Christian Rohmann | Matthias Hofschen

Best Practices for Operating HiveMQ and MQTT on Kubernetes

Technical White Paper
2021

HIVEMQ

inovex

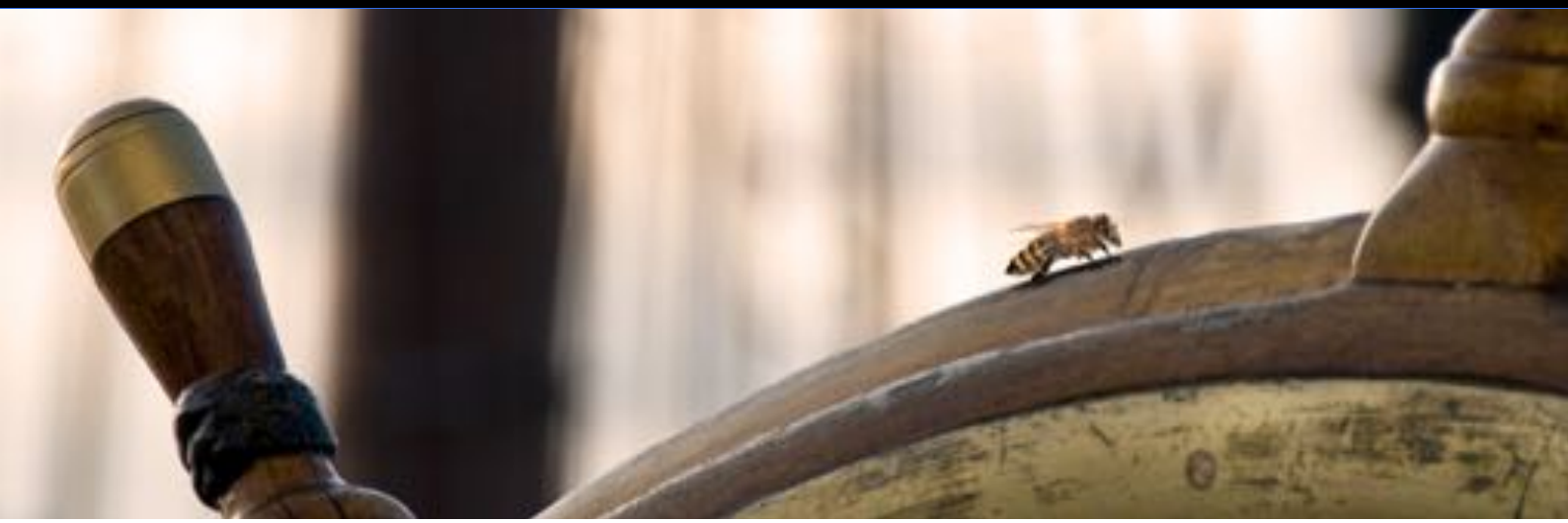


Table of Contents

Executive Summary

Kubernetes is the next step to provide a unified operational platform for cloud-native applications across most available public clouds and on-premise infrastructures. With the release of the HiveMQ Kubernetes Operator, the management and operation of HiveMQ clusters on Kubernetes is now significantly simplified. In this white paper, we show the benefits of using the HiveMQ Kubernetes Operator for running your IoT communication workloads on Kubernetes and share our experiences from the field.

Introduction

The Internet of Things (IoT) continues to drive enormous growth in the number of connected systems and devices. The majority of these connected devices use the MQTT protocol to communicate with each other and backend systems. To meet the demands of rapidly expanding networks of devices, MQTT solutions usually require high operational reliability and scalability; be it for vehicles that need mobility services, or machines on the factory floor that require predictive maintenance.

Many enterprises use DevOps teams to achieve the level of reliability and scalability their applications require. The DevOps teams are typically tasked with building and maintaining systems that are designed with operational automation, monitoring, and upgradeability in mind. Increasingly, infrastructure as code (IaC) provides the working foundation for the operational processes of these systems.

The ability of Kubernetes to address many of the operational needs and requirements such systems create has fueled its widespread adoption across numerous industries. Kubernetes abstracts the differences that exist between cloud providers and on-premise infrastructures. As an open-source container orchestration platform, Kubernetes provides a descriptive approach to defining, managing, and operating applications and workloads.

In particular, the Kubernetes operator pattern has emerged as a way to address the need for operational automation. The HiveMQ Kubernetes Operator uses this pattern to provide the extensive operational experience of the HiveMQ team encapsulated in code. The HiveMQ operator makes it possible to run your MQTT communication solution straight out of the box.

In this white paper, we discuss the best practices and benefits of running your production IoT applications using the HiveMQ MQTT broker at scale on Kubernetes and share practical experience from the field.

