Microservices on Kubernetes

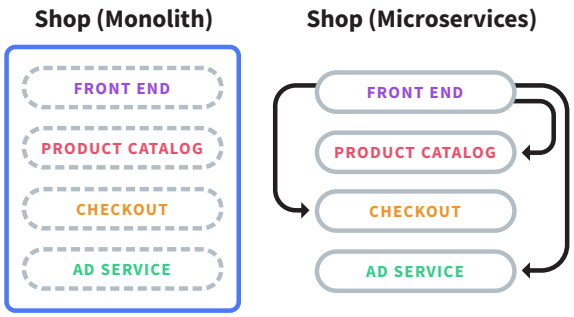
CONTENTS

WRITTEN CHRISTIAN MELENDEZ CLOUD ARCHITECT AT EQUINIX EMEA

Microservices is not a new architecture style. Back in 2014, Martin Fowler and James Lewis published a post defining microservices. One year later, Kubernetes (the most popular container orchestration at the moment) was open-sourced by Google. In this Refcard, you will find an introduction to these technologies, as well as the essential characteristics and design principles of microservices in an orchestration world. This Refcard will also dive into monitoring and logging, which are essential tools in distributed systems like microservice architectures.

Microservices Overview

Microservices is an architectural style where every piece of a system lives separately, as an individual service — rather than condensing all the different parts of a system into a single, larger service, as happens with traditional monolithic architectures. It is useful to keep in mind that microservices should not mean as small as possible, but rather should be taken to mean as small as *necessary*.



Having multiple services instead of just one brings increased levels of complexity. It adds such questions as: *Which service should we update first when it's time to update, especially when there are several services to update simultaneously?* There are so many pieces now that automating the deployment of each service becomes a requirement. If there are issues in production, which service is a problem or is causing a problem? Good observability is needed to spot the issue is an imminent need when working with microservices. It seems that with monolith systems, things were simpler. So, is it worth having the additional complexity of decoupling a monolith into several microservices? Sometimes it is, especially when working with systems utilizing Kubernetes in the containers world.

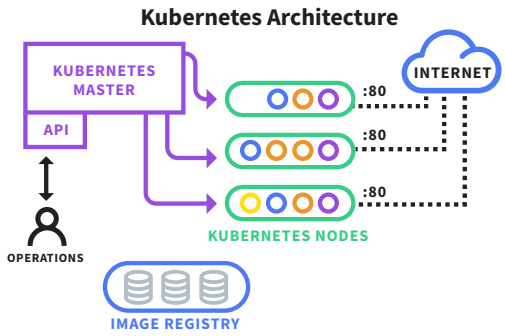
An advantage of having separate services is that it's possible to work with each microservice independently. Applying an update or patch, or even replacing an entire piece, means not having to notify or change any other parts in the system. This independence provides flexibility for developing each microservice in the appropriate programming language — even different programming languages, different frameworks, different database engines, or different operative systems. As long as there's always a way of communicating between services (like with REST, HTTP, or gRPC), all technology decisions will be reasonable.

A good approach when working with different technologies and platforms is to use containers. Containers simplify the distribution of a microservice, since the application and all its dependencies are

packaged and ready to be used anywhere, as long as the container engine is running. But then, isn't it counterproductive having to manually instantiate a container in a server cluster every time a deployment happen? Yes — and that's where orchestration for containers comes into play.

Orchestration Overview

Having to supervise the state of each different container manually, then restart it manually every time it goes down, won't scale as the volume of containers increases. A number of questions arise: Where do we deploy the container? Are there enough resources available in the cluster of servers for our needs? An orchestrator assumes the burden of working with multiple containers in a microservices architecture (or any containerized applications) in an automated fashion. Other benefits come into play — like deploying with minimum or zero downtime, the capability to automatically scale in or scale out the containers or the hosts, or even having distinct ways to separate the work of developers and operations. (More about these topics later.)



There are different orchestration flavors; the three most popular today are Kubernetes, Swarm, and DC/OS. Kubernetes has gained a lot of popularity and marketshare, mostly due to its built-in, ready-to-use features. It's an open-source project written in Go, and it's based on several years of experience that Google has with its Borg project. Kubernetes combines many of the best practices from companies that operate at scale, as well as different ideas and practices from the larger community.

