

POLITECNICO DI TORINO

Master's Degree in Computer Engineering



Master's Degree Thesis

GitOps and ArgoCD: Continuous deployment and maintenance of a full stack application in a hybrid cloud Kubernetes environment

Supervisor

Candidate

Prof. Fulvio RISSO

Matteo D'AMORE

Academic Year 2020-2021

Summary

The thesis work focuses on the development of a solution for the Continuous Deployment of environments on Kubernetes clusters (hybrid cloud), based on the GitOps approach. Nowadays, cloud computing and containers are increasingly used in business environments.

Kubernetes is the most common container orchestrator, and it can be used with the vanilla version or with other distributions. Kubernetes allows users to manage application infrastructures using a declarative approach, thanks to YAML files. For this reason, it makes sense to place these files within a Git repository.

If the infrastructure updates are made only by operating on the infrastructure repository and as a result of these changes, the environments on Kubernetes are updated, GitOps approach is achieved. In a nutshell, GitOps embraces many good practices introduced by the DevOps philosophy, like the CI/CD pipelines and Infrastructure as Code. It introduces something new: Git repository as a unique source of truth. It is a developer-centric approach, since developers are familiar with Git and because of this, it is interesting to exploit this paradigm to give developers more autonomy in managing the infrastructure.

“With the demands made on today’s infrastructure, it is crucial for organizations to implement infrastructure automation that is repeatable, traceable, and less prone to human error. GitOps takes DevOps best practices used for application development, such as version control, collaboration, compliance, and CI/CD, and applies them to modern infrastructure automation”.

As a result, the development team becomes more independent of infrastructure team. These environments are used to develop new features, test new releases, and update versions of the application in the production state. At the same time, the company does not want to give developers free access to the infrastructure repositories, which contain YAML manifests (Secrets, Deployments, Services, etc.), used for defining the environments to be deployed within Kubernetes clusters.

Part of the thesis work concerns the development of a “pull-based” GitOps pipeline, using ArgoCD as a continuous deployment operator and GitHub Actions for the continuous integration and other automated tasks. Developers can operate on infrastructure repositories in a controlled and indirect way.

By streamlining the interaction between the development and the infrastructure team, one of the advantages is the reduction in the average time required to provision or modify an environment on Kubernetes. In order to maintain the principles of “need to know” and “least privilege”, developers can’t access infrastructure repositories directly.

Thanks to the GitOps paradigm, to ArgoCD and to an abstraction level provided by the YAML file called "input" (conceived during the thesis work), developers can generate custom environments on Kubernetes, without them knowing the Kubernetes details and the management/compilation of YAML files describing k8s resources. The simplicity of interfacing is another strong point of this solution, since only the knowledge of Git (a tool familiar to developers) and the compilation of the “input” file is required.

The input file is a sort of “access key” to interact with the infrastructure files in a controlled way.

By using the “input” file, the following things can be chosen for each environment: the clusters in which to deploy, the frontend and backend images to be used, the infrastructure parameters (replicas, ports, etc), the secrets and the configMaps. These parameters can be redefined as many times as users like, as long as the environment exists on k8s.

Two Kubernetes cluster were created and managed during the thesis work. The first one is an on-premise cluster, which implements the vanilla version of Kubernetes. It is composed by a worker and a master node. These two nodes were created from two VMs with Centos7 OS. The second one is a Microsoft Azure cluster. A different cloud service provider was chosen to demonstrate that the solution is also multi-vendor.

The thesis work also includes the use of a branching strategy and a tag strategy, which is automatically managed by the GitHub Actions workflows.

Within the infrastructure repository, the Kubernetes environments are managed by Kustomize, which is a configuration management tool. This tool is very interesting because of its feature of redefining resources written declaratively in YAML files, leaving the base files unchanged. The directory structure is also very readable.

Acknowledgements

Voglio ringraziare tutti coloro che mi hanno supportato ed accompagnato in
questo percorso di laurea magistrale.

Table of Contents

List of Tables	IX
List of Figures	X
Acronyms	XIII
1 Introduction	1
2 Background	3
2.1 Containers	3
2.1.1 Container vs VM	3
2.2 Docker	5
2.3 Cloud Computing	9
2.3.1 Cloud Solutions	9
2.4 Mutable and Immutable infrastructure	10
2.5 Kubernetes	11
2.5.1 Architecture	12
2.6 Kustomize	18
2.6.1 <i>kustomization</i> file	18
2.6.2 Build Kustomize resources	19
2.6.3 Overlays	19
2.6.4 secretGenerator	24
2.6.5 images update	24
2.6.6 Patches	25
2.7 GitHub Actions	28
3 Towards GitOps	29
3.1 Git	29
3.2 DevOps	31
3.2.1 Principles	31
3.2.2 DevOps benefits	32

3.2.3	DevOps tools	33
3.3	GitOps	34
3.3.1	Principles	34
3.3.2	Benefits	35
3.3.3	Tools	36
3.3.4	Push-based pipeline	37
3.3.5	Pull-based pipeline	37
4	Argo CD	39
4.1	Introduction	39
4.2	How it works?	39
4.3	Why ArgoCD?	41
4.3.1	ArgoCD vs other solutions	41
4.3.2	Why not a CI/CD pipeline?	42
4.3.3	Features	42
4.3.4	ArgoCD Architecture	44
4.4	Application	45
4.5	Project	48
4.6	App of Apps	48
4.7	High Availability	48
4.8	Disaster Recovery	48
5	Solution	49
5.1	Introduction	49
5.2	Business Case	50
5.3	Branching strategy	51
5.3.1	Introduction	51
5.3.2	<i>Git Flow</i> branching strategy	51
5.3.3	Branches	52
5.4	<i>input</i> file	56
5.4.1	<i>backend, frontend</i> and <i>db</i> fields	56
5.4.2	image field	57
5.4.3	branch	59
5.4.4	clusters	60
5.5	<i>config</i> file	60
5.6	Repositories	62
5.6.1	Code Repositories	62
5.6.2	Infrastructure Repository	63
5.6.3	ArgoCD Repository	66
5.6.4	Image Repositories	68
5.6.5	Backup-files Repositories	68

5.7	Workflows	69
5.7.1	Code Repositories	69
5.7.2	Infrastructure Repository	71
5.7.3	ArgoCD Repository	74
5.8	Example	76
5.8.1	Initial setup	76
5.8.2	Startup and deployment of the production environment . . .	78
5.9	Objectives and Validation	88
5.9.1	Infrastructure incident recovery	88
5.9.2	Deployment time of environments	89
5.9.3	<i>input</i> file	91
5.9.4	Workflows execution time	92
5.9.5	Reduction of provisioning and update time	93
5.9.6	Approval process	94
5.10	Liqo	94
5.10.1	Liqo transparency	96
5.10.2	Liqo alongside ArgoCD	96
5.10.3	Implementation	96
6	Conclusion	98
	Bibliography	100

List of Tables

4.1	Tools comparison	42
4.2	ArgoCD vs CI/CD tool.	43

List of Figures

2.1	Containers vs VMs [1]	4
2.2	Kubernetes adoption [9]	11
2.3	Kubernetes architecture [11]	12
3.1	DevOps Process [25]	33
3.2	Push-based pipeline [27]	37
3.3	Pull-based pipeline [28]	38
4.1	ArgoCD keeps the live state synchronised with the desired state	40
4.2	ArgoCD web UI example	44
4.3	ArgoCD Architecture [31]	44
5.1	<i>master</i> and <i>release</i> branches point to two different commits	51
5.2	Branching strategy [33]	52
5.3	How <i>input</i> file is applied	56
5.4	Repositories and teams	62
5.5	kustomize directory structure	65
5.6	kustomize example	65
5.7	Manifests directory	66
5.8	<i>config</i> files content	76
5.9	<i>config</i> files content	77
5.10	<i>input</i> file within the <i>master</i> branch	79
5.11	<i>prod</i> directory within the <i>thesis-infrastructure</i> repository	80
5.12	Application manifests within the ArgoCD repository	80
5.13	Command to apply the initial manifest	80
5.14	Content of the initial manifest	81
5.15	ArgoCD web UI	81
5.16	<i>prod</i> environment deployed in the on-premise cluster	82
5.17	<i>prod</i> environment deployed in the Azure cluster	82
5.18	The <i>input</i> file related to the new feature <i>features/f1</i>	83
5.19	The new directory <i>f1</i> created within the infrastructure repository	83

5.20	The manifests within the ArgoCD repo, after the creation of the new feature <i>f1</i>	84
5.21	ArgoCD web UI with the new environments associated with <i>features/f1</i>	84
5.22	<i>features/f1</i> deployed in the on-premise cluster	84
5.23	<i>features/f1</i> deployed in the Azure cluster	85
5.24	The <i>input</i> file related to the new feature <i>releases/r1</i>	85
5.25	ArgoCD web UI with the new environment associated with <i>releases/r1</i>	86
5.26	<i>releases/r1</i> deployed in the Azure cluster	86
5.27	ArgoCD reacts to the deletion of the manifest associated with the <i>release/r1</i> branch	86
5.28	New tag for the master branch	87
5.29	Re-synchronisation time (U=update, D=delete, BE=backend, FE=frontend, Dep=Deployment, Svc=Service)	89
5.30	First test on deployment time	90
5.31	Second test on deployment time	90
5.32	Third test on deployment time	91
5.33	Example of an <i>input</i> file	91
5.34	Ratings	92
5.35	Average execution time for each workflow. Code, Infr and ArgoCD refer to the repositories	93
5.36	Percentage of time required to build and push the Docker image . .	93

Acronyms

K8S

Kubernetes

CRD

Custom Resource Definition

YAML

Yet Another Markup Language

VM

Virtual Machine

CI

Continuous Integration

CD

Continuous Delivery

IaC

Infrastructure as Code

UI

User Interface

VMM

Virtual Machine Manager

OS

Operative System

PAT

Personal Access Token

Chapter 1

Introduction

The IT world is constantly evolving. In the mid 90's, infrastructures consisted only of hardware purchased by the company and maintained within its own buildings. There were specialised operators who had to take care of the physical servers manually. In this context, the servers were managed using the mutable infrastructure approach, which consists of maintaining and modifying the server in place.

With regard to software development, at that time code was written in isolation. The structured versioning did not exist. Compilation and testing of code was handled manually by the development team, as well as for creating release packages. Then, the release package was given to the operations team, who were in charge of deploying it. There was a variable time lag, from a few hours to many days before the new release actually went into production. This time depended on the availability and skills of the operations team. Moreover, there was little communication and co-operation between development and operations teams. From the point of view of the development team, there was frustration because of the possible long deployment times for new releases. On the other hand, the operations team was frustrated if the new release had problems when deployed. This resulted in employee dissatisfaction and slow software development and release times.

At the beginning of 2000, the Agile methodology took off. It improved the development process but not the deployment process, so something was still missing.

Around 2008, the concept of DevOps emerged, as a consequence of a discussion on the drawbacks of the Agile approach. DevOps is a philosophy, which with the help of practices and tools, has enabled the automation and integration of processes between software development and operations teams. From that moment, the software can be built, tested and released in a faster and more reliable way. It reduced the separation between the development and operations teams.

Among the many new features introduced, it is worth mentioning CI/CD pipelines and Infrastructure as Code.

Returning to the subject of mutable infrastructures, there have also been big

improvements in this area. The first improvement was the introduction of the virtualization. It is a technology that made possible the introduction of virtual machines first and containers (lightweight VMs) later. VMs have made it possible for companies to create virtual servers instead of physical ones. Thanks to it, a single physical machine can host multiple instances of servers.

In this scenario, the immutable infrastructure approach emerged. It means that a server, once it has been provisioned, will no longer be modified. If a new version is needed, a new virtual server will be provisioned and the old one will be removed. In 2006, Amazon launched Amazon Web Services, which offers services to other clients via the internet. From this event, came the era of cloud computing. Nowadays, cloud computing is becoming increasingly popular.

Cloud computing, DevOps best practices, containers and containers orchestrator, like Kubernetes, have generated the GitOps approach. In a nutshell, GitOps is focused on Continuous Deployment, in a cloud native environment. When it was conceived, it was made to work with Kubernetes. It introduces Git as a single source of truth and Git repositories are used to hold the infrastructure files, expressed declaratively, following the Infrastructure as Code approach. One of the great advantages of GitOps is the pull based pipeline, which is possible thanks to operators such as ArgoCD or Flux.

Today, cloud native applications are more and more common and Kubernetes is heavily used to manage them. It is necessary for developers, in addition to working on the source code, to be more independent in infrastructure management, as daily deployments can be done hundreds of times.

Automation of the processes of provisioning and updating of k8s environments, related to features, releases and production is something that would improve the productivity and cooperation between development and operations teams, speeding up deployment times and consequently the development life cycle.

As a result, operations team is relieved of a major burden and can invest its time in other actions, which bring value to the company. Developers can work with the infrastructure, without having to know in detail how Kubernetes works and the commands it requires. They can build and customize environments on Kubernetes without directly writing infrastructure YAML files.

Chapter 2

Background

2.1 Containers

Containers are an alternative to VMs. Specifically, the container is a form of lightweight virtualization. The idea behind containers is a system that can guarantee the properties of computer virtualization, consuming less resources.

2.1.1 Container vs VM

Containers and virtual machines are often confused. Although there are some similarities, containers differ from VMs in several aspects.

Virtual machines emulate the hardware system. Each VM runs a specific guest operating system, along with libraries, binaries, and applications. A physical server can host multiple VMs, each with potentially a different OS. Generally, there is one application per VM, in order to isolate the environment. The hypervisor or VMM, is a software, firmware or hardware that allows the creation and management of multiple VMs.

VMs are a very good solution, but they consume a large amount of system resources and generate overhead, especially when there are multiple VMs running on the same physical host.

On the other hand, containers share the same host OS. Thanks to this, containers are much lighter in size and generate less overhead. Lightness turns into speed, as a container takes just a few seconds to run, whereas a VM can take up to a few minutes.

Container advantages

- **Encapsulation:** containers enclose into a unique entity the application source code, dependencies and network configuration.

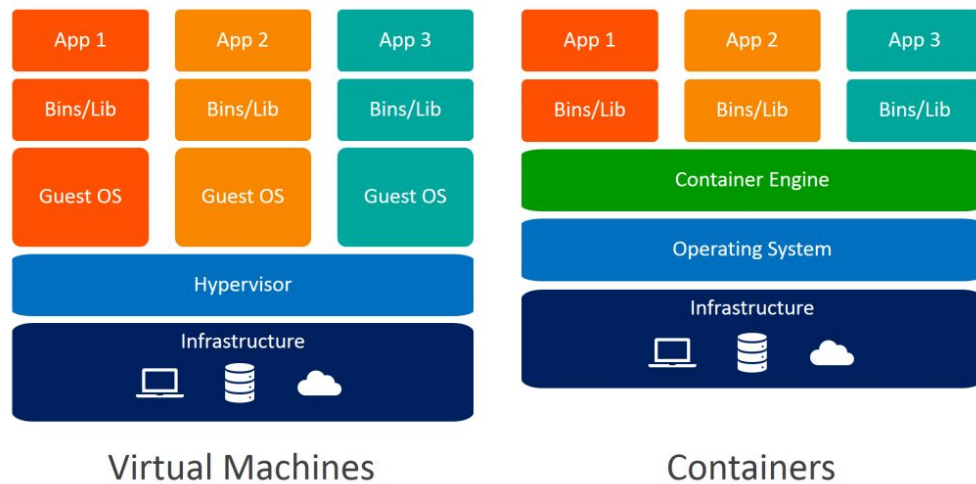


Figure 2.1: Containers vs VMs [1]

- **Lightweight:** containers require less system resources than virtual machines because they don't include operating system overhead.
- **Scalability:** physical host machines can contain hundreds of containers.
- **Portability:** containers can be deployed easily to multiple different operating systems and hardware platforms.
- **Versioning:** containers are versioned, in fact the container name is composed by a hash, which contains also the version.
- **Consistent operation:** DevOps teams know applications in containers will run the same, regardless of where they are deployed.
- **Efficiency:** containers allow applications to be more rapidly deployed, patched and scaled.
- **DevOps oriented:** containers support Agile and DevOps efforts to accelerate development, test and production cycles.

2.2 Docker

Docker is an open source tool designed to make it easier to manage applications by using containers [2]. It provides an implementation that standardises the use of containers on different platforms. The main components are:

- **Docker Engine;**
- **Docker Client;**
- **Docker Image;**
- **Docker Container.**

Docker Engine

Docker Engine is the platform core. It is a daemon process, executed in background on the host machine. Docker Engine provides access to all the functionalities and services made available by Docker.

A Docker container can be moved across different machines (with Docker Engine installed) and it will work in the same way, even if the machines are running two different OSes. This is a great achievement, as it is often the case that when moving an application to different execution environments, something gets broken [3].

Docker Client

Docker Client communicates with Docker daemon (Docker Engine). It is not necessary for Docker Client and the daemon to run on the same physical machine. Docker Client can connect to a remote Docker daemon. Communication takes place through a REST API, over UNIX sockets or a network interface.

Docker Image and Container

A Docker Image is an immutable template that contains a set of instructions for creating a Docker container [4]. Then, a Docker Container is a running instance of an image.

The image is immutable, so it never changes and it is a strong advantage, since users always know what they are going to run, independently from the environment. Generally, an image is identified by *registry/user/nameImg:tag*, where *latest* is the default tag. Modularity is a feature of images, indeed an image can be composed by many read-only layers, which are images too. When a container is created from an image, it is added a new writable layer, called *container layer*. Even if a container is stopped and restarted, it maintains changes within the filesystem. Thanks to the

modularity, it can be possible that multiple images share the same N base layers and this is an advantage in terms of memory usage.

Docker Registry

Docker images are pulled and pushed from and to repositories. Repositories lies into Registries. Every host has a own local registry. A user can create its own remote registries. The official remote Docker registry is *Docker Hub*.

Dockerfile

It is possible to build a new image from a Dockerfile. Dockerfile is a text file that contains a set of instructions used to build an image.

```
1  #Dockerfile example
2
3  FROM ubuntu:16.04
4  LABEL version="v1.1"
5  RUN apt get update && apt get y install apache2
6  COPY index.html /
7  var /www/ index.html
8  VOLUME /
9  var /www
10 EXPOSE 80
11 CMD ["D", "FOREGROUND"]
12 ENTRYPOINT ["apachectl"]
```

Each instruction in a Dockerfile creates a new layer that will compose the resulting image. For the sake of efficiency, it may be useful to group instructions whenever possible.

```
1  #this command creates only one layer, BETTER
2  RUN apt get update && apt get y install ifconfig && apt get install traceroute
3
4  #these three commands create three different layers
5  RUN apt get update
6  RUN apt get y install ifconfig
7  RUN apt get y install traceroute
```

The main commands are described below.

FROM

This command must be written first in the Dockerfile. It defines the base image on which to build the image.

LABEL

It is an optional command, used to add metadata to the image. Labels are key-value pairs.

RUN

This instruction executes into Docker Engine a shell command. It is used to install software and packages useful to execute the container based on this image.

CMD

The CMD instruction is used to execute a shell command at runtime, when a container is executed from the image. In each Dockerfile, there can be only one CMD instruction. If there are more than one, only the last one is executed. The main goal is to give default instruction that the container will execute.

ENTRYPOINT

ENTRYPOINT has the same purpose of CMD. It defines the program to be executed when the container starts up. It is possible to parameterize the image by passing via the CLI parameters to the executable defined by ENTRYPOINT. However, there are slight differences with CMD:

- A command passed with the CLI can override CMD but not ENTRYPOINT.
- The CMD command, if present together with the command ENTRYPOINT command, defines parameters treated in the same way as those the same way as those passed from the command line.

COPY

COPY instruction copies a new file into image destination directory, from a source path. Source file must be inside the host "context of build".

ENV

ENV instruction sets the environment variables that will reside inside the container.

VOLUME

The VOLUME instruction creates a mount point for the specified path.

EXPOSE

The EXPOSE instruction exposes a container port through which it can be contacted.

2.3 Cloud Computing

Cloud computing means delivering services over the internet. Cloud computing appeared after the advent of virtualization. VMs enable a much more flexible use of the resources and servers are put in data centers. Amazon in 2006 created the first cloud computing service, Amazon AWS. It was offering to customers, on-demand computing, storage and networking resources.

2.3.1 Cloud Solutions

If a company wants to interface with the world of the cloud computing, there are several solutions it can adopt, depending on business needs [5].

Private cloud

The private cloud can be hosted by a cloud service provider or can be on-premise (physically located within company data center). In both cases, resources are used exclusively and are delivered via a secure private network. The company has full control over data and services. This is the best solution if the company wants to match a fine grained custom specifications.

With the on-premise solution, the company must buy and manage its own servers. The management is done by its own staff.

The company has full control over data and services and the physical resources are not shared with anyone else. The disadvantage of this solution is the cost, since the building which host physical devices, cooling systems, physical devices and management are a big expense for the company. Another disadvantage is the lack of flexibility. If the company wants to increase its own resources, it has to buy, configure and manage new resources, and it takes a lot of time.