About HiveMQ

HiveMQ is one of the world's leading companies for connecting machines, devices, and applications in the Internet of Things (IoT) sector. Our product, the HiveMQ MQTT Broker, is based on the IoT standard communication protocol MQTT and enables absolutely secure and highly available data transfer between connected devices and the cloud at all times. What makes the HiveMQ MQTT broker unique is the high availability guarantee as well as the high scalability. More than 130 customers, including many Fortune 500 companies, rely on HiveMQ in production for business-critical use cases such as connected cars, transport and logistics, Industry 4.0, and connected IoT products.

www.hivemq.com

HiveMQ and MQTT

MQTT

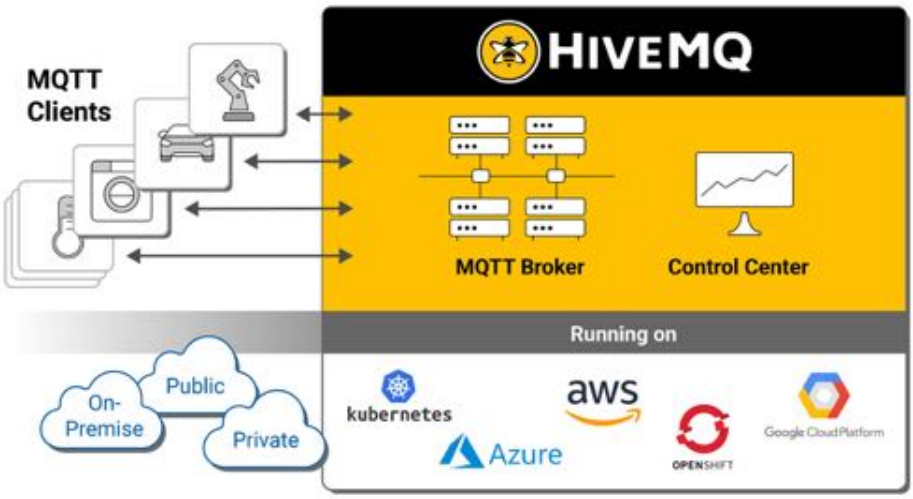
Throughout the Internet of Things (IoT), MQTT is the protocol of choice for communication between devices, machines, and humans. The MQTT messaging protocol is an ISO 20922 standard that was defined in 1999 by Andy Stanford-Clark and Arlen Nipper. Originally developed to support embedded-systems use-cases in the oil industry, MQTT is now the dominant protocol for IoT applications. Due in part to the protocol's long history, MQTT is well established across numerous industries and widely supported by most key players and technology solutions. The broad adaptation of MQTT makes the implementation and architecture of connected use cases a lot easier.

To learn more about the history and use of MQTT, read HiveMQ's free MQTT Essentials E-Book.

HiveMQ MQTT Broker

MQTT brokers are the central infrastructure components of every MQTT solution. The MQTT broker must be fail-safe, robust, scalable, and offer the necessary delivery guarantees for messages in high-performance, heavy-load scenarios. Most importantly, the MQTT broker needs to provide 100% compatibility with the MQTT specification. The MQTT specification defines many requirements for an MQTT broker to implement that reduce the responsibility of constrained devices and sensors. As the sheer volume of interconnected devices skyrockets, strict adherence to the MQTT specification is crucial for evolving use cases between highly disparate devices, systems, and architectures.

The HiveMQ MQTT broker fully supports the MQTT specification. In fact, the HiveMQ broker supports all MQTT specifications from MQTT 3 to MQTT 5. This complete adherence is essential for supporting diverse sets of devices and clients that often speak different levels of MQTT. Support for all MQTT features and versions makes it possible to seamlessly connect older and newer devices in a single network with one HiveMQ cluster.



HiveMQ runs in many different environments and infrastructures.

Operating the HiveMQ MQTT Broker

The HiveMQ Enterprise MQTT broker is usually operated as a cluster of nodes that are distributed across physical or virtual machines. Nodes in the cluster share the load of communicating with connected clients evenly. The number of cluster nodes for a specific use case is calculated based on the number of clients, message throughput, and message size.

The HiveMQ MQTT broker provides sophisticated clustering capabilities that ensure reliability, horizontal scalability, and performance for a wide array of MQTT use cases. HiveMQ's unique clustering mechanism is highly adaptable to different environments and can be customized depending on the use case and environment. A HiveMQ cluster can scale elastically during operations to adapt to changing workloads. As new nodes are added to a cluster, the workload is spread to include the new nodes. Likewise, if the cluster size is reduced the workload is redistributed.

The elasticity of a HiveMQ cluster also enables zero-downtime-upgrades. Newly-upgraded nodes join the cluster and take over part of the workload. Existing nodes can be shut down and removed from the cluster while their workload is distributed across the other cluster nodes. This methodology of rolling upgrades provides migration support for new versions and configuration changes during runtime. Now let's take a look at Kubernetes and how to operate your HiveMQ MQTT workloads on Kubernetes.