In Kubernetes, every container is accessible through its DNS endpoint; its consumers don't have to worry about losing the IP address when Kubernetes reschedules a container because of a failure in the cluster or inside the container. Kubernetes will make sure all the resources from the clusters are evenly distributed when scheduling a container. And using metrics like CPU or memory, Kubernetes can also auto-scale the containers or hosts horizontally. Kubernetes rolls out new changes progressively and can roll them back if something goes wrong using healthchecks. All these features work whether the application is stateless or stateful. Kubernetes is capable of managing the state by orchestrating the storage, too; it doesn't matter if it's local or external (like a cloud service or a network storage system).

Key Characteristics of Microservices With Orchestration

Based on all the features of Kubernetes, let's take a look at the key characteristics that a microservice should have to integrate better in an orchestration ecosystem.

IDEMPOTENCE

Every time you deploy a microservice, its behavior should always be the same and should always be expected. One of the main reasons to work with containers in microservices is that you can port the application easily from one environment to another. In Kubernetes, the same portability principle applies — you can use the same commands to deploy in different environments. When a pod terminates due to a memory leak and then starts again, the pod should continue working because a new deployment will happen. Think of it as if a VM restart has happened, but much faster, with less overall impact — and it's done automatically by the orchestrator.

FAULT TOLERANCE

Assume that a microservice can terminate unexpectedly. A node in the cluster can (and will) go down, and the microservice or the system won't suffer, or it will suffer a minimal disruption with a downtime of only a few seconds. Rollouts in Kubernetes happen progressively. It starts by creating the new pod. When the new pod is ready (healthcheck pass), Kubernetes will terminate a pod from the previous version and replace it with the new one. So, containers are continually going down, and a microservice should expect that. Ideally, a microservice is stateless to increase its fault tolerance. Kubernetes will make sure that the state of the cluster is the desired one, so it will self-recover applications automatically.

SINGLE PURPOSE

A microservice should do only one thing, and do it well. In a distributed system, there are going to be common problems for all microservices — for example, transferring application logs. A library could exist to abstract the logic of managing logs, but it won't be transparent to the microservice. Every time a new version of the library comes, the microservice might need to be redeployed to use the latest version. But that breaks the rule of having a single purpose in a microservice. There are design patterns that help solve this problem, especially when working with Kubernetes where a single pod could co-schedule multiple containers. More of this is covered in the design principles section.

SERVICE DISCOVERY

When working with orchestrators like Kubernetes, service discovery comes built-in. Kubernetes has a service object, which is the way

to group pods that share the same group of labels. A consumer of a microservice won't have to keep a list of the pods to consume them; only having the service endpoint will be enough. Kubernetes will take care of registering new pods and removing those that can't provide a service. Therefore, a microservice only needs to have the endpoints of its dependencies, ideally as a configuration variable.

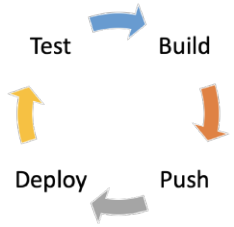
CONFIGURABLE

Remember that one of the benefits of using containers is portability. One way to make a microservice portable, while building the container only once, is to make use of configuration variables. In Kubernetes, this is called a configmap. Configmaps allow for decoupling of the configuration from the application. The same container image with Kubernetes YAML files can be used in different environments. Kubernetes translates these configurations into environment variables or as files inside a container. A microservice should always make its configuration pluggable.

Why Orchestration Matters in Microservices

It's true that microservices add more complexity, but when working with platforms like Kubernetes, much of that complexity is orchestrated away. The state of the applications is defined through YAML files. When doing a deployment of a microservice into Kubernetes, the operations team has to determine how the service will scale horizontally (at the pod or host level), or how to roll out changes in small portions (like 5 percent of the traffic first, then the rest). Kubernetes will watch the state all the time, and when there's a mismatch, it will try to make the state match in the cluster. This reconfiguration will happen even if it requires rescheduling pods or replacing a Kubernetes worker because one was malfunctioning.

Once you have defined the desired state, the aim should be to have a simple workflow like the one in the following image. What is inside the files defining the state at any point could become complicated, but it will be worth it if every deploy after development delivers the intended (and same) results every time.