Public cloud

With this kind of solution, cloud resources are provided by a third-party cloud service provider via the internet. Cooling, hardware, software and other infrastructure devices are managed by the cloud service provider. There is no need to buy hardware and software or even to pay someone for the maintenance. It is easy to scale up and down the resources, since cloud service provider provides them on-demand. In a public cloud, resources are shared among multiple tenants and they are not used exclusively.

Hybrid cloud

It is a mix of private and public cloud solutions. A possible solution is to use private cloud environment for sensitive activities and the public cloud for tasks that are not sensitive.

2.4 Mutable and Immutable infrastructure

Before virtualization, all infrastructures were mutable. All servers were physical and once deployed, needed continuous maintenance, such as bug fixes, upgrades and modifications. Mutable means that modifications are made in place [6]. Maintenance had to be done manually, server by server. This approach has many disadvantages [7]:

- Each server is unique because of the myriad manual changes made over time. This makes it very difficult to diagnose and manage each server.
- Many times, changes on a server are not documented. This makes versioning impossible to maintain.
- Updates can fail, as in a complex environment it can happen that something stops working properly. It introduces non-negligible downtimes.
- Hard Debugging.

With the advent of virtualization and cloud computing, the concept of immutable infrastructure has emerged. This change was caused by the movement from physical to virtual servers. Immutable infrastructure means that servers are never modified after deployment. When a server needs to be upgraded or modified, it is replaced by a new, properly updated one (a new version). Every update is versioned, automated and new servers are provisioned quickly. Errors and configuration drifts are much more rare. The horizontal scaling is easy to do, thanks to virtualization, which allows multiple identical copies of the server to be deployed quickly. Docker and Kubernetes are tools related to the immutable infrastructure.

2.5 Kubernetes

"Kubernetes (also known as k8s or "kube") is an open source container orchestration platform that automates many of the manual processes involved in deploying, managing, and scaling containerized applications" [8].

It is currently the most used orchestrator, widely adopted across diverse businesses.

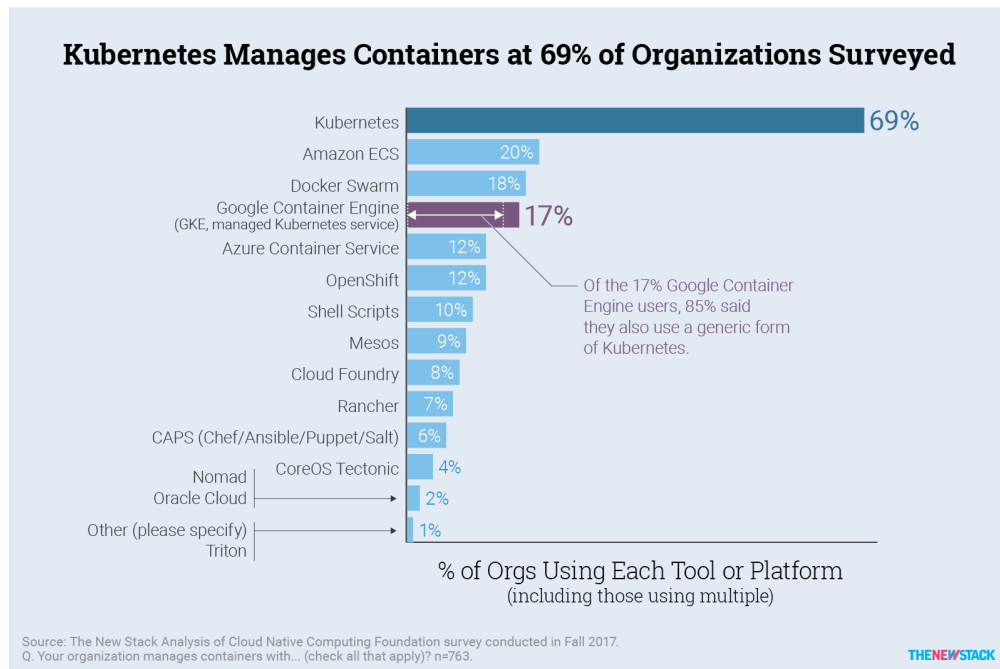


Figure 2.2: Kubernetes adoption [9]

Kubernetes was introduced in 2014 as an open source version of the internal Google orchestrator Borg.

Borg was introduced by Google around 2003-2004. It was a large-scale internal cluster management system, which ran a huge number of jobs, from a great number of applications, across many clusters, each with up to tens of thousands of machines [10].

Starting from 2017-2018, Kubernetes has been adopted by many companies. One of the reasons why Kubernetes gained popularity so fast is its open source architecture and an incredible number of documentation and support provided by its community. There are multiple Kubernetes distributions. There is a vanilla version, the basic one, and many others, such as RedHat OpenShift, Tectonic, Rancher and so on.

2.5.1 Architecture

The k8s architecture is shown in 3.1. A cluster is a set of resources that are coordinated by a single instance of k8s. Each Kubernetes cluster is composed by a master node only or a master node and one or more worker nodes.

- **Master node:** it must necessarily exist, since it is the central point of control of the entire cluster. It may be duplicated for redundancy and load balancing. It controls and manages a set of worker nodes. It can work also as worker node, if the cluster is composed only by this node.
- **Worker node:** it hosts the pods, which are the components of the application workload. It also contains some agents which communicate with the master node.

Thanks to this structure, Pods can be distributed on different nodes, balancing the resource consumption. Master node is composed by many components such as API server, controller manager, scheduler and etcd database. Instead, worker nodes have only kube-proxy and kubelet.

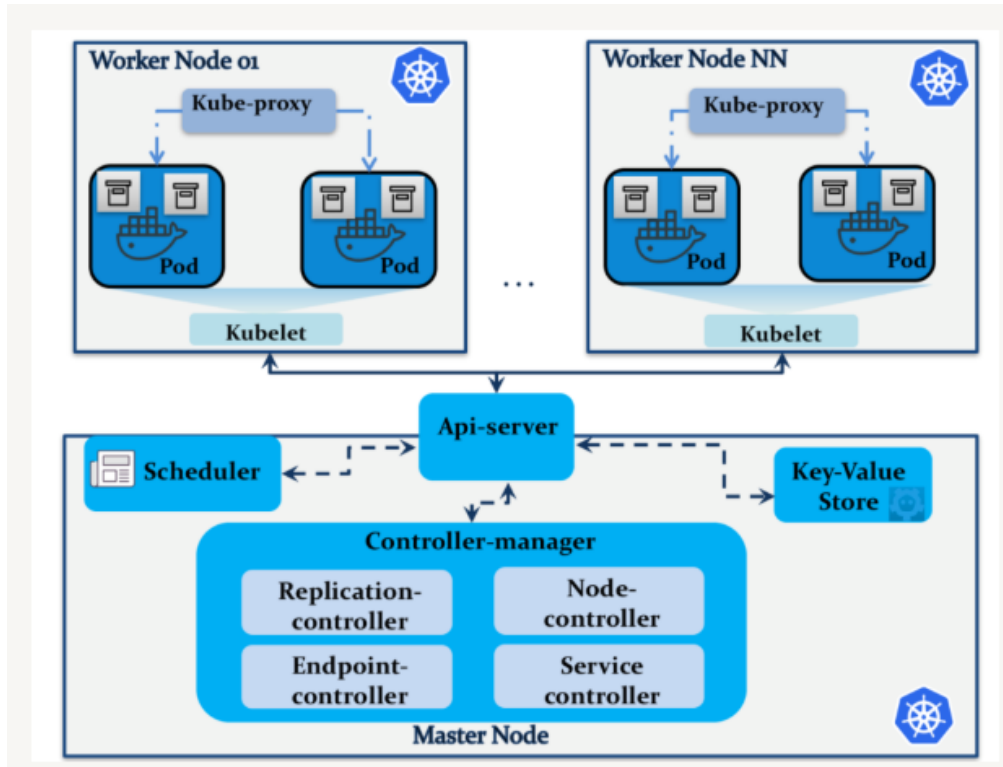


Figure 2.3: Kubernetes architecture [11]

Master Node Components

Master node, the control plane unit of Kubernetes cluster, is responsible for making global decisions, maintaining the status of all objects in the cluster and reacting to events. It continuously performs control loops to monitor changes.

Master node is the component used by users to interact with Kubernetes. Clients make requests and it make sure to take the actual state of cluster towards the desired one. The Master node components contains:

- **API Server:** the API Server exposes the Kubernetes API. It allows users to validate and configure all cluster objects such as pods, deployments, services etc. Moreover, it processes requests, validates them, and updates the objects in etcd.
- **Controller Manager:** it embeds the control loop. If there are changes, the controller manger synchronise the current state of cluster with the desired one. In Kubernetes, there are multiple controllers, like Endpoints Controller, Namespace Controller, Replication Controller.
- **etcd:** etcd is a consistent, highly available and distributed key-value data store. It is used to store all cluster data. Etcd is written in Go and acts based on the Raft consensus algorithm.
- **Scheduler:** if there are unscheduled pods, it assigns them to different worker nodes, evaluating multiple metrics, such as availability of computing resources, affinity and anti-affinity specifications, policy constraints of pods or quality of service requirements.

Worker Node Components

Worker nodes are simpler than Master node. Each worker node hosts Pods and two daemons:

- **Container Runtime:** Container runtime is responsible for running containers. There are multiple choices supported by Kubernetes, but the most common is Docker.
- **Kubelet:** this daemon is responsible for monitoring the containers. It makes sure that all containers are running properly in Pods. Using PodSpecs, the kubelet checks that the containers described into them are running and healthy. For instance, if a pod fails, the kubelet restarts it. In any case, it manages only containers which are created by Kubernetes.

- **Kube-Proxy:** Kube-Proxy is a network proxy, which maintains network rules on each worker node. These rules allow network communications to the pods from inside or outside of the Kubernetes cluster.

Pod

The smallest deployable and manageable unit in Kubernetes is called pod. Each pod consists of one or more containers, which are deployed on the same physical host.

All the containers inside a pod are tightly coupled, because they share the same resources, like storage volumes and networking.

Pod has a unique IP address, shared by all containers. Containers within the pod see each other on the *localhost* interface, whereas containers that belong to different pods communicate with the IP address of their pod. Kubernetes orchestrate pods instead of containers.

Hence, a pod is very similar to a virtual machine or in terms of Docker, it is similar to a group of Docker containers with shared filesystem volumes and namespaces. A pod is bound to a node throughout its lifecycle.

Replication Controller

Replication Controller is responsible to ensure that the specified number of pods are always up and running at any one time, and if not, it converges the number of pods to the required one. More specifically, it terminates pods if they are too many or it starts more pods if they are too few. Each pod replication is called *replica*.

It is a good practice to define a Replication Controller instead of creating manually a pod. If a pod is created manually, it can be evicted in case of any failure. Using a replication controller to create pods, it replaces pods if they are deleted, fail or are terminated. This controller ensures a safer method to maintain application healthy.

ReplicaSet

Traditional infrastructure model treats servers as pets (*pets model*). If a server goes down, it must be fixed, in order to make it available again. With this approach, servers are unique and indispensable.

The other model is the *cattle model*, used by Kubernetes. With this approach, servers are only replicas. Servers can fail. There are no worries if a server fails, as a new one, equal to the previous, is created.

ReplicaSet is an API object in Kubernetes, which manages scaling of pods. It maintains the desired number of pods at any given time, constantly checking their

state. ReplicaSets are the upgrade of Replication Controllers, since they provide more features.

Deployment

Deployment is a higher-level concept than ReplicaSet and generally users rely on this object to handle pods. Deployments provide a declarative way to enforce updates to pods and ReplicaSets. When a new version of the Deployment is provided, a new ReplicaSet is created. Pods are moved increasingly from the old ReplicaSet to the new one.

Service

A Service allows to expose applications running on the pods. As pods are ephemeral, they can be created or removed by Replication Controller and their IP addresses are not stable. Each newly created pod will receive a new IP address.

Service instead is a stable access point that never changes.

Users should not connect to the applications with pod IPs. Kubernetes services solve the issue offering an endpoint API, which lets services being accessible externally. In addition, Kubernetes has an internal domain discovery process, where each service has assigned a single DNS name. The DNS name corresponds to:

- *<serviceName>*, if source and destination are inside the same namespace.
- *<serviceName.namespace.svc.cluster.local>*, if the source and destination are in different namespaces.

There are different ways to expose a service [12]:

- **ClusterIP**: it exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default *ServiceType*.
- **NodePort**: it exposes the Service on each Node's IP at a static port (*NodePort*). A ClusterIP Service, to which the NodePort Service routes, is automatically created. Users will be able to contact the NodePort Service, from outside the cluster, by requesting *<NodeIP>:<NodePort>*.
- **LoadBalancer**: it exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
- **ExternalName**: it maps the Service to the contents of the externalName field, by returning a CNAME record with its value.

Namespace

Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

Namespaces are intended for use in environments with many users spread across multiple teams, or projects. If a cluster is shared among few users, it is not mandatory to leverage the namespace mechanism. Namespaces provide a scope for names. Names of resources need to be unique within a namespace. Instead, accross different namespaces there can be equal resource names. Namespaces cannot be nested and each k8s resource can only be in one namespace.

kubectl

This is a line tool that interacts with *kube-apiserver* and send commands to the Master node. Each command is converted into an API call. The *kubectl* tool supports three kinds of object management:

- Imperative commands
- Imperative object configuration
- Declarative object configuration

Declarative Management of Kubernetes Objects

"Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using kubectl apply to recursively create and update those objects as needed. This method retains writes made to live objects without merging the changes back into the object configuration files" [13].

Thanks to configuration YAML file, it is possible to define the k8s objects that describe the infrastructure in a declarative way. An example of a YAML file that represents a Deployment k8s object is given below:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: myApp
5  spec:
6    selector:
7      matchLabels:
8        app: myApp
9    template:
10     metadata:
11       labels:
12         app: myApp
```

```
13     spec:
14       containers:
15         - name: nginx
16           image: nginx:latest
17           ports:
18             - containerPort: 80
```

Custom resource definition

Kubernetes is highly configurable and extensible. Custom resource definition (CRD) is one of the things that k8s allows you to do. The CRD API resource allows users to define custom resources, which are extensions of the Kubernetes API [14]. A resource is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind.

Once a custom resource is installed, users can create and access its objects using `kubectl`. The name of a CRD object must be a valid DNS subdomain name.

2.6 Kustomize

Kustomize is a Kubernetes native configuration management tool [15]. It is a standalone tool that customizes k8s objects through a *kustomization* file [16]. Kustomize adds, removes and updates k8s files without forking them, with a pure declarative approach [17]. To make customization simpler, Kustomize offers a range of useful methods, such as *generators* (Secret generator, ConfigMap generator). Using *patches*, Kustomize adds environment-specific updates to existing files without touching them.

2.6.1 *kustomization* file

This is the core file which enables the power of Kustomize. It is a YAML file, and it must be defined inside each folder that it is wanted to be managed by Kustomize.

Let's suppose there is the folder *myApp* that contains k8s YAML files and it is wanted to be managed by Kustomize. The file structure is the following:

```
1 myApp/
2   myDeployment.yaml
3   myService.yaml
4   kustomization.yaml
```

A basic content of the *kustomization* file could be:

```
1 apiVersion: kustomize.config.k8s.io/v1beta1
2 kind: Kustomization
3 resources:
4 - myDeployment.yaml
5 - myService.yaml
```

As it can be seen, in order to include the files that will be managed by Kustomize, they must be added as list, in *resources*. You have to add to the list only those files that you want Kustomize to manage (e.g. only *myDeployment.yaml* or *myService.yaml* could have been added).

A *kustomization* file can contain a huge amount of fields, which can be divided in four big categories:

- **Resources:** the existing resources that are to be customized, like Deployments, Services, CRDs;
- **Generators:** what new resources should be created. In this set for instance there is the *secretGenerator*, which allows the creation of k8s secrets;

- **Transformers:** they transform resources (e.g. change the container image of a specific pod);
- **Meta:** they are fields used to influence the three categories above.

In these sections only some fields will be inspected (i.e. fields that were used to develop the thesis work solution).

2.6.2 Build Kustomize resources

Once the *kustomization* file is defined, in order to obtain the resources updated, there are two different approaches. The first one is to use a specific Kustomize command alone:

```
kustomize build <folder_path>
```

It is important to remark that the folder path must contain the *kustomization* file. If it is wanted also to deploy in k8s the new resources generated, the command is:

```
kustomize build <folder_path> | kubectl apply -f -
```

There is a second way to do the same thing. Starting with Kubernetes 1.14, Kustomize had been integrated with kubectl. Then the command in this case is:

```
kubectl apply -k <folder_path>
```

2.6.3 Overlays

A very interesting and helpful feature of Kustomize is the concept of *bases* and *overlays*. A *base* directory, is a directory that contains a *kustomization* file and a set of k8s resources expressed within YAML files.

An *overlay* directory, is a directory with a *kustomization* file that refers to other directories, called *bases*. An *overlay* may have multiple *bases*.

In the *overlays* directory, base files can be updated without touching them. It is added a new logical layer where resources are manipulated. With this approach, let's suppose there is a base directory that contain basic k8s objects, which describe an application. It can be defined an *overlay* directory, which customises the base version, changing the number of replica, the namespace, the image version and so on.

While all this is being done, the base directory is not touched and files within it remain the same.

Overlays Example

Assume that there is a simple application, composed by a Deployment and a Service, and you want to deploy it on Kubernetes, using two different Docker images and two different namespaces.

```
1 kustomize/
2   base/
3     myDeployment.yaml
4     myService.yaml
5     kustomization.yaml
6   overlays/
7     version1/
8       kustomization.yaml
9     version2/
10      kustomization.yaml
```

In the *base* directory there are the definitions of Deployment, Service k8s objects and *kustomization* file:

```
1 #myDeployment.yaml
2
3 apiVersion: apps/v1
4 kind: Deployment
5 metadata:
6   name: thesis-app-dep
7   labels:
8     app: thesis-app
9 spec:
10  selector:
11    matchLabels:
12      app: thesis-app
13  template:
14    metadata:
15      labels:
16        app: thesis-app
17    spec:
18      containers:
19        - image: app-owner/thesis-app:latest
20          name: thesis-app
21          imagePullPolicy: Always
22          ports:
23            - name: http
24              containerPort: 8080
```

```
1 #myService.yaml
2
3 apiVersion: v1
4 kind: Service
5 metadata:
6   name: service
7   labels:
8     app: thesis-app
9 spec:
10  selector:
11    app: thesis-app
12  ports:
13    - protocol: TCP
14      port: 8080
15      targetPort: 8080
```

```
1 #kustomization.yaml
2
3 apiVersion: kustomize.config.k8s.io/v1beta1
4 kind: Kustomization
5 namespace: base-ns
6 resources:
7 - myDeployment.yaml
8 - myService.yaml
```

In order to see the output resources, you have to build with the command *kubectl build kustomize/base*. The output is:

```
1 #kustomize base output
2
3 apiVersion: v1
4 kind: Service
5 metadata:
6   labels:
7     app: thesis-app
8   name: service
9   namespace: base-ns
10 spec:
```

```
11   ports:
12   - port: 8080
13     protocol: TCP
14     targetPort: 8080
15   selector:
16     app: thesis-app
17   ---
18   apiVersion: apps/v1
19   kind: Deployment
20   metadata:
21     labels:
22       app: thesis-app
23     name: thesis-app-dep
24     namespace: base-ns
25   spec:
26     selector:
27       matchLabels:
28         app: thesis-app
29     template:
30       metadata:
31         labels:
32           app: thesis-app
33       spec:
34         containers:
35         - image: app-owner/thesis-app:latest
36           imagePullPolicy: Always
37           name: thesis-app
38           ports:
39           - containerPort: 8080
40             name: http
```

Now, as has been said previously, the *kustomization* file within the folder *kustomize/overlays/version1/* was written with the intention of modifying the Docker image and the namespace name. Moreover, in order to use *kustomize/base/kustomization.yaml* as base, this must be specified in the *kustomize/overlays/version1/kustomization.yaml*:

```
1 #kustomization.yaml within version1 directory
2
3 apiVersion: kustomize.config.k8s.io/v1beta1
```

```
4  kind: Kustomization
5  namespace: version1-ns
6  resources:
7  - ../../base
8  images:
9  - name: app-owner/thesis-app:latest
10  newTag: custom-tag
```

After building, the output is:

```
1  #output
2
3  apiVersion: v1
4  kind: Service
5  metadata:
6    labels:
7      app: thesis-app
8    name: service
9    namespace: version1-ns
10 spec:
11   ports:
12   - port: 8080
13     protocol: TCP
14     targetPort: 8080
15   selector:
16     app: thesis-app
17   ---
18 apiVersion: apps/v1
19 kind: Deployment
20 metadata:
21   labels:
22     app: thesis-app
23   name: thesis-app-dep
24   namespace: version1-ns
25 spec:
26   selector:
27     matchLabels:
28       app: thesis-app
29   template:
30     metadata:
```

```
31     labels:
32       app: thesis-app
33   spec:
34     containers:
35     - image: app-owner/thesis-app:custom-tag
36       imagePullPolicy: Always
37       name: thesis-app
38       ports:
39       - containerPort: 8080
40       name: http
```

As it can be seen, the namespace and the image tag has changed as expected. It is important to remember that the resources within *base* directory, have not been touched during these steps. A different customisation could be generated for *version2* folder.

2.6.4 secretGenerator

Secret resources can be easily generated in the *kustomization* file thanks to the *secretGenerator* field. The same is for *configMapGenerator*. Each entry in the list corresponds to a secret resource. Secrets can be generated from files, literals or environment variables.

2.6.5 images update

kustomization file includes "images" field, which contains a list of images that can be modified. For each existing image, users can modify the name, tag or digest. "images" only updates existing images and it does not create new ones. In order to point a specific image, the field *name* must be the name of an existing image.

