

2023 Edition

Cloud Native Microservices With Kubernetes

A Comprehensive Guide to Building,
Scaling, and Managing Highly-Available
Microservices in Kubernetes

Hands-On Guide

Aymen EL Amri



www.faun.dev

Cloud Native Microservices With Kubernetes

A Comprehensive Guide to Building, Scaling,
Deploying, Observing, and Managing
Highly-Available Microservices in Kubernetes

Aymen El Amri @eon01

This book is for sale at

<http://leanpub.com/cloud-native-microservices-with-kubernetes>

This version was published on 2023-06-21



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2023 Aymen El Amri - FAUN (www.faun.dev)

Contents

1 Cloud Native Microservices: How and Why	1
1.1 Common approaches	1
1.2 The twelve-factor app	1
1.2.1 I. Codebase	2
1.2.2 II. Dependencies	2
1.2.3 III. Config	3
1.2.4 IV. Backing services	3
1.2.5 V. Build, release, run	4
1.2.6 VI. Processes	5
1.2.7 VII. Port binding	6
1.2.8 VIII. Concurrency	6
1.2.9 IX. Disposability	7
1.2.10 X. Dev/prod parity	8
1.2.11 XI. Logs	8
1.2.12 XII. Admin processes	9
1.3 Microservices	9
1.3.1 Database per service	11
1.3.2 API Composition	11
1.3.3 Service instance per container	12
1.3.4 Externalized configuration	12
1.3.5 Server-side service discovery	12
1.3.6 Circuit breaker	13
1.3.7 Cloud native	13
1.4 From monolith to cloud native microservices	14
2 Requirements	15
2.1 A Development server	15

CONTENTS

2.2 Install kubectl	18
3 Kubernetes: creating a cluster	19
3.1 Creating a development Kubernetes cluster using minikube	19
3.1.1 minikube installation	20
3.1.2 minikube creating a cluster	21
3.1.3 minikube profiles	23
3.1.4 K8s dashboard on minikube	24
3.1.5 Creating a Deployment	26
3.1.6 Kubernetes events	26
3.1.7 Exposing a deployment	26
3.1.8 Deleting K8s resources	27
3.1.9 minikube addons	28
3.1.10 Using Kubectl with minikube	28
3.1.11 Deleting clusters	29
3.2 Creating a development Kubernetes cluster using Rancher	30
3.2.1 Requirements	30
3.2.2 Using Terraform to launch the cluster	30
3.2.3 Creating Kubernetes resources using Rancher UI	31
3.3 Creating an on-premises Kubernetes cluster using Rancher	33
3.3.1 Requirements before starting	34
3.3.2 Creating a cluster using Rancher server	34
3.3.3 Notes about high availability	36
3.4 Creating an on-premises Kubernetes cluster: other options	37
3.5 Managed clusters	38
3.6 Creating a managed DOK cluster using Terraform	38
4 Kubernetes architecture overview	43
4.1 Introduction	43
4.2 The Control Plane	43
4.2.1 etcd	43
4.2.2 API Server (kube-apiserver)	43
4.2.3 Controller Manager (kube-controller-manager)	44
4.2.4 Cloud Controller Manager (cloud-controller-manager)	44
4.2.5 Scheduler (kube-scheduler)	44
4.3 Worker nodes	44

CONTENTS

4.3.1 Kubelet	44
4.3.2 Container Runtime	44
4.3.3 Kube-proxy	45
4.4 Node pools	45
4.5 An overview of the architecture	46
5 Stateless and stateful microservices	48
5.1 Introduction	48
5.2 Stateless workloads	48
5.3 Stateful workloads	49
6 Deploying Stateless Microservices: Introduction	50
6.1 Requirements	50
6.2 Creating a Namespace	54
6.3 Creating the Deployment	55
6.4 Examining Pods and Deployments	59
6.5 Accessing Pods	60
6.6 Exposing a Deployment	61
6.6.1 ClusterIP Service	62
6.6.2 NodePort Service	65
6.6.3 LoadBalancer Service	68
6.6.4 Headless Service	72
6.6.5 Ingress Service	74
7 Deploying Stateful Microservices: Persisting Data in Kubernetes	83
7.1 Requirements	83
7.2 Creating a Namespace	84
7.3 Creating a ConfigMap for the PostgreSQL database	85
7.3.1 What is a ConfigMap?	85
7.3.2 ConfigMap for PostgreSQL	85
7.4 Persisting data storage on PostgreSQL	86
7.4.1 Kubernetes Volumes	86
7.4.2 VolumeClaims	86
7.4.3 StorageClass	87
7.4.4 Adding storage to PostgreSQL	87
7.4.5 Creating a Deployment for PostgreSQL	90
7.4.6 Creating a Service for PostgreSQL	92

CONTENTS

7.4.7 Creating a Deployment for our application	92
7.4.8 Creating a Service for our application	97
7.4.9 Creating an external Service for our application	97
7.4.10 Creating an Ingress for our application	98
7.5 Checking logs and making sure everything is working	99
7.6 Summary	101
8 Deploying Stateful Microservices: StatefulSets	102
8.1 What is a StatefulSet?	102
8.2 StatefulSet vs Deployment	102
8.3 Creating a StatefulSet	103
8.4 Creating a Service for the StatefulSet	105
8.5 Post deployment tasks	106
8.6 StatefulSet vs Deployment: persistent storage	107
8.7 StatefulSet vs Deployment: associated service	110
9 Microservices Patterns: Externalized Configurations	112
9.1 Storing configurations in the environment	112
9.2 Kubernetes Secrets and environment variables: why?	112
9.3 Kubernetes Secrets and environment variables: how?	113
10 Best Practices for Microservices: Health Checks	122
10.1 Health Checks	122
10.2 Liveness and Readiness probes	125
10.3 Types of probes	125
10.4 Implementing probes	126
11 Microservices Resource Management Strategies	131
11.1 Resource management and risks: from Docker to Kubernetes	131
11.2 Requests and limits	131
11.3 CPU resource units	134
11.4 Memory resource units	136
11.5 Considerations when setting resource requests and limits	139
11.6 Node reserve resources vs allocatable resources	141
11.7 Quality of Service (QoS) classes	142
11.7.1 Guaranteed	143
11.7.2 Burstable	144

CONTENTS

11.7.3 BestEffort	145
11.7.4 QoS class of a Pod	146
11.7.5 Eviction order	147
11.7.6 PriorityClass: a custom class	148
12 Autoscaling Microservices in Kubernetes: Introduction	151
12.1 Best practices for microservices scalability	151
12.1.1 Use a service registry for service discovery	151
12.1.2 Implement health checks	151
12.1.3 Designing for scalability and other best practices	152
13 Autoscaling Microservices in Kubernetes: Horizontal Autoscaling	154
13.1 Horizontal scaling	154
13.2 Horizontal Pod Autoscaler	159
13.3 Autoscaling based on custom Kubernetes metrics	164
13.4 Autoscaling based on more specific custom Kubernetes metrics	166
13.5 Using multiple metrics	169
13.6 Autoscaling based on custom non-Kubernetes metrics	170
13.7 Cluster autoscaler	172
14 Autoscaling Microservices in Kubernetes: Vertical Scaling	176
14.1 Vertical Scaling	176
14.2 Vertical Pod Autoscaler	178
14.3 VPA modes	187
14.3.1 Auto	187
14.3.2 Initial	187
14.3.3 Recreate	188
14.3.4 Off	188
14.4 VPA recommendations	188
14.4.1 VPA Limitations	191
15 Scaling Stateful Microservices: PostgreSQL as an Example	194
15.1 StatefulSets and scaling	194
15.2 Stolon: introduction	196
15.3 Stolon: installation	199
15.4 Stolon: usage	201

CONTENTS

16 Microservices Deployment Strategies: One Service Per Node	207
16.1 DaemonSet: role and use cases	207
16.2 DaemonSet: creating and managing	208
17 Microservices Deployment Strategies: Assigning Workloads to Specific Nodes	213
17.1 Assigning your workloads to specific nodes: why?	213
17.2 Taints and Tolerations	214
17.2.1 Taints and Tolerations: definition	214
17.2.2 Taints and Tolerations: example	216
17.3 nodeSelector	220
17.3.1 The simplest form of node affinity	220
17.3.2 nodeSelector: example	220
17.4 Node affinity and anti-affinity	222
17.4.1 Node affinity: like nodeSelector but with more options	223
17.4.2 Node affinity: example	223
17.4.3 Node anti-affinity: example	227
17.4.4 Affinity weight	229
17.4.5 Affinity and anti-affinity types	232
18 Kubernetes: Managing Infrastructure Upgrades and Maintenance Mode	233
18.1 Why do we need to upgrade our infrastructure?	233
18.2 What to upgrade?	233
18.3 Upgrading worker nodes: draining	234
18.4 Upgrading worker nodes: cordoning	236
18.5 Upgrading Node Pools	236
18.6 Zero downtime upgrades: Pod Disruption Budgets	237
19 Microservices Deployment Strategies: Managing Application Updates and Deployment	240
19.1 Cloud Native practices	240
19.2 Deployment strategies	240
19.2.1 Blue/Green deployment: introduction	241
19.2.2 Canary deployment: introduction	243
19.2.3 Canary deployment: an example using Istio	246
19.2.4 Canary Deployment: testing in production	258
19.3 Rolling updates: definition	260

CONTENTS

19.3.1 Rolling updates: example	262
20 Microservices Observability in a Kubernetes World: Introduction	270
20.1 Introduction to observability	270
20.2 What is monitoring?	270
20.3 What is observability?	270
20.4 White-box monitoring vs black-box monitoring	271
20.5 Pillars of observability	272
20.5.1 Logs	272
20.5.2 Metrics	272
20.5.3 Tracing	273
20.5.4 Observability pillars in action	273
20.6 Four golden signals of monitoring	274
20.6.1 Latency	274
20.6.2 Traffic	274
20.6.3 Errors	275
20.6.4 Saturation	275
20.7 Monitoring vs Observability: what's the difference?	275
21 Microservices Observability in a Kubernetes World: Prometheus, Grafana, Loki, Promtail, OpenTelemetry, and Jaeger	277
21.1 Introduction to Prometheus	277
21.2 How Prometheus works	277
21.3 Installing Prometheus	279
21.4 Accessing Prometheus web UI	280
21.5 Metrics available in Prometheus	281
21.6 Using Grafana to visualize Prometheus metrics	282
21.7 Promtail: Gathering logs from Kubernetes logs	283
21.8 Loki logging stack	284
21.9 Using Loki to query logs	289
21.10 Using Jaeger and OpenTelemetry for distributed tracing	294
22 GitOps: Cloud Native Continuous Delivery	305
22.1 GitOps: introduction and definitions	305
22.2 GitOps: benefits and drawbacks	305
22.3 GitOps: tools	306

CONTENTS

23 GitOps: Example of a GitOps workflow using Argo CD	308
23.1 Argo CD: introduction	308
23.2 Argo CD: installation and configuration	308
23.3 Argo CD: creating an application	310
23.4 Argo CD: automatic synchronization and self-healing	312
23.5 Argo CD: rollback	314
23.6 Argo CD the declarative way	315
23.7 Argo CD: configuration management	316
23.8 Argo CD: managing different environments	324
23.9 Argo CD: deployment hooks	327
24 Creating CI/CD Pipelines for Microservices	332
24.1 Continuous integration, delivery, and deployment of microservices . .	332
24.2 CI/CD tools	332
24.2.1 Jenkins	333
24.2.2 Spinnaker	333
24.2.3 Argo CD	333
24.2.4 GitHub Actions	334
24.2.5 GitLab CI/CD	334
24.3 Creating a CI/CD pipeline for a microservice	335
24.3.1 Install and configure Argo CD	337
24.3.2 Create a GitHub repository for our microservice	338
24.3.3 Create a Docker Hub account	343
24.3.4 Setting up GitHub Actions	343
24.3.5 Create a Helm chart	346
24.3.6 Create an Argo CD Application	351
24.3.7 Automating the deployment of new versions	353
25 Afterword	357
25.1 What's next?	357
25.2 Thank you	357
25.3 About the author	357
25.4 Join the community	358
25.5 Feedback	358

1 Cloud Native Microservices: How and Why

1.1 Common approaches

The field of software engineering is constantly evolving, and laws such as “Continuing Change”, “Increasing Complexity”, and “Declining Quality” are at the core of every software solution that addresses a real-world problem.

The laws of software evolution were formulated in 1974 by [Lehman](#)¹. These laws describe a balance between the forces driving new developments and those that slow down progress. They have been revised and extended several times.

Lehman’s laws of software evolution include:

- The Law of **Continuing Change**: software satisfaction decreases over time unless it is continually adapted to meet new needs.
- The Law of **Increasing Complexity**: as a large program is continuously changed, its complexity increases unless work is done to maintain or reduce it.
- The Law of **Declining Quality**: the quality of a system appears to decline unless it is rigorously maintained and adapted to operational environment changes.

Since software is often subject to change, Lehman aimed to identify the laws that govern such changes and ensure the software remains viable.

Software systems have continued to evolve, but these principles have always existed. As a result, developers have sought new approaches to manage the evolution of their systems while adhering to these principles.

Some of the most common approaches are the Twelve-factor App, microservices, and Cloud Native Computing.

¹https://en.wikipedia.org/wiki/Meir_Manny_Lehman

1.2 The twelve-factor app

This methodology is a set of best practices for building web applications or software-as-a-service. It was developed by the team at [Heroku](#)², a cloud platform provider, and has since been widely adopted by the software development community. As you can understand from its name, this methodology highlights twelve factors:

1.2.1 I. Codebase

The twelve-factor app methodology states that an app should have only one codebase which is tracked in a version control system like Git or Mercurial. The codebase should be a single repository or a set of repositories that share a root commit. Multiple codebases indicate a distributed system, where each component should be treated as a separate app. Sharing code across multiple apps violates the twelve-factor principle, and shared code should be factored into libraries that can be included through the dependency manager.

An app can have many deploys, which are running instances of the app, including production, staging, and local development environments. Although different versions may be active in each deploy, the codebase remains the same across all deploys. Developers should always have a copy of the same codebase in their local development environment.

Read more [here](#)³.

1.2.2 II. Dependencies

The 2nd factor emphasizes the need to explicitly declare and isolate dependencies. A twelve-factor app should declare all dependencies via a dependency declaration manifest and use a dependency isolation tool during execution to ensure that no implicit dependencies “leak in” from the surrounding system.

²<https://www.heroku.com/>

³<https://12factor.net/codebase>

This principle applies uniformly to both production and development environments.

Dependency declaration and isolation must always be used together to satisfy the twelve-factor. Explicit dependency declaration simplifies the setup for new developers and ensures that the app runs consistently across different environments.

Twelve-factor apps also do not rely on the implicit existence of any system tools and should vendor any necessary system tools into the app to ensure compatibility.

Read more [here⁴](#).

1.2.3 III. Config

Storing configurations in the environment rather than in the code is the third factor.

An app's configuration should be stored in environment variables, which are easy to change between deployments, language- and OS-agnostic, and not likely to be accidentally checked into the code repository.

Configurations should not be stored as constants in the code since config varies substantially across deploys, unlike the code. Internal application config, such as how code modules are connected, is best done in the code.

Grouping of configurations into named environments is discouraged, and instead, env vars should be granular controls that are independently managed for each deployment.

This model scales up smoothly as the app expands into more deployments over its lifetime.

Read more [here⁵](#).

1.2.4 IV. Backing services

The 4th factor of the 12 Factor App methodology highlights treating backing services as attached resources, accessed via a URL or other locator/credentials stored in the config.

⁴<https://12factor.net/dependencies>

⁵<https://12factor.net/config>

Backing services include any service the app consumes over the network as part of its normal operation, such as data stores (e.g MySQL), messaging/queueing systems (e.g AWS SQS), and caching systems (e.g. Redis).

The code for a twelve-factor app treats local and third-party services the same way. When swapping out a local resource with a third-party resource, only the resource handle in the configuration needs to be changed. Each separate backing service is treated as an attached resource that can be easily attached to or detached from deployments without any need to change the app's code.

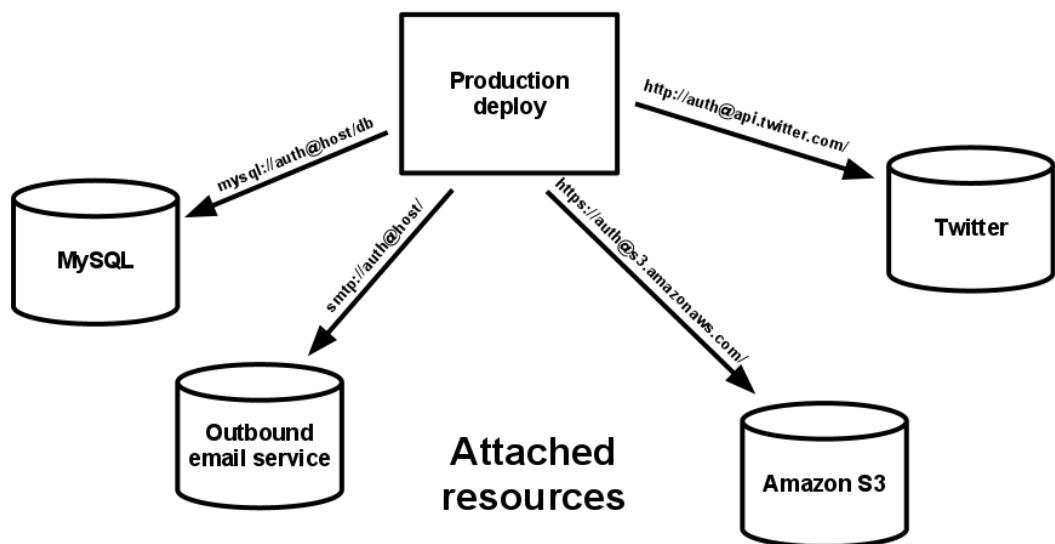


Image credit: <https://12factor.net/>

Read more [here](#)⁶.

1.2.5 V. Build, release, run

The 5th factor stresses on the importance of strictly separating the build, release, and run stages.

The build stage transforms the code repo into an executable bundle known as a build, the release stage combines the build with the deploy's current config, and the run stage launches the app in the execution environment.

⁶<https://12factor.net/backing-services>

Code cannot be altered at runtime, and each release should have a unique release ID that cannot be modified once it is created. Builds are initiated by the app's developers, while runtime execution can occur automatically. Therefore, the run stage should be kept as simple as possible to avoid breaking the app when developers are not present.

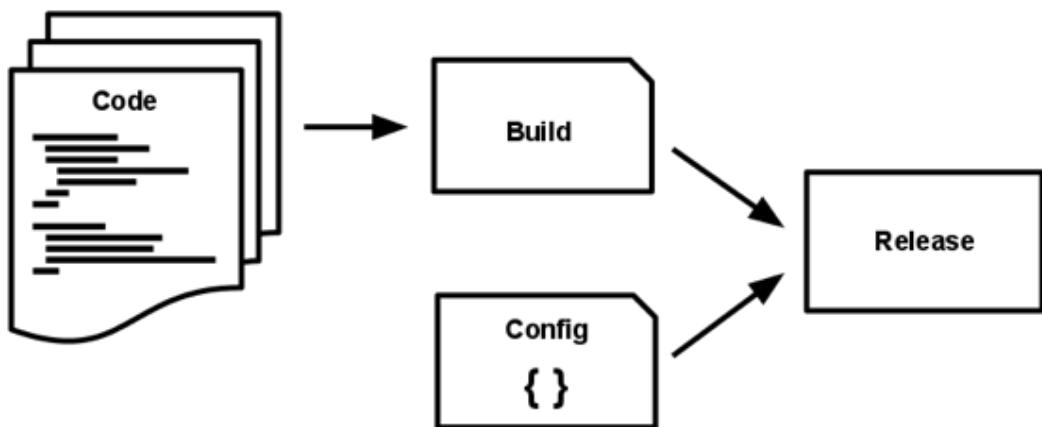


Image credit: <https://12factor.net/>

Read more [here](#)⁷.

1.2.6 VI. Processes

This factor concerns executing the app as one or more stateless processes, which are launched in the execution environment.

Twelve-factor processes are stateless and share nothing, meaning any data that needs to persist must be stored in a stateful backing service like a stateful data store.

However, the memory space or file system of a process can be used as a brief, single-transaction cache. Nevertheless, the application should not assume that anything cached will be available in the future. The use of sticky sessions, which cache user session data in memory, violates the twelve-factor methodology. In-

⁷<https://12factor.net/build-release-run>

stead, session state data should be stored in a data store that offers time expiration, such as [Memcached](#)⁸ or [Redis](#)⁹.

Read more [here](#)¹⁰.

1.2.7 VII. Port binding

The seventh factor is exporting services via port binding.

Unlike traditional web apps that rely on a web server container, a twelve-factor app should be self-contained and should not require a web server to create a web-facing service.

Instead, the software should export HTTP as a service by binding to a port and listening for incoming requests. Port binding can also be used to export other types of server software, allowing one application to act as the backing service for another.

In deployment, a routing layer handles routing requests from a public-facing hostname to the port-bound web processes.

Read more [here](#)¹¹.

1.2.8 VIII. Concurrency

In the twelve-factor app world, processes are a first-class citizen, and scale is expressed as running processes.

Developers can assign each type of work to a process type. The share-nothing, horizontally partitionable nature of these processes makes adding more concurrency a simple operation.

Processes should never daemonize or write PID files, and the operating system's process manager should be relied on to manage output streams and handle restarts and shutdowns.

⁸<https://memcached.org/>

⁹<https://redis.com/>

¹⁰<https://12factor.net/processes>

¹¹<https://12factor.net/port-binding>

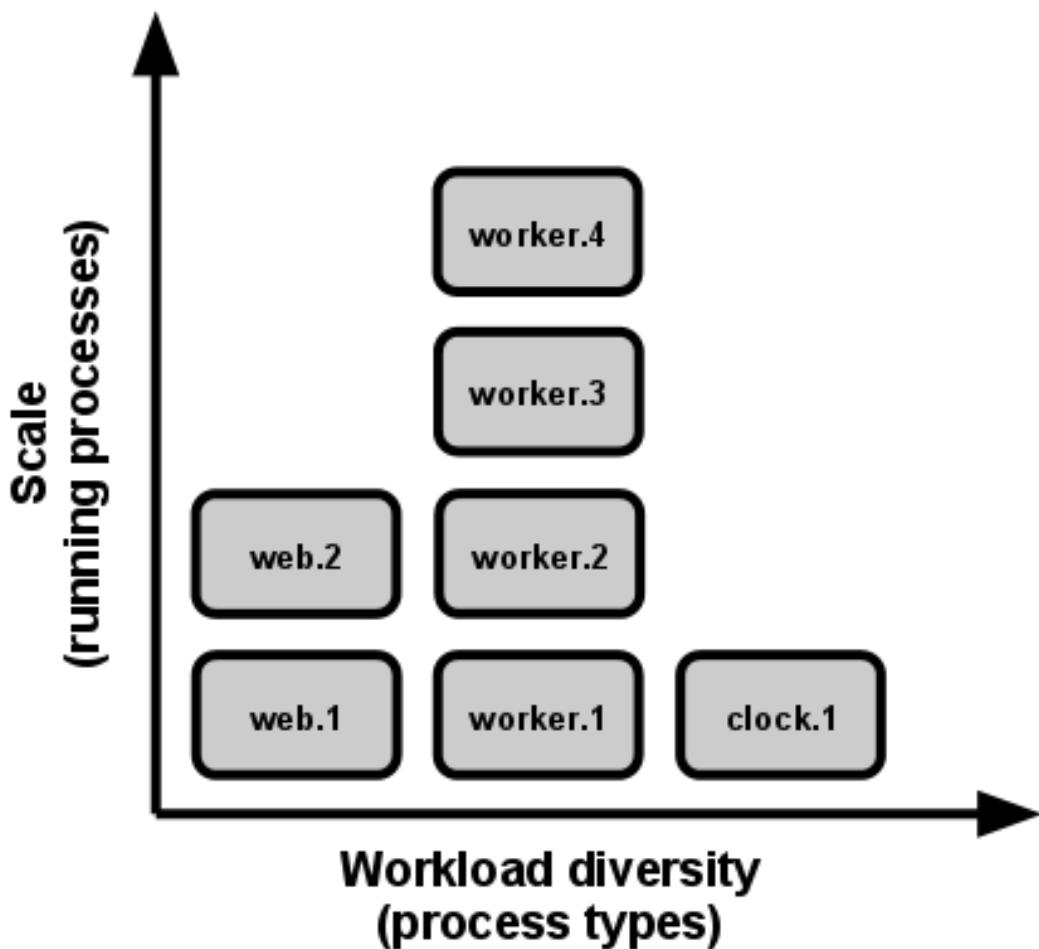


Image credit: <https://12factor.net/>

Read more [here](#)¹².

1.2.9 IX. Disposability

This is the ninth factor of the twelve-factor app and it refers to the ability of processes to be started or stopped quickly, making it possible to rapidly deploy changes, scale elastically, and ensure robustness.

¹²<https://12factor.net/concurrency>

Processes should be designed to start up quickly and shut down gracefully when receiving a SIGTERM signal. Workers should return any current jobs to the queue before exiting, and processes should be resilient against sudden failures.

Additionally, a twelve-factor app should be able to handle unexpected, non-graceful terminations.

Read more [here](#)¹³.

1.2.10 X. Dev/prod parity

The Dev/prod parity principle of the twelve-factor app strives to reduce the gaps between development and production environments, including time, personnel, and tools.

The principle involves deploying code in a matter of hours or even minutes, keeping the gap between writing and deploying code as small as possible, involving the same people in both processes and ensuring that the development and production environments are as similar as possible.

It also advises using the same type and version of backing services across all deployments.

Lightweight local services are not as compelling as they used to be, since modern packaging systems and declarative provisioning tools allow developers to run local environments that closely resemble production environments.

Read more [here](#)¹⁴.

1.2.11 XI. Logs

The twelve-factor app treats logs as event streams¹⁵ that are time-ordered and provide visibility into the behavior of a running app.

A twelve-factor app writes its event stream, unbuffered, to [STDOUT](#)¹⁶ and should never concern itself with routing or storage of its output stream.

¹³<https://12factor.net/disposability>

¹⁴<https://12factor.net/dev-prod-parity>

¹⁵https://adam.herokuapp.com/past/2011/4/1/logs_are_streams_not_files/

¹⁶https://en.wikipedia.org/wiki/Standard_streams

The execution environment captures each process' stream, collates it with all other streams from the app, and routes it to one or more final destinations for viewing and long-term archival.

Open-source log routers such as [Logplex¹⁷](#) and [Fluentd¹⁸](#) and more are available for this purpose.

Optionally, the event stream for an app can be routed to a file or sent to a log indexing, analysis, and visualization system such as [ELK¹⁹](#). This allows for powerful and flexible introspection of an app's behavior over time, including the ability to find specific events in the past, graph trends on a large scale, and set up active alerting.

Read more [here²⁰](#).

1.2.12 XII. Admin processes

The twelfth and final factor of twelve-factor apps focuses on running admin or management tasks as one-off processes. A common example of an admin process is database migration or static file collection.

These processes should be run in an identical environment to the regular long-running processes of the app, using the same codebase and configuration.

The code responsible for these admin processes must ship with the application code to avoid synchronization issues.

One-off admin processes should be run with the same dependency isolation techniques as regular processes.

The twelve-factor methodology recommends using languages that provide a [REPL²¹](#) shell out of the box, and make it easy to run one-off scripts. In production, developers can use SSH or other remote command execution mechanisms provided by the deployment environment to run one-off admin processes.

Read more [here²²](#).

¹⁷<https://devcenter.heroku.com/articles/logplex>

¹⁸<https://www.fluentd.org/>

¹⁹<https://www.elastic.co/what-is/elk-stack>

²⁰<https://12factor.net/logs>

²¹http://en.wikipedia.org/wiki/Read-eval-print_loop

²²<https://12factor.net/admin-processes>

1.3 Microservices

The principles of the twelve-factor app methodology have had a significant impact on modern software development practices. They have inspired a move towards developing applications as loosely coupled services, known as microservices.

Microservices is an architecture for building software systems. It breaks down the different components of the system into smaller, more manageable pieces called microservices. Each microservice has a single responsibility and can be developed and deployed independently of the others.

The microservices approach addresses the challenges identified in Lehman's laws of software evolution.

- The Law of Continuing Change is addressed by making it easier to adapt and update individual microservices, without having to change the entire system.
- The Law of Increasing Complexity is addressed by breaking down the system into smaller, more manageable components.
- The Law of Declining Quality is addressed by making it easier to maintain and update individual microservices, with fewer constraints on the overall system, and by providing mechanisms for monitoring and managing the system as a whole.

A microservices architecture also provides additional benefits, including scalability, flexibility, easier testing, and deployment. These advantages make it an optimal technique for building large and complex software systems.

Before the rise of the patterns of microservices, developers had to deal with monolithic systems.

Monolithic applications package all features and modules together into a single, large application. While this approach can work well for small-scale projects, it can become problematic as the application grows in size and complexity.

In other words, as the monolithic application becomes larger, it can become harder to maintain and update. Additionally, the different components of the application may start to interfere with each other, causing issues that are difficult to diagnose and fix.

On the other hand, microservices is an architectural approach to building applications in which the different components are broken down into smaller, more manageable pieces.

Each microservice has a single responsibility and can be developed and deployed independently of the others. This makes it easier to develop and maintain the application, as well as increasing its flexibility and scalability.

Microservices enable teams to work independently on different parts of an application, using the programming language, framework, tools, and data stores that they are most comfortable with. When managed correctly, this can lead to significant productivity improvements. Additionally, microservices can be developed and deployed independently, making it easier to test and deploy new features with little or no downtime.

One of the most notable documentation works around microservices is [Chris Richardson's microservices patterns](#)²³. These patterns are publically accessible on [microservices.io](#)²⁴. Some of the most common ones are:

1.3.1 Database per service

In a microservices architecture, each service should have its own private database that can only be accessed via its API. This approach ensures loose coupling between services. Since services may have different data storage requirements, they can use different types of databases.

However, implementing transactions and queries that involve multiple services can be challenging, and may require the use of patterns such as [Saga](#)²⁵, [API Composition](#)²⁶, and [CQRS](#)²⁷. The “Shared Database” anti-pattern should be avoided, as it can lead to issues when scaling and maintaining the system.

1.3.2 API Composition

This pattern is about implementing queries in a microservice architecture using an API Composer for service collaboration.

²³<https://microservices.io/patterns/index.html>

²⁴<https://microservices.io/patterns/index.html>

²⁵<https://microservices.io/patterns/data/saga.html>

²⁶<https://microservices.io/patterns/data/api-composition.html>

²⁷<https://microservices.io/patterns/data/cqrs.html>

It allows for a simple way to query data from multiple services, but can result in inefficient joins of large datasets.

This pattern is necessary due to the Database per Service pattern, and the CQRS pattern is an alternative solution.

1.3.3 Service instance per container

This pattern emphasizes the importance of using containers, such as Docker, for packaging and deploying services in a microservice architecture. The pattern involves packaging each service as a Docker image and deploying each service instance as a container.

Using containers provides benefits such as scalability, isolation, and resource constraints. However, compared to deploying virtual machines, containers have the drawback of having a less rich infrastructure.

1.3.4 Externalized configuration

The pattern involves externalizing application configuration data to enable a service to run in multiple environments without modification. This is important because different environments may have different instances of infrastructure and third-party services.

The solution involves configuring services to read the configuration from an external source during startup.

The benefits of this pattern include having an application that can run in multiple environments without the need for modification or recompilation. However, it is important to consider how to ensure that the supplied configuration matches what is expected when the application is deployed. Server-side and client-side service discovery patterns can help solve this related problem.

1.3.5 Server-side service discovery

The pattern of inter-service communication and service discovery is required in a microservice-based application.

Since the number and location of service instances can change dynamically, clients of services need a mechanism to make requests to a changing set of ephemeral service instances.

The solution is to use a server-side discovery router that queries a service registry and forwards the request to available instances.

This simplifies client code and is offered by cloud environments such as [AWS Elastic Load Balancer](#)²⁸. However, it requires additional system components and network hops.

1.3.6 Circuit breaker

This pattern is focused on handling failures of remote services in microservices architecture.

A circuit breaker acts as a proxy to prevent failures from cascading to other services. When the number of consecutive failures exceeds a certain threshold, the circuit breaker trips and all attempts to invoke the remote service fail immediately for a timeout period. If the limited number of test requests during the timeout period succeed, the circuit breaker resumes normal operation; otherwise, the timeout period begins again. This pattern can be implemented using libraries like [Netflix Hystrix](#)²⁹ (currently in maintenance mode) or any of its alternatives, such as [Istio](#)³⁰. It can be used in services designed with a Microservice Chassis, API Gateway, or Server-side discovery router.

The challenge here is to choose the right timeout values without causing false positives or excessive latency.

1.3.7 Cloud native

The microservices approach does not only require new ways of thinking and designing software systems but also requires new tools and technologies to manage deployment, coordination, monitoring, and other aspects of software engineering.

²⁸<https://aws.amazon.com/elasticloadbalancing/>

²⁹<https://github.com/Netflix/Hystrix>

³⁰<https://istio.io/>

With the rise of microservices, a new concept of cloud-native applications has emerged.

A cloud-native approach takes advantage of the benefits of cloud computing containers like Docker and orchestration tools such as Kubernetes, such as elasticity and availability, to build and operate applications that are designed to run in the cloud.

Cloud-native applications are built using containerized microservices, which allows for effortless deployment, scaling, and administration in a cloud environment.

1.4 From monolith to cloud native microservices

The shift in software architecture, from monolith to microservices to cloud-native apps, has been a phenomenal experience that has influenced the way we design and deliver applications. Microservices break down monoliths into smaller, independent components, whereas cloud-native microservices have led to a new era of designing and delivering systems on the cloud. Kubernetes, along with its ecosystem of tools and technologies, has played an important role in facilitating this change by enabling developers to swiftly manage, scale, and orchestrate their microservices-based platforms. Its robust capabilities, such as container orchestration, service discovery, and auto-scaling, have been beneficial in accelerating the development of cloud-native microservices.

This guide explores Kubernetes and its ecosystem, discusses the new challenges and opportunities it introduces, and teaches you how to use it to deploy, manage, secure, monitor, and scale cloud-native microservices.

We will also discuss some best practices and patterns for building resilient, scalable, and maintainable microservices-based systems with Kubernetes.

2 Requirements

2.1 A Development server

For the sake of simplicity, we are going to use an Ubuntu 22.04 server as our development server. If you are using another OS, only the installation instructions provided in this guide will change.

I also recommend creating a DigitalOcean account, you can use this [link³¹](#) or my [referral link³²](#) to get \$200 USD credit.

After creating your DigitalOcean account, you will need to create the Ubuntu machine manually.

You can also use Terraform using the following code:

```
1 # ssh-keygen -t rsa -b 4096 -C "mydigitaloceankey" -f ~/.ssh/mykey
2 # export DIGITALOCEAN_TOKEN=xxxxxx
3
4 terraform {
5     required_providers {
6         digitalocean = {
7             source = "digitalocean/digitalocean"
8             version = "~> 2.0"
9         }
10    }
11 }
12 #
13 variable "do_token" {
14     type = string
15 }
```

³¹<https://cloud.digitalocean.com/registrations/new>

³²<https://m.do.co/c/9e1ec82fb675>

```
16
17 variable "region" {
18   default = "<REGION>"
19 }
20
21 variable "vpc_uuid" {
22   default = "<VPC_UUID>"
23 }
24
25 data "digitalocean_project" "playground" {
26   name = "<PROJECT_NAME>"
27 }
28
29 resource "digitalocean_ssh_key" "my_ssh_key" {
30   name      = "mykey"
31   public_key = file("~/ssh/mykey.pub")
32 }
33
34 resource "digitalocean_project_resources" "playground" {
35   project = data.digitalocean_project.playground.id
36   resources = [for droplet in digitalocean_droplet.mydroplets : droplet\
37     .urn]
38 }
39
40 // Define a list of names for the droplets
41 variable "names" {
42   default = [
43     "<NAME_OF_DROPLET>",
44   ]
45 }
46
47 // Use a for_each loop to create a droplet for each name in the list
48 resource "digitalocean_droplet" "mydroplets" {
49   for_each = { for name in var.names : name => name }
50
51   image      = "ubuntu-22-04-x64"
```

```
52   name      = each.value
53   region    = var.region
54   size       = "s-1vcpu-1gb"
55   ssh_keys   = [digitalocean_ssh_key.my_ssh_key.id]
56   monitoring = false
57   vpc_uuid  = var.vpc_uuid
58   provisioner "remote-exec" {
59     inline = [
60       "sudo apt-get update",
61       "sudo apt-get install -y net-tools",
62     ]
63   }
64 }
65
66 // Use a for_each loop to output the IP address of each droplet
67 output "droplet_ip_addresses" {
68   value = {
69     for name, droplet in digitalocean_droplet.mydroplets : name => drop\
70     let.ipv4_address
71   }
72 }
```

Before starting Terraform, you will need:

- At least 1 project on your DigitalOcean account.
- A token that you generated in the DigitalOcean dashboard.

Make sure to change the values in the Terraform HCL file with your values (e.g: REGION, PROJECT_NAME..etc).

If you are not using Windows, make sure to update the directory where your SSH key is generated (~/.ssh/mykey). You will also need an SSH client such as PuTTY or the built-in SSH client.

Then execute:

```
1 ssh-keygen -t rsa -b 4096 -C "mydigitaloceankey" -f ~/.ssh/mykey
2 export DIGITALOCEAN_TOKEN=xxxxxx
3 terraform init
4 terraform plan
5 terraform apply
```

2.2 Install kubectl

kubectl is a command-line tool that is widely used in the management and deployment of applications within Kubernetes clusters. It is designed to make it easy for developers, operators, and administrators to interact with Kubernetes clusters, and comes with a wide range of features.

To communicate with the Kubernetes API server and manage our Kubernetes clusters and resources, we need to install this CLI.

```
1 curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/rele\
2 ase/stable.txt)/bin/linux/amd64/kubectl"
3 sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Activate the auto-completion feature:

```
1 kubectl completion bash | sudo tee /etc/bash_completion.d/kubectl > /de\
2 v/null
3 sudo chmod a+r /etc/bash_completion.d/kubectl
4 source ~/.bashrc
```

3 Kubernetes: creating a cluster

3.1 Creating a development Kubernetes cluster using minikube

minikube is a tool that enables you to have a running Kubernetes cluster in your local machine in a matter of minutes. This solution has its limitations:

- Scalability: Minikube is designed for testing and development purposes and is not recommended for production environments with high traffic.
- Limited Resource Allocation: Since Minikube is designed to run on a single node, resource allocation is limited by the resources available on that node.
- Limited Functionality: Minikube is designed to provide a lightweight and simplified Kubernetes environment, which means that some advanced features of Kubernetes may not be available. Example: Load balancer services.
- Limited Storage Options: Minikube only supports a limited number of storage options and may not be suitable for applications with complex storage requirements.
- Limited Networking Options: The tool provides a simplified networking environment and may not be suitable for applications with complex networking requirements.
- Limited Security: Since it is designed for testing/development purposes it does not provide the same level of security as a production Kubernetes environment.

However, it is a good way to start experimenting with Kubernetes.

You will need a machine with:

- 2 CPUs or more
- 2GB of free memory

- 20GB of free disk space
- A provisioner such as Docker³³, QEMU³⁴, Hyperkit³⁵, Hyper-V³⁶, KVM³⁷, Parallels³⁸, Podman³⁹, VirtualBox⁴⁰, or VMware Fusion/Workstation⁴¹

3.1.1 minikube installation

In the following example, we are going to use Docker, this means that the different components of our Kubernetes cluster will be running inside containers.

We can install minikube by downloading its binary:

```
1 curl -LO https://storage.googleapis.com/minikube/releases/latest/miniku\
2 be-linux-amd64
3 sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Since we are going to use Docker, we need to install it:

```
1 sudo apt-get remove -y docker docker-engine docker.io containerd runc
2
3 sudo apt-get update
4
5 sudo apt-get install -y ca-certificates curl gnupg
6
7 sudo install -m 0755 -d /etc/apt/keyrings
8
9 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --de\
10 armor -o /etc/apt/keyrings/docker.gpg
11
```

³³<https://minikube.sigs.k8s.io/docs/drivers/docker/>

³⁴<https://minikube.sigs.k8s.io/docs/drivers/qemu/>

³⁵<https://minikube.sigs.k8s.io/docs/drivers/hyperkit/>

³⁶<https://minikube.sigs.k8s.io/docs/drivers/hyperv/>

³⁷<https://minikube.sigs.k8s.io/docs/drivers/kvm2/>

³⁸<https://minikube.sigs.k8s.io/docs/drivers/parallels/>

³⁹<https://minikube.sigs.k8s.io/docs/drivers/podman/>

⁴⁰<https://minikube.sigs.k8s.io/docs/drivers/virtualbox/>

⁴¹<https://minikube.sigs.k8s.io/docs/drivers/vmware/>

```
12 sudo chmod a+r /etc/apt/keyrings/docker.gpg
13
14 echo \
15   "deb [arch="$(dpkg --print-architecture)" signed-by=/etc/apt/keyrings\
16 /docker.gpg] https://download.docker.com/linux/ubuntu \
17   "$(. /etc/os-release && echo "$VERSION_CODENAME")" stable" | \
18   sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
19
20 sudo apt-get update
21
22 sudo apt-get install -y docker-ce docker-ce-cli containerd.io docker-bu\
23 ildx-plugin docker-compose-plugin
24
25 sudo service docker start
26
27 sudo groupadd docker
28
29 sudo usermod -aG docker $USER
30
31 newgrp docker
```

Now, we can start a cluster, but before that, since the Docker driver (provisioned) should not be used with root privileges, we are going to create a user that is not root and add it to the docker group:

```
1 adduser user
2 usermod -aG sudo user
3 su - user
4 sudo groupadd docker
5 sudo usermod -aG docker $USER
6 newgrp docker
```

3.1.2 minikube creating a cluster

Then start our first cluster:

```
1 minikube start --driver=docker
```

Now execute the following command to make sure it works:

```
1 minikube kubectl -- get pods -A
```

kubectl is the command-line tool used to interact with Kubernetes clusters. It allows users to deploy, inspect, and manage applications and resources within a Kubernetes cluster.

We use the command as follows with regular remote Kubernetes clusters:

```
1 kubectl get pods -A
```

But with minikube we always start with minikube command, followed by kubectl -- <options>.

It is also possible to start a minikube cluster with different configuration such as memory, which limits the memory usage of the cluster:

```
1 minikube start --driver=docker --memory=<memory>
```

or:

```
1 minikube start --driver=docker --extra-config=kubeadm.ignore-preflight-\
2 errors=NumCPU --force --cpus=1
```

or:

```
1 minikube start --driver=docker --memory=1977mb --cpus=2 --disk-size=20g\
2 --kubernetes-version=v1.25.0
```

Let's delete this cluster:

```
1 minikube delete
```

3.1.3 minikube profiles

Then let's start another cluster with a profile name. You can create multiple clusters locally, so having different profiles could be helpful:

```
1 minikube start --driver=docker -p c1
2 minikube start --driver=docker -p c2
3 minikube profile list
```

We now have 2 clusters: c1 and c2.

In order to switch the context to the first one, we execute:

```
1 minikube profile c1
```

Now, all our kubectl commands will be executed on the first cluster:

```
1 minikube kubectl -- get pods -A
2 minikube kubectl -- cluster-info
```

You can check the logs of the cluster using:

```
1 minikube logs
```

Or redirect the logs to a file:

```
1 minikube logs --file=logs.txt
```

Let's try some other commands:

```
1 # Get pods in all namespaces
2 minikube kubectl -- get pods -A
3 # Get nodes
4 minikube kubectl -- get nodes
5 # get pods in default namespace
6 minikube kubectl -- get pods
7 # get pods in default namespace with wide output
8 minikube kubectl -- get pods -o wide
9 # get pods in all namespaces with wide output
10 minikube kubectl -- get pods -o wide --all-namespaces
11 # get pods in all namespaces with wide output and watch
12 minikube kubectl -- get pods -o wide --all-namespaces --watch
13 # get pods in all namespaces with wide output and sort by name
14 minikube kubectl -- get pods -o wide --all-namespaces --sort-by=.meta\d\
15 ta.name
16 # get pods in all namespaces with wide output and sort by name and filt\
17 er by status
18 minikube kubectl -- get pods -o wide --all-namespaces --sort-by=.meta\d\
19 ta.name --field-selector=status.phase=Running
```

3.1.4 K8s dashboard on minikube

Kubernetes has a dashboard that we can use with minikube too. This is done using the following command:

```
1 minikube dashboard
```

After executing it, you should be able to see a similar output:

```
1 [ .. ]
2 [ .. ]
3 [ .. ]
4 🎉 Opening http://127.0.0.1:44485/api/v1/namespaces/kubernetes-dashboard\s
5 d/services/http:kubernetes-dashboard:/proxy/ in your default browser...
6 👉 http://127.0.0.1:44485/api/v1/namespaces/kubernetes-dashboard/services\
7 http:kubernetes-dashboard:/proxy/
8 [ .. ]
9 [ .. ]
```

where 127.0.0.1:44485 is the address and port to access the dashboard.

Since we are running a remote Ubuntu VM, one solution we can do is create an SSH tunnel from your local machine to access the remote server. The tunnel forwards any traffic from a local port that you define to port 44485 of the remote server:

```
1 export IP=<REMOTE_IP_ADDRESS>
2 ssh -NfL 44485:127.0.0.1:44485 root@$IP
```

If the dashboard was exposed on a different port, adapt the command by changing 44485 to the right port number.

Once the tunnel is created, open your browser, then go to the URL printed by the command `minikube dashboard`. It should look like this:

```
1 http://127.0.0.1:37925/api/v1/namespaces/kubernetes-dashboard/services\
2 http:kubernetes-dashboard:/proxy/
```

You can now view all the resources (services, pods, etc) that the cluster has.

For example, to see all pods in the system namespace, switch to “kube-system” namespace, then choose “Pods” from the “Workloads” sidebar.

Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created
coredns-787d4945fb-vh4mq	registry.k8s.io/coredns/coredns:v1.9.3	k8s-app: kube-dns pod-template-hash: 787d4945fb dd5	c1	Running	0	-	-	54 minutes ago
kube-proxy-n96kj	registry.k8s.io/kube-proxy:v1.26.3	controller-revision-hash: 5cbfd0d k8s-app: kube-proxy pod-template-generation: 1	c1	Running	0	-	-	54 minutes ago
storage-provisioner	gcr.io/k8s-minikube/storage-provisioner:v8	add-onmanager:kubernetes.io/mode:Reconcile integration-test: storage-provisioner	c1	Running	1	-	-	54 minutes ago
etcd-c1	registry.k8s.io/etcd:3.5.6-0	component: etcd tier: control-plane	c1	Running	0	-	-	54 minutes ago
kube-apiserver-c1	registry.k8s.io/kube-apiserver:v1.26.3	component: kube-apiserver tier: control-plane	c1	Running	0	-	-	54 minutes ago
kube-controller-manager-c1	registry.k8s.io/kube-controller-manager:v1.26.3	component: kube-controller-manager tier: control-plane	c1	Running	0	-	-	54 minutes ago
kube-scheduler-c1	registry.k8s.io/kube-scheduler:v1.26.3	component: kube-scheduler tier: control-plane	c1	Running	0	-	-	54 minutes ago

3.1.5 Creating a Deployment

Let's try creating a deployment using the command line:

```
1 minikube kubectl -- create deployment hello-node --image=registry.k8s.io/echoserver:1.4
```

After deploying the “hello-node” deployment, we can check the list of deployments and pods using:

```
1 minikube kubectl -- get deployments
2 minikube kubectl -- get pods
```

3.1.6 Kubernetes events

It is also possible to see the list of events to understand how Kubernetes did step by step to find and deploy a pod using the image “registry.k8s.io/echoserver:1.4”.

```
1 minikube kubectl -- get events
```

3.1.7 Exposing a deployment

In order to access the pods from the internet, we need to expose the deployment using the following command:

```
1 minikube kubectl -- expose deployment hello-node --type=LoadBalancer --\n2 port=8080
```

This will create a service using port 8080:

```
1 minikube kubectl -- get services\n2 # or\n3 minikube service hello-node
```

The output:

NAMESPACE	NAME	TARGET PORT	URL
default	hello-node	8080	http://192.168.49.2:31914

To access the application using its URL from our browser, we need to create a tunnel:

```
1 ssh -NfL 31914:192.168.49.2:31914 root@$IP
```

You should change “31914” to the right port used by your cluster.

The web page should now be accessible using the following address:

```
1 http://127.0.0.1:31914
```

Here too, you should change “31914” to the right port used by your cluster.

3.1.8 Deleting K8s resources

To clean up, delete the service and then the deployment:

```
1 minikube kubectl -- delete service hello-node  
2 minikube kubectl -- delete deployment hello-node
```

3.1.9 minikube addons

minikube has a system of addons that enables installing additional software on your cluster.