At the command line, using Docker will look like this:

```
docker build -t company/microservicea:1.0 .
docker push company/microservicea:1.0
kubectl -f ./kubernetes-manifests/
```

There's a good microservices demo application developed by Google, so clone it with git. From now on, we'll be referencing this application. You can skip the step for building the application and deploy it. When needed, I'll reference portions of code or commands to run.

Let's start by deploying the application in a Kubernetes cluster. You could use the latest version of Docker for Windows or Mac, or Minikube. Once the cluster is ready to use, run the following commands:

```
kubectl apply -f ./release/kubernetes-manifests.yaml
watch kubectl get pods
```

Wait a few minutes until all pods are in the "running" state. Then, to get the IP address of the frontend service, run the following command:

```
kubectl get service/frontend-external
```

Visit the application on your browser to confirm that it's working. It's that easy.

Lastly, more and more recently, there's been a lot of noise for the service meshes in the microservices and Kubernetes ecosystem. The term "service mesh," per Istio's documentation, is "used to describe the network of microservices that make up such applications and the interactions between them." In other words, it defines how microservices can communicate. Istio is a popular open-source service mesh platform that runs on top of Kubernetes. It provides a uniform way to operate, control, and secure microservices. Kubernetes and Istio apply some design principles that the industry has been advocating for for years, so let's give them a look.

Design Principles for Microservices in an Orchestrated World

In general, when designing software, simplicity matters. Even though the system may be complex under the hood, its operation and development should be relatively simple. The aim when designing a microservice is to be able to reuse code and avoid adding code that it doesn't need. Nonetheless, there are certain design principles that have become a prerequisite to success.

OBSERVABILITY

Having a microservice that emits enough information to make it observable is key, especially when it's time to troubleshoot problems in a live environment. For that reason, the microservice code should have instrumentation to generate metrics that will help to understand the error rate failure, or spot latency problems by sending traces among all microservices involved in a request, then create a correlation. This can be as simple as including a header to identify a request that is forwarded through the entire application. A typical library for collecting and emitting telemetries like metrics and traces is OpenCensus. With OpenCensus, it's possible to export metrics

to an external service like Jaeger, Zipkin, Prometheus, and others. OpenCensus provides libraries for Go, Java, C#, Node.js, C++, Ruby, Erlang/Elixir, Python, Scala, and PHP.

In the microservices demo, each microservice uses OpenCensus to send traces in every request. It's as simple as registering an exporter like this:

```
exporter, err := jaeger.NewExporter(jaeger.Options{
    Endpoint: fmt.Sprintf("http://%s", svcAddr),
    Process: jaeger.Process{
        ServiceName: "productcatalogservice",
    },
})
if err != nil {
    log.Fatal(err)
}
trace.RegisterExporter(exporter)
```

Another important aspect is that the microservice should provide endpoints for its health status. In Kubernetes, a liveness probe can be configured to know when a pod should be restarted, perhaps due to a deadlock. And to make sure Kubernetes only sends traffic to healthy pods, there's a readiness probe that can hit an endpoint in the API and confirm that it gets an affirmative answer back. Ideally, these endpoints should include logic to verify connections to the database or run a smoke test for basic functionality.

IMMUTABILITY

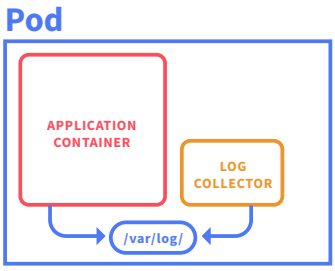
Once a container has been built, the expectation is that it won't change while running. Containers should attempt to be immutable, thus insuring that a container will be the same when it restarts. A microservice will be stateless, and if it depends on the state, it should do so by using external storage like volumes or databases. Everything that a container stores at runtime might (i.e. will) get lost.

This native immutability in containers and orchestrators helps ensure that every change is done through the desired state files definition (Dockerfile or YAML). Promoting changes through these files, even if it's to fix a bug in production, increases overall visibility for the team. No more surprises because someone forgot to apply a fix made in a test server to a production server. Not knowing what exactly fixed the problem is no longer an issue because it's now possible to compare the current state to the previous state of the desired state files.