```
1  # myDeployment.yaml
2
3  apiVersion: apps/v1
4  kind: Deployment
5  metadata:
6    name: dep
7  spec:
8    template:
9      spec:
10     containers:
```

```
11     - name: mynginx
12       image: nginx:1.7.0
```

```
1  # kustomization.yaml
2
3  apiVersion: kustomize.config.k8s.io/v1beta1
4  kind: Kustomization
5  images:
6  - name: mynginx
7    newName: mynginx-newName
8    newTag: newTag
9  resources:
10 - deployment.yaml
```

```
1  # output result
2
3  apiVersion: apps/v1
4  kind: Deployment
5  metadata:
6    name: dep
7  spec:
8    template:
9      spec:
10        containers:
11        - image: mynginx-newName
12          name: newTag
```

2.6.6 Patches

In the previous sections it has been shown how to update resource values like the namespace or the images and how to add secrets. If users want to operate in a more specific way, adding or overriding fields on existing resources, they must use *patches* field in *kustomization* file.

This field contains a list of patches to be applied to the resources. There are two different ways to patch a resource: *strategic merge patch* and *JSON patch*. Only the *JSON patch* will be discussed here, as it is the approach used in the thesis work.

In general, *patches* field contain a list of element:

```
1 # patches field example
2
3 patches:
4 - path: backend_backend-dep_deployment_patch.yaml
5   target:
6     labelSelector: tier=backend
7     name: backend-dep
8 - path: frontend_frontend-service_service_patch.yaml
9   target:
10    labelSelector: tier=frontend
11    name: frontend-service
```

The *path* field identifies a file containing a list of entries which follow the *json 6902* standard. The *target* field is used to select one or more YAML files to apply the customisations indicated in the file which is the *path* value.

Since *target* can include multiple fields for filtering, the selected resources are all those that match all fields. In the example, *name* identifies the *metadata.name* of the resource, while *labelSelector* identifies one of the labels expressed in *metadata.labels*. The resource that are pointed are those in the *base* directory.

```
1 # json 6902 file example
2
3 - op: add
4   path: /spec/replicas
5   value: 1
6 - op: add
7   path: /spec/template/spec/containers/0/env
8   value:
9     - name: MYSQL_ROOT_PASSWORD
10     valueFrom:
11       secretKeyRef:
12         key: MYSQL_ROOT_PASSWORD
13         name: db-db-dep
14 - name: MYSQL_DATABASE
15   valueFrom:
16     secretKeyRef:
17       key: MYSQL_DATABASE
18       name: db-db-dep
```

Every entry is composed by:

- **op**: this field can assume different values, like `add`, `remove`, `replace`, etc. The most interesting is *add*, as this is enough to do everything necessary on resources. The *add* operation adds (if the target does not exist) or replaces (if the target exists) the value in the target location [18].
- **path**: this field indicates the target object within the YAML file.
- **value**: this field indicates the value to be taken by the object indicated in the *path*.

2.7 GitHub Actions

GitHub Actions is a tool provided by GitHub that implements workflows, with whom CI/CD pipelines can be created [19]. Each workflow is represented by a YAML file, that must be created into `.github/workflows/` path. This path must be within the GitHub repository.

Workflows can be triggered by many events, like push, deletion of a branch, merging of a pull request, etc. A workflow can be triggered also by another workflow, which may be in the same repository or in a different one. An example of a workflow could be the following:

```
1 name: push-backend
2 on:
3   push:
4     branches:
5       - 'master'
6       - 'features/**'
7       - 'releases/**'
8 jobs:
9   setup:
10    name: Setup
11    runs-on: ubuntu-latest
12    steps:
13      - name: Checkout code
14        uses: actions/checkout@v2
15    # ...
```

- **name** identifies the name of the workflow.
- **on** it is used to choose which events trigger the workflow. In this specific case, the workflow "push-backend" is triggered by push on one of the listed branches.
- **jobs** contains one or more jobs. Each job inside "jobs" contains several actions which will be executed. In that example, "Setup" is a job.
- **runs-on** specifies in which running environment the job has to be executed. There are multiple choices, including "ubuntu-latest", "macos-latest" or "windows-latest".
- **steps** are the tasks inside a job. Each step can run commands, actions within the repository, setup tasks, etc.

Chapter 3

Towards GitOps

3.1 Git

Git is an open source distributed version control system [20]. It is principally used for managing source code in software projects, but it can also be used for generic files.

Version Control

Version control is a system that records changes to one or multiple files. Versions are always accessible and if a user wants to return to a previous state of files handled by Git, he can do it.

Distributed Version Control System

Firstly, there was the *Local Version Control System*, an approach that had a simple database that kept all the changes to a file under revision control. The evolution was the *Centralized Version Control System*, that added the collaboration feature. This feature is achieved thanks to a single centralized server that contains all the versioned files. But this solution has the problem of the single point of failure.

If the centralized server goes down for an hour, nobody can work on it. Then, there is the *Distributed Version Control System*, that is the approach used by Git. With this solution, users don't just check out the latest snapshot of the files, but they clone the entire repository, with the entire history. In this case, if the server goes down, every collaborator has locally a full backup of data.

Git treats its data like a series of snapshots, taking a picture at every *commit* of what all files look like and stores a reference to that snapshot. If the files have not changed, Git does not store them again, but uses just a link to the previous ones it

has already stored, for efficiency reasons.

To share a user's work with other team members, it is needed to connect to the repository located on a remote server on the network. The remote repository has the complete history of all files.

File states

There are three main states that files can reside in: *modified*, *staged*, and *committed*.

- Modified means that it has been changed files but they have not been committed on the local database;
- Staged means that it has been marked a modified file in its current version to go into next commit snapshot;
- Committed means that the data is safely stored in local database.

Commit

A commit is a list of all changes since the parent commit, it is like a snapshot.

Moreover, commits have associated some metadata information, like a unique id that identify the commit within the repository. Every commit must be described by a message, which should describe the changes.

Branch

A branch is just a stream of consecutive commits. It is nothing more than a pointer to a certain commit.

A new branch can be "branched" from any commit, creating a separate branch from that point forward.

It is a stream of one or more contiguous commits. Generally, there is always a starting branch, called *master*.

3.2 DevOps

DevOps is nothing but a set of philosophies, practices, and tools that help an organisation to deliver better products faster by facilitating an integration of the development and operations functions [21].

The word *DevOps* is a combination of the terms *development* and *operations*, meant to represent a collaborative or shared approach to the tasks performed by a company's application development and IT operations teams.

The concept of DevOps emerged out of a discussion between Andrew Clay and Patrick Debois in 2008. They were concerned about the drawbacks of Agile and wanted to come up with a solution [22].

3.2.1 Principles

DevOps is associated with a series of principles [23].

Automation

Automating as much as possible is a DevOps key feature. Automation includes development, testing, configuration and deployment operations. It allows for no human errors, if it is implemented correctly, gaining in reliability.

Moreover, there is a gain also in agility and repeatability. Developers and operations teams can concentrate on other processes, such as adding value and quality to the product.

Continuous Integration

Since developers push code changes frequently, it is convenient to automate the process of building and testing. This is done by Continuous Integration, implemented through pipelines. It allows to detect bugs early and to maintain code in a state that can be deployed effortlessly.

Continuous Deployment/Delivery

It is a consequence of the CI. When the pipeline builds and tests the code changes, Continuous Deployment/Delivery deploys to production that changes, always in an automatic way.

It allows faster time to market, steady deployment process and reliable rollbacks. Continuous Delivery and Continuous Deployment are similar concepts, often confused. Both share the acronym *CD* and can be used with CI. The biggest difference concerns the deployment process.

In Continuous Delivery, code flows automatically through multiple steps to prepare it for production deployment, but does not automatically go live. The code changes must first be manually approved.

Instead, in Continuous Deployment, the code is automatically released into a live production environment, monitoring if the new deployed environment presents problem. In that case a rollback is made.

Infrastructure as Code

IaC allows to manage and provision IT infrastructure through code and automation. Benefits are consistent resource creation and management, reusability, scalability, self-documented infrastructure and simplification of complex infrastructures.

"*Infrastructure as Code* is the management of infrastructure in a descriptive model, using the same versioning as DevOps team uses for source code" [24].

IaC model generates always the same environment every time it is applied, like a source code, which generates always the same binary every time it is compiled. Without IaC, each target environment must be treated individually, which becomes an hard and complex task, especially if there are many different targets.

Continuous Monitoring

This practice ensures that the application runs without problems, collecting data about the performance and the stability of services, systems and infrastructures. It allows to detect problems, in order to make fast recovery.

Containers

If possible, it is better to use containers instead of virtual machines, because of lightness and deployment rapidity. Moreover, a container can be tested easily.

3.2.2 DevOps benefits

Some benefits are:

- fewer silos and increased communication between IT groups;
- faster time to market for software;
- rapid improvement based on feedback;
- less downtime;
- improvement to the whole software delivery pipeline through builds, validation and deployment;



Figure 3.1: DevOps Process [25]

- less manual work, thanks to automation;

3.2.3 DevOps tools

In general, users can rely on CI/CD pipeline, containers and cloud hosting. These tools can be open source, proprietary or supported distributions of open source technology.

- **Code repositories:** version-controlled source code repositories enable multiple developers to work on code and rollback to a previous version of code if needed. Moreover, tracking allows to check changes. GitHub, GitLab, can be two of multiple other choices.
- **Artifact repositories:** source code is compiled into artifacts for testing. Also artifacts should be collected into repositories, like JFrog.
- **CI/CD pipeline:** they enable to validate and deliver application to the end user through automation during the development lifecycle.
Continuous Integration tool initializes processes so that developers can create, test and validate code in a shared repository.
Continuous Delivery extends these automatic steps through production-level tests and configuration setups for release management.
Continuous Deployment does something more, invoking tests, configuration and provisioning, as well as monitoring and potential rollback capabilities. Common tools are Jenkins, GitLab CI, CircleCI, GitHub Actions.
- **Configuration Management:** Puppet, Chef, Ansible.
- **Monitoring:** Dynatrace, Prometheus. They are used to observe the performance and security of code releases.

3.3 GitOps

GitOps was introduced by Weaveworks in 2017. It is a paradigm or a set of practices, which was primarily designed to do Kubernetes cluster management and application delivery. More generally, it can be considered as a cloud native paradigm, strictly related to the continuous delivery process. GitOps is focused on Git as a single source of truth. As it will be seen later, GitOps leverages the many feature and benefits of Git.

It is a developer-centric approach, since developers are familiar with Git and using it to manage infrastructure, makes everything easier. The Git repository contains a declarative description of the infrastructure desired, then it can be called "infrastructure repository". GitOps embraces the IaC pattern, placing it in a cloud native context and combining it with the concepts of orchestration, observability, declarativity and immutable infrastructures. One of the aims of GitOps is to automate the deployment process of what is inside the infrastructure repository. But that is not all, as it is also wanted to automate the synchronization process between the desired state (contained within the Git repository) and the live state (deployed on k8s).

Therefore, it is an operating model for k8s (or other cloud native technologies), for building cloud native applications. Not only does GitOps provide best practices that unify deployment, management and monitoring for containerized applications, but also provides the developer with a better experience for managing applications.

3.3.1 Principles

There are principles to follow if it is wanted to use GitOps:

- **Declarative approach:** declarative means that configuration is guaranteed by a set of facts instead of a set of instruction. Everything must be described in a declarative way and must stay in a Git repository. GitOps was created to be applied to k8s, which uses a predominantly declarative approach. Therefore, it is good that GitOps also follows the same methodology.
- **Desired state versioned:** the desired system state must stay in a Git repository. Since Git is a version control system, the infrastructure repository contains the history of the infrastructure. There is a single place from which everything is derived and driven. It is easy to rollback the infrastructure to a previous state.
- **Changes automatically applied:** the desired state is described in the Git repository. Every time it changes, also the desired state changes. When the change takes place, the system automatically synchronizes with the new

desired state. The most remarkable feature is that the user does not need cluster credentials to make that change.

- **Software agents to ensure correctness and alert on divergence:** once the state of the system is declared and kept under version control, software agents can inform the user whenever the live state does not match the desired expectations and eventually they could make Continuous Deployment, synchronising the live state. The use of agents also ensures that the entire system is self-healing.

Self-healing reinforces the mechanisms already present in k8s.

The software agents act as the feedback and control loop for the operations.

3.3.2 Benefits

The benefits of applying GitOps best practices are:

- **Stored history of infrastructure changes:** as the application infrastructure can only be changed by performing updates in the Git repository, all changes are recorded.
- **Productivity:** Continuous Deployment automation with an integrated feedback control loop speeds up the mean time to deployment. Development and operations teams can ship 30-100 times more changes per day, increasing overall development output 2-3 times [26]. If the user needs a new environment or wants to update an existing one, he just has to modify the Git repository by pushing updates. There is no need to write scripts for Continuous Delivery.
- **Better Developer Experience:** developers can use familiar tools like Git for managing infrastructure. It is not needed to know k8s in details. Furthermore, each team member can check the repository and better understand the changes, leveraging verbose commits and documentation, which should be written by other team members.
- **Stability:** Git repository can be used as a external audit log source, since it contains all cluster changes. It can be known who did what and when, and this can be used to meet SOC 2 compliance.
- **Decoupling CI from CD:** GitOps offers flexibility in the choice of tools. There are GitOps tools like Flux or ArgoCD, which synchronise the infrastructure repository with the live state on Kubernetes (CD). These tools can be used independently with GitHub Actions, GitLab CI, etc. (CI) without any kind of constraint.

- **Reliability:** Git repository allows to easily rollback the infrastructure. Because the entire system is described in Git, there is a single source of truth from which to recover after a meltdown, reducing the mean time to recovery (MTTR) from hours to minutes.
- **Reduction of human error:** infrastructure is completely described in YAML files, declaratively. If files are written properly, the output is always the same.
- **Approval process:** if the company uses tool like GitHub or GitLab, it could rely on the approval mechanism of these tools (pull requests). Some critical branches such as production could pass through a more fine-grained process of acceptance and review. Other branches, such as feature branches, may be subject to fewer controls, being less critical.
- **Consistency and Standardization:** GitOps provides a consistent and standardized end-to-end workflow across the entire organization.
- **Security:** Git's strong correctness and security guarantees, backed by the strong cryptography used to track and manage changes, as well as the ability to sign changes to prove authorship and origin is key to a secure definition of the desired state of the cluster.

3.3.3 Tools

GitOps provides the freedom to use different tools to implement the CI/CD pipeline, there are not strict rules. Open source or proprietary tools can be chosen. The three main tools GitOps native, which provide the continuous delivery, are the following:

- **ArgoCD;**
- **Flux;**
- **Jenkins X.**

It is important to understand that GitOps provides a solution for the Continuous Delivery or Deployment. In order to develop a complete solution, considering Continuous Integration also, other tools must flank the GitOps CD tool, like Jenkins, GitLabCI, Github Actions.

It can be used a k8s controller that implements the operator pattern, to listen for and synchronize deployments to the k8s cluster (e.g. ArgoCD). This solution is more secure and automates the complex task to update YAML files. It allows to achieve Continuous Deployment, because when there are differences between the desired state (Git repository) and the live one, the operator synchronizes the live state.

3.3.4 Push-based pipeline

A push-based pipeline means that code starts with the Continuous Integration steps and could continue its path through a series of scripts or uses *kubectl* by hand to push changes to the k8s cluster. This is the standard CI/CD approach nowadays.

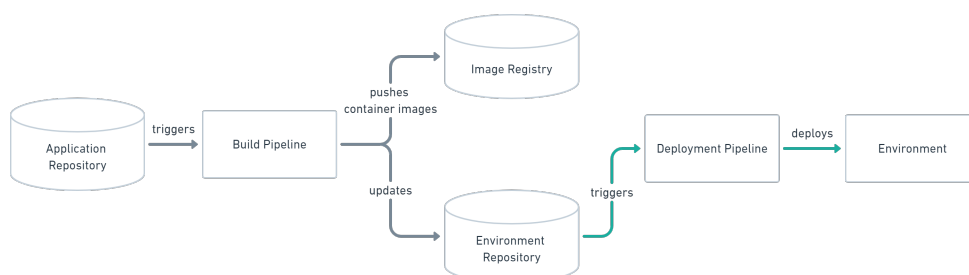


Figure 3.2: Push-based pipeline [27]

This is not the best option, since the deployment is not declarative. Moreover, the CI/CD pipeline must contain the k8s cluster credential and it is a risk. Finally, the disadvantage of this approach is that the live state does not synchronize with the desired one automatically, since there is not a k8s operator like ArgoCD or Flux.

3.3.5 Pull-based pipeline

The pull-based pipeline has the same CI implementation of the pull-based one, but it differs from how the continuous deployment is implemented. An operator is introduced here, like ArgoCD or Flux. It takes over the role of the pipeline by continuously comparing the desired state in the infrastructure repository with the live state in the k8s cluster. Whenever differences are noticed, the operator updates the infrastructure to match the live state. Additionally, the image registry can be monitored to find new versions and to update them. Just like the push-based deployment, this variant updates the environment whenever the environment repository changes.

Moreover, with the operator, changes can also be noticed in the other ways. If the live environment deployed changes in any way not described in the infrastructure repository, these changes are reverted. This ensures that all changes are made traceable in the Git log, by making all direct changes to the cluster impossible. This change in direction solves the problem of push-based deployments, where the environment is only updated when the infrastructure repository changes. Since the operator compares the live state with the desired one, the k8s cluster credential

are not supplied to the pipeline.

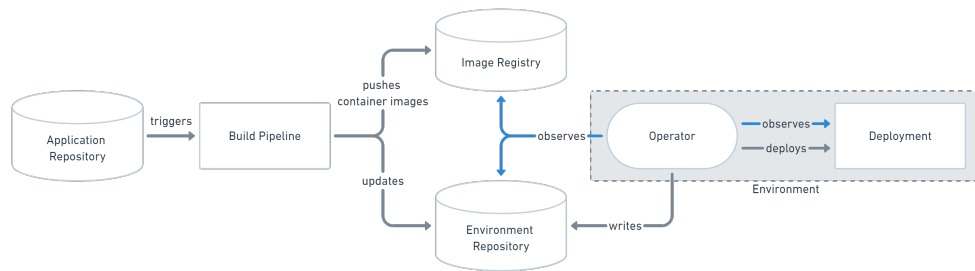


Figure 3.3: Pull-based pipeline [28]

Chapter 4

Argo CD

4.1 Introduction

ArgoCD is a declarative, GitOps continuous delivery tool used in Kubernetes. Application definitions, configurations, and environments should be declarative and version controlled. Application deployment and lifecycle management should be automated, auditable, and easy to understand [29].

ArgoCD fully embraces the GitOps approach, then, it is used alongside of Git repositories, as unique source of truth. As collateral effect, the user benefits from version control, which is an intrinsic Git feature.

Moreover, it is a cloud native tool, since it is designed to be a Kubernetes operator. It is a relatively new technology, since the first release was published on March 2018. The main goal of this technology is to automate the deployment process of applications.

4.2 How it works?

In a nutshell, if you want to deploy an application into the Kubernetes cluster, you can create an ArgoCD manifest that is a declarative yaml file. Inside it, You must specify the remote Git repository you want to connect. The repository might be a Github repository or a GitLab repository and so on, it is not important which one is used, as the the ArgoCD manifest only needs the https URL to perform the connection. Inside the repository, the k8s manifests can be specified in several ways, like kustomize application, helm charts, ksonnet applications, plain yaml/json files, ksonnet applications or a custom management tool. Another fundamental parameter that must be specified is the destination clusters, where the application is expected to be deployed.

The ArgoCD manifest can be deployed on kubernetes. It represents a Custom

Resource Definition and it is called Application. The Git repository connected to the Application, contains all the necessary yaml files that allow the application to be deployed within the k8s cluster.

Once the application is deployed, ArgoCD checks for any changes within the repository. If it finds updates, automatically it will keep the deployed application in the desired remote state.

For instance, if an user decides to increase the number of replicas of a specific kubernetes Deployment object, he can make it directly on the Git repository, committing and pushing the change in the specific yaml file. ArgoCD will notice inside the repository and converges the Deployment to the desired state, increasing the number of replicas.

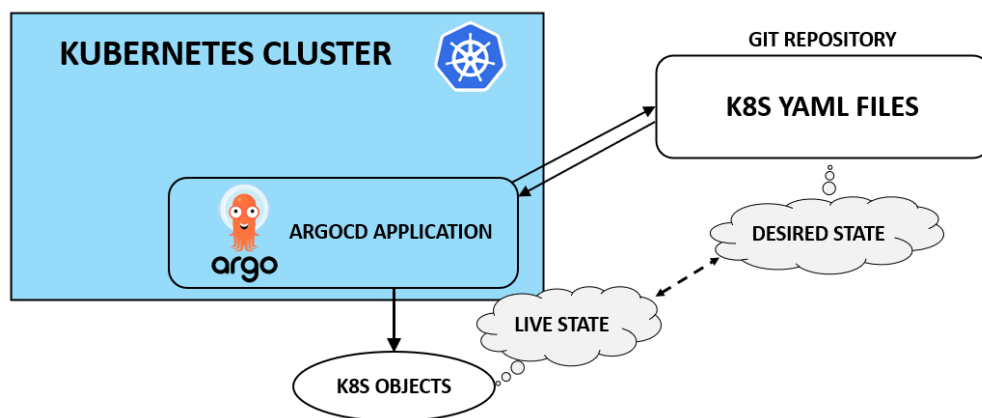


Figure 4.1: ArgoCD keeps the live state synchronised with the desired state

ArgoCD is implemented as a Kubernetes controller, which monitors running application and compares the current state against the desired target state, where the desired state is represented by yaml files in the Git repository. When a deployed application differs from the desired target state, it is defined as “OutOfSync”. On the other hand, when there are not differences, it is defined as “Sync”. The process of synchronization can be automatic or manual, the user can choose the preferred modality, for each Application (e.g. Application1 can be synchronized automatically and Application2 manually).

4.3 Why ArgoCD?

4.3.1 ArgoCD vs other solutions

Nowadays there are plenty of tools that can be used for generic CI/CD, like GitLab CI, GitHub Actions, Jenkins. But these tools originated before the coming of GitOps approach. GitOps was originally created to handle infrastructure files only through Git, by leveraging on the already existing IaC approach. When the Git repository changes, as a result of a commit, the live state should converge to the desired one. Then, deployment process is separated from the CI process, since it springs from a change in the Git repository and not necessarily from a natural consequence of CI steps.

ArgoCD, Flux and Jenkins X instead, follow the GitOps paradigm. They have common goals but also different features, since they are built to cover different use-cases [30].

FluxCD is a small and lightweight component. It is the simplest, without many features. It can run with very limited RBAC permissions. A Flux instance can only observe one repository, which makes the tool a bit awkward, if you want to manage a huge number of applications.

Jenkins X collects together a great number of tools to build a development workflow around repositories in GitHub. It can run Continuous Integration pipelines also. It manages deployments based on changes in Git repositories.

ArgoCD can manage deployments for multiple applications in different clusters. It runs with cluster-wide permissions in the cluster but also manages access and permissions for teams and projects. It has a intuitive web UI, very complete and useful. It can be used for monitoring, provisioning and other actions. Moreover, ArgoCD offers additional features that may be useful.

ArgoCD was chosen for several reasons:

- **Web UI:** the user interface is simple, expressive and allows users to monitor and interact with applications. The monitoring is in real time.
- **Multi-tenancy:** among all the solutions, it is the one that best manages multi-tenancy.
- Trivial installation and configuration.
- With one instance you can manage multiple Git repositories and deploy in multiple clusters, with minimal configuration effort.

	ArgoCD	Flux	Jenkins X
Sync desired state - live state	:)	:)	:)
Run on k8s natively	:)	:)	:)
Single instance in multicluster environment	:)	:(:
Watching multiple repositories	:)	:(:)
Multi-tenancy	:)	:(:
Mutli-cluster	:)	:	:
Native web UI	:)	:(:(
Kustomize support	:)	:)	:(
GitOps to maintain toll itself	:)	:(:)
Straightforward usability	:)	:)	:
CI	:(:(:)
CD	:)	:)	:)
Market Presence	:)	:	:

Table 4.1: Tools comparison

- All known companies that have implemented the GitOps approach have chosen ArgoCD, which is therefore the tool with the greatest market presence.

4.3.2 Why not a CI/CD pipeline?

Generally CI pipeline is triggered by a change in the source code. Once the CI steps are finished, the pipeline executes commands through kubectl, to apply the changes within the Kubernetes cluster. These are imperative commands to reach the desired state. Since one of the GitOps principles is the declarative approach to describing the infrastructures, in order to follow the IaC model, it is good that all the tools associated with GitOps, are declarative too.

Moreover, the CD steps can be removed from the pipeline, switching from a push based to a pull based pipeline. This goal can be achieved through an external tool, ArgoCD, which detects the drift in Git repository and manages CD autonomously.

Finally, ArgoCD provides out of the box an interesting series of features that a standard CI/CD pipeline cannot provide. On the other hand, ArgoCD only deals with CD. If you want to integrate CI also, another tool is needed for that purpose.

4.3.3 Features

ArgoCD tool has a multitude of features and customizable parameters. Here the most interesting:

	ArgoCD	GitHub Actions
Purely declarative	:)	:(
K8s native	:)	:(
Rollbacks	:)	:(
No exposure of credentials outside the cluster	:)	:(
Agnostic with respect to git-like repositories	:)	:(
Auto-sync if live state differs from the desired one	:)	:(
CI	:(:)
CD	:)	:)

Table 4.2: ArgoCD vs CI/CD tool.

- Once ArgoCD is installed in one k8s cluster, external clusters can be bound to it. Then, you can choose in which clusters to deploy the k8s objects that are in the Git repository pointed by the ArgoCD Application.
- Web UI is a really simple and intuitive, which allows you to do a lot of things, like creating or deleting an Application. Moreover it is useful to monitor applications in real-time (see figure 4.2).
- Thanks to Git, the history of the deployed infrastructure is maintained. It is possible to rollback to a previous state, using commits. It can be also done directly from the UI.
- RBAC and Multi-tenancy policies.
- Support for management and templating tools like Kustomize, Helm, Jsonnet, Ksonnet.
- Exposure of Prometheus metrics.
- A single instance of ArgoCD can handle many applications of different teams thanks to the Project CRD. A Project object can hold multiple Applications and this abstraction fits well with the team logic concept. Members of a team will only see the Projects assigned to them, then, only the applications associated with the Projects. This model is very similar to the k8s namespaces.

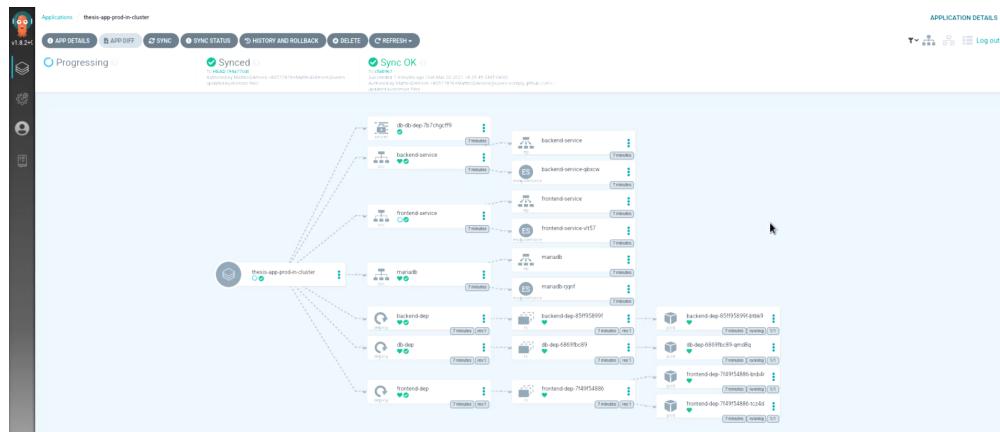


Figure 4.2: ArgoCD web UI example

4.3.4 ArgoCD Architecture

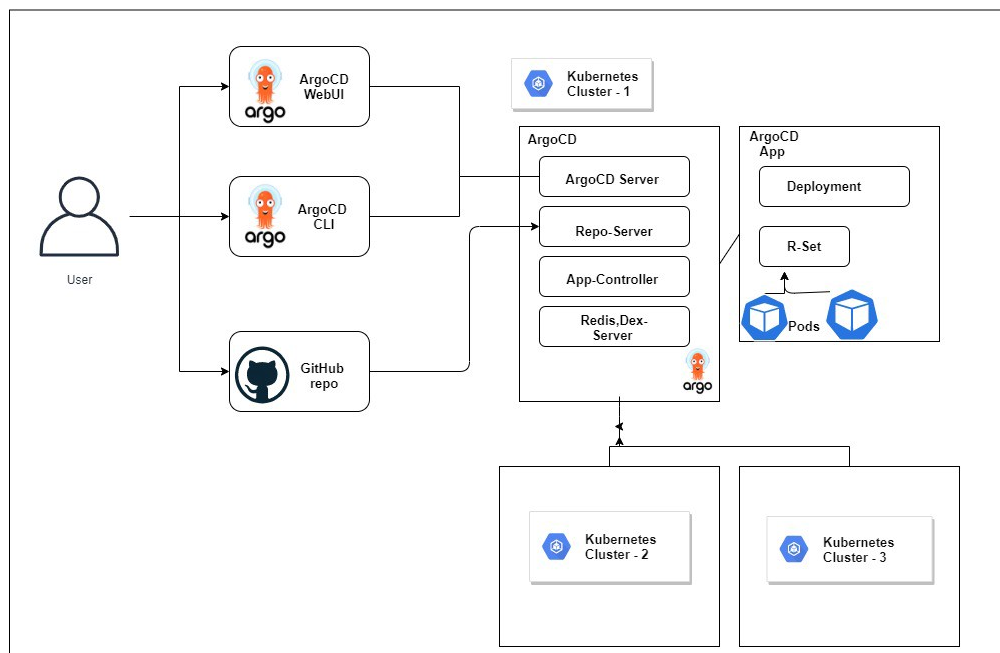


Figure 4.3: ArgoCD Architecture [31]

Once argoCD is deployed in the Kubernetes cluster, with its own namespace “argocd”, running:

```
$ kubectl get pods -n argocd
```

The output is:

NAME	READY	STATUS	RESTARTS
argocd-application-controller-0	1/1	Running	0
argocd-dex-server-748c65b578-cmtr4	1/1	Running	0
argocd-redis-6fb68d9df5-6m28h	1/1	Running	0
argocd-repo-server-64f4ddf469-td8nb	1/1	Running	0
argocd-server-846cf6844-h72v5	1/1	Running	0

ArgoCD Server

It is the pod that represent the API server, that is a gRPC/REST server. It exposes the API, that can be consumed by CLI, Web UI and CI/CD systems.

ArgoCD Repo Server

It is an internal service. It maintains a local cache of the Git repository, holding manifests that describe applications.

ArgoCD Application Controller

It is a Kubernetes controller, which given the deployed applications, compares the live state with the desired one. It also takes corrective actions in order to sync the live state.

ArgoCD Dex Server

It uses an in-memory database.

ArgoCD Redis

Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. In this case, it is pre-configured with the understanding of only three total redis servers/sentinels.

4.4 Application

The Application CRD is the Kubernetes resource object representing a deployed application instance in an k8s cluster.

```
1 apiVersion: argoproj.io/v1alpha1
2 kind: Application
3 metadata:
```

```
4   finalizers:
5     - resources-finalizer.argocd.argoproj.io
6   name: thesis-demo-prod-in-cluster
7   namespace: argocd
8 spec:
9   project: default
10  source:
11    path: kustomize/overlays/prod/
12    repoURL: https://github.com/owner/thesis-demo-infrastructure.git
13    targetRevision: HEAD
14  destination:
15    name: in-cluster
16    namespace: thesis-demo-prod
17  syncPolicy:
18    automated:
19      allowEmpty: true
20      prune: true
21      selfHeal: true
22    syncOptions:
23      - CreateNamespace=true
```

As it can be seen in the above manifest it is written in YAML language. This is an extended example, since the basic information that generally you write in the manifest are fewer. In any case, it is interesting to inspect the content of this Application manifest:

- **metadata.finalizers.[0].resources-finalizer.argocd.argoproj.io:** allows cascading deletion of resources. When this manifest is deleted from ArgoCD, all the associated resources are deleted automatically. Without this finalizer, only the app would be deleted, leaving the resources deployed on k8s.
- **metadata.name:** is the name of the Application inside an ArgoCD instance. It must be unique.
- **metadata.namespace:** must be the same namespace where ArgoCD has been installed, generally "argocd".
- **spec.project:** is the project the application belongs to.
- **spec.source.repoURL:** is the git repository URL, to which the application is bound to. It is the source of the application manifests.
- **spec.source.path:** is the relative path inside the repository where there are the application manifests.
- **spec.source.targetRevision:** is the symbolic reference (typically HEAD).

- **spec.destination.name**: is the cluster name where the application is to be deployed. Alternatively, it can be used *spec.destination.server*, which contains the cluster URL, but it is less intuitive (e.g. *https://kubernetes.default.svc*).
- **spec.destination.namespace**: is the namespace of the k8s resources when deployed.
- **syncPolicy.automated.prune**: by default, automated sync will not delete resources when Argo CD detects the resource is no longer defined in the Git repository. To prune the resources, a manual sync can always be performed, with pruning button checked. But if this key is set to true, the pruning mechanism is done automatically as part of the automated sync.
- **syncPolicy.automated.allowEmpty**: by default, automated sync with prune set to true have a protection from any automation/human errors when there are no target resources. It prevents application from having empty resources. To allow applications have empty resources, it must be set allowEmpty key to true.
- **syncPolicy.automated.selfHeal**: specifies if partial app sync should be executed when resources are changed only in target Kubernetes cluster and no git change detected. If, after the auto-sync, there is a difference between live and desired state, the live state is converged to the desired one. By default, it is set to false.
- **syncPolicy.syncOptions.[0].CreateNamespace**: ensures that namespace specified as the application destination exists in the destination cluster. If it does not exist, it is created.

The Application can be created with ArgoCD CLI, kubectl or through web UI.

```
1 $ kubectl apply -n argocd -f <filename or URL to k8s manifest for the app>
2
3 $ argocd app create APPNAME --file <filename or URL to k8s manifest for the app>
```

The same applies to cancellation.

```
1 $ kubectl delete -n argocd -f <filename or URL to k8s manifest for the app>
2
3 $ argocd app delete APPNAME
```

4.5 Project

The AppProject CRD is the Kubernetes resource object representing a logical grouping of applications. It is defined by the following key pieces of information:

- **sourceRepos:** is the reference to the repositories that applications within the project can pull manifests from.
- **destinations:** is the reference to clusters and namespaces that application within the project can deploy into.
- **roles:** list of entities with definitions of their access to resources within the project.

4.6 App of Apps

This is a pattern that allows an app to create other apps, which in turn can create other apps. This allows the user to declaratively manage a group of app that can be deployed and configured together. It can be useful if it is wanted to deploy multiple environment related to the same application.

The idea is to apply an Application resource manifest. This Application, which we call *X*, points to a relative path in a remote Git repository, which contains one or more Application manifests. Every Application manifest within this repository points in turn to other relative paths in other repositories. Therefore, running the *kubectl apply* command on *X*, automatically ArgoCD deploys all the other applications. When a manifest is added, removed or updated, within the repository pointed by *X*, ArgoCD takes care of adding, removing or updating the Application on the live state on Kubernetes.

4.7 High Availability

Argo CD is largely stateless, all data is persisted as Kubernetes objects, which in turn is stored in Kubernetes' etcd. Redis is only used as a throw-away cache and can be lost. When lost, it will be rebuilt without loss of service. A set HA of manifests are provided for users who wish to run Argo CD in a highly available manner. This runs more containers, and run Redis in HA mode.

4.8 Disaster Recovery

You can use *argocd-util* to import and export all Argo CD data.

Chapter 5

Solution

5.1 Introduction

DevOps was conceived to answer to the problem of co-operation between developer and operations teams. Thanks to DevOps, a lot of things changed in better, but there are still improvements that can be made.

In a company, when a software project is developed in a hybrid cloud environment, there is still the issue of provisioning and managing of infrastructures. If a developer requires an environment on Kubernetes to develop a new feature, he has to ask an employee of the operations team to do it. From the moment the request is made, several days or even weeks may pass before the environment is provisioned. It happens because of the necessary approval steps.

Then, the operations employee schedules the commitment and when the time comes, additional time is needed to do the provisioning. Generally, it is not an automated process, then it requires time to be done.

After provisioning, the environment should require modifications, i.e. infrastructure parameters, new version of Docker images, different clusters where to deploy the environment. Also this task requires time. It would be nice if the developer could manage this process more autonomously.

The starting point toward this goal is the GitOps approach, because it is developer-centric. Shortly, GitOps could be seen as a union of Git tool (well-known by developers) and DevOps practices. It has been chosen GitOps also for its cloud native purpose, because hybrid cloud is increasingly becoming common as a business solution.

Moreover, there is a need for a company to have an improved approach to the infrastructural management of Kubernetes environments. Also in this case, GitOps may be the answer, since it is focused on the infrastructure automation and maintenance.

5.2 Business Case

The business case concerns a company that wants to develop and maintain a full stack application. The full stack application is composed by the *backend* and *frontend*. The company wants to leverage a hybrid cloud environment, with an on-premise cluster and a cluster of a cloud service provider, in order to have more flexibility, redundancy and deployment options. These are not stringent constraints, since other operational choices could be made (e.g. multiple public cloud cluster that belongs to different cloud service providers).

When developing a project, you want to add new features, testing new releases and update the production environment.

The idea is to give developers a strong independence from the operations team. A developer should create, update and delete Kubernetes environments on demand, automatically, modifying them as needed.

The problem is that the operations team can't give to developers the access rights to operate freely on infrastructure. Therefore, a solution must be developed that allows the operations team to provide developers with a secure and controlled tool to access and interact with infrastructure repositories. During the thesis work, a possible solution was developed and proposed.

5.3 Branching strategy

5.3.1 Introduction

If a company starts to work on a new software project and it decides to rely on Git, one of the very first steps is to define a valid and consistent branching strategy. In Git, a branch represents a pointer to a commit. It is a useful abstraction for the edit, stage and commit processes.

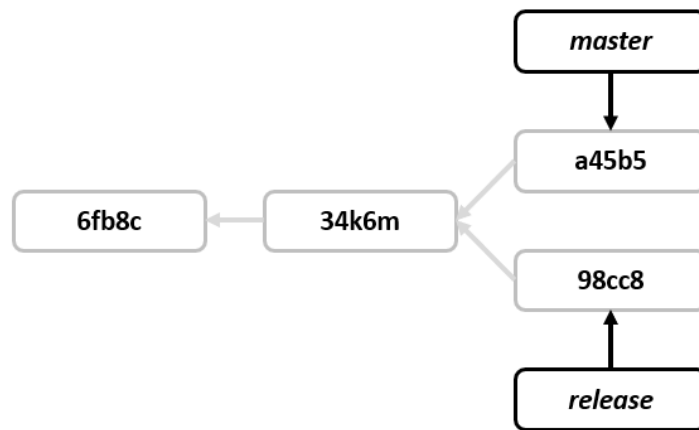


Figure 5.1: *master* and *release* branches point to two different commits

When a Git project is created, a default branch must be defined, typically called *master*. After creating the *master* branch, multiple other branches may be created, with a unique name inside the project. The branching strategy is nothing more than a structured solution of branches managing.

5.3.2 *Git Flow* branching strategy

Being that the thesis work focuses on the use case of developing a full stack application, consisting of *frontend* and *backend*, the *Git Flow* branching strategy has been chosen [32]. Although the thesis work is predominantly inspired by this model, some minor modifications and variations have been introduced, as they were considered appropriate.

This strategy was created by Vincent Driessen in 2010. These are the branches involved:

- *master*;

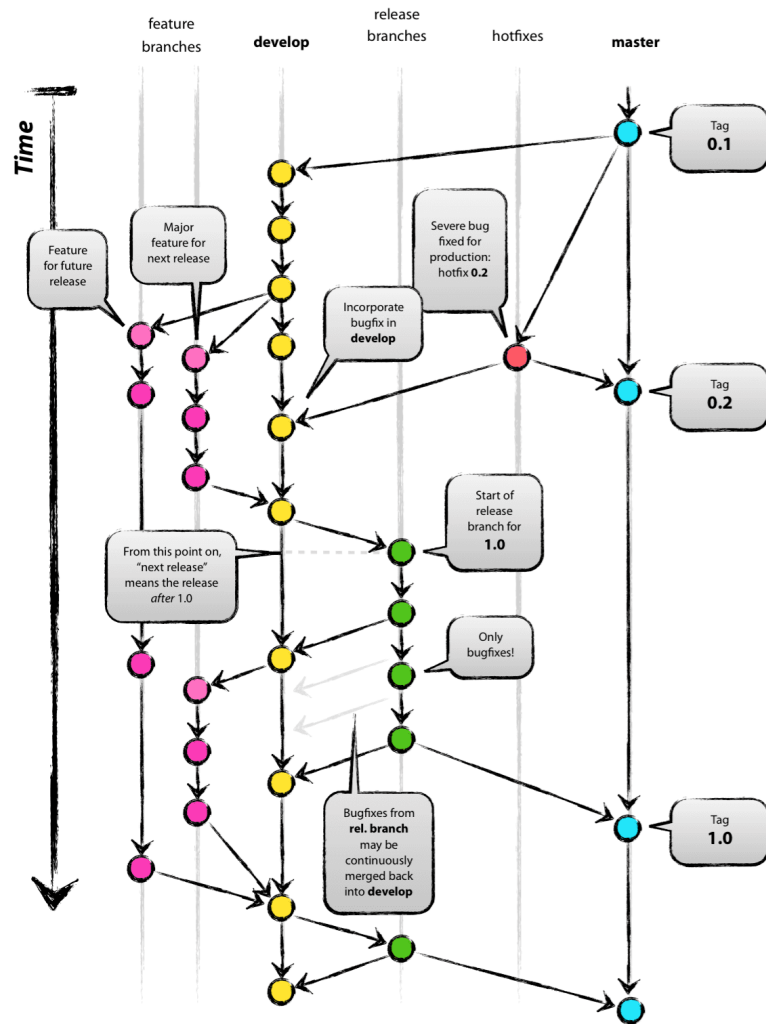


Figure 5.2: Branching strategy [33]

- *develop*;
- *features/***;
- *releases/***;
- *hotfix/***;

5.3.3 Branches

There are only two branches created at the beginning of the repository creation and with an infinite lifetime: *master* and *develop*. The *master* branch contains the

source code that reflects the production-ready state.

The *develop* branch is the "operative" branch. From that branch, features and releases branch are forked. When it reaches a stable point, after that one or more feature branches have been merged with *develop*, a release branch can be branched off.

Feature, *release* and *hotfix* branches have a limited lifetime, being that once they are terminated and merged back to the right branch, as a result of a pull request, they are deleted.

Each one of these branches have a specific purpose and is bound to strict rules, such as from which branches it can originate or into which branches it must be merged back.

Feature branches

- This kind of branches are branched off from the develop branch.
- Once they are completed, they are merged back into the develop branch.
- Conventionally, the branch naming convention is *features/*** (e.g. *features/firstFeature*, *features/myFeature*). ***** can be replaced by any other value except */*.

Feature branches are used to develop new features that will be included in the next release. When the development of a feature starts, the target release in which this feature will be incorporated may be known or already unknown. Substantially, a feature branch exists as long as the feature is in development, but after it is merged back into the develop branch, it is deleted.

A developer can pull that branch from remote and work on it locally.

But when he wants to test progress, he must push commits, in order to trigger the pipeline, which generate the new Docker image and update the k8s environment. The branch creation can be made locally or remotely.

If it is done locally and the developer wants the k8s environment to be created, he must push the feature branch to *origin*, to trigger the pipeline.