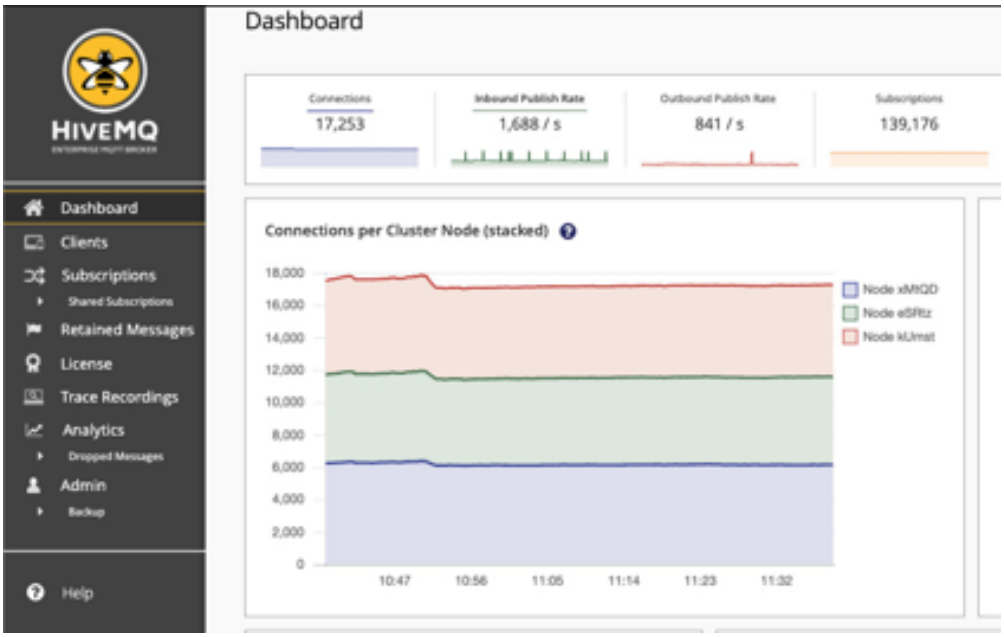
Kubernetes

Cloud computing has seen wide adoption across many industries. One of the advantages cloud computing offers is that compute resources can be provisioned and used elastically without time-consuming procurement processes. In addition, cloud providers generally offer ready-to-consume managed services such as databases, queues, and more. These services can be provisioned via API in seconds and subsequently used as building blocks for complex IT infrastructures.

Companies that explore cloud-native approaches to run application workloads often shift their architecture towards a landscape of microservices that are made up of containerized applications. The study "*3 Critical Mistakes That I&O Leaders Must Avoid With Containers*" from Gardner states: "*By 2023, more than 70% of global organizations will be running more than two containerized applications in production, up from less than 20% in 2019.*"

The attraction of containerized applications that can be easily developed and tested on an individual developer machine, then conveniently deployed, integrated, and run on large scale production infrastructures has certainly helped cloud computing gain momentum. The concepts and foundational technologies for cloud computing have been available for some time, but it was the introduction of Docker in 2013 that significantly accelerated the adoption of cloud computing.

The biggest catalyst for moving application workloads to a cloud-based environment has been the release of Kubernetes by Google in 2014. Kubernetes is an open-source container orchestration software that is based on the experiences that Google made with their internal



HiveMQ's Dashboard offers out of the box insights for your MQTT workloads.

system called Borg. Since its launch, Kubernetes has received unprecedented attention within the IT industry. Users and contributors to the project range from the well-known hyperscalers and traditional infrastructure providers to software vendors and individual developers. As an open-source project governed by the Cloud Native Computing Foundation (CNCF), Kubernetes is not owned or controlled by a single vendor. Hence there is no danger of vendor lock-in.

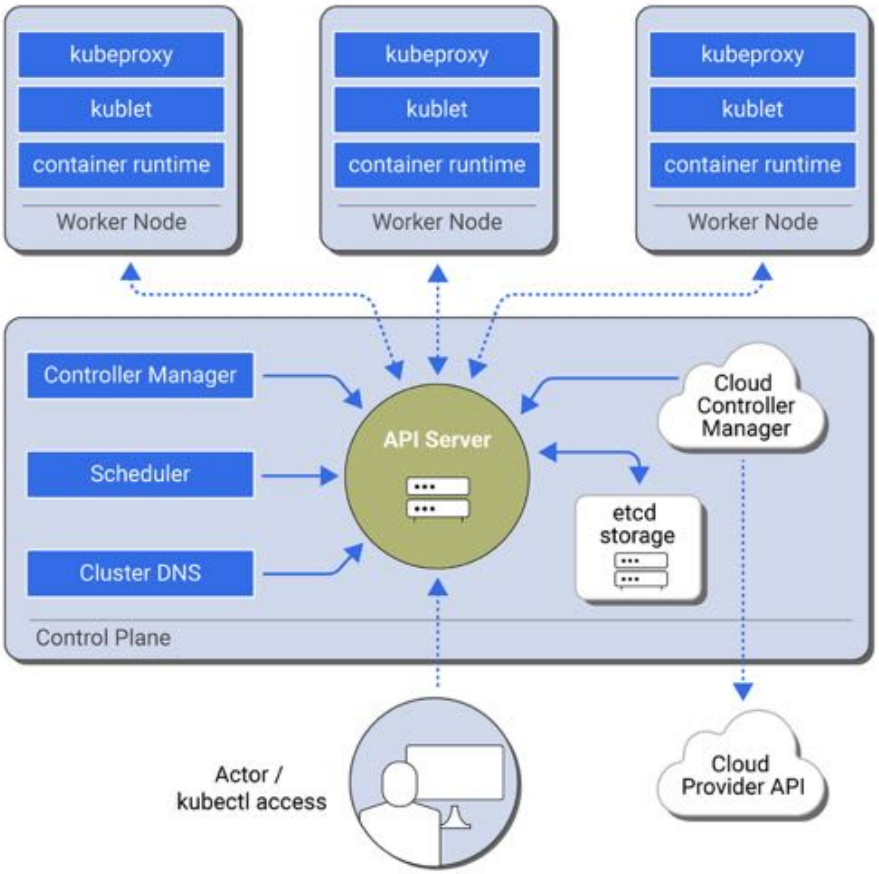
In summary, Kubernetes, K8s for short, is currently the quintessential ecosystem for the deployment and operation of cloud-native containerized applications.