Every time a change is required, a new version is created. This provides the option to do a rollback with a working version in case there's a problem with the new one.

SIDECAR PATTERN

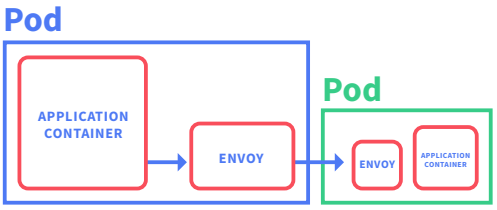
The sidecar pattern is used to deploy components of the application to extend its behavior as an isolated, decoupled and reusable container. Kubernetes will schedule this container in conjunction with the application and group them in a pod. Containers in a pod share the same network and storage, allowing you to implement a log collector (for example) as a sidecar pattern. The application container could write logs to STDOUT and STDERR, and the collector container will read the log data, apply transformations, and send it to a desired centralized location. Log management then becomes transparent to the application. If a change needs to happen regarding how to treat logs, it doesn't interfere with the application.



AMBASSADOR PATTERN

The ambassador pattern is used as a proxy for the application container. It's commonly used for applications that are difficult to upgrade. An example of this pattern is present when working with Istio. Every time a new pod has to be deployed, Kubernetes injects an Envoy container to route the pod's traffic. This container applies network security policies, circuit breakers, routes, and telemetry. The container application keeps its simplicity, and operators can adapt the proxy container as they need to.

Other patterns might be useful, but the ambassador and the sidecar patterns are the most common patterns used in an orchestrated world.



DEFINED RESOURCES

It's essential to have a clear definition of what resources a microservice will need to run effectively. In Kubernetes, it's possible to define how much memory and CPU a pod needs. When you provide this information to Kubernetes, you will get predictable behavior on how your pods will run. Furthermore, it's going to be easier for Kubernetes to determine where to schedule a pod or when to scale out/in based on telemetry.

Defining resources in a pod is as simple as this:

```
requests:
    cpu: 100m
    memory: 64Mi
limits:
    cpu: 200m    memory: 128Mi
```

If you want to know more about this topic, Google published a blog post with a video explaining how to define the resources a pod will normally need.

Monitoring in a Microservices/Kubernetes World

In distributed system architectures like microservices, having visibility from different perspectives will be critical at troubleshooting time. Many things could happen in a request when there are many parts constantly interacting at the same time. The most common method is to write logs to the `stdout` and `stderr` streams.

For example, a latency problem in the system could exist because a microservice is not responding correctly. Maybe Kubernetes is restarting the pod too frequently, or perhaps the cluster is out of capacity and can't schedule any more pods. But for this reason, tools like Istio exist; by injecting a container in every pod, you can get a pretty good baseline of telemetry. Additionally, when you add instrumentation with libraries like OpenCensus, you can deeply understand what's happening with and within each service.

All this information will need a storage location, and as a good practice, you might want to have it a centralized location to provide access to anyone in the team — not just for the operations team.

CENTRALIZED LOGGING

You need to have a centralized logging solution due to the volatile nature of containers and orchestrators. All the information that you collect might (i.e. will) get lost. Moreover, a good approach will be to have this centralized logging solution isolated from the application environment, e.g. in a different cluster of servers. You'll reuse this solution for any upcoming system, and it will be easier to triage information from various sources.

There are several storage solutions for logs, like Elasticsearch, InfluxDB, cAdvisor, and Prometheus. There are also managed solutions like Scalyr, Datadog, or New Relic.

WHAT TO MONITOR IN MICROSERVICES

Once data has been collected, it's time to keep an eye on it — while not abusing monitoring without purpose. This means that every monitor will have a clear threshold to alert, and this alert will trigger an action to solve the problem. Kubernetes will try to solve most common problems, as I've explained in previous sections, like recreating a pod if the liveness probe fails. But there is a certain group of metrics that, in a microservices world, are mandatory to monitor.

ERROR LOGS

It's important that (depending on the language of the microservice), any exception and its context are properly logged. This means that when adding instrumentation, the only reason a try/catch block will exist is if an action needs to be taken, not to hide an error from the end user. The language Go, for example, understands this problem very well by not using exceptions.