You can view the list of these add-ons using the following command:

```
1 minikube addons list
```

To enable/disable an addon, you can use:

```
1 minikube addons enable <ADDON_NAME>  
2 minikube addons disable <ADDON_NAME>
```

For example, we can enable the `metrics-server`⁴²:

```
1 minikube addons enable metrics-server
```

Test if the installation has succeeded:

```
1 minikube -p c1 kubectl -- get deploy,svc -n kube-system | egrep metrics\  
2 -server
```

Then show some metrics using the following commands:

```
1 minikube kubectl -- top nodes  
2 minikube kubectl -- top pods
```

3.1.10 Using Kubectl with minikube

If you prefer using kubectl without calling `minikube kubectl` command, you will need to install it first:

⁴²<https://github.com/kubernetes-sigs/metrics-server>

```
1 curl -LO "https://dl.k8s.io/release/$(curl -L -s https://dl.k8s.io/rele\\
2 ase/stable.txt)/bin/linux/amd64/kubectl"
3 sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Activate the completion:

```
1 kubectl completion bash | sudo tee /etc/bash_completion.d/kubectl > /de\\
2 v/null
3 sudo chmod a+r /etc/bash_completion.d/kubectl
4 source ~/.bashrc
```

Now you can see the contexts using:

```
1 kubectl config get-contexts
```

The context should be set to “c1”, which is the cluster we are using. If not, use the following command:

```
1 kubectl config set-context c1
```

Then you can execute regular kubectl commands such as:

```
1 kubectl get pods -A
2 kubectl get nodes
3 kubectl get pods
4 kubectl get pods -o wide
5 kubectl -n kube-system get pods -o wide
```

3.1.11 Deleting clusters

In order to delete a cluster created using minikube, all you have to do is delete the profile:

```
1 minikube delete -p c1 # or minikube delete
2 minikube delete -p c2
```

3.2 Creating a development Kubernetes cluster using Rancher

3.2.1 Requirements

The following instructions were tested using VMs on DigitalOcean provisioned with Ubuntu 22.04.

The fact that we are using DigitalOcean, does not mean that the same instructions will not work on other VM types.

3.2.2 Using Terraform to launch the cluster

We are going to launch a Rancher Server and a testing cluster.

There are multiple ways to do that, but the fastest and easiest one is using Terraform. Proceed with the following steps:

```
1 git clone https://github.com/rancher/quickstart
2 cd quickstart/rancher/do
3 mv terraform.tfvars.example terraform.tfvars
```

Now edit the “terraform.tfvars” file and update the following variables:

- `do_token`: Your DigitalOcean account access key.
- `rancher_server_admin_password`: The administrator password for the Rancher Server.
- `do_region`: The DigitalOcean region you want to use.
- `prefix`: A prefix to use for all created resources.
- `droplet_size`:

- The droplet size used. The minimum is s-2vcpu-4gb
- s-4vcpu-8gb could be used for more performance.

Now, run `terraform init`, then `terraform apply --auto-approve`. The latest command will create two Kubernetes clusters, one is running Rancher Server, and the other one is created for experimentation deployments.

The output of the command should show the following:

- `rancher_node_ip`: The IP address of the created node (Rancher Server).
- `rancher_server_url`: The dashboard URL accessible using sslip.io⁴³
- `workload_node_ip`: The IP address of the created node (Rancher)

Use the `rancher_server_url` to access the dashboard of Rancher Server. The login is “admin” while the password is what you defined for `rancher_server_admin_password` in Terraform file “`terraform.tfvars`”.

If you want to ssh to the Rancher Server, you can use the “`id_rsa`” key generated automatically in “`quickstart/rancher/do`”.

3.2.3 Creating Kubernetes resources using Rancher UI

Let’s test creating some Kubernetes resources using the experimentation cluster.

To begin, navigate to the Rancher dashboard and select the “quickstart-do-custom” cluster from the homepage. From there, click on “Workloads” in the sidebar, then select “Deployments” and click on “Create”.

⁴³<https://sslip.io/>

On the “default” namespace, create a deployment with the name webserver-deployment with 3 replicas.

Use nginx as a container name and nginx:latest as the container image.

Add a port without a service, name it http, and use the private port 80.

Deployment: Create

Namespace *
default

Name *
webserver-deployment

Description
Any text you want that better describes this resource

Replicas *
3

Deployment Pod nginx + Add Container

General

Health Check

Resources

Security Context

Storage

General

Container Name: nginx

Init Container Standard Container

Image

Container Image: nginx:latest

Pull Policy: Always

Ports

Service Type: Do not create a service

Name: http

Private Container Port: 80

Protocol: TCP

Add Host Remove

Add Port

Command

Arguments

Cancel Edit as YAML Create

Click on “Pod” and add the label app with the value webserver. This is going to be useful, since we are going to use it as a selector in the next step.

Keep everything else without changes, then click on “Create”. Wait for the 3 replicas to be created.

The web server is not accessible publically, we need to create a ClusterIP service for the deployment then an ingress resource to make this happen.

Click on “Service Discovery” in the sidebar, click on “Services”, then “Create”. Choose “Cluster IP”.

Create a Cluster IP service in the default namespace, call it nginx-clusterip.

Add a port named http, listening on port 8000 and having 80 as a target port (the port used in deployment configuration).

Go to “Selectors”, and a key `app` with the value `webserver` , this will include the pods created by the previous deployment having the same label.

Create the service.

Now, click on “Ingresses” under “Service Discovery”, name it `ingress` . We can use sslip.io⁴⁴ to create a subdomain to use it on “Request Host”. For example, if the node where our Kubernetes cluster is deployed (namely “quickstart-quickstart-node”) has the following IP:

- 165.22.82.137

We can use the following name:

- 165.22.82.137.sslip.io⁴⁵

Adapt the above to your IP address.

Add the “Target Service”, “Path” and “Port” to point to the previously created Cluster IP service/port.

The screenshot shows the Kubernetes Ingress configuration interface. At the top, it displays the Ingress details: Namespace: default, Name: ingress, and Active status. Below this, there are fields for Request Host (set to 165.22.82.137.sslip.io) and a Description (left empty). The main area is titled 'Rules' and contains a single rule entry. This rule has a Request Host of 165.22.82.137.sslip.io, a Path of / (selected as Prefix), a Target Service of nginx-clusterip, and a Port of 8000. Buttons for 'Add Path' and 'Add Rule' are visible at the bottom of the rules section.

Don't forget to choose the “Ingress Class”. Use `nginx`.

Click on “Create”.

Visit `<YOUR_IP>[.sslip.io]`(<http://165.22.82.137.sslip.io/>) to see access the ClusterIP service through the Ingress. You should see the “Welcome to nginx!” web page.

⁴⁴<http://sslip.io>

⁴⁵<http://165.22.82.137.sslip.io>

3.3 Creating an on-premises Kubernetes cluster using Rancher

3.3.1 Requirements before starting

The following instructions were tested using VMs on DigitalOcean provisioned with Ubuntu 22.04.

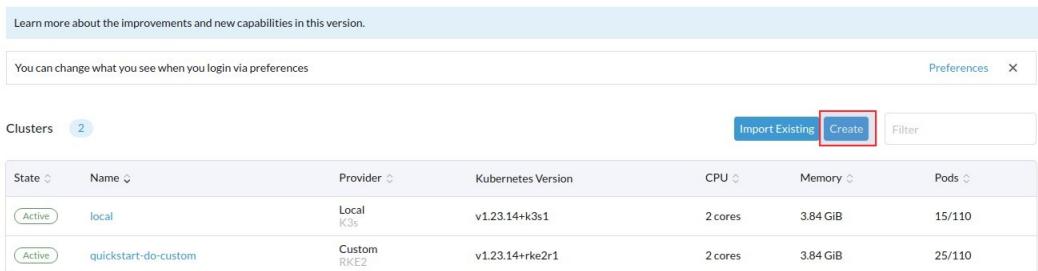
Using the Rancher Server and multiple nodes (VMs), we are going to create a Kubernetes cluster.

We have 3 machines:

- A Control plane. The hostname we are using for this machine is “control-plane”.
- Worker 1. The hostname we are using here is “worker1”
- Worker 2. The hostname we are using here is “worker2”. This is optional but in case you want two nodes in your Kubernetes cluster, add it.
- Worker n: You can add as many workers as you need for your cluster.

3.3.2 Creating a cluster using Rancher server

Now go to the Rancher Server web UI and click on “Create” to create a new cluster.



The screenshot shows the Rancher Server web interface. At the top, there is a message: "Learn more about the improvements and new capabilities in this version." Below this is a navigation bar with a "Preferences" link and a close button. The main area is titled "Clusters" and shows a table with two entries:

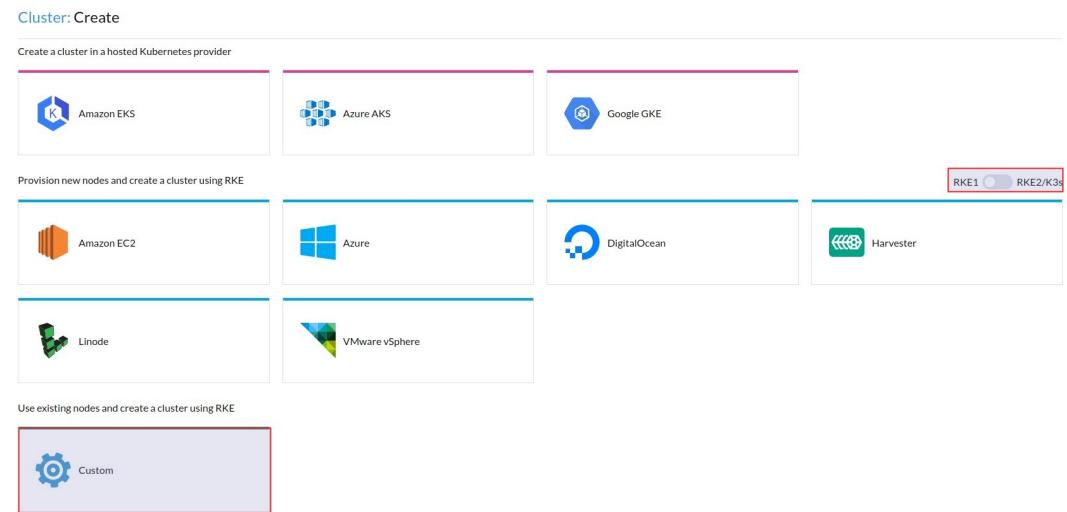
State	Name	Provider	Kubernetes Version	CPU	Memory	Pods
Active	local	Local K3s	v1.23.14+k3s1	2 cores	3.84 GiB	15/110
Active	quickstart-do-custom	Custom RKE2	v1.23.14+rke2r1	2 cores	3.84 GiB	25/110

At the top right of the table, there are buttons for "Import Existing", "Create" (which is highlighted with a red box), and "Filter".

When you start creating a cluster using Rancher, you have multiple choices depending on the infrastructure you use. For example, you can deploy an Amazon

EKS cluster if you are using AWS or a Google GKE cluster if you are using Google Cloud.

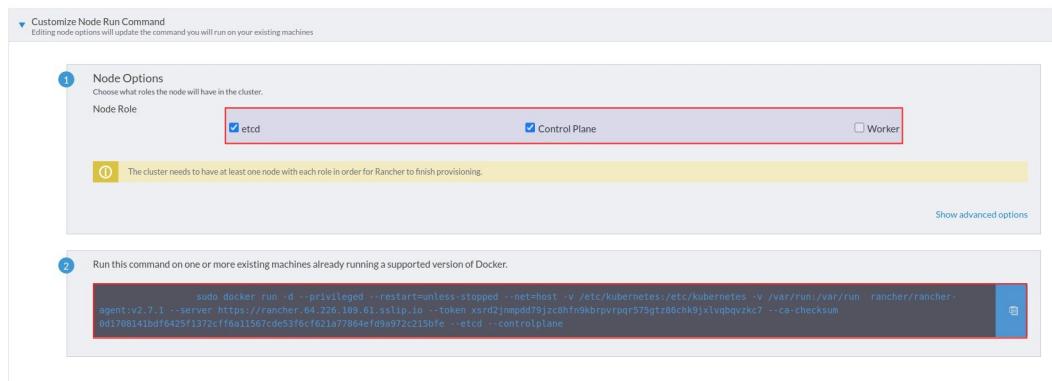
We are using DigitalOcean, so it is possible to select the Rancher provisioner for this cloud and Rancher will require your DigitalOcean API keys to create the cluster for you, including creating the nodes and deploying Kubernetes components. However, our goal is to deploy an on-premises cluster, so we are not going to use this option. Instead, we are going to use the “Custom” provisioned, which should be the option to choose when creating an on-premise cluster.



In this case, you need to provide Rancher with the VMs as it will not create them for you.

Choose **RKE1**, click on “Custom”, and give the cluster a name, for example, “mycluster”. We are going to use “v1.24.10-rancher4-1” as the Kubernetes version, change this value and keep everything else as is.

We are going to start by deploying the control plane node, so choose the roles “etcd” and “Control Plane”, then on the “controlplane” node execute the Docker agent provided by Rancher.



Next, deselect “etcd” and “Control Plane”, select “Worker” and execute the Docker agent on every worker node you want to use. You should see a confirmation of the node registration on the web UI.

Rancher will create the different components of Kubernetes such as the API server on the control plane and the kubelet on the worker nodes. This should take some time before the cluster is ready.

RKE (Rancher Kubernetes Engine) is a CNCF-certified Kubernetes distribution that runs entirely within Docker containers. If you go to “controlplane” node, and type `watch docker ps` you should be able to see etcd and the other components running inside Docker containers.

If for some reason, you encounter a problem, you can clean your nodes using [this script⁴⁶](#) or [this one⁴⁷](#).

3.3.3 Notes about high availability

The setup as we did is not considered the best for high availability. For example, it is recommended to run the etcd database on more than 1 server. You can either choose to run [a stacked topology or an external etcd topology⁴⁸](#).

In either case, the worker role should not be used or added to nodes with the etcd or controlplane role.

⁴⁶<https://github.com/rancherlabs/support-tools/blob/master/extended-rancher-2-cleanup/extended-cleanup-rancher2.sh>

⁴⁷<https://gist.github.com/Illerayo/1bef407602208911e86f42d5d208c1fb>

⁴⁸<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/>

In addition, the etcd database requires an odd number of nodes. Therefore, it is recommended to use three nodes to prevent split brain. To keep your cluster running, the number of nodes you can lose depends on how many nodes are assigned the etcd role. So it is recommended to create etcd nodes in 3 different availability zones.

You will also need to set up a load balancer to prevent any single node's outage from taking down communications to the Rancher management server.

To ensure high availability of the master component, you should have at least two nodes with the role of controlplane. It is recommended to have these nodes different availability zones.

To enable workload rescheduling in the event of node failure, you need at least two nodes with the worker role. The more nodes you add, the more your workloads become tolerant to failure.

More information are available [in the official documentation⁴⁹](#) of Rancher.

3.4 Creating an on-premises Kubernetes cluster: other options

We used Rancher to start a cluster, but there are multiple other tools.

There are many tools available for creating an on-premises Kubernetes cluster, including:

- kubeadm: This is a tool provided by Kubernetes that automates the creation of a cluster on a set of nodes. It is a good choice if you are familiar with Kubernetes and want to have more control over the creation process.
- kops: This is a tool provided by Kubernetes that automates the creation of a cluster on AWS. It is a good choice if you want to use AWS and need a tool specifically designed for that environment.
- OpenShift: This is a Kubernetes distribution provided by Red Hat that includes additional features and tools for managing Kubernetes clusters. It is a good choice if you want a more enterprise-focused solution.

⁴⁹<https://ranchermanager.docs.rancher.com/how-to-guides/new-user-guides/infrastructure-setup/ha-rke1-kubernetes-cluster>

- Kubespray: If you need more flexibility and customizations. It is also a good choice if you are familiar with Ansible since the latter is heavily used for the deployment.

3.5 Managed clusters

Managed Kubernetes clusters, such as Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), and DigitalOcean Kubernetes (DOK), provide a simpler method of running Kubernetes clusters without any operational overhead. With these managed services, developers can concentrate on their application workloads while relying on the platform to handle cluster management and scaling. This approach lets developers focus on their core responsibilities and achieve efficient results without worrying about the underlying infrastructure. However, this may come with significant costs.

These services offer features like automatic scaling, load balancing, and monitoring, and can be accessed through a web-based console or command-line interface.

For example, GKE provides a fully managed Kubernetes environment with features such as automatic scaling of nodes, automatic upgrades, and integration with Google Cloud services such as Cloud Monitoring.

EKS also provides a managed Kubernetes environment that integrates with Amazon Web Services (AWS) services.

AKS offers a managed Kubernetes service with built-in monitoring, log analytics, and support for hybrid deployments across on-premises and cloud environments.

DOK is the managed Kubernetes service of DigitalOcean and it has flexible pricing plans and a user-friendly interface.

Creating a managed cluster is relatively easy and can be done using a few clicks.

3.6 Creating a managed DOK cluster using Terraform

We are going to use DigitalOcean as a cloud provider to create a managed Kubernetes cluster. We have different options to create a cluster:

- Using the web interface of the cloud provider
- Using the command line interface of DigitalOcean, “`doctl`⁵⁰”
- Using Terraform

We are going to use the third option. This is why you will need to create the following “main.tf” file:

```
1 cat << EOF > main.tf
2 // token automatically sourced from env var
3 // execute: export DIGITALOCEAN_TOKEN=dop_v1_xxxxxxxxxxxxxxxxxxxxxxx\
4 xxxxxx
5
6 terraform {
7   required_providers {
8     digitalocean = {
9       source = "digitalocean/digitalocean"
10      version = "~> 2.0"
11    }
12  }
13 }
14
15 variable "region" {
16   default = "fra1"
17 }
18
19 data "digitalocean_kubernetes_versions" "versions" {
20   version_prefix = "1.26."
21 }
22
23 // Define the names of the clusters you want to create
24 // To create a single cluster, define a single name
25 // To create two clusters define two names
26 // ..etc
27 variable "names" {
```

⁵⁰<https://docs.digitalocean.com/reference/doctl/>

```
28     default = [
29         "my_cluster",
30         # "my_other_cluster"
31     ]
32 }
33
34 // create a cluster for each name in the list
35 resource "digitalocean_kubernetes_cluster" "cluster" {
36     for_each = { for name in var.names : name => name }
37
38     name = each.value
39     region = var.region
40     version = data.digitalocean_kubernetes_versions.versions.latest_ver\
41 sion
42
43     node_pool {
44         name = "default"
45         size = "s-2vcpu-2gb"
46         auto_scale = true
47         min_nodes = 1
48         max_nodes = 3
49     }
50 }
51 EOF
```

Change the name of the cluster and the node pool to whatever names you want to use. If you want to create multiple clusters, add more names to the variable names.

Create an API token, then execute the following commands to export the token and create the cluster:

```
1 export DIGITALOCEAN_TOKEN=dop_v1_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
2 terraform init
3 terraform plan
4 terraform apply
```

You can now download the “kubeconfig” file from the cluster as it’s shown in the following image:

The screenshot shows a step in a wizard titled "Connecting and managing this cluster". On the left, there is a vertical list of four steps: 1. Create a Kubernetes cluster (marked with a green checkmark), 2. Connecting to Kubernetes (marked with a blue circle), 3. Verify connectivity, and 4. Deploy a workload. To the right of the steps is a main content area. At the top of the content area, it says "Connecting and managing this cluster". Below that, a note recommends using the Kubernetes official client and DigitalOcean's command-line tool, `doctl`, to interact with and manage clusters. It also mentions adding an authentication token or certificate to your `kubectl` configuration file. There are two tabs at the top of this section: "Automated (recommended)" and "Manual", with "Automated" being selected. Below the tabs, there is a note about a dependency-free authentication method: "This is a dependency free way to authenticate. First, [download the cluster configuration file](#)". A command line instruction follows: "Update the <path to directory> in the command below and run it to authenticate: `kubectl --kubeconfig=/<path to directory>/aymen-kubeconfig.yaml get nodes`". Below the command is a link "Having trouble installing or connecting?". At the bottom of the content area is a blue "Continue" button.

Upload the “kubeconfig” file to your development server and place it under “`$HOME/.kube/config`” on the server.

For example, if you are using the root user, execute the following commands:

```
1 export IP=<your_development_server_ip>
2 scp kubeconfig.yaml root@$IP:/root/.kube/config
```

You will need to create the “.kube” folder remotely if it is not created yet.

If you are not using another user, let’s say “myuser”, then you should execute:

```
1 export IP=<your_development_server_ip>
2 scp kubeconfig.yaml myuser@$IP:/home/myuser/.kube/config
```

To make sure the `kubectl` works with the downloaded `kubeconfig` file, execute the following command:

```
1 ssh root@$IP  
2 kubectl get services -A
```

You should be able to see the default services running on your DigitalOcean cluster.

4 Kubernetes architecture overview

4.1 Introduction

Kubernetes is a distributed system composed of several components that work together to provide a highly available, scalable, and resilient platform.

The architecture of Kubernetes is built on a cluster of nodes, with each node representing an individual machine in the cluster. These nodes work together to form a highly available and fault-tolerant system that is capable of running a large number of containerized applications.

Kubernetes is made of two main elements:

- The Control Plane
- Nodes

Every cluster has at least 1 Control Plane and 1 worker node.

4.2 The Control Plane

The Control Plane is composed of several other components, each playing a specific role in managing and monitoring the cluster.

Here is a brief description of each component:

4.2.1 etcd

etcd is a distributed key-value data storage system used to store the configuration and state of the cluster.

4.2.2 API Server (kube-apiserver)

The API Server is the main entry point for interactions with the cluster. It exposes a RESTful API that allows users to create, modify, and delete objects in the cluster using a CLI like kubectl, an SDK, or by calling the API directly.

4.2.3 Controller Manager (kube-controller-manager)

The Controller Manager is responsible for managing Kubernetes controllers. These controllers monitor the state of objects in the cluster and take action to ensure that the desired state is maintained.

4.2.4 Cloud Controller Manager (cloud-controller-manager)

This is a cloud-specific component that manages services related to specific platforms, such as AWS, GCP, and so on.

4.2.5 Scheduler (kube-scheduler)

The Scheduler assigns Pods to Nodes in the cluster, ensuring that they are placed on Nodes with the necessary resources to run them.

4.3 Worker nodes

Nodes are the machines where Kubernetes runs the user workloads. Each node in a cluster runs the following components:

4.3.1 Kubelet

Kubelet is the Kubernetes agent that runs on each Node. It is responsible for communication with the Control Plane and for managing Pods and containers on the Node.

4.3.2 Container Runtime

The Container Runtime is the tool used to run containers. The most common Container Runtimes are Docker⁵¹, containerd⁵², and CRI-O⁵³.

4.3.3 Kube-proxy

Kube-proxy is a network proxy that runs on each Node. It is responsible for managing network traffic between the Pods in the cluster.

4.4 Node pools

A node pool is a group of identical nodes (or workers) in a Kubernetes cluster. It allows nodes to be grouped based on their processing capabilities and role in the cluster.

A node pool can be used for specific tasks, such as running workloads that require high computing power or hosting applications that require high data storage.

Each node pool has a specific configuration and label that allows it to be identified and managed independently from other nodes in the cluster.

Node pools enable more efficient and granular management of resources in a Kubernetes cluster, allowing administrators to deploy and manage nodes based on the specific needs of applications and services.

A typical use case of node pools in Kubernetes is when you need to scale your application horizontally by adding or removing worker nodes to handle the workload. Node pools allow you to create groups of nodes with similar characteristics and assign different workloads to them. For example, you can create a node pool for CPU-intensive tasks and another node pool for memory-intensive tasks, and assign Pods to these node pools based on their resource requirements.

If you want more control over where Pods are scheduled, you can use node taints to mark nodes as unsuitable for scheduling certain Pods, or node tolerations to allow certain Pods to be scheduled on specific nodes despite their taints.

⁵¹<https://github.com/moby/moby>

⁵²<https://github.com/containerd/containerd>

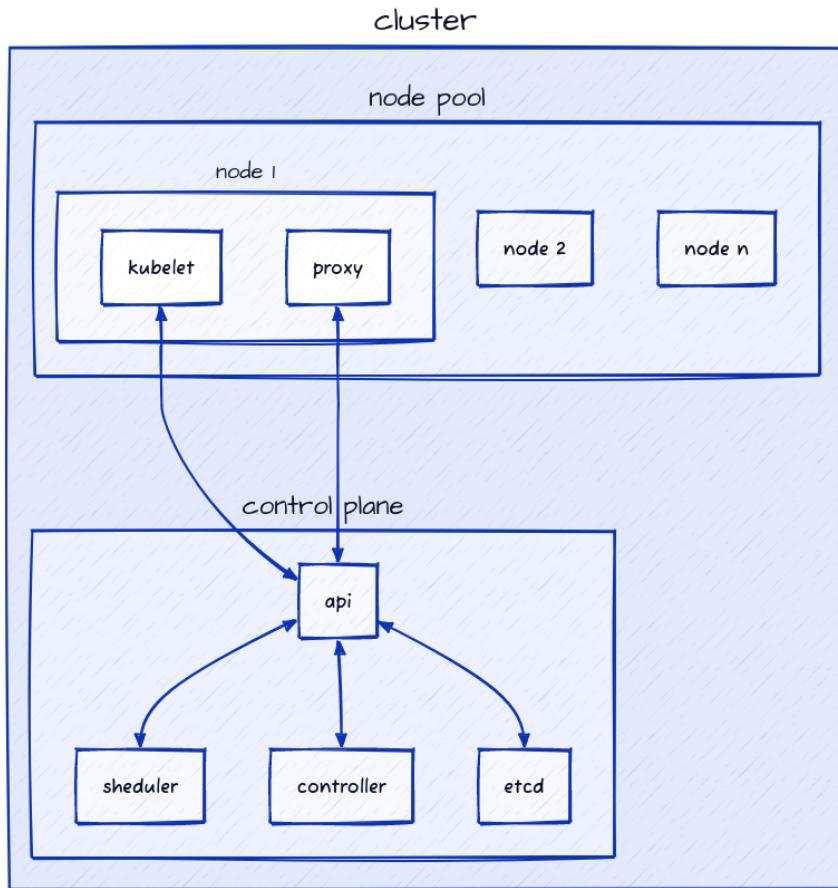
⁵³<https://github.com/cri-o/cri-o>

In a cluster, individual node pools can be created, upgraded, and deleted without affecting the entire cluster or the other nodes in different node pools. However, when making configuration changes to a node pool, the changes will apply to all nodes within that pool; it is not possible to only configure a single node within a pool.

Node pools can also assist in upgrading Kubernetes. By creating a new pool of nodes with the updated version of Kubernetes, an administrator can gradually move workloads from the old pool to the new pool.

4.5 An overview of the architecture

The following diagram shows the different components of a Kubernetes cluster.



K8s Architecture

5 Stateless and stateful microservices

5.1 Introduction

In a microservice architecture, different components work together but independently to achieve a specific goal or solve a problem. These services can be categorized into two types: stateless and stateful workloads.

Understanding the difference between stateless and stateful workloads is important when designing and deploying applications to the cloud.

In the following chapters of this guide, we are going to study how they are different when it comes to deploying and managing them in a Kubernetes cluster.

5.2 Stateless workloads

Stateless workloads refer to applications that do not require information about previous interactions or events to complete a task. Each request is treated independently, with no knowledge of any previous requests.

A good example of a stateless workload is a web server that serves static content, such as HTML, JS, or CSS files. Another example is a container that runs a batch job to perform data transformation.

When a stateless workload needs to store a state, it usually relies on an external source of data.

These applications do not store any data between requests, so they can be easily scaled by creating additional instances of the application or service without any special constraints.

When deploying or scaling this kind of workload, there are no complex overhead operations.

Given the operational flexibility of this type of workload, they became increasingly popular in our cloud-native era. Containers like Docker for example are stateless by default, when a container is down, all data stored inside it is lost. When you create a new container with the same image, the workload resets to its initial state - the state described in the Dockerfile.

The more your application relies on external data sources, the easier their deployment, scaling, and maintenance become.

However, these advantages come with some disadvantages, primarily due to the delegation of data responsibility to external sources. This can result in higher latencies, as computation is not located near storage.

5.3 Stateful workloads

Even if stateless has become popular with the rise of containers and cloud-native approaches, data is essential to most applications developers build. In most cases, our applications rely on data stores of all sorts: Redis, PostgreSQL, MySQL..etc

This is when stateful workloads come into play.

Stateful workloads are applications that require persistent data storage and rely on maintaining state across multiple instances. These applications are typically more complex than stateless applications and can be more challenging to deploy and manage in Kubernetes given the additional complexity of managing storage.

Examples of stateful workloads include databases, message queues, and file servers. Databases (e.g. MongoDB) and message queues (e.g. Kafka) are stateful.

While stateless approaches have been the primary choice for developers, they may introduce processing patterns with their own overhead and performance costs, whereas a stateful approach may be more efficient for high-performance, near real-time, and stream-based systems.

6 Deploying Stateless Microservices: Introduction

6.1 Requirements

We are going to use our Ubuntu development server when we installed kubectl and uploaded the kubeconfig file.

For the next examples, we are going to use Python as the main programming language and Flask as a framework. Python and Flask are easy to understand even if you are not a Python developer, therefore with these choices, this guide will be easy to read and understand for all developers.

In your development server, start by installing:

- pip3: The official package manager and pip command for Python 3.
- virtualenvwrapper: A set of extensions for creating and deleting Python virtual development environments

```
1 apt get update && apt install -y python3-pip
2 pip install virtualenvwrapper
3 export WORKON_HOME=~/Envs
4 mkdir -p $WORKON_HOME
5 export VIRTUALENVWRAPPER_PYTHON='/usr/bin/python3'
6 source /usr/local/bin/virtualenvwrapper.sh
```

Let's create a new virtual environment:

```
1 mkvirtualenv stateless-flask
```

Then create the folders for our Flask application and install its dependencies.

```
1 mkdir -p stateless-flask
2 cd stateless-flask
3 mkdir -p app
4 mkdir -p kubernetes
5 pip install Flask==2.2.3
6 pip freeze > app/requirements.txt
```

The following code, create a simple todo application:

```
1 cat << EOF > app/app.py
2 from flask import Flask, jsonify, request
3
4 app = Flask(__name__)
5
6 # Define a list of tasks
7 tasks = []
8
9 # route for getting all tasks
10 @app.route('/tasks', methods=['GET'])
11 def get_tasks():
12     return jsonify({'tasks': tasks})
13
14 # Route for getting a single task
15 @app.route('/tasks', methods=['POST'])
16 def add_task():
17     task = {
18         'id': len(tasks) + 1,
19         'title': request.json['title'],
20         'description': request.json['description'],
21     }
22     tasks.append(task)
23     return jsonify(task), 201
24
25 if __name__ == '__main__':
26     app.run(debug=True, host='0.0.0.0', port=5000)
27 EOF
```

This code creates a Flask application that defines a list of tasks. It has two routes - one to get all tasks and another to add a task.

The `get_tasks()` function returns a JSON response containing all the tasks in the list.

The `add_task()` function creates a new task with an ID, title, and description, adds it to the list of tasks, and returns a JSON response containing the new task.

The following block of code runs the Flask application and makes it available on the localhost on port 5000. The debug mode is enabled to help with development.

```
1 if __name__ == '__main__':
2     app.run(debug=True, host='0.0.0.0', port=5000)
```

Next, we are going to create a Dockerfile:

```
1 cat << EOF > app/Dockerfile
2 # Use an official Python runtime as a parent image
3 FROM python:3.9-slim-buster
4 # Set the working directory to /app
5 WORKDIR /app
6 # Copy the current directory contents into the container at /app
7 COPY . /app
8 # Install any needed packages specified in requirements.txt
9 RUN pip install --no-cache-dir -r requirements.txt
10 # Make port 5000 available to the world outside this container
11 EXPOSE 5000
12 # Define environment variable
13 CMD ["python", "app.py"]
14 EOF
```

We can now build the image and run it to test it.

```
1 docker build -t stateless-flask:v0 -f app/Dockerfile app
2 docker run -it -p 5000:5000 stateless-flask:v0
```

Since we are exposing the container on port 5000 of the host, we will query the Flask-generated API by using a tool like Postman, the public IP of our droplet followed by port 5000.

Download and install [Postman](#)⁵⁴.

The screenshot shows the Postman application interface. At the top, there are tabs for 'New Request', 'GET Untitled Request', and 'POST http://stateless-flask.16'. Below the tabs, it says 'Temporary / New Request'. The main area shows a 'POST' request to 'http://stateful-flask.144.126.246.58.nip.io/tasks'. Under the 'Body' tab, the request body is set to 'raw' and contains the following JSON:

```

1  {
2      "title": "Kubernetes",
3      "description": "Master the art of using containers, Kubernetes and \
4      microservices",
5      "id": 1,
6      "title": "Kubernetes"

```

At the bottom right of the interface, there is a red arrow pointing from the text 'Send your request' to the 'Send' button in the top right corner.

You can use a POST request with the following data:

```

1  {
2      "description": "Master the art of using containers, Kubernetes and \
3      microservices",
4      "id": 1,
5      "title": "Kubernetes"
6  }

```

Don't forget to add the header Content-Type and make it accept JSON by giving it the value application/json.

After saving the first element, you can execute a GET request on /tasks, you will get the same JSON data you stored. However, when stopping the container, the data will be erased, which is normal since the application is stateless and does not store in any external datastore.

Now, create an account on [Docker Hub](#)⁵⁵ and push the image.

⁵⁴<https://www.postman.com/downloads/>

⁵⁵<https://hub.docker.com/>

```
1 docker login
2 docker tag stateless-flask:v0 <dockerhub_username>/stateless-flask:v0
3 docker push <dockerhub_username>/stateless-flask:v0
```

Change <dockerhub_username> by your Docker Hub username.

6.2 Creating a Namespace

We can deploy the application to the default Namespace, however, this is not recommended, especially when you have multiple applications running in the same cluster. In this case, the recommended approach is to create a separate Namespace for each application.

We are going to deploy the Flask API in a Namespace different from the default one. Let's create it:

```
1 cat <<EOF > kubernetes/namespace.yaml
2 apiVersion: v1
3 kind: Namespace
4 metadata:
5   name: stateless-flask
6 EOF
```

Now, apply the YAML.

```
1 kubectl apply -f kubernetes/namespace.yaml
```

Print a list of all Namespaces:

```
1 kubectl get ns
```

You should be able to see the newly created Namespace.



A Namespace provides a virtual separation of resources within a cluster.

It allows for multiple teams or applications to use the same Kubernetes cluster while having their own isolated sets of resources such as pods, services, and volumes.

This virtual separation helps with managing resources and avoids naming conflicts between teams or applications.

For example, when you want to run multiple applications on a single cluster, you can deploy each application to a different Namespace and an Ingress resource that routes traffic to all these applications in another Namespace.

6.3 Creating the Deployment

To deploy our Flask container, we need to create a Deployment. The Deployment will create a ReplicaSet and the latter is responsible of maintaining a stable set of replica Pods.



A Pod is the smallest and simplest unit of the Kubernetes object model, and it represents a single instance of a running process in the cluster.

A Pod can contain one or more containers, which are guaranteed to run together on the same host and share the same network Namespace.

All containers in a Pod share the same set of Linux Namespaces, including the network Namespace, which means they can communicate with each other over the localhost network interface.

Containers in a single Pod share the same lifecycle, when a Pod is created, all containers are created and when a Pod is killed or deleted, all containers inside that Pod disappear.

For example, if “container_a” and “container_b” run on the same Pod, they both have the same IP, which is the Pod IP. “container_a” will access “container_b” like it would access the localhost⁵⁶ and vice-versa.

In the following YAML manifest, make sure to change <your-docker-registry> to your Docker Hub username.

```
1 cat << EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateless-flask
6   namespace: stateless-flask
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: stateless-flask
12   template:
13     metadata:
14       labels:
15         app: stateless-flask
16   spec:
17     containers:
18       - name: stateless-flask
19         image: <your-docker-registry>/stateless-flask:v0
20         ports:
21           - containerPort: 5000
22 EOF
```

Let’s see what the Deployment file does line by line:

- **apiVersion** specifies the Kubernetes API version to use for this object, which is “apps/v1” in this case.

⁵⁶<http://localhost/>

- **kind** specifies the kind of object being created, which is a Deployment.
- **metadata** contains some metadata about the Deployment, including the name and Namespace of the deployment.
- **spec** specifies the desired state of the Deployment. It contains the number of replicas to be created (in this case, 3), as well as a selector to match the labels for the Pods that should be managed by the Deployment.
- The **selector** field in a Kubernetes Deployment specifies the labels used to identify and select which Pods the Deployment manages. In our code, the selector specifies that the Deployment will manage Pods with the label **app: stateless-flask**. This means that any Pods that match this label will be included in the set of replicas maintained by the Deployment, and any Pods that do not match this label will not be included. The **matchLabels** field in the selector specifies the labels that must match for a Pod to be included in the set.
- **template** contains the specification for the Pods that should be created by the Deployment. It specifies the labels to be applied to the Pods, which should match the selector in the **spec**.
- **containers** contains the list of containers to be deployed in the Pods. In this case, it only has one container named “stateless-flask”, which uses the Docker image specified in **image** and exposes port 5000.

We can apply the YAML file and get the list of Deployments in “stateless-flask” Namespace. Again, make sure to change `<your-docker-registry>` by your Docker Hub username.

```
1 kubectl apply -f kubernetes/deployment.yaml  
2 kubectl get pods -n stateless-flask
```

The following schema shows the different Kubernetes resources we have created.

Note that our cluster has one node. If we had two, the 3 Pods we created will most likely be distributed across nodes.



A Deployment is responsible for managing a set of replicas of a specific Pod.

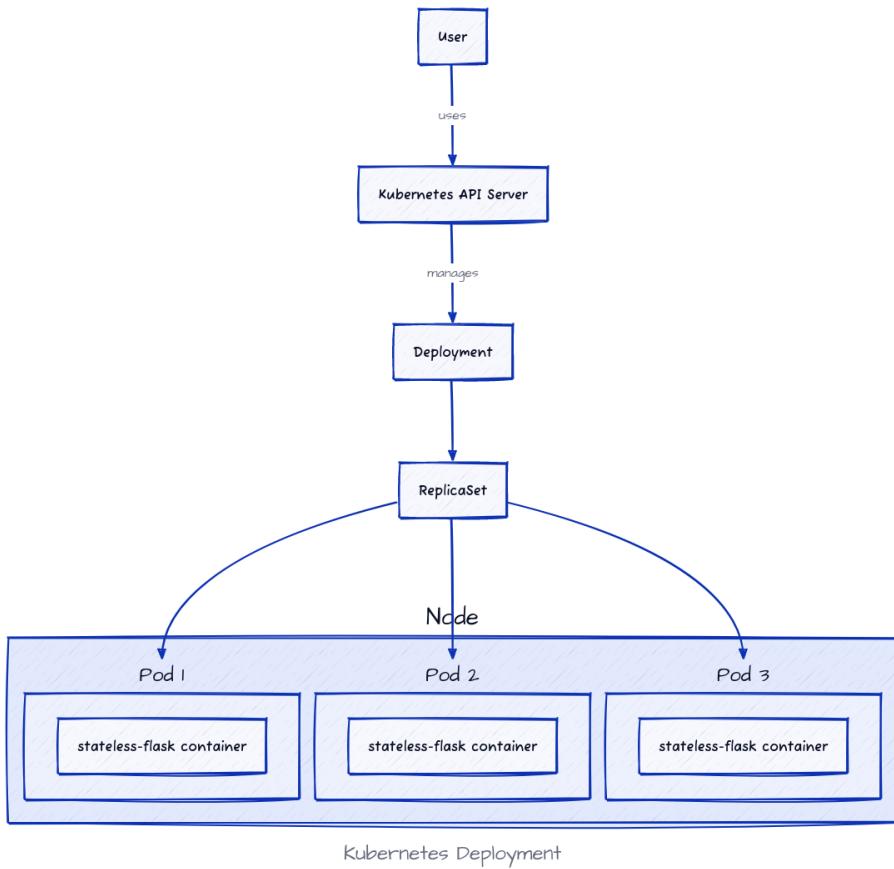
The Deployment ensures that the desired number of replicas are running at any given time and provides a way to do rolling updates and rollbacks of the underlying Pod template.

It provides a declarative interface to deploy, update and delete Pods and ReplicaSets.



A ReplicaSet is another Kubernetes object that is responsible for maintaining a stable set of replica Pods. It works by ensuring that a specific number of replica Pods based on a defined template (usually defined in a Deployment) are running.

ReplicaSets are often used as part of a Deployment, with the Deployment managing the ReplicaSet to ensure the desired number of replicas are maintained. So it is not recommended to create a ReplicaSet directly. Instead, create a Deployment.



6.4 Examining Pods and Deployments

A useful command we can use to check the status, events, and configuration of the Pod is `kubectl describe`.

If you used Docker, you are certainly familiar with the `docker inspect <container>` command. The `kubectl describe pod <pod>` command, is somehow similar.

```
1 kubectl -n stateless-flask describe pod <pod>
```

Change `<pod>` by the name of any Pod. You can get the list of Pods by using the `kubectl -n stateless-flask get pods` command.

You can also describe the Deployment using:

```
1 kubectl -n stateless-flask describe deployment <deployment>
```

One of the most helpful pieces of information that the `describe` command provides is the list of events.

Events provide a way to monitor the state of different objects within the system. These objects could be nodes, pods, deployments, services, and so on. They provide an audit trail of changes that occur to these objects over time.

Pods' logs are another way to understand what's happening inside a Pod. Using the following command, we can see the logs of a Pod:

```
1 kubectl -n stateless-flask logs <pod>
```

You need to change `<pod>` by the name of any Pod.

If you want to get the logs of all pods in a single command, you can do it by filtering by the label. This is an example:

```
1 kubectl -n stateless-flask logs -l app=stateless-flask
```

We are using `app=stateless-flask` because that's what we configured in the template of our Pods in the Deployment file.

To show logs and follow the changes, you can add the `-f` flag.

```
1 kubectl -n stateless-flask logs <pod> -f
```

```
2 kubectl -n stateless-flask logs -l app=stateless-flask -f
```

6.5 Accessing Pods

A Pod does not explore any external address/port, therefore, if we want to see the Flask application running, we need to forward the internal Pod port to one of the ports in our development server. This is how to do it:

```
1 kubectl port-forward stateless-flask-<pod_id> 5000:5000 -n stateless-flask\n2 ask
```

Make sure to change `stateless-flask-<pod_id>` by the name of any Pod. You can get the list of Pods using `kubectl get pods -n stateless-flask`.

You can also use AWK to get the first pod that appears on the list when you type `kubectl get pods` as follows:

```
1 export pod=$(kubectl get pods -n stateless-flask | awk 'FNR==2{print $1}\n2 }' )\n3\n4 kubectl port-forward $pod 5000:5000 -n stateless-flask
```

In both cases, the Pod is now accessible via our local server, we can check its response using:

```
1 curl http://0.0.0.0:5000/tasks
```

6.6 Exposing a Deployment

If we want to use our Flask application in a regular way, we should make it accessible via a web browser. However, there is no external access to our Pod, unless we create a port forwarding. This is why we need to create a Service.



A Service is a powerful abstraction layer that provides a seamless way to expose a Deployment of Pods as a network service.

A Deployment is the backbone of your application, managing the creation and scaling of a set of identical Pods with ease. Meanwhile, a Service gives you a reliable IP address and DNS name to access those Pods.

By defining a Service for a Deployment, you can easily access the pods from other applications within the cluster or from outside the cluster through the Service's IP address or DNS name.

This smart decoupling of the Pods from the Service entitles easy management of the Pods and provides more flexibility in how they are accessed.

In Kubernetes, there are several ways to expose an application on the network, including ClusterIP, NodePort, LoadBalancer, Ingress, and Headless services.

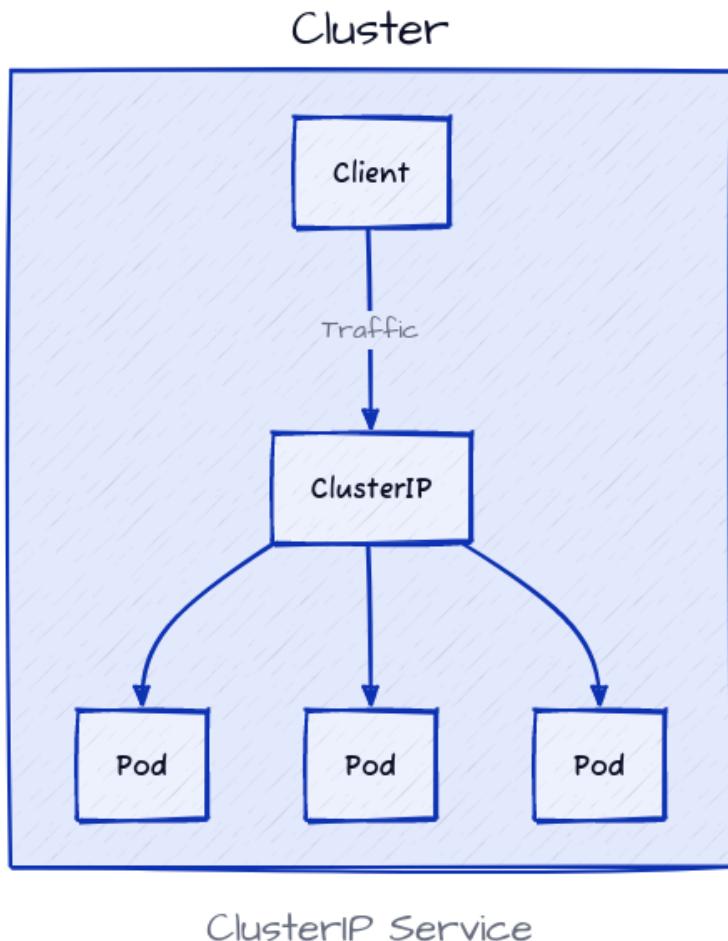
6.6.1 ClusterIP Service

ClusterIP is the default Service type that provides a stable IP address to reach the service within the cluster. It exposes the Pods in a cluster to other objects in the same cluster using an internal IP address, allowing for seamless communication between components.

ClusterIP service is used to enable communication between different components or microservices within a Kubernetes cluster. It acts as a [virtual IP](#)⁵⁷ address assigned to a set of Pods.

This is what a ClusterIP looks like:

⁵⁷https://en.wikipedia.org/wiki/Virtual_IP_address



This is how to create a ClusterIP for our service:

```
1 cat <<EOF > kubernetes/cluserip-service.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: stateless-flask-clusterip-service
6   namespace: stateless-flask
7 spec:
8   type: ClusterIP
9   selector:
10    app: stateless-flask
11   ports:
12     - port: 5000
13     protocol: TCP
14     targetPort: 5000
15 EOF
```

- **apiVersion**: The version of the Kubernetes API that this manifest is written in.
- **kind**: The type of Kubernetes object that this manifest describes. In this case, it's a Service.
- **metadata**: Metadata about the Service, including its name and Namespace.
- **spec**: The specification of the Service, which includes its type, selector, and ports.
- **type: ClusterIP** indicates that this Service is a ClusterIP Service, which means that it's only accessible within the cluster.
- **selector**: defines a selector that identifies the Pods that this Service should route traffic to. In this case, it selects the Pods with the label **app: stateless-flask**.
- **ports**: specifies the port configuration of the Service. In this case, it exposes port 5000 of the Pods as port 5000 of the Service.
- **port** refers to the port that the service will listen on, while the **targetPort** refers to the port that the backend Pods of the Service are listening on.

Apply the YAML file and check the list of Services:

```
1 kubectl apply -f kubernetes/cluserip-service.yaml
2 kubectl get svc -n stateless-flask
```

To access the application, we need to use the ClusterIP service. To do this, we will use Kubernetes port-forwarding:

```
1 kubectl port-forward svc/stateless-flask-clusterip-service 5000:5000 -n \
2 stateless-flask
```

Now, we can test the connection to the service using a cURL command.

```
1 curl [http://127.0.0.1:5000/tasks] (http://127.0.0.1:5000/tasks)
```

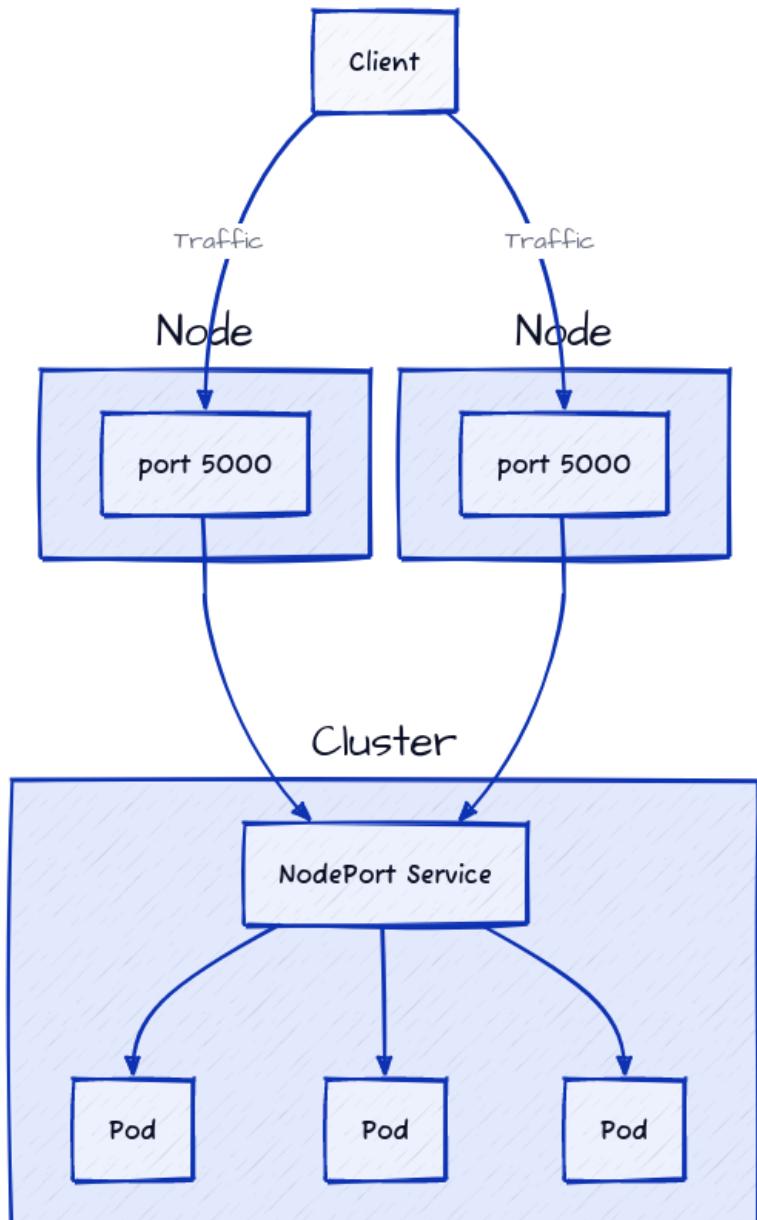
6.6.2 NodePort Service

NodePort is a service that exposes a deployment externally on a static port on each node in the cluster. The port range for NodePort services is 30000-32767, which allows external traffic to reach the service on the specified port, which is then forwarded to the Pod.

One of the advantages of NodePort is simplicity, as it doesn't require any external load balancers or network components. It's also easy to use and doesn't require much configuration.

However, there are some drawbacks to using NodePort. It exposes the service on a static port on every node in the cluster, which can cause security problems if you have potential security problems. It also requires opening up a port range on all nodes in the cluster, which can be a potential attack vector.

NodePort can be useful for testing and debugging purposes, however, if you have a production workload and you want to allow external traffic, it is not the most suitable solution.



Kubernetes NodePort

This is how we can create a NodePort service:

```
1 cat <<EOF > kubernetes/nodeport-service.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: stateless-flask-nodeport-service
6   namespace: stateless-flask
7 spec:
8   type: NodePort
9   selector:
10    app: stateless-flask
11   ports:
12     - port: 5000
13     protocol: TCP
14     targetPort: 5000
15     nodePort: 30000
16 EOF
```

The ports that this service will listen on are defined in **ports**, including **port** (the port that the service listens on), **protocol** (the protocol used by the port), **targetPort** (the port on the pod that traffic should be sent to), and **nodePort** (the port that the service is exposed on each node in the cluster). In this case, the service listens on port **5000** and is exposed on each node in the cluster on port **30000**.

Now you can apply that YAML:

```
1 kubectl apply -f kubernetes/nodeport-service.yaml
```

List the services:

```
1 kubectl get svc -n stateless-flask
```

Get the external IP address of any of your cluster nodes:

```
1 kubectl get nodes -o wide
```

Using your web browser, a tool like Postman or cURL, you can check the response of the API using the external IP of the node and port 30000.