Kubernetes Architecture

Kubernetes is a distributed system that consists of a control-plane and a data-plane architecture. The control plane hosts the API server and other active components known as controllers. Data accessed via the API server is persisted in Etcd, a key-value store. Etcd usually runs in a distributed way and provides consensus on the current state of all API objects that are stored. The API server is the central hub for all other component communications in Kubernetes. Users such as developers or operators also interact with the API server through tools such as *kubectl* or Helm.

The real work of running application workloads is performed by worker machines called nodes. Nodes are usually installed with a flavor of the Linux operating system and run a component called the *Kubelet*. The *Kubelet* agent cooperates with a container runtime such as *Containerd* to manage the application containers on each node. *Kubelet* manages fetching images, starting and stopping containers, monitoring and collecting resource usage or log output, and other aspects of the container lifecycle.

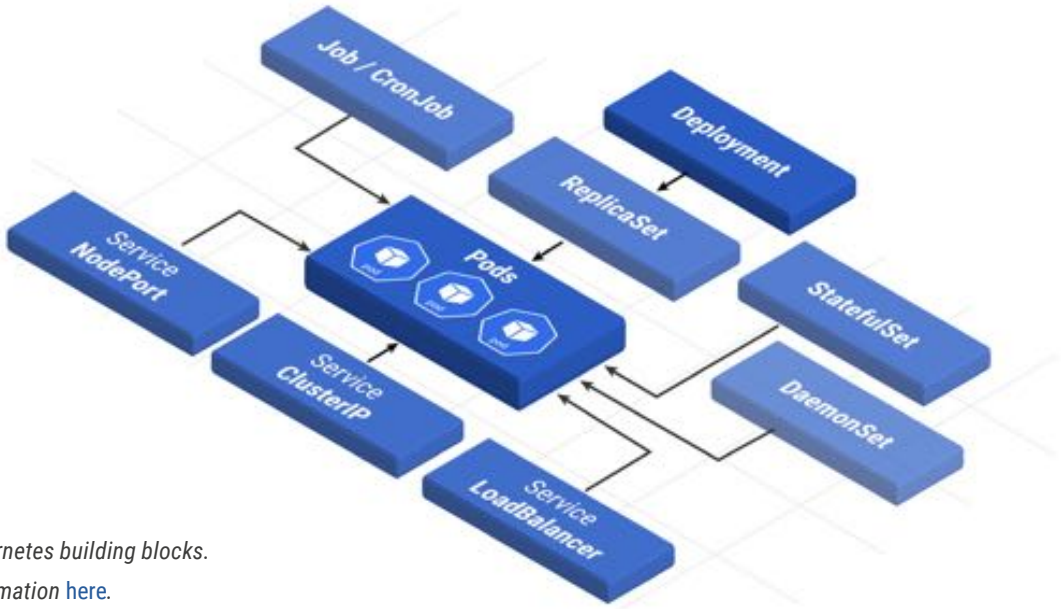
Kubelet constantly communicates with the API server to check the desired state of an application and reports back the current state of its node and the running application pods. As a built-in core resource type, pods bundle one or more application containers. The constant reconciliation of desired state versus existing state is referred to as a control-loop and represents one of the core paradigms of Kubernetes.



The Kubernetes architecture consists of a control plane and worker nodes. Find a more detailed description here.

Kubernetes building blocks

Resources, such as *Deployments, Jobs, ReplicaSets, StatefulSets* and *DaemonSets* as well as *Pods* and *Services* provide the basic building blocks to assemble an application architecture on Kubernetes.



The basic Kubernetes building blocks. Find more information here.

Different controllers operate on each of the resources and run specific control-loops. The controllers are fully independent of each other. Addons enable Kubernetes to manage outside resources. For example, the Cloud Controller is used to manage load balancers of cloud providers such as AWS or Azure. Other controllers can manage persistent storage and attach storage to containers.

The simple but powerful combination of a standardized, yet extensible and versioned API on one side with independently acting controllers on the other side extends beyond the Kubernetes core project. This concept of extensibility has spawned a continuous wave of additional custom functionality. The CNCF publishes an ever-growing landscape of software and providers that integrate with Kubernetes.

The Kubernetes operator pattern is an example of the extensibility of the Kubernetes API. Custom functionality and custom resources can be added by simply creating an API object that is described in a YAML file and a controller that acts on behalf of the new API. The new resources are submitted to the Kubernetes API using the standard *kubectl* command line client or packaging tools such as Helm and become available for usage.

The Kubernetes Operator Pattern

The core resources of Kubernetes, such as pods and services cover the most common requirements for creating stateless applications and can be extensively combined and configured to form application architectures. Stateless applications usually don't require any special treatment in terms of distributed storage or consensus to reach a healthy state. For example, the pods of a stateless application can be easily upgraded to a new version without having to consider storage requirements or state.

For stateful applications, Kubernetes provides some limited support in the form of StatefulSets. However, StatefulSets do not address many of the specific needs for real stateful and distributed applications such as databases, caches, or queues.

In 2016, CoreOS introduced the operator pattern to address the challenge of managing stateful applications on Kubernetes with this blog post.

CoreOS proposed the creation of a custom API for a new Kubernetes resource and a corresponding controller for managing the resource. Since the Kubernetes API can be extended, custom resource APIs can be easily added to the API server. This new resource API is called the Custom Resource Definition (CRD). CRDs are API objects and hold the fields and basic syntax restrictions defining the new resource.