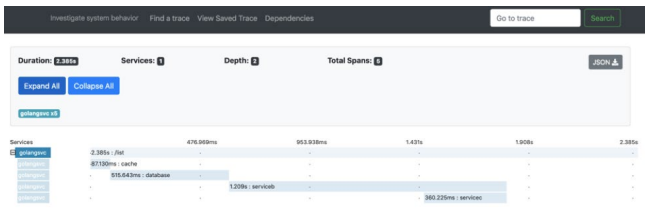
Once you successfully collect error logs, you can define alerts and actions, like: "When more than 10% of the request has an error log, notify." Initially, you might want to just notify yourself to tune the threshold and choose a definitive action.

FAILURE METRICS

There might be times when errors logs exist even if the request succeeds. And there are other times when the failure is apparent just from looking at the HTTP code in the response. For example, monitor for all 5XX error codes, because that means that there has been a failure in the microservice (like a timeout when calling a database). 4XX codes will tell you that clients are having problems either because the request is malformed due to an attack or mistake, or because there's still a call to a legacy endpoint. When using distributed tracing like OpenCensus, you can get default metrics for HTTP and gRPC services like latency, request count, and bytes received.

PERFORMANCE

Latency is a very important metric in microservices. Latency problems in one service will impact the overall request latency when chaining calls to different microservices. Every call to a microservice should record a trace, which is basically a record of how much time it took to respond. It's possible to add more details to the function level, including the action, the result, and the pass to the next service. The hard part is triaging all traces in a request from a client. Usually, a trace ID header has to be sent in every request. If there isn't one, the logging library creates it and it will represent the first trace in a request. As we saw in a previous section on microservice observability, adding traces with OpenCensus is simply a matter of including the libraries and registering an exporter. Below is a screenshot of a trace in Zipkin.



LOGGING LEVELS IN KUBERNETES

Different components in Kubernetes architecture will emit logs in different ways and locations. Let's take a look at how to treat each of the different logging levels that exist when working with Kubernetes.

APPLICATION

As I've said previously, you achieve this logging level by adding enough instrumentation to the code with libraries like OpenCensus and custom logging information (developer's responsibility). It's better if the application writes logs to the stdout and stderr location, not through HTTP to reduce network traffic.

POD

All containers in a pod write their logs by default to the `stdout` and `stderr`. In Kubernetes, you can get the logs with the command

`kubectl logs`. This command will print the logs of a container in a pod; if there's more than one container, then a container name will need to be specified.

Let's say you want the logs from a pod called nginx; you'd use this command:

```
kubectl logs nginx
```

If you want to get the logs from all the containers, use this:

```
kubectl logs nginx --all-containers=true
```

Although, I'd recommend you to use cAdvisor to get the logs in a more standard way by consuming its API.

CONTAINER

With Docker, you have the option to send logs to the `stdout` and `stderr`, too, but you also have the option to send them somewhere else by using logging drivers. You can send logs to syslog, Fluentd, AWS CloudWatch, or hosted services.

You can get logs using the Docker command:

```
docker logs nginx
```

Or you can get logs from a container using `kubectl`:

```
kubectl logs -c ruby microservice
```

You can also consume logs from cAdvisor for containers instead of using the native tools for Docker or Kubernetes. With cAdvisor, you have both a UI and an API to centralize logging.

ORCHESTRATOR

Kubernetes also logs data for itself. For containers that are part of the kube-system namespace (e.g. kube-dns or kube-proxy), you get logs in the same way as with containers or pods. If systemd is present, the rest of the logs (like the kubelet and the container runtime) are written to journald. If systemd is not present, Kubernetes writes logs to the /var/log/ directory. If you're using a managed solution from a cloud provider, these logs might not be available directly.

Managing logs in microservices is an important process; this is the data that will help the team to solve problems in the most effective way. Logs are also going to be the source of truth needed to replicate a scenario that happened in the past. Therefore, data consistency matters in this world, especially in microservices distributed architectures.

After a container has been terminated, a cleanup process might get rid of the data stored when it was running. Or a node could go down because it is no longer needed after an auto-scaling action. Reasons may vary, and that's why microservices shouldn't depend on state — they should be stateless.