```
# Locally creation of a feature branch
$ git checkout -b features/myFeature develop
```

A feature branch can only be merged back to the develop branch by means of a pull request.

Release branches

- This kind of branches are branched off from the develop branch.
- Once they are completed, they are merged back into the master branch. When the merge is done, the "master" branch must be merged to the "develop" branch, to ensure that they remain aligned.
- Conventionally, the branch naming convention is *releases/*** (e.g. *releases/first-release*, *features/release1*). ***** can be replaced with any other value except */*.

Release branches support the preparation of a new production release. The release is tested to see that everything works properly, before opening the pull request.

It allows for minor bug fixes, since it is expected that all new implemented features have already been tested. But, if a release incorporates several feature branches, it can be tested whether they, as a whole, work as expected. The key moment to branch off a new release branch from develop is when the develop branch reflects what you would like to have in the new release.

The release branch exists until it is merged back into the master branch.

Unlike feature branches, a release branch must be created directly on the remote repository, since everyone must know of its existence. Moreover, also in this case, it is necessary to create the environment on k8s, in order to start tests and the environment is created at the end of the workflows, so it must stay in the remote repository. From this point, a developer can pull locally that branch in order to fix bugs. A wise use of *issue* must be made, in order to avoid more than one developer fixing the same bugs. After the changes have been made, the developer must remotely push the changes, to update the Kubernetes environment and to continue to test it. When the release is ready, the pull request can be opened. After proper reviews, the pull request is merged to the master branch. Since the develop branch must reflect the master one, develop branch is rebased with master branch.

HotFix branches

- This kind of branches are branched off from the master branch.
- Once they are completed, they are merged back into the master branch.
- Conventionally, the branch naming convention is *hotfix/*** (e.g. *hotfix/fast-hotfix*, *features/hotfix1*). ***** can be replaced with any other value except */*.

Hotfix branches are very much like release branches, since they are also meant to prepare for a new production release, albeit unplanned. They arise from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

When the hotfix is resolved, a pull request is opened and the hotfix branch is merged back to "master" branch. Now, there are two different scenarios:

- If a release branch exists, it is rebased with the master branch.
- If a release branch does not exist, the change is rebased on the develop branch.

Tag Strategy

The master branch, which reflects the production state, must be tagged properly with version numbers. The tag is composed by a major and a minor numbers and it is written in that form $vX.Y$ (e.g. $v2.4$). The tag approach is as follows:

- When the master branch is created, there is no version.
- At the end of the first release, the master branch is tagged with $v0.1$.

After the first release:

- When a new release is merged, the major number increases by one and the minor number is set to 0 (e.g. from $v1.1$ to $v2.0$).
- When a new hotfix is merged, the major number remains the same, whereas the minor grows by 1 (e.g. from $v1.1$ to $v1.2$).

5.4 *input* file

As it has been said at the beginning of this chapter, developers should be more free to operate on infrastructure files.

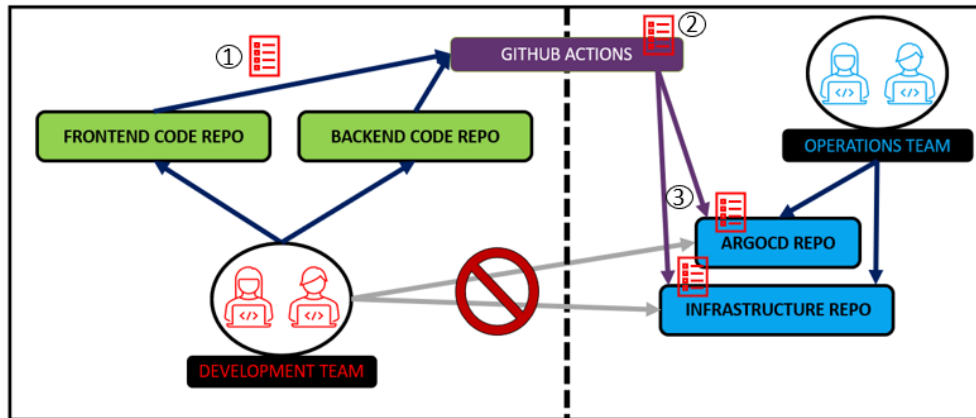


Figure 5.3: How *input* file is applied

The *input* file is written in YAML language and is the controlled access point, which allows developers (but not only them) to edit infrastructural YAML files. We will look in detail at all the fields that can be inserted into this file.

5.4.1 *backend*, *frontend* and *db* fields

```

1 backend:
2   - replicas: 1
3     type: deployment
4   - port: 8080
5     type: service
6 frontend:
7   - replicas: 1
8     type: deployment
9   - port: 80
10    type: service
11 db:
12   - type: deployment
13     secrets:
14       - MYSQL_ROOT_PASSWORD: password
15       - MYSQL_DATABASE: mydb

```

These fields are optional and contain a list of object. They allow an user to add or override some parameters which are inside the backend, frontend or db

YAML files. This customization can be done either at the provisioning time of the environment or later, once it has been provisioned. The updates are propagated every time the *input* file is modified and a developer pushes the changes to the code repository.

Moreover, it is possible to inject also one or more secrets or configMaps into the Deployment YAML file. Every object which belong to the list must include the "type" field, related to the k8s object (i.e. deployment or service). For instance, if a developer wants to update the port number of the frontend pod, he must insert into the *input* file:

- frontend.type : "deployment"
- frontend.port : "80"

All the fields must be written in lower case. Some of the possible values that can be inserted within the *input* file are as follows ("*" can assume "backend", "frontend" or "db" as possible values):

1. Deployment:

- **.replicas*: it refers to *spec.replicas*.
- **.port*: it refers to *spec.template.spec.containers.[0].ports.[0].containerPort*. In the solution developed, the *one-container-per-Pod* was considered, which is the most common use case.
- **.secrets*: this field contains a list of secrets, composed by *Secret-Name: Value*. Values must be written in plain text and not base64 encoded. Once a secret is written in the *input* file, automatically it is inserted within the Deployment YAML file as environment variable. To do this, the *secretGenerator* is used, which is a field in the *kustomization* file that allows secrets to be generated.

2. Service:

- **.port*: it refers to *spec.ports*.

5.4.2 image field

```
1 image:
2   frontend:
3     tag: custom-tag1
4   backend:
5     tag: custom-tag2
```

The *image* field allows users to update the image tag for the *frontend* and the *backend*. It is an optional field, so it may be completely or partially omitted (the tag can only be specified for the frontend or backend image). If wrong tag names are inserted, the Kubernetes environment will not be generated correctly and will present errors, which will be reported by ArgoCD. In this case, it is up to someone to fix this problem.

- *image.backend.tag*: the tag value for the *backend* image.
- *image.frontend.tag*: the tag value for the *frontend* image.

In order to understand the logic behind *image*, here are the general rules:

1. Push on the backend repository:

(a) push on *master* branch:

- i. if there is already a production environment deployed, *image.frontend.tag* is neglected and it is used the image tag related to the last stable image, i.e. the image that is currently in production. *image.backend.tag* is neglected and the latest build image is used, i.e. the one generated as a result of the current push.
- ii. if there is not a production environment deployed, it is important to specify an existing frontend image tag.

(b) push on *features/*** or *releases/*** branches:

- i. if there is already a production environment deployed and *image.frontend.tag* is specified, it is used this tag for the frontend image. If instead *image.frontend.tag* is not specified, it is used the image tag related to the last stable image, i.e. the image that is currently in production. *image.backend.tag* is neglected and the latest build image is used, i.e. the one generated as a result of the current push.
- ii. if there is not a production environment deployed, *image.frontend.tag* must be specified necessarily and it must be an existing frontend image tag. *image.backend.tag* is neglected and the latest build image is used, i.e. the one generated as a result of the current push.

2. **Push on the frontend repository:** the same of above but must be inverted frontend and backend names.

The problem of the production environment

If there is a feature branch called *features/f1* on the frontend side and there is a feature branch called in the same way on the backend side, it does not generate inconsistencies. Two separate environments will be generated on k8s, mutually independent.

It is not the same for the production environment. The master branch on frontend and backend repositories converges to the same k8s environment. Then, it has been chosen to use a flag value into the *config* file within the infrastructure repository, called *prod-input-master*. This flag can be equals to *backend* or *frontend* and if it is omitted, the default value is *backend*.

Suppose that the *prod-input-master* is equals to *backend*. In this case, when changes are pushed on the master branch into the frontend repository, the *input* file will be ignored. Then, there is only one repository that can update the production infrastructure environment. In any case, the images are correctly updated both for the frontend and backend.

5.4.3 branch

The *branch* value has been inserted only for security and consistency reasons. There are generally two situations where the *input* file in a branch is incorrect.

For instance, when a feature branch is merged back to the develop branch, the *input* file is overwritten. Because of this, the develop branch now contains the *input* file related to the feature branch. When it happens that a release branch is generated from develop, some workflows start automatically in order to provision the new environment on k8s. But this is not correct, because the release branch contains the *input* file related to the old feature branch. In order to break this automatic mechanism, the workflow which starts every time a release branch is created, checks the value of the *branch* field.

If this value is different from the actual name of the branch where the *input* file lies, the workflow fails.

Another example is when a release branch is merged to the master branch, after a pull request. In this other case, the master branch, which corresponds to the production state, contains the *input* file related to the release branch. This is the default behaviour of the solution. When it happens, to make everything work again, there are multiple options:

- Leave the *input* file unchanged and update only the *branch* value properly;
- Update manually the entire *input* file;
- Replace the *input* file with a new one, which is a stable version for that type of branch, just changing the *branch* value. This file can be stored into a different

repository, designed to contain this kind of backup file.

There is another option, designed to have more automation but less customization. This option uses an additional repository, a *backup-files repository*. This repository, contains a default *input* file for the production, releases and features branches.

It is useful to have this repository because it can be used also as a guide for inexperienced developers. In the *config* file contained in the code repositories, there are three flags that can be set to *true* or *false* (false by default):

- *default-input-prod*: related to the master branch.
- *default-input-release*: related to the release branch.
- *default-input-feature*: related to the feature branch.

When these values are set to true, the *input* file inside the branch is never considered but the one in the backup repository. As it can be seen, this option is more automation-oriented. This could be useful especially for the production environment, which is the most critical. So, *default-input-prod* could be set to true and the others to false. Anyway, there is freedom of choice.

5.4.4 clusters

The *clusters* field contains a list of names. These names corresponds to the cluster names, from the internal point of view of ArgoCD. ArgoCD is installed on a specific cluster, but it can connect to many others. Each cluster is recognised by a name. For instance, there are two different clusters. The one where ArgoCD is installed is called "in-cluster" (the default name given by ArgoCD) and the other "second-cluster". If a developer wants to deploy an environment on both clusters, the *input* file will contain:

```
1 clusters:
2   - in-cluster
3   - second-cluster
```

If he wishes to remove the environment from the "in-cluster" after the provisioning, he must remove "in-cluster" from the list associated to the "clusters" field, then push changes, in order to propagate them.

5.5 *config* file

It is a YAML file that must be added to all repositories except the backup repository. It is used to set necessary parameters about the solution configuration.

In the code repositories

It is the same for the backend and frontend repositories. The allowed fields are as follows:

- **docker-backend-repo**: this is a mandatory field and its value is the name of the repository where the backend Docker images are stored (e.g. *myAppName-backend*).
- **docker-frontend-repo**: this is a mandatory field and its value is the name of the repository where the frontend Docker images are stored (e.g. *myAppName-frontend*).
- **infrastructure-repo**: this is a mandatory field and its value is the name of infrastructure repository.
- **backup-input-repo**: this is a mandatory field its value is the name of the backup-files repository.
- **tier**: this is mandatory field and its value identifies whether the code repository is backend or frontend.
- **default-input-prod**: this is an optional field and can be true or false. The default value is false.
- **default-input-release**: this is an optional field and can be true or false. The default value is false.
- **default-input-feature**: this is an optional field and can be true or false. The default value is false.

In the infrastructure repository

The allowed fields are as follows:

- **app-name**: this is a mandatory field and its value is the name of the application managed and associated with Kustomize.
- **argocd-repo**: this is a mandatory field and its value is the name of the ArgoCD repository.
- **prod-input-master**: this is an optional value that can take either *backend* or *frontend* as values. The default value is *backend*.

In the ArgoCD repository

The allowed fields are as follows:

- **source-repo-url**: this is a mandatory field and its value is the https URL of the infrastructure repository. This value is needed to allow ArgoCD to look at the remote repository, which represents the desired state of the Kubernetes environments.

5.6 Repositories

There must be remote repositories for both code and infrastructure management. The choice of git-like web tool to use is not binding. For this reason, it was decided to use GitHub, since it is the most common and widely used tool. Because of this choice, GitHub Actions implements the CI pipeline. The development team has direct access to the code repositories and does not have direct access to infrastructure repositories, which are managed by the operations team.

This is done because of the role separation. As will be seen later, the development team can access the infrastructure repositories with the help of GitHub Actions and the *input* file.

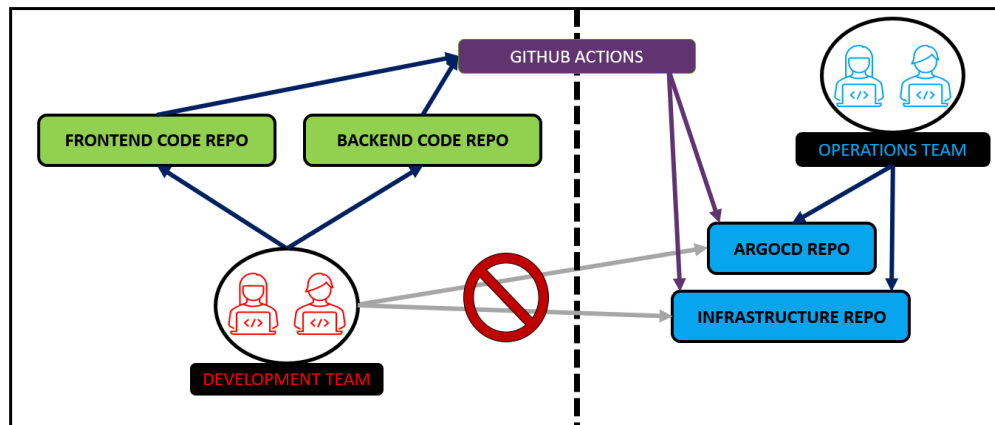


Figure 5.4: Repositories and teams

5.6.1 Code Repositories

The applications that can be developed in this thesis solution are those formed by a frontend and a backend. Two separate repositories have been created, one for the frontend and the other for the backend.

These repositories contains:

- The source code and related files;
- Dockerfile used to build the Docker image;
- The script directory, which contains Python scripts used in conjunction with the workflows;
- The config.yaml file, used for configuration purpose;
- The input.yaml file, used to customize the Kubernetes environment.
- The workflows directory, which contains the GitHub Action workflows.

Moreover, also some secrets must be set:

- **DOCKERHUB-USERNAME**: the owner name of DockerHub repositories which contain Docker images for frontend and backend.
- **DOCKERHUB-TOKEN**: the access token to push new Docker images on DockerHub.
- **PAT-TOKEN**: a GitHub token. It must be in all repositories that have workflows which interact with other ones.

Only the developer team has access rights to these repositories. The choice of keeping the two repositories separate instead of having only one has been made for many reasons:

- Two separate sets of pipeline. It is more difficult to identify if only frontend or backend source code has been changed, as a result of a Git push. As a consequence, it is harder to choose which CI pipelines to trigger.
- Some developers can only interface with a repository (e.g. two developers are frontend specialists). It enables a better independence and a cleaner separation.
- If the project becomes very large, it is more difficult to manage a single repository.

5.6.2 Infrastructure Repository

The purpose of the infrastructure repository is primarily to contain the infrastructure files used to deploy environments on Kubernetes. These files are handled through Kustomize, following the principle of base and overlays structure. This repository contains:

- The *kustomize* directory;
- The *script* directory, which contains Python scripts used in conjunction with the workflows;
- The *config.yaml* file, used for configuration purpose;
- The *workflows* directory, which contains the GitHub Action workflows.

Moreover it must also have set the secrets **DOCKER-USERNAME** and **PAT-TOKEN**, which is a GitHub token. It must be in all repositories that have workflows which interact with other ones.

This repository is designed to be dedicated to a single application.

kustomize directory

The *kustomize* directory contains all the files needed to deploy environments on Kubernetes. In the figure 5.5 is shown the basic directory structure. Each leaf directory must contain the *kustomization* file, to activate the Kustomize mechanisms. The *base* directory contains all the basic YAML files which describe the application. These files could be Deployment and Service files for backend, frontend and database. It is not important that these are completely filled in, as the *input* file will add multiple fields, such as secrets, replicas, ports, namespaces, etc. But for the solution to work, some fields must be specified:

- *kind*;
- *metadata.name*;
- *metadata.labels.app*;
- *metadata.labels.tier*.

All the above fields are essential to identify the files and customise them.

In any case, it is important to say that not only these fields should be specified, because the solution takes care of a finite number of fields (in the current version). So, in general, the files within the *base* directory should contain a semi-complete template of how the application should be structured (e.g. the type of a Service, the selector of a Service, etc.), because the *input* file is focused on customization and not on creating the environment from scratch. The *overlays* directory contains the structure for defining new environments to be deployed. There is the *prod* directory which is associated with the production environment, unique in the application (there is no difference between frontend and backend).

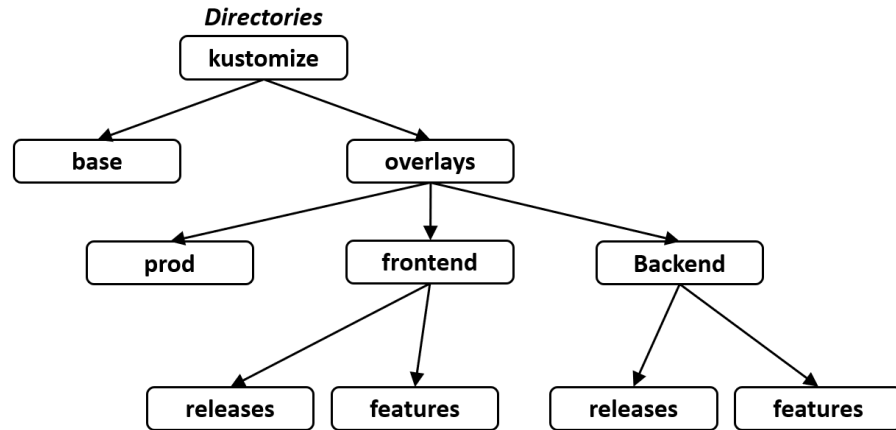


Figure 5.5: kustomize directory structure

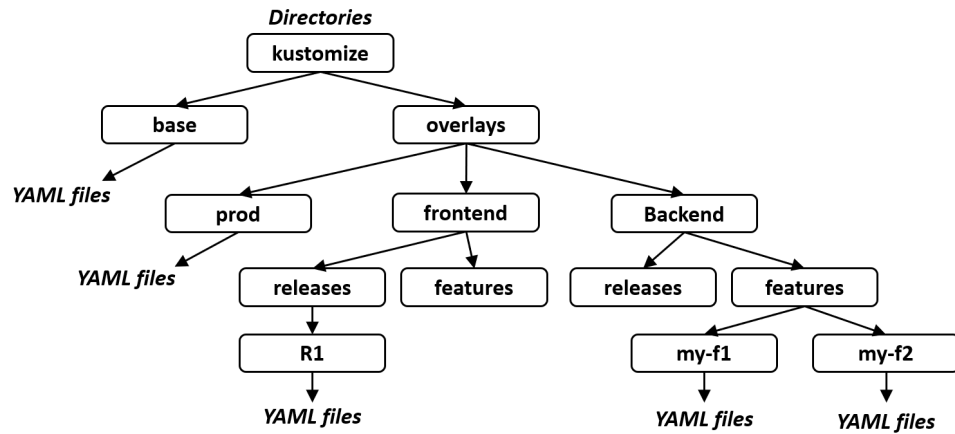


Figure 5.6: kustomize example

The *backend* and *frontend* directories contain directories associated to the environments related to the two portion of the application. Specifically, they can contain either feature or release environments.

In the figure 5.6 there is an example of a possible situation. There are four different directories inside *overlays*, which are generated and filled automatically and they describe four environments:

- *prod*: it is associated with the production environment (*master* branch);
- *R1*: it is associated with the frontend release environment (*releases/R1* branch);

- *my-f1*: it is associated with a backend feature environment (*features/f1* branch);
- *my-f2*: it is associated with a backend feature environment (*features/f2* branch).

It is important to remember that for each leaf directory within *overlays*, there is a branch in one of the two code repositories (or just one, in case of *prod*). In addition, users only has to manually create the *kusutomize/base* directories and fill *base* with YAML files. Everything else is generate automatically with a Python script, if one or more directories are still missing.

5.6.3 ArgoCD Repository

This repository is used to contain the YAML manifests used by ArgoCD to deploy the k8s environments.

This repository contains:

- The manifests directory;
- The script directory, which contains Python scripts used in conjunction with the workflows;
- The config.yaml file, used for configuration purpose;
- The workflows directory, which contains the GitHub Action workflows.

Moreover, it must also have set the secret **PAT-TOKEN**, which is a GitHub token. It must be in all repositories that have workflows which interact with other ones.

manifests directory

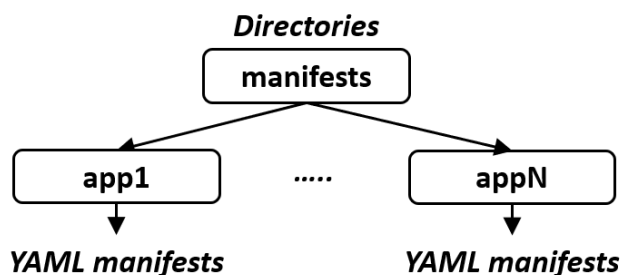


Figure 5.7: Manifests directory

This repository is designed to be shared across different application, and for this reason in the figure 5.7 is shown that *manifests* hosts multiple directories, one per application.

Every manifest is associated to an environment in a specific cluster (this means that the same environment, in order to be deployed over two clusters, must have associated two manifests).

These manifests are generated automatically, based on *input* file, *config* files and the branch name associated with the environment. Each manifest has a unique name, which has the following format:

- `"prod"+"-"+"<CLUSTER-NAME>":` for the production environment;
- `"<tier>"+ "-" + "<BRANCH-NAME>"+ "-" + "<CLUSTER-NAME>":` for the other environments.

An example of manifest is the following:

```
1  apiVersion: argoproj.io/v1alpha1
2  kind: Application
3  metadata:
4    finalizers:
5      - resources-finalizer.argocd.argoproj.io
6    name: thesis-demo-prod-in-cluster
7    namespace: argocd
8  spec:
9    destination:
10     name: in-cluster
11     namespace: thesis-demo-prod
12    project: default
13    source:
14     path: kustomize/overlays/prod/
15     repoURL: https://github.com/owner/thesis-demo-infrastructure.git
16     targetRevision: HEAD
17    syncPolicy:
18     automated:
19       allowEmpty: true
20       prune: true
21       selfHeal: true
22     syncOptions:
23       - CreateNamespace=true
```

***anchor* file**

Every directory within *manifests* contains an *anchor* file, which is an empty file that only keeps the folder visible. In fact, GitHub does not show empty folders. This is

a problem when an application does not have temporarily active environments and ArgoCD has an Application CRD which points to that directory.

5.6.4 Image Repositories

The image repository is the place where are stored multiple versions of a Docker image. There is a repository for the frontend and another for the backend images. It has been adopted DockerHub as registry, but the solution can work also with other public or private registries. Due to limitations on private repositories (only one can be private with the free version of DockerHub), it was decided to make the repositories public. In order to have only 2 repositories, the image name is built in this way: `<DOCKER-OWNER>/<DOCKER-REPO>:<COMMIT-SHA>-<BRANCH-ID>`. Where,

- **DOCKER-REPO** can be the backend or frontend repository;
- **COMMIT-SHA** is the SHA value associated to the last pushed commit, which has triggered the workflow;
- **BRANCH-ID** can assume *prod*, *features-<NAME-FEATURE>* or *releases-<NAME-RELEASE>* as values (e.g. if there is a branch called *releases/my-release*, the tag name will be `<COMMIT-SHA>-releases-my-release`).

To be more precise, each time a new version of the image is built, the tag with *latest* is also pushed on DockerHub. This tag is overwritten every time.

5.6.5 Backup-files Repositories

This repository is responsible for containing backups of some files, such as *input* files. There are no constraints on what it can contain. Certainly, for the purpose of this thesis, it must contain:

- `<app-name>` directory: this repository contains a directory for every application handled by the company. Inside this folder, there must necessarily be:
 1. *main-app-manifests*: this directory contains the ArgoCD manifest that must be deployed manually in Kubernetes to activate the *app of apps* pattern. It points to the *manifests/<app-name>* folder within the ArgoCD repository;
 2. *feature, release and prod directories*: these directories contains the three *input* files, one for the production environment, one for the release environment and one for the feature environment.

They serve as templates on which to base new *input* files and are used if the *default-input* flags within the *config* file in the code repositories are set to true.

5.7 Workflows

The pipelines used to implement the Continuous Integration are those related to GitHub, i.e. GitHub Actions and they are called *workflows*. Now, they will be analysed one by one, repository by repository. In the following sections the steps of each workflow will be analysed, focusing on the general logic and not on each technical aspect.

5.7.1 Code Repositories

The workflows are the same for the frontend and backend repositories.

Workflow Push

This workflow is triggered when there is a push on the *master*, *features/*** or *releases/*** branch, where ***** means any name (e.g. *releases/my-release*, *features/f1*).

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Check the existence of the *Dockerfile*, *input* and *config* files:** if at least one of these file does not exist within the repository, the workflow is interrupted.
3. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.
4. **Validation of the *config* file:** a Python script processes the *config* file, checking that it contains all the necessary fields and valid values for every field.
5. **Creation of variable from the *config* file:** a Python script extract values from the *config* file.
6. **Build and push of Docker images:** the same image is built and pushed on the DockerHub repository, with two different tags (see 5.6.5).
7. **Trigger the Workflow Update (see 5.7.2) of the infrastructure repository:** the next workflow is triggered using the *workflow-dispatch* action. Input values are passed to the next workflow. In order to be able to do this, it is necessary to set the same *PAT-TOKEN* within the repositories.

Workflow Delete

This workflow is triggered when a *feature/*** or *releases/*** branch is deleted.

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Check the existence of the *config* file:** if this file does not exist within the repository, the workflow is interrupted.
3. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.
4. **Validation of the *config* file:** a Python script processes the *config* file, checking that it contains all the necessary fields and valid values for every field.
5. **Creation of variable from the *config* file:** a Python script extract values from the *config* file.
6. **Trigger the Workflow Delete (see 5.7.2) of the infrastructure repository:** the next workflow is triggered using the *workflow-dispatch* action. Input values are passed to the next workflow. In order to be able to do this, it is necessary to set the same *PAT-TOKEN* within the repositories.

Workflow Rebase

This workflow is triggered when a Pull Request is merged, and it happens only when a release branch is merged back to master, a hotfix branch is merged back to master or a feature branch is merged back to develop. The last case is not interesting, as this workflow is intended to keep the develop branch synchronised with the master branch, using the Git *rebase* command. The workflow splits into two, depending on which branch has been merged with the pull request. If a hotfix branch is merged back to master:

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.
3. **Extract branch name:** If a release branch exists, the rebase of the master must be done on that branch. Otherwise, it must be done on the develop branch. In order to choose the right branch, a Python script is run.

4. **Rebase:** it is done the command *git rebase master* on the correct branch.
5. **Save changes:** it is made a push to save changes.

If a release branch is merged back to master:

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Rebase:** in this case, there are no choices, since the rebase action must be done on the develop branch.
3. **Save changes:** it is made a push to save changes.

Workflow Update-Tag

This workflow is triggered when a hotfix or release branch is merged back to master, as a consequence of a Pull Request. The purpose is to update the tag of the master. The master tag is in the form *vX.Y* and the tag mechanism is explained in section 5.3.3.

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.
3. **Generation of the new tag:** a Python script generate the new tag, depending on the branch that has been merged back to the master branch (see 5.3.3).
4. **Save changes:** the tag is pushed and then saved.

5.7.2 Infrastructure Repository

Workflow Update

This workflow is triggered by the Workflow Push of the code repository (see 5.7.1). It takes as inputs the following values, passed by the Workflow Push:

- **Docker image tag;**
- **Code branch name;**
- **Code repository name;**
- **Tier:** it can assume *frontend* or *backend* as values;

- **Docker frontend repository:** the repository name on DockerHub for the frontend images;
- **Docker backend repository:** the repository name on DockerHub for the backend images;
- **Default input flag:** it can be *true* or *false*. If it is true, the *input* file inside the backup-files repository is used, otherwise the one within the code repository, in the proper branch;
- **Backup-files Repository:** if the default input flag is set to true, the *input* file will be fetched from this repository.

The steps of the workflow are as follows:

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Check the existence of the *config* file:** if this file does not exist within the repository, the workflow is interrupted.
3. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.
4. **Validation of the *config* file:** a Python script processes the *config* file, checking that it contains all the necessary fields and valid values for every field.
5. **Creation of variable from the *config* file:** a Python script extract values from the *config* file.
6. **If Default Input Flag is equal to true:**
 - (a) **Checkout backup-files repository:** it is done in order to fetch the *input* file.
7. **If Default Input Flag is equal to false:**
 - (a) **Checkout code repository:** it is checked out the code repository in order to fetch the *input* file.
8. **Creation of the directory structure within *overlays*:** since this workflow is triggered as a consequence of creating or updating a *master*, *features/*** or *releases/*** branch, a Python script is executed in order to generate the leaf directory associated with one of these branch.
Moreover, the script also adds the *kustomization* file and fills it with some

basic values. If the directory associated to the environment already exists, it means that the environment has been already provisioned. In this case, the directory is emptied and the *kustomization* file is recreated.

This happens because the infrastructure changes as a result of the push may be multiple, so rebuilding allows a leaner script to be written, without losing speed of execution.

9. **Customization of the environment:** once the directory has been created, the *input* file is parsed by a Python script, in order to apply customisations.
10. **Update the docker image tag:** a Python scripts parses the *input* file again, to update the frontend and backend image tags to be used for this environment.
11. **Save changes:** the changes within the infrastructure repository are saved with the *git commit* and *git push*.
12. **Trigger the Workflow Update (see 5.7.3) of the ArgoCD repository:** the next workflow is triggered using the *workflow-dispatch* action. Input values are passed to the next workflow. In order to be able to do this, it is necessary to set the same *PAT-TOKEN* within the repositories.

Workflow Delete

This workflow is triggered by the Workflow Delete of the code repository (see 5.7.1). It takes as inputs the following values, passed by the Workflow Push:

- *Code Branch Name*;
- *Tier*: it can assume *frontend* or *backend* as values.

The steps of the workflow are as follows:

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Check the existence of the *config* file:** if this file does not exist within the repository, the workflow is interrupted.
3. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.
4. **Validation of the *config* file:** a Python script processes the *config* file, checking that it contains all the necessary fields and valid values for every field.

5. **Creation of variable from the *config* file:** a Python script extract values from the *config* file.
6. **Delete directory:** a Python script deletes the directory associated with the branch that has been delete.
7. **Save changes:** the changes within the infrastructure repository are saved with the *git commit* and *git push*.
8. **Trigger the Workflow Delete (see 5.7.3) of the ArgoCD repository:** the next workflow is triggered using the *workflow-dispatch* action. Input values are passed to the next workflow. In order to be able to do this, it is necessary to set the same *PAT-TOKEN* within the repositories.

5.7.3 ArgoCD Repository

Workflow Update

This workflow is triggered by the Workflow Update of the infrastructure repository (see 5.7.2). It takes as inputs the following values, passed by the Workflow Push:

- *Code Branch Name*;
- *Code Repository Name*;
- *Tier*: it can assume *frontend* or *backend* as values;
- *App Name*: this is the name of the application;
- *Default input flag*: it can be *true* or *false*. If it is true, the *input* file inside the backup-files repository is used, otherwise the one within the code repository, in the proper branch.
- *Backup-files Repository*: if the default input flag is set to true, the *input* file will be fetch from this repository.

The steps of the workflow are as follows:

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Check the existence of the *config* file:** if this file does not exist within the repository, the workflow is interrupted.
3. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.

4. Validation of the *config* file: a Python script processes the *config* file, checking that it contains all the necessary fields and valid values for every field.
5. **Creation of variable from the *config* file:** a Python script extract values from the *config* file.
6. **If Default Input Flag is equal to true:**
 - (a) **Checkout Backup-files Repository.**
7. **If Default Input Flag is equal to false:**
 - (a) **Checkout code repository:** it is checked out the code repository in order to fetch the *input* file.
8. **Update Manifests:** a Python script is in charge to generates new ArgoCD Application manifests.
9. **Save changes:** the changes within the infrastructure repository are saved with the *git commit* and *git push*.

Workflow Delete

This workflow is triggered by the Workflow Delete of the infrastructure repository (see 5.7.2). It takes as inputs the following values, passed by the Workflow Push:

- *Code Branch Name*;
- *Tier*: it can assume *frontend* or *backend* as values;
- *App name*: this is the name of the application.

The steps of the workflow are as follows:

1. **Checkout code:** This action checks-out the repository containing the current workflow. Thanks to it, the workflow can access the repository.
2. **Setup Python:** Python environment is set up, with version 3.x and additional modules are installed.
3. **Delete manifests:** a Python script delete all the manifests associated to a branch. It happens for the *releases/*** or *features/*** branches.
4. **Save changes:** the changes within the infrastructure repository are saved with the *git commit* and *git push*.

5.8 Example

In order to fully understand how the proposed solution works, a complete example will follow. Since the thesis solution was designed to be applied to an applications consisting of a backend and frontend, the example is based on such an application, which is composed by an Angular frontend, a Spring Boot backend and MariaDB database. The name of the application is *thesis-app*.

5.8.1 Initial setup

Repositories

The initial setup consists of creating all the necessary repositories, so: code-frontend-repo, code-backend-repo, infrastructure-repo, argocd-repo and backup-files-repo (the choice of names is arbitrary). Every repository must be filled with the proper files (see 5.6) and secrets. It is not necessary to create immediately the *input* files in the code repositories. They can be created when developers want to deploy a new environment. Here the repositories have been named as follows:

- *thesis-code-backend*;
- *thesis-code-frontend*;
- *thesis-infrastructure*;
- *thesis-argocd*;
- *thesis-backup-files*.

The figures 5.8 and 5.9 show the content of the *input* files.

1	<code>docker-backend-repo: thesis-backend</code>	1	<code>docker-backend-repo: thesis-backend</code>
2	<code>docker-frontend-repo: thesis-frontend</code>	2	<code>docker-frontend-repo: thesis-frontend</code>
3	<code>infrastructure-repo: thesis-infrastructure</code>	3	<code>infrastructure-repo: thesis-infrastructure</code>
4	<code>backup-input-repo: thesis-backup-files</code>	4	<code>backup-input-repo: thesis-backup-files</code>
5	<code>tier: frontend</code>	5	<code>tier: backend</code>
6	<code>default-input-prod: false</code>	6	<code>default-input-prod: false</code>
7	<code>default-input-release: false</code>	7	<code>default-input-release: false</code>
8	<code>default-input-feature: false</code>	8	<code>default-input-feature: false</code>
	thesis-code-frontend		thesis-code-backend

Figure 5.8: *config* files content

To make the workflows work, it is important to set the same Personal Access Token (PAT) within the repositories containing the workflows.



Figure 5.9: *config* files content

In GitHub, PAT is set by going to "*Settings > Developer Settings > Personal access tokens*" and clicking to "*Generate new token*". Once this is done, the "*workflow*" box must be checked and the token generated must be inserted in all repositories containing communicating workflows. To insert the token inside the repository, the user has to click on "*Settings > Secrets*" and then on "*Generate new token*", pasting in this place the token.

As code repositories must push images to DockerHub, they must contain an access token (set as secret), which is generated by going to the *DockerHub settings* (in the DockerHub website). The *kustomize/base/* directory, which is contained in the *thesis-infrastructure-repo*, must be filled with the application YAML files. In this case, it contains the following files:

- *deployment-backend.yaml*;
- *deployment-frontend.yaml*;
- *deployment-database.yaml*;
- *service-backend.yaml*;
- *service-frontend.yaml*;
- *service-database.yaml*;
- *kustomization.yaml*.

Clusters

Two Kubernetes clusters were chosen to be administered. The first one is a *on-premise* cluster, which consists of two nodes, *master* and *worker*, generated from two VMs with Centos7 as OS. The second one is a cluster created with *Microsoft Azure*, then it is a public cloud cluster. It was chosen to install ArgoCD only in the on-premise cluster, as it does not need to be installed in all clusters. In order for ArgoCD to deploy in the Azure cluster, there is a simple procedure to follow:

1. Copy the the Azure *kubeconfig* file into the *.kube* directory of the on-premise cluster, which contains ArgoCD;

2. Set the *KUBECONFIG* environment variable so that it contains the paths to both on-premise and Azure kubeconfigs (the two paths must be separated by `":"`);
3. Run `argocd cluster add CONTEXT --name <name-azure-cluster>`, where *CONTEXT* value can be found within the *kubeconfig* file and *name* can be chosen arbitrarily. The chosen name will be the one used by ArgoCD to identify the cluster.

ArgoCD

After installing ArgoCD, it is ready to be used. The simplest way to interact with it, is through the web UI, even because the solution is thought to be used by developer firstly. But for the deploying of the first manifest, which follows the *App of apps pattern* (see section 4.6), it is easier to use the `kubectl apply` command. The manifest, which is in the *thesis-backup-files* repository, can be downloaded locally by cloning the repository.

5.8.2 Startup and deployment of the production environment

The example starts with a production environment already created, in terms of YAML files within the *prod* directory of the *thesis-infrastructure* repository, ready to be deployed. The related *input* file can be seen in the figure 5.10.

The DockerHub repositories already contain an image for the frontend and one for the backend.

The infrastructure repository, as a consequence of the *Workflow Push*, started the *Workflow Update*, which created the *overlays* directory and within it, the *prod* directory.

The *Workflow Update* which belongs to the infrastructure repository, triggered the *Workflow Update* of the ArgoCD repository, which have generated the two ArgoCD manifests.

These manifests will allow the deployment of the production environment, both in the on-premise cluster (in-cluster) and in the Azure cluster (azure-cluster).

In order for ArgoCD to start managing the *thesis-app*, an user must manually execute the command shown in figure 5.13, which applies the manifest, described in the figure 5.14, inside k8s.

Watching the ArgoCD web UI (figure 5.15), three Applications has been deployed:

- **thesis-app-prod-azure-cluster**: the production environment, which is deployed in the Azure cluster, as can be deduced from its name;

```
1  backend:
2    - replicas: 1
3      type: deployment
4    - port: 8080
5      type: service
6  frontend:
7    - replicas: 2
8      type: deployment
9    - port: 80
10     type: service
11  db:
12    - type: deployment
13      secrets:
14        - MYSQL_ROOT_PASSWORD: password
15        - MYSQL_DATABASE: thesisAppMariadb
16  branch: master
17  image:
18    frontend:
19      tag: latest-prod
20    backend:
21      tag: latest-prod
22  clusters:
23    - in-cluster
24    - azure-cluster
```

Figure 5.10: *input* file within the *master* branch

- **thesis-app-prod-in-cluster:** the production environment, which is deployed in the on-premise cluster;
- **thesis-app:** this is the Application which has applied manually. It watches the *thesis-argocd* repository, maintaining synchronized the environments related to the *thesis-app* web application.

As it can be seen from the figure 5.16 and figure 5.17, the pods have been created in the two clusters.

Development of a new feature

The second step is to provision a new environment for the development of a new feature, on the backend.

To do this, you need to create a new feature branch from develop, in the *thesis-code-backend* repository. The feature branch is called *features/f1*.

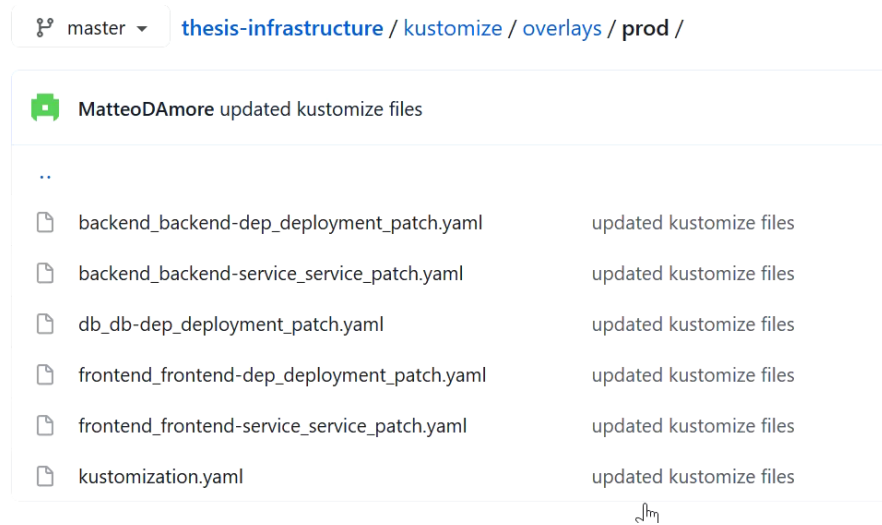


Figure 5.11: `prod` directory within the *thesis-infrastructure* repository

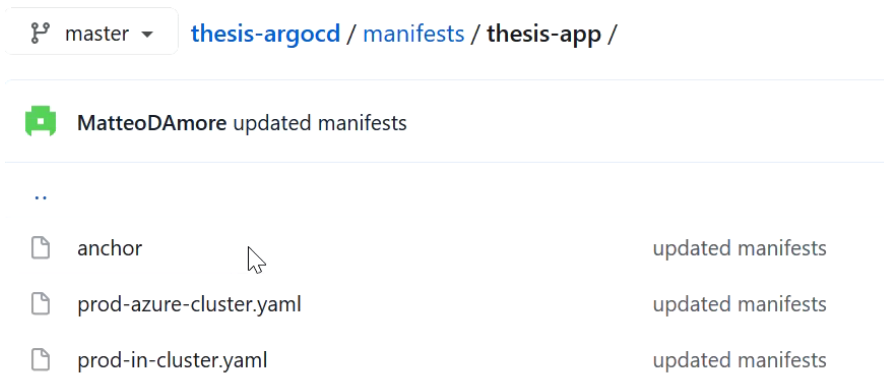


Figure 5.12: Application manifests within the ArgoCD repository

```
[root@kube-master:thesis-app]# kubectl apply -f thesis-app-manifest.yaml
application.argoproj.io/thesis-app created
```

Figure 5.13: Command to apply the initial manifest

The *input* file for this branch is shown in figure 5.18. It differs from that of the production environment in the number of replicas for the frontend, which in this case is only one.

Also in this case, workflows generate a new folder in the *thesis-infrastructure* repository (figure 5.19) and two new manifests in the *thesis-argocd* repository (figure 5.20).