The custom controller, commonly known as an operator, communicates with the API server to handle the new API's resource requests. The operator encapsulates the domain knowledge needed to implement and manage a particular kind of application and its specific workflows and lifecycles. Similar to the way that a human administrator can be asked to set up or scale a database instance, increase storage, or create a backup, such tasks can be implemented with the operator.

Since its introduction, the Kubernetes operator pattern has been applied to hundreds of different applications with new operators reporting for duty every week.

The HiveMQ Kubernetes Operator

The HiveMQ MQTT broker can run in many different environments, ranging from on-premise bare metal servers to virtual machines or compute instances on public cloud providers. The HiveMQ broker can also run as a container and HiveMQ provides readily built images for this use case.

Although environments vary, the tasks required to operate a HiveMQ cluster remain similar:

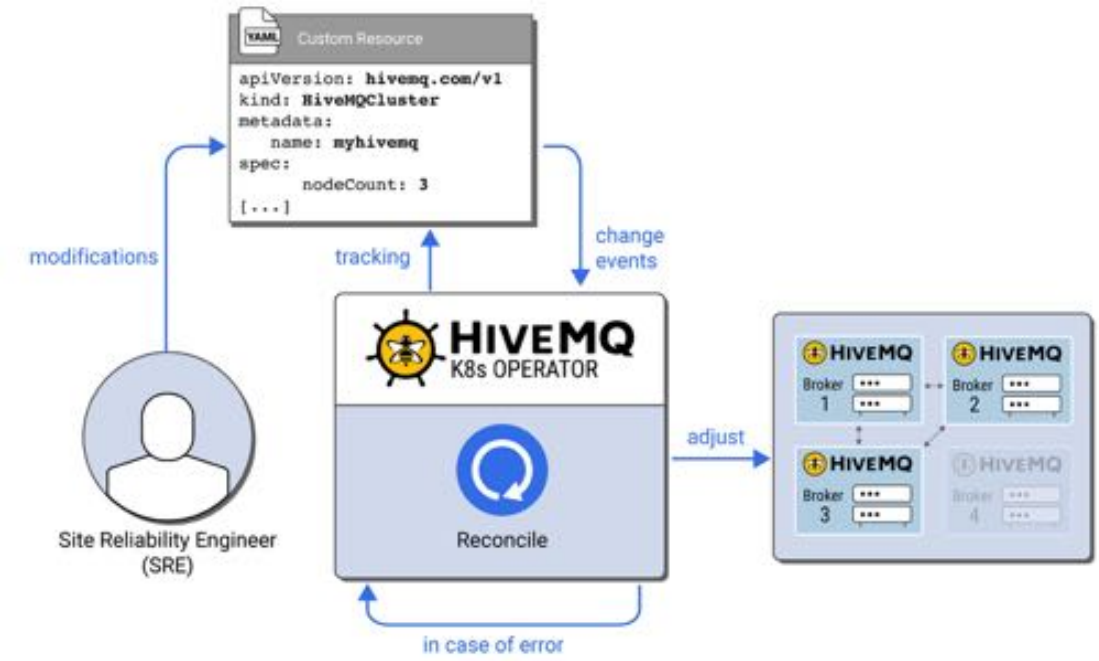
- spawning multiple HiveMQ nodes for high availability
- setting up service discovery to form a cluster
- configuring a load balancer to distribute incoming client connections

Depending on the infrastructure, specific steps and procedures are needed to install HiveMQ. For example, this blog post explains the installation on AmazonWebServices (AWS). Other approaches to install HiveMQ use tools such as Terraform and are explained in this blog post. Regardless of which infrastructure is used, all target platforms require customized code and tools. The customized code prevents easy reusability and portability of infrastructure code between platforms.

By contrast, Kubernetes as a container platform provides an abstraction layer that unifies the installation on different infrastructures. Kubernetes creates a common standardized deployment platform across different environments and providers. Developers and system administrators can construct tooling and CI/CD pipelines in an agnostic way. Naturally, Kubernetes also provides the environment and functionality to install and run a HiveMQ cluster.

HiveMQ containers can be started as pods and placed (scheduled) onto different worker nodes to evenly distribute the MQTT workloads and handle individual pod failures without service downtime. Kubernetes-internal DNS provides service discovery and is used as cluster discovery for HiveMQ pods. The result is a masterless HiveMQ cluster. If a Cloud Controller is available, a service of type LoadBalancer can be defined. The HiveMQ MQTT service is then reachable via the defined external LoadBalancer from outside the Kubernetes cluster.