How to Handle Stateful Containers

Even though microservices — especially with containers — should be stateless, specific workloads can require stateful behavior, such as a database. That's also the case when containers in a pod need to share data between them. A Kubernetes volume solves each of these problems.

Volumes in Kubernetes are merely directories that are accessible to the containers in a pod. The container sees this as a directory, but in reality, it could be a directory stored on the host or on a storage provider in the cloud.

Let's say, for example, that the Kubernetes cluster lives in Google Cloud Platform (GCP). You first need to create a persistent disk:

```
gcloud compute disks create --size=100GB --zone=us-
central1-a k8s
```

Then, you'd use the persistent disk in a pod by including this:

```
- name: data
    gcePersistentDisk:
        pdName: k8s
        fsType: ext4
```

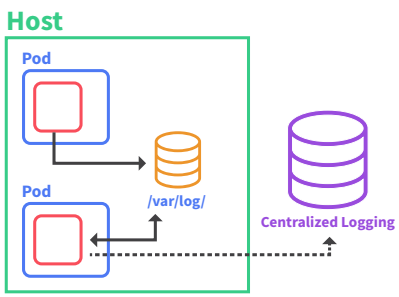
Methods for Handling Microservice Monitoring

As a general recommendation, avoid collecting information directly from microservices, as this impacts performance and availability. The goal should be to send data in the background as a detached model. And even if there are problems with the centralized logging storage or with the process of collecting data, the capabillity for each service to continue providing value shouldn't get lost. There are two ways to collect logs and metrics — one is as a daemonset and a second one is as a sidecar.

DAEMONSET

A daemonset is a user "service" that can create a pod to run in some or all of the nodes in a cluster. Every time a new node spins up, Kubernetes will create a pod based on what is defined in a daemonset. This pod could include a logging agent with the primary purpose of exposing logs or pushing them to a centralized logging location. The approach to access logs from other containers or the host is by using volumes.

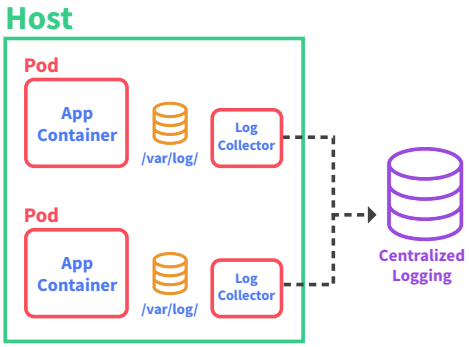
By utilizing a daemonset, you no longer need to change your application unless you need to directly change or specify a volume to write the logs.



SIDECAR

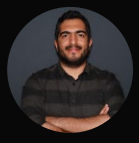
A sidecar is about having a container co-scheduled with the application container in the same pod. This new container could also include a logging agent and collect logs from a shared location between containers in the pod. It could expose the logs or push them to a centralized logging location.

It is also possible to combine a sidecar with a daemonset. The sidecar could read logs from the journald, a socket, or a file in a volume and then share logs with the daemonset by writing logs to `stdout` or `stderr` or any other volume.



Summary Impact of Orchestration on Microservices

Microservices are massively relevant. Existing tools and platforms that are being developed on top of orchestrators like Kubernetes are helping with adoption, especially in a loosely coupled way. Docker and Kubernetes help to automate the building, shipping, and running of microservices in a distributed system. Furthermore, the ecosystem of tools like Jaeger and Prometheus, as well as libraries like OpenCensus, are helping increase the quality of observability in microservices.



Written by **Christian Melendez,** *Cloud Architect at Equinix EMEA*

Christian is a technologist who started as a software developer and has more recently become a cloud architect focused on helping companies implement continuous delivery pipelines. Christian is also a technical writer for topics around Kubernetes, containers, cloud, and DevOps. He's contributed to the community and specific companies with talks and workshops, too. You can find more about what he's doing on cmelendeztech.com or on Twitter @Christianhxc.

DZone
A DEVADA MEDIA PROPERTY

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects, and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code, and more. "DZone is a developer's dream," says PC Magazine.

Devada, Inc.
600 Park Offices Drive
Suite 150
Research Triangle Park, NC

888.678.0399 919.678.0300