```
1 curl http://<external_ip>:30000/tasks
```

If you want to do it all in a single command, you can use:

```
1 curl HTTP://$(kubectl get nodes -o wide | awk '{print $7}' | tail -n +2\\
2 ):30000/tasks
```

`kubectl get nodes -o wide | awk '{print $7}' | tail -n +2` will get the external IP address without showing all the output of `kubectl get nodes -o wide`.

6.6.3 LoadBalancer Service

LoadBalancer is another type of service available for use in a Kubernetes cluster. It helps distribute incoming traffic to multiple replicas of a Pod. It is an implementation of the [Load Balancing](#)⁵⁸ pattern, which is used to distribute traffic across multiple nodes, services, or systems.

In Kubernetes, things are a little bit different: To implement this pattern, you will need two things:

- A LoadBalancer service
- A load balancer machine

While the service is a Kubernetes object that is part of the cluster, the machine is external and does not make part of the cluster. If you are managing a bare-metal Kubernetes or an on-premises cluster, the implementation could be challenging, however, if you are using a managed cluster using a public cloud such as AWS, GCP, or DigitalOcean, things are much easier.

⁵⁸[https://en.wikipedia.org/wiki/Load_balancing_\(computing\)](https://en.wikipedia.org/wiki/Load_balancing_(computing))

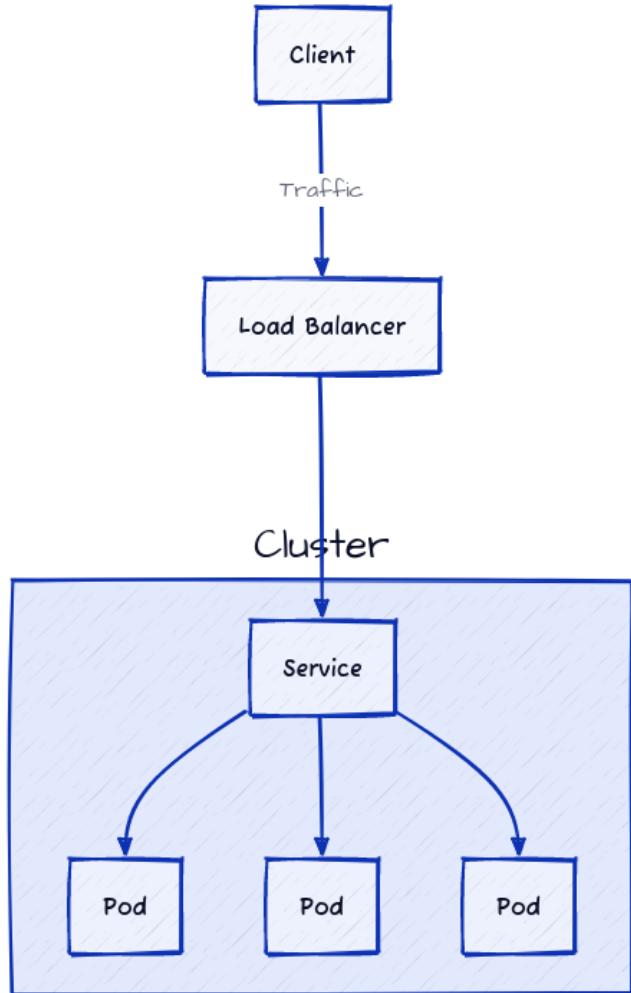
Usually, the cloud provider automatically provisions the load-balancing machine for you.

A LoadBalancer service can help improve the availability and scalability of an application by distributing traffic across multiple replicas, which reduces the likelihood of a single point of failure.

One of its important advantages is its ability to automatically detect unhealthy replicas and redirect traffic to healthy ones, which adds another layer of reliability to your workload.

Additionally, when you create a LoadBalancer, you can add advanced features like SSL termination, connection draining, and session affinity.

However, creating a Load Balancer machine for each of your services can be expensive. Also, adding an additional networking layer may introduce latency, but in most cases, it is not significant.



Kubernetes Load Balancer

To create a LoadBalancer service, use the following YAML:

```
1 cat <<EOF > kubernetes/loadbalancer-service.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: stateless-flask-loadbalancer-service
6   namespace: stateless-flask
7 spec:
8   type: LoadBalancer
9   selector:
10    app: stateless-flask
11   ports:
12     - port: 5000
13     protocol: TCP
14     targetPort: 5000
15 EOF
```

Here, a single port is defined with the following properties:

- **port**: The port number that the service will be available on.
- **protocol**: The protocol used by the service, in this case, TCP.
- **targetPort**: The port number that the service will route traffic to on the Pods.

Run the apply command:

```
1 kubectl apply -f kubernetes/loadbalancer-service.yaml
```

Watch the list of your services using `kubectl get svc -n stateless-flask -w` and when the external IP address of your load balancer machine is ready, start testing the API using the same IP and the port 5000 (port).

Let's test:

```
1 load_balancer_ip = $(kubectl get svc -n stateless-flask | grep "LoadBal\`  
2 ancer" | awk '{print $4}')  
3  
4 curl http://$load_balancer_ip:5000/tasks
```

6.6.4 Headless Service

Normally, a Kubernetes service has a virtual IP address (ClusterIP) and a DNS name that resolves to this IP. When a client connects to the service IP, it is transparently redirected to one of the pods that are backing the service. What if I want to access each Pod using its own IP address?

This is possible with the Headless services feature of Kubernetes.

Headless services are created using the same syntax as regular services but with the `clusterIP` set to "None".

This is how to create a Headless service:

```
1 cat <<EOF > kubernetes/headless-service.yaml  
2 apiVersion: v1  
3 kind: Service  
4 metadata:  
5   name: stateless-flask-headless-service  
6   namespace: stateless-flask  
7 spec:  
8   clusterIP: None  
9   selector:  
10    app: stateless-flask  
11   ports:  
12    - port: 5000  
13      protocol: TCP  
14      targetPort: 5000  
15 EOF
```

You can now create the service:

```
1 kubectl apply -f kubernetes/headless-service.yaml
```

Then you can list the services:

```
1 kubectl get svc -n stateless-flask
```

To access a Headless service from outside the cluster, you can use the public IP address of one of the Pods in the service followed by the port specified in the Kubernetes manifest we deployed (5000 in our case).

In order to test the DNS resolution, let's create a temporary Pod `tmp01` that runs the image “[tutum/dnsutils](#)⁵⁹”. We are deploying this Pod because it has dnsutils installed, which is a set of tools that we can use to test DNS resolution.

We will use the `nslookup` command to test the DNS resolution of the `stateless-flask-headless-service.stateless-flask.svc.cluster.local` service.

Note that the Headless service does not have an IP, however it can be accessible by its internal DNS name:

```
1 stateless-flask-headless-service.stateless-flask.svc.cluster.local
```

Deploy the temporary Pod `tmp01` using `kubectl`:

```
1 kubectl run tmp01 --image=tutum/dnsutils -- sleep infinity
```

Use `nslookup` to test the DNS resolution of the Headless service. We can execute a command from a Pod without being inside it using:

```
1 kubectl exec -it tmp01 -- nslookup stateless-flask-headless-service.stateless-flask.svc.cluster.local
```

The output should be similar to:

⁵⁹<https://hub.docker.com/r/tutum/dnsutils>

```
1 Server: <IP of the DNS server>
2 Address: <IP of the DNS server>#53
3
4 Name: stateless-flask-headless-service.stateless-flask.svc.cluster.local
5 Address: <IP of the Pod 1>
6 Name: stateless-flask-headless-service.stateless-flask.svc.cluster.local
7 Address: <IP of the Pod 2>
```

Notice how the DNS server returns the IP addresses of the Pods that are running the `stateless-flask` application. A ClusterIP will never return the IP addresses of the Pods, it will only return its own IP address.

Database clustering is a common use case for Headless services. RabbitMQ, MySQL, and PostgreSQL are examples of applications that can be clustered using Headless services.



Internal names are only accessible from within the cluster. They use the format `<service-name>. <namespace>. svc.cluster.local`. For example, to access the `stateless-flask-headless-service` service from another Namespace (different from `stateless-flask`) we use `stateless-flask-headless-service.stateless-flask.svc.cluster.local` where `stateless-flask` is the Namespace where the service is running, `stateless-flask-headless-service` is the name of the service and `svc.cluster.local` is the domain name of the cluster.

6.6.5 Ingress Service

An Ingress is a component that allows inbound connections to the cluster. It provides a way to manage external access to the services in the cluster. Instead of creating a separate LoadBalancer service for each service, Ingress enables you to define a set of rules that route traffic to different services based on the request's host or path.

Ingress services simplify access to your services by using a single IP address and domain names, it is also able to use an SSL/TLS certificate to encrypt the traffic. It

has a flexible configuration that allows you to define the rules to route the traffic to the services.

For example, I can redirect the traffic to the service `my-service` when the request is made to the host `myhost.com` and the path `/my-service`. While when the request is made to the host `myhost.com` and the path `/my-other-service` the traffic will be redirected to the service `my-other-service`.

Even if some cloud providers do not support native Ingress services, you can always use an open-source or commercial Ingress controller like [Traefik](#)⁶⁰ or [Nginx](#)⁶¹.

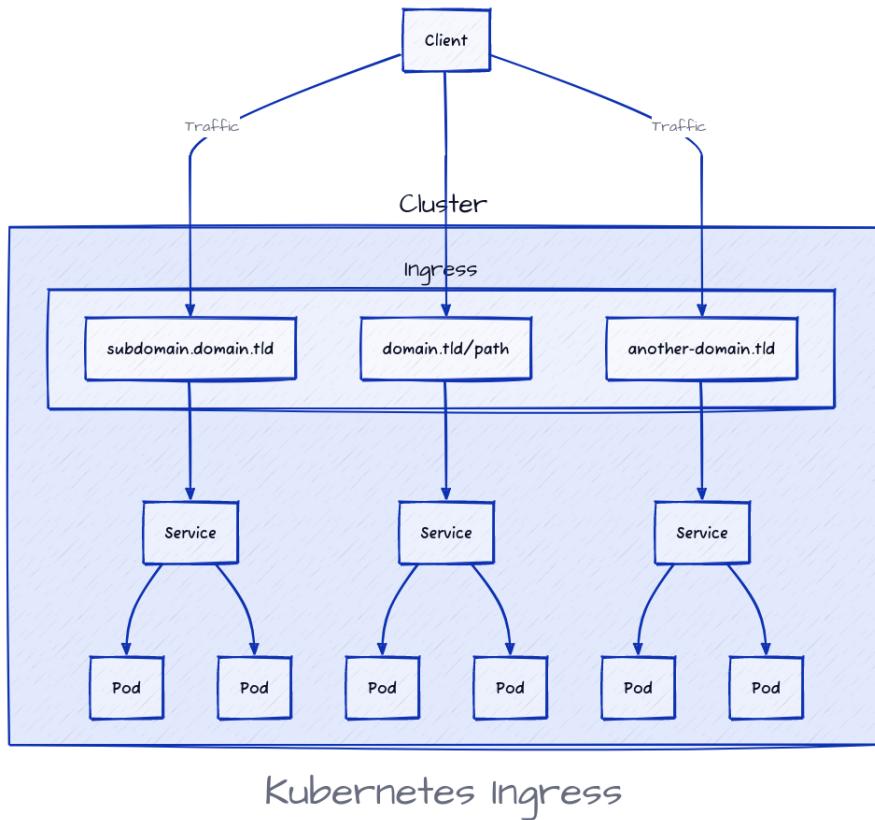
This is a list of some of them:

- Nginx Ingress: This is one of the most popular controllers, it uses the Nginx web server to manage Ingress traffic.
- Traefik Ingress: This controller supports multiple proxy technologies and offers easy integration with cloud service providers.
- HAProxy Ingress: This controller uses the HAProxy proxy server to manage Ingress traffic.
- Istio Ingress: Istio is a service mesh platform that offers an Ingress controller with advanced security features such as authentication, authorization, and encryption.
- Contour Ingress: This controller uses the Envoy proxy server to manage Ingress traffic and offers easy integration with Kubernetes resources.

It is also important to note that routing rules are based on the HTTP(S) request, so it is not able to route TCP or UDP traffic. If you want to expose a TCP/UDP port, you will need to use a LoadBalancer or a NodePort.

⁶⁰<https://traefik.io/>

⁶¹<https://www.nginx.com/>



Before creating the Ingress, let's clean all the previously created services:

- 1 `kubectl delete -f kubernetes/nodeport-service.yaml`
- 2 `kubectl delete -f kubernetes/loadbalancer-service.yaml`
- 3 `kubectl delete -f kubernetes/clusterip-service.yaml`

We will need a ClusterIP service in front of our Pods. This service will be used by the Ingress controller to route the traffic to the Pods. Let's create it:

```
1 cat <<EOF > kubernetes/stateless-flask-service.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: stateless-flask
6   namespace: stateless-flask
7 spec:
8   selector:
9     app: stateless-flask
10  ports:
11    - name: http
12      protocol: TCP
13      port: 5000
14      targetPort: 5000
15 EOF
16
17 kubectl apply -f kubernetes/stateless-flask-service.yaml
```

Now, let's create the Ingress resource.

As we have seen there are multiple Ingress controllers available. We will use the Nginx Ingress controller. This controller is deployed as a Pod in the cluster.

To install the Nginx Ingress controller, we will use a Helm chart.

Helm⁶² is a package manager for Kubernetes. It allows to easily install applications on kubernetes/cluserip-service.yaml

Start by installing Helm on your development server:

```
1 curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-hel\
2 m-3 | bash
```

Then, add the Nginx Ingress controller repository:

⁶²<https://helm.sh/>

```
1 helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
2 helm repo update
3 helm install nginx-ingress ingress-nginx/ingress-nginx --set controller\
4 .publishService.enabled=true
```

It is a good practice to create a dedicated Namespace for the Ingress controller but here we will use the default Namespace to keep things simple. If you want to use Helm to install the Ingress controller in a dedicated Namespace, you can use the following command:

```
1 helm install nginx-ingress ingress-nginx/ingress-nginx --set controller\
2 .publishService.enabled=true --namespace ingress-nginx
```

The `helm install` command installs the Nginx Ingress controller from the stable charts repository. It defines the `publishService` parameter to true which means that the Ingress controller will be published as a service. Because we didn't specify a Namespace, Helm will install the controller in the default Namespace.

Now, we can list the Pods and Services in the default Namespace:

```
1 kubectl get pods -n default
2 # or kubectl get pods
3 kubectl get services -n default
4 # or kubectl get services
```

The Nginx Ingress controller is deployed as a Pod in the default Namespace. It is also published as a service in the default Namespace. We should configure the Ingress controller to use the ClusterIP service we created earlier. To do so, we need to create the Ingress resource.

Note that the ingress Controller is not the Ingress resource. The Ingress resource is a Kubernetes resource that defines the routing rules and is used by the Ingress controller to route the traffic to the Pods through the ClusterIP service.

This is an example of an Ingress resource:

```
1 cat <<EOF > kubernetes/ingress.yaml
2 apiVersion: networking.k8s.io/v1
3 kind: Ingress
4 metadata:
5   name: my-ingress
6 spec:
7   rules:
8     - host: <host>
9       http:
10         paths:
11           - path: <path>
12             pathType: Prefix
13             backend:
14               service:
15                 name: <service-name>
16                 port:
17                   number: 5000
18   ingressClassName: nginx
19 EOF
```

- The spec section defines the ingress rules. In this case, there is a single rule that matches a specific <host> name and path prefix <path>.
- The backend section specifies which Kubernetes Service to route traffic to for requests matching the specified host and path. In this case, it specifies a service named <service-name> on port 5000.
- The pathType field is set to Prefix, which means that the path /tasks and any path that starts with /tasks/ will match this rule.
- The ingressClassName field specifies the name of the ingress class to use for this ingress. In this case, it specifies the Nginx ingress controller.

We can also add namespace field to the metadata section to specify the Namespace where the service is deployed.

If we use the above Ingress resource, we will have an Ingress in the default Namespace, while our application was deployed in the stateless-flask Namespace. This is a problem as the Ingress controller will not be able to route the traffic to the Pods.

To solve this problem, we need to create an ExternalName service in the `stateless-flask` Namespace that points to the ClusterIP service in the default Namespace. The ExternalName service is a special type of service that allows to route traffic to a service in another Namespace.

Create this file:

```
1 cat <<EOF > kubernetes/stateless-flask-service-externalname.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: stateless-flask-service-externalname
6   namespace: default
7 spec:
8   type: ExternalName
9   externalName: stateless-flask.stateless-flask.svc.cluster.local
10 EOF
```

And apply it:

```
1 kubectl apply -f kubernetes/stateless-flask-service-externalname.yaml
```

Verify that the ExternalName service is created:

```
1 kubectl get services -n default
```

Now we can create the Ingress resource:

```

1 cat <<EOF > kubernetes/ingress.yaml
2 apiVersion: networking.k8s.io/v1
3 kind: Ingress
4 metadata:
5   name: my-ingress
6 spec:
7   rules:
8     - host: stateless-flask.<ingress-ip>.nip.io
9       http:
10         paths:
11           - path: /tasks
12             pathType: Prefix
13             backend:
14               service:
15                 name: stateless-flask-service-externalname
16                 port:
17                   number: 5000
18             ingressClassName: nginx
19 EOF

```

In the above Ingress resource, we have specified the host name `stateless-flask.<ingress-ip>.nip.io`. Make sure to change `<ingress-ip>` with the IP address of the Ingress controller. You can get the IP address of the Ingress controller with the following command:

```

1 kubectl get services nginx-ingress-ingress-nginx-controller | awk '{pri\
2 nt $4}' | tail -n 1

```

Ingress works with a public DNS, however, we don't have one. We will use nip.io⁶³ to create a subdomain name on the fly from an IP address. You are free to buy and use any other domain name from a provider or use another free service like sslip.io⁶⁴.

Apply the Ingress resource:

⁶³<https://nip.io/>

⁶⁴[https://sslip.io/](https://sslip.io)

```
1 kubectl apply -f kubernetes/ingress.yaml
```

Test the Ingress resource:

```
1 kubectl get ingress
```

You should see the Ingress resource in the output. The ADDRESS field should contain the IP address of the Ingress controller.

Now, you can open your browser and go to:

```
1 http://stateless-flask.<ingress-ip>.nip.io/tasks
```

to see the list of tasks.

If you have other use cases, you can check the [official documentation](#)⁶⁵ for more details.

⁶⁵<https://kubernetes.github.io/ingress-nginx/user-guide/basic-usage/>

7 Deploying Stateful Microservices: Persisting Data in Kubernetes

7.1 Requirements

We are still using the same Ubuntu 22.04 development server. We use the server to create and deploy Kubernetes resources.

In this part of the guide, we are going to deploy a PostgreSQL database and change our Flask application code to make it use the database to store its state. When a user adds an element to the todo list, it will not disappear when the Pod restarts or disappear since everything will be saved to the PostgreSQL database. This will not make our microservice stateful, but a stateless microservice that stores its state on an external datastore.

Make sure that you already installed virtualenvwrapper:

```
1 apt get update && apt install -y python3-pip
2
3 pip install virtualenvwrapper
4 export WORKON_HOME=~/Envs
5 mkdir -p $WORKON_HOME
6 export VIRTUALENVWRAPPER_PYTHON='/usr/bin/python3'
7 source /usr/local/bin/virtualenvwrapper.sh
```

Then activate the virtual environment:

```
1 mkvirtualenv stateful-flask
```

Now, create a folder for the new application:

```
1 cd $HOME
2 mkdir -p stateful-flask
3 cd stateful-flask
4 mkdir -p app
5 mkdir -p kubernetes
```

We are going to use `stateful-flask` as a name as opposed to `stateless-flask` in the previous part. However, as said, this does not mean that the new Flask application that we are going to create is stateful, it is still stateless but uses an external datastore (PostgreSQL) to store its state. The database, on the other hand, is a stateful service.

7.2 Creating a Namespace

Let's start by creating a namespace for the database:

```
1 cat <<EOF > kubernetes/postgres-namespace.yaml
2 apiVersion: v1
3 kind: Namespace
4 metadata:
5   name: postgres
6 EOF
```

And a namespace for the Flask application:

```
1 cat <<EOF > kubernetes/namespace.yaml
2 apiVersion: v1
3 kind: Namespace
4 metadata:
5   name: stateful-flask
6 EOF
```

Then we apply the namespaces:

```
1 kubectl apply -f kubernetes/postgres-namespace.yaml  
2 kubectl apply -f kubernetes/namespace.yaml
```

7.3 Creating a ConfigMap for the PostgreSQL database

7.3.1 What is a ConfigMap?

A ConfigMap is a resource that stores non-confidential configuration data as key-value pairs. It allows configuration data to be decoupled from containerized applications, making it easy to update and manage configurations without having to rebuild and redeploy the entire application.

ConfigMaps can store configuration data mainly environment variables, configuration files, and any other non-confidential data that your applications might need. When a Pod starts, it can read the configuration data of a container from the ConfigMap as environment variables or files mounted in a volume.

This resource is particularly useful in several scenarios, including when multiple applications share the same configuration when the configuration data needs to be changed frequently without redeploying Pods, or when you want to organize your Deployments by decoupling operations from data. Instead of managing the configuration for each application separately, a single ConfigMap can be used to store the configuration data, which can then be accessed by multiple applications.

ConfigMaps can be created using the `kubectl` command-line tool or by declaring a ConfigMap object in a manifest file.

7.3.2 ConfigMap for PostgreSQL

In the following steps, we are going to create a ConfigMap for PostgreSQL. The ConfigMap will contain the following files:

- POSTGRES_DB - the name of the database to create
- POSTGRES_USER - the name of the user to create
- POSTGRES_PASSWORD - the password of the user to create

```
1 cat <<EOF > kubernetes/postgres-config.yaml
2 apiVersion: v1
3 kind: ConfigMap
4 metadata:
5   name: postgres-config
6   namespace: postgres
7   labels:
8     app: postgres
9 data:
10  POSTGRES_DB: stateful-flask-db
11  POSTGRES_USER: stateful-flask-user
12  POSTGRES_PASSWORD: stateful-flask-password
13 EOF
```

Apply the ConfigMap:

```
1 kubectl apply -f kubernetes/postgres-config.yaml
```

7.4 Persisting data storage on PostgreSQL

7.4.1 Kubernetes Volumes

A volume is an abstraction layer between the container and physical storage. It allows containers to store and access data independently of the underlying infrastructure (AWS, GCP, Azure, etc.).

Volumes provide a way to store and persist data in a container beyond the lifetime of a Pod and enable data sharing between Pods.

Kubernetes volumes can be created from various sources such as a local disk, a network file system, or a cloud provider's block storage. They can be mounted into a container as a directory or a file, and accessed and manipulated like any other file system.

7.4.2 VolumeClaims

A volume claim is a user's request for storage. It specifies the amount of storage that a Pod should have access to and uses a StorageClass to define how the storage should be provisioned.

7.4.3 StorageClass

The StorageClass is an object that defines the type of storage to be used to dynamically provision a Persistent Volume (PV) in response to a Persistent Volume Claim (PVC) made by a Pod and configured by a user.

Some examples of StorageClass in Kubernetes are:

- aws-ebs: A storage class for Amazon Elastic Block Store (EBS) volumes.
- azure-disk: A storage class for Azure disks.
- csi-cephfs: A storage class for CephFS volumes using the Container Storage Interface (CSI) driver.
- do-block-storage: A type of persistent storage offered by DigitalOcean (DO) that provides SSD-based storage volumes to store data.
- local-storage: A storage class for local storage on a node.

7.4.4 Adding storage to PostgreSQL

To add storage to PostgreSQL, you need to create a `PersistentVolumeClaim` and a `PersistentVolume`.

```
1 cat <<EOF > kubernetes/postgres-pvc-pv.yaml
2 kind: PersistentVolume
3 apiVersion: v1
4 metadata:
5   name: postgres-pv-volume
6   labels:
7     app: postgres
8 spec:
9   storageClassName: do-block-storage
10  capacity:
11    storage: 5Gi
12  accessModes:
13    - ReadWriteMany
14  hostPath:
15    path: "/mnt/data"
16 ---
17 apiVersion: v1
18 kind: PersistentVolumeClaim
19 metadata:
20   name: postgres-pv-claim
21   namespace: postgres
22   labels:
23     app: postgres
24 spec:
25   accessModes:
26     - ReadWriteOnce
27   resources:
28     requests:
29       storage: 5Gi
30   storageClassName: do-block-storage
31 EOF
```

There are two parts in this YAML file:

- PersistentVolume

- The PV is defined with the `kind` and `apiVersion` set to `PersistentVolume` and `v1`, respectively.
 - It has a `name` and `labels` for identification purposes.
 - The `storageClassName` is set to `do-block-storage`, which defines the type of storage class used.
 - The `capacity` is set to `5Gi` and `accessModes` is set to `ReadWriteMany`, which means that the volume can be read and written to by multiple pods at the same time.
 - The `hostPath` specifies the physical path of the volume in the host machine.
- `PersistentVolumeClaim`:
- The PVC is defined with `kind` set to `PersistentVolumeClaim` and `metadata` defining the `name`, `namespace`, and `labels`.
 - The `accessModes` is set to `ReadWriteOnce`, which means the volume can be read and written to by a single pod at a time.
 - The `resources` defines the amount of storage requested for the PVC.
 - Finally, `storageClassName` is set to `do-block-storage`, which corresponds to the storage class used by the PV.

There are different access modes for `PersistentVolumes` and `PersistentVolumeClaims` as per the official documentation:

- **ReadWriteOnce**: The volume can be mounted as read-write by a single node. ReadWriteOnce access mode still can allow multiple pods to access the volume when the pods are running on the same node.
- **ReadOnlyMany**: The volume can be mounted as read-only by many nodes.
- **ReadWriteMany**: The volume can be mounted as read-write by many nodes.
- **ReadWriteOncePod**: The volume can be mounted as read-write by a single Pod. Use ReadWriteOncePod access mode if you want to ensure that only one pod across the whole cluster can read that PVC or write to it. This is only supported for [CSI volumes](#)⁶⁶ and Kubernetes version 1.22+.

Let's create the `PersistentVolume` and `PersistentVolumeClaim`:

⁶⁶<https://kubernetes.io/blog/2019/01/15/container-storage-interface-ga/>

```
1 kubectl apply -f kubernetes/postgres-pvc-pv.yaml
```

You can check the status of the `PersistentVolume` and `PersistentVolumeClaim` with the following commands:

```
1 kubectl get pv
2 kubectl get pvc -n postgres
```

7.4.5 Creating a Deployment for PostgreSQL

We are going to use the following command to create a Deployment for PostgreSQL.

Although not the best resource to use with PostgreSQL, we will use the Deployment for now.

```
1 cat <<EOF > kubernetes/postgres-deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: postgres
6   namespace: postgres
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: postgres
12    template:
13      metadata:
14        labels:
15          app: postgres
16      spec:
17        containers:
18          - name: postgres
19            image: postgres:10.1
```

```
20      imagePullPolicy: Always
21      ports:
22          - containerPort: 5432
23      envFrom:
24          - configMapRef:
25              name: postgres-config
26      volumeMounts:
27          - mountPath: /var/lib/postgresql/data
28              subPath: postgres
29              name: postgredb
30      volumes:
31          - name: postgredb
32      persistentVolumeClaim:
33          claimName: postgres-pv-claim
34 EOF
```

- The `spec` field defines the desired state of the Deployment.
- `replicas` specify the number of replicas we want to create
- `selector` specifies how to select the Pods that the Deployment should manage. In this case, the selector matches the Pods with the `app: postgres` label.
- The `template` field specifies the Pod template used to create new Pods in the Deployment.
- `metadata` contains labels that identify the Pod as a member of the `app: postgres` group.
- `spec` field contains the container configuration.
- The container runs a PostgreSQL image version 10.1, and exposes port 5432 for communication.
- `envFrom` defines an environment variable from a ConfigMap resource named `postgres-config`.
- The container requires a volume for persistent storage, defined in the `volumes` field.
- The volume is named `postgredb` and is backed by a PersistentVolumeClaim (PVC) named `postgres-pv-claim`.
- The `volumeMounts` field in the container configuration specifies the volume and its mount path.

- In this case, the mount path is `/var/lib/postgresql/data` and the subpath is `postgres`, which means that the PostgreSQL database data is stored in the `postgres` directory of the volume.

Let's apply the Deployment:

```
1 kubectl apply -f kubernetes/postgres-deployment.yaml
```

7.4.6 Creating a Service for PostgreSQL

In this section, we will create a service for PostgreSQL. To do this, we will use the following YAML file:

```
1 cat <<EOF > kubernetes/postgres-service.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: postgres # Sets service name
6   namespace: postgres
7   labels:
8     app: postgres # Labels and Selectors
9 spec:
10   type: NodePort # Sets service type
11   ports:
12     - port: 5432 # Sets port to run the postgres application
13   selector:
14     app: postgres
15 EOF
```

Then we will create the service:

```
1 kubectl apply -f kubernetes/postgres-service.yaml
```

7.4.7 Creating a Deployment for our application

We are going to change the application code in a way that it will connect to the PostgreSQL database and store every todo item in the database. To do this, we will use the following YAML file:

Make sure the new virtual environment is activated:

```
1 workon stateful-flask
2 # or mkvirtualenv stateful-flask
```

Then install the required dependencies:

```
1 pip install Flask==2.2.3
2 pip install Flask-SQLAlchemy==3.0.3
3 pip install psycopg2-binary==2.9.6
4 pip install Flask-Migrate==4.0.4
```

Freeze the dependencies:

```
1 pip freeze > requirements.txt
```

Now we will change the application code. We will add the following code to the app.py file:

```
1 cat <<EOF > app/app.py
2 from flask import Flask, jsonify, request
3 from flask_sqlalchemy import SQLAlchemy
4 from flask_migrate import Migrate
5
6 app = Flask(__name__)
7 app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://stateful-flask-us\
8 er:stateful-flask-password@postgres.postgres.svc.cluster.local:5432/sta\
9 teful-flask-db'
10 db = SQLAlchemy(app)
```

```
11 migrate = Migrate(app, db)
12
13 class Task(db.Model):
14     id = db.Column(db.Integer, primary_key=True)
15     title = db.Column(db.String(80), nullable=False)
16     description = db.Column(db.String(200))
17
18 @app.route('/tasks', methods=['GET'])
19 def get_tasks():
20     tasks = Task.query.all()
21     return jsonify({'tasks': [{id: task.id, 'title': task.title, 'des\
22 cription': task.description} for task in tasks]})  

23
24 @app.route('/tasks', methods=['POST'])
25 def create_task():
26     data = request.get_json()
27     title = data['title']
28     description = data['description']
29     task = Task(title=title, description=description)
30     db.session.add(task)
31     db.session.commit()
32     return jsonify({'task': {id: task.id, 'title': task.title, 'descr\
33 iption': task.description}})  

34
35 if __name__ == '__main__':
36     app.run(debug=True, host='0.0.0.0', port=5000)
37 EOF
```

We also need this Dockerfile to build the image:

```
1 cat <<EOF > app/Dockerfile
2 FROM python:3.9-slim-buster
3 WORKDIR /app
4 COPY . /app
5 RUN pip install --no-cache-dir -r requirements.txt
6 EXPOSE 5000
7 CMD ["python", "app.py"]
8 EOF
```

Build the image:

```
1 docker build -t stateful-flask:v0 -f app/Dockerfile app
```

Run the container to test the application:

```
1 docker run -it -p 5000:5000 stateful-flask:v0
```

You should see an error while connecting to the database, which is normal because we haven't deployed the application to the cluster, it's just a local test.

We can re-tag and push the image to Docker Hub:

```
1 docker login
2 # change <dockerhub_username> to your Docker Hub username
3 export DOCKERHUB_USERNAME=<dockerhub_username>
4 docker tag stateful-flask:v0 $DOCKERHUB_USERNAME/stateful-flask:v0
5 docker push $DOCKERHUB_USERNAME/stateful-flask:v0
```

Let's create the Deployment:

```
1 cat <<EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: stateful-flask
12   template:
13     metadata:
14       labels:
15         app: stateful-flask
16     spec:
17       containers:
18         - name: stateful-flask
19           image: eon01/stateful-flask:v0
20         ports:
21           - containerPort: 5000
22           imagePullPolicy: Always
23 EOF
```

I'm using the image `eon01/stateful-flask:v0` which is the image I pushed to Docker Hub. You can use your own image.

Then we will create the Deployment:

```
1 kubectl apply -f kubernetes/deployment.yaml
```

We can check the status of the Deployment:

```
1 kubectl get pods -n stateful-flask
```

Finally, we need to migrate the database:

```
1 export pod=$(kubectl get pods -n stateful-flask -l app=stateful-flask -\\
2 o jsonpath='{.items[0].metadata.name}')
3 kubectl exec -it $pod -n stateful-flask -- flask db init
4 kubectl exec -it $pod -n stateful-flask -- flask db migrate
5 kubectl exec -it $pod -n stateful-flask -- flask db upgrade
```

7.4.8 Creating a Service for our application

We are going to use a ClusterIP service to expose the application to the cluster. To do this, we will use the following YAML file:

```
1 cat <<EOF > kubernetes/stateful-flask-service.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   selector:
9     app: stateful-flask
10  ports:
11    - name: http
12      protocol: TCP
13      port: 5000
14      targetPort: 5000
15 EOF
```

Apply the service:

```
1 kubectl apply -f kubernetes/stateful-flask-service.yaml
```

7.4.9 Creating an external Service for our application

Since the Ingress controller is in a different namespace, we need to create an external service to expose the application to the cluster. To do this, we will use the following YAML file:

```
1 cat <<EOF > kubernetes/stateful-flask-service-externalname.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: stateful-flask-service-externalname
6   namespace: default
7 spec:
8   type: ExternalName
9   externalName: stateful-flask.stateful-flask.svc.cluster.local
10 EOF
```

Apply the service:

```
1 kubectl apply -f kubernetes/stateful-flask-service-externalname.yaml
```

7.4.10 Creating an Ingress for our application

We are going to use the following YAML file to create the Ingress:

```
1 cat <<EOF > kubernetes/ingress.yaml
2 apiVersion: networking.k8s.io/v1
3 kind: Ingress
4 metadata:
5   name: my-ingress
6 spec:
7   rules:
8     - host: stateful-flask.<ip>.nip.io
9       http:
10         paths:
11           - path: /tasks
12             pathType: Prefix
13             backend:
14               service:
15                 name: stateful-flask-service-externalname
```

```
16      port:  
17          number: 5000  
18      ingressClassName: nginx  
19 EOF
```

Make sure to replace <ip> with the IP of the Ingress controller.

Apply the Ingress:

```
1 kubectl apply -f kubernetes/ingress.yaml
```

7.5 Checking logs and making sure everything is working

In order to see the application logs, we can use the `kubectl logs` command. For example, to see the logs of the `hello-world` Pod, we can run:

```
1 kubectl logs hello-world
```

If the Pod is in another namespace, we can specify the namespace with the `-n` flag:

```
1 kubectl logs hello-world -n my-namespace
```

To apply this to `stateful-flask` application, we can run:

```
1 kubectl -n stateful-flask logs <POD_NAME>
```

You will need to replace `<POD_NAME>` with the name of the Pod that you want to see the logs for. Alternatively, you can get the Pod name from the Deployment and use it in the command:

```
1 export pod=$(kubectl -n stateful-flask get pods -l app=stateful-flask -l
2   o jsonpath='{.items[0].metadata.name}')
3 kubectl -n stateful-flask logs $pod
```

However if you have multiple Pods and want to see the logs of all of them, you can use the `-l` flag to specify the label selector:

```
1 kubectl -n stateful-flask logs -l app=stateful-flask
```

You can also use the `-f` flag to follow the logs:

```
1 kubectl -n stateful-flask logs -f -l app=stateful-flask
```

Then we can go to `http://stateful-flask.<ip>.nip.io/tasks` to see the application in action.

To make a POST request to the application and add a task, we can use `curl` (or Postman). For example, to add a task with the title “My first task” and the description “This is my first task”, we can run:

```
1 curl -X POST -H "Content-Type: application/json" -d '{"title":"My first\
2   task", "description":"This is my first task"}' "http://stateful-flask.
3 <ip>.nip.io/tasks"
```

Make sure to replace `http://stateful-flask.<ip>.nip.io` with the URL of your application which can be found in the output of the `kubectl get ingress` command or by executing the following command:

```
1 kubectl get ingress | awk '{print $3}' | tail -n 1
```

You can also combine both commands into one:

```
1 curl -X POST -H "Content-Type: application/json" -d '{"title":"My first\
2   task", "description":"This is my first task"}' "http://$(kubectl get i
3 ngress | awk '{print $3}')/tasks"
```

Now, you can make a GET request to the application to see the tasks that you have added:

```
1 curl "http://$(kubectl get ingress | awk '{print $3}' | tail -n 1)/task\\
2 s"
```

If we delete all Pods of the application and all Pods of the database, we will not lose the data because the data is stored in the `PersistentVolumeClaim`.

To delete all Pods of the application, we can run:

```
1 kubectl delete -f kubernetes/deployment.yaml
2 kubectl delete -f kubernetes/postgres-deployment.yaml
```

Redeploy the application:

```
1 kubectl apply -f kubernetes/deployment.yaml
2 kubectl apply -f kubernetes/postgres-deployment.yaml
```

Now, if we make a GET request to the application, we will see the tasks that we added before:

```
1 curl "http://$(kubectl get ingress | awk '{print $3}' | tail -n 1)/task\\
2 s"
```

7.6 Summary

Even if the Flask application does not store its state internally (stateless), it stores the state of the whole application in the database. Therefore, we need to make sure that the database is persistent.

In this part of the guide, we have seen how to deploy a stateful database to Kubernetes used by a stateless Flask microservice. We have also seen how to use `PersistentVolumeClaim` to store the state of the application in a `PersistentVolume`. Even if the example seems simple, it is a good starting point to understand how to combine stateful and stateless microservices in Kubernetes.

8 Deploying Stateful Microservices: StatefulSets

8.1 What is a StatefulSet?

In simple terms, StatefulSet is a workload API object that manages stateful applications.

When running stateful applications in Kubernetes, it is crucial to ensure stability, scalability, and data integrity. The stateful application must:

- Run in a stable and reliable manner.
- Be scalable up and down without data loss.
- Be upgradable without data loss.
- Be rolled back to a previous version without data loss.
- Be restartable without data loss.

StatefulSets are API objects in Kubernetes that are designed to manage stateful applications such as databases. They provide features such as scalability and upgradeability while preserving the ordering and uniqueness of the Pods.

In summary, StatefulSets are useful for applications that require specific features such as unique and stable network identifiers, persistent storage that remains unchanged when Pods are restarted, and a way to deploy and scale in a specific order.

8.2 StatefulSet vs Deployment

A StatefulSet assigns a unique and persistent identifier to each Pod, unlike a Deployment. This allows the Pod to maintain its identity even when rescheduled.

StatefulSets are particularly well-suited for workloads that require persistence through the use of storage volumes.

The persistent identifiers assigned to the Pods make it easier to match the volumes to the Pods that replace any that have failed, ensuring that data is not lost in case of failure.

If an application does not require unique and stable network identifiers and persistent storage that does not change when Pods are restarted, it is better to use a Deployment instead.

8.3 Creating a StatefulSet

In the previous examples, we deployed a stateful PostgreSQL database using a Deployment. In this example, we will deploy the same database using a StatefulSet instead. The StatefulSet is more suitable for our use case since we are running a persistent PostgreSQL database.

Before proceeding, delete the Deployment and the Service that we created in the previous example.

```
1 cd $HOME/stateful-flask
2
3 kubectl delete -f kubernetes/postgres-deployment.yaml
4 kubectl delete -f kubernetes/postgres-service.yaml
```

We can also delete the Volume and PersistentVolumeClaim that we created in the previous example:

```
1 kubectl delete -f kubernetes/postgres-pvc-pv.yaml
```

Now, create the following StatefulSet manifest file:

```
1 cat <<EOF > kubernetes/postgres-statefulset.yaml
2 apiVersion: apps/v1
3 kind: StatefulSet
4 metadata:
5   name: postgres
6   namespace: postgres
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: postgres
12   template:
13     metadata:
14       labels:
15         app: postgres
16     spec:
17       containers:
18         - name: postgres
19           image: postgres:10.1
20           imagePullPolicy: Always
21         ports:
22           - containerPort: 5432
23         envFrom:
24           - configMapRef:
25             name: postgres-config
26         volumeMounts:
27           - name: postgredb-volume
28             mountPath: /var/lib/postgresql/data
29             subPath: postgres
30         env:
31           - name: PGDATA
32             value: /var/lib/postgresql/data/pgdata
33   volumeClaimTemplates:
34     - metadata:
35       name: postgredb-volume
36     spec:
```

```
37     accessModes: [ "ReadWriteOnce" ]  
38     storageClassName: "do-block-storage"  
39     resources:  
40         requests:  
41             storage: 5Gi  
42 EOF
```

Then, create the StatefulSet:

```
1 kubectl apply -f kubernetes/postgres-statefulset.yaml
```

Check the StatefulSet:

```
1 kubectl get statefulset -n postgres
```

If you delete and recreate the StatefulSet, the Pods will have the same identifiers.

8.4 Creating a Service for the StatefulSet

StatefulSets require a Headless Service. As a reminder, Kubernetes allows clients to discover Pods IPs through DNS lookups and this is possible thanks to the Headless Service. Without a Headless Service, the Pods IPs cannot be discovered directly, instead the DNS server returns a single IP which is the IP of the Service itself. The Service then may load-balances the traffic to the underlying Pods.

This is how we create a Headless Service:

```
1 cat <<EOF > kubernetes/postgres-headless-service.yaml
2 apiVersion: v1
3 kind: Service
4 metadata:
5   name: postgres
6   namespace: postgres
7 spec:
8   clusterIP: None
9   selector:
10      app: postgres
11   ports:
12     - name: postgres
13       port: 5432
14 EOF
```

Then, create the Headless Service:

```
1 kubectl apply -f kubernetes/postgres-headless-service.yaml
```

8.5 Post deployment tasks

Now that we have a new StatefulSet, we need to recreate the database schema and apply any SQL migrations.

```
1 kubectl exec -it <pod> -n stateful-flask -- flask db init
2 kubectl exec -it <pod> -n stateful-flask -- flask db migrate
3 kubectl exec -it <pod> -n stateful-flask -- flask db upgrade
```

Replace <pod> with the name of one of the application's Pods. You can get the name of the Pods with the following command:

```
1 kubectl get pods -n stateful-flask
```

Or you can combine the two commands:

```
1 export pod=$(kubectl get pods -n stateful-flask | awk '{print $1}' | tail -n 1)
2 kubectl exec -it $pod -n stateful-flask -- flask db init
3 kubectl exec -it $pod -n stateful-flask -- flask db migrate
4 kubectl exec -it $pod -n stateful-flask -- flask db upgrade
```

Finally, check that the application is working:

```
1 export url="http://$(kubectl get ingress | awk '{print $3}' | tail -n 1 \
2 )/tasks"
3
4 # add a new task
5 curl -X POST -H "Content-Type: application/json" -d '{"title":"New task\
6 ", "description":"New description"}' $url
7
8 # get all tasks
9 curl $url
```

You can also delete and then recreate the StatefulSet and check that the data is persisted.

```
1 kubectl delete -f kubernetes/postgres-statefulset.yaml
2 kubectl apply -f kubernetes/postgres-statefulset.yaml
3
4 curl $url
```

8.6 StatefulSet vs Deployment: persistent storage

Previously, we created a Deployment that utilized a PersistentVolumeClaim to persist data. Here's how we did it:

```
1 apiVersion: v1
2 kind: PersistentVolumeClaim
3 metadata:
4   name: postgres-pv-claim
5   namespace: postgres
6   labels:
7     app: postgres
8 spec:
9   accessModes:
10    - ReadWriteOnce
11   resources:
12     requests:
13       storage: 5Gi
14   storageClassName: do-block-storage
```

Then we mounted the PersistentVolumeClaim in the Deployment manifest file:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 [...]
4 spec:
5   [...]
6   template:
7     [...]
8     spec:
9       containers:
10      - name: postgres
11        image: postgres:10.1
12        [...]
13       volumes:
14         - name: postgredb
15         persistentVolumeClaim:
16           claimName: postgres-pv-claim
```

The Deployment will work fine, but it's not the best way to persist data. Once we scale the Deployment, the new Pods may not have access to the same

`PersistentVolumeClaim`. This means that the new Pods will not have access to the same data. If you look back at the `PersistentVolumeClaim`, you will notice that we used:

```
1 accessModes:
2   - ReadWriteOnce
```

This means that the `PersistentVolumeClaim` can only be mounted as read-write by a single node. Therefore, the new Pods created on new nodes may not have access to the same `PersistentVolumeClaim`.

What if we change the `accessModes` to `ReadWriteMany`?

```
1 accessModes:
2   - ReadWriteMany
```

In this case, the Deployment will still not be able to persist data properly. Since `ReadWriteMany` allows running across multiple nodes, our application should be able to handle the concurrent read-write of the same file and this is not the case.

In order to avoid these problems and because we don't want to manage data concurrency at the application level, we will use `VolumeClaimTemplates`. This is how we did in the StatefulSet manifest file:

```
1 volumeClaimTemplates:
2   - metadata:
3     name: postgredb-volume
4   spec:
5     accessModes: [ "ReadWriteOnce" ]
6     storageClassName: "do-block-storage"
7     resources:
8       requests:
9         storage: 5Gi
```

The `VolumeClaimTemplates` will request a `PersistentVolumeClaim` from the `StorageClass` dynamically. If you have “x” Pods, the StatefulSet will create “x” `PersistentVolumeClaims`, each with a name in the following format:

```
1 <volumeClaimTemplate-name>-<pod-name>-<ordinal-index>
```

When we delete the StatefulSet, the volumes associated with it will not be deleted. The Pod that was using a PersistentVolumeClaim will reuse the same volume when the StatefulSet is recreated.

This is done to ensure data safety and integrity.

8.7 StatefulSet vs Deployment: associated service

In the very first example of the PostgreSQL Deployment, we create a ClusterIP, this is because we needed to create a Service for the Deployment that is accessible only from within the cluster. Here is how we did this:

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: postgres # Sets service name
5   namespace: postgres
6   labels:
7     app: postgres # Labels and Selectors
8 spec:
9   type: NodePort # Sets service type
10  ports:
11    - port: 5432 # Sets port to run the postgres application
12  selector:
13    app: postgres
```

However, in the StatefulSet manifest file, we not only needed to access the PostgreSQL Pod using a Service, but we also wanted to be able to scale the number of Pods without impacting the data integrity, such as concurrent read-write of the same file.

To solve this issue, we utilized StatefulSet and VolumeClaimTemplates to allow all Pods to share the data directory `/var/lib/postgresql/data`. However, accessing the Pods through a Service remained problematic. The regular Kubernetes implementation of a Service does not work with PostgreSQL because it uses a load balancer or proxy to direct traffic to the Pods. Instead, each Pod must be discovered directly by the client. This is why we employed a Headless Service.

Using a Headless Service ensures that the stateful application functions properly by providing a stable network identity for the database cluster. Even if something unexpected happens, such as the failure of a Pod and the provisioning of a new one with a new IP address, the Headless Service ensures that the new Pod has a stable network identity.

9 Microservices Patterns: Externalized Configurations

9.1 Storing configurations in the environment

One of the main benefits of microservices is the ability to scale and manage each service independently.

However, managing configuration data for multiple services and multiple environments can be challenging. For example, two or more services may share the same configurations, if we decide to change the configuration, we need to do the same for all services using it.

Another challenge comes from the fact that changing hard-coded configurations requires redeploying the code, which involves different steps such as building and deploying and this can become time-consuming when changes are frequent.

Implementing the externalized configuration pattern addresses this problem by storing all application configuration data outside of the codebase. In a [Twelve-factor App](#)⁶⁷, configurations are stored in the environment.

This approach is language- and OS-agnostic and scales smoothly as the app expands into more deployments. By externalizing configuration data, microservices can run in multiple environments without modification or recompilation, making it easier to manage and scale each service independently.

9.2 Kubernetes Secrets and environment variables: why?

In the previous example, we had this code:

⁶⁷<https://12factor.net/config>

```
1 from flask import Flask, jsonify, request
2 from flask_sqlalchemy import SQLAlchemy
3 from flask_migrate import Migrate
4
5 app = Flask(__name__)
6 app.config['SQLALCHEMY_DATABASE_URI'] = 'postgresql://stateful-flask-us\
7 er:stateful-flask-password@postgres.postgres.svc.cluster.local:5432/sta\
8 teful-flask-db'
9 ..etc
10 ..etc
11 ..etc
```

There are three problems in this code:

- The database password is in the code, which is not a good practice
- Whenever we want to change a variable such as the database name, we need to change it in multiple places in our code
- We need to change the code and redeploy the application

These problems can be solved by using environment variables and secrets.

9.3 Kubernetes Secrets and environment variables: how?

Let's start by changing the code and making it use environment variables in the connection string.

Go the application directory:

```
1 cd $HOME/stateful-flask
```

We did this step before, but make sure to install the required dependencies:

```
1 workon stateful-flask
2
3 pip install Flask==2.2.3
4 pip install Flask-SQLAlchemy==3.0.3
5 pip install psycopg2-binary==2.9.6
6 pip install Flask-Migrate==4.0.4
```

Freeze the dependencies:

```
1 pip freeze > requirements.txt
```

Then, change the code to use environment variables:

```
1 cat <<EOF > app/app.py
2 from flask import Flask, jsonify, request
3 from flask_sqlalchemy import SQLAlchemy
4 from flask_migrate import Migrate
5 import os
6
7 app = Flask(__name__)
8 DB_USER = os.environ.get('DB_USER')
9 DB_PASSWORD = os.environ.get('DB_PASSWORD')
10 DB_HOST = os.environ.get('DB_HOST')
11 DB_NAME = os.environ.get('DB_NAME')
12 DB_PORT = os.environ.get('DB_PORT')
13 app.config['SQLALCHEMY_DATABASE_URI'] = "postgresql://{}:{}@{}:{}/{}".format(DB_USER, DB_PASSWORD, DB_HOST, DB_PORT, DB_NAME)
14 db = SQLAlchemy(app)
15 migrate = Migrate(app, db)
16
17
18 class Task(db.Model):
19     id = db.Column(db.Integer, primary_key=True)
20     title = db.Column(db.String(80), nullable=False)
21     description = db.Column(db.String(200))
22
```

```

23 @app.route('/tasks', methods=['GET'])
24 def get_tasks():
25     tasks = Task.query.all()
26     return jsonify({'tasks': [ {'id': task.id, 'title': task.title, 'des\
27 cription': task.description} for task in tasks] })
28
29 @app.route('/tasks', methods=['POST'])
30 def create_task():
31     data = request.get_json()
32     title = data['title']
33     description = data['description']
34     task = Task(title=title, description=description)
35     db.session.add(task)
36     db.session.commit()
37     return jsonify({'task': { 'id': task.id, 'title': task.title, 'descr\
38 iption': task.description}} )
39
40 if __name__ == '__main__':
41     app.run(debug=True, host='0.0.0.0', port=5000)
42 EOF

```

The following line we updated in our Flask app sets the URI for Flask's SQLAlchemy database connection.

```

1 app.config['SQLALCHEMY_DATABASE_URI'] = "postgresql://{}:{}@{}:{}/{}".f\
2 ormat(DB_USER, DB_PASSWORD, DB_HOST, DB_PORT, DB_NAME)

```

It uses the string formatting method to create the URI using variables defined earlier in the code (DB_USER, DB_PASSWORD, DB_HOST, DB_PORT, and DB_NAME). The resulting URI has the format:

```
1 postgresql://<user>:<password>@<host>:<port>/<database>
```

where each of the variables is replaced with the corresponding value.

Now, build and push after changing <DOCKERHUB_USERNAME> with your Docker Hub username:

```
1 export DOCKERHUB_USERNAME=<DOCKERHUB_USERNAME>
2 docker build -t stateful-flask:v0 -f app/Dockerfile app
3 docker tag stateful-flask:v0 $DOCKERHUB_USERNAME/stateful-flask:v0
4 docker push $DOCKERHUB_USERNAME/stateful-flask:v0
```

Now, let's create a Secret for the database user and password.

Kubernetes Secrets are objects that allow you to store and manage sensitive information, such as passwords, tokens, and keys. They provide a secure way to store sensitive data, rather than hard-coding it in your application or configuration files.

We will use the same user and password as before.

When creating a Secret in Kubernetes, we need to encode the data using base64.

We can do this using the echo command and the | base64 pipe.

```
1 echo -n 'stateful-flask-user' | base64
2 echo -n 'stateful-flask-password' | base64
```

The result should be:

```
1 c3RhdGVmdWwtZmxhc2stdXN1cg==
2 c3RhdGVmdWwtZmxhc2stcGFzc3dvcmQ=
```

Kubernetes secrets are stored in an encoded format for security reasons. This is because Secrets can contain unrecognizable characters such as new lines or special characters ..etc. The echo -n followed by the | base64 command encodes the data in base64 format without any unrecognizable character.

However, it is important to note that the sensitive data is not encrypted. It is **only encoded** in base64, which is a reversible encoding. To encrypt the data, we will describe a Secret resource:

```
1 cat <<EOF > kubernetes/stateful-flask-secret.yaml
2 apiVersion: v1
3 kind: Secret
4 metadata:
5   name: stateful-flask-secret
6   namespace: stateful-flask
7 type: Opaque
8 data:
9   DB_USER: c3RhGdGVmdWwtZmxhc2stdXN1cg==
10  DB_PASSWORD: c3RhGdGVmdWwtZmxhc2stcGFzc3dvcmQ=
11 EOF
```

Then create the Secret object:

```
1 kubectl apply -f kubernetes/stateful-flask-secret.yaml
```

To use these Secret, we need to update the Deployment manifest to use the Secret as environment variables.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: stateful-flask
5   namespace: stateful-flask
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10    app: stateful-flask
11   template:
12     metadata:
13       labels:
14         app: stateful-flask
15   spec:
16     containers:
```

```
17      - name: stateful-flask
18        image: eon01/stateful-flask:v0
19        imagePullPolicy: Always
20        ports:
21          - containerPort: 5000
22        env:
23          - name: DB_USER
24            valueFrom:
25              secretKeyRef:
26                name: stateful-flask-secret
27                key: DB_USER
28          - name: DB_PASSWORD
29            valueFrom:
30              secretKeyRef:
31                name: stateful-flask-secret
32                key: DB_PASSWORD
```

- In this code, we defined environment variables for the new Deployment. Specifically, two environment variables are created, `DB_USER` and `DB_PASSWORD`.
- Instead of directly setting their values, the `valueFrom` field is used to retrieve their values from a Kubernetes Secret called `stateful-flask-secret`.
- The `secretKeyRef` field specifies that the value of the environment variable should be retrieved from a key in the `stateful-flask-secret` Secret, where `DB_USER` and `DB_PASSWORD` are the keys for the respective values.

Now, we are going to add other environment variables to the Deployment manifest. These variables will be used to connect to the database: `DB_HOST`, `DB_NAME`, and `DB_PORT`.

```
1 cat <<EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: stateful-flask
12 template:
13   metadata:
14     labels:
15       app: stateful-flask
16   spec:
17     containers:
18       - name: stateful-flask
19         image: eon01/stateful-flask:v0
20         imagePullPolicy: Always
21       ports:
22         - containerPort: 5000
23       env:
24         - name: DB_USER
25           valueFrom:
26             secretKeyRef:
27               name: stateful-flask-secret
28               key: DB_USER
29       - name: DB_PASSWORD
30           valueFrom:
31             secretKeyRef:
32               name: stateful-flask-secret
33               key: DB_PASSWORD
34       - name: DB_HOST
35           value: postgres.postgres.svc.cluster.local
36       - name: DB_NAME
```

```
37      value: stateful-flask-db
38    - name: DB_PORT
39      value: "5432"
40 EOF
```

These variables are not as sensitive as the database user and password, so we can hard-code them in the Deployment manifest. It is also possible to store them as Secrets, but we will not do that in this guide.