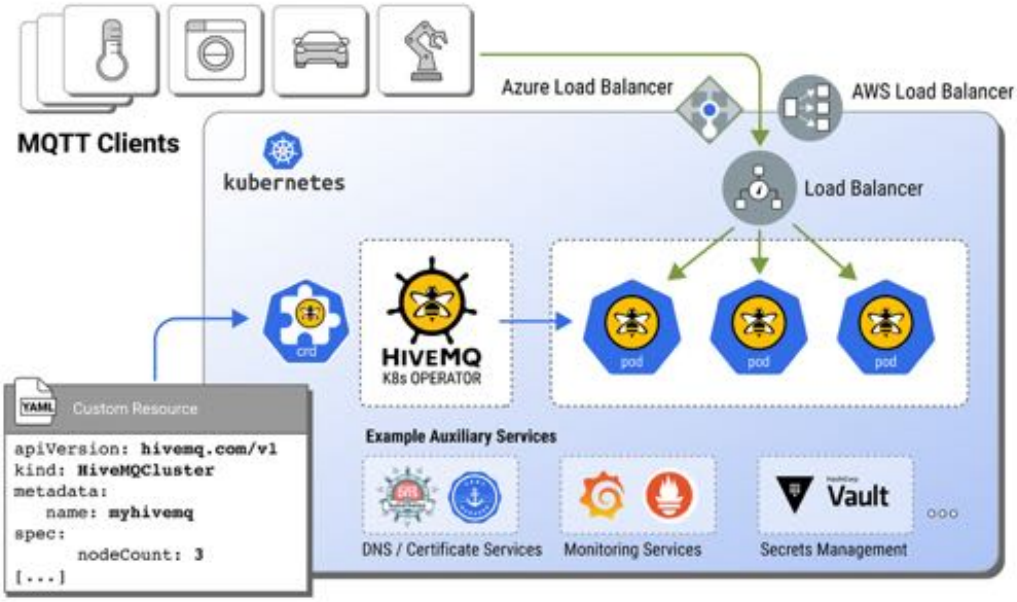
HiveMQ introduced the HiveMQ Kubernetes Operator to automate and simplify this process of installing and operating HiveMQ MQTT clusters on Kubernetes.



The HiveMQ Kubernetes Operator reconciles the HiveMQ cluster definition with the running HiveMQ cluster.

As the diagram illustrates, the HiveMQ Kubernetes Operator only requires the configuration of a single resource, the *HiveMQCluster* custom resource. The *HiveMQCluster* is a new API object defined by a Custom Resource Definition (CRD) that is registered with Kubernetes. Developers and system administrators can now install and operate HiveMQ clusters as one consistent Kubernetes API object by submitting a single YAML file that describes how the HiveMQ cluster should be configured. The HiveMQ Kubernetes Operator ensures that the desired state of the cluster is installed on Kubernetes and regularly checks that the running HiveMQ cluster matches the desired state.

The YAML file defines relevant configuration options such as the number of nodes desired for the cluster or the version of HiveMQ that is installed. The defined configuration is analyzed and transformed by the HiveMQ Kubernetes Operator into the required resources that are provided by the Kubernetes ecosystem, such as *Deployments, Services, ConfigMaps*, and others.



The HiveMQ cluster and the HiveMQ Kubernetes Operator integrate seamlessly into the Kubernetes ecosystem

The HiveMQ Kubernetes Operator can also connect a HiveMQ cluster with other Kubernetes resources. In Kubernetes environments, the popular Prometheus and Grafana monitoring applications are often used to collect and visualize application metrics. HiveMQ integrates seamlessly with this monitoring stack and provides its metrics in the corresponding format via the HiveMQ Prometheus-Extension. The HiveMQ Operator creates the configuration to automatically monitor the HiveMQ cluster with Prometheus and Grafana. Monitoring dashboards for Grafana are included.

In some use cases, it makes sense to include other external systems and resources. For example, HashiCorp Vault to manage secrets for TLS certificates, CertManager to manage certificate creation, or External-DNS to manage DNS records in external DNS systems such as AWS Route53.

The HiveMQ Kubernetes Operator can handle much more than transforming a list of variables into a running HiveMQ cluster. Just like a knowledgeable and experienced human operator, the HiveMQ Kubernetes Operator verifies the desired HiveMQ cluster resource definition for consistency and accuracy. The HiveMQ operator verifies that all definitions and changes are correctly applied to the HiveMQ cluster. If a configuration could cause issues, the operator actively rejects the change and provides a response with an explanation.

Take a look at our Quick Start guide to get a better understanding of the HiveMQ Kubernetes Operator and to try it out. Helm charts are provided as well that make it easy to try out a HiveMQ cluster in your Kubernetes environment.

HiveMQ and Kubernetes best practices

Storage Considerations for HiveMQ

MQTT is a real-time IoT protocol. In general, MQTT brokers deliver messages to subscribed clients immediately. However, to guarantee the delivery of QoS 1 and 2 messages, storage must be part of the workflow. For example, the broker may need to store messages for subscribers that are temporarily disconnected. IoT solutions that utilize MQTT retained messages require storage as well. As a distributed system, HiveMQ replicates queued data among its cluster nodes. The replication ensures that the loss of a single HiveMQ node cannot cause any data loss. The replication factor can be configured and needs to be taken into account when configuring storage.

The HiveMQ cluster nodes that the HiveMQ Kubernetes Operator deploys and manages are configured with an ephemeral storage volume that holds 15 GB of data by default. For production scenarios, HiveMQ requires a minimum of 100 GB of storage. The storage configuration can be easily adjusted. In all cases, Kubernetes worker nodes must be configured with enough local storage to accommodate the amounts requested by the HiveMQ pods. It is good practice to monitor storage usage such as IOPS and throughput. Monitoring and adequate configuration ensure that storage does not become a bottleneck for the performance of the HiveMQ broker.

Find more details on message storage in this inovex blog post.

Networking Aspects in Kubernetes

There are two networking aspects to consider in Kubernetes:

- **North/South traffic**: This networking aspect represents network requests that are routed into the cluster from external systems and are served by application containers running in pods.
- **East/West traffic**: This networking aspect represents network requests that are routed inside the cluster between pods, services, and other Kubernetes resources.