```

1  apiVersion: argoproj.io/v1alpha1
2  kind: Application
3  metadata:
4    finalizers:
5      - resources-finalizer.argocd.argoproj.io
6    name: thesis-app
7    namespace: argocd
8  spec:
9    destination:
10     name: in-cluster
11     namespace: thesis-app
12    project: default
13    source:
14     path: manifests/thesis-app
15     repoURL: https://github.com/MatteoDAmore/thesis-argocd.git
16     targetRevision: HEAD
17    syncPolicy:
18     automated:
19       allowEmpty: true
20       prune: true
21       selfHeal: true
22     syncOptions:
23       - CreateNamespace=true

```

Figure 5.14: Content of the initial manifest

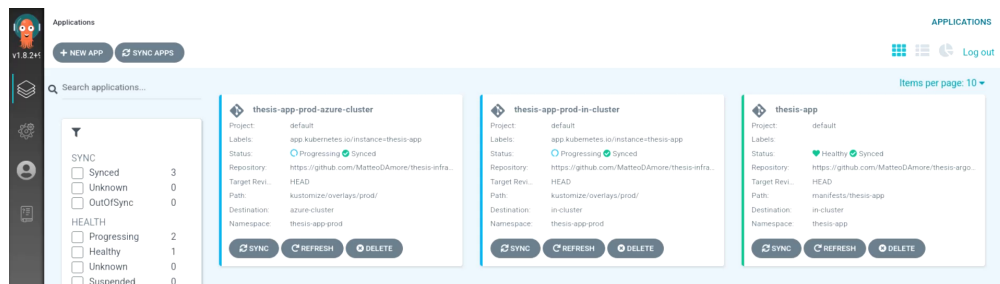


Figure 5.15: ArgoCD web UI

Once the workflows have been completed, ArgoCD reacts by synchronising the live state with the desired one (in the ArgoCD repository), adding the two new environments (figure 5.21).


Figures 5.22 and 5.23 show the new pods created automatically.

Once the development of the feature is finished, a pull request must be opened, to merge it back into develop. The code can be validated and reviewed and then, merged. By doing so, the workflows will remove the files associated with that


```
[root@kube-master:thesis-app]# kubectl get po --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
thesis-app-prod	backend-dep-85ff95899f-btbk9	0/1	ContainerCreating	0	21s
thesis-app-prod	db-dep-6869fbc89-qmd8q	0/1	ContainerCreating	0	21s
thesis-app-prod	frontend-dep-7f49f54886-bnb4r	0/1	ContainerCreating	0	20s
thesis-app-prod	frontend-dep-7f49f54886-tcz4d	0/1	ContainerCreating	0	20s

Figure 5.16: *prod* environment deployed in the on-premise cluster

Home > azure-cluster  ...
Kubernetes service

```
Bash s257359@Azure:~$ kubectl get po --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
thesis-app-prod	backend-dep-5b4c9c858d-7fjks	0/1	ContainerCreating	0	33s
thesis-app-prod	db-dep-54d6f86b7d-bt8pk	0/1	ContainerCreating	0	33s
thesis-app-prod	frontend-dep-776d6986bf-h41s9	0/1	ContainerCreating	0	33s
thesis-app-prod	frontend-dep-776d6986bf-npjnb	1/1	Running	0	33s

Figure 5.17: *prod* environment deployed in the Azure cluster

branch and consequently, the environments will also be removed from the clusters.

Creation of a release branch

Now the backend is ready for a new release. Then from the *develop* branch, a new release branch is generated, called *releases/r1*. The workflows are always the same, and once they have finished processing, ArgoCD will generate the new environment. In this case, as can be seen from the contents of the *input* file (figure 5.24), it was decided to deploy only to the Azure cluster (figure 5.25).

After few seconds, the new pods are up and running in the Azure cluster (figure 5.26).

The new release is tested and once it is ready, a new pull request is opened for it to be merged with the master branch. The pull request is checked, validated, reviewed and finally merged. As a result of this merge, multiple workflows are

```

1  backend:
2    - replicas: 1
3      type: deployment
4    - port: 8080
5      type: service
6  frontend:
7    - replicas: 1
8      type: deployment
9    - port: 80
10     type: service
11  db:
12    - type: deployment
13      secrets:
14        - MYSQL_ROOT_PASSWORD: password
15        - MYSQL_DATABASE: thesisAppMariadb
16  branch: features/f1
17  image:
18    frontend:
19      tag: latest-prod
20    backend:
21      tag: latest-prod
22  clusters:
23    - in-cluster
24    - azure-cluster

```

Figure 5.18: The *input* file related to the new feature *features/f1*



Figure 5.19: The new directory *f1* created within the infrastructure repository

triggered in the *thesis-code-backend* repository:

- **Workflow Push:** this workflow updates the new backend image for the production environment.
- **Workflow Delete:** this workflow will trigger other workflows, in order to remove files and the environment on the k8s cluster related to the *releases/r1* branch (figure 5.27).
- **Workflow Rebase:** this workflow will rebase the develop branch with the master branch.

- 📄 anchor
- 📄 backend-features-f1-azure-cluster.yaml
- 📄 backend-features-f1-in-cluster.yaml
- 📄 prod-azure-cluster.yaml
- 📄 prod-in-cluster.yaml

Figure 5.20: The manifests within the ArgoCD repo, after the creation of the new feature *f1*

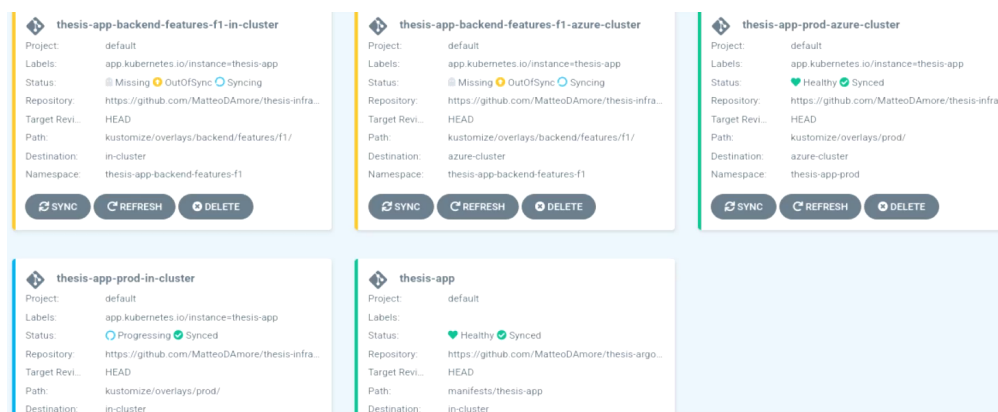


Figure 5.21: ArgoCD web UI with the new environments associated with *features/f1*

```
[root@kube-master:thesis-app]# kubectl get po --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
thesis-app-backend-features-f1	backend-dep-5956d4999-5z7q8	0/1	ContainerCreating	0	35s
thesis-app-backend-features-f1	db-dep-6869fbc89-7c6nk	0/1	ContainerCreating	0	35s
thesis-app-backend-features-f1	frontend-dep-7f49f54886-lt5gs	0/1	ContainerCreating	0	36s
thesis-app-prod	backend-dep-85ff95899f-btbk9	1/1	Running	0	7m45s
thesis-app-prod	db-dep-6869fbc89-qmd8q	1/1	Running	0	7m45s
thesis-app-prod	frontend-dep-7f49f54886-bnb4r	1/1	Running	0	7m44s
thesis-app-prod	frontend-dep-7f49f54886-tcz4d	1/1	Running	0	7m44s

Figure 5.22: *features/f1* deployed in the on-premise cluster

```
s257359@Azure:~$ kubectl get po --all-namespaces
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
thesis-app-back	backend-dep-77d9489c96-wjldf	1/1	Running	1	20s
thesis-app-back	db-dep-54d6f86b7d-jfb56	1/1	Running	0	20s
thesis-app-back	frontend-dep-776d6986bf-f98h7	1/1	Running	0	20s
thesis-app-prod	backend-dep-5b4c9c858d-7fjks	1/1	Running	1	7m18s
thesis-app-prod	db-dep-54d6f86b7d-bt8pk	1/1	Running	0	7m18s
thesis-app-prod	frontend-dep-776d6986bf-h4ls9	1/1	Running	0	7m18s
thesis-app-prod	frontend-dep-776d6986bf-npjb	1/1	Running	0	7m18s

```
s257359@Azure:~$
```

Figure 5.23: *features/f1* deployed in the Azure cluster

```

1  backend:
2    - replicas: 1
3      type: deployment
4  - port: 8080
5      type: service
6  frontend:
7    - replicas: 1
8      type: deployment
9  - port: 80
10     type: service
11 db:
12   - type: deployment
13     secrets:
14       - MYSQL_ROOT_PASSWORD: password
15       - MYSQL_DATABASE: thesisAppMariadb
16 branch: releases/r1
17 image:
18   frontend:
19     tag: latest-prod
20   backend:
21     tag: latest-prod
22 clusters:
23   - azure-cluster

```

Figure 5.24: The *input* file related to the new feature *releases/r1*

- **Workflow Update-tag:** this workflow will add a new tag to the master branch. As there were no tags associated with the master before, the new version will be *v0.1* (figure 5.28). See section 5.3.3 for more details about the tag strategy.

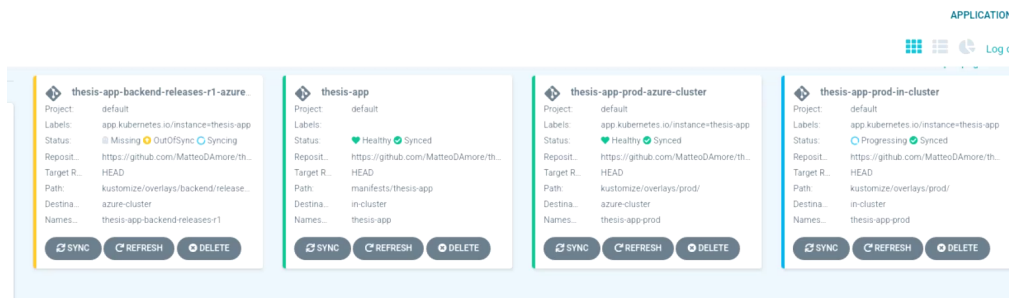


Figure 5.25: ArgoCD web UI with the new environment associated with *releases/r1*

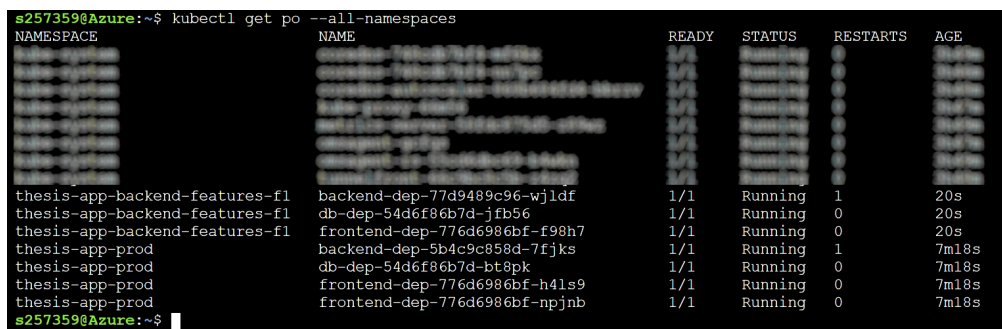


Figure 5.26: *releases/r1* deployed in the Azure cluster

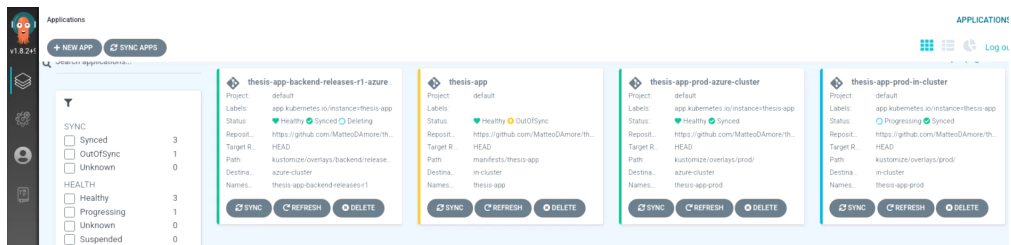


Figure 5.27: ArgoCD reacts to the deletion of the manifest associated with the *release/r1* branch

HotFix branch

After the new version of the application was put into production, a bug was discovered. Therefore, it is needed to create a hotfix branch from the master branch to fix the production environment. In the example, a branch called *hotfix/newhotfix* has been created. The creation of this branch does not automatically deploy any environment, as it is designed to fix small problems at speed.

When the bug has been fixed, this branch must be merged back to the master

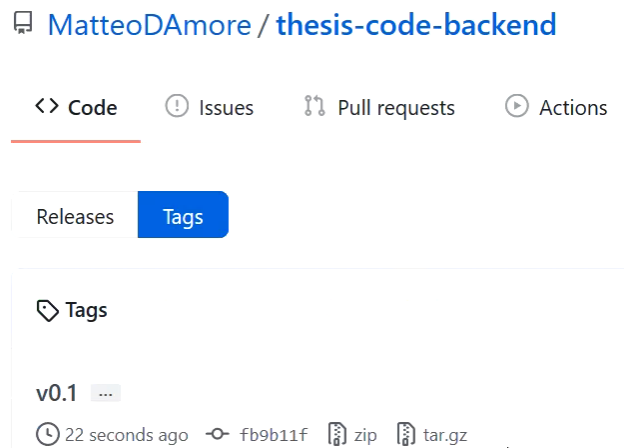


Figure 5.28: New tag for the master branch

branch, via a pull request. When the pull request will be closed, as a consequence of the merge, two workflows will be triggered:

- **Workflow Rebase:** the develop branch is rebased with the master branch.
- **Workflow Update-tag:** now the new value of the master tag is *v0.2*.

5.9 Objectives and Validation

In general, there are already advantages and peculiarity that have been explored in the GitOps section (see 3.3) and in the ArgoCD chapter (see 4). What has been achieved in the development of this project is a structured solution to manage infrastructure files related to a full stack application, taking advantages of all the benefits of Git. Infrastructure files can be created, modified and destroyed automatically, thus enabling Continuous Delivery and Continuous Deployment and achieving greater performance in terms of time and minimising the possibility of human error. All the best practices related to GitOps are respected, with the adoption of a pull based pipeline, leveraging on ArgoCD. The *input* file is a simple operating interface, which also allows developers to generate customised environments, which will be deployed on Kubernetes. This can be done without them having to know specifically how Kubernetes works, how to write *kubectl* commands or YAML files related to k8s objects. The solution also works well in a multi-cluster and multi-vendor environment, providing flexibility and the possibility to be used in application contexts other than the one studied.

5.9.1 Infrastructure incident recovery

Thanks to ArgoCD and the continuous check between the state of the environment deployed live on k8s and the desired state in the remote Git repository, improper changes (voluntary or involuntary) to the live infrastructures are immediately corrected. So there is a drastic reduction in the re-synchronisation time of the environment due to an infrastructure incident. If there is no ArgoCD, which reacts promptly, the recovery would have to be done manually, which would take much longer and consequently increase the downtime of the application. Moreover, it takes no effort to understand what is not working, only to realise that infrastructure parameters have been improperly changed or Kubernetes objects have been removed. In order to evaluate the reaction time of ArgoCD, evaluation experiments were carried out. Re-synchronisation times were calculated following a change made in the application's live environment. For the experiment, three replicas of the *repository server* and the *API server* were used.

As it can be seen, these operations all take a few seconds, since as soon as something is modified or deleted, ArgoCD reacts. Two tests have been carried out: In the first, the live environment was modified in a context where ArgoCD contained only this environment. In the second, it contained 10 different environments. The average time did not change, so reporting the second graph would have been insignificant, as it is practically the same as the first.

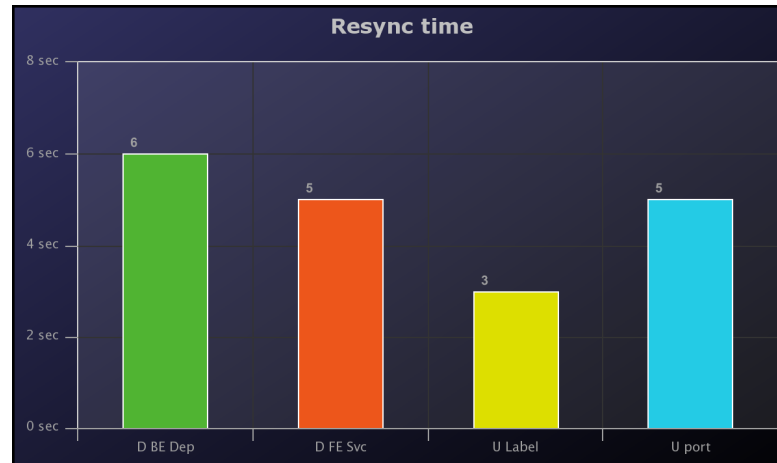


Figure 5.29: Re-synchronisation time (U=update, D=delete, BE=backend, FE=frontend, Dep=Deployment, Svc=Service)

5.9.2 Deployment time of environments

Another interesting fact is the time taken by ArgoCD to deploy N environments on Kubernetes. This can be interesting in the case of disaster recovery, if you have to redeploy a lot of environments. If the time taken is short, the disruption is less. Another case is the provisioning of multiple environments in a multi-cluster context. The first test was made with the standard configuration, already expressed in the previous sections:

- The on-premise cluster composed by two nodes, the master and the worker. The master has allocated 2 of the 8 CPU's of the laptop used to host the cluster and 4600MB RAM. The worker has allocated 1/8 CPU and 3600MB RAM.
- The Azure cluster, with 2vCPU and 8GB RAM.
- ArgoCD installed within the on-premise cluster.
- The environments have been distributed equally between the two clusters.

As it can be seen in the figure 5.30, the deployment time is reasonable, but after the deployment of 7 environments, the time starts to grow quite fast. The bottleneck is in the pod replicas concerning ArgoCD and also in the scarce resources set up for the on-premise cluster. In the second test, everything was left unchanged, but the pod replicas of ArgoCD were increased. The number of replicas for the *API server* and *repository server* were increased from one to three. It is clear that the

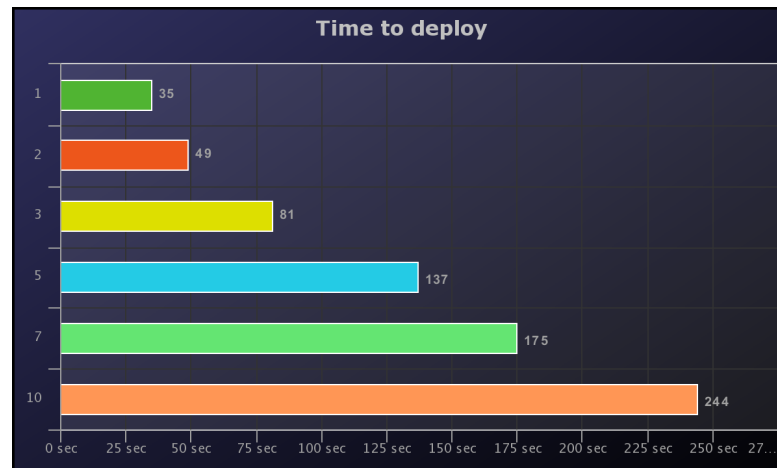


Figure 5.30: First test on deployment time

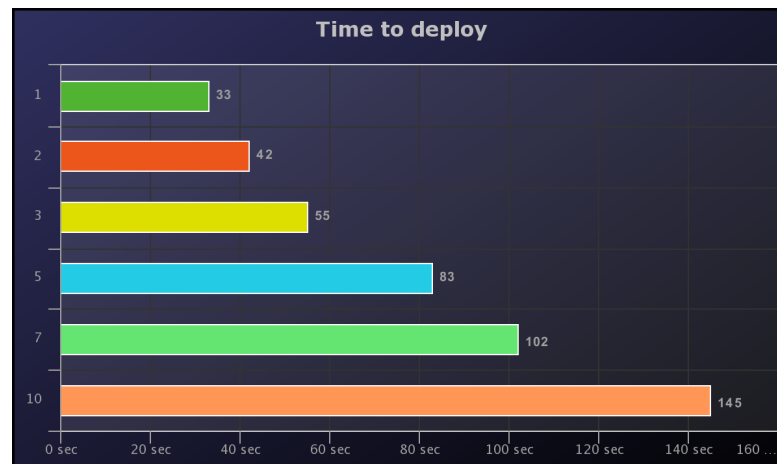


Figure 5.31: Second test on deployment time

situation is much better (figure 5.31), but the resources of the on-premise cluster are still a bottleneck.

Then, it has been done a last test. The only way to improve performance, as the resources of the laptop used to host the on-premise cluster are limited, was to use only the Azure cluster, which is more powerful.

So, ArgoCD was installed on Azure, and the number of replicas concerning the *API server* and the *repository server* were set to three. In this last test (figure 5.32), a significant increase in performance is observed. This leads to the conclusion that, by using an appropriate number of cluster resources, depending on the average workload of the cluster, deployment times are very low.

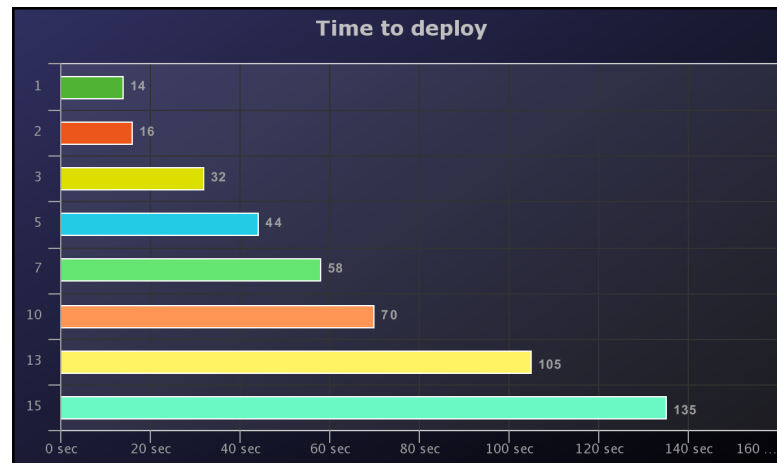


Figure 5.32: Third test on deployment time

5.9.3 *input* file

One thing that was sought during the development of the solution was the simplicity and intuitiveness of the *input* file. This is fundamental, in order to have a tool that is as complete as possible, but at the same time, it is less complex than the editing of multiple YAML files related to Kubernetes resources. The average length of an *input* file is about 23-26 rows.