By default Kubernetes Secrets are required, if they are not found, the Pod will not start. However, we can make the Secret optional by setting the `optional` field to `true`.

```
1 env:
2   - name: DB_USER
3     valueFrom:
4       secretKeyRef:
5         name: stateful-flask-secret
6         key: DB_USER
7         optional: true
```

This is not the case for the `DB_USER` and `DB_PASSWORD` variables, because they are required to connect to the database and the application will not work without them. That's why we did not set the `optional` field to `true`.

Now, let's create the Deployment:

```
1 kubectl apply -f kubernetes/deployment.yaml
```

You can also view the Secret and Deployment using the `kubectl get` command:

```
1 kubectl get secret,deployment -n stateful-flask
```

You can now access the application using the Ingess IP address:

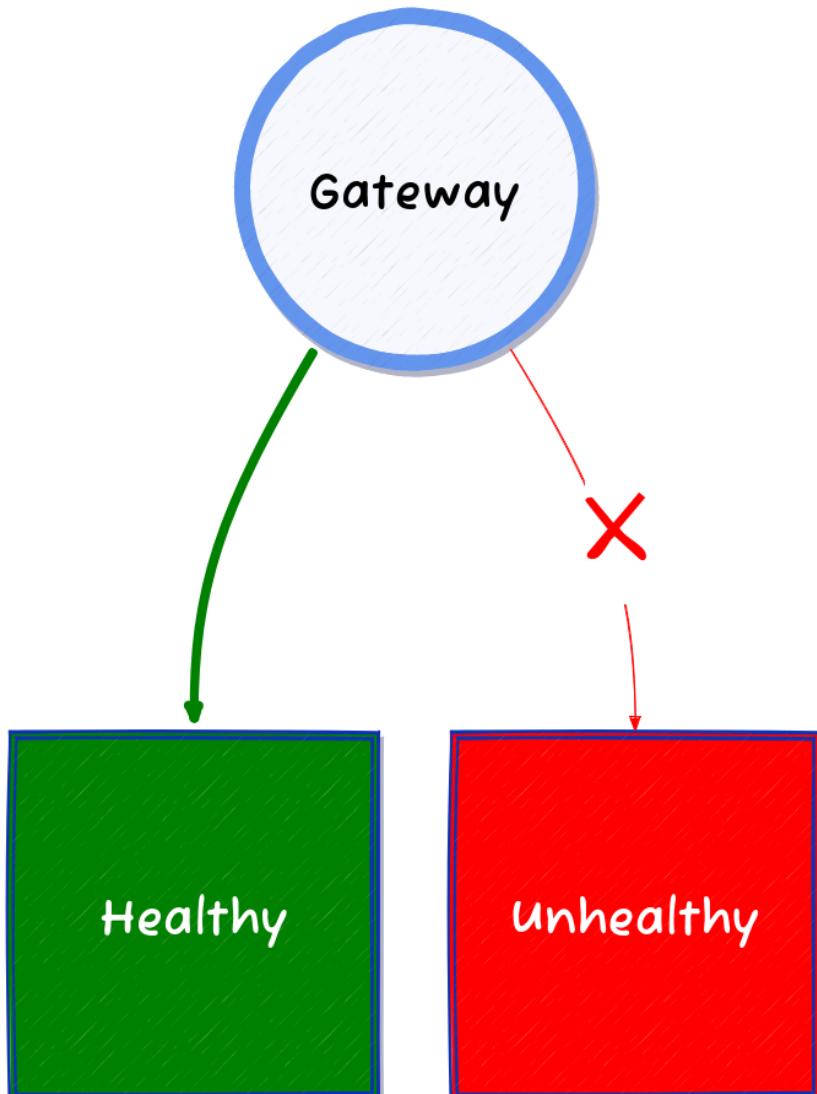
```
1 curl http://$(kubectl get ingress | awk '{print $3}' | tail -n 1)/tasks
```

10 Best Practices for Microservices: Health Checks

10.1 Health Checks

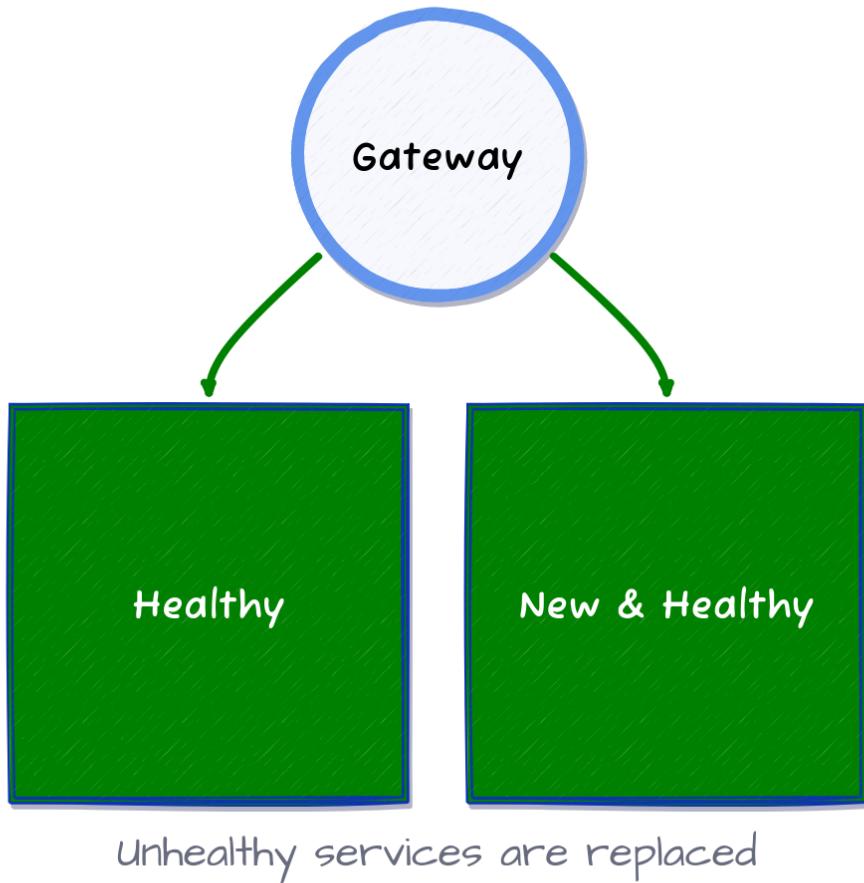
One of the best practices in developing microservices is to implement health checks.

Health checks are a way to monitor the health of a microservice. They are used by the infrastructure to determine if a microservice is healthy or not. If a microservice is not healthy, it is cut off from the network and no longer receives requests.



The health check pattern

Resilient systems use health checks to detect, remove unhealthy microservices and replace them with new ones.



Health checks are also used by the infrastructure to determine if a microservice is ready to receive requests. This is important when a microservice is starting up. The infrastructure will not send requests to a microservice until it is ready to receive them. This is why there are different types of health checks and concepts

to take into consideration when implementing them such as giving a microservice time to start up before considering it unhealthy or shutting a microservice down after a certain amount of failed health checks..etc

Fortunately, Kubernetes provides a way to implement health checks. Kubernetes health checks are implemented using probes and this is what we are going to cover in this part.

10.2 Liveness and Readiness probes

Kubernetes health checks are implemented using probes. There are two types of probes: liveness and readiness.

The liveness probe is used to determine if a container within a Pod is running properly. Kubernetes periodically executes the liveness probe and if it succeeds, the container is considered alive. If it fails, the container is considered dead and Kubernetes restarts it.

The readiness probe is used to determine if a container within a Pod is ready to receive requests. Same as the liveness probe, Kubernetes periodically executes the readiness probe and if it succeeds, the container is considered ready. If it fails, the container is considered not ready yet.

10.3 Types of probes

There are three types of probes: HTTP, TCP and Exec.

- **HTTP probes** are used to determine if a container is alive or ready by sending an HTTP request to a specific endpoint. If the endpoint returns any code greater than or equal to 200 and less than 400 the probe succeeds. If the endpoint returns any other code, the probe fails.
- **TCP probes** are used to determine if a container is alive or ready by trying to establish a TCP connection to a specific port. If the connection is established, the probe succeeds. If the connection fails, the probe fails.

- **Exec probes** are used to determine if a container is alive or ready by executing a command inside the container. If the command returns with a zero exit code, the probe succeeds. If the command returns with a non-zero exit code, the probe fails.

The following table summarizes the 3 types of probes:

Probe Type	Description
HTTP Probe	Sends an HTTP request to a specified endpoint.
TCP Probe	Opens a TCP connection to a specified port.
Exec Probe	Executes a specified command inside the container.

10.4 Implementing probes

Probes are implemented in the Pod specification.

Here is an example of a Pod specification with a liveness and readiness HTTP probes:

```
1 kubectl apply -f - <<EOF
2 ---
3 # create a temporary namespace to perform our test
4 apiVersion: v1
5 kind: Namespace
6 metadata:
7   name: healthcheck
8 ---
9 # create a Deployment with liveness and readiness probes
10 apiVersion: apps/v1
11 kind: Deployment
12 metadata:
13   name: stateless-flask
14 spec:
```

```
15   replicas: 1
16   selector:
17     matchLabels:
18       app: stateless-flask
19   template:
20     metadata:
21       labels:
22         app: stateless-flask
23     spec:
24       containers:
25         - name: stateless-flask
26           image: eon01/stateless-flask
27         ports:
28           - containerPort: 5000
29         livenessProbe:
30           httpGet:
31             path: /tasks
32             port: 5000
33             initialDelaySeconds: 5
34             periodSeconds: 5
35             timeoutSeconds: 3
36             successThreshold: 1
37             failureThreshold: 3
38         readinessProbe:
39           httpGet:
40             path: /tasks
41             port: 5000
42             initialDelaySeconds: 10
43             periodSeconds: 15
44             timeoutSeconds: 10
45             successThreshold: 1
46             failureThreshold: 15
47             # terminationGracePeriodSeconds: 10
48 EOF
```

The liveness and readiness probes are configured using the `livenessProbe` and

`readinessProbe` fields and both are using HTTP probes.

- The `initialDelaySeconds` field is used to specify the number of seconds to wait before executing the first probe. The default value is 0 seconds.
- The `periodSeconds` field is used to specify the number of seconds to wait between each probe. The default value is 10 seconds.
- The `timeoutSeconds` field is used to specify the number of seconds to wait for a probe to succeed before considering it failed. The default value is 1 second.
- The `successThreshold` field is used to specify the number of consecutive successful probes before considering the container alive or ready. The default value is 1.
- The `failureThreshold` field is used to specify the number of consecutive failed probes before considering the container dead or not ready. The default value is 3.
- The `terminationGracePeriodSeconds` field is used to specify the number of seconds to wait before terminating a container after it receives a SIGTERM signal. This can be set at the Pod level or at the container's probe level. If set at the probe level, it overrides the value set at the Pod level.

Here is a table that summarizes the different fields:

Field	Description
<code>initialDelaySeconds</code>	Number of seconds to wait before executing the first probe.
<code>periodSeconds</code>	Number of seconds to wait between each probe.
<code>timeoutSeconds</code>	Number of seconds to wait for a probe to succeed before considering it failed.
<code>successThreshold</code>	Number of consecutive successful probes before considering the container alive or ready.

Field	Description
failureThreshold	Number of consecutive failed probes before considering the container dead or not ready.
terminationGracePeriodSeconds	Number of seconds to wait before terminating a container after it receives a SIGTERM signal.

The other types of probes are configured the same way. Here is an example of a Pod specification with a liveness and readiness TCP probes:

```

1 [ ... ]
2   livenessProbe:
3     tcpSocket:
4       port: 5000
5   readinessProbe:
6     tcpSocket:
7       port: 5000
8 [ ... ]

```

This is useful for microservices that don't expose an HTTP endpoint but listen on a specific port.

Here is an example of a Pod specification with a liveness and readiness Exec probes:

```

1 [ ... ]
2   livenessProbe:
3     exec:
4       command:
5         - cat
6         - /tmp/healthy
7   readinessProbe:
8     exec:
9       command:
10      - cat

```

```
11      - /tmp/healthy  
12  [..]
```

This is useful for microservices that require more complex checks. For example, a microservice should be considered not ready until the database it's using is ready. In this case, a script can be used to check if the database is ready and return a zero exit code if it is and a non-zero exit code if it's not.

Example:

```
1  #!/bin/bash  
2  
3  # wait for the database to be ready  
4  while ! mysqladmin ping -h mysql -u root -p$MYSQL_ROOT_PASSWORD --silent; do  
5    sleep 1  
6  done  
7  
8  # return a zero exit code if the database is ready  
9  exit 0
```

Then the script can be used in the readiness probe:

```
1  [..]  
2  readinessProbe:  
3    exec:  
4      command:  
5        - /bin/bash  
6        - /tmp/wait-for-db.sh  
7  [..]
```

11 Microservices Resource Management Strategies

11.1 Resource management and risks: from Docker to Kubernetes

If you have used standalone Docker containers, you are probably aware of the risks associated with not properly allocating memory and CPU resources. If you have not, you may be surprised when your containers crash due to running out of memory or CPU.

Containers running on a host should not consume excessive memory, as the kernel may activate an Out Of Memory exception and start killing processes, including the Docker daemon itself. Docker attempts to prevent this by adjusting the OOM priority on the daemon, making it less likely to be killed. However, this does not apply to container processes, which are more likely to be killed.

It is therefore recommended to limit container access to the host's resources. Here are some examples:

```
1 # guarantees the container at most 50% of the CPU every second.  
2 docker run -it --cpus=".5" ubuntu /bin/bash  
3  
4 # limit the maximum amount of memory the container can use to 256 megab\\  
5 ytes.  
6 docker run -it --memory=256m ubuntu /bin/bash
```

If the node where a Pod is running has enough of a resource available, it's possible (and allowed) for a container to use more resources than its request for that resource specifies. However, a container is not allowed to use more than its resource limit.

11.2 Requests and limits

If the node on which a Pod is running has enough of a resource available, a container can use more of that resource than its request specifies. However, a container is not allowed to use more than its resource limit.

The Kubernetes scheduler uses requests and limits to determine which node to place a Pod on. It attempts to fit the total of the resource requests for all scheduled containers onto available nodes. For example, if a Pod has a CPU request of 100m and a memory request of 200Mi, the scheduler will not place it on a node with less than 100m of CPU and 200Mi of memory available.

Following is an example of how we can set resource requests and limits for a container in a Pod:

```
1 cd $HOME/stateful-flask
2
3 cat << EOF > kubernetes/deployment.yaml
4 apiVersion: apps/v1
5 kind: Deployment
6 metadata:
7   name: stateful-flask
8   namespace: stateful-flask
9 spec:
10  replicas: 1
11  selector:
12    matchLabels:
13      app: stateful-flask
14  template:
15    metadata:
16      labels:
17        app: stateful-flask
18    spec:
19      containers:
20        - name: stateful-flask
21          image: eon01/stateful-flask:v0
```

```
22      resources:
23        requests:
24          memory: "64Mi"
25          cpu: "250m"
26        limits:
27          memory: "128Mi"
28          cpu: "500m"
29      imagePullPolicy: Always
30    ports:
31      - containerPort: 5000
32    env:
33      - name: DB_USER
34        valueFrom:
35          secretKeyRef:
36            name: stateful-flask-secret
37            key: DB_USER
38      - name: DB_PASSWORD
39        valueFrom:
40          secretKeyRef:
41            name: stateful-flask-secret
42            key: DB_PASSWORD
43      - name: DB_HOST
44        value: postgres.postgres.svc.cluster.local
45      - name: DB_NAME
46        value: stateful-flask-db
47      - name: DB_PORT
48        value: "5432"
49 EOF
```

Apply the deployment:

```
1 kubectl apply -f kubernetes/deployment.yaml
```

In the above example, we set the following resource requests and limits:

- 250m CPU request and 500m CPU limit

- 64Mi memory request and 128Mi memory limit

11.3 CPU resource units

CPU resources are measured in CPU units. One CPU, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors.

For instance, a container with a CPU request of `100m` is guaranteed 100 millicpu on each node it is scheduled on. Meanwhile, a container with a CPU request of `1` is guaranteed 1 CPU on each node where it is scheduled.

Note that we can also use `0.1` instead of `100m`, or `0.01` instead of `10m`. We can also use `1000m` instead of `1`.

Example:

```
1 resources:  
2   requests:  
3     cpu: "250m"
```

is equivalent to:

```
1 resources:  
2   requests:  
3     cpu: "0.25"
```

Both mean that the container is guaranteed 250 millicpu on each node that it is scheduled on. In other words, the allocated CPU is 25% of a CPU. When there are multiple CPU requests, the scheduler sums them up to get the total CPU request for the Pod.

In our previous example, we set the CPU to 250m. This means that the container is guaranteed 250 millicpu on each node it is scheduled on.

We can set higher CPU requests and limits, for example:

```
1 resources:
2   requests:
3     cpu: "1.5"
4   limits:
5     cpu: "3"
```

This means that the container is guaranteed 1.5 CPU on each node that it is scheduled on, and can use up to 3 CPUs on each node that it is scheduled on. If we have 2 CPUs on each node, the container will be able to use up to 150% of the total CPU on each node and if we have 4 CPUs on each node, the container will be able to use up to 75% of the total CPU on each node.

Now if you request more than what your node has, the scheduler will not place the Pod on that node. For example, if you have a node with 2 CPUs and your CPU requests is 3, the scheduler will not place the Pod on that node. If you have another node with 4 CPUs, the scheduler will *probably* place the Pod on that node. This is because the Kubelet and other Kubernetes components are running on the node need CPU resources as well.

When no node is available, the Pod will be in a Pending state.

This is an example requesting 10 CPUs cores:

```
1 cat << EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: stateful-flask
12   template:
13     metadata:
14       labels:
```

```
15      app: stateful-flask
16      spec:
17          containers:
18              - name: stateful-flask
19                  image: eon01/stateful-flask:v0
20          resources:
21              requests:
22                  cpu: "10"
23              limits:
24                  cpu: "20"
25          imagePullPolicy: Always
26          ports:
27              - containerPort: 5000
28          env:
29              - name: DB_USER
30                  valueFrom:
31                      secretKeyRef:
32                          name: stateful-flask-secret
33                          key: DB_USER
34              - name: DB_PASSWORD
35                  valueFrom:
36                      secretKeyRef:
37                          name: stateful-flask-secret
38                          key: DB_PASSWORD
39              - name: DB_HOST
40                  value: postgres.postgres.svc.cluster.local
41              - name: DB_NAME
42                  value: stateful-flask-db
43              - name: DB_PORT
44                  value: "5432"
45 EOF
```

If you apply it with `kubectl apply -f kubernetes/deployment.yaml`, you will see that the Pod will be in a Pending state.

11.4 Memory resource units

Memory resources are measured in bytes.



A byte is a unit of memory size and it's equal to 8 bits. A bit is a unit of data and it's equal to either a 0 or 1. Byte is the smallest addressable unit of memory.

You can express memory using one of these suffixes: E, P, T, G, M, K. Where:

- E = Exabyte (1,000,000,000,000,000,000 bytes)
- P = Petabyte (1,000,000,000,000,000 bytes)
- T = Terabyte (1,000,000,000,000 bytes)
- G = Gigabyte (1,000,000,000 bytes)
- M = Megabyte (1,000,000 bytes)
- K = Kilobyte (1,000 bytes)

Or using the following suffixes: Ei, Pi, Ti, Gi, Mi, Ki. Where:

- Ei = Exbibyte (1,152,921,504,606,846,976 bytes)
- Pi = Pebibyte (1,125,899,906,842,624 bytes)
- Ti = Tebibyte (1,099,511,627,776 bytes)
- Gi = Gibibyte (1,073,741,824 bytes)
- Mi = Mebibyte (1,048,576 bytes)
- Ki = Kibibyte (1,024 bytes)

You can use the suffix `m` to mean `milli`. For example, `100m` means 100 milli, which is $100/1000 = 0.1$. So `100m` is equivalent to `0.1` bytes.

Let's, for example, create a Pod with a container requesting 64Mi of memory and limiting it to 128Mi:

```
1 cat << EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: stateful-flask
12   template:
13     metadata:
14       labels:
15         app: stateful-flask
16     spec:
17       containers:
18         - name: stateful-flask
19           image: eon01/stateful-flask:v0
20       resources:
21         requests:
22           memory: "64Mi"
23         limits:
24           memory: "128Mi"
25       imagePullPolicy: Always
26     ports:
27       - containerPort: 5000
28     env:
29       - name: DB_USER
30       valueFrom:
31         secretKeyRef:
32           name: stateful-flask-secret
33           key: DB_USER
34       - name: DB_PASSWORD
35       valueFrom:
36         secretKeyRef:
```

```
37         name: stateful-flask-secret
38         key: DB_PASSWORD
39     - name: DB_HOST
40         value: postgres.postgres.svc.cluster.local
41     - name: DB_NAME
42         value: stateful-flask-db
43     - name: DB_PORT
44         value: "5432"
45 EOF
```

11.5 Considerations when setting resource requests and limits

Independently of Kubernetes, it is important to understand that memory and CPU have two different behaviors when it comes to resource management. Consider the following example:

- CPU (the computing power) is like time, regenerated every cycle. If you don't use it, you lose it. It is also called a compressible resource. A compressible resource means that if its usage reaches the limit, any process that needs to use it **will first need to wait** for the CPU to be available.
- Memory is similar to space in that it is a non-cyclical and incompressible resource. This means that if memory usage reaches its limit, any process that requires it **will not be able to run**.

When a container requests a certain amount of memory, it does not mean that the container will use all of it. It will only be guaranteed that amount of memory when multiple containers are running on the system. The container can use less than that amount.

In other words, the container runtime use requests as a hint to set `memory.min` and `memory.low` of the container using [cgroups v2](#)⁶⁸.

⁶⁸<https://docs.kernel.org/admin-guide/cgroup-v2.html>



cgroup or control group is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

If a container exceeds its memory request, the Pod it belongs to may be evicted if the node runs out of memory.

If a container exceeds its memory limit, the container will be terminated.

However, a container can exceed its CPU limit for a certain amount of time without being evicted. This temporary increase in CPU usage is called a CPU burst.



The CPU burst is a period of time when the container is allowed to use more CPU than its limit. The CPU burst is only limited to a reasonable amount of time.

If you do not set a CPU limit for a container, one of two situations may occur:

1 - The container may have no limit on the amount of CPU resources it can use, **potentially consuming all available CPU resources on the node where it is running.** 2 - The container will inherit a default CPU limit if the Namespace it is running in has one set. This default value can be set by cluster administrators using a Kubernetes resource called a LimitRange.



The LimitRange is a Kubernetes resource that defines the default values for CPU and memory limits for a Namespace.

This is an example of a LimitRange for the Namespace `stateful-flask`:

```
1 apiVersion: v1
2 kind: LimitRange
3 metadata:
4   name: stateful-flask-limit-range
5   namespace: stateful-flask
6 spec:
7   limits:
8     - default:
9       memory: 512Mi
10      cpu: 500m
11     defaultRequest:
12       memory: 256Mi
13       cpu: 250m
14     type: Container
```

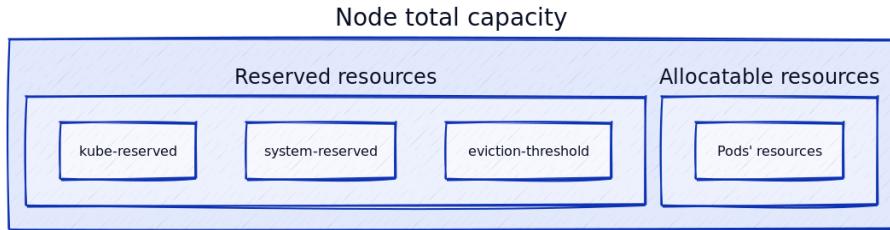
11.6 Node reserve resources vs allocatable resources

In a Kubernetes cluster, each node has resources that are reserved for the operating system and Kubernetes components. These resources are referred to as **node reserve** resources. The remaining resources are known as **allocatable resources**.

The node reserve resources are meant for components such as:

- The operating system
- The kubelet
- The kube-proxy
- The container runtime
- The node-level logging and monitoring agents
- The node-level network plugins

These resources should not be used by Pods. Pods have their own resources called **allocatable resources**.



kube-reserved: These resources are reserved for Kubernetes system daemons, such as the kubelet or container runtime, to ensure their availability and stability. They are not meant to reserve resources for system daemons that are run as pods. The amount of resources reserved is typically a function of pod density on the nodes.



system-reserved: These resources are reserved for OS system daemons, such as sshd or udev.



eviction-threshold: To avoid system OOMs, the kubelet provides out-of-resource management by reserving some memory. The kubelet attempts to evict Pods whenever memory availability on the node drops below the reserved value. Resources reserved for evictions are not available for Pods.

With these concepts in mind, we can conclude that allocatable resources are the total resources minus the node reserve resources. Resource allocation for containers is done based on the allocatable resources only.

11.7 Quality of Service (QoS) classes

Kubernetes employs Quality of Service (QoS) classes to specify the performance characteristics of a Pod. The class of a Pod is determined by the resource requests and limits of its containers.

These are the three QoS classes that Kubernetes supports:

- Guaranteed
- Burstable
- BestEffort

11.7.1 Guaranteed

To call a Pod Guaranteed, all of its containers must have the following characteristics:

- Memory limit and a memory request are set.
- Memory limit is equal to the memory request.
- CPU limit and a CPU request are set.
- CPU limit is equal to the CPU request.

Example:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: guaranteed-pod
5 spec:
6   containers:
7     - name: guaranteed-container
8       image: eon01/stateful-flask:v0
9       resources:
10         requests:
```

```
11     memory: "64Mi"
12     cpu: "250m"
13   limits:
14     memory: "64Mi"
15     cpu: "250m"
```

Guaranteed is the strictest QoS class. When a Pod is Guaranteed, it means that the container runtime will ensure that the Pod never exceeds its resource requests; if it does, the Pod will be killed.

11.7.2 Burstable

To call a Pod Burstable when they satisfy the following criteria:

- The Pod is not Guaranteed. This means that `requests` value is different from the `limits` value for CPU and memory in all containers.
- At least one container has a memory or a CPU request or limit.

This is an example:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: guaranteed-pod
5 spec:
6   containers:
7     - name: guaranteed-container
8       image: eon01/stateful-flask:v0
9       resources:
10         requests:
11           memory: "64Mi"
12             cpu: "250m"
13         limits:
14           memory: "128Mi"
15             cpu: "500m"
```

Another example:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: guaranteed-pod
5 spec:
6   containers:
7     - name: guaranteed-container
8       image: eon01/stateful-flask:v0
9     resources:
10       requests:
11         memory: "64Mi"
```

Or

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: burstable-pod
5 spec:
6   containers:
7     - name: burstable-container
8       image: eon01/stateful-flask:v0
9     resources:
10       requests:
11         cpu: 100m
12       limits:
13         cpu: 500m
```

When a Pod is specified as “Burstable,” it means that it has some minimum resource guarantees based on the request, but does not require a specific limit. If no limit is set, the limit is assumed to be the capacity of the Node, which allows the Pod to use more resources if available.

Burstable Pods are evicted only after all BestEffort Pods are evicted in case of Node resource pressure. Since a Burstable Pod can have a Container with no resource limits or requests, it can try to use any amount of Node resources.

11.7.3 BestEffort

To call a Pod BestEffort, all of its containers must have the following characteristic:

- The Pod does not have any memory or CPU requests or limits.

The BestEffort QoS class is the lowest priority in Kubernetes. Pods in this class can use any resources that are not specifically assigned to other QoS classes. For example, if a node has 16 CPU cores and a Pod in the Guaranteed QoS class is assigned 4 of those cores, then any Pod in the BestEffort class can use the remaining 12 CPU cores.

However, if a node becomes under resource pressure, the kubelet will prefer to evict BestEffort Pods first as they are considered the lowest priority. This means that Pods in this class have the lowest resource guarantee and may be the first to be evicted if there is a shortage of resources on the node.

Example:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: besteffort-pod
5 spec:
6   containers:
7     - name: besteffort-container
8       image: eon01/stateful-flask:v0
```

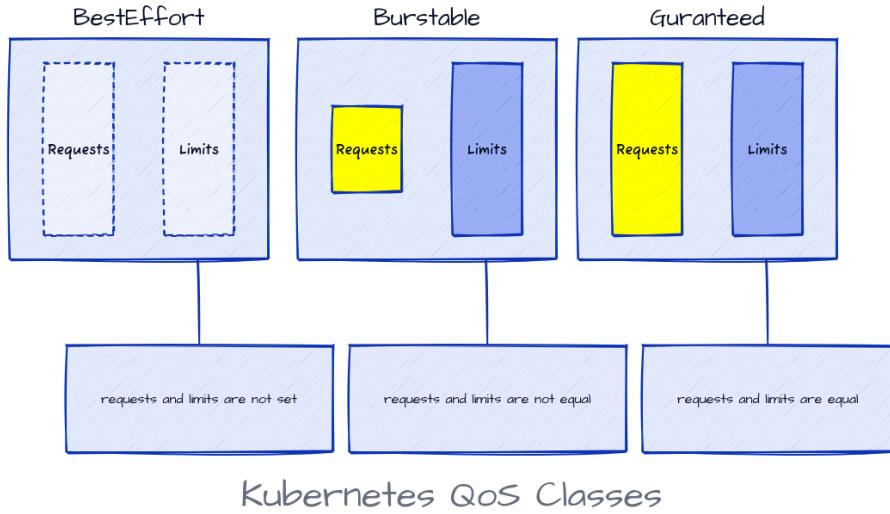
11.7.4 QoS class of a Pod

To check the QoS class of a Pod, you can use the following command:

```
1 kubectl get pods <pod-name> -o jsonpath="{ .status.qosClass}"
```

The result will be one of the following:

- Guaranteed
- Burstable
- BestEffort



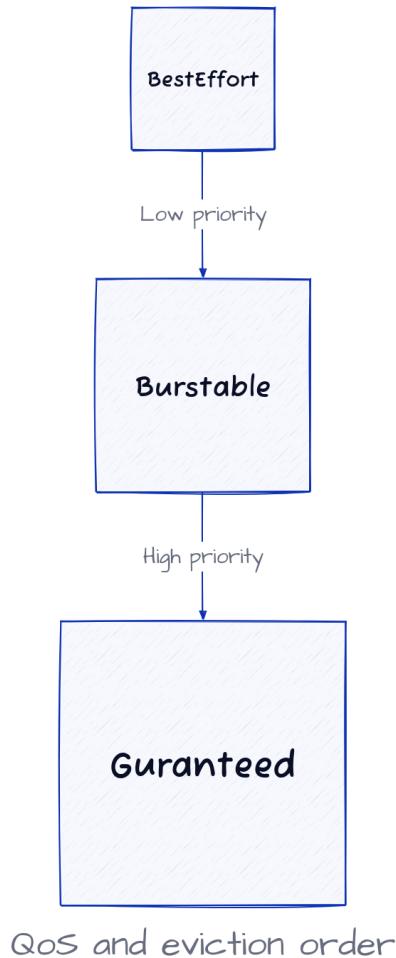
11.7.5 Eviction order

As seen before, there are 3 classes of QoS. When a node is under resource pressure, the kubelet will evict Pods in the following order:

1. BestEffort
2. Burstable
3. Guaranteed

Having this in mind, it is important to understand the eviction order to avoid unexpected behavior. For example, if you have a Pod with a QoS class of Guaranteed

and a Pod with a QoS class of BestEffort, and the node is under resource pressure, the Pod with the QoS class of BestEffort will be evicted first.



11.7.6 PriorityClass: a custom class

PriorityClass is a resource that can be used to specify the priority of a Pod. It determines the order in which Pods are evicted when a node is under resource pressure.

PriorityClass is a non-namespaced resource in the `scheduling.k8s.io` API group.

This is an example:

```
1 apiVersion: scheduling.k8s.io/v1
2 kind: PriorityClass
3 metadata:
4   name: high-priority
5   value: 1000000
6 globalDefault: false
7 description: "This priority class is used for critical pods only such a\
8 s our monitoring agents"
9 preemptionPolicy: PreemptLowerPriority
```

The example above creates a PriorityClass named `high-priority` with the following characteristics:

- `value`: The priority value of the PriorityClass. This value must be an integer between -2147483648 and 1000000000.
- `globalDefault`: If set to true, this PriorityClass will be used as the default PriorityClass for all Pods that do not have a `PriorityClassName` set. You can only have one PriorityClass with this field set to true.
- `description`: A description of the PriorityClass.
- `preemptionPolicy`: The preemption policy of the PriorityClass. This can be set to `PreemptLowerPriority` or `Never`. This is used to determine whether the Pod with the PriorityClass can preempt other Pods that have a lower priority. This configuration is available since Kubernetes 1.24.

The following example is a lower priority class:

```
1 apiVersion: scheduling.k8s.io/v1
2 kind: PriorityClass
3 metadata:
4   name: low-priority
5 value: 1000
6 globalDefault: false
7 description: "This priority class is used for non-critical pods only su\
8 ch as our internal tools"
9 preemptionPolicy: Never
```

To apply a PriorityClass to a Pod, you can use the `priorityClassName` field in the Pod spec:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: stateful-flask
5   namespace: stateful-flask
6 spec:
7   priorityClassName: high-priority
8   containers:
9     - name: stateful-flask
10    image: eon01/stateful-flask:v0
11    resources:
12      requests:
13        memory: "64Mi"
14        cpu: "250m"
15      limits:
16        memory: "128Mi"
17        cpu: "500m"
```

12 Autoscaling Microservices in Kubernetes: Introduction

12.1 Best practices for microservices scalability

When designing microservices, scalability should be a key consideration. This requires implementing a set of best practices. We are going to see the most important ones in the following sections.

12.1.1 Use a service registry for service discovery

A service registry is a central directory that enables services to discover and communicate with each other. This ensures effective communication among microservices, even as the number of services grows.

In other words, adding a new microservice to the system should not require any changes to the existing microservices. The service registry should be able to handle the new microservice automatically.

In Kubernetes, cluster DNS implements this functionality. When a Service or Pod is created, Kubernetes creates DNS records for it.

Example: When creating a LoadBalancer Service named `service-name` in the `namespace-name` namespace, all Pods under this Service can be accessed by the DNS name `service-name.namespace-name.svc.cluster.local`. When scaling up the Deployment, the DNS record will always point to the current set of Pods, regardless of the node they are running on. This includes all Pods under the named Service.

12.1.2 Implement health checks

Health checks are used to determine the condition of a microservice. They indicate whether a microservice is ready to receive requests or if it is unhealthy and needs to be restarted.

When scaling up a microservice, it typically takes some time to start. During this time, the microservice is not yet ready to receive requests. Therefore, when scaling up, the new instances should not be automatically considered healthy. They should be considered unhealthy until they are ready to receive requests.

This can be managed in Kubernetes by implementing a Readiness probe. A Readiness probe is a health check that determines if a microservice is ready to receive requests. If it fails, the Pod will not receive any traffic from the Service.

There is also another useful health check called a Liveness probe. A Liveness probe determines if a microservice is still running. If it fails, the Pod will be restarted.

We will discuss these probes in more detail in the next sections.

12.1.3 Designing for scalability and other best practices

When designing microservices, scalability should be a key consideration. This requires implementing a set of best practices like:

- **Stateless services:** Prioritize stateless services over stateful services since they are easier to scale while maintaining high availability. This means that your microservices should not store any state in memory. Instead, they should store it in a database or other external storage.
- **Idempotent services:** Idempotent microservices are required for scalability. This means that your microservices should be able to handle the same request multiple times without causing any side effects. This property is required for scalability.
- **Asynchronous communication:** Favor asynchronous communication over synchronous communication. This means that your microservices should not wait for a response from another service before continuing. Instead, they should send a message and continue processing.

- **Caching:** Services with caching performs better in a dynamic auto-scaling environment. Since stateless services do not store any state in memory, they need to retrieve it from a database or other external storage. This can be slow and expensive. Therefore, it is recommended to cache the data in memory to improve performance and reduce costs.
- **Load balancing:** Services should be load balanced.
- **Fault tolerance:** Fault tolerance is required for scalability. This means that your microservices should be able to handle failures gracefully. For example, if a microservice fails, it should not affect other microservices. Instead, it should be restarted automatically.
- **Observability:** Build observable services. This means that your microservices should expose metrics and logs. This will help you to identify bottlenecks and other issues.

There are several best practices that can be applied to microservices scalability. Fortunately, building microservices for Kubernetes, forces us to create Kubernetes-native and Kubernetes-ready workloads. In consequence, multiple best practices are already implemented for us.

In the next sections, we will discuss these best practices.

13 Autoscaling Microservices in Kubernetes: Horizontal Autoscaling

13.1 Horizontal scaling

At this point, we have the following YAML for the API:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: stateful-flask
5   namespace: stateful-flask
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10       app: stateful-flask
11   template:
12     metadata:
13       labels:
14         app: stateful-flask
15   spec:
16     containers:
17       - name: stateful-flask
18         image: eon01/stateful-flask:v1
19         resources:
20           requests:
21             memory: "64Mi"
```

```
22      cpu: "250m"
23      limits:
24          memory: "128Mi"
25          cpu: "500m"
26      imagePullPolicy: Never
27      ports:
28          - containerPort: 5000
29      env:
30          - name: DB_USER
31              valueFrom:
32                  secretKeyRef:
33                      name: stateful-flask-secret
34                      key: DB_USER
35          - name: DB_PASSWORD
36              valueFrom:
37                  secretKeyRef:
38                      name: stateful-flask-secret
39                      key: DB_PASSWORD
40          - name: DB_HOST
41              value: postgres.postgres.svc.cluster.local
42          - name: DB_NAME
43              value: stateful-flask-db
44          - name: DB_PORT
45              value: "5432"
```

And the following StatefulSet for the database:

```
1  apiVersion: apps/v1
2  kind: StatefulSet
3  metadata:
4    name: postgres
5    namespace: postgres
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10        app: postgres
11    template:
12      metadata:
13        labels:
14          app: postgres
15      spec:
16        containers:
17          - name: postgres
18            image: postgres:10.1
19            imagePullPolicy: Always
20          ports:
21            - containerPort: 5432
22        envFrom:
23          - configMapRef:
24            name: postgres-config
25        volumeMounts:
26          - mountPath: /var/lib/postgresql/data
27            subPath: postgres
28            name: postgredb-volume
29      volumeClaimTemplates:
30        - metadata:
31          name: postgredb-volume
32        spec:
33          accessModes: [ "ReadWriteOnce" ]
34          storageClassName: "do-block-storage"
35        resources:
36          requests:
```

```
37     storage: 5Gi
```

If you list all the Pods in all namespaces, you will see that we have 1 instance of the API and 2 instances of the database:

```
1 kubectl get pods -A
2 #or kubectl get pods --all-namespaces
```

We have already scaled the database horizontally. If you look at the StatefulSet definition, you will see that we have two replicas.

```
1 spec:
2   replicas: 2
```

In order to scale the API horizontally, we need to change the Deployment definition. For example to have 3 replicas, use the following:

```
1 spec:
2   replicas: 3
```

This is the full YAML code to use:

```
1 cat <<EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: stateful-flask
12   template:
13     metadata:
```

```
14     labels:
15         app: stateful-flask
16     spec:
17         containers:
18             - name: stateful-flask
19                 image: eon01/stateful-flask:v1
20             resources:
21                 requests:
22                     memory: "64Mi"
23                     cpu: "250m"
24                 limits:
25                     memory: "128Mi"
26                     cpu: "500m"
27             imagePullPolicy: Never
28         ports:
29             - containerPort: 5000
30         env:
31             - name: DB_USER
32                 valueFrom:
33                     secretKeyRef:
34                         name: stateful-flask-secret
35                         key: DB_USER
36             - name: DB_PASSWORD
37                 valueFrom:
38                     secretKeyRef:
39                         name: stateful-flask-secret
40                         key: DB_PASSWORD
41             - name: DB_HOST
42                 value: postgres.postgres.svc.cluster.local
43             - name: DB_NAME
44                 value: stateful-flask-db
45             - name: DB_PORT
46                 value: "5432"
47 EOF
```

Apply and watch the changes:

```
1 kubectl apply -f kubernetes/deployment.yaml  
2 kubectl get pods -n stateful-flask -w
```

The ReplicaSet is the object that will create the Pods, you can see the ReplicaSet by running the following command:

```
1 kubectl get rs -n stateful-flask  
2 # or kubectl get replicsets -n stateful-flask
```

 A ReplicaSet is created with certain attributes. These include a selector that describes the criteria for Pods to be managed, the number of replicas specifying how many Pods should be maintained, and a pod template that defines the configuration of new Pods to meet the required number of replicas.

The ReplicaSet is responsible for generating and removing Pods as necessary to achieve the desired number based on the current state of the system.

Using the command line it is also possible to scale the Deployment:

```
1 kubectl scale --replicas=3 deployment/stateful-flask -n stateful-flask
```

13.2 Horizontal Pod Autoscaler

Kubernetes offers automatic scaling as a feature, allowing a workload resource like a StatefulSet or Deployment to adjust to changing demand. Known as Horizontal Pod Autoscaling (HPA), this feature scales the workload up or down based on metrics monitored by the HPA controller.

The HPA adds more Pods to the workload to respond to increased load.

To use HPA, a metric server must be running in the cluster. This server is responsible for aggregating resource usage data across the cluster. The metric server collects metrics from various resources and exposes them through the Kubernetes

apiserver via the Metrics API. These metrics are used by HPA and later by the Vertical Pod Autoscaler (VPA).

There are different ways to install the metric server:

1- Directly from YAML:

```
1 kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/rele\
2 ases/latest/download/components.yaml
```

2- Using Helm:

```
1 helm repo add metrics-server https://kubernetes-sigs.github.io/metrics-\
2 server/
3 helm repo update
4 helm install metrics-server metrics-server/metrics-server
```

3- Using [the DigitalOcean Marketplace](#)⁶⁹. If you are using another Cloud provider, you may use the option provided by your Cloud provider if available.

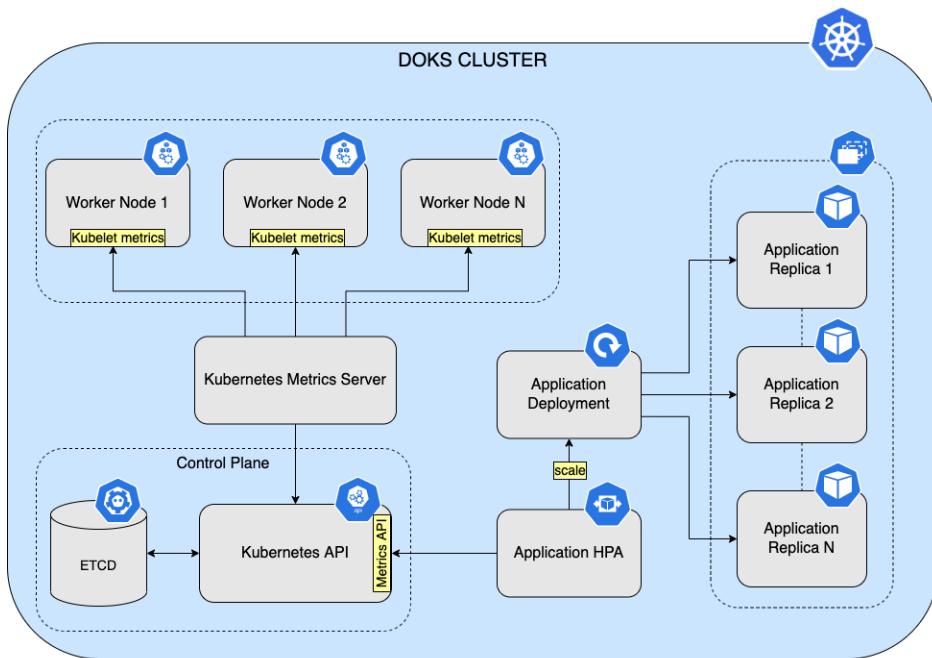
We are going to use the first option. Run:

```
1 kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/rele\
2 ases/latest/download/components.yaml
```

The following diagram [from the official documentation of DigitalOcean](#)⁷⁰ shows an overview of how HPA works in conjunction with metrics-server in a DigitalOcean Kubernetes cluster:

⁶⁹<https://marketplace.digitalocean.com/apps/kubernetes-metrics-server>

⁷⁰<https://marketplace.digitalocean.com/apps/kubernetes-metrics-server>



Overview:

- Deploy Metrics Server to your DOKS cluster.
- Metrics server scrapes kubelet metrics from each worker node, collecting CPU and memory utilization data for each application workloads.
- Metrics server exposes CPU and memory usage metrics via the Kubernetes API server.
- Horizontal Pod Autoscaler fetches CPU and memory usage metrics, via the Kubernetes API server. Then, based on metrics observation and target threshold, decides when to scale up or down your application deployment Pods.

https://raw.githubusercontent.com/digitalocean/marketplace-kubernetes/master/stacks/metrics-server/assets/images/arch_hpa.png

To check if the metric server is running, run the following command:

```
1 kubectl top nodes
```

Next, we will create a Horizontal Pod Autoscaler (HPA) for the API. The HPA will scale the API based on CPU usage. If the CPU usage exceeds 20%, the HPA will add more Pods. If the CPU usage falls below 20%, the HPA will remove Pods.

This is an example:

```
1 cat <<EOF > kubernetes/hpa.yaml
2 apiVersion: autoscaling/v2
3 kind: HorizontalPodAutoscaler
4 metadata:
5   name: stateful-flask-hpa
6   namespace: stateful-flask
7 spec:
8   scaleTargetRef:
9     apiVersion: apps/v1
10    kind: Deployment
11    name: stateful-flask
12   minReplicas: 1
13   maxReplicas: 10
14   metrics:
15     - type: Resource
16       resource:
17         name: cpu
18       target:
19         type: Utilization
20         averageUtilization: 20
21 EOF
```

Apply the HPA:

```
1 kubectl apply -f kubernetes/hpa.yaml
```

You can see the HPA by running the following command:

```
1 kubectl get hpa -n stateful-flask
2 # or kubectl get horizontalpodautoscalers -n stateful-flask
```

Let's stress the API to see how the HPA works. Run the following command:

```
1 kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart\
2 =Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://stateful-
3 flask.stateful-flask.svc.cluster.local:5000/tasks; done"
```

The command above creates a Pod that stresses the API, causing an increase in CPU usage. Open another terminal and run the following command to observe how the HPA is functioning:

```
1 kubectl get hpa -n stateful-flask -w
```

The Horizontal Pod Autoscaler (HPA) adds more Pods when the CPU usage exceeds 20% of the requested CPU. Conversely, the HPA removes Pods when the CPU usage falls below 20% of the requested CPU. Therefore, it is important to set requests for your Pods when using the HPA.

By default, metrics are checked every 15 seconds.

We can also scale based on memory usage. This is an example:

```
1 cat <<EOF > kubernetes/hpa.yaml
2 apiVersion: autoscaling/v2
3 kind: HorizontalPodAutoscaler
4 metadata:
5   name: stateful-flask-hpa-memory
6   namespace: stateful-flask
7 spec:
8   scaleTargetRef:
9     apiVersion: apps/v1
10    kind: Deployment
11    name: stateful-flask
12   minReplicas: 1
13   maxReplicas: 10
14   metrics:
15   - type: Resource
16     resource:
17       name: memory
18     target:
19       type: Utilization
20       averageUtilization: 20
21 EOF
```

Or based on both CPU and memory usage:

```
1 cat <<EOF > kubernetes/hpa.yaml
2 apiVersion: autoscaling/v2
3 kind: HorizontalPodAutoscaler
4 metadata:
5   name: stateful-flask-hpa-cpu-memory
6   namespace: stateful-flask
7 spec:
8   scaleTargetRef:
9     apiVersion: apps/v1
10    kind: Deployment
11    name: stateful-flask
12   minReplicas: 1
13   maxReplicas: 10
14   metrics:
15   - type: Resource
16     resource:
17       name: cpu
18     target:
19       type: Utilization
20       averageUtilization: 20
21   - type: Resource
22     resource:
23       name: memory
24     target:
25       type: Utilization
26       averageUtilization: 20
27 EOF
```

13.3 Autoscaling based on custom Kubernetes metrics

The HPA can also scale based on custom metrics. This is an example:

```
1 cat <<EOF > kubernetes/hpa.yaml
2 apiVersion: autoscaling/v2
3 kind: HorizontalPodAutoscaler
4 metadata:
5   name: stateful-flask-hpa-custom
6   namespace: stateful-flask
7 spec:
8   scaleTargetRef:
9     apiVersion: apps/v1
10    kind: Deployment
11    name: stateful-flask
12   minReplicas: 1
13   maxReplicas: 10
14   metrics:
15     - type: Resource
16       resource:
17         name: cpu
18       target:
19         type: Utilization
20         averageUtilization: 20
21 EOF
```

In the example above, we are scaling based on CPU usage. We instruct the HPA to scale when CPU usage exceeds 20% of the requested CPU. Instead of using Utilization, we can opt to use AverageValue or Value.

- The AverageValue is calculated by dividing the value returned by the metrics server by the number of Pods in the scale target.
- Value is the raw value as returned by the metrics server.

This is an example with AverageValue:

```
1 cat <<EOF > kubernetes/hpa.yaml
2 apiVersion: autoscaling/v2
3 kind: HorizontalPodAutoscaler
4 metadata:
5   name: stateful-flask-hpa-custom
6   namespace: stateful-flask
7 spec:
8   scaleTargetRef:
9     apiVersion: apps/v1
10    kind: Deployment
11    name: stateful-flask
12   minReplicas: 1
13   maxReplicas: 10
14   metrics:
15     - type: Resource
16       resource:
17         name: cpu
18       target:
19         type: AverageValue
20         averageValue: 45m
21 EOF
```

When `target.type` is set to `AverageValue` or `Value`, we need to use values (e.g. `45m`), unlike the previous example where we used a percentage (`20`).

Kubernetes supports the following metric types by default:

- `cpu`: the CPU usage of the Pods in the scale target.
- `memory`: the memory usage of the Pods in the scale target.

They are both called Resource metrics.

Kubernetes also supports other non-resource metrics types:

- `Object`: the number of objects in a Kubernetes object store, as reported by a metrics adapter.
- `Pods`: the number of Pods in the scale target.
- `External`: a metric from an external metrics provider.

13.4 Autoscaling based on more specific custom Kubernetes metrics

We can also scale based on custom kubernetes metrics types like Pods and Object.

This is a Pod metric example:

```
1 apiVersion: autoscaling/v2
2 kind: HorizontalPodAutoscaler
3 metadata:
4   name: stateful-flask-hpa-custom
5   namespace: stateful-flask
6 spec:
7   scaleTargetRef:
8     apiVersion: apps/v1
9     kind: Deployment
10    name: stateful-flask
11    minReplicas: 1
12    maxReplicas: 10
13    metrics:
14      - type: Pods
15        pods:
16          metric:
17            name: custom_metric
18            target:
19              type: AverageValue
20              averageValue: 10k
```

Pod metrics describe Pods as the name suggests. They are always averaged together across Pods and compared with a target value to determine the number of Pods to scale.

They are similar to Resource metrics, except that they only support a target type of AverageValue.

This is an Object metric example:

```
1 apiVersion: autoscaling/v2
2 kind: HorizontalPodAutoscaler
3 metadata:
4   name: stateful-flask-hpa-custom
5   namespace: stateful-flask
6 spec:
7   scaleTargetRef:
8     apiVersion: apps/v1
9     kind: Deployment
10    name: stateful-flask
11   minReplicas: 1
12   maxReplicas: 10
13   metrics:
14   - type: Object
15     object:
16       metric:
17         name: custom_metric
18   describedObject:
19     apiVersion: networking.k8s.io/v1
20     kind: Ingress
21     name: my-ingress
22   target:
23     type: Value
24     value: 10M
```

These metrics describe other object types within the same namespace, as opposed to describing Pods. Unlike Pod metrics, object metrics support two types of targets: `Value` and `AverageValue`.

When using `Value`, the target value is compared directly to the metric returned by the API. In contrast, when using `AverageValue`, the value returned from the custom metrics API is divided by the number of Pods and then compared to the target value.

All custom metrics should be provided by an external metric provider who is not part of Kubernetes. The metric provider is responsible for gathering the metric values and making them available to Kubernetes.

There are many providers available, including Prometheus, Datadog, and New Relic.

To use [Prometheus](#)⁷¹ as an example, you first need to install and configure it to scrape the metric `custom_metric` from your application or another resource in your cluster. Then, install the [Prometheus Adapter](#)⁷². This adapter can replace the metric server used before, as it implements Custom, Resource, and External metrics APIs.

Note that not all metrics providers support all metric types, and not all metric providers can replace the metric server. However, it is possible to keep the metric server and use other metric providers at the same time.

13.5 Using multiple metrics

We can use multiple metrics in the same HPA. This is an example:

```
1 apiVersion: autoscaling/v2
2 kind: HorizontalPodAutoscaler
3 metadata:
4   name: stateful-flask-hpa-custom
5   namespace: stateful-flask
6 spec:
7   scaleTargetRef:
8     apiVersion: apps/v1
9     kind: Deployment
10    name: stateful-flask
11    minReplicas: 1
12    maxReplicas: 10
13    metrics:
14      - type: Resource
15        resource:
16          name: cpu
17          target:
```

⁷¹<https://prometheus.io/>

⁷²<https://github.com/kubernetes-sigs/prometheus-adapter>

```
18      type: Utilization
19      averageUtilization: 20
20  - type: Pods
21    pods:
22      metric:
23        name: custom_metric
24      target:
25        type: AverageValue
26        averageValue: 10k
```

In this case, the Horizontal Pod Autoscaler (HPA) will consider each metric in turn and scale based on the metric that requires the most replicas. For example, if CPU usage is at 20% and the average value of the `custom_metric` is at 5k, the HPA will scale based on the CPU usage.

13.6 Autoscaling based on custom non-Kubernetes metrics

In previous sections, we covered scaling based on default Kubernetes metrics such as CPU and memory (Resource) as well as custom Kubernetes metrics like Pods and Objects. All of these metrics are provided by the Metric Server, with Resource metrics being provided by Kubernetes (specifically, the kubelet) and Pod and Object metrics being provided by an external metric provider like Prometheus. However, it's also possible to scale based on non-Kubernetes metrics, such as metrics from an external application or service. In this section, we'll explore this type of scaling.

We can also scale based on custom non-Kubernetes metrics like `custom_metric` in the following example:

```
1 apiVersion: autoscaling/v2
2 kind: HorizontalPodAutoscaler
3 metadata:
4   name: stateful-flask-hpa-custom
5   namespace: stateful-flask
6 spec:
7   scaleTargetRef:
8     apiVersion: apps/v1
9     kind: Deployment
10    name: stateful-flask
11    minReplicas: 1
12    maxReplicas: 10
13    metrics:
14    - type: External
15      external:
16        metric:
17          name: custom_metric
18          selector:
19            matchLabels:
20              app: external-app
21          target:
22            type: AverageValue
23            averageValue: 30
```

In the example above, we are configuring an External metric named `custom_metric` to be used for scaling. The HPA will use the average value (`type: AverageValue`) of this metric to determine whether to scale up or down. Specifically, it will compare the desired average value (30 in this case) with the current average value of the collected metrics. If the current average value is greater than the desired average value, the HPA will scale up. Otherwise, it will scale down.

Note that the metric section should be defined using a label selector to identify the relevant metric. In this case, it would be `app: external-app`. If multiple time series are returned by a metric provider, the HPA will filter them using the label selector. The HPA will then use the average value of the remaining time series to determine whether to scale up or down.

The available types of metrics for External metrics are `Value` and `AverageValue`.

13.7 Cluster autoscaler

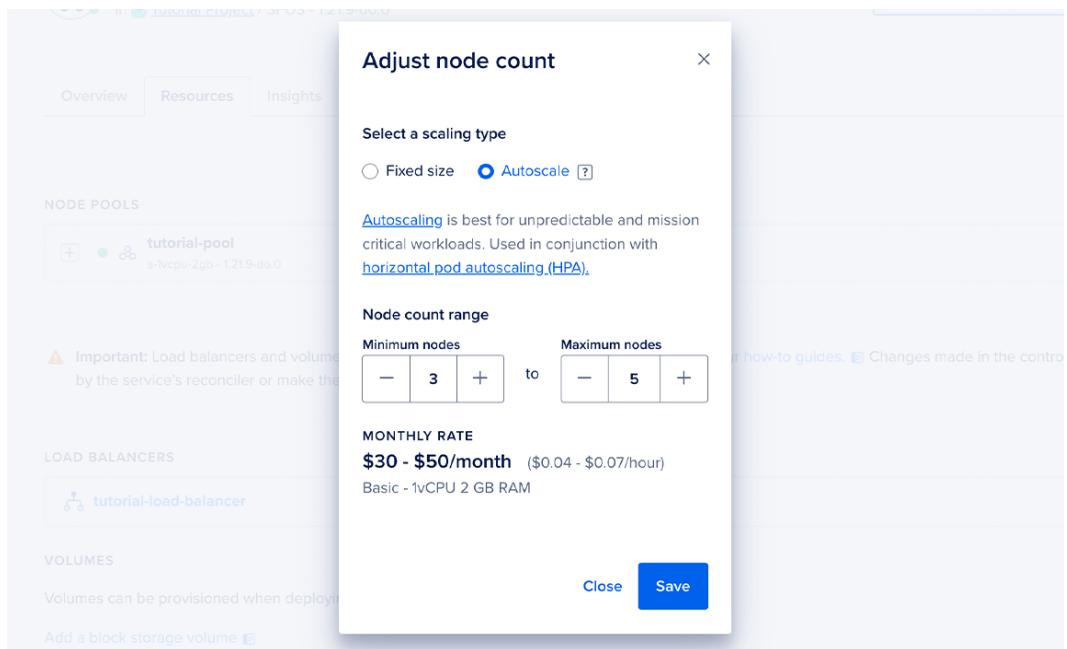
The Cluster Autoscaler (CA) is a tool that automatically adjusts the size of a Kubernetes cluster in response to the following conditions:

- Pods fail to run in the cluster due to insufficient resources, and
- Nodes in the cluster remain underutilized for an extended period of time, and their Pods can be placed on other existing nodes. and their Pods can be placed on other existing nodes

The Cluster Autoscaler is a separate component and has no relation with the HPA, but it could be considered as a horizontal autoscaler for the cluster itself. Also it is not part of the Kubernetes control plane, it is deployed as a Deployment in the `kube-system` namespace.

Most cloud providers have their own implementation of the Cluster Autoscaler and most cases you should use the one provided by your cloud provider since it is optimized for the cloud provider's infrastructure, it is easier to install (usually from the CLI or the web interface) and it is easier to configure.

For example, we're using DigitalOcean, we can use the web interface to install and configure the Cluster Autoscaler.



Or the CLI:

```
1 doctl kubernetes cluster node-pool update mycluster mypool --auto-scale \
2   --min-nodes 1 --max-nodes 10
```

where:

- `mycluster` is the name of the cluster
- `mypool` is the name of the node pool
- `--auto-scale` enables the Cluster Autoscaler
- `--min-nodes` is the minimum number of nodes that the Cluster Autoscaler will scale down to
- `--max-nodes` is the maximum number of nodes that the Cluster Autoscaler will scale up to

You can disable it using:

```
1 doctl kubernetes cluster node-pool update mycluster mypool --auto-scale\\
2 =false
```

These are the official docs for the Cluster Autoscaler for the supported cloud providers:

- GCE⁷³
- GKE⁷⁴
- AWS⁷⁵
- Azure⁷⁶
- AliCloud⁷⁷
- BaiduCloud⁷⁸
- BizflyCloud⁷⁹
- Brightbox⁸⁰
- CherryServers⁸¹
- Civo⁸²
- CloudStack⁸³
- ClusterAPI⁸⁴
- DigitalOcean⁸⁵
- Exoscale⁸⁶
- Equinix Metal⁸⁷
- External gRPC⁸⁸

⁷³<https://kubernetes.io/docs/concepts/cluster-administration/cluster-management/>

⁷⁴<https://cloud.google.com/container-engine/docs/cluster-autoscaler>

⁷⁵<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md>

⁷⁶<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/azure/README.md>

⁷⁷<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/alicloud/README.md>

⁷⁸<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/baiducloud/README.md>

⁷⁹<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/bizflycloud/README.md>

⁸⁰<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/brightbox/README.md>

⁸¹<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/cherryservers/README.md>

⁸²<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/civo/README.md>

⁸³<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/cloudstack/README.md>

⁸⁴<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/clusterapi/README.md>

⁸⁵<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/digitalocean/README.md>

⁸⁶<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/exoscale/README.md>

⁸⁷<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/packet/README.md>

⁸⁸<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/externalgrpc/README.md>

- Hetzner⁸⁹
- HuaweiCloud⁹⁰
- IonosCloud⁹¹
- Kamatera⁹²
- Linode⁹³
- Magnum⁹⁴
- OracleCloud⁹⁵
- OVHcloud⁹⁶
- Rancher⁹⁷
- Scaleway⁹⁸
- TencentCloud⁹⁹
- Vultr¹⁰⁰

⁸⁹<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/hetzner/README.md>

⁹⁰<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/huaweicloud/README.md>

⁹¹<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/ionoscloud/README.md>

⁹²<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/kamatera/README.md>

⁹³<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/linode/README.md>

⁹⁴<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/magnum/README.md>

⁹⁵<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/oci/README.md>

⁹⁶<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/ovhcloud/README.md>

⁹⁷<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/rancher/README.md>

⁹⁸<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/scaleway/README.md>

⁹⁹<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/tencentcloud/README.md>

¹⁰⁰<https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/vultr/README.md>

14 Autoscaling Microservices in Kubernetes: Vertical Scaling

14.1 Vertical Scaling

Vertical scaling is different from horizontal scaling, which involves adding more Pods to handle increased demand. Vertical scaling is useful for workloads that require more resources per Pod, such as CPU or memory-intensive applications.

In other words, vertical scaling in Kubernetes means increasing the resources allocated to an individual Pod, such as CPU or memory, to meet an increased demand.

The easiest way (but not the most practical) to do this is to edit the Pod's resource limits and requests in the Pod specification.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: stateful-flask
5   namespace: stateful-flask
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10    app: stateful-flask
11   template:
12     metadata:
13       labels:
14         app: stateful-flask
15   spec:
```

```
16     containers:
17       - name: stateful-flask
18         image: eon01/stateful-flask:v0
19       resources:
20         requests:
21           memory: "64Mi"
22           cpu: "250m"
23         limits:
24           memory: "128Mi"
25           cpu: "500m"
```

The above example shows how to set the resource `limits` and `requests` for a Pod. When we have high demand, we can edit the Pod specification to increase these resources:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: stateful-flask
5   namespace: stateful-flask
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10    app: stateful-flask
11   template:
12     metadata:
13       labels:
14         app: stateful-flask
15     spec:
16       containers:
17         - name: stateful-flask
18           image: eon01/stateful-flask:v0
19           resources:
20             requests:
21               memory: "128Mi"
```

```
22      cpu: "500m"  
23      limits:  
24          memory: "256Mi"  
25          cpu: "1000m"
```

In this example, we have doubled the resource `limits` and `requests` for the Pod. When the Pod is updated, Kubernetes will terminate the old Pod and create a new one with the new resource limits and requests.

In Kubernetes, resource `requests` and `limits` are set at the container level within a Pod, rather than at the Pod or replica level. When a Pod is created, each container in the Pod can have its own CPU and memory `requests` and `limits` set.

As soon as these values change, Kubernetes scheduler determines which nodes to schedule the Pod onto and how much resources should be allocated to the Pod on each node.

14.2 Vertical Pod Autoscaler

The manual operation described can also be automated using the Vertical Pod Autoscaler (VPA). This autoscaler is actively developed by [SIG-Autoscaling](#)¹⁰¹, the group responsible for the Horizontal Pod Autoscaler in Kubernetes.

There are different ways to install the VPA.

1- Using the GitHub repository:

```
1 # clone the repository  
2 git clone https://github.com/kubernetes/autoscaler.git  
3 # change directory  
4 cd autoscaler/vertical-pod-autoscaler  
5 # install the VPA  
6 ./hack/vpa-up.sh
```

2 - Using your Cloud Provider. For example, if you are using GKE, you can install the VPA using the following command:

¹⁰¹<https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler>

```
1 gcloud container clusters update CLUSTER_NAME --enable-vertical-pod-autoscaling
```

If you are using DigitalOcean, you can install the VPA using the following command:

```
1 doctl kubernetes cluster node-pool update CLUSTER_NAME NODE_POOL_NAME -auto-scale --min-nodes 1 --max-nodes 10
```

The same can be done using the web interface of your cloud provider. This is usually provided as an option when creating a cluster or after the cluster is created.

We are going to use the first method to install the VPA.

```
1 # clone the repository
2 git clone
3 # change directory
4 cd autoscaler/vertical-pod-autoscaler
5 # install the VPA
6 ./hack/vpa-up.sh
```

The VPA is installed in the `kube-system` namespace. You can check the status of the VPA using the following command:

```
1 kubectl get pods -n kube-system | grep vpa
```

There are 3 components that make up the VPA:

- `vpa-admission-controller`
- `vpa-recommender`
- `vpa-updater`

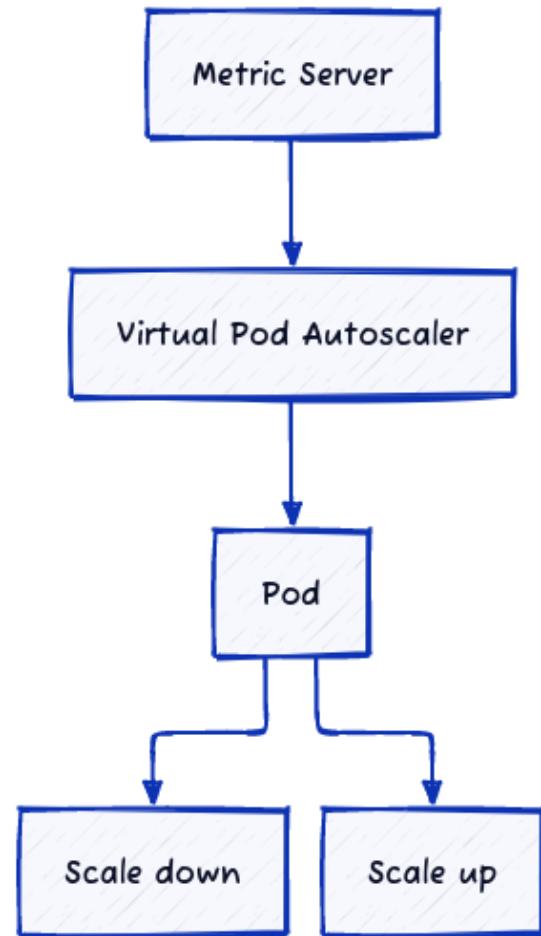
The first component is a binary that registers itself as a [Mutating Admission Webhook](#)¹⁰². It intercepts all Pod creations. Whenever a new Pod is created,

¹⁰²<https://kubernetes.io/docs/reference/access-authn-authz/extensible-admission-controllers/>

this component receives a request from the API Server. It then checks whether a matching VPA configuration is available. If there is no matching configuration, it takes no action. However, if there is a matching configuration, it uses the current recommendation to set the resource requests in the Pod.

The VPA Recommender is the second component and is responsible for generating recommendations for the VPA Admission Controller. It periodically queries the metrics API to obtain current resource usage information of Pods. Using this data, the VPA Recommender generates recommendations for the VPA Admission Controller.

The third component is the VPA Updater, which updates the requests and limits of Pods based on recommendations from the VPA admission controller. The VPA Updater periodically polls the Kubernetes API server to check for changes in the VPA configuration or Pod status and applies any necessary updates to the Pod resources. This ensures that the Pod resources are always set appropriately based on the current demand and usage of the application.



Kubernetes Vertical Autoscaling

Once the VPA is installed, our system can recommend and set resource requests for our Pods. To enable automatic computation of resource requirements, we need to insert a VPA resource for each Deployment.

Here's an example of how to do it:

```
1 cd $HOME/stateful-flask
2
3 cat <<EOF > kubernetes/vpa.yaml
4 apiVersion: autoscaling.k8s.io/v1
5 kind: VerticalPodAutoscaler
6 metadata:
7   name: stateful-flask-vpa
8   namespace: stateful-flask
9 spec:
10   targetRef:
11     apiVersion: "apps/v1"
12     kind: Deployment
13     name: stateful-flask
14   updatePolicy:
15     updateMode: "Auto"
16 EOF
```

Let's also use this Deployment with 2 replicas:

```
1 cat <<EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   replicas: 2
9   selector:
10    matchLabels:
11      app: stateful-flask
```

```
12  template:  
13    metadata:  
14      labels:  
15        app: stateful-flask  
16    spec:  
17      containers:  
18        - name: stateful-flask  
19          image: eon01/stateful-flask:v1  
20      resources:  
21        requests:  
22          memory: "1Mi"  
23          cpu: "1m"  
24        limits:  
25          memory: "128Mi"  
26          cpu: "100m"  
27      imagePullPolicy: Always  
28      ports:  
29        - containerPort: 5000  
30      env:  
31        - name: DB_USER  
32          valueFrom:  
33            secretKeyRef:  
34              name: stateful-flask-secret  
35              key: DB_USER  
36        - name: DB_PASSWORD  
37          valueFrom:  
38            secretKeyRef:  
39              name: stateful-flask-secret  
40              key: DB_PASSWORD  
41        - name: DB_HOST  
42          value: postgres.postgres.svc.cluster.local  
43        - name: DB_NAME  
44          value: stateful-flask-db  
45        - name: DB_PORT  
46          value: "5432"  
47 EOF
```

Now, let's apply the Deployment:

```
1 kubectl apply -f kubernetes/deployment.yaml
```

Let's also double check the requests and limits of the Pod:

```
1 # get the name of the Pods
2 POD1=$(kubectl -n stateful-flask get pod -l app=stateful-flask -o jsonp\
3 ath='[.items[0].metadata.name}')
4 POD2=$(kubectl -n stateful-flask get pod -l app=stateful-flask -o jsonp\
5 ath='[.items[1].metadata.name]')
6
7 # get the resources of the Pods
8 kubectl -n stateful-flask get pod $POD1 -o=jsonpath='{.spec.containers[\
9 0].resources}'; echo
10 kubectl -n stateful-flask get pod $POD2 -o=jsonpath='{.spec.containers[\
11 0].resources}'; echo
```

You should be able to see the following output for both Pods:

```
1 {"limits":{"cpu":"100m", "memory":"128Mi"}, "requests":{"cpu": "1m", "memor\
2 y": "1Mi"}}
```

Now, let's apply the VPA:

```
1 kubectl apply -f kubernetes/vpa.yaml
```

Let's check the status of the VPA:

```
1 kubectl get vpa -n stateful-flask
```

To witness VPA in action, we need to generate some load on the application. This can be achieved by executing the following command:

```

1 # Remove the load-generator pod if it already exists
2 kubectl delete pod load-generator
3 # Create a new load-generator pod
4
5 kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart\
=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://stateful-\
7 flask.stateful-flask.svc.cluster.local:5000/tasks; done"

```

After a few minutes, re-examine the resources of the Pods:

```

1 # get the name of the Pods
2 POD1=$(kubectl -n stateful-flask get pod -l app=stateful-flask -o jsonp\
3 ath='[.items[0].metadata.name}')
4 POD2=$(kubectl -n stateful-flask get pod -l app=stateful-flask -o jsonp\
5 ath='[.items[1].metadata.name]')
6
7 # get the resources of the Pods
8 kubectl -n stateful-flask get pod $POD1 -o=jsonpath='{.spec.containers[\
9 0].resources}'; echo
10 kubectl -n stateful-flask get pod $POD2 -o=jsonpath='{.spec.containers[\
11 0].resources}'; echo

```

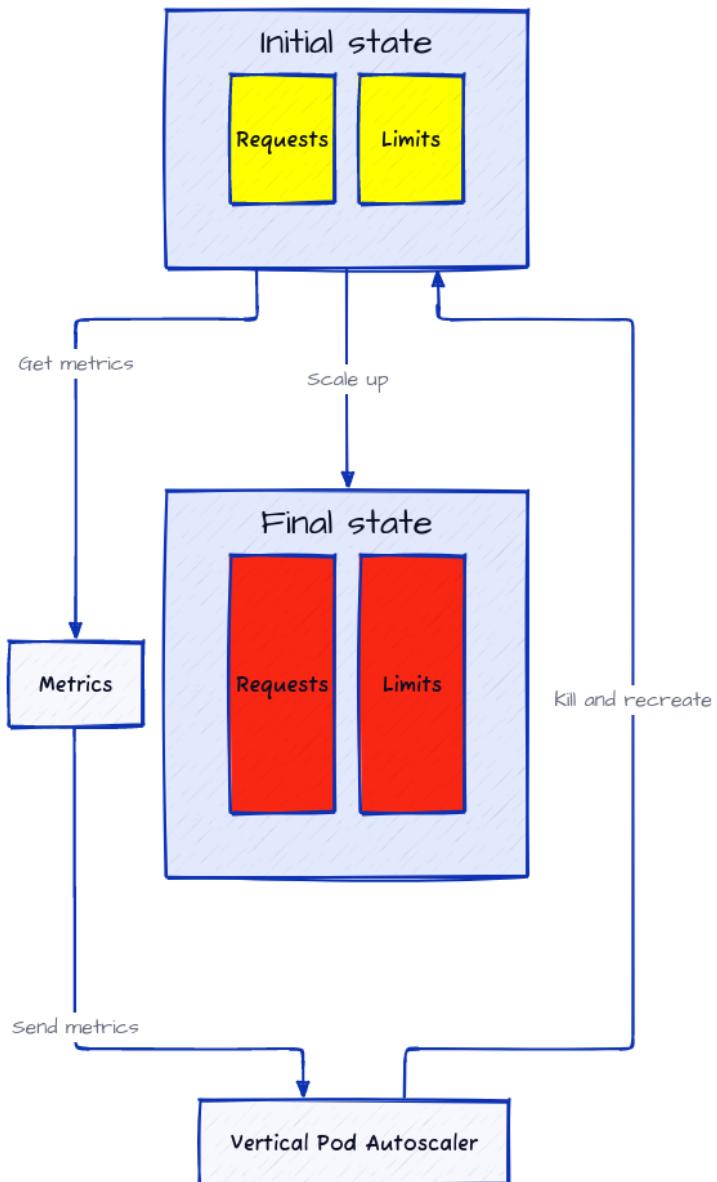
You will notice that the requests and limits of the Pods have been updated to reflect the current usage of the application. This is the new values in my case:

```

1 {"limits":{"cpu":"2500m", "memory": "32000Mi"}, "requests": {"cpu": "25m", "m\
2 emory": "262144k"}}
3 {"limits":{"cpu":"2500m", "memory": "32000Mi"}, "requests": {"cpu": "25m", "m\
4 emory": "262144k"}}

```

The VPA will continue to monitor the application and update the requests and limits of the Pods accordingly.



Kubernetes VPA Scaling up

14.3 VPA modes

In the previous example, this is how we used the VPA:

```
1 apiVersion: autoscaling.k8s.io/v1
2 kind: VerticalPodAutoscaler
3 metadata:
4   name: stateful-flask-vpa
5   namespace: stateful-flask
6 spec:
7   targetRef:
8     apiVersion: "apps/v1"
9     kind: Deployment
10    name: stateful-flask
11   updatePolicy:
12     updateMode: "Auto"
```

As you can see, we have an `updateMode` of `Auto`. This means that the VPA will automatically update the `requests` and `limits` of the Pods whether there's a scale up or scale down event.

There are 4 modes in total that can be used. We will go through each of them.

14.3.1 Auto

The VPA will automatically update the `requests` and `limits` of the Pods whether there's a scale up or scale down event.

This can be useful for your critical workloads that require high availability and cannot tolerate any downtime and any use case where you want to adjust resource requirements dynamically while minimizing the impact on the running Pods.

This mode may be suitable for stateful applications that require a more graceful way of updating their resource requirements.

14.3.2 Initial

The VPA will only update the requests and limits of the Pods when they are created.

This mode can be useful when you have predictable workloads that do not change significantly over time and you want to set resource requests upfront to ensure that they are met when the Pods are scheduled.

14.3.3 Recreate

The VPA will only update the requests and limits of the Pods when they are recreated.

This can be useful when you have variable workloads that can change over time, and you want to adjust resource requirements dynamically to ensure that the Pods have enough resources to run efficiently. This mode may be suitable for stateless applications that can tolerate short disruptions during Pod evictions.

14.3.4 Off

The VPA will not update the requests and limits of the Pods however it will still collect metrics and make recommendations. So the VPA will still be able to tell you if your Pods are over or under provisioned.

If you want to experiment with VPA recommendations and evaluate the impact on your applications without automatically changing their resource requirements then this mode is for you. It could be useful to understand how your application behaves under different load conditions.

We are going to use this mode in the next section and experiment with the VPA recommendations.

14.4 VPA recommendations

The VPA recommender, as seen, is a component of the VPA that collects metrics and makes recommendations. It is responsible for calculating the resource requirements for the Pods based on the metrics it collects.

Let's see an example. You can create this YAML file to use the Off mode:

```
1 cat <<EOF > kubernetes/vpa.yaml
2 apiVersion: autoscaling.k8s.io/v1
3 kind: VerticalPodAutoscaler
4 metadata:
5   name: stateful-flask-vpa
6   namespace: stateful-flask
7 spec:
8   targetRef:
9     apiVersion: "apps/v1"
10    kind: Deployment
11    name: stateful-flask
12   updatePolicy:
13     updateMode: "Off"
14 EOF
```

Then apply it:

```
1 kubectl apply -f kubernetes/vpa.yaml
```

Proceed with a load test as we did before and you will see that the requests and limits of the Pods will not be updated.

```
1 # Remove the load-generator pod if it already exists
2 kubectl delete pod load-generator
3 # Create a new load-generator pod
4
5 kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart\
6 =Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://stateful-
7 flask.stateful-flask.svc.cluster.local:5000/tasks; done"
```

You can then notice the recommendations made by the VPA by running the following command:

```
1 kubectl -n stateful-flask describe vpa stateful-flask-vpa
```

This is an example of the output:

```
1 Name: stateful-flask-vpa
2 [ ... ]
3 Spec:
4   Target Ref:
5     API Version: apps/v1
6     Kind: Deployment
7     Name: stateful-flask
8   Update Policy:
9     Update Mode: Off
10  Status:
11    Conditions:
12      Last Transition Time: 2023-05-05T15:37:17Z
13      Status: True
14      Type: RecommendationProvided
15    Recommendation:
16      Container Recommendations:
17        Container Name: stateful-flask
18        Lower Bound:
19          Cpu: 25m
20          Memory: 262144k
21        Target:
22          Cpu: 25m
23          Memory: 262144k
24        Uncapped Target:
25          Cpu: 25m
26          Memory: 262144k
27        Upper Bound:
28          Cpu: 25m
29          Memory: 262144k
30  Events: <none>
```

What is important to notice is the Recommendation section. This is where the VPA will tell you what the requests and limits of the Pods should be.

In this section there are 4 fields:

- Lower Bound is the minimum amount of resources that the Pod should have
- Target is the amount of resources that the Pod should have
- Uncapped Target is the amount of resources that the Pod should have if there were no limits
- Upper Bound is the maximum amount of resources that the Pod should have

These results are based on the metrics that the VPA has collected and they change based on the load that the application is receiving. You should therefore test your application with a realistic load to get the best results.

After executing the load test, you may want to set the Pod resources to the recommended values which is usually the Target value.

14.4.1 VPA Limitations

Vertical Pod Autoscaler (VPA) has several limitations that users should consider before implementing it in their Kubernetes cluster.

- **Recreating Pods when updating resources:** Whenever the VPA updates the Pod resources, the Pod is recreated, which causes all running containers to be recreated. Note that the Pod may be recreated on a different node.
- **Pod recreation guarantee limitations:** The VPA cannot guarantee that Pods it evicts or deletes to apply recommendations (when configured in Auto and Recreate modes) will be successfully recreated.
- **Limitations for non-controller Pods:** The VPA does not evict Pods which are not run under a controller. For such Pods Auto mode is equivalent to Initial. In all cases, creating Pods without a controller is not recommended.
- **Incompatibility with HPA on CPU or memory:** The VPA should not be used with the Horizontal Pod Autoscaler (HPA) on CPU or memory. However, if your HPA is based on custom or external metrics then it's possible to use VPA with HPA.

Incomplete handling of out-of-memory events: The VPA reacts to most out-of-memory events, but not in all situations.

Performance limitations in large clusters: The VPA performance has not been tested in large clusters. Also there is no clear definition of what constitutes a large cluster in terms of the number of Pods or nodes. In general, larger clusters with more Pods and nodes may experience more performance limitations, but the exact threshold will depend on many factors and may vary from one cluster to another. It is therefore recommended to test the performance of your VPA in your specific cluster configuration to ensure it meets your requirements.

Recommendation exceeding available resources: VPA recommendation might exceed available resources (e.g. Node size, available size, available quota) and cause Pods to go pending. This can be partly addressed by using VPA together with Cluster Autoscaler.

Undefined behavior with multiple VPA resources for the same Pod: Multiple VPA resources matching the same Pod have an undefined behavior. Therefore, it is recommended to avoid creating multiple VPA resources for the same Pod.

Number of replicas: The VPA is by default configured to apply changes only when there are at least 2 replicas. This is to avoid the situation where the VPA would recommend a change that would cause your application to be unavailable. This behavior can be changed by setting the `minReplicas` field in the `updatePolicy` section of the VPA spec.

Example:

```
1 apiVersion: autoscaling.k8s.io/v1
2 kind: VerticalPodAutoscaler
3 metadata:
4   name: stateful-flask-vpa
5   namespace: stateful-flask
6 spec:
7   targetRef:
8     apiVersion: "apps/v1"
9     kind: Deployment
10    name: stateful-flask
11 updatePolicy:
```

```
12      updateMode: "Off"  
13      minReplicas: 1
```

15 Scaling Stateful Microservices: PostgreSQL as an Example

15.1 StatefulSets and scaling

Previously, we created a PostgreSQL database using a StatefulSet. Below is the manifest we used:

```
1 apiVersion: apps/v1
2 kind: StatefulSet
3 metadata:
4   name: postgres
5   namespace: postgres
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10    app: postgres
11   template:
12     metadata:
13       labels:
14         app: postgres
15     spec:
16       containers:
17         - name: postgres
18           image: postgres:10.1
19           imagePullPolicy: Always
20         ports:
```

```

21      - containerPort: 5432
22      envFrom:
23          - configMapRef:
24              name: postgres-config
25      volumeMounts:
26          - name: postgredb-volume
27              mountPath: /var/lib/postgresql/data
28              subPath: postgres
29      env:
30          - name: PGDATA
31              value: /var/lib/postgresql/data/pgdata
32  volumeClaimTemplates:
33      - metadata:
34          name: postgredb-volume
35      spec:
36          accessModes: [ "ReadWriteOnce" ]
37          storageClassName: "do-block-storage"
38      resources:
39          requests:
40              storage: 5Gi

```

In the above YAML, we define `volumeClaimTemplates` as a template for a `PersistentVolumeClaim`. This allows us to scale the `StatefulSet`, with each replica having its own `PersistentVolumeClaim`. However, we may encounter data consistency issues because the data is not replicated between the replicas.

For example, if we create an entry in the database using an API POST request that is sent to the first replica, scaling the `StatefulSet` to 2 replicas could result in a GET request being sent to the second replica before the data is replicated. This would cause an error.

PostgreSQL is not just a tool, but an ecosystem of tools. Think of PostgreSQL as the kernel of an operating system; it provides basic functionality, but you need additional tools to make it useful. There are different tools and techniques available to solve the problem of data consistency, such as building our own PostgreSQL cluster using streaming replication, using a tool like [Patroni](#)¹⁰³, or one

¹⁰³<https://github.com/zalando/patroni>

of its alternatives.

Although it may seem easy on the surface, data consistency is actually a complex problem. Therefore, it is better to use a stable tool that is already available and maintained by a community, rather than reinventing the wheel.

When it comes to Kubernetes, there are multiple tools we can use:

- [Zalando PostgreSQL Operator](#)¹⁰⁴: Postgres operator creates and manages PostgreSQL clusters running in Kubernetes
- [PostgreSQL HA packaged by Bitnami](#)¹⁰⁵: This PostgreSQL cluster solution includes the PostgreSQL replication manager, an open-source tool for managing replication and failover on PostgreSQL clusters.
- [PGO](#)¹⁰⁶: Production PostgreSQL for Kubernetes, from high availability Postgres clusters to full-scale database-as-a-service.
- [Stolon](#)¹⁰⁷: PostgreSQL cloud native High Availability and more.
- And [more](#)¹⁰⁸

In the following sections, we are going to experiment with Stolon.

15.2 Stolon: introduction

Stolon is a cloud-native PostgreSQL manager for high availability. It allows you to maintain a highly available PostgreSQL instance within your containers (with Kubernetes integration) as well as on other types of infrastructure, such as cloud IaaS and old-style infrastructures.

The Stolon architecture comprises three main components:

- **Keeper**: It manages a PostgreSQL instance, converging to the “[clusterview](#)¹⁰⁹” computed by the leader sentinel.

¹⁰⁴<https://github.com/zalando/postgres-operator>

¹⁰⁵<https://github.com/bitnami/charts/tree/main/bitnami/postgresql-ha>

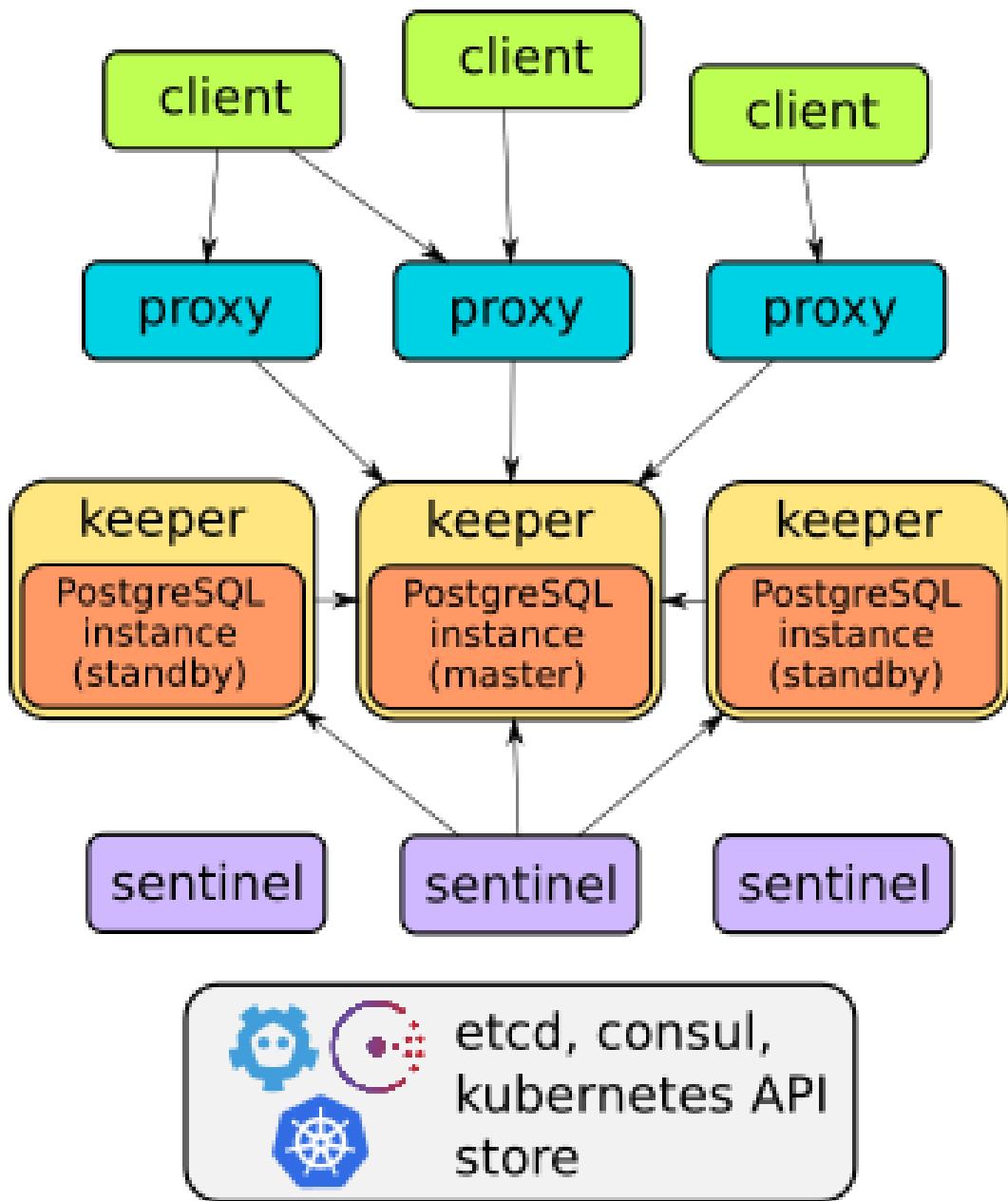
¹⁰⁶<https://github.com/CrunchyData/postgres-operator>

¹⁰⁷<https://github.com/sorintlab/stolon>

¹⁰⁸<https://www.google.com/search?q=postgres+highly+available+on+kubernetes>

¹⁰⁹<https://github.com/sorintlab/stolon/blob/master/doc/architecture.md>

- **Sentinel:** It discovers and monitors keepers and proxies, and computes the optimal “clusterview”.
- **Proxy:** The client’s access point. It enforces connections to the right PostgreSQL master and forcibly closes connections to old masters.



resources/images/architecture_stolon.png

Stolon has many helpful features as described in the [documentation](#)¹¹⁰:

- Leverages PostgreSQL streaming replication.
- Resilient to any kind of partitioning, it prefers consistency over availability while trying to maintain maximum availability.
- Provides Kubernetes integration for achieving PostgreSQL high availability.
- Uses a cluster store such as etcd, Consul, or Kubernetes API server as a highly available data store and for leader election.
- Supports asynchronous (default) and synchronous replication.
- Allows for full cluster setup in minutes and easy cluster administration.
- Can perform point-in-time recovery by integrating with your preferred backup/restore tool.
- Supports standby cluster for multi-site replication and near-zero downtime migration.
- Offers automatic service discovery and dynamic reconfiguration, handling PostgreSQL and Stolon processes changing their addresses.
- Can use pg_rewind for fast instance resynchronization with the current master.

We will install Stolon on our Kubernetes cluster and evaluate its performance.

15.3 Stolon: installation

To start, we are going to clone the repository:

```
1 cd $HOME/stateful-flask
2 git clone https://github.com/sorintlab/stolon.git
3 cd stolon/examples/kubernetes
```

Use this command to initialize the cluster:

¹¹⁰<https://github.com/sorintlab/stolon>

```
1 kubectl run -i -t stolonctl --image=sorintlab/stolon:master-pg10 --rest\\
2 art=Never --rm -- /usr/local/bin/stolonctl --cluster-name=kube-stolon -
3 -store-backend=kubernetes --kube-resource-kind=configmap init
```

Next, we need to create the sentinels. By default, two sentinels are created, but you can change this number by editing the `stolon-sentinel.yaml` file or by scaling the Deployment later on.

```
1 kubectl create -f stolon-sentinel.yaml
```

Create the keeper's password secret

```
1 kubectl create -f secret.yaml
```

This secret contains a password that will be used by the keeper to set up the initial database user. You can change the password by editing the `secret.yaml` file. The default password is `password1`. If you want to change the password, don't forget to encode it in Base64.

```
1 echo -n "<Your_PASSWORD>" | base64
```

For simplicity's sake, we will keep the default password.

Now, let's move on to creating the Stolon Keepers StatefulSet:

```
1 kubectl create -f stolon-keeper.yaml
```

This will define a StatefulSet with 2 replicas. You can change the number of replicas by editing the `stolon-keeper.yaml` file or by scaling the StatefulSet later. The sentinel will choose a random keeper as the initial master; this keeper will initialize a new database cluster while the other keeper becomes a standby.

Create the proxy:

```
1 kubectl create -f stolon-proxy.yaml
```

Finally, create the service for the proxy.

```
1 kubectl create -f stolon-proxy-service.yaml
```

All components were installed in the default namespace.

To check the status of the cluster, run the following command:

```
1 kubectl run -i -t stolonctl --image=sorintlab/stolon:master-pg10 --rest\  
2   art=Never --rm -- /usr/local/bin/stolonctl --cluster-name=kube-stolon -  
3   -store-backend=kubernetes --kube-resource-kind=configmap status
```

15.4 Stolon: usage

As we have a proxy service, we can use it to connect to the database. However, the proxy service is part of the default Namespace, while our Flask application is in the `stateful-flask` Namespace.

To redirect traffic to the proxy service, we can create an `ExternalName` service in the `stateful-flask` Namespace.

```
1 cat <<EOF > kubernetes/stolon-proxy-service-externalname.yaml  
2 apiVersion: v1  
3 kind: Service  
4 metadata:  
5   name: db  
6   namespace: stateful-flask  
7 spec:  
8   type: ExternalName  
9   externalName: stolon-proxy-service.default.svc.cluster.local  
10 EOF
```

As shown above, the code creates a service named db in the stateful-flask namespace. This service redirects traffic to the stolon-proxy-service service in the default namespace.

To update the deployment and:

- Use the new service
- Create the database before starting the Flask application

You need to update the kubernetes/deployment.yaml file.

```
1 cat <<EOF > kubernetes/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: stateful-flask
6   namespace: stateful-flask
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: stateful-flask
12 template:
13   metadata:
14     labels:
15       app: stateful-flask
16 spec:
17   containers:
18     - name: stateful-flask
19       image: eon01/stateful-flask:v0
20       imagePullPolicy: Always
21     ports:
22       - containerPort: 5000
23     env:
24       - name: DB_USER
25         value: stolon
```

```
26      - name: DB_PASSWORD
27          value: password1
28      - name: DB_HOST
29          value: db
30      - name: DB_NAME
31          value: mydb
32      - name: DB_PORT
33          value: "5432"
34
35      initContainers:
36          - name: init-db
37              image: postgres:9.6
38              imagePullPolicy: IfNotPresent
39              command: ["bash", "-c", "createdb -h db -U $(DB_USER) $(DB_NA\
40 ME) || true"]
41              env:
42                  - name: PGPASSWORD
43                      value: password1
44                  - name: DB_NAME
45                      value: mydb
46                  - name: DB_USER
47                      value: stolon
48
49 EOF
```

In the YAML code above, we use the db service to establish a connection with the database. Additionally, we utilize an init container to create the database before launching the Flask application.



Init containers are a special type of container that runs before the main container. They are useful for performing initialization tasks, such as creating a database in our case. For the init container to work properly, it must exit with a 0 exit code. If the init container fails, the main container will not start.

Now, we can apply the changes to both the service and the deployment.

```
1 kubectl apply -f kubernetes/stolon-proxy-service-externalname.yaml
2 kubectl apply -f kubernetes/deployment.yaml
```

Let's check the initialization status:

```
1 export pod=$(kubectl get pods -n stateful-flask -l app=stateful-flask -\
2 o jsonpath='{.items[0].metadata.name}')
3 kubectl -n stateful-flask logs -f $pod -c init-db
```

If everything went well, you should be able to move on to the next step, which is creating the database schema.

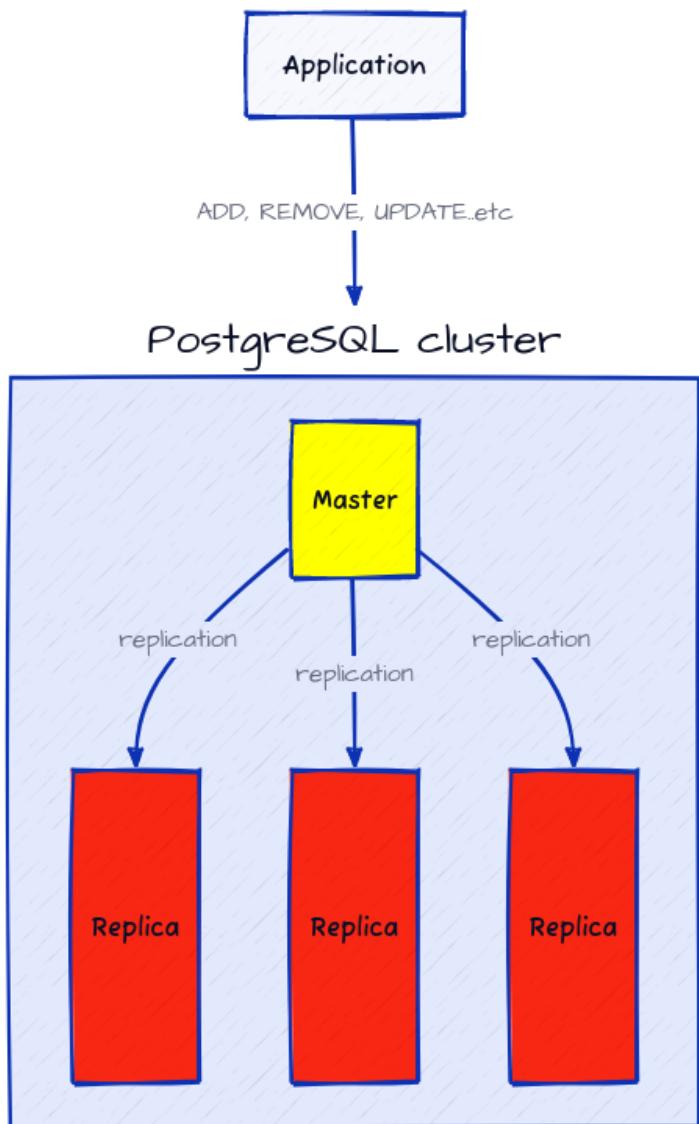
```
1 export pod=$(kubectl get pods -n stateful-flask -l app=stateful-flask -\
2 o jsonpath='{.items[0].metadata.name}')
3
4 kubectl -n stateful-flask exec -it $pod -- db init
5 kubectl -n stateful-flask exec -it $pod -- db migrate
6 kubectl -n stateful-flask exec -it $pod -- db upgrade
```

Finally, we can scale the Deployment and the StatefulSet to as many replicas as desired.

```
1 kubectl scale deployment stateful-flask -n stateful-flask --replicas=5
2 kubectl scale statefulset stolon-keeper -n default --replicas=3
```

You can also add new tasks to the to-do list.

```
1 export url="$(kubectl get ingress | awk '{print $3}' | tail -n 1\
2 )/tasks"
3
4 curl -X POST -H "Content-Type: application/json" -d '{"title":"New task\
5 1", "description":"New description"}' $url
6 curl -X POST -H "Content-Type: application/json" -d '{"title":"New task\
7 2", "description":"New description"}' $url
8 curl -X POST -H "Content-Type: application/json" -d '{"title":"New task\
9 3", "description":"New description"}' $url
```



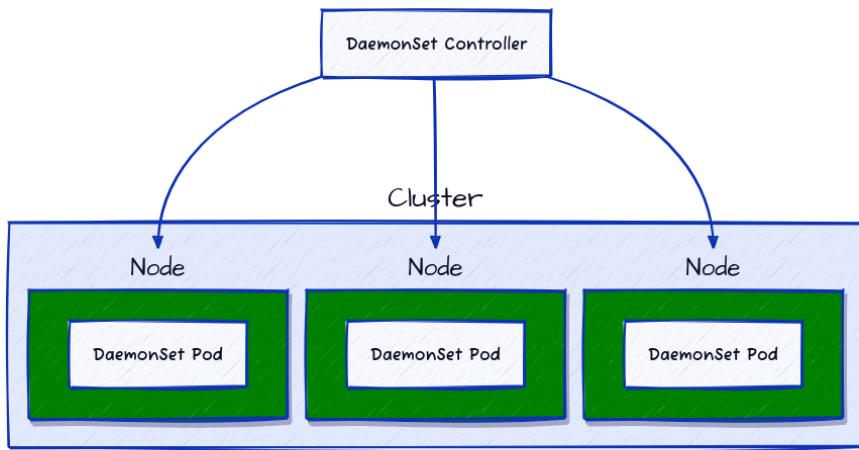
PostgreSQL Replication

Every task that is added will be stored on the master database replica and will be available to all other replicas thanks to the whole stolon setup. You can read more about this setup at <https://sgotti.dev/post/stolon-introduction/>.

16 Microservices Deployment Strategies: One Service Per Node

16.1 DaemonSet: role and use cases

A DaemonSet is a type of resource controller in Kubernetes that ensures a Pod is running on each node in a cluster. Unlike other resource controllers, which deploy multiple replicas of a Pod across a cluster, a DaemonSet deploys a single Pod on each node and no more. This ensures that one instance of the application is running on each node in the cluster.



Our DaemonSet Controller ensures that all Nodes run a copy of the Pod

DaemonSets are often used for tasks such as log collection, monitoring, or network proxies on every node.

A typical example of a DaemonSet is Fluentd. It collects logs from all nodes in a cluster and sends them to a central log aggregator.

16.2 DaemonSet: creating and managing

Following is an example of a DaemonSet manifest that deploys Fluentd to all nodes in a cluster.

Start by running the following command to create a directory named `fluentd` in your home directory:

```
1 cd $HOME/  
2 mkdir -p fluentd  
3 cd fluentd
```

Then create a file named `daemonset.yaml` with the following content:

```
1 cat <<EOF > daemonset.yaml  
2 apiVersion: apps/v1  
3 kind: DaemonSet  
4 metadata:  
5   name: fluentd-elasticsearch  
6   namespace: kube-system  
7   labels:  
8     k8s-app: fluentd-logging  
9 spec:  
10   selector:  
11     matchLabels:  
12       name: fluentd-elasticsearch  
13   template:  
14     metadata:  
15       labels:  
16         name: fluentd-elasticsearch  
17     spec:  
18       containers:  
19         - name: fluentd-elasticsearch  
20           image: quay.io/fluentd_elasticsearch/fluentd:v4  
21           volumeMounts:  
22             - name: varlog  
23               mountPath: /var/log  
24           terminationGracePeriodSeconds: 30  
25       volumes:  
26         - name: varlog  
27           hostPath:  
28             path: /var/log  
29 EOF
```

- The YAML above defines a DaemonSet object that ensures all nodes in the cluster run a copy of the specified Pod. The DaemonSet is named fluentd-elasticsearch and is located in the kube-system namespace.
- The spec includes a selector that matches the Pods that the DaemonSet will manage. In this case, the selector matches Pods with the label name: fluentd-elasticsearch. This mechanism is similar to the one used by ReplicaSets in Deployments.
- The Pod runs a container named fluentd-elasticsearch that uses the image quay.io/fluentd_elasticsearch/fluentd:v4.
- The container also mounts the host's /var/log directory to the container's /var/log directory. This is necessary because Fluentd is used to collect logs from the host.
- The manifest specifies a termination grace period of 30 seconds. This means that when the DaemonSet is deleted, the Pods will be terminated after 30 seconds.

We can apply the DaemonSet manifest using the following command:

```
1 kubectl apply -f daemonset.yaml
```

Then check if the DaemonSet Pods are running one replica on each node:

```
1 kubectl get pods -n kube-system -o wide | grep fluentd-elasticsearch
```

The 7th column of the output displays the node on which the Pod is currently running.

The master node is typically tainted to prevent Pods from being scheduled on it. Therefore, if we intend to deploy a DaemonSet to the master node, we must add a Tolerations to the DaemonSet.

This is just an example. We will learn more about Taints and Tolerations in the next section.

