).

Getting Started with YugabyteDB on Red Hat OpenShift

YugabyteDB brings support for both scale-out RDBMS and internet-scale OLTP workloads onto Red Hat OpenShift. This enables customers to transition these workloads to enter-prise-grade Kubernetes. The YugabyteDB Operator allows developers to run YugabyteDB clusters on Red Hat OpenShift using the same cloud native practices they have come to use with stateless applications, such as scaling and managing the lifecycle of workloads using CI/CD pipelines. You can find the YugabyteDB Kubernetes Operator in the OperatorHub of your Red Hat OpenShift environment.

The below figure shows a high-level overview of the components involved for deploying a YugabyteDB cluster on Red Hat OpenShift using the YugabyteDB Operator.

A YugabyteDB cluster deployment consists of two distributed services: yb-tserver and yb-master. The yb-tserver service is responsible for storing the application data and serving the client requests. yb-master is a lightweight service responsible for maintaining the system metadata (including table-to-shard-to-node mapping) and performing background operations such as automatic data rebalancing. You can learn more about the components in a YugabyteDB cluster here.



```
Kubectl apply -f
yugabytedb-cr-rf-3.yaml

apiVersion:
yugabyte.com/v1alpha1
kind: YBCluster
metadata:
   name: yb-cluster-1
   namespace: yb-demo
Spec:
   replicationFactor: 3
...
```

Red Hat OpenShift provides a managed Kubernetes cluster. Operator Lifecycle Manager (OLM) is responsible for managing the lifecycle of the YugabyteDB Operator pods and CRDs that are registered with the Kube API. On creating an instance of the custom resource ybclusters.yugabyte.com, the YugabyteDB Operator creates the necessary statefulset pods with provided attributes like replication factor and desired pod count. It bootstraps the additional services including a LoadBalancer service for exposing the YugabyteDB admin console.

Yugabyte Platform, as shown in the above figure, can also be deployed in a Red Hat OpenShift environment. It provides the simplicity and support to deliver a self-managed DBaaS at scale. Yugabyte Platform is Yugabyte's recommended mechanism for provisioning and managing the lifecycle of YugabyteDB clusters.

Here are step-by-step instructions for getting started with YugabyteDB on Red Hat OpenShift:

1. Deploy Yugabyte Platform using the YugabyteDB operator
2. Configure Red Hat OpenShift within Yugabyte Platform
3. Create and manage deployments using Yugabyte Platform

Conclusion

Kubernetes has been a paradigm shift in the way enterprises build and deploy applications to cater to the needs of an increasingly cloud native world. There is no one-size-fits-all database reference architecture that works for all applications in this environment. Depending on the requirements of the application and tradeoffs involved, enterprises will choose different topologies to meet their needs, and change the topologies when needs change. Distributed SQL databases offer a powerful and versatile data layer for running applications in both the cloud and Kubernetes environments.

About Yugabyte, Inc.

Yugabyte is the company behind YugabyteDB, the open source, high-performance distributed SQL database for building global, cloud-native applications. YugabyteDB serves business-critical applications with SQL query flexibility, high performance and cloud-native agility, thus allowing enterprises to focus on business growth instead of complex data infrastructure management. It is trusted by global companies in cybersecurity, financial markets, IoT, retail, e-commerce, and other verticals. Founded in 2016 by former Facebook and Oracle engineers, Yugabyte is backed by Lightspeed Venture Partners, 8VC, Dell Technologies Capital, Sapphire Ventures, and others.

About Red Hat, Inc.

Red Hat is the world's leading provider of enterprise open source software solutions, using a community-powered approach to deliver reliable and high-performing Linux, hybrid cloud, container, and Kubernetes technologies. Red Hat helps customers integrate new and existing IT applications, develop cloud-native applications, standardize on our industry-leading operating system, and automate, secure, and manage complex environments. Award-winning support, training, and consulting services make Red Hat a trusted adviser to the Fortune 500. As a strategic partner to cloud providers, system integrators, application vendors, customers, and open source communities, Red Hat can help organizations prepare for the digital future.

Get in Touch

www.yugabyte.com | contact@yugabyte.com

yugabyte.com /slack

twitter.com /yugabyte

instagram.com /yugabyte

linkedin.com/company /yugabyte