Published in Level Up Coding

Alvin Lee Follow

Apr 27 · 7 min read · · Listen

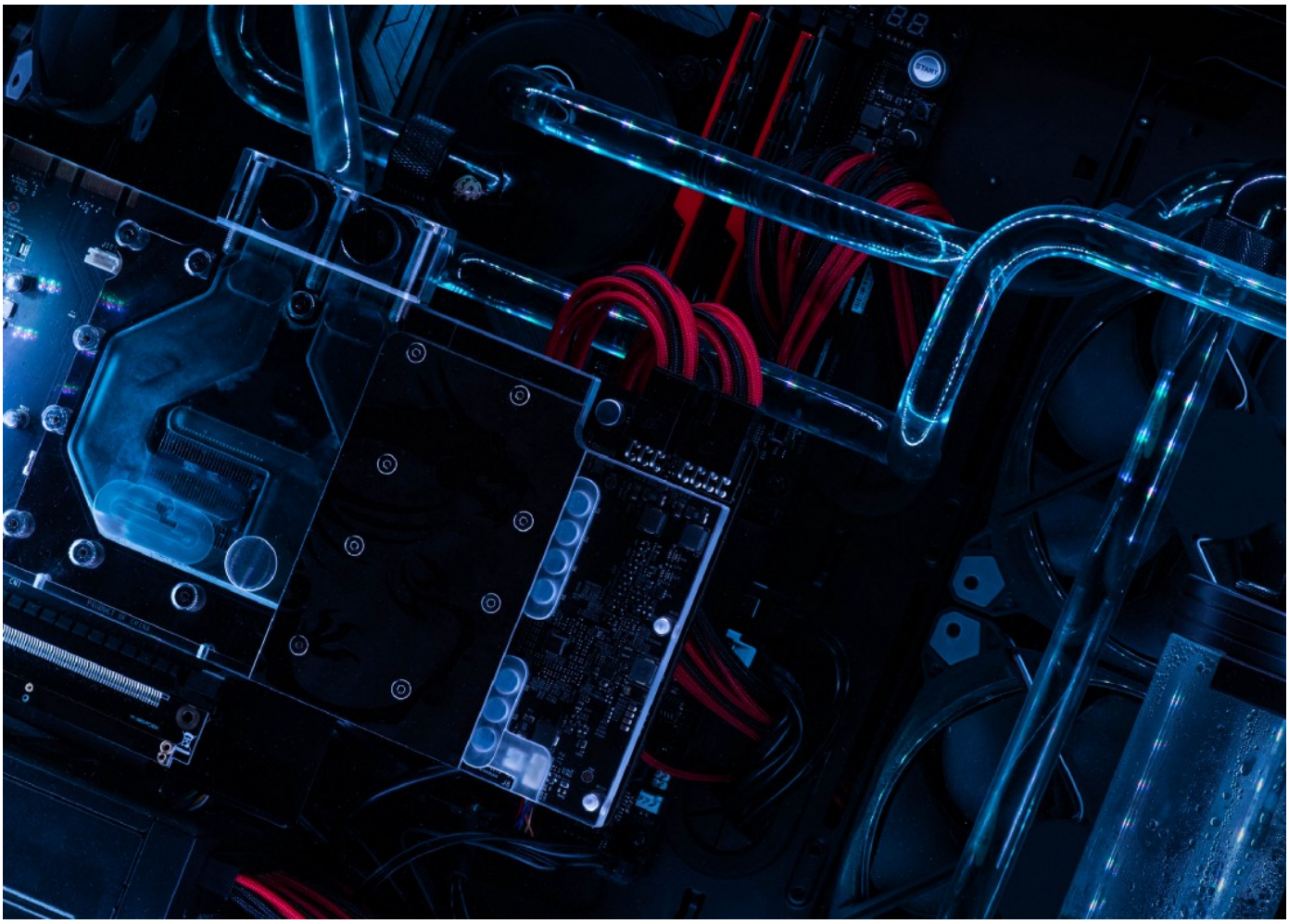

Photo by Cristiano Firmani on Unsplash

CODING IN A COFFEE SHOP

How To Deploy Apache Kafka With Kubernetes

orchestrating containerized services. For many organizations, deploying Kafka on Kubernetes is a low-effort approach that fits within their architecture strategy.