```
1 apiVersion: apps/v1
2 kind: DaemonSet
3 metadata:
4   name: fluentd-elasticsearch
5   namespace: kube-system
6   labels:
7     k8s-app: fluentd-logging
8 spec:
9   selector:
10    matchLabels:
11      name: fluentd-elasticsearch
12   template:
13     metadata:
14       labels:
15         name: fluentd-elasticsearch
16     spec:
17       # Tolerations added to allow DaemonSet to be deployed to the cont\
18   tolerations:
19     - key: node-role.kubernetes.io/master
20       operator: Exists
21       effect: NoSchedule
22     - key: node-role.kubernetes.io/control-plane
23       operator: Exists
24       effect: NoSchedule
25   # End of Tolerations definition
26   containers:
27     - name: fluentd-elasticsearch
28       image: quay.io/fluentd_elasticsearch/fluentd:v4
29       volumeMounts:
30         - name: varlog
31         mountPath: /var/log
32   terminationGracePeriodSeconds: 30
33   volumes:
34     - name: varlog
35     hostPath:
```

37

path: /var/log

There is an exception if you are using a managed Kubernetes service such as DigitalOcean's DOK. Managed Kubernetes services usually have restrictions and limitations in place to ensure the stability, security, and performance of the control plane nodes. Therefore, if you use a managed Kubernetes service, you will not be able to deploy a DaemonSet to the control plane nodes. However, this limitation does not apply to non-managed Kubernetes clusters.

17 Microservices Deployment Strategies: Assigning Workloads to Specific Nodes

17.1 Assigning your workloads to specific nodes: why?

There are situations where you may need to assign some or all of your workloads to specific nodes. Here are some examples:

- Special hardware: Some applications require specific hardware capabilities. In these cases, you may need to run some workloads on certain nodes that have the necessary hardware.
- Licensing: Certain software licenses limit the number of nodes that can run a specific application. If you can't run a software application on more than one node, you need to assign the workload to a specific node.
- Compliance: Sometimes, you need to run a workload on a specific node to comply with regulations or policies. For instance, you may need to run a workload on a node located in a specific region or country.
- Data locality: If your application relies on accessing data stored on a specific node, you may want to assign the workload to that node to optimize data access and minimize network latency.
- Performance optimization: You may want to run a workload on a specific node to optimize the performance of your microservices. This node could be used only for running a specific type of optimized workload.
- Testing and debugging: Running a workload on a specific node can help you test and debug your application. For example, you may want to run a workload on a specific node to test a new version of your application before deploying it to the rest of the cluster.

- **Legacy systems:** Some legacy systems may require you to run a workload on a specific node. For instance, you may need to run a workload on a node that is running a specific version of an operating system.
- **Security:** In some cases, you may want to run a workload on a specific node to enhance security. For example, you may want to run a workload on a node that has special OS hardening or security software.

The above list is not exhaustive. There are certainly other use cases where you may want to assign workloads to specific nodes.

In Kubernetes, running a workload or a microservice on a certain node(s) translates to a set of rules used to select the node(s) where the Pods running your workloads and microservices will be deployed. These rules are defined using the following Kubernetes concepts:

- **Taints and Tolerations:** Ensure Pods are not scheduled on nodes with specific taints unless they have corresponding tolerations.
- **nodeSelector:** A simple feature that allows scheduling Pods on nodes with matching labels specified by the user.
- **Node Affinity:** An enhanced version of nodeSelector that provides fine-grained control over Pod scheduling based on node labels.

We will cover the three concepts in this section. The remaining two are very similar and use almost the same mechanisms.

17.2 Taints and Tolerations

17.2.1 Taints and Tolerations: definition

Taints and Tolerations typically work together to ensure that Pods are not scheduled on inappropriate nodes. Taints are applied to Nodes, while Tolerations are applied to Pods.

When you add a Taint to a node, Kubernetes will repel all Pods from it except those that have a matching Toleration.

Each Taint has three components:

- a key, which is a name used to identify the Taint.
- a value, which is an optional value used to identify the Taint.
- an effect, which is the action that Kubernetes will take on any Pods that do not tolerate the Taint.

The effect can be one of the following:

- NoSchedule: Kubernetes will not schedule any Pods that do not tolerate the Taint.
- PreferNoSchedule: Kubernetes will try not to schedule any Pods that do not tolerate the Taint, but this is not guaranteed.
- NoExecute: Kubernetes will evict any existing Pods that do not tolerate the Taint.

You can view the Taints of your nodes using the following command:

```
1 kubectl get nodes -o=custom-columns=NodeName:.metadata.name,TaintKey:.s\
2 pec.taints[*].key,TaintValue:.spec.taints[*].value,TaintEffect:.spec.ta\
3 ints[*].effect
```

Alternatively, you can create an alias for the above command.

```
1 cat <<EOF >> ~/.bashrc
2 alias taints='kubectl get nodes -o=custom-columns=NodeName:.metadata.na\
3 me,TaintKey:.spec.taints[*].key,TaintValue:.spec.taints[*].value,TaintE\
4 ffect:.spec.taints[*].effect'
5 EOF
```

Next, reload the bashrc file and execute the alias.

```
1 source ~/.bashrc
2 taints
```

By default, Kubernetes Node Controllers automatically taint nodes when certain conditions are met. Here are some of the default Taints:

- node.kubernetes.io/not-ready: Applied to nodes that are not ready.
- node.kubernetes.io/unreachable: Applied to nodes that are unreachable.
- node.kubernetes.io/memory-pressure: Applied to nodes that are under memory pressure.
- node.kubernetes.io/disk-pressure: Applied to nodes that are under disk pressure.
- node.kubernetes.io/pid-pressure: Applied to nodes that are under PID pressure.
- node.kubernetes.io/network-unavailable: Applied to nodes that are experiencing network issues.
- node.kubernetes.io/unschedulable: Applied to nodes that cannot be scheduled.
- node.cloudprovider.kubernetes.io/uninitialized: Applied to nodes that have not yet been initialized and is used by cloud providers.

You can also manually add a Taint to a node using the following command:

```
1 kubectl taint nodes <node-name> <taint-key>=<taint-value>:<taint-effect>
```

17.2.2 Taints and Toleration: example

Let's see an example. Taint all your nodes with the following command:

```
1 kubectl taint nodes --all mykey=myvalue:NoSchedule
```

You can check the Taints of your nodes again using the alias:

```
1 taints
```

Now apply the following DaemonSet to your cluster:

```
1 cat <<EOF > daemonset.yaml
2 apiVersion: apps/v1
3 kind: DaemonSet
4 metadata:
5   name: fluentd-elasticsearch
6   namespace: kube-system
7   labels:
8     k8s-app: fluentd-logging
9 spec:
10   selector:
11     matchLabels:
12       name: fluentd-elasticsearch
13   template:
14     metadata:
15       labels:
16         name: fluentd-elasticsearch
17   spec:
18     containers:
19       - name: fluentd-elasticsearch
20         image: quay.io/fluentd_elasticsearch/fluentd:v4
21         volumeMounts:
22           - name: varlog
23             mountPath: /var/log
24         terminationGracePeriodSeconds: 30
25     volumes:
26       - name: varlog
27         hostPath:
28           path: /var/log
29 EOF
```

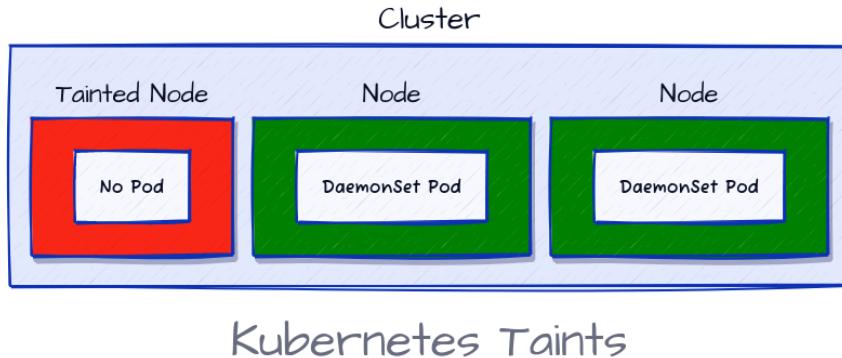
Apply the DaemonSet:

```
1 kubectl apply -f daemonset.yaml
```

To check where the DaemonSet Pods are running, use the following command:

```
1 kubectl -n kube-system get pods -o=custom-columns=PodName:.metadata.name,NodeName:.spec.nodeName | grep fluentd-elasticsearch
```

The DaemonSet Pods should not appear in the output. This is because the taints on the nodes are repelling the Pods.



To create an exception for the DaemonSet Pods using a Toleration, use the following YAML:

```
1 cat <<EOF > daemonset.yaml
2 apiVersion: apps/v1
3 kind: DaemonSet
4 metadata:
5   name: fluentd-elasticsearch
6   namespace: kube-system
7   labels:
8     k8s-app: fluentd-logging
9 spec:
10   selector:
11     matchLabels:
```

```
12      name: fluentd-elasticsearch
13  template:
14    metadata:
15      labels:
16        name: fluentd-elasticsearch
17  spec:
18    tolerations:
19      - key: mykey
20        operator: Equal
21        value: myvalue
22        effect: NoSchedule
23  containers:
24    - name: fluentd-elasticsearch
25      image: quay.io/fluentd_elasticsearch/fluentd:v4
26      volumeMounts:
27        - name: varlog
28          mountPath: /var/log
29    terminationGracePeriodSeconds: 30
30  volumes:
31    - name: varlog
32      hostPath:
33        path: /var/log
34 EOF
```

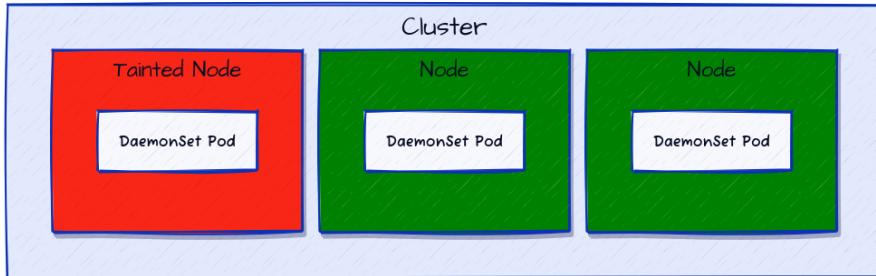
Now apply the DaemonSet:

```
1 kubectl apply -f daemonset.yaml
```

Check where the DaemonSet Pods are running:

```
1 kubectl -n kube-system get pods -o=custom-columns=PodName:.metadata.name\
2 ,NodeName:.spec.nodeName | grep fluentd-elasticsearch
```

You should see the DaemonSet Pods running on all nodes.



The tainted node will not repel the DaemonSet Pod when it has the right Toleration

If you want to remove the Taints from your nodes, you can use the following command:

```
1 kubectl taint nodes --all mykey-
```

Verify that the taints have been removed.

```
1 taints
```

17.3 nodeSelector

17.3.1 The simplest form of node affinity

This is probably the simplest way to constrain Pods to specific nodes because it uses an easy-to-understand label selector. However, nodeSelector is a limited form of node affinity. It can only express simple requirements. If you have more complex requirements, you should use node affinity instead.

17.3.2 nodeSelector: example

Let's see an example.

Create a new workspace:

```
1 cd $HOME/ && mkdir -p nodeSelector && cd nodeSelector
```

Create a Pod that is scheduled on a node with the label `disktype: ssd`.

```
1 cat <<EOF > nodeSelector.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: nginx-deployment
6 spec:
7   replicas: 3
8   selector:
9     matchLabels:
10    app: nginx
11   template:
12     metadata:
13       labels:
14         app: nginx
15     spec:
16       nodeSelector:
17         disktype: ssd
18       containers:
19         - name: nginx
20           image: nginx
21           ports:
22             - containerPort: 80
23 EOF
```

Apply the Pod manifest:

```
1 kubectl apply -f nodeSelector.yaml
```

Check where the Pod is running:

```
1 kubectl get pods -o=custom-columns=PodName:.metadata.name,NodeName:.spec\n
2 c.nodeName
```

To ensure that the Pod runs on a node with the label `disktype: ssd`, check that a node with that label exists. If such a node does not exist, the Pod will remain in the Pending state.

Normally, the label is used to match the node and the Pod is scheduled on that node. If there's no node with that label, the Pod will always remain in the Pending state.

To view the default labels of your nodes, execute the following command:

```
1 kubectl get nodes --show-labels
```

Alternatively, you can format the output to show only the labels, one per line:

```
1 kubectl get nodes --show-labels | xargs -L1 | tail -n 1 | tr "," "\n"
```

If the output does not contain the label `disktype:ssd`, you can add it to all nodes with SSD disks using the following command:

```
1 # add label to all nodes
2 kubectl label nodes --all disktype=ssd
3 # or add label to one node
4 kubectl label nodes <node-name> disktype=ssd
```

You should observe the Pod transitioning from the Pending state to the Running state.

Remember to clean up afterwards:

```
1 kubectl delete -f nodeSelector.yaml
```

17.4 Node affinity and anti-affinity

17.4.1 Node affinity: like nodeSelector but with more options

In Kubernetes, administrators usually depend on the scheduler to automatically choose nodes for scheduling Pods based on resource availability. However, there are scenarios where manual selection is necessary, such as when Pods need to be placed on specific nodes with certain attributes. These attributes can be labels, so in this case, we can simply use nodeSelector. However, in other scenarios where more complex rules are required, node affinity can be used.

17.4.2 Node affinity: example

In this example, we aim to schedule our microservices on nodes with specific hostnames.

Suppose we have a cluster of three nodes:

- default-f72fj
- default-f72fo
- default-f8wx9

Our objective is to deploy Nginx on the node with the hostname default-f8wx9.

To begin, create a new workspace.

```
1 cd $HOME/ && mkdir -p node-affinity && cd node-affinity
```

Next, create a Pod that is scheduled on a node with the label hostname: default-f8wx9.

```
1 cat <<EOF > affinity.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: nginx
6 spec:
7   replicas: 5
8   selector:
9     matchLabels:
10    app: nginx
11   template:
12     metadata:
13       labels:
14         app: nginx
15     spec:
16       containers:
17         - name: nginx
18           image: nginx
19         ports:
20           - containerPort: 80
21     affinity:
22       nodeAffinity:
23         requiredDuringSchedulingIgnoredDuringExecution:
24           nodeSelectorTerms:
25             - matchExpressions:
26               - key: kubernetes.io/hostname
27                 operator: In
28               values:
29                 - default-f8wx9
30 EOF
```

Apply the Deployment:

```
1 kubectl apply -f affinity.yaml
```

Check where our Pods are running:

```
1 kubectl get pods -o=custom-columns=PodName:.metadata.name,NodeName:.spec\n2 c.nodeName | grep nginx
```

You should see the Pod running on a node with the label `hostname: default-f8wx9`.

To add another node to the list of nodes where the Pod can be scheduled, use the following command:

```
1 kubectl edit deployment nginx
```

Then add the following lines:

```
1     affinity:
2       nodeAffinity:
3         requiredDuringSchedulingIgnoredDuringExecution:
4           nodeSelectorTerms:
5             - matchExpressions:
6               - key: kubernetes.io/hostname
7                 operator: In
8                 values:
9                   - default-f8wx9
10                  - default-f72fj
```

This is another example of node affinity using the `topology.kubernetes.io/zone` label.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx
5 spec:
6   replicas: 5
7   selector:
8     matchLabels:
9       app: nginx
10  template:
11    metadata:
12      labels:
13        app: nginx
14    spec:
15      containers:
16        - name: nginx
17          image: nginx
18          ports:
19            - containerPort: 80
20      affinity:
21        nodeAffinity:
22          requiredDuringSchedulingIgnoredDuringExecution:
23            nodeSelectorTerms:
24              - matchExpressions:
25                - key: topology.kubernetes.io/zone
26                  operator: In
27                  values:
28                    - FRA
29                    - LON
```

The above example schedules the Pod on nodes in the FRA (Frankfurt) and LON (London) zones. It uses the label `topology.kubernetes.io/zone`, which is typically added to nodes by Kubernetes by the cloud provider (in our case, DigitalOcean).

Reminder: You can see the labels of your nodes using `kubectl get nodes --show-labels`.

There are other operators that can be used with node affinity:

- In: The target of the matchExpression is in the set.
- NotIn: The target of the matchExpression is not in the set.
- Exists: The label exists.
- DoesNotExist: The label does not exist.
- Gt: The target of the matchExpression is greater than.
- Lt: The target of the matchExpression is less than.

17.4.3 Node anti-affinity: example

The DoesNotExist operator and the NotIn operator are used to create anti-affinity rules.

For example, we previously used this YAML:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx
5 spec:
6   replicas: 5
7   selector:
8     matchLabels:
9       app: nginx
10  template:
11    metadata:
12      labels:
13        app: nginx
14      spec:
15        containers:
16          - name: nginx
17            image: nginx
18            ports:
19              - containerPort: 80
```

```
20      affinity:
21        nodeAffinity:
22          requiredDuringSchedulingIgnoredDuringExecution:
23            nodeSelectorTerms:
24              - matchExpressions:
25                - key: kubernetes.io/hostname
26                  operator: In
27                  values:
28                    - default-f8wx9
```

As we have the following 3 nodes:

- default-f72fj
- default-f72fo
- default-f8wx9

We can apply the anti-affinity rule to achieve the same effect:

```
1  kubectl delete -f affinity.yaml
2
3  cat <<EOF > anti-affinity.yaml
4  apiVersion: apps/v1
5  kind: Deployment
6  metadata:
7    name: nginx
8  spec:
9    replicas: 5
10   selector:
11     matchLabels:
12       app: nginx
13   template:
14     metadata:
15       labels:
16         app: nginx
17   spec:
```

```
18     containers:
19       - name: nginx
20         image: nginx
21         ports:
22           - containerPort: 80
23   affinity:
24     nodeAffinity:
25       requiredDuringSchedulingIgnoredDuringExecution:
26         nodeSelectorTerms:
27           - matchExpressions:
28             - key: kubernetes.io/hostname
29               operator: NotIn
30               values:
31                 - default-f72fj
32                 - default-f72fo
33 EOF
```

To check where our Pods are running, use the following command:

```
1 kubectl get pods -o=custom-columns=PodName:.metadata.name,NodeName:.spe\
2 c.nodeName | grep nginx
```

You should see all Pods running on a node with the label `hostname: default-f8wx9` and none on the other nodes.

17.4.4 Affinity weight

When creating affinity and anti-affinity rules, there are other options available, such as the `weight` option. Let's distribute our workloads evenly across all nodes in the cluster, assigning different weights to each node.

```
1 cat <<EOF > weight.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: nginx
6 spec:
7   replicas: 5
8   selector:
9     matchLabels:
10    app: nginx
11   template:
12     metadata:
13       labels:
14         app: nginx
15     spec:
16       containers:
17         - name: nginx
18           image: nginx
19           ports:
20             - containerPort: 80
21   affinity:
22     nodeAffinity:
23       preferredDuringSchedulingIgnoredDuringExecution:
24         - weight: 100
25           preference:
26             matchExpressions:
27               - key: kubernetes.io/hostname
28                 operator: In
29                 values:
30                   - default-f72fj
31         - weight: 50
32           preference:
33             matchExpressions:
34               - key: kubernetes.io/hostname
35                 operator: In
36                 values:
```

```
37          - default-f72fo
38      - weight: 25
39      preference:
40          matchExpressions:
41              - key: kubernetes.io/hostname
42                  operator: In
43                  values:
44                      - default-f8wx9
45 EOF
```

There are three node preferences defined, each with a different weight:

- The first preference has a weight of 100 and specifies that the deployment prefers to schedule Pods on a node with the label `kubernetes.io/hostname` matching `default-f72fj`.
- The second preference has a weight of 50 and prefers nodes with the label `kubernetes.io/hostname` matching `default-f72fo`.
- The third preference has a weight of 25 and prefers nodes with the label `kubernetes.io/hostname` matching `default-f8wx9`.

In other words, the Kubernetes scheduler tries to schedule Pods on nodes that match the specified preferences, giving more weight to preferences with higher values. However, if the preferred nodes are unavailable for any reason, the scheduler can still schedule the Pods on other available nodes.

To see where our Pods are running, apply the Deployment.

```
1 kubectl apply -f weight.yaml
2 kubectl get pods -o=custom-columns=PodName:.metadata.name,NodeName:.spe\
3 c.nodeName --sort-by=.spec.nodeName | grep nginx
```

To observe how the scheduler schedules Pods on nodes, you can scale up to 10, 20, or more replicas.

```
1 kubectl scale deployment nginx --replicas=20
2 kubectl get pods -o=custom-columns=PodName:.metadata.name,NodeName:.spe\
3 c.nodeName --sort-by=.spec.nodeName | grep nginx
```

17.4.5 Affinity and anti-affinity types

There are two types of affinity and anti-affinity rules:

- requiredDuringSchedulingIgnoredDuringExecution
- preferredDuringSchedulingIgnoredDuringExecution

We have already used both types in previous examples. Now, let's take a closer look at the differences between them.

- `requiredDuringSchedulingIgnoredDuringExecution`: The Pod can only be scheduled if the rule is met.
- `preferredDuringSchedulingIgnoredDuringExecution`: The scheduler will try to find a node that meets the rule. If a node that matches the rule is unavailable, the scheduler will still schedule the Pod on a different node.

In both cases, the node label rule is ignored during execution if a Pod is already running on a node and the node label changes. The Pod will continue to run on the node, ignoring the new rule. This is why we use the `IgnoredDuringExecution` suffix.

18 Kubernetes: Managing Infrastructure Upgrades and Maintenance Mode

18.1 Why do we need to upgrade our infrastructure?

There are several reasons why we need to upgrade our infrastructure, but the most common one is to keep up with the latest security patches. Additionally, we need to upgrade our infrastructure to take advantage of new features and bug fixes.

In some cases, a required feature may not be available in the current version of Kubernetes. In these instances, upgrading our infrastructure to the latest version is necessary to benefit from the new feature.

18.2 What to upgrade?

In a cluster, there are several components that you may need to upgrade depending on your infrastructure. For instance, you may want to upgrade the Kubernetes version, the operating system of your Nodes, the container runtime, or your Node Pool.

To upgrade the Control Plane, if you are using a managed Kubernetes service, follow the instructions provided by your cloud provider. Your cloud provider will take care of upgrading the Control Plane for you.

If you're running your own Kubernetes cluster, there are various ways to upgrade the Control Plane depending on the tool you're using to manage your cluster. For

example, if you’re using Kubeadm, you’ll need to follow [specific instructions¹¹¹](#) to upgrade the Control Plane.

Upgrading the worker Nodes in your cluster is the most common upgrade case. However, cloud providers manage this if you’re using a managed Kubernetes service. Nonetheless, there are some cases where you need to upgrade the worker Nodes yourself.

Before exploring that case, let’s review how upgrading worker Nodes works.

18.3 Upgrading worker nodes: draining

When upgrading worker nodes, it is important to ensure that the pods running on those Nodes are not affected by the upgrade process. To accomplish this, it is necessary to drain the Nodes before upgrading them.

For example, consider a cluster with two nodes:

- Node 1: default-frp7j
- Node 2: default-frp7r

To begin, let’s explore the Pods running on the Nodes to abstract the next steps.

```
1 export NODE1=default-frp7j
2 export NODE2=default-frp7r
```

Let’s create a Deployment and scale it to 5 replicas:

```
1 kubectl create deployment hello-world --image=gcr.io/google-samples/hello-app:1.0
2
3 kubectl scale deployment hello-world --replicas=5
```

If you want to see which Pod are running on each Node, you can use the following command:

¹¹¹<https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/>

```
1 kubectl get pods -o wide
```

With 5 replicas, there's a big chance that our Pods are running on both Nodes. Let's see which Pods are running on each Node:

```
1 kubectl get pods -o wide --field-selector spec.nodeName=$NODE1  
2 kubectl get pods -o wide --field-selector spec.nodeName=$NODE2
```

If you observe Pods running on only one Node, you can scale the Deployment to 10 replicas and retest. Keep scaling the Deployment until you see Pods running on both Nodes.

Suppose you want to upgrade NODE2. To accomplish this, you must first drain the Node:

```
1 kubectl drain $NODE2
```

Draining is the process of evicting all Pods running on a Node. When you drain a Node, Kubernetes starts evicting every Pod running on the Node. The Pods are then rescheduled on other available Nodes in the cluster, as long as there are enough resources available. Keep this in mind when draining Nodes.

Executing `kubectl drain $NODE2` will typically result in an error due to DaemonSets running on the Node. If you are using DigitalOcean, several networking components run as DaemonSets. To drain the Node correctly, use the `-ignore-daemonsets` flag.

```
1 kubectl drain $NODE2 --ignore-daemonsets
```

If you check the Pods running on the Node now, you will see that there are no Pods currently running on it.

```
1 kubectl get pods -o wide --field-selector spec.nodeName=$NODE2
```

NODE2 is now ready for an upgrade. You can see that the node is in a `SchedulingDisabled` state.

```
1 kubectl get nodes
```

This means that no Pods will be scheduled on the Node until you uncordon it. Now, you can safely upgrade the Node.

18.4 Upgrading worker nodes: cordonning

After upgrading your Node, you need to uncordon it. This means that Pods can once again be scheduled on the Node.

```
1 kubectl uncordon $NODE2
```

You should be able to see that the Node is now in the Ready state.

```
1 kubectl get nodes
```

Pods will be scheduled on the Node again. You can check this by running the following command:

```
1 kubectl get pods -o wide --field-selector spec.nodeName=$NODE2
```

If you do not see any Pods running on the Node, try scaling the Deployment to 10 replicas and check again.

```
1 kubectl scale deployment hello-world --replicas=10  
2 kubectl get pods -o wide --field-selector spec.nodeName=$NODE2
```

18.5 Upgrading Node Pools

Whether you're using a managed Kubernetes service or running your own Kubernetes cluster, you may want to upgrade your node pools. This is often necessary when you want to change your machine type.

Kubernetes node pools are groups of nodes that share the same configuration. For example, if you're using DigitalOcean, you can create a node pool with three nodes of type s-1vcpu-2gb, and another node pool with three nodes of type s-2vcpu-4gb.

To upgrade your node pool, create a new node pool with the new configuration and migrate your workload to it. Once you're finished, you can delete the old node pool.

Suppose we create a new node pool consisting of 3 nodes of type s-2vcpu-4gb.

```
1 doctl kubernetes cluster node-pool create <cluster-id|cluster-name> --size s-2vcpu-4gb --count 3
```

Next, we need to migrate our workload to the new Node Pool. To do this, drain all nodes running on the old Node Pool.

```
1 kubectl drain $NODE1 --ignore-daemonsets
2 kubectl drain $NODE2 --ignore-daemonsets
```

You can now delete the old Node Pool.

18.6 Zero downtime upgrades: Pod Disruption Budgets

When you drain nodes during an upgrade process, it is important to ensure that your workload is not affected. To do this, you need to ensure that there are enough resources available to reschedule your pods on other nodes. In Kubernetes, you can define a Pod Disruption Budget (PDB) to limit the number of pods of a replicated application that can be down simultaneously due to voluntary disruptions. PDBs are not only used for upgrades but also for other cases, such as hardware failures and kernel panics.



A Pod Disruption Budget (PDB) limits the number of Pods of a replicated application that are down simultaneously from voluntary

disruptions. PDB is not just used for upgrades, but also for other cases such as hardware failures, kernel panics, etc. In other words, using PDB is one of the best practices to create highly available applications and ensure zero downtime upgrades.

Assuming you have already deployed a deployment with 5 replicas, here's how you can create a PDB for it:

```
1 kubectl create pdb hello-world --selector app=hello-world --min-available=3
```

The command above creates a PDB (Pod Disruption Budget) for the deployment labeled `app=hello-world`. This ensures that at least three pods are always available.

You can check the PDB by running the following command:

```
1 kubectl get pdb hello-world
```

If you want to use YAML, you must delete the previous PDB before applying the new one.

```
1 kubectl delete pdb hello-world
```

Next, you can use the following manifest:

```
1 kubectl apply -f - <<EOF
2 ---
3 apiVersion: policy/v1
4 kind: PodDisruptionBudget
5 metadata:
6   name: hello-world
7 spec:
8   minAvailable: 3
9   selector:
10    matchLabels:
11      app: hello-world
12 EOF
```

In the above manifest, we are creating a PDB with `minAvailable` set to 3. This ensures that at least 3 Pods will be available at all times. If you prefer to use `maxUnavailable`, you can use the following manifest instead:

```
1  ---
2  kubectl apply -f - <<EOF
3  apiVersion: policy/v1
4  kind: PodDisruptionBudget
5  metadata:
6    name: hello-world
7  spec:
8    maxUnavailable: 2
9    selector:
10      matchLabels:
11        app: hello-world
12 EOF
```

The math here is simple:

- We have 5 replicas.
- We want to ensure that at least 3 pods are available at all times.
- Therefore, we can have up to 2 pods unavailable at any given time.
- This is why we are using either `minAvailable: 3` or `maxUnavailable: 2`.

19 Microservices Deployment Strategies: Managing Application Updates and Deployment

19.1 Cloud Native practices

Cloud Native refers to the set of practices that enable organizations to build and manage applications at scale. Typically, a Cloud Native application consists of multiple microservices that are packaged into containers and deployed as loosely coupled components. These components are managed and orchestrated through declarative APIs.

The scope of Cloud Native extends beyond the design and implementation of applications. It also includes the design and implementation of infrastructure, as well as deployment strategies used to manage the application lifecycle.

With the help of DevOps and DevSecOps practices, Cloud Native organizations can deliver faster and more frequently. They can also adapt to the fast-changing cloud environment and market. To achieve this, organizations need to adopt the right deployment strategies. These strategies will allow them to manage the application lifecycle consistently with their business goals.

This section will discuss the most common deployment strategies.

19.2 Deployment strategies

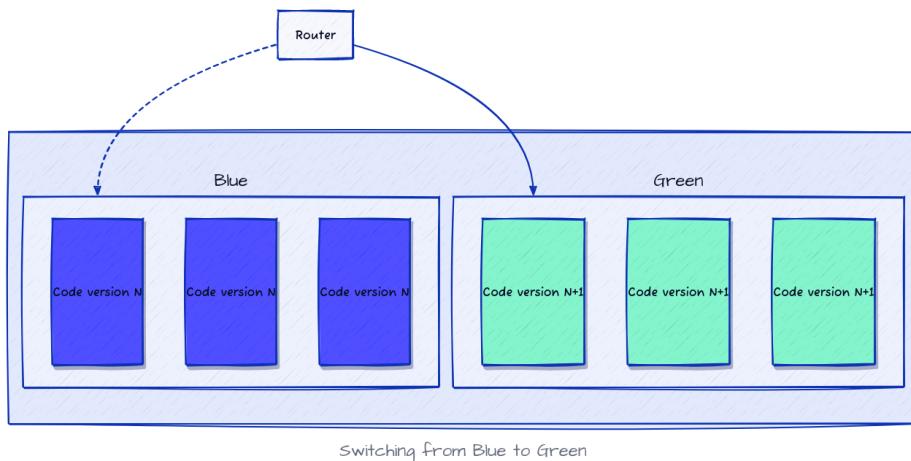
19.2.1 Blue/Green deployment: introduction

The Blue/Green strategy involves deploying a new version of the application (Green version) while keeping the previous version (Blue version) running in production.

This strategy uses a service, such as a router, to direct traffic, allowing the Blue environment to operate seamlessly for end users in production while you test and deploy to the Green environment.

Once deployment and testing are successful, you can switch your router to target the Green environment without any noticeable change for your users.

If any issues are detected, it is easy to roll back to the Blue version.



For example, you have the Blue environment that is already running in production.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: blue-app
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: blue-app
10  template:
11    metadata:
12      labels:
13        app: blue-app
14    spec:
15      containers:
16        - name: app-container
17          image: myregistry/blue-app:1.0
```

To deploy the new version of the application, you need to replicate the production environment (Blue) to create a Green environment.

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: green-app
5 spec:
6   replicas: 3
7   selector:
8     matchLabels:
9       app: green-app
10  template:
11    metadata:
12      labels:
13        app: green-app
14    spec:
15      containers:
```

```
16      - name: app-container  
17      image: myregistry/green-app:1.0
```

Once the Green environment has the new version of the application, test it. If testing is successful, switch your router to target the Green environment. If any issues are detected, roll back to the Blue version.

Implementing the Blue/Green strategy can be facilitated by using advanced tools. Here are some examples:

- Istio: A service mesh that provides observability, security, and management features to speed up deployment cycles. Istio is a popular choice for implementing the blue/green strategy.
- Spinnaker: An open source, multi-cloud continuous delivery platform for releasing software changes.
- Jenkins: An open source automation server that supports the blue/green strategy.
- AWS CodeDeploy: A fully managed deployment service that automates software deployments to a variety of compute services such as Amazon EC2, AWS Fargate, AWS Lambda, and on-premises servers.
- Google Cloud Deploy: An infrastructure deployment service that automates the creation and management of Google Cloud resources.

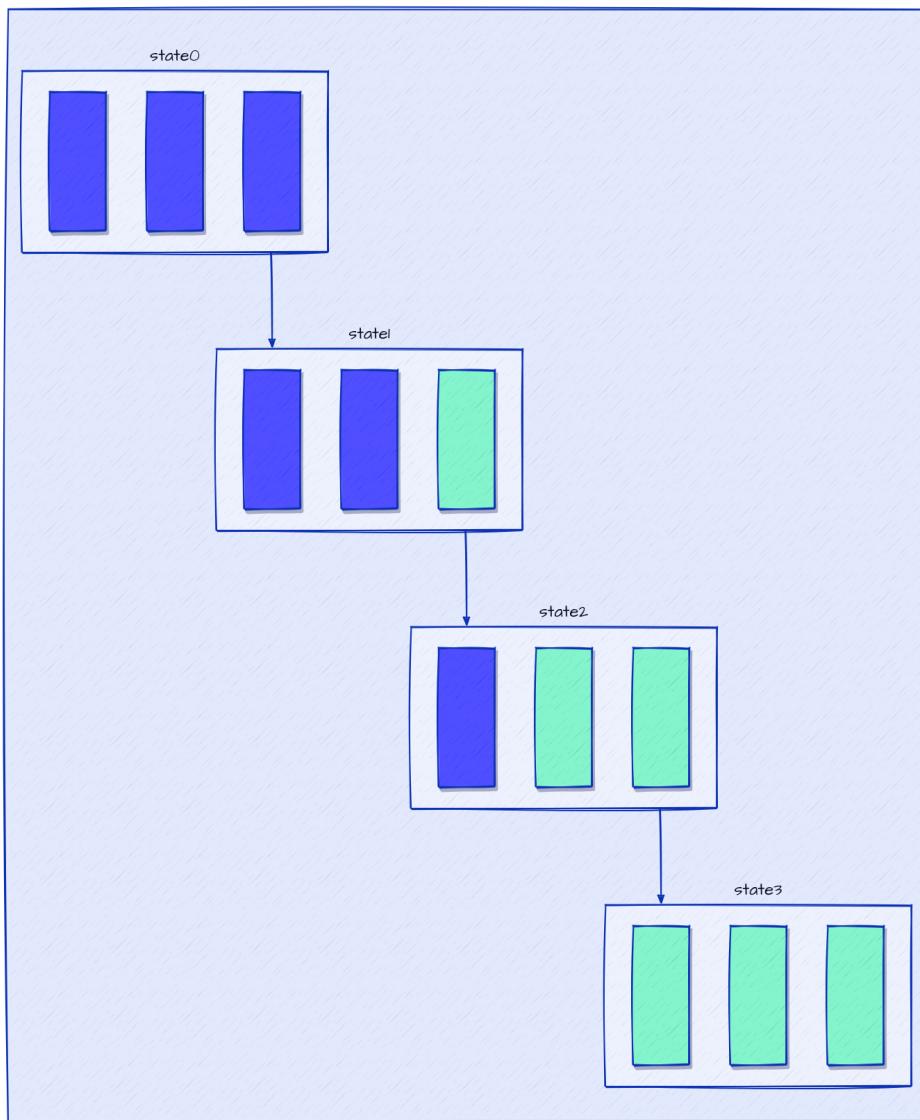
The Blue/Green strategy is a popular choice for organizations. It enables zero-downtime deployments, safe rollbacks, testing in production, and better isolation. However, maintaining two identical production environments can be expensive. Moreover, deploying a new version of the application may take a long time. Blue/Green deployments are not suitable for applications that require complex or large-scale data migrations.

19.2.2 Canary deployment: introduction

The canary strategy involves deploying a new version of an application to a small subset of users (typically 5-10%) to test it in real conditions while minimizing disruption for the majority of users. If the new version proves to be stable, it can

be rolled out to all users. If any issues are detected, the previous version can be restored without affecting the entire user base.

Canary deployment operates similarly to blue-green deployment but follows a slightly different approach. Instead of having another complete environment waiting to be switched once the deployment is complete, canary deployments first switch a small subset of servers or nodes before proceeding with the remaining ones.



The Canary strategy enables controlled testing in production without creating

a new environment, unlike the Blue/Green strategy. This allows for faster deployment of the new version of the application and reduces maintenance costs. It provides a zero-downtime deployment and allows for a simple rollback. However, it has the same complexity as the Blue/Green strategy, especially when it comes to managing complex data migrations.

The Canary strategy can be performed manually or with the help of advanced tools such as Istio, Spinnaker, Jenkins, AWS CodeDeploy, and Google Cloud Deploy.

19.2.3 Canary deployment: an example using Istio

We are going to perform a Canary deployment using Istio.

First, we need to install Istio. To download version 1.17.2 of Istio for the x86_64 architecture, use the following command (you can use the command `uname -m` to get the architecture of your machine):

```
1 curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.17.2 TARGET_ARCH=x86_64 sh -
```

Next, set the PATH environment variable to the “bin” directory of the Istio installation directory.

```
1 cd istio-1.17.2
2 export PATH=$PWD/bin:$PATH
```

Perform a pre-check of your environment by running the following command:

```
1 istioctl x precheck
```

Continue the Istio installation by executing the following command:

```
1 istioctl install --set profile=demo -y
```

We are using the profile `demo` which is a good starting point for most deployments. You can find more information about the different profiles [here](#)¹¹².

The command above installs Istio and creates the necessary resources in the Kubernetes cluster, including an Istio Ingress Gateway. As a result, we no longer need the NGINX Ingress Controller that was installed in the previous section. Let's delete it:

```
1 helm delete nginx-ingress
```

To enable automatic injection of Envoy sidecar proxies when deploying your application later, add a namespace label to instruct Istio.

```
1 kubectl label namespace default istio-injection=enabled
```



Sidecar containers in Kubernetes are additional containers that run alongside the main container within a Pod. They are designed to enhance and extend the functionalities of the main container without modifying its codebase.

Sidecar containers share resources like storage and network interfaces with the main container, allowing them to work together seamlessly. They can be developed in different languages and provide increased flexibility.

Common use cases for sidecar containers include sharing storage or networks, performing functionality more efficiently in another programming language, and integrating logging and monitoring applications to increase observability and performance.

In the context of Istio, a sidecar refers to an additional container that is injected into a Pod alongside the main application container. The Istio sidecar container is responsible for intercepting network traffic to and from the main container. It enables advanced service mesh features such as traffic management, security, and observability. Acting as a proxy, it implements the data plane functionality of Istio.

¹¹²<https://istio.io/latest/docs/setup/additional-setup/config-profiles/>

The sidecar container is automatically injected by Istio's sidecar injector, which modifies the Pod specification to include the sidecar container. This architecture enables Istio to provide powerful capabilities such as traffic routing, load balancing, circuit breaking, and telemetry without requiring changes to the application's code.

Let's create a new workspace and deploy our test application:

```
1 cd $HOME && mkdir canary && cd canary
```

The idea is to deploy two versions of the application: v1 and v2. Istio will be used to route 90% of the traffic to v1 and 10% to v2. The new version will be tested, and if everything is fine, 100% of the traffic will be routed to v2.

Here is the first version of the application:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: helloapp-v1
5 spec:
6   replicas: 1
7   selector:
8     matchLabels:
9       app: helloapp
10      version: v1
11 template:
12   metadata:
13     labels:
14       app: helloapp
15      version: v1
16 spec:
17   containers:
18     - name: hello-app-v1
19       image: gcr.io/google-samples/hello-app:1.0
20       imagePullPolicy: Always
```

```
21      ports:  
22          - containerPort: 8080
```

This is the second version of the application:

```
1  apiVersion: apps/v1  
2  kind: Deployment  
3  metadata:  
4      name: helloapp-v2  
5  spec:  
6      replicas: 1  
7      selector:  
8          matchLabels:  
9              app: helloapp  
10             version: v2  
11      template:  
12          metadata:  
13              labels:  
14                  app: helloapp  
15                  version: v2  
16      spec:  
17          containers:  
18              - name: hello-app-v2  
19                  image: gcr.io/google-samples/hello-app:2.0  
20                  imagePullPolicy: Always  
21          ports:  
22              - containerPort: 8080
```

We will use the following service to expose the application:

```
1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: helloapp
5 spec:
6   selector:
7     app: helloapp
8   ports:
9     - protocol: TCP
10    port: 80
11    targetPort: 8080
```

Up to this point, nothing has changed from what we did in the previous section. We now have two versions of the application deployed and exposed through a service. If we had an Ingress Controller, the application would use both versions without any specific weight distribution.

Let's move on to the next step and create an Istio Gateway resource. But first, we need to obtain the IP address of the Istio Ingress Gateway:

```
1 export INGRESS_IP=$(kubectl get svc -n istio-system istio-ingressgatewa\
2 y -o jsonpath='{ .status.loadBalancer.ingress[0].ip}'
```

Now, we can use this YAML:

```
1 apiVersion: networking.istio.io/v1alpha3
2 kind: Gateway
3 metadata:
4   name: istio-gateway
5 spec:
6   selector:
7     istio: ingressgateway
8   servers:
9     - port:
10        number: 80
11        name: http
```

```
12     protocol: HTTP
13   hosts:
14     - "app.${INGRESS_IP}.nip.io"
```



The Gateway is an Istio custom resource.

The reason we need to add the Gateway resource in Istio is to define the routing rules for incoming traffic to the Istio service mesh. While we already have an Istio Ingress Gateway installed, it acts as a default entry point for external traffic. However, without explicitly defining a Gateway resource, the Ingress Gateway will not know how to handle the incoming traffic and route it to the appropriate services within the service mesh.

In other words, the Gateway is an extension of the Ingress Gateway that allows more control and granularity over configuring the routing rules as well as other features.

Next, we need to create a VirtualService resource to define the routing rules for the application:

```
1 apiVersion: networking.istio.io/v1alpha3
2 kind: VirtualService
3 metadata:
4   name: helloapp
5 spec:
6   hosts:
7     - "app.${INGRESS_IP}.nip.io"
8   gateways:
9     - istio-gateway
10  http:
11    - route:
12      - destination:
13        host: helloapp
14        subset: v1
15        weight: 90
```

```
16      - destination:  
17          host: helloapp  
18          subset: v2  
19          weight: 10
```

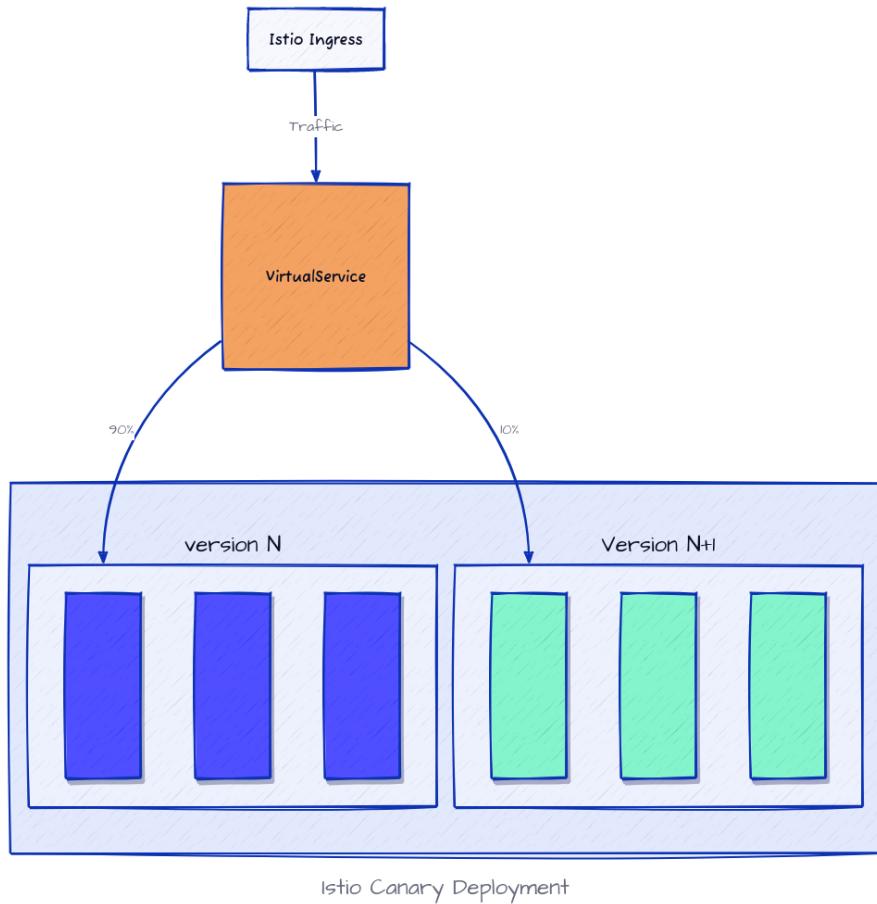
The above YAML defines a VirtualService in Istio, which is used to configure traffic routing and request handling rules for specific hosts.

In this case, the VirtualService is named `helloapp` and is configured to handle traffic for the host `app.${INGRESS_IP}.nip.io`.

The VirtualService is associated with the `istio-gateway` gateway, which means it will be applied to traffic received through that gateway (which we already created in the previous step).

For HTTP traffic, the VirtualService defines two routes using the “route” field:

- The first route directs 90% of the traffic to the “v1” subset of the “helloapp” service.
- The second route directs 10% of the traffic to the “v2” subset of the same “helloapp” service.



Once you define the VirtualService, the next step is to create a DestinationRule resource. This will define the subsets of the application.

```
1 apiVersion: networking.istio.io/v1alpha3
2 kind: DestinationRule
3 metadata:
4   name: helloapp
5 spec:
6   host: helloapp
7   subsets:
8     - name: v1
9       labels:
10      version: v1
11     - name: v2
12       labels:
13         version: v2
```



The DestinationRule in Istio is used to define policies and settings for how traffic is routed to specific subsets of a service.

The provided YAML defines a DestinationRule named `helloapp`. The host `helloapp` is specified as the destination for this rule.

The DestinationRule defines two subsets, `v1` and `v2`. Each subset is identified by a name and associated with labels that match the labels defined in the corresponding VirtualService.

The purpose of the DestinationRule, in conjunction with the VirtualService, is to enable fine-grained control over traffic routing and load balancing to the different subsets of a service. Specifically, these subsets are named `v1` and `v2`. In other words, the VirtualService directs traffic based on the defined weights, while the DestinationRule provides additional configuration for each subset.

Taking into consideration the above manifests, we can put everything together in a single YAML file:

```
1 cat <<EOF > app.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: helloapp-v1
6 spec:
7   replicas: 1
8   selector:
9     matchLabels:
10    app: helloapp
11    version: v1
12 template:
13   metadata:
14     labels:
15       app: helloapp
16       version: v1
17   spec:
18     containers:
19       - name: hello-app-v1
20         image: gcr.io/google-samples/hello-app:1.0
21         imagePullPolicy: Always
22       ports:
23         - containerPort: 8080
24   ---
25 apiVersion: apps/v1
26 kind: Deployment
27 metadata:
28   name: helloapp-v2
29 spec:
30   replicas: 1
31   selector:
32     matchLabels:
33       app: helloapp
34       version: v2
35 template:
36   metadata:
```

```
37     labels:
38         app: helloapp
39         version: v2
40
41     spec:
42         containers:
43             - name: hello-app-v2
44                 image: gcr.io/google-samples/hello-app:2.0
45                 imagePullPolicy: Always
46                 ports:
47                     - containerPort: 8080
48
49     ---
50
51     apiVersion: v1
52     kind: Service
53
54     metadata:
55         name: helloapp
56
57     spec:
58         selector:
59             app: helloapp
60
61         ports:
62             - protocol: TCP
63                 port: 80
64                 targetPort: 8080
65
66     ---
67
68     apiVersion: networking.istio.io/v1alpha3
69     kind: Gateway
70
71     metadata:
72         name: istio-gateway
73
74     spec:
75         selector:
76             istio: ingressgateway
77
78         servers:
79             - port:
80                 number: 80
81                 name: http
82                 protocol: HTTP
83
84         hosts:
```

```
73         - "app.${INGRESS_IP}.nip.io"
74     ---
75     apiVersion: networking.istio.io/v1alpha3
76     kind: VirtualService
77     metadata:
78       name: helloapp
79     spec:
80       hosts:
81         - "app.${INGRESS_IP}.nip.io"
82       gateways:
83         - istio-gateway
84       http:
85         - route:
86           - destination:
87             host: helloapp
88             subset: v1
89             weight: 90
90           - destination:
91             host: helloapp
92             subset: v2
93             weight: 10
94     ---
95     apiVersion: networking.istio.io/v1alpha3
96     kind: DestinationRule
97     metadata:
98       name: helloapp
99     spec:
100       host: helloapp
101       subsets:
102         - name: v1
103           labels:
104             version: v1
105         - name: v2
106           labels:
107             version: v2
108 EOF
```

Now, we can apply the manifest:

```
1 kubectl apply -f app.yaml
```

To test how the traffic is routed to the different subsets, we can use the curl command:

```
1 while true; do curl -s http://app.${INGRESS_IP}.nip.io ; sleep 1; echo;\\
2 done
```

You should be able to see the output of the application with the version number changing between v1 and v2:

```
1 Hello, world!
2 Version: 1.0.0
3 Hostname: helloapp-v1-<pod-id>
4
5 Hello, world!
6 Version: 2.0.0
7 Hostname: helloapp-v2-<pod-id>
```

Since we assigned different weights to the two subsets, the application's output should consist mostly of v1, with some v2 mixed in.

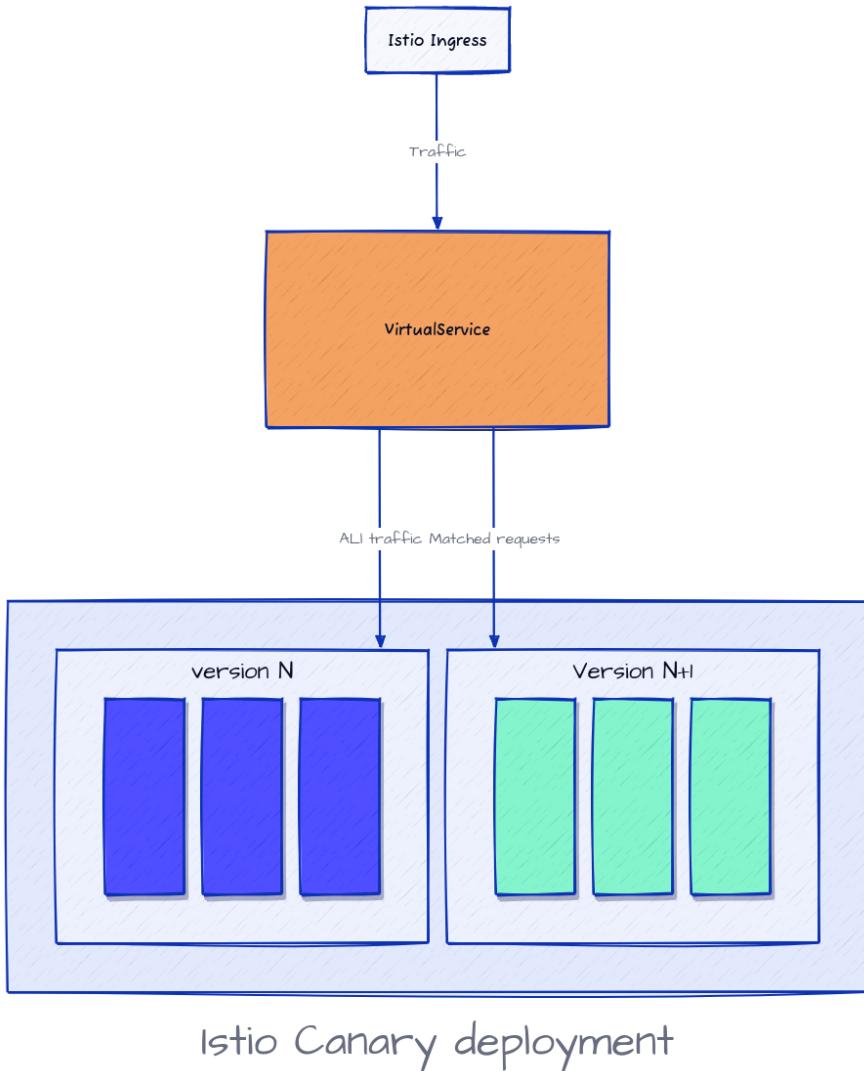
19.2.4 Canary Deployment: testing in production

It is possible to perform testing in production by using a prefix-based routing rule. In the following example, only requests with the header internal-testing: True will be routed to the v2 subset, while all other requests will be routed to the v1 subset.

```
1 kubectl apply -f - <<EOF
2 ---
3 apiVersion: networking.istio.io/v1alpha3
4 kind: VirtualService
5 metadata:
6   name: helloapp
7 spec:
8   hosts:
9     - "app.${INGRESS_IP}.nip.io"
10  gateways:
11    - istio-gateway
12  http:
13    - match:
14      - headers:
15        internal-testing:
16          exact: "True"
17    route:
18      - destination:
19        host: helloapp
20        subset: v2
21    - route:
22      - destination:
23        host: helloapp
24        subset: v1
25 EOF
```

Now you can test with and without the header `internal-testing: True`:

```
1 curl -s -H "internal-testing: True" http://app.${INGRESS_IP}.nip.io
2 curl -s http://app.${INGRESS_IP}.nip.io
```



More matching rules can be found in the [Istio documentation](#)¹¹³.