Kubernetes treats MQTT traffic as a plain TCP service with no application-layer proxy in front of the broker. MQTT clients usually maintain long-lasting connections to the MQTT broker. For efficiency reasons, clients tend to avoid frequent connects/reconnects. This behavior results in thousands or even millions of clients holding persistent TCP connections to the MQTT broker and therefore also to Kubernetes.

Most of the networking inside the Kubernetes cluster is handled by plugins that are compatible with the Kubernetes container networking interface. Depending on the Kubernetes environment and user requirements, these CNI plugins cover different aspects of networking. For example, packet filtering or the encryption of internal traffic. There can be different reasons to select a specific CNI plugin. Project Calico and Cilium are two popular choices, but numerous other options are available. Kubernetes as a Service (KaaS) providers frequently offer a default CNI implementation that utilizes their existing virtual networking as well as custom configuration options. You can learn more about the essentials of Kubernetes networking in this blog post.

In addition to the CNI plugins, *kube-proxy* is a Kubernetes component that runs on each node to manage *iptables* or *ipvs* rules. *Kube-proxy* manages access to individual pods through defined services. Some CNI plugins also use *iptables* to apply additional packet filtering to the network traffic inside the Kubernetes cluster. Each connection is inserted into the *conntrack* tables of the Linux Kernel. A high number of TCP connections or a high churn rate places significant stress on this stack. Depending on your particular connection scenario and the available memory and CPU, some configuration adjustments may be necessary. *Kube-proxy* options are available here.

It is important to monitor the metrics and limits over time. The *Node-Exporter* from Prometheus provides the required metrics and additional insights into the tcpstat subsystem of the kernel.

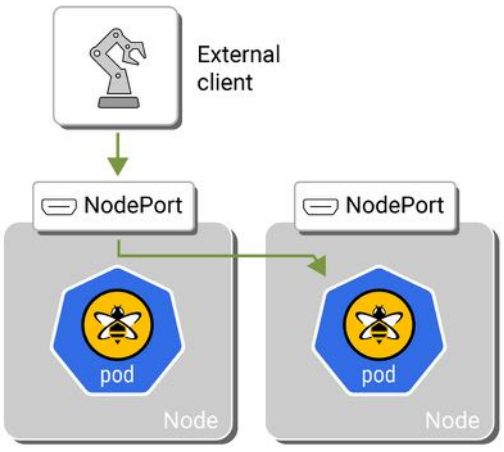
If the *iptables / conntrack* stack is no longer able to scale to the required number of connections, consider using *Cilium CNI*. *Cilium uses eBFP* for packet forwarding and can replace kube-proxy.

Load Balancing Client Connections

Usually an external load balancer handles incoming TCP connections (north/south traffic) to applications deployed on Kubernetes. Public cloud offerings typically provide load balancing as a managed service. For on-premise Kubernetes clusters, the load balancer can also be a physical appliance.

For each service of type LoadBalancer Kubernetes opens a common NodePort on each of its worker nodes. Each node is then configured as a backend for the external load balancer. The load balancer, however, has no knowledge of the application pod placement on Kubernetes nodes, and therefore attempts to forward TCP connections to all nodes equally.

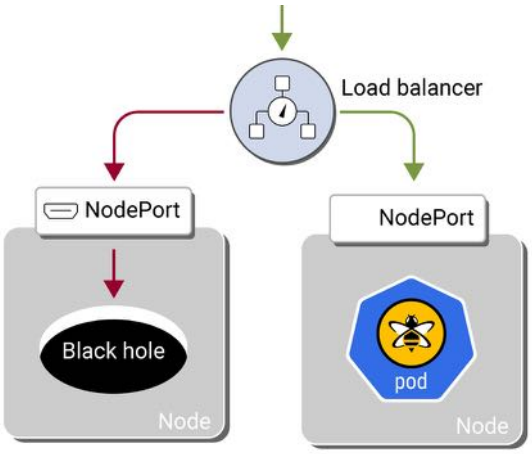
With the *externalTrafficPolicy* setting of a service, the distribution of TCP connections to Kubernetes nodes can be influenced. The default setting for the *externalTrafficPolicy* is *Cluster*. External connections are forwarded from the load balancer to any Kubernetes node and from there to a Pod belonging to the relevant application service. The pod that finally receives the connection can be on the local node or any other node within the cluster.



Example with externalTrafficPolicy set to cluster.

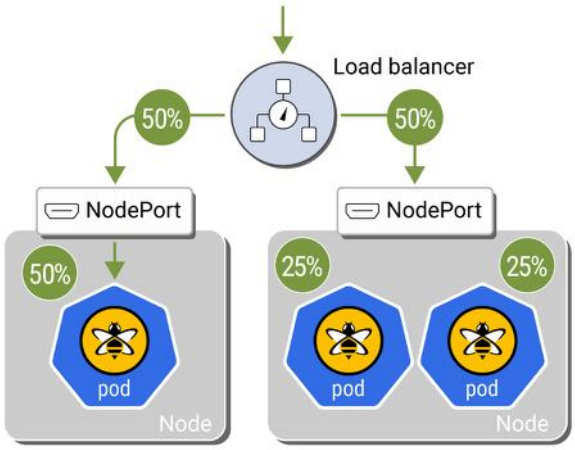
Therefore the incoming (north/south) connection often has to be routed via an additional, cluster-internal (east/west) connection even though a suitable HiveMQ pod is available locally. This extra forwarding decision ensures close to equal distribution among all pods of a service, but the additional (east/west) connection has to be managed at the kernel level and uses additional resources.