In this post, we'll look at the appeal of hosting Kafka on Kubernetes, providing a quick primer on both applications. Finally, we'll walk through a cloud-agnostic method to configure Kubernetes for deploying Kafka and its sibling services.

A Primer on the Tech

Let's start with a quick overview of Kubernetes and Kafka.

Kubernetes

Google started developing what eventually became Kubernetes (k8s) in 2003. At the time, the project was known as Borg. In 2014, Google released k8s as an open-source project on Github, and k8s quickly picked up partners through Microsoft, Red Hat, and Docker. Over the years, more and more endeavors used Kubernetes, including GitHub itself and the popular game, Pokémon GO. In 2022, we see k8s usage growing in the AI/ML space and with an increasing emphasis on security.

The introduction of k8s into the cloud development lifecycle provided several key benefits:

- Zero downtime deployments
- Scalability
- Immutable infrastructure
- Self-healing systems

Many of these benefits come from the use of declarative configuration in k8s. In order to change an infrastructure configuration, resources must be destroyed and rebuilt, thereby enforcing immutability. In addition, if k8s detects resources that have drifted out of the declared specification, it attempts to rebuild the state of the system to match that specification again.




production environments directly by hand.

Apache Kafka

Kafka is an open-source distributed stream processing tool. Kafka allows for multiple "producers" to add messages (key-value pairs) to "topics". These messages are ordered in each topic as a queue. "Consumers" subscribe to the topic and can retrieve messages in the order they arrived in the queue.

Kafka is hosted on a server typically called a "broker." There can be many different Kafka brokers in different regions. In addition to Kafka brokers, another service named Zookeeper keeps different brokers in sync and helps coordinate topics and messages.

The brilliance of Kafka is that it can handle hundreds of thousands of messages a second while being distributed and at a relatively cheap cost per MB.

Kafka often occupies a spot akin to the "central nervous system" of a microservice architecture. Messages are passed along between producers and consumers, which in reality, are services inside your cloud. The same service could both be a consumer and a producer of messages from the same or different topics inside Kafka.

An example use case is creating a new user in your application. The User service publishes a message on a "Provision User" topic. The Email service consumes this message about a new user and then sends a welcome email to them. The User and Email services did not have to directly message each other, but their respective jobs were executed asynchronously.

Deploying Kafka With Kubernetes

For our mini project walkthrough, we'll set up Kubernetes and Kafka in a cloud-neutral way using Minikube, which allows us to run an entire k8s cluster on a single machine. We installed the following applications:

- Minikube version v1.25.2




Minikube Setup

With Minikube installed, we can start it with the minikube start command. Then, we can see the status:

```
$ minikube status

minikube

type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Instructions for setting up Kubernetes to run in your cloud provider of choice can be found in the documentation for each provider (for example, AWS, GCP, or Azure), but the YAML configuration files listed below should work across all providers, with minor adjustments for IP addresses and related fields.

Defining a Kafka Namespace

First, we define a namespace for deploying all Kafka resources, using a file named `00-namespace.yaml`:

```
apiVersion: v1
kind: Namespace
metadata:
  name: "kafka"
  labels:
    name: "kafka"
```

We apply this file using `kubectl apply -f 00-namespace.yaml`.

We can test that the namespace was created correctly by running `kubectl get namespaces`, verifying that Kafka is a namespace present in Minikube.




```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: zookeeper-service
  name: zookeeper-service
  namespace: kafka
spec:
  type: NodePort
  ports:
    - name: zookeeper-port
      port: 2181
      nodePort: 30181
      targetPort: 2181
  selector:
    app: zookeeper
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: zookeeper
  name: zookeeper
  namespace: kafka
spec:
  replicas: 1
  selector:
    matchLabels:
      app: zookeeper
  template:
    metadata:
      labels:
        app: zookeeper
    spec:
      containers:
        - image: wurstmeister/zookeeper
          imagePullPolicy: IfNotPresent
          name: zookeeper
          ports:
            - containerPort: 2181
```

There are two resources created in this YAML file. The first is the service called




internal k8s network. In this case, we use the standard Zookeeper port of `2181`, which the Docker container also exposes.

We apply this file with the following command: `kubectl apply -f 01-zookeeper.yaml`.

We can test for the successfully created service as follows:

```
$ kubectl get services -n kafka
NAME                TYPE       CLUSTER-IP      PORT(S)          AGE
zookeeper-service   NodePort   10.100.69.243   2181:30181/TCP   3m4s
```

We see the internal IP address of Zookeeper (`10.100.69.243`), which we'll need to tell the broker where to listen for it.

Deploying a Kafka Broker

The last step is to deploy a Kafka broker. We create a `02-kafka.yaml` file with the following contents, be we replace `<ZOOKEEPER-INTERNAL-IP>` with the `CLUSTER-IP` from the previous step for Zookeeper. The broker will fail to deploy if this step is not taken.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: kafka-broker
  name: kafka-service
  namespace: kafka
spec:
  ports:
  - port: 9092
  selector:
    app: kafka-broker
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
```




```
  selector:
    matchLabels:
      app: kafka-broker
  template:
    metadata:
      labels:
        app: kafka-broker
    spec:
      hostname: kafka-broker
      containers:
      - env:
        - name: KAFKA_BROKER_ID
          value: "1"
        - name: KAFKA_ZOOKEEPER_CONNECT
          value: <ZOOKEEPER-INTERNAL-IP>:2181
        - name: KAFKA_LISTENERS
          value: PLAINTEXT://:9092
        - name: KAFKA_ADVERTISED_LISTENERS
          value: PLAINTEXT://kafka-broker:9092
        image: wurstmeister/kafka
        imagePullPolicy: IfNotPresent
        name: kafka-broker
        ports:
        - containerPort: 9092
```

Again, we are creating two resources — service and deployment — for a single Kafka Broker. We run `kubectl apply -f 02-kafka.yaml`. We verify this by seeing the pods in our namespace:

```
$ kubectl get pods -n kafka
NAME                            READY   STATUS    RESTARTS   AGE
kafka-broker-5c55f544d4-hrgnv   1/1     Running   0          48s
zookeeper-55b668879d-xc8vd      1/1     Running   0          35m
```

The Kafka Broker pod might take a minute to move from `ContainerCreating` status to `Running` status.

Notice the line in `02-kafka.yaml` where we provide a value for




```
127.0.0.1 kafka-broker
```

Testing Kafka Topics

In order to test that we can send and retrieve messages from a topic in Kafka, we will need to expose a port for Kafka to make it accessible from localhost. We run the following command to expose a port:

```
$ kubectl port-forward kafka-broker-5c55f544d4-hrgnv 9092 -n kafka
Forwarding from 127.0.0.1:9092 -> 9092
Forwarding from [::1]:9092 -> 9092
```

The above command `kafka-broker-5c55f544d4-hrgnv` references the k8s pod that we saw above when we listed the pods in our `kafka` namespace. This command makes the port `9092` of that pod available outside of the Minikube k8s cluster at `localhost:9092`.

To easily send and retrieve messages from Kafka, we'll use a command-line tool named KCat (formerly Kafkacat). To create a message and a topic named `test`, we run the following command:

```
$ echo "hello world!" | kafkacat -P -b localhost:9092 -t test
```

The command should execute without errors, indicating that producers are communicating fine with Kafka in k8s. How do we see what messages are currently on the queue named `test`? We run the following command:

```
$ kafkacat -C -b localhost:9092 -t test
```




Next Steps

Kafka is a key player for organizations that are interested in implementing real-time, event-driven architectures and systems. Deploying Kafka with Kubernetes is a great start, but organizations will also need to figure out how to make Kafka work seamlessly and securely with their existing API ecosystems. What they'll need are tools to handle management and security for the entire lifecycle of their APIs.

Among the providers out there, I came across Gravitee, one of the leading solutions that's particularly focused on helping organizations manage, secure, govern, and productize their API ecosystems — no matter what protocols, services, or styles they're building on top of. Gravitee even has a Kafka connector that ingests data by exposing endpoints that transform requests into messages that can then be published to your Kafka topic. It can also stream Kafka events to consumers with web-friendly protocols like Websocket.

Conclusion

In this article, we've talked about how Kafka helps choreograph microservice architectures by being a central nervous system relaying messages to and from many different services. For deploying Kafka, we've looked at Kubernetes, a powerful container orchestration platform that you can run locally (with Minikube) or in production environments with cloud providers. Lastly, we demonstrated how to use Minikube to set up a local Kubernetes cluster, deploy Kafka, and then verify a successful deployment and configuration using KCat.

Happy deploying!