¹¹³<https://istio.io/latest/docs/reference/config/networking/virtual-service/#HTTPMatchRequest>

19.3 Rolling updates: definition

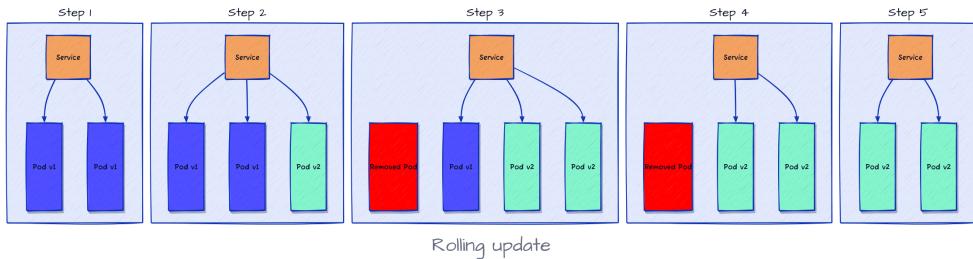
The Rolling Update strategy is a commonly used deployment strategy that involves gradually replacing instances of the old version of an application with instances of the new version. It is typically used for deploying applications in a Kubernetes environment.

In Kubernetes, the Deployment resource is used to implement the rolling update strategy. This resource defines the desired state of the application, while the ReplicaSet maintains the actual state of the application.

With the Rolling Update strategy, instances of an application are gradually updated to a new version, ensuring minimal disruption to the overall application availability.

Here are the steps involved in a rolling update:

1. The deployment begins by creating new instances of the updated version alongside the existing instances of the previous version.
2. The load balancer or router directs a portion of the incoming traffic to the new instances while still forwarding some traffic to the existing instances.
3. Kubernetes monitors the health and performance of the new instances to ensure they are functioning properly.
4. If the new instances are performing well and meeting the desired criteria, the deployment process gradually shifts more traffic towards the new instances and reduces traffic to the old instances.
5. As more traffic is routed to the new instances, Kubernetes monitors for any issues or errors and automatically rolls back to the previous version if necessary.
6. This process continues until all instances have been updated to the new version.
7. Once the deployment is complete, the old instances are removed.
8. The deployment is complete, and the new version is now running. Kubernetes continues to monitor the health and performance of the new instances to ensure they are functioning properly.



In this section, we will deploy a new version of the application and perform a rolling update.

19.3.1 Rolling updates: example

We are going to remove Istio from the cluster:

```
1 istioctl uninstall --purge
2 kubectl delete namespace istio-system
```

Then we are going to install NGINX Ingress Controller using Helm:

```
1 helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
2 helm repo update
3 helm install nginx-ingress ingress-nginx/ingress-nginx --set controller\
4 .publishService.enabled=true
```

Create a new working directory:

```
1 cd $HOME && mkdir -p rollingupdate && cd rollingupdate
```

Check when the external Ingress IP is no longer in the pending state using `kubectl get svc nginx-ingress-ingress-nginx-controller`.

Once we have the external IP, we can export it as an environment variable:

```
1 export INGRESS_IP=$(kubectl get svc nginx-ingress-ingress-nginx-control\\
2 ler -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
```

Then we are going to deploy the first version of the application, the Kubernetes Service, and the Ingress resource:

```
1 cat <<EOF > app.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: hello-app
6 spec:
7   replicas: 3
8   selector:
9     matchLabels:
10    app: hello-app
11 template:
12   metadata:
13     labels:
14       app: hello-app
15   spec:
16     containers:
17       - image: gcr.io/google-samples/hello-app:1.0
18         imagePullPolicy: Always
19       name: hello-app
20       ports:
21         - containerPort: 8080
22 ---
23 apiVersion: v1
24 kind: Service
25 metadata:
26   name: hello-app
27 spec:
28   selector:
29     app: hello-app
30   ports:
```

```
31     - protocol: TCP
32         port: 80
33         targetPort: 8080
34     ---
35 apiVersion: networking.k8s.io/v1
36 kind: Ingress
37 metadata:
38     name: hello-app
39 spec:
40     rules:
41         - host: app.${INGRESS_IP}.nip.io
42             http:
43                 paths:
44                     - path: /
45                     pathType: Prefix
46                 backend:
47                     service:
48                         name: hello-app
49                         port:
50                             number: 80
51             ingressClassName: nginx
52 EOF
```

To release a new version (`gcr.io/google-samples/hello-app:2.0`) and follow the rolling update process, you must first add the `RollingUpdate` strategy to the Deployment manifest.

```
1 strategy:
2 type: RollingUpdate
3 rollingUpdate:
4     maxSurge: 1
5     maxUnavailable: 0
```

By setting `maxSurge: 1`, a new version of the application can be deployed by gradually replacing instances of the old version with instances of the new version.

With `maxSurge: 2`, a new version of the application can be deployed by gradually replacing instances of the old version with instances of the new version, while deploying 2 new instances at the same time.

When `maxUnavailable: 0`, the number of unavailable Pods during the deployment should not exceed 0. This ensures that the application's functionality cannot be interrupted during the deployment.

However, when `maxUnavailable: 1`, the number of unavailable Pods during the deployment could be 1. This means that the application's functionality can be interrupted during the deployment.

Percentages can also be used instead of numbers. For example, `maxSurge: 50%` means that a new version of the application can be deployed by gradually replacing instances of the old version with instances of the new version, deploying 50% of the new instances at the same time.

```
1 strategy:
2   type: RollingUpdate
3   rollingUpdate:
4     maxSurge: 1
5     maxUnavailable: 50%
```

To indicate that a Pod is ready, Kubernetes requires a `readinessProbe` to be added to the container. Once the container is ready, Kubernetes will deem the Pod ready and begin sending traffic to it.

```
1 readinessProbe:
2   httpGet:
3     path: /
4     port: 8080
5     scheme: HTTP
6   initialDelaySeconds: 20
7   timeoutSeconds: 10
8   periodSeconds: 10
```

In the above example, we instruct Kubernetes to check the `/` path on port 8080 every 10 seconds. The first check will be performed after 20 seconds. If the check fails, Kubernetes will wait for 10 seconds before attempting another check.

This is how it works:

- The `readinessProbe` waits for `initialDelaySeconds` before starting to check if the Pod is ready.
- If the operation exceeds `timeoutSeconds`, the Pod is considered not ready.
- If the check is successful, the Pod is considered ready.
- After `successThreshold` successful checks, the Pod is considered ready.
- If the check fails `failureThreshold` times, the Pod is considered not ready.
- This check is repeated every `periodSeconds` seconds.

As a consequence, this is the new Deployment manifest:

```
1 cat <<EOF > v1.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: hello-app
6   namespace: default
7 spec:
8   replicas: 3
9   selector:
10    matchLabels:
11      app: hello-app
12   strategy:
13     type: RollingUpdate
14     rollingUpdate:
15       maxSurge: 1
16       maxUnavailable: 0
17   template:
18     metadata:
19       labels:
20         app: hello-app
21     spec:
22       containers:
23         - image: gcr.io/google-samples/hello-app:1.0
```

```
24     imagePullPolicy: Always
25     name: hello-app
26     ports:
27       - containerPort: 8080
28     readinessProbe:
29       httpGet:
30         path: /
31         port: 8080
32         scheme: HTTP
33       initialDelaySeconds: 1
34       timeoutSeconds: 2
35       periodSeconds: 10
36     ---
37   apiVersion: v1
38   kind: Service
39   metadata:
40     name: hello-app
41     namespace: default
42   spec:
43     selector:
44       app: hello-app
45     ports:
46       - protocol: TCP
47         port: 80
48         targetPort: 8080
49     ---
50   apiVersion: networking.k8s.io/v1
51   kind: Ingress
52   metadata:
53     name: hello-app
54   spec:
55     rules:
56       - host: app.${INGRESS_IP}.nip.io
57         http:
58           paths:
59             - path: /
```

```
60      pathType: Prefix
61      backend:
62          service:
63              name: hello-app
64              port:
65                  number: 80
66      ingressClassName: nginx
67 EOF
```

Apply the manifest:

```
1 kubectl apply -f v1.yaml
```

Now let's update the application image to version 2.0:

```
1 kubectl set image deployment/hello-app hello-app=gcr.io/google-samples/\
2 hello-app:2.0
```

Observe how the Pods are being updated:

```
1 kubectl get pods -w
```

Behold the rolling updates, where old Pods gently fade away, making space for the new ones:

```
1 kubectl rollout status deployment/hello-app
```

You can also check the rollout history:

```
1 kubectl rollout history deployment hello-app
```

Feeling nostalgic? Time travel is possible! Roll back to a previous version:

```
1 kubectl rollout undo deployment hello-app
```

If you want to roll back to a specific revision, you can use the `--to-revision` flag:

```
1 kubectl rollout undo deployment hello-app --to-revision=1
```

You can also pause and resume a rollout:

```
1 kubectl rollout pause deployment hello-app  
2 kubectl rollout resume deployment hello-app
```

If you want to add the change caused to the rollout history, you can use:

```
1 kubectl patch deployment hello-app -p '{"spec": {"template": {"metadata": \\\n2 {"annotations": {"kubernetes.io/change-cause": "Featuring an epic update"\\\n3 }}}}}'
```

Check the rollout history again to see the change cause:

```
1 kubectl rollout history deployment hello-app
```

20 Microservices Observability in a Kubernetes World: Introduction

20.1 Introduction to observability

In recent years, microservices and distributed architectures have become popular models for building and serving applications. While this approach increases scalability, it also results in growing complexity and dynamicity within the system. The increased interaction between services in a microservice architecture makes it difficult to understand, identify, and resolve abnormal behaviors in their IT environments. Professionals have explored concepts such as monitoring and observability to address these issues.

20.2 What is monitoring?

Monitoring in IT refers to the periodic tracking of software and infrastructure performance by systematically collecting, analyzing, and implementing data from the system. The purpose of monitoring is to determine how well your software and underlying infrastructure perform in real-time, ensuring that performance levels meet expectations. Monitoring IT environments involves using a combination of tools and technologies to establish simultaneous infrastructure performance and resolution of identified issues. While monitoring is a well-known concept among developers, DevOps engineers, and IT professionals, observability is just gaining momentum.

20.3 What is observability?

Observability is a broad field that encompasses various aspects of system monitoring and analysis. By definition, observability refers to the ability to observe a system, understand what is happening inside, and make informed decisions based on that understanding.

The term “observability” originated in the 20th century, but was first used in IT by Twitter’s engineering team in 2013 in a blog post describing how they monitor their systems. Since then, it has become a popular topic in the software industry. (source: [Encyclopedia¹¹⁴](#), [Twitter Blog¹¹⁵](#))

The concept of Site Reliability Engineering (SRE) was further popularized, particularly in the field of microservices and distributed systems, by other companies such as Google, Uber, and Netflix. Google engineer Rob Ewaschuk also contributed to this effort in the book [SRE Book¹¹⁶](#).

20.4 White-box monitoring vs black-box monitoring

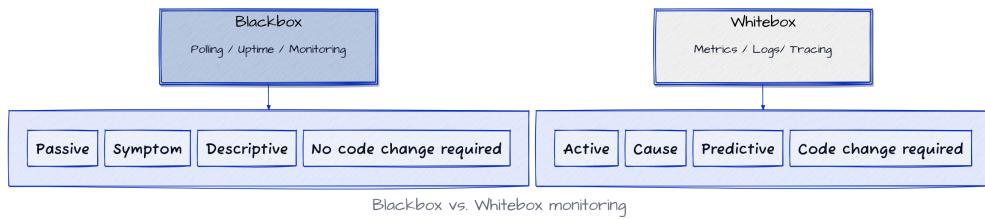
Whitebox monitoring involves monitoring the internal metrics of applications running on a server. This type of monitoring tracks the number of HTTP requests to a server, MySQL queries running on a database server, and similar metrics.

Blackbox monitoring, on the other hand, refers to the monitoring of servers and server resources such as CPU usage, memory, and load averages. In modern IT environments, application developers, DevOps engineers, and SREs share the responsibilities of both Whitebox and Blackbox monitoring.

¹¹⁴<https://www.encyclopedia.com/history/encyclopedias-almanacs-transcripts-and-maps/kalman-rudolf-emil?ref=faun>

¹¹⁵https://blog.twitter.com/engineering/en_us/a/2013/observability-at-twitter

¹¹⁶<https://sre.google/sre-book/monitoring-distributed-systems/>



Both white-box and black-box monitoring are useful for gaining insights into the application and underlying infrastructure and ensuring optimal performance.

Observability is a function of white-box monitoring. White-box monitoring targets logs, metrics, and tracing of external events in a system, which are the pillars of observability.

20.5 Pillars of observability

To understand why a distributed system composed of different microservices develops a fault or behaves in a certain arbitrary way and achieve observability, you need to gather and examine telemetry data from each service in the system.

Logs, metrics, and tracing are the three types of telemetry data widely referred to as the pillars of observability.

20.5.1 Logs

Logs are messages structured or unstructured generated by a system when certain code runs. They provide comprehensive records of events in a system, detailing an event such as an error, its location, occurrence time, and the reason it occurred. In a microservices environment, logs are especially useful in uncovering the details of unknown faults or emergent behaviors.

Analyzing log data is crucial to achieving observability. By examining log details, you can debug and troubleshoot the location, timing, and cause of an error in the system.

20.5.2 Metrics

Metrics are collective values that represent the aggregate performance of a system over a period of time. Unlike logs, metrics provide a holistic view of a system's events and performance.

You can gather metrics such as system uptime, number of requests, response time, failure rate, memory usage, and other resource usage over time. Engineers typically use metrics to trigger alerts or certain actions when the metric value goes above or below a specified threshold.

Metrics are also easy to correlate across multiple systems, allowing you to observe trends in performance and identify issues in the system.

20.5.3 Tracing

The third component of observability, tracing, involves tracking the root source of faults, especially in distributed systems. Tracing records the journey of a request or action as it moves from one service to another in a microservice architecture. This enables professionals to identify system bottlenecks and resolve issues faster.

Tracing is particularly useful when debugging complex applications, as it allows us to understand a request's journey from its starting point and identify the service from which a fault originated in a microservice architecture. Although the first two components, logs, and metrics, provide adequate information about a system's behavior and performance, tracing enhances this information by providing helpful insight into the lifecycle of requests in the system.

20.5.4 Observability pillars in action

From these three pieces of data, you can estimate or determine the current state of an IT system without further investigation, which makes the system observable. However, these pillars are only components of observability and are not actionable enough.

To implement observability in your system, the Twitter engineering team highlighted four actionable steps:

- **Collection:** Aggregating telemetry data, logs, metrics, and tracing, with their unique identifiers and timestamps from various endpoints in the system.
- **Storage:** Storing the collected data in a database responsible for filtering, validating, indexing, and reliably storing the data for future use.
- **Query:** Querying relevant information from the storage system to use the collected and stored data.
- **Visualization:** Visualizing the stored data by querying and displaying it in charts and dashboards for analysis. By analyzing the visualized telemetry data, you can achieve observability in any system.

“While collecting and storing the data is important, it is of no use to our engineers unless it is visualized in a way that can immediately tell a relevant story.” ~ Twitter Observability team

20.6 Four golden signals of monitoring

Monitoring involves tracking and recording various metrics from an environment. According to [Google](#)¹¹⁷, there are four golden signals of monitoring:

- Latency
- Traffic
- Errors
- Saturation

These key metrics can help you achieve optimal performance in your system when measured properly.

20.6.1 Latency

Latency is the time it takes for a system to respond to a request. Separating the latency of failed requests from successful requests is also important, as it can aid in diagnosing failures more accurately.

¹¹⁷<https://sre.google/sre-book/monitoring-distributed-systems/>

20.6.2 Traffic

Traffic is a measure of the amount of service requests sent to a system over a period of time. The nature of these requests may vary depending on the type of service the system provides. For example, traffic for a web service is measured in HTTP requests per second, while for an audio streaming system, it is measured by network I/O rate or concurrent sessions. In a database or key-value storage system, it is measured in the number of transactions or retrievals per second. Measuring traffic is important because it helps you understand the workload on your system.

20.6.3 Errors

Rather than simply filtering out errors, recording errors encountered by users can help improve the system. To do this, monitor the rate of requests that fail, and store the type of request and the latency.

20.6.4 Saturation

Saturation is a measure of how “full” a system is. A system is considered saturated when its underlying resources, such as memory, I/O, and CPU, cannot handle any further requests. Saturation is a valuable metric because it allows you to test the limit of how much traffic or workload your system can handle. It also helps predict how your resources will be utilized over time.

20.7 Monitoring vs Observability: what's the difference?

Monitoring is a prerequisite for observability. While monitoring deals with collecting data, observability involves collecting, storing, querying, and visualizing this data. This makes it easier for professionals to understand the reasons behind a system’s behavior.

This “data” is also different. Monitoring data is usually limited to specific metrics, while observability data is more about measuring deeper insights into the system’s behavior.

Monitoring provides information about problems or failures in your system, while observability helps you understand what caused the failure, where it occurred, and why it happened. A system that is not monitored cannot be observed.

21 Microservices Observability in a Kubernetes World: Prometheus, Grafana, Loki, Promtail, OpenTelemetry, and Jaeger

21.1 Introduction to Prometheus

Prometheus¹¹⁸ is an open-source monitoring and alerting system that records real-time metrics in a time series database. Originally developed by SoundCloud¹¹⁹ in 2012, it was open-sourced in 2015.

Prometheus is a popular tool for monitoring Kubernetes, microservices, containerized applications, and other Cloud Native workloads. It is a graduated project of the Cloud Native Computing Foundation (CNCF) and is widely used by companies such as Google, DigitalOcean, and Red Hat, among many others.

21.2 How Prometheus works

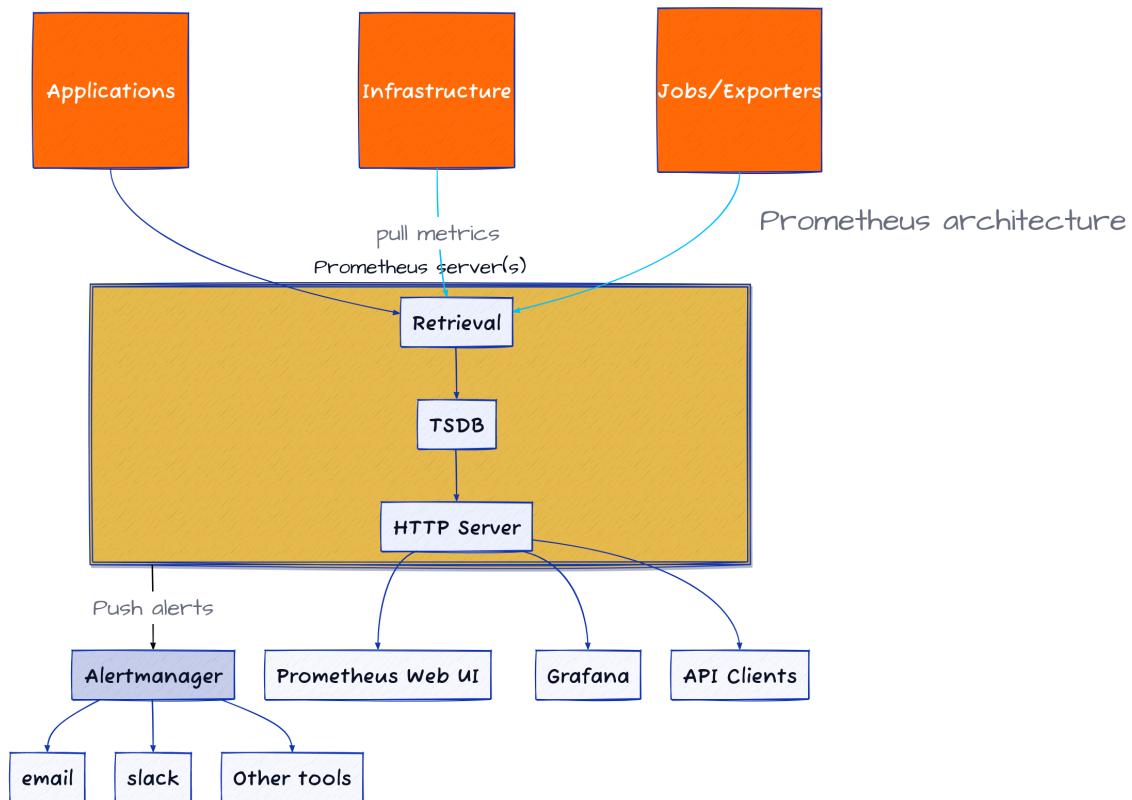
Prometheus collects metrics at regular intervals from configured targets, evaluates rule expressions, displays the results, and can trigger alerts when specified conditions are observed.

The following features distinguish Prometheus from other monitoring and metrics systems:

¹¹⁸<https://github.com/prometheus/prometheus>

¹¹⁹<https://soundcloud.com/>

- A multidimensional data model that defines time series by a metric name and a set of key/value dimensions
- **PromQL**¹²⁰, which is a powerful and flexible query language that leverages this dimensionality
- No dependency on distributed storage
- An HTTP pull model is used for time series collection
- Targets are discovered via service discovery or static configuration
- Support for multiple modes of graphing and dashboards



Prometheus works in the following step-by-step process:

¹²⁰<https://prometheus.io/docs/prometheus/latest/querying/basics/>

1. **Data collection:** Prometheus collects metrics from various targets, such as applications, services, and systems, using the pull model. In this model, the Prometheus server periodically scrapes metrics from the targets over HTTP. The targets can expose metrics in a specific format, such as the Prometheus exposition format or through third-party exporters.
2. **Metric Storage:** It stores collected metrics in its time-series database. Each metric is stored as a time-series data point, with a timestamp and corresponding value. Prometheus has a custom storage engine optimized for time-series data, which enables efficient querying and retrieval of metrics and is one of its most powerful features.
3. **Querying and Processing:** Prometheus provides a powerful query language, called Prometheus Query Language (PromQL), for analyzing and processing collected metrics. With PromQL, users can perform operations such as filtering, aggregation, mathematical calculations, and more. This tool is useful for extracting meaningful insights from the metrics.
4. **Alerting:** It includes a built-in alerting system that enables users to define alert rules based on specific conditions or thresholds. Engineers can configure alerting rules using PromQL expressions. When the conditions specified in the alerting rules are met, Prometheus triggers alerts and sends notifications to specified alerting channels, such as Slack or email.
5. **Visualization and monitoring:** Prometheus provides a web-based expression browser called the Prometheus UI. This allows users to visualize metrics in real-time, explore data, and debug issues. Prometheus also integrates with other visualization tools like Grafana, which is a popular open-source visualization tool.

21.3 Installing Prometheus

This section explains one way to install Prometheus on Kubernetes using the Prometheus Operator.



Operators are defined as software extensions to Kubernetes that make use of custom resources to manage applications and their components.

To install Prometheus, we will be using the [Prometheus Operator](#)¹²¹.

Create a working directory and navigate into it:

```
1 cd $HOME && mkdir -p monitoring && cd monitoring
```

Clone the Prometheus Operator repository:

```
1 git clone https://github.com/prometheus-operator/kube-prometheus.git
2 cd kube-prometheus
```

Create the namespace and CRDs:

```
1 kubectl create -f manifests/setup
```

To wait for the CRDs to be created, you can monitor the progress with the following command:

```
1 until kubectl get servicemonitors --all-namespaces ; do date; sleep 1; \
2 echo ""; done
```

When you see a `No resources found` message, it means that the custom resource definitions (CRDs) were created successfully. Now we can proceed to create the rest of the components.

```
1 kubectl create -f manifests/
```

21.4 Accessing Prometheus web UI

Next, we will create a port-forward to access the Prometheus UI:

¹²¹<https://prometheus-operator.dev/>

```
1 kubectl -n monitoring port-forward svc/prometheus-k8s 9090 > /dev/null \\\n2 2>&1 &
```

You can access the Prometheus UI from your machine at <http://localhost:9090>¹²². If you are using a remote machine, create an SSH tunnel to the remote machine and access the Prometheus UI from your local machine at <http://localhost:9090>¹²³.

```
1 export IP=<IP>\n2 ssh -NfL 9090:localhost:9090 root@$IP
```

Replace <IP> with the IP address of your remote machine.

21.5 Metrics available in Prometheus

When you install Prometheus, you will have access to a Prometheus instance, a Grafana instance, and a set of Prometheus exporters. The Prometheus Operator also creates kube-state-metrics, which is an add-on agent that listens to the Kubernetes API server and generates metrics about the state of Kubernetes objects, such as Nodes, Pods, Deployments, and more.

We can see a list of available metrics thanks to kube-state-metrics in its official repository at <https://github.com/kubernetes/kube-state-metrics>. For example, metrics for Pods are available [here](#)¹²⁴. The following are some examples of available metrics:

- `kube_pod_overhead_memory_bytes`: The Pod overhead in regards to memory associated with running a Pod
- `kube_node_status_capacity`: The total amount of resources available for a node

In addition to Prometheus metrics, Prometheus Operator also uses other sources of metrics, such as the [Kubernetes API](#)¹²⁵, blackbox exporters, and node exporters. You can find the list of available metrics in the [Prometheus documentation](#)¹²⁶.

¹²²<http://localhost:9090/>

¹²³<http://localhost:9090/>

¹²⁴<https://github.com/kubernetes/kube-state-metrics/blob/main/docs/pod-metrics.md>

¹²⁵<https://kubernetes.io/docs/concepts/overview/kubernetes-api/>

¹²⁶<https://prometheus.io/docs/prometheus/latest/querying/basics/>

To view a complete list of metrics available in your Prometheus instance, go to the Prometheus UI and enter the following search query in the search bar: `{__name__=~".+"}`. This will display all the available metrics in your Prometheus instance.

- The syntax `{__name__=~".+"}` is a PromQL expression that allows you to filter metrics by name.
- The `=~` operator is a regular expression match operator and it means that the value of the `__name__` label matches the regular expression `".+"`.
- The `".+"` regular expression matches any string with at least one character.

21.6 Using Grafana to visualize Prometheus metrics

The Prometheus Operator also creates a Grafana instance for visualizing metrics. To access Grafana, create a port-forward to the Grafana service:

```
1 kubectl -n monitoring port-forward svc/grafana 3000 > /dev/null 2>&1 &
```

To access Grafana on a remote machine, create an SSH tunnel to it from your local machine. Once the tunnel is established, you can access Grafana at <http://localhost:3000>¹²⁷.

```
1 export IP=<IP>
2 ssh -NfL 3000:localhost:3000 root@$IP
```

The default username and password for Grafana are `admin` and `admin`.

Grafana supports multiple data sources, including Prometheus. In this installation, Prometheus is already configured as a data source. To view the list of data sources, click on the gear icon on the left panel (Administration), and then click on Data Sources. You should see Prometheus listed as a data source.

¹²⁷<http://localhost:3000>

You can customize your dashboards by adding new ones or importing dashboards from [Grafana.com](#)¹²⁸.

To add a new dashboard, click on the plus icon on the left panel and then click on “Add visualization”. Choose Prometheus as the data source and type a PromQL expression in the query field. For example, to visualize the memory usage of our containers, type `container_memory_usage_bytes`. Click on the “Run query” button to view the results.

To add a dashboard from [Grafana.com](#)¹²⁹, follow these steps:

1. Click on the plus icon on the left panel.
2. Click on Import.
3. In the “Import via grafana.com” section, type 1860 in the “Grafana Dashboard” field and click on Load.
4. Choose Prometheus as the data source.
5. Click on Import to import the dashboard.

By following these instructions, you will load the [Kubernetes cluster monitoring dashboard](#)¹³⁰ from [Grafana.com](#)¹³¹.

Promtail functions similarly to Prometheus in that it discovers targets by monitoring the Kubernetes API server. It performs label queries against the Kubernetes API at regular intervals to find Pods that match the label selectors. Once it has discovered the targets, it attaches to them and sends the contents of the logs to Loki.

There are [multiple ways](#)¹³³ to install this tool, the easiest one is to use Helm.

```
1 helm repo add grafana https://grafana.github.io/helm-charts
2 helm repo update
3 helm install promtail grafana/promtail --namespace monitoring
```

You can verify that it's working by running these commands:

```
1 kubectl --namespace monitoring port-forward daemonset/promtail 3101 > />
2 dev/null 2>&1 &
3 curl http://127.0.0.1:3101/metrics
```

21.8 Loki logging stack



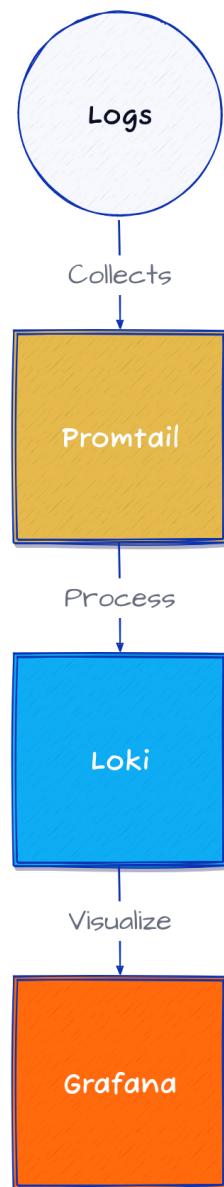
Loki is a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus. It is designed to be very cost-effective and easy to operate. It does not index the contents of the logs, but rather a set of labels for each log stream. Grafana is the developer of Loki and describes it as "Prometheus for logs".

Loki works in conjunction with Promtail, which gathers logs from Kubernetes. Together, the logging stack comprises the following components:

- Promtail: Gathers logs from Kubernetes and forwards them to Loki.
- Loki: Stores logs and enables querying of them.

¹³³<https://grafana.com/docs/loki/latest/clients/promtail/>

- **Grafana:** Visualizes logs.



A 3-components Loki-based logging stack

The integration process is relatively straightforward. First, we need to install the Loki data source plugin in Grafana. Then, we must configure it to use Loki as a data source.

To begin, we'll install the Loki data source plugin using Helm with a custom configuration. Follow these steps:

Create a working directory and navigate to it.

```
1 cd $HOME/monitoring  
2 mkdir loki && cd loki
```

Add the Loki Helm repository:

```
1 helm repo add grafana https://grafana.github.io/helm-charts  
2 helm repo update
```

Create a `values.yaml` file with the following content:

```
1 cat <<EOF > values.yaml  
2 loki:  
3     auth_enabled: false  
4     commonConfig:  
5         replication_factor: 1  
6     storage:  
7         type: 'filesystem'  
8     singleBinary:  
9         replicas: 1  
10 EOF
```

The above configuration installs Loki with a single replica and uses the filesystem as the storage backend. Authentication is disabled for simplicity. More information about configuration options can be found in the [official documentation](#)¹³⁴.

Install Loki with Helm in the namespace `monitoring`:

¹³⁴<https://grafana.com/docs/loki/latest/installation/helm/>

```
1 helm install --values values.yaml loki grafana/loki --namespace monitor\\
2 ing
```

If you want to install and run Loki with more than one replica, you should configure a distributed storage backend like S3. Here is an example:

```
1 cat <<EOF > values.yaml
2 loki:
3     storage:
4         bucketNames:
5             chunks: chunks
6             ruler: ruler
7             admin: admin
8         type: s3
9         s3:
10            endpoint: <endpoint>
11            region: <AWS region>
12            secretAccessKey: <AWS secret access key>
13            accessKeyId: <AWS access key ID>
14            s3ForcePathStyle: false
15            insecure: false
16 EOF
```

To add Loki as a data source in Grafana, follow these steps:

1. Go to Grafana and click on the gear icon on the left panel (Administration).
2. Click on “Data Sources”.
3. Click on the “Add data source” button.
4. Choose Loki as the data source.
5. In the HTTP section, type `http://loki.monitoring.svc:3100` in the URL field. (`loki` is the name of the Loki service and `monitoring` is the namespace where Loki is installed.)
6. Click on the “Save & Test” button to save the data source.

To visualize logs, we can create a dashboard. First, click on the plus icon in the left panel, then click on Import. In the Import via grafana.com section, type 15141

in the Grafana.com Dashboard field and click on Load. Choose Loki as the data source, and click on Import to load the [Loki Kubernetes Logs dashboard¹³⁵](#).

```
1 sum(rate({app="my-app", level="warn"}[1m])) / sum(rate({app="my-app", 1\\
2 evel="error"}[1m]))
```

This query calculates the ratio between the rate of logs with the label `level` equal to `warn` and the rate of logs with the label `level` equal to `error`.

The other operators you can use besides the equals (=) operator are:

- `!=`: not equals
- `=~`: regex matches
- `!~`: regex does not match

Comparison operators:

- `==`: equals
- `!=`: not equals
- `>`: greater than
- `>=`: greater than or equal
- `<`: less than
- `<=`: less than or equal

Logical operators:

- `and`
- `or`
- `unless`

Arithmetic operators:

- `+`: addition
- `-`: subtraction
- `*`: multiplication
- `/`: division
- `%`: modulo
- `^`: power

Examples:

```
1 {app!="spring-petclinic"}
```

This query filters logs from all Pods with the `app` label that is not equal to `spring-petclinic`.

```
sum(count_over_time({app="my-app"}[1m])) * 100
```

This query returns the values multiplied by 100 of the sum of the number of logs with the label `app` equal to `my-app` over the last minute.

You can also use regex in the values of the labels:

- `{filename=~"/var/log/auth.log|/var/log/syslog"}:` This query filters logs from the files `/var/log/auth.log` and `/var/log/syslog`.
- `{app=~"spring-petclinic|my-app"}:` This query filters logs from all Pods that have the label `app` equal to either `spring-petclinic` or `my-app`.

To have better control over filtering, you can use the various filters available in LogQL:

- `|:` This filter selects only the logs that match the given label filter.
- `!=:` This filter selects only the logs that do not match the given label filter.
- `|~:` This filter selects only the logs that match the given label filter using regex.
- `!~:` This filter selects only the logs that do not match the given label filter using regex.

Examples:

```
1 {app="spring-petclinic"} | "error"
```

This query filters logs from all Pods that have the `app` label equal to `spring-petclinic`, and selects only those that contain the string `error`.

```
1 {app="spring-petclinic"} |~ "error|exception"
```

This query filters logs from all Pods that have the label `app` equal to `spring-petclinic`, and selects only those that contain the strings `error` or `exception`

```
1 {app="spring-petclinic"} |~ "error|exception" != "404"
```

This query filters logs from all Pods that have the app label equal to `spring-petclinic`. It selects only those containing the strings `error` or `exception` but not `404`.

To extract meaningful metrics from your logs, you can use scalar vectors such as:

- `count_over_time`: Shows the total count of log lines for the time range.
- `rate`: Similar to `count_over_time`, but converted to the number of entries per second.
- `bytes_over_time`: Number of bytes in each log stream in the range.
- `bytes_rate`: Similar to `bytes_over_time`, but converted to the number of bytes per second.

Examples:

```
1 count_over_time({app="spring-petclinic"}[60m])
```

This query counts the number of logs from all Pods with the label `app` equal to `spring-petclinic` over the last hour.

```
1 rate({app="spring-petclinic"}[60m])
```

This query calculates the rate of logs from all Pods with the label `app` equal to `spring-petclinic` over the last hour.

```
1 bytes_over_time({app="spring-petclinic"}[60m])
```

This query calculates the number of bytes of logs from all Pods with the label `app` equal to `spring-petclinic` over the last hour.

```
1 bytes_rate({app="spring-petclinic"}[60m])
```

This query calculates the rate of bytes of logs from all Pods with the label `app` equal to `spring-petclinic` over the last hour.

We can also use the following functions to aggregate the logs:

- `sum`: Calculates the total of all vectors in the range at a given time
- `min`: Shows the minimum value from all vectors in the range at a given time
- `max`: Shows the maximum value from all vectors in the range at a given time
- `avg`: Calculates the average of the values from all vectors in the range at a given time
- `stddev`: Calculates the standard deviation of the values from all vectors in the range at a given time
- `stdvar`: Calculates the standard variance of the values from all vectors in the range at a given time
- `count`: Counts the number of elements in all vectors in the range at a given time
- `bottomk`: Selects the lowest k values in all the vectors in the range at a given time
- `topk`: Selects the highest k values in all the vectors in the range at a given time

Examples:

```
1 sum(count_over_time({app="spring-petclinic"}[60m]))
```

This query counts the number of logs from all Pods with the label `app` equal to `spring-petclinic` over the last hour and sums the result.

```
1 avg(count_over_time({app="spring-petclinic"}[60m]))
```

This query counts the number of logs from all Pods with the label `app` equal to `spring-petclinic` over the last hour and calculates the average of the result.

..etc

To group a series of vectors by a label, you can use the `by` clause:

```
1 sum(count_over_time({app="spring-petclinic"}[60m])) by (filename)
```

This query counts the number of logs from all Pods with the label app equal to `spring-petclinic` over the last hour and sums the result by the label `filename`.

21.10 Using Jaeger and OpenTelemetry for distributed tracing

Jaeger and OpenTelemetry are complementary tools that allow you to trace requests across multiple services.



OpenTelemetry is an observability framework for cloud-native software. It is a merger of OpenCensus and OpenTracing and it provides APIs, libraries, agents, and collector services to capture distributed traces and metrics from your applications. OpenTelemetry provides a vendor-neutral way to instrument applications, collect telemetry data, and send it to different monitoring systems. It supports capturing metrics, logs, and distributed traces.



Jaeger is a distributed tracing system inspired by Dapper and OpenZipkin. It is used for monitoring and troubleshooting microservices-based distributed systems and is a CNCF project. Jaeger is one of the popular backends that can receive and store distributed traces generated by applications instrumented with OpenTelemetry.

OpenTelemetry is used to instrument application code in order to capture distributed traces and other telemetry data. These captured traces are then exported to Jaeger, which processes and stores them for later analysis.

The OpenTelemetry Collector is a component that can receive traces from various sources, perform aggregation, filtering, and other processing, and then export the

traces to Jaeger or another backend system. The collector acts as a central point for telemetry data collection and distribution.

In summary, OpenTelemetry provides the means to instrument applications and capture distributed traces, while Jaeger acts as a backend to receive, store, and analyze those traces.

To begin, create a working directory and navigate to it:

```
1 mkdir -p $HOME/monitoring/jaeger && cd $HOME/monitoring/jaeger
```

Before proceeding, make sure you install Cert Manager in your cluster:

```
1 kubectl apply -f https://github.com/cert-manager/cert-manager/releases/\n2 download/v1.6.3/cert-manager.yaml
```

According to the official documentation, starting from version 1.31, the Jaeger Operator uses webhooks to validate Jaeger custom resources (CRs). This requires an installed version of cert-manager.

Create the new namespace “observability”:

```
1 kubectl create namespace observability
```

Install the Jaeger Operator:

```
1 kubectl apply -f https://github.com/jaegertracing/jaeger-operator/releases/\n2 download/v1.45.0/jaeger-operator.yaml -n observability
```

Deploy the Jaeger instance:

```
1 kubectl apply -f - <<EOF
2 ---
3 apiVersion: jaegertracing.io/v1
4 kind: Jaeger
5 metadata:
6   name: simplest
7 EOF
```

This is the simplest Jaeger instance that you can deploy. It will create a Jaeger instance with the default configuration. You can find more information about the configuration options in the [official documentation](#)¹³⁸.

There are many configuration options available to customize your Jaeger instance. For instance, you can enable authentication, configure the storage backend (such as Elasticsearch or Cassandra), and more. If you want to deploy a more advanced Jaeger instance with Elasticsearch as the storage backend, you can use Helm:

```
1 helm repo add jaegertracing https://jaegertracing.github.io/helm-charts
2 helm repo update
3 helm install jaeger jaegertracing/jaeger \
4   --set provisionDataStore.cassandra=false \
5   --set provisionDataStore.elasticsearch=true \
6   --set storage.type=elasticsearch
```

To use Cassandra as the storage backend, you can use the following command:

```
1 helm install jaeger jaegertracing/jaeger \
2   --set provisionDataStore.elasticsearch=false \
3   --set provisionDataStore.cassandra=true \
4   --set storage.type=cassandra
```

Cassandra and Elasticsearch are deployed as part of the Jaeger instance. Alternatively, you can use an existing Cassandra or Elasticsearch instance by providing connection details in the configuration. For more information on datastore configuration options, please refer to the [official documentation](#)¹³⁹.

¹³⁸<https://www.jaegertracing.io/docs/1.25/operator/>

¹³⁹<https://github.com/jaegertracing/helm-charts/tree/main/charts/jaeger>

If you chose to use the simplest Jaeger instance, you can forward the Jaeger UI port to your local machine.

```
1 kubectl port-forward svc/simplest-query 16686:16686 > /dev/null &2>1
```

Then deploy the OpenTelemetry Operator:

```
1 kubectl apply -f https://github.com/open-telemetry/opentelemetry-operator\\
2 or/releases/download/v0.76.1/opentelemetry-operator.yaml
```

Create the OpenTelemetry Collector instance:

```
1 kubectl apply -f - <<EOF
2 ---
3 apiVersion: opentelemetry.io/v1alpha1
4 kind: OpenTelemetryCollector
5 metadata:
6   name: otel
7 spec:
8   config: |
9     receivers:
10       otlp:
11         protocols:
12           grpc:
13           http:
14   processors:
15     memory_limiter:
16       check_interval: 5s
17       limit_percentage: 90
18       spike_limit_percentage: 95
19   batch:
20     send_batch_size: 8192
21     timeout: 5s
22   exporters:
23     logging:
```

```
24     jaeger:  
25         endpoint: "simplest-collector:14250"  
26         tls:  
27             insecure: true  
28     service:  
29         pipelines:  
30             traces:  
31                 receivers: [otlp]  
32                 processors: []  
33                 exporters: [jaeger]  
34 EOF
```



The OTel Collector receives traces from different sources including OpenTelemetry SDKs, OpenTelemetry Collector, and other sources. It then performs aggregation, filtering, and other processing on the traces and exports them to Jaeger.

The YAML above deploys an OTel Collector while defining the receivers, processors, exporters, and pipelines.

- `receivers` specifies the telemetry data receivers. In our example, it includes the [OTLP receiver](#)¹⁴⁰, which can receive telemetry data via gRPC and HTTP protocols.
- `processors` specifies the data processors to apply to the collected telemetry data. In our example, no processors are defined.
- `exporters` specifies the destinations where the processed telemetry data should be exported. In our example, it includes a logging exporter and a Jaeger exporter. The logging exporter sends telemetry data to the logging system, while the Jaeger exporter sends data to a Jaeger collector (the Jaeger instance that we deployed earlier).
- `service` defines the service configuration for the OpenTelemetry Collector. The pipelines section defines the processing pipeline for the collected telemetry data. In our collector, there is a single pipeline named `traces` that receives

¹⁴⁰<https://github.com/open-telemetry/opentelemetry-collector/blob/main/receiver/otlpreceiver/README.md>

data from the OTLP receiver, applies no processors, and exports the data to the Jaeger exporter.

One of the most interesting features of OpenTelemetry is its ability to auto-instrument your application. Let's see how it works.

```
1 kubectl apply -f - <<EOF
2 ---
3 apiVersion: opentelemetry.io/v1alpha1
4 kind: Instrumentation
5 metadata:
6   name: my-instrumentation
7 spec:
8   exporter:
9     endpoint: http://otel-collector:4317
10  propagators:
11    - tracecontext
12    - baggage
13    - b3
14  sampler:
15    type: parentbased_traceidratio
16    argument: "0.5"
17  java:
18    image: ghcr.io/open-telemetry/opentelemetry-operator/autoinstrument\
19  ation-java:1.26.0
20 EOF
```

The OpenTelemetry Instrumentation resource created above will auto-instrument the Spring application that we are going to deploy. It configures the exporter endpoint (`http://otel-collector:4317`), propagators (tracecontext, baggage, and b3), and sampler. It also specifies the auto-instrumentation image for Java applications (`ghcr.io/open-telemetry/opentelemetry-operator/autoinstrumentation-java:1.26.0`).

Since our OpenTelemetry Collector is deployed in the same cluster, we can use its service name as the exporter endpoint. If you are using an external OpenTelemetry Collector, you can use its external IP address or domain name.

In OpenTelemetry, we use the concept of Signals to refer to the different types of telemetry data. There are four types of signals: `trace`, `metrics`, `logs`, and `baggages`.

Traces give an overview of what is happening in your application. They are a sequence of events that are linked together to form a trace. Each event in the trace is called a span. In other words, the span is the building block of a trace. A span can be used to represent a request, a function call, a database query, or any other event in your application.

A trace can have multiple spans. For example, a request to your application can be represented as a trace with multiple spans. Each span can represent a call to a function, a query on the database, and then an HTTP request. In order to have spans correlated and assembled together to form a trace, we need Context Propagation.

Context Propagation is a key concept in Distributed Tracing, and it involves the process of passing the context of a request from one service to another.

Context represents the information needed to correlate Spans and associate them with a Trace. For example, when Service A calls Service B, the Span from Service A becomes the parent Span for the next Span created in Service B. Propagation is the mechanism that transfers the Context between services, allowing for the assembly of a Distributed Trace. It serializes and deserializes the Span Context, enabling the transfer of relevant Trace information from one service to another. This mechanism is known as `tracecontext`.

The `baggages` propagator is used to propagate baggage items. Baggage is a mechanism to pass data between spans. It is similar to the concept of HTTP headers, but it is used to pass data between spans instead of passing data between HTTP requests and responses.

Finally, the `b3` propagator is used to propagate the B3 context. B3 is a distributed tracing protocol that defines the format of the trace context. It is supported by many tracing systems including Jaeger, Zipkin, and AWS X-Ray.

The `parentbased_traceidratio` sampler is used to sample traces based on the parent trace ID. It is a probabilistic sampler that samples a trace if the parent trace ID is less than the specified ratio. In our example, we set the ratio to 0.5, which means that 50% of the traces will be sampled. This is useful for reducing the

amount of data that is sent to the backend. You don't want to send all the traces to the backend because it will be too much data to process. For example, if you have a service that receives 1000 requests per second, while all of them are successful, you don't want to generate 1000 traces per second. Instead, you can sample 50% of the traces and send them to the backend.

Finally, we configured the image to use for auto-instrumentation. In our example, we are using the Java auto-instrumentation image. There are also auto-instrumentation images for other languages including Python, Node.js, and .NET.

We can now deploy our application:

```
1 kubectl apply -f - <<EOF
2 ---
3 apiVersion: apps/v1
4 kind: Deployment
5 metadata:
6   name: spring-petclinic
7 spec:
8   selector:
9     matchLabels:
10    app: spring-petclinic
11   replicas: 1
12   template:
13     metadata:
14       labels:
15         app: spring-petclinic
16       annotations:
17         sidecar.opentelemetry.io/inject: "true"
18         instrumentation.opentelemetry.io/inject-java: "true"
19   spec:
20     containers:
21       - name: app
22         image: ghcr.io/pavolloffay/spring-petclinic:latest
23 EOF
```

The YAML code above deploys the Spring PetClinic application¹⁴¹.

To accomplish this, the following annotations are used to inject the OpenTelemetry Collector sidecar and the Java auto-instrumentation agent into the Pod running the application:

- `sidecar.opentelemetry.io/inject`
- `instrumentation.opentelemetry.io/inject-java`

Now you can describe the created Pod:

```
1 export pod=$(kubectl get pod -l app=spring-petclinic -o jsonpath='{.items[0].metadata.name}')
2
3 kubectl describe pod $pod
```

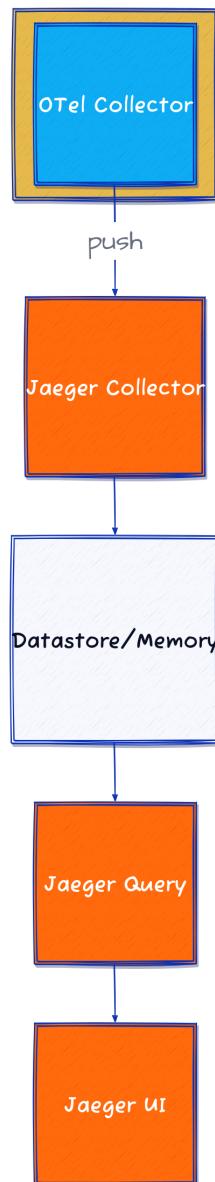
You should be able to see two containers in the Pod:

- The app container is the main container that runs the Spring PetClinic application.
- The sidecar container.

The application has also new environment variables used for the instrumentation:

```
1 kubectl exec $pod -- printenv | grep OTEL
```

¹⁴¹<https://github.com/spring-projects/spring-petclinic>



OpenTelemetry Auto Instrumentation and Jaeger

Now we can access the Jaeger UI to see the traces. Execute the following command to forward port 16686 of the `simplest-query` service to your local machine if it's not already done:

```
1 kubectl port-forward svc/simplest-query 16686:16686 > /dev/null &2>1
```

Then if you want to access the Jaeger UI, you can open the following URL in your browser: <http://localhost:16686>¹⁴² after ssh-tunneling to the development machine.

```
1 export IP=<IP>
2 ssh -NfL 16686:localhost:16686 root@$IP
```

You should be able to see two services in the Jaeger UI: `spring-petclinic` and `jaeger-query`. The `spring-petclinic` service is the application that we deployed, and `jaeger-query` is the Jaeger Query service itself.

22 GitOps: Cloud Native Continuous Delivery

22.1 GitOps: introduction and definitions

With the rise of “everything-as-code,” Git has become the de facto standard for version control and source code management. GitOps is a new approach to infrastructure management and application delivery that uses Git as a single source of truth for environment configurations and application deployments.

In a GitOps environment, every change is validated via code review processes and applied consistently to the environment using CI/CD (Continuous Integration/- Continuous Deployment) tools such as Jenkins, Travis CI, or CircleCI.

By using Git as a single source of truth, development and operations teams can follow an approval and validation process before changes are applied, which improves the security and stability of the environment. Git, being the central component of the process, also provides a complete audit trail of all changes, whether it is a configuration change, a new release, a rollback, a security patch, or a new infrastructure component.

In fact, GitOps can be used to manage a variety of environments, including server infrastructures, container platforms, and microservices environments.

22.2 GitOps: benefits and drawbacks

Teams that use GitOps define their desired state in Git and then use automation to ensure that the actual state of the system always reflects the desired state. All changes are proposed via pull requests and reviewed, allowing teams to collaborate and provide feedback before changes are applied to the environment.

Deployments can be automated and triggered by changes in Git, keeping the environment up to date with the desired state. This also allows for easy rollbacks by reverting changes in Git. GitOps keeps track of all changes, providing a complete audit trail that can be useful for compliance and security.

GitOps also simplifies the management of multiple environments by using the same tools and processes.

While GitOps is relatively simple, it has some drawbacks. Teams need to use Git as a single source of truth, which can be challenging. GitOps also requires a high level of automation, which can be difficult for some teams to achieve.

22.3 GitOps: tools

GitOps is a recent approach to infrastructure management and application delivery, and its ecosystem is still evolving. Nonetheless, there are already several tools available to implement GitOps in your environment.

The most popular tools for implementing GitOps are:

- [Flux¹⁴³](#): Flux is a tool that automates the deployment of applications and infrastructure using Git as a single source of truth. It can be used to manage Kubernetes clusters, server infrastructures, and microservices environments.
- [Argo CD¹⁴⁴](#): A tool that automates the deployment of applications and infrastructure using Git as a single source of truth. It can be used to manage Kubernetes clusters, server infrastructures, and microservices environments. We are going to study this tool later in this guide.
- [Jenkins X¹⁴⁵](#): Jenkins X is a tool that automates the deployment of applications and infrastructure using Git as a single source of truth. It can be used to manage Kubernetes clusters, server infrastructures, and microservices environments.
- [GitLab¹⁴⁶](#): GitLab is a complete DevOps platform, delivered as a single application. It provides a complete DevOps toolchain out-of-the-box and can be used to implement GitOps in your environment.

¹⁴³<https://fluxcd.io/>

¹⁴⁴<https://argoproj.github.io/argo-cd/>

¹⁴⁵<https://jenkins-x.io/>

¹⁴⁶<https://about.gitlab.com/>

- **Werf¹⁴⁷**: A tool that automates the deployment of applications and infrastructure using Git as a single source of truth. It can be used to manage Kubernetes clusters, server infrastructures, and microservices environments.

¹⁴⁷<https://werf.io/>

23 GitOps: Example of a GitOps workflow using Argo CD

23.1 Argo CD: introduction

Argo CD is an open-source continuous deployment (CD) tool for Kubernetes environments, based on GitOps. It enables developers to deploy their applications to a Kubernetes cluster simply by pushing to a specific Git branch. This triggers a continuous deployment pipeline that updates Kubernetes resources according to changes in the source code.

Argo CD uses concepts such as applications, environments, and syncs to enable the management and deployment of applications in a Kubernetes cluster. It offers a user interface and a REST API for managing and monitoring applications deployed on a Kubernetes cluster. Additionally, it provides a CLI tool that can be used to manage and monitor applications deployed on a Kubernetes cluster, and can be integrated with CI/CD tools.