Setting the *externalTrafficPolicy* to *Local*, directs network traffic only to a node's local pods. While this approach seems more efficient at first glance, two issues need to be considered.



Example with externalTrafficPolicy set to local and a node without a HiveMQ pod causing traffic to be blackholed.

First, as visualized in the graphic above, when the node that receives the connection lacks a locally running HiveMQ pod, the connection fails. To avoid connection failure, the load balancer must only forward connections to nodes that have suitable and healthy pods. To achieve this distribution, the load balancer must be configured with a health check that only reports K8s nodes with HiveMQ pods. In most environments, no manual action is required, the cloud controller also manages the health-check configuration of the external load balancers.



Example with externalTrafficPolicy set to cluster and an uneven pod distribution.

Secondly, if the HiveMQ pods are unevenly distributed across Kubernetes nodes, the pods can receive uneven traffic loads. Since the load balancer does not know how many pods of each service run on each node, the load balancer sends an equal number of connections to every available node. As a result, HiveMQ pods can receive an uneven number of TCP connections.

The HiveMQ Kubernetes Operator automatically sets affinity rules to avoid such a scenario. These affinity or anti-affinity rules prompt Kubernetes to spread HiveMQ pods evenly

across worker nodes and avoid co-locating HiveMQ pods. This distribution requires that enough physical nodes are available so that each HiveMQ pod can be placed onto a unique worker node. The default affinity rules will also try to avoid co-locating pods from other HiveMQ clusters on the same worker nodes if possible. The affinity rules can be configured via the HiveMQ Operator Helm chart.

The externalTrafficPolicy and other settings can be configured to your particular requirements via the port-patch configuration section of the HiveMQ Kubernetes Operator. The HiveMQ documentation contains more details.

TCP Keepalive for MQTT

MQTT has its own KEEPALIVE mechanism. The interval for this mechanism must align with the TCP idle timeout settings of the infrastructure that is used. Check the settings for your AWS Network Load Balancer (NLB), Azure Load-Balancer or any other load balancer used. For example, if the external load balancer drops TCP connections after 350 seconds of idle time, then the MQTT KEEPALIVE should be set accordingly to a lower setting.
If the MQTT KEEPALIVE cannot be reconfigured to a lower setting, then TCP keep-alives can be set on the Kubernetes node to avoid a connection teardown. The following kernel settings serve as an example:

```
# cat /etc/sysctl.d/tcpkeepalive.conf

net.ipv4.tcp_keepalive_time=300
net.ipv4.tcp_keepalive_intvl=300
net.ipv4.tcp_keepalive_probes=2
```

More information on setting sysctl configurations is available here.

Kubernetes Custom Resource Definitions

Keep in mind that some resources in Kubernetes are namespaced and others are not. To find out which API resources are namespaced issue this command: *kubectl api-resources*. The resulting list shows, for example, that *CustomResourceDefinitions* are not namespaced:

"customresourcedefinitions crd,crds apiextensions.k8s.io **false** CustomResourceDefinition"

The *CustomResourceDefinition* (CRD) for the *HiveMQCluster* resource applies to all HiveMQ clusters that are running on the selected Kubernetes environment.

The HiveMQ Kubernetes Operator is usually installed via Helm using a preconfigured HiveMQ Helm Chart that includes the installation of the *CustomResourceDefinition* for the *HiveMQCluster*. If the *CustomResourceDefinition* is deleted, Kubernetes automatically uninstalls all deployed *CustomResources* that are associated with the deleted CRD. If the CRD for the *HiveMQCluster* resource is removed, Kubernetes will uninstall all HiveMQ clusters regardless of their namespace.

Additional Resources

The HiveMQ broker comes with extensive documentation that covers configuration options and many operational topics. The HiveMQ Kubernetes Operator documentation shows how to configure HiveMQ specifically for Kubernetes. A quick start guide is included in the documentation to help with the Helm commands that are needed for the installation.

Feel free to explore the installation of a HiveMQ cluster on a Kubernetes environment of your choice (minimum Kubernetes version 1.13 and Helm 3). Free trial licenses allow you to fully test your cluster installation with a limited number of client connections.

As always, if you have questions, don't hesitate to contact inovex or HiveMQ. We are happy to help.

- inovex blog https://www.inovex.de/blog/
- HiveMQ blog https://www.hivemq.com/blog
- HiveMQ documentation: https://www.hivemq.com/docs

Summary and Outlook

Kubernetes is a flexible and rapidly evolving ecosystem for operating containerized applications. The Kubernetes operator pattern and the HiveMQ Kubernetes Operator offer the advantage of running your MQTT workloads on Kubernetes using the experience and automation provided by the HiveMQ team. Operations teams benefit from increased productivity and improved workflows through the automation of complex distributed systems tasks. In situations where Kubernetes is already used for deploying containerized applications, it makes sense to add MQTT workloads to Kubernetes as well. The standardization of deployments and operations for Kubernetes makes it easier to avoid vendor lock-in especially in public cloud environments.

HiveMQ GmbH
Ergoldingerstr. 2a
84030 Landshut
Germany

www.hivemq.com

inovex GmbH
Ludwig-Erhard-Allee 6
76131 Karlsruhe
Germany

www.inovex.com