23.2 Argo CD: installation and configuration

Argo CD can be installed on a Kubernetes cluster using the following command:

```
1 kubectl create namespace argocd
2 kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/a\
3 rgo-cd/stable/manifests/install.yaml
```

To access the Argo CD UI, you need to create a port-forward to the Argo CD server (or expose the service using a LoadBalancer or an Ingress):

```
1 kubectl port-forward svc/argocd-server -n argocd 8080:443 > /dev/null 2>&1 &
```

We can now create an SSH tunnel to access the Argo CD UI from our local machine:

```
1 export ip=<IP>
2 ssh -NfL 8080:localhost:8080 root@$ip
```

The default username is `admin` and the password can be retrieved using the following command:

```
1 kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{\.
2 .data.password}" | base64 -d; echo
```

To install the Argo CD CLI, use the following commands:

```
1 curl -sSL -o argocd-linux-amd64 https://github.com/argoproj/argo-cd/releases/latest/download/argocd-linux-amd64
2 sudo install -m 555 argocd-linux-amd64 /usr/local/bin/argocd
4 rm argocd-linux-amd64
```

Connect the CLI to the Argo CD server:

```
1 export password=$(kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d; echo)
3 argocd login localhost:8080 --username admin --password $password --secure
```

Now that we have installed the Argo CD CLI and connected it to the Argo CD server, we can add our cluster to the Argo CD server. However, this step is only required when Argo CD is not running on the same cluster that we want to manage. In our case, we can skip this step.

```
1 export cluster=$(kubectl config get-contexts -o name)
2 argocd cluster add $cluster -y
```

We can double-check that our cluster has been added using the following command:

```
1 argocd cluster list
```

23.3 Argo CD: creating an application

We will use [this Github repository](#)¹⁴⁸ to deploy a simple Flask application on our Kubernetes cluster using Argo CD. You can clone this repository to get started. The repository contains several examples of applications that can be deployed using Argo CD, each stored in a separate folder.

Let's start by creating a namespace for our application:

```
1 kubectl create namespace flask-app
```

We can now deploy the application manifest using the following command:

```
1 argocd app create flask-app \
2   --repo https://github.com/eon01/argocd-examples/ \
3   --path flask-app \
4   --dest-server https://kubernetes.default.svc \
5   --dest-namespace flask-app \
6   --revision main \
7   --sync-policy automated
```

The above command will create an application named `flask-app` using the manifest stored in the `flask-app` folder of the Github repository. The application will be deployed in the `flask-app` namespace of the Kubernetes cluster. The branch used is `main`. We are using the `--sync-policy automated` flag to enable automatic synchronization of the application when a change is detected in the Git repository.

The folder `flask-app` contains a file `kubernetes.yaml` that defines the Deployment and Service resources required to deploy the application.

¹⁴⁸<https://github.com/eon01/argocd-examples/>

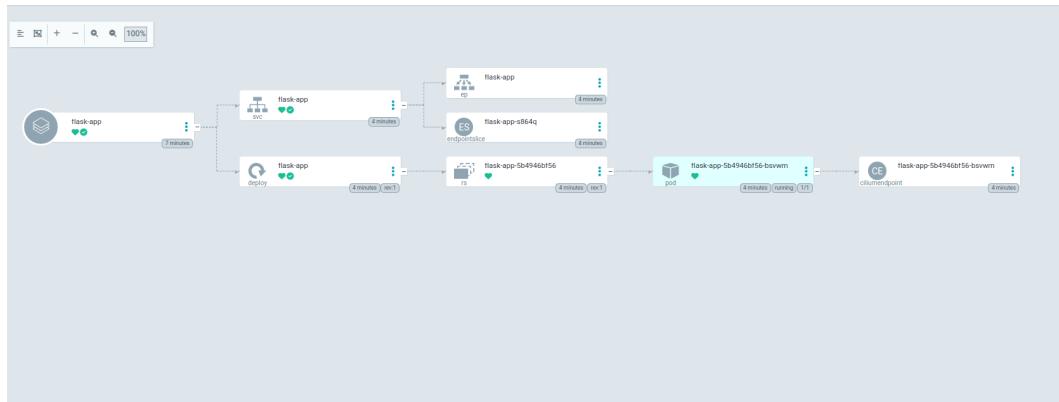
```
1  ---
2  apiVersion: apps/v1
3  kind: Deployment
4  metadata:
5    name: flask-app
6  spec:
7    replicas: 1
8    revisionHistoryLimit: 3
9    selector:
10      matchLabels:
11        app: flask-app
12    template:
13      metadata:
14        labels:
15          app: flask-app
16    spec:
17      containers:
18        - image: eon01/stateless-flask:v0
19        name: flask-app
20      ports:
21        - containerPort: 5000
22      resources:
23        limits:
24          cpu: 100m
25          memory: 128Mi
26        requests:
27          cpu: 100m
28          memory: 128Mi
29      readinessProbe:
30        httpGet:
31          path: /tasks
32          port: 5000
33        initialDelaySeconds: 5
34        periodSeconds: 10
35  ---
36  apiVersion: v1
```

```

37 kind: Service
38 metadata:
39   name: flask-app
40 spec:
41   ports:
42     - port: 5000
43       protocol: TCP
44       targetPort: 5000
45   selector:
46     app: flask-app
47 type: LoadBalancer

```

You can visit the Argo CD UI to see the new application using <https://localhost:8080/applications>. Here, you can view the status, health, and history of the application, as well as the application details tree.



You can also see that the application has been deployed in the `flask-app` namespace of the Kubernetes cluster.

```

1 export APPNAME=flask-app
2 kubectl -n $APPNAME get all

```

23.4 Argo CD: automatic synchronization and self-healing

Since we activated the automatic synchronization of the application, any changes made to the application manifest will trigger a synchronization of the application. For example, we can change the number of replicas from 1 to 2 in the kubernetes .yaml file and push the change to the Github repository. This will trigger a synchronization of the application and update the number of replicas to 2. By default, Argo CD polls the Git repository every 3 minutes to check for changes.

Remember to commit and push the changes to the Github repository.

```
1 git add .
2 git commit -m "update replicas"
3 git push
```

The concept of the desired state is central to Argo CD. The desired state is the state defined in the application's Git repository. Argo CD will continuously compare the desired state with the actual state and will automatically synchronize the application if there is a difference between the two states.

However, sometimes the automated sync policy also means deleting resources. To prevent accidents, Argo CD will not delete resources by default. However, if you know what you are doing, you can enable the deletion of resources using the following command:

```
1 argocd app set $APPNAME --auto-prune
```

Another safety mechanism that is enabled by default is protection against having empty resources. If you want to allow empty resources, you can disable this safety mechanism using the following command:

```
1 argocd app set $APPNAME --allow-empty
```

Another useful mechanism is the ability to return to the desired state when the current state has been modified. For example, if you manually change the number of replicas to 3, Argo CD will automatically revert the change to 2. You can enable this mechanism using the following command:

```
1 argocd app set $APPNAME --self-heal
```

To activate automated synchronization without having to wait for the next poll, you can use the following command:

```
1 argocd app sync $APPNAME
```

Alternatively, you can set up a webhook in your GitHub repository that is triggered when a change is detected. The webhook should use the following URL: <https://<argocd>/api/webhook>, where <argocd> is the public domain that points to your Argo CD installation. You can find more information about webhooks [here](#)¹⁴⁹.

23.5 Argo CD: rollback

You can see the revision history of the application by using the following command:

```
1 argocd app history $APPNAME
```

You can rollback to a previous revision using the following command:

```
1 argocd app rollback $APPNAME <revision-number>
```

However, a rollback cannot be performed against an application with automated sync enabled. You need to disable automated sync before performing a rollback.

```
1 argocd app set $APPNAME --sync-policy none
```

You can re-enable automated sync after the rollback:

¹⁴⁹<https://argo-cd.readthedocs.io/en/stable/operator-manual/webhook/>

```
1 argocd app set $APPNAME --sync-policy automated
```

23.6 Argo CD the declarative way

Previously, we created an application using the following command:

```
1 argocd app create flask-app \
2   --repo https://github.com/eon01/argocd-examples/ \
3   --path flask-app \
4   --dest-server https://kubernetes.default.svc \
5   --dest-namespace flask-app \
6   --revision main \
7   --sync-policy automated \
8   --self-heal \
9   --auto-prune \
10  --allow-empty
```

This is the imperative way of creating an application, but since we are in a world of “everything-as-code,” it is also possible to create an application using a declarative approach. This means that we can define the application in a manifest file and then apply the manifest file to create the application.

Let’s delete the previous application:

```
1 argocd app delete flask-app
```

And let’s recreate it using the declarative approach.

```
1  kubectl apply -f - <<EOF
2  ---
3  apiVersion: argoproj.io/v1alpha1
4  kind: Application
5  metadata:
6    name: flask-app
7    namespace: argocd
8  spec:
9    destination:
10      namespace: flask-app
11      server: https://kubernetes.default.svc
12    project: default
13    source:
14      path: flask-app
15      repoURL: https://github.com/eon01/argocd-examples/
16      targetRevision: main
17    syncPolicy:
18      automated:
19        prune: true
20        selfHeal: true
21        allowEmpty: true
22 EOF
```

23.7 Argo CD: configuration management

If you are continuously deploying applications, you will need to manage the configuration of your applications. There are several ways to manage the configuration of your applications. Some of them are native to Argo CD, like [Helm¹⁵⁰](#), [Jsonnet¹⁵¹](#), and [Kustomize¹⁵²](#), but you can also [write your own configuration management plugin¹⁵³](#) and use it with Argo CD.

¹⁵⁰<https://helm.sh/>

¹⁵¹<https://jsonnet.org/>

¹⁵²<https://kustomize.io/>

¹⁵³<https://argo-cd.readthedocs.io/en/stable/operator-manual/config-management-plugins/>



Helm uses charts to define the structure of the application. A chart is a collection of files that describe a related set of Kubernetes resources.

A Helm chart is composed of the following files:

- `Chart.yaml`: contains the metadata of the chart
- `values.yaml`: contains the default values for the chart
- `templates/`: contains the templates of the Kubernetes resources
- `_helpers.tpl`: contains the helper functions used in the templates of the Kubernetes resources

Helm uses Go template language to generate Kubernetes resource files based on the provided values. You can define variables in templates using the `{} .Values.variableName }}` syntax, where `variableName` corresponds to a key in the `values.yaml` file.

Conditionals and loops can be used in templates to generate different configurations based on certain conditions or to iterate over lists of values.

Helpers are reusable template functions that can simplify complex logic or calculations. You can define custom helper functions in the `_helpers.tpl` file located in the root directory of your chart. Helper functions can be used in your templates to perform actions like string manipulation, calculations, or conditional checks.

Let's use Helm with Argo CD in the next examples. We already have another subfolder in the same GitHub repository that contains a Helm chart. Let's update the application to use the Helm chart instead of the Kubernetes manifests.

```
1  kubectl apply -f - <<EOF
2  ---
3  apiVersion: argoproj.io/v1alpha1
4  kind: Application
5  metadata:
6    name: flask-app
7    namespace: argocd
8  spec:
9    destination:
10      namespace: flask-app
11      server: https://kubernetes.default.svc
12    project: default
13    source:
14      repoURL: https://github.com/eon01/argocd-examples
15      targetRevision: main
16      path: flask-app-helm
17      helm:
18        valueFiles:
19          - values.yaml
20    syncPolicy:
21      automated:
22        prune: true
23        selfHeal: true
24        allowEmpty: true
25 EOF
```

Our `Chart.yaml` defines the following metadata:

```
1 # Specify the version of the Helm chart API
2 apiVersion: v2
3 # Name of the Helm chart
4 name: flask-app-helm
5 # Description of the Helm chart
6 description: A Helm chart for Kubernetes to learn Argo CD
7 # Type of the chart (e.g., application, library)
8 type: application
9 # Version of the Helm chart
10 version: 0.1.0
11 # Version of the application being deployed by the Helm chart
12 appVersion: "1.0"
```

This is the deployment.yaml file that defines the Deployment of the application:

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: {{ template "flask-app.fullname" . }}
5   labels:
6     app: {{ template "flask-app.fullname" . }}
7     chart: {{ template "flask-app.chart" . }}
8     release: {{ .Release.Name }}
9     heritage: {{ .Release.Service }}
10  spec:
11    replicas: {{ .Values.replicaCount }}
12    revisionHistoryLimit: 3
13    selector:
14      matchLabels:
15        app: {{ template "flask-app.fullname" . }}
16        release: {{ .Release.Name }}
17    template:
18      metadata:
19        labels:
20          app: {{ template "flask-app.fullname" . }}
21          release: {{ .Release.Name }}
```

```

22   spec:
23     containers:
24       - image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
25         name: {{ .Chart.Name }}
26       ports:
27         - containerPort: 5000
28       resources:
29         limits:
30           cpu: 100m
31           memory: 128Mi
32         requests:
33           cpu: 100m
34           memory: 128Mi
35       readinessProbe:
36         httpGet:
37           path: /tasks
38           port: 5000
39         initialDelaySeconds: 5
40         periodSeconds: 10

```

This is the `service.yaml` file that defines the Service:

```

1 apiVersion: v1
2 kind: Service
3 metadata:
4   name: {{ template "flask-app.fullname" . }}
5   labels:
6     app: {{ template "flask-app.fullname" . }}
7     chart: {{ template "flask-app.chart" . }}
8     release: {{ .Release.Name }}
9     heritage: {{ .Release.Service }}
10  spec:
11    ports:
12      - port: 80
13        protocol: TCP
14        targetPort: 5000

```

```
15 selector:
16   matchLabels:
17     app: {{ template "flask-app.fullname" . }}
18     release: {{ .Release.Name }}
19   type: {{ .Values.service.type }}
```

This is the `_helpers.tpl` file that defines some reusable template logic to call them from within our templates.

```
1 {{/*
2 This file contains helper templates and functions that can be used across
3 multiple templates in the current chart.
4 */}}
5
6 {{/* Define a helper function to generate the full name of the application */}
7 define "flask-app.fullname" -
8 {{- printf "%s-%s" .Release.Name .Chart.Name -}}
9 {{- end -}}
10
11 {{/* Define a helper function to get the chart name */}
12 define "flask-app.chart" -
13 {{- printf "%s-%s" .Chart.Name .Chart.Version -}}
14 {{- end -}}
```

Finally, here is our `values.yaml` file, which defines the default values for our application:

```
1 replicaCount: 3
2
3 image:
4   repository: eon01/stateless-flask:v0
5   tag: latest
6
7 service:
8   type: LoadBalancer
```

This is a brief explanation of how the Helm templating language works:

- Variables like `{{ .Values.replicaCount }}` are replaced with the corresponding values defined in the `values.yaml` file. The `.Values` object allows you to access the values defined in the chart's `values.yaml` file and use them within your templates.
- Variables like `{{ .Release.Service }}` are replaced with the values from the `release` object created by Helm. The `.Release` object provides access to information about the current release, such as the release name, release namespace, and release timestamp. The `.Release.Service` refers to the `service` field of the `release` object which is set to `Helm` by default.
- Variables like `{{ template "flask-app.fullname" . }}` are replaced with the values returned by the helper functions defined in the `_helpers.tpl` file. Helper functions allow you to define reusable template logic and call them from within your templates. In this case, the `flask-app.fullname` helper function returns the full name of the application by combining the release name and the chart name.



The “Helm template language” is not exclusive to Helm itself; it is a combination of the Go template language, additional functions, and wrappers that provide access to specific objects within the templates. This means that many resources and guides available for Go templates can be valuable references when learning about Helm templating.

Currently, we have deployed the application using Helm by utilizing the following YAML:

```
1 apiVersion: argoproj.io/v1alpha1
2 kind: Application
3 metadata:
4   name: flask-app
5   namespace: argocd
6 spec:
7   destination:
8     namespace: flask-app
9     server: https://kubernetes.default.svc
10    project: default
11    source:
12      repoURL: https://github.com/eon01/argocd-examples
13      targetRevision: main
14      path: flask-app-helm
15      helm:
16        valueFiles:
17        - values.yaml
18    syncPolicy:
19      automated:
20        prune: true
21        selfHeal: true
22        allowEmpty: true
```

As you can see, the values file that Argo CD will use to deploy the application is the `values.yaml` file, which should be provided in the `helm` section of the `source` field.

Alternatively, we can define the values directly in the `values` field of the `source` section.

```
1 apiVersion: argoproj.io/v1alpha1
2 kind: Application
3 metadata:
4   name: flask-app
5   namespace: argocd
6 spec:
7   destination:
8     namespace: flask-app
9     server: https://kubernetes.default.svc
10    project: default
11    source:
12      repoURL: https://github.com/eon01/argocd-examples
13      targetRevision: main
14      path: flask-app-helm
15      helm:
16        values:
17          replicaCount: 3
18          image:
19            repository: eon01/stateless-flask:v0
20            tag: latest
21          service:
22            type: LoadBalancer
23    syncPolicy:
24      automated:
25        prune: true
26        selfHeal: true
27        allowEmpty: true
```

23.8 Argo CD: managing different environments

As seen previously, you can manage different clusters using the same instance of Argo CD. This is very useful when you want to manage different environments of

the same application. For example, you can have a development environment, a staging environment, and a production environment.

To add a cluster, use the following command:

```
1 export cluster=<cluster-name>
2 argocd cluster add $cluster -y
```

For example, we have a production cluster called `production` and a staging cluster called `staging`. We can add them to Argo CD using the following commands:

```
1 export cluster=production
2 argocd cluster add $cluster -y
3
4 export cluster=staging
5 argocd cluster add $cluster -y
```

In the Argo CD UI, you can see the different clusters in the `Settings` tab. Otherwise, list your cluster using:

```
1 argocd cluster list
```

Export the name of the cluster you want to use, depending on the environment to which you want to deploy the application.

```
1 export CLUSTER=<cluster-name>
```

Deploy the application to that cluster using the following command:

```
1  kubectl apply -f - <<EOF
2  ---
3  apiVersion: argoproj.io/v1alpha1
4  kind: Application
5  metadata:
6    name: flask-app
7    namespace: argocd
8  spec:
9    destination:
10      namespace: flask-app
11      # $CLUSTER is the name of the cluster you want to deploy the applic\
12      ation to
13      server: $CLUSTER
14    project: default
15    source:
16      repoURL: https://github.com/eon01/argocd-examples
17      targetRevision: main
18      path: flask-app-helm
19      helm:
20        valueFiles:
21          - values.yaml
22    syncPolicy:
23      automated:
24        prune: true
25        selfHeal: true
26        allowEmpty: true
27 EOF
```

Additionally, you can create different values for each environment. For example, you can create a `values-production.yaml` file and a `values-staging.yaml` file. Then, you can use the following command to deploy the application to the production environment:

```
1  kubectl apply -f - <<EOF
2  ---
3  apiVersion: argoproj.io/v1alpha1
4  kind: Application
5  metadata:
6    name: flask-app
7    namespace: argocd
8  spec:
9    destination:
10      namespace: flask-app
11      # $CLUSTER is the name of the cluster you want to deploy the applic\
12      action to
13      server: $CLUSTER
14    project: default
15    source:
16      repoURL: https://github.com/eon01/argocd-examples
17      targetRevision: main
18      path: flask-app-helm
19      helm:
20        valueFiles:
21          - values-production.yaml # or values-staging.yaml ...etc
22    syncPolicy:
23      automated:
24        prune: true
25        selfHeal: true
26        allowEmpty: true
27 EOF
```

23.9 Argo CD: deployment hooks

It is possible to define hooks that will be executed before or after the deployment of the application. For example, you can execute a script that sends a notification to a Slack channel or an email to your team.

To define a hook, you should define a Job. For instance, you can define a Job that sends an email.

```
1  kubectl create -f - <<EOF
2  apiVersion: batch/v1
3  kind: Job
4  metadata:
5    generateName: email-
6    namespace: argocd
7    annotations:
8      argocd.argoproj.io/hook: PostSync
9      argocd.argoproj.io/hook-delete-policy: HookSucceeded
10 spec:
11   template:
12     spec:
13       containers:
14         - name: email
15           image: bytemark/smtp
16           command: ["/bin/sh"]
17           args: ["-c", "echo 'Hello World!' | mail -s 'Hello World!' <ema\
18             il>"]
19         env:
20           - name: RELAY_HOST
21             value: <smtp-host>
22           - name: RELAY_PORT
23             value: <smtp-port>
24           - name: RELAY_USERNAME
25             value: <smtp-username>
26           - name: RELAY_PASSWORD
27             valueFrom:
28               secretKeyRef:
29                 name: smtp-password
30                 key: password
31             restartPolicy: Never
32             backoffLimit: 4
33 ---
```

```
34 apiVersion: v1
35 kind: Secret
36 metadata:
37   name: smtp-password
38   namespace: argocd
39 type: Opaque
40 stringData:
41   password: <smtp-password>
42 EOF
```

Then, you can redeploy the application to trigger the hook:

```
1 kubectl apply -f - <<EOF
2 ---
3 apiVersion: argoproj.io/v1alpha1
4 kind: Application
5 metadata:
6   name: flask-app
7   namespace: argocd
8 spec:
9   destination:
10     namespace: flask-app
11     server: https://kubernetes.default.svc
12   project: default
13   source:
14     repoURL: https://github.com/eon01/argocd-examples
15     targetRevision: main
16     path: flask-app-helm
17     helm:
18       valueFiles:
19         - values.yaml
20   syncPolicy:
21     automated:
22       prune: true
23       selfHeal: true
```

```
24      allowEmpty: true  
25 EOF
```

You should be able to see the Job after executing the previous command:

```
1 kubectl get jobs -n argocd
```

As well as the created container logs:

```
1 kubectl logs -n argocd email-<pod-id>-<container-id>
```

Same as the above, we can create a Job that will send a notification to a Slack channel:

```
1 apiVersion: batch/v1  
2 kind: Job  
3 metadata:  
4   generateName: slack-  
5   namespace: argocd  
6   annotations:  
7     argocd.argoproj.io/hook: PostSync  
8     argocd.argoproj.io/hook-delete-policy: HookSucceeded  
9 spec:  
10   template:  
11     spec:  
12       containers:  
13         - name: slack  
14           image: curlimages/curl  
15           command:  
16             - "curl"  
17             - "-X"  
18             - "POST"  
19             - "--data-urlencode"  
20             - "payload={"channel\"": "#<channel-name>\", \"username\": \"  
21 \"hello\", \"text\": \"App Sync failed\", \"icon_emoji\": \":ghost:\""}"
```

```
22      - "https://hooks.slack.com/services/..."\n23  restartPolicy: Never\n24  backoffLimit: 2
```

In addition to PostSync hooks, you can define other hooks such as:

- PreSync
- SyncFail
- Skip
- Sync

And instead of HookSucceeded, you can use HookFailed or BeforeHookCreation.

24 Creating CI/CD Pipelines for Microservices

24.1 Continuous integration, delivery, and deployment of microservices

In a microservice architecture, multiple services work together to provide business functionality. The system achieves its goals through the collaboration of these services. If a service is not working properly, the entire system may be affected. Therefore, it is important to have a CI/CD pipeline for each service. Each service could have its own repository, or you could have a mono-repository for all services. This is a matter of preference. However, it is important to build and test each service independently with new changes. Additionally, the entire system should also be tested whenever a new service is added or an existing service is changed.

Before being tested, all libraries and dependencies should be installed. After tests are passed, the service should be packaged and deployed to a staging environment that is as close as possible to the production environment. If there are no issues in the staging environment, the service can be deployed to the production environment.

This is a typical CI/CD pipeline for a microservice. Depending on your needs, microservices architecture, team topology, business requirements, infrastructure, and other factors, things can be more complex.

24.2 CI/CD tools

There are many CI/CD tools available for creating a pipeline for your microservices. These tools have different features, capabilities, learning curves, and

drawbacks. Some are open-source while others are not. Some are cloud-native while others are not. Additionally, they can work together or with other tools. For example, you can create a pipeline with Jenkins and use Argo CD for deployment. You can use GitHub actions for building and testing and use Jenkins for deployment..etc

Here are some of the most popular CI/CD tools:

24.2.1 Jenkins

One of the most popular CI/CD tools. It is open-source and it has a large community. It is easy to install and configure and it has a lot of plugins that you can use to extend its functionality. Jenkins is written in Java and it is cross-platform.

However, Jenkins could be sometimes resource intensive. It is also not easy to scale. Additionally, you may need to have some knowledge of Groovy to create advanced pipelines.

24.2.2 Spinnaker

Spinnaker is an open-source CI/CD tool created by Netflix. Written in Java, it is cross-platform and designed to be cloud-native. Spinnaker works with Kubernetes, AWS, Google Cloud, Azure, and other cloud providers.

Spinnaker provides two core sets of features:

- Application management: offers management features to view and manage your cloud resources.
- Application deployment: provides deployment features to construct and manage continuous delivery workflows.

Spinnaker consists of several microservices that work together to provide the above features.

While Spinnaker is a powerful tool, it may be difficult for beginners to learn and use. Additionally, it is more focused on deployment.

24.2.3 Argo CD

Argo CD is an open-source declarative, GitOps continuous delivery tool for Kubernetes. It's often used to manage microservices deployments, especially when microservices are packaged as Docker containers and orchestrated with Kubernetes.

Written in Go and cross-platform, Argo CD is easy to install and configure. It also offers a user-friendly UI that you can use instead of the CLI.

Although powerful, Argo CD requires some knowledge of Kubernetes, Helm, Kustomize, and other tools. It's more focused on deployment than other aspects of the development process.

24.2.4 GitHub Actions

GitHub Actions is a CI/CD tool that integrates with GitHub. It uses a YAML declarative approach, which is common in the Cloud Native ecosystem. Developers can define workflows that are triggered by events such as push, pull request, and issue.

GitHub Actions is easy to use and free for public repositories. Due to GitHub's popularity as a source code management tool, GitHub Actions has a wide community with many examples, tutorials, and workflows available in its marketplace.

However, it is not open-source or entirely free. Additional usage beyond the free tier of 2,000 minutes of GitHub Actions per month and 500 MB of storage will be billed. More information about pricing can be found [here](#)¹⁵⁴.

Furthermore, GitHub Actions is not an option if your source code is hosted elsewhere.

24.2.5 GitLab CI/CD

GitLab provides a robust continuous integration and deployment system that is built into its platform. With its rich features, GitLab CI/CD can be configured to meet the needs of nearly any development workflow. Some of its features include:

¹⁵⁴<https://docs.github.com/en/github/setting-up-and-managing-billing-and-payments-on-github/about-billing-for-github-actions>

- Auto DevOps: a set of default CI/CD templates for popular languages and frameworks.
- ChatOps: a way to run and obtain results of CI/CD pipelines from within a chat window.
- Integration with cloud services: GitLab CI/CD uses OpenID Connect to authenticate users and authorize access to cloud providers.

The free version includes 400 units of compute per month and 5GB of storage. If you require more, you can upgrade to a paid plan. More information about the pricing can be found [here](#)¹⁵⁵.

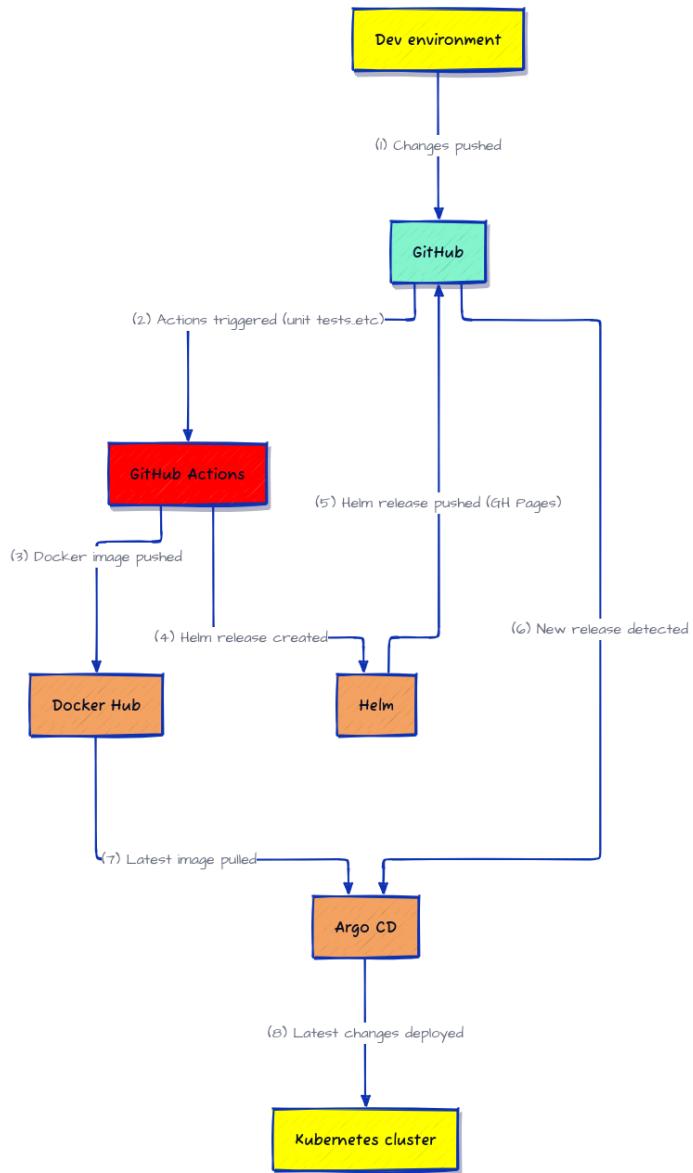
24.3 Creating a CI/CD pipeline for a microservice

In the following sections, we will create a CI/CD pipeline for a microservice. Our goal is to simplify the process to help you better understand it. Once you have grasped the basics, everything will become easier. If you have specific needs, you can refer to the documentation to find the right solutions. We recommend starting small and then growing and customizing your pipelines. Complexity may hide several glitches that could be detected when things are kept basic on the first run.

We will use GitHub to host our source code and GitHub Actions to create our pipeline. The pipeline will begin by executing unit tests. If the tests pass, the application will be built, and a Docker image will be created and pushed to Docker Hub with the “latest” tag. Finally, a Helm package will be created and deployed to Kubernetes using Argo CD.

The following diagram illustrates the process:

¹⁵⁵<https://about.gitlab.com/pricing/>



CI/CD with GitOps, GitHub, Argo CD and Helm

24.3.1 Install and configure Argo CD

Let's start by deploying an Ingress controller. We are going to use the NGINX Ingress controller:

```
1 helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
2 helm repo update
3 helm install nginx-ingress ingress-nginx/ingress-nginx --set controller\\.\\
4 .publishService.enabled=true
```

Store the IP address of the ingress controller:

```
1 export INGRESS_IP=$(kubectl -n default get svc | grep ingress | grep LoadBalancer | awk '{print $4}')
```

Install Argo CD:

```
1 kubectl create namespace argocd
2 kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml
```

Create an Ingress for Argo CD:

```
1 kubectl apply -f - <<EOF
2 apiVersion: networking.k8s.io/v1
3 kind: Ingress
4 metadata:
5   name: argocd-server-ingress
6   namespace: argocd
7   annotations:
8     kubernetes.io/ingress.class: "nginx"
9     nginx.ingress.kubernetes.io/force-ssl-redirect: "true"
10    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
11 spec:
12   rules:
```

```
13    - http:
14      paths:
15        - pathType: Prefix
16          path: /
17          backend:
18            service:
19              name: argocd-server
20              port:
21                name: https
22              host: argocd.${INGRESS_IP}.nip.io
23            tls:
24              - hosts:
25                - argocd.${INGRESS_IP}.nip.io
26              secretName: argocd-secret
27 EOF
```

You can now get the public URL of Argo CD:

```
1 echo https://argocd.${INGRESS_IP}.nip.io
```

The default login is admin and the password can be retrieved with the following command:

```
1 kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{\\"
2 .data.password\\}" | base64 -d; echo
```

24.3.2 Create a GitHub repository for our microservice

To begin, create a GitHub repository. From your development machine, install the necessary tools for our Flask microservice.

```
1 apt get update && apt install -y python3-pip
2
3 pip install virtualenvwrapper
4 export WORKON_HOME=~/Envs
5 mkdir -p $WORKON_HOME
6 export VIRTUALENVWRAPPER_PYTHON='/usr/bin/python3'
7 source /usr/local/bin/virtualenvwrapper.sh
```

Create a virtual environment and install the required dependencies:

```
1 mkvirtualenv ci-cd-with-argocd-helm-and-github-actions
```

Next, create the folders for your Flask application and install its dependencies.

```
1 mkdir -p ci-cd-with-argocd-helm-and-github-actions
2 cd ci-cd-with-argocd-helm-and-github-actions
3 mkdir -p app
4 mkdir -p charts
5 pip install Flask==2.2.3
6 pip install Flask-Testing==0.8.1
7 pip freeze > app/requirements.txtbash
```

Use the following code to create a simple Flask application:

```
1 cat << EOF > app/app.py
2 """ A simple todo application """
3 from flask import Flask, jsonify, request
4
5 app = Flask(__name__)
6
7 tasks = []
8
9 @app.route('/tasks', methods=['GET'])
10 def get_tasks():
11     """
```

```
12     Get all tasks
13     """
14     return jsonify({'tasks': tasks})
15
16 # Route for getting a single task
17 @app.route('/tasks', methods=['POST'])
18 def add_task():
19     """
20     Add a task
21     """
22     task = {
23         'id': len(tasks) + 1,
24         'title': request.json['title'],
25         'description': request.json['description'],
26     }
27     tasks.append(task)
28     return jsonify(task), 201
29
30 if __name__ == '__main__':
31     app.run(debug=True, host='0.0.0.0', port=5000)
32 EOF
```

We are going to create some unit tests for our application:

```
1 cat << EOF > app/test_app.py
2     """ Test the app """
3
4     import unittest
5     import json
6     from app import app
7
8     class TestApp(unittest.TestCase):
9         """
10            Test the app
11            """
12
```

```
13     def setUp(self):
14         """
15             Set up the test client
16         """
17         self.app = app.test_client()
18
19     def test_get_tasks(self):
20         """
21             Test get all tasks
22         """
23         response = self.app.get('/tasks')
24         self.assertEqual(response.status_code, 200)
25
26     def test_add_task(self):
27         """
28             Test add a task
29         """
30         response = self.app.post('/tasks', json={
31             'title': 'Test task',
32             'description': 'Test description'
33         })
34         self.assertEqual(response.status_code, 201)
35         self.assertEqual(json.loads(response.get_data()), {
36             'id': 1,
37             'title': 'Test task',
38             'description': 'Test description'
39         })
40
41 if __name__ == '__main__':
42     unittest.main()
43 EOF
```

We can also test the application manually to ensure that it is working as expected:

```
1 python app/test_app.py
```

You should see the following output:

```
1 Ran 2 tests in 0.009s
2
3 OK
```

Next, we are going to create a Dockerfile:

```
1 cat << EOF > app/Dockerfile
2 # Use an official Python runtime as a parent image
3 FROM python:3.9-slim-buster
4 # Set the working directory to /app
5 WORKDIR /app
6 # Copy the current directory contents into the container at /app
7 COPY . /app
8 # Install any needed packages specified in requirements.txt
9 RUN pip install --no-cache-dir -r requirements.txt
10 # Make port 5000 available to the world outside this container
11 EXPOSE 5000
12 # Define environment variable
13 CMD ["python", "app.py"]
14 EOF
```

In this guide, the repository is referred to as “eon01/ci-cd-with-argocd-helm-and-github-actions”. If you want to follow the instructions, use your own repository name. Alternatively, you can clone the repository from [my GitHub account](#)¹⁵⁶.

To push the repository from the command line:

¹⁵⁶<https://github.com/eon01/ci-cd-with-argocd-helm-and-github-actions>

```
1 export user_name=<your username>
2 export repository_name="ci-cd-with-argocd-helm-and-github-actions"
3
4 git init
5 git add .
6 git commit -m "first commit"
7 git branch -M main
8 git remote add origin git@github.com:${user_name}/${repository_name}.git
9 git push -u origin main
```

Before proceeding with this step, you should have already generated an SSH key and added it to your GitHub account with read/write permission. If you have not done so already, please follow the instructions provided [here](#)¹⁵⁷.

24.3.3 Create a Docker Hub account

To push our Docker image and use it in our Kubernetes cluster, we need to create a Docker Hub account. You can create one [here](#)¹⁵⁸.

For the rest of this guide, we assume that your Docker Hub username is the same as your GitHub username.

24.3.4 Setting up GitHub Actions

The setup for GitHub Actions is now complete.

Before proceeding, we need to configure a few secrets in our GitHub repository. These secrets will be used by GitHub Actions to push our Docker image to Docker Hub. To do this, go to your GitHub repository and click on “Settings”, then “Secrets and variables”. Choose “Actions”, then click on “New repository secret” and add the following secrets:

- DOCKER_USERNAME: Your Docker Hub username
- DOCKER_PASSWORD: Your Docker Hub password

¹⁵⁷<https://docs.github.com/en/github/authenticating-to-github/connecting-to-github-with-ssh>

¹⁵⁸<https://hub.docker.com/signup>

GitHub Actions are workflows triggered by events. In our case, we want to trigger a workflow when we push to the main or master branch of our repository.

Each workflow is defined in a YAML file. In our case, we will use the following file:

```
1 .github/workflows/main.yml
```

Create the directory structure:

```
1 mkdir -p .github/workflows
```

Create the workflow file:

```
1 cat << EOF > .github/workflows/main.yml
2 name: Test, build, push Docker image and release Helm chart
3 on:
4   push:
5     branches: [main, master]
6   workflow_dispatch:
7 jobs:
8   perform_unit_tests:
9     runs-on: ubuntu-latest
10    steps:
11      - uses: actions/checkout@v2
12      - uses: actions/setup-python@v2
13      - run: pip install -r app/requirements.txt
14      - run: python app/test_app.py
15   build_and_push_to_docker_hub:
16     runs-on: ubuntu-latest
17     needs: perform_unit_tests
18     steps:
19       - uses: actions/checkout@v2
20       - run: echo "\$\{\{ secrets.DOCKER_PASSWORD \}\}" | docker login -u "\${{ secrets.DOCKER_USERNAME }}"
21     - run: docker build -t \${{ secrets.DOCKER_USERNAME }}/${repository}
```

```

23   ry_name}:latest -f app/Dockerfile app
24     - run: docker push \${{ secrets.DOCKER_USERNAME }}/${repository_n\
ame}:latest
25   ame}:latest
26   release_helm_chart:
27     permissions:
28       contents: write
29     runs-on: ubuntu-latest
30     needs: build_and_push_to_docker_hub
31     steps:
32       - name: Checkout
33         uses: actions/checkout@v3
34         with:
35           fetch-depth: 0
36       - name: Configure Git
37         run: |
38           git config user.name "\$GITHUB_ACTOR"
39           git config user.email "\$GITHUB_ACTOR@users.noreply.github.co\
m"
40         m"
41       - name: Install Helm
42         uses: azure/setup-helm@v3
43       - name: Run chart-releaser
44         uses: helm/chart-releaser-action@v1.5.0
45         env:
46           CR_TOKEN: "\${{ secrets.GITHUB_TOKEN }}"
47 EOF

```

The above code defines a GitHub Actions workflow for automating the testing, building, and pushing of Docker images, and the release of Helm charts.

name: - This line specifies the name of the workflow.

on: - This part defines the events that trigger the workflow. In this case, any push to the ‘main’ or ‘master’ branch or manually triggering the workflow (`workflow_dispatch`) will initiate the workflow.

The workflow has three jobs:

1. **perform_unit_tests:** - This job is responsible for running unit tests:

- It specifies that the job should run on the latest version of Ubuntu.
- It checks out the code using `actions/checkout@v2` and sets up Python.
- It then installs the application's requirements and runs the unit tests.

2. `build_and_push_to_docker_hub`: - This one builds a Docker image and pushes it to Docker Hub:

- This job also runs on the latest version of Ubuntu and depends on the successful completion of the 'perform_unit_tests' job.
- It checks out the code.
- It logs into Docker using the secrets `DOCKER_USERNAME` and `DOCKER_PASSWORD`.
- It then builds the Docker image using the Dockerfile located in the app directory and pushes it to Docker Hub.
- It is possible to use your own Docker registry instead of Docker Hub.

3. `release_helm_chart`: - This job releases the Helm chart:

- It configures the job permissions to allow write access to the repository contents.
- It checks out the code and configures Git with the GitHub actor as the user name and email.
- It installs Helm using the `azure/setup-helm` action.
- It releases the chart using the `helm/chart-releaser-action` with the `GITHUB_TOKEN` for authentication.

Add the new files to the repository and push them:

```
1 git add .
2 git commit -m "Add CI/CD workflow"
3 git push origin main
```

24.3.5 Create a Helm chart

Helm will help us package and deploy our application to Kubernetes. Whenever we need to deploy a new version of our application, we simply need to update the Helm chart and push it to our GitHub repository. GitHub Actions will then create a new release with the latest version of our application.

You will need to install Helm on your machine. You can find installation instructions [here¹⁵⁹](#).

To start, create a new Helm chart:

```
1 mkdir -p charts/cicd-flask
2 helm create charts/cicd-flask
```

To modify the default values of our Helm chart, edit the `values.yaml` file. Use the following values:

```
1 cat << EOF > charts/cicd-flask/values.yaml
2 replicaCount: 1
3 image:
4   repository: ${user_name}/${repository_name}
5   pullPolicy: IfNotPresent
6   tag: "latest"
7 imagePullSecrets: []
8 nameOverride: ""
9 fullnameOverride: ""
10 serviceAccount:
11   create: true
12   annotations: {}
13   name: ""
14 service:
15   type: ClusterIP
16   port: 5000
17 ingress:
```

¹⁵⁹<https://helm.sh/docs/intro/install/>

```
18   enabled: true
19   className: ""
20   annotations:
21     kubernetes.io/ingress.class: nginx
22   hosts:
23     - host: flask.${INGRESS_IP}.nip.io
24       paths:
25         - path: /tasks
26           pathType: ImplementationSpecific
27   tls: []
28 autoscaling:
29   enabled: false
30   minReplicas: 1
31   maxReplicas: 100
32   targetCPUUtilizationPercentage: 80
33 EOF
```

Let's create another values file to use for our staging environment:

```
1 cp charts/cicd-flask/values.yaml charts/cicd-flask/values-staging.yaml
```

To customize the values in the values-staging.yaml file, you can adjust the number of replicas or the image tag, among other things.

Now, edit the deployment.yaml file to update the default livenessProbe and readinessProbe:

```
1 cat << EOF > charts/cicd-flask/templates/deployment.yaml
2 apiVersion: apps/v1
3 kind: Deployment
4 metadata:
5   name: {{ include "cicd-flask.fullname" . }}
6   labels:
7     {{- include "cicd-flask.labels" . | nindent 4 }}
8 spec:
9   {{- if not .Values.autoscaling.enabled }}
```

```
10  replicas: {{ .Values.replicaCount }}
11  {{- end --}}
12  selector:
13    matchLabels:
14      {{- include "cicd-flask.selectorLabels" . | nindent 6 --}}
15  template:
16    metadata:
17      {{- with .Values.podAnnotations --}}
18        annotations:
19          {{- toYaml . | nindent 8 --}}
20      {{- end --}}
21    labels:
22      {{- include "cicd-flask.selectorLabels" . | nindent 8 --}}
23  spec:
24    {{- with .Values.imagePullSecrets --}}
25    imagePullSecrets:
26      {{- toYaml . | nindent 8 --}}
27    {{- end --}}
28    serviceAccountName: {{ include "cicd-flask.serviceAccountName" . \
29  }}
30  securityContext:
31    {{- toYaml .Values.podSecurityContext | nindent 8 --}}
32  containers:
33    - name: {{ .Chart.Name }}
34    securityContext:
35      {{- toYaml .Values.securityContext | nindent 12 --}}
36      image: "{{ .Values.image.repository }}:{{ .Values.image.tag }} \
37 default .Chart.AppVersion }}"
38    imagePullPolicy: {{ .Values.image.pullPolicy }}
39    ports:
40      - name: http
41        containerPort: {{ .Values.service.port }}
42        protocol: TCP
43    livenessProbe:
44      httpGet:
45        path: /tasks
```

```
46      port: http
47      readinessProbe:
48          httpGet:
49              path: /tasks
50              port: http
51      resources:
52          {{- toYaml .Values.resources | nindent 12 --}}
53          {{- with .Values.nodeSelector --}}
54      nodeSelector:
55          {{- toYaml . | nindent 8 --}}
56          {{- end --}}
57          {{- with .Values.affinity --}}
58      affinity:
59          {{- toYaml . | nindent 8 --}}
60          {{- end --}}
61          {{- with .Values.tolerations --}}
62      tolerations:
63          {{- toYaml . | nindent 8 --}}
64          {{- end --}}
65 EOF
```

After editing the Helm chart, we typically need to package it and push it to a Helm repository. Tools like [ChartMuseum](#)¹⁶⁰ can help with this process. However, for this guide, we will publish the Helm chart to a GitHub Pages site.

However, we have already configured our GitHub workflow to publish the chart.

To access the packages using a URL, we need to convert the GitHub repository into a GitHub Pages site. Follow these steps:

Create a new empty branch named `gh-pages` and push it to the repository.

¹⁶⁰<https://chartmuseum.com/>

```
1 git checkout --orphan gh-pages
2 git rm -rf .
3 git commit -m "Initial commit" --allow-empty
4 git push --set-upstream origin gh-pages
```

To deploy your website using GitHub Pages, follow these steps:

1. Go to your repository settings and scroll down to the “GitHub Pages” section.
2. Under “Source”, select “Deploy from a branch”.
3. Select the gh-pages branch and the / (root) directory.
4. Give your workflow read and write permissions to the repository by going to “Actions” in your repository settings, clicking on “General”, and selecting “Read and write permissions” in the Workflow permissions section.

Add the new files to the repository.

```
1 git add .
2 git commit -m "Add Helm chart"
3 git push
```

24.3.6 Create an Argo CD Application

Create the Argo CD Application resource, which will instruct Argo CD to deploy our application using the Helm chart created earlier. It will search for the Helm chart in the GitHub Pages site we previously created, using the “index.yaml” file. This file contains a list of packages available in the repository, and was generated by the GitHub workflow using the Helm Chart Releaser action.

Note that we are using `https://{$user_name}.github.io/{$repository_name}/` when specifying the Helm repository URL. This is because we are hosting our Helm chart on GitHub Pages. If your repository URL is `https://github.com/you/app`, then the Helm repository URL will be `https://you.github.io/app/`.

We are also using the `values-staging.yaml` file as the values file for our staging environment. If you want to deploy to production, use the `values.yaml` file instead. You can also create new values files for other environments.

```
1 kubectl apply -f - <<EOF
2 apiVersion: argoproj.io/v1alpha1
3 kind: Application
4 metadata:
5   name: cicd-flask
6   namespace: argocd
7 spec:
8   destination:
9     namespace: default
10    server: https://kubernetes.default.svc
11  project: default
12  source:
13    chart: cicd-flask
14    repoURL: https://${user_name}.github.io/${repository_name}/
15    targetRevision: 0.1.0
16    helm:
17      valueFiles:
18        - values-staging.yaml
19  syncPolicy:
20    automated:
21      prune: true
22      selfHeal: true
23      allowEmpty: true
24 EOF
```

In this example, we are using DigitalOcean. However, there is a known issue with the CiliumIdentity CRD (as described in [this GitHub issue¹⁶¹](#)) that prevents the application from being deployed. To resolve this issue, we need to add an exclusion to the Argo CD ConfigMap.

¹⁶¹<https://github.com/argoproj/argo-cd/issues/10456>

```
1 kubectl apply -f - <<EOF
2 apiVersion: v1
3 kind: ConfigMap
4 metadata:
5   labels:
6     app.kubernetes.io/name: argocd-cm
7     app.kubernetes.io/part-of: argocd
8   name: argocd-cm
9   namespace: argocd
10 data:
11   resource.exclusions: |
12     - apiGroups:
13       - cilium.io
14     kinds:
15       - CiliumIdentity
16     clusters:
17       - "*"
18 EOF
```

If you are using a different cloud provider, you might not need to do this.

24.3.7 Automating the deployment of new versions

Argo CD continuously monitors the Git repository and automatically deploys any changes to the application. To test this, we will update the image tag in the values-staging.yaml file.

Instead of having a single replica, we will have three replicas.

```
1 cat << EOF > charts/cicd-flask/values-staging.yaml
2 # change the replica count to 3 here
3 replicaCount: 3
4 image:
5   repository: ${user_name}/${repository_name}
6   pullPolicy: IfNotPresent
7   tag: "latest"
8 imagePullSecrets: []
9 nameOverride: ""
10 fullnameOverride: ""
11 serviceAccount:
12   create: true
13   annotations: {}
14   name: ""
15 service:
16   type: ClusterIP
17   port: 5000
18 ingress:
19   enabled: true
20   className: ""
21   annotations:
22     kubernetes.io/ingress.class: nginx
23   hosts:
24     - host: flask.${INGRESS_IP}.nip.io
25       paths:
26         - path: /tasks
27           pathType: ImplementationSpecific
28   tls: []
29 autoscaling:
30   enabled: false
31   minReplicas: 1
32   maxReplicas: 100
33   targetCPUUtilizationPercentage: 80
34 EOF
```

Upgrade the release version inside the file charts/cicd-flask/Chart.yaml:

```
1 sed -i 's/version: .*/version: 0.2.0/g' charts/cicd-flask/Chart.yaml
```

Add the new files to the repository:

```
1 git add .
2 git commit -m "Update values-staging.yaml"
3 git push
```

Upgrade the Argo CD Application

```
1 kubectl apply -f - <<EOF
2 apiVersion: argoproj.io/v1alpha1
3 kind: Application
4 metadata:
5   name: cicd-flask
6   namespace: argocd
7 spec:
8   destination:
9     namespace: default
10    server: https://kubernetes.default.svc
11  project: default
12  source:
13    chart: cicd-flask
14    repoURL: https://${user_name}.github.io/${repository_name}/
15    targetRevision: 0.2.0
16    helm:
17      valueFiles:
18        - values-staging.yaml
19  syncPolicy:
20    automated:
21      prune: true
22      selfHeal: true
23      allowEmpty: true
24 EOF
```

You should see the new version of the application deployed.

This is exactly what happens during the whole process:

- You push the changes to the main branch of the repository.
- The GitHub workflow is triggered.
- The workflow builds the Docker image and pushes it to Docker Hub.
- The workflow packages the Helm chart and pushes it to the GitHub Pages site.
- A new index.yaml file is created that references the new version of the Helm chart and the archive file.
- Argo CD detects the changes in the index.yaml file and deploys the new version of the application.
- The application is deployed with the new changes.

It is recommended to create a new release whenever you have made changes to a chart. This is because chart releases should be immutable in order to avoid any issues and preserve the reproducibility of deployments.

25 Afterword

25.1 What's next?

Congratulations on completing “Cloud Native Microservices with Kubernetes”! I hope that this journey into the world of cloud-native microservices and Kubernetes has provided you with valuable insights and practical knowledge.

Throughout this guide, we explored various topics related to building, deploying, and managing microservices using Kubernetes and we delved into important concepts like containerization, architecture, resource management strategies, autoscaling, deployment strategies, observability, and GitOps.

By now, you should have a solid understanding of the principles and best practices for developing and operating highly scalable, resilient, and efficient cloud-native microservices in a Kubernetes environment.

Remember that practice makes perfect, and the best way to learn is by doing. So, if you haven’t already, I encourage you to try out the examples in this guide and experiment with the concepts and techniques presented here.

I hope you had a great time reading this guide. Keep learning and stay curious!

25.2 Thank you

Thank you for joining me on this exciting adventure, and I wish you all the best in your future endeavors.

25.3 About the author

Aymen El Amri is a polymath software engineer, author, and entrepreneur. He is the founder of FAUN Developer Community, a platform that helps developers learn and grow in their careers. He is also the author of multiple books on

software development and cloud computing. You can find him on [Twitter¹⁶²](#) and [LinkedIn¹⁶³](#).

25.4 Join the community

If you enjoyed this guide, you can join [FAUN community¹⁶⁴](#) to be notified of future free and paid guides, courses, and other resources.

25.5 Feedback

I eagerly anticipate hearing your insights and feedback on this new chapter. Your thoughts will contribute significantly to this book's continued refinement and expansion.

If my work has resonated with you, I would be delighted if you could share your testimonial by sending me an email to aymen@faun.dev. Your experiences and words can serve as a beacon for future readers. And of course, I'd love to share your testimonial with my wider reader community.

¹⁶²<https://twitter.com/eon01>

¹⁶³<https://www.linkedin.com/in/elamriaymen/>

¹⁶⁴<https://faun.dev/join>