```

1  backend:
2    - replicas: 1
3      type: deployment
4    - port: 8080
5      type: service
6  frontend:
7    - replicas: 1
8      type: deployment
9    - port: 80
10     type: service
11  db:
12    - type: deployment
13      secrets:
14        - MYSQL_ROOT_PASSWORD: password
15        - MYSQL_DATABASE: thesisAppMariadb
16  branch: releases/r1
17  image:
18    frontend:
19      tag: latest-prod
20    backend:
21      tag: latest-prod
22  clusters:
23    - azure-cluster

```

Figure 5.33: Example of an *input* file

In order to assess the user-friendliness, it was evaluated by 8 employees, who had little or no knowledge of Kubernetes. The "how-to" was explained to them within five minutes. After that, they were asked to use it by deploying one or more environments.

Finally, they were asked to give a grade from 0 to 5 about the user-friendliness, simplicity and intuitiveness of the *input* file (figure 5.34).

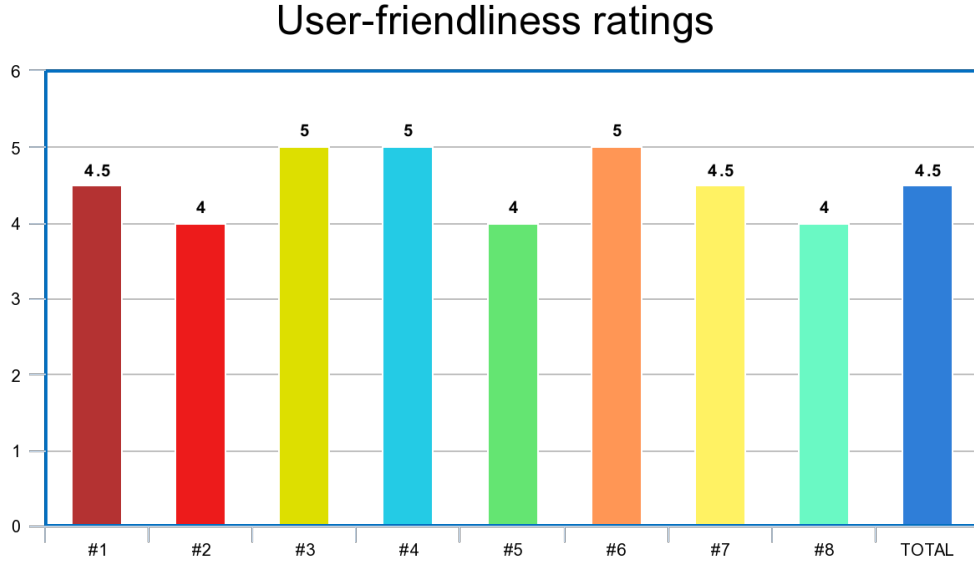


Figure 5.34: Ratings

As can be seen from the graph, the grades were quite satisfactory.

5.9.4 Workflows execution time

An important part of the thesis work concerns the writing of workflows. It is therefore important to consider whether the execution times of these workflows are acceptable. In the graph 5.35 there are the average times of all workflows, calculated over multiple executions at different times and in different contexts.

The graph shows that all workflows take a few seconds to run, except for the Workflow Push. However, it can be noticed from the graph 5.36, that much of the time is taken up by the build and push process for the Docker image. For the compilation of these graphs, two different applications were considered, one simpler and the other more complex, in terms of LOC.

So, considering the benefits these workflows give, in terms of process automation, the time they take is very low.

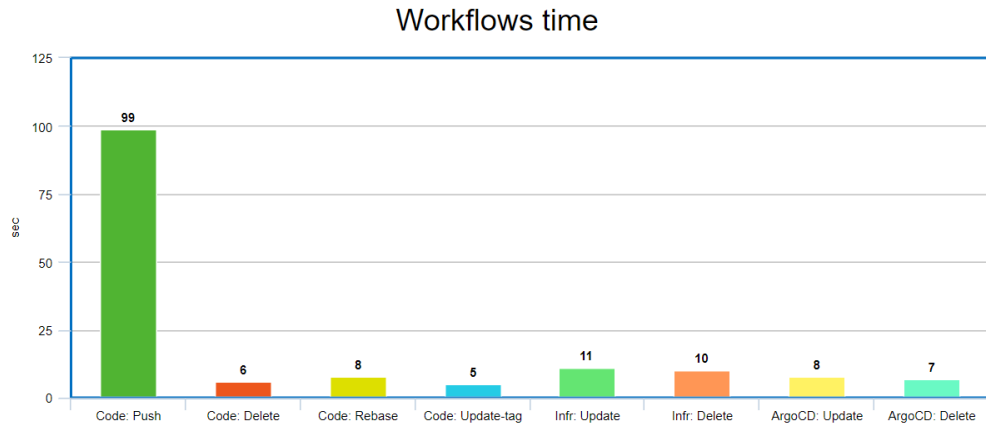


Figure 5.35: Average execution time for each workflow. Code, Infr and ArgoCD refer to the repositories

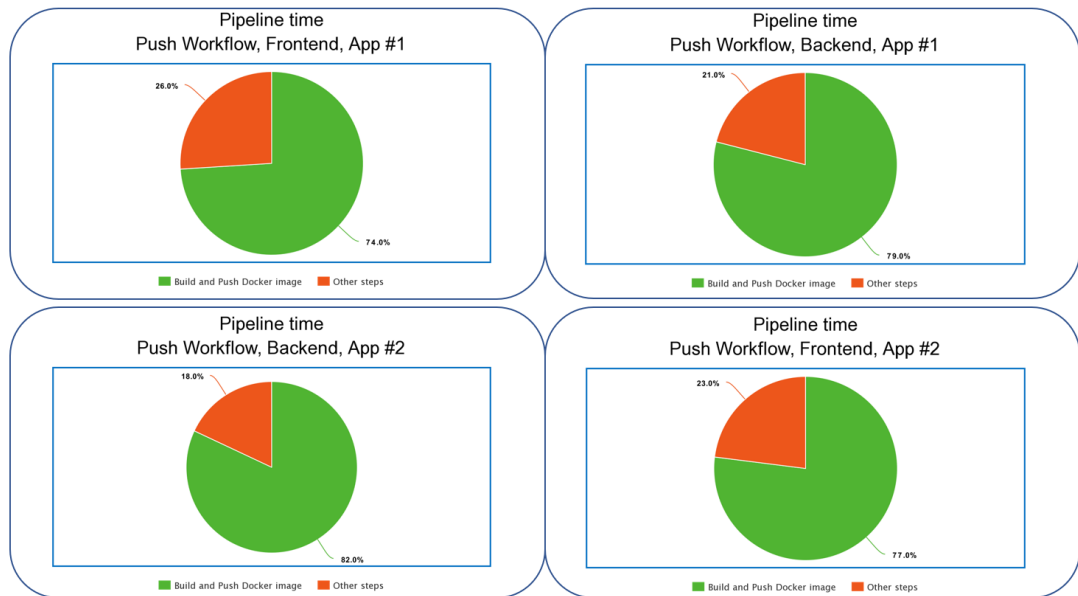


Figure 5.36: Percentage of time required to build and push the Docker image

5.9.5 Reduction of provisioning and update time

In a business context, when a developer requests the provisioning or modification of an environment on k8s, the time required can be considerable. Instead, with the solution proposed, doing this takes much less time. A manual provisioning takes time and efforts by operations team. The operator, after the approval cycles, has to schedule when to create the environment. Then, he has to create it manually

and this takes time. If the developer then needs updates, the process starts all over again. In the thesis solution, there may be the same approval cycle, but once it is finished, deployment is automated or semi-automated.

The process could be semi-automatic because, due to internal policies, a validation check by an operations team employee may be required. In this case, validation is a faster and leaner process, because it is limited to the review of the single *input* file, and not of all the infrastructure YAML files, which may be several.

5.9.6 Approval process

Since the solution uses GitHub repositories, you could rely on the *pull requests*, which allows changes to be reviewed before they are approved.

There are less critical situations where reviews could be bypassed, streamlining the provisioning or modification process, depending on the branches. Some critical branches such as production could pass through a more fine-grained process of acceptance and review.

Other branches, such as feature branches, may be subject to fewer controls.

5.10 Ligo

Ligo is an open source project started at Politecnico of Turin that allows Kubernetes to seamlessly and securely share resources and services, so you can run your tasks on any other cluster available nearby [34].

Assume that a company manages N clusters and in each one, Ligo is installed. From the point of view of a specific cluster, the other $N-1$ remote clusters are seen as worker nodes, or more precisely *virtual nodes*.

After that, pods can be deployed in one of the N available clusters. There are several ways, in k8s, to force a pod to be scheduled to a specific node.

nodeSelector

In Kubernetes, nodes can be labelled with multiple values. The way to do this is very simple:

```
$ kubectl label nodes <node-name> keyLabel=valueLabel
```

To see the labels of a node:

```
$ kubectl get nodes --show-labels
```

For instance, assume that the worker node called *w1* has been labelled with *secondName=myFavourite*. If you want to deploy a pod on this node, using the new label, you have to use *nodeSelector* field:

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: db
5    labels:
6      tier: db
7  spec:
8    containers:
9      - name: mariadb
10        image: mariadb
11    nodeSelector:
12      secondName: myFavourite
```

nodeName

You can also schedule a pod to one specific node via setting *nodeName*.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: db
5  spec:
6    nodeName: w1
7    containers:
8      - name: mariadb
9        image: mariadb
```

Affinity and anti-affinity

nodeSelector provides a simple way to constrain pods to nodes with specific labels. If you want a more complex way to select the target node, *affinity/anti-affinity* are a better choice:

- The language offers a huge number of matching rules;
- The rules can be *soft* or *preference* instead of a strict constraint. If the scheduler can't satisfy rules, the pod will still be scheduled;
- There are also constraints against labels on other pods running on the node, rather than against labels on the node itself, which allows rules about which pods can and cannot be co-located. For this reason, there are *node affinity* and *inter-pod affinity/anti-affinity*.

5.10.1 Ligo transparency

One of the greatest advantages of using Ligo is that it is transparent.

When it has been installed in the two clusters used in the thesis work, ArgoCD continued to function correctly. Nothing has been broken. This is the reason why we have tried to introduce it in the thesis work, finding an alternative way to use the ArgoCD multi-cluster feature.

5.10.2 Ligo alongside ArgoCD

In the thesis work, an application can be deployed in one of the available k8s cluster. The cluster names are meaningful within the context of ArgoCD and they are unique. This means that once a cluster has a name, it is unique.

The name actually could be changed, but this is strongly discouraged because it could affect Applications that already exist. If a manifest is deployed on a specific cluster, that manifest contains the cluster name.

The problem is that a developer, who needs to provision a new application, might find the cluster name insufficient to choose the right cluster. Perhaps the developer does not even know the association between the cluster name and the actual cluster. By using labels, users can achieve greater expressiveness, being able to describe clusters with keywords, making the choice of deployment easier and more guided. It would be convenient if he could benefit from a more accurate and fine-grained description of clusters, through multiple labels. He could even add custom labels, to make it easier to recognise clusters.

This can be done using the *nodeSelector* field or *affinity/anti-affinity* into the proper infrastructural YAML files. Thanks to Ligo, all the other clusters are seen as k8s nodes, then it is easy to apply labels to them. If the developer has the rights, he can do this himself, otherwise, he can ask to someone who has the rights. The operation of adding a label needs only one command, then it is nothing onerous.

5.10.3 Implementation

For this solution to work, a few changes must be made. Firstly, Ligo must be installed in every Kubernetes cluster. In addition, a new field (a flag) must be implemented in the *input* file, specifying whether the Ligo approach or the standard approach is to be used. With the standard approach, the name of the clusters in which the environment is to be deployed are indicated in the *clusters* field.

On the other hand, with the Ligo approach, clusters must be indicated within the Deployment YAML files, using *nodeSelector* or *affinity/anti-affinity* fields and indicating the labels. For the sake of uniformity, the list of labels could be listed in a *clusterLabels* field. Then, a Python script should automatically fill in the

Deployment resources, with the aim of having a simpler, more readable solution, that is less prone to human error.

Chapter 6

Conclusion

In the Master's Thesis the GitOps approach was studied, with the intention of applying it in a concrete case and understanding its potential. Together with this, it was decided to use ArgoCD, a native GitOps tool among the most widely adopted in company contexts, so that its advantages could be exploited.

The use case was the deployment of a solution to allow provisioning and one-click configuration of Kubernetes environments, with the peculiarity of being a multi-cluster and multi-vendor solution, to achieve scalability and portability.

To achieve this, a practical and simple tool, the *input* file, has been devised, which allows even employees such as developers to be as autonomous as possible in the creation, modification and deletion of Kubernetes environments. These environment are related to feature development, release testing and production deployment, so they are generally necessary environments for the deployment and maintenance of an application.

The thesis work, while covering phases related to the Continuous Integration, focused on the automation of infrastructure processes and their maintenance. In addition, the focus was on the Continuous Delivery and Continuous Deployment, as GitOps is an approach designed to deal with these aspects.

It was decided to build a solution tailored to a web application, consisting of two code repositories, one for the backend and one for the frontend.

Although this is a limitation, with appropriate modifications and rearrangements of the architecture and scripts, it is possible to adapt this solution to a microservices application.

The scripts and workflows are designed to be as generic as possible. In fact, for example, if you want to extend the *input* file by adding the customisation of a new infrastructure parameter, the average number of lines of code required to do this is less than 15.

The thesis work was carried out in a consultancy firm. As a consequence, after having been evaluated and validated by internal staff, it was decided to start from

this solution to put in place a new version, for both internal and external use.

Bibliography

- [1] Ryan. *Container e Containerization: cos'è e come funziona*. 2019. URL: <https://www.ryadel.com/wp-content/uploads/2019/08/containers-vs-virtual-machines-1024x518.jpg> (cit. on p. 4).
- [2] *What is Docker*. URL: <https://opensource.com/resources/what-docker> (cit. on p. 5).
- [3] *Docker overview*. URL: <https://docs.docker.com/get-started/overview/> (cit. on p. 5).
- [4] Edward Kisler. *A Beginner's Guide to Understanding and Building Docker Images*. 2020. URL: <https://jfrog.com/knowledge-base/a-beginners-guide-to-understanding-and-building-docker-images/> (cit. on p. 5).
- [5] *What are public, private and hybrid clouds?* URL: <https://azure.microsoft.com/en-in/overview/what-are-private-public-hybrid-clouds/> (cit. on p. 9).
- [6] Toye Idowu. *Introduction to Immutable Infrastructure*. 2019. URL: <https://www.bmc.com/blogs/immutable-infrastructure/> (cit. on p. 10).
- [7] Kris Flores. *Mutable Vs Immutable Infrastructure – A Comprehensive Guide to Choose the Best*. 2020. URL: <https://www.bridge-global.com/blog/mutable-vs-immutable-infrastructure/> (cit. on p. 10).
- [8] *What is Kubernetes?* URL: <https://www.redhat.com/en/topics/containers/what-is-kubernetes> (cit. on p. 11).
- [9] Lawrence E Hecht. *What data says about Kubernetes deployments patterns*. 2018. URL: <https://cdn.thenewstack.io/media/2018/03/40c0a560-chart-kubernetes-manages-containers-at-69-of-organizations-surveyed.png> (cit. on p. 11).
- [10] Ferenc Hámori. *The history of Kubernetes on a Timeline*. 2018. URL: <https://blog.risingstack.com/the-history-of-kubernetes/> (cit. on p. 11).
- [11] Sindhuja Cynixit. *Kubernetes Architecture*. 2019. URL: https://miro.medium.com/max/924/1*2y516oRxWBY9ASyN25t0mQ.png (cit. on p. 12).

- [12] *Kubernetes Service*. URL: <https://kubernetes.io/docs/concepts/services-networking/service/> (cit. on p. 15).
- [13] *Declarative Management of Kubernetes Objects Using Configuration Files*. 2021. URL: <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/declarative-config/> (cit. on p. 16).
- [14] *Custom Resources*. URL: <https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/> (cit. on p. 17).
- [15] *Kustomize official website*. URL: <https://kustomize.io> (cit. on p. 18).
- [16] *Declarative Management of Kubernetes Objects Using Kustomize*. 2021. URL: <https://kubernetes.io/docs/tasks/manage-kubernetes-objects/kustomization/> (cit. on p. 18).
- [17] *Kustomize Documentation*. 2020. URL: <https://kubect1.docs.kubernetes.io/references/kustomize/> (cit. on p. 18).
- [18] IETF. *JavaScript Object Notation (JSON) Patch*. URL: <https://tools.ietf.org/html/rfc6902#section-4.1> (cit. on p. 27).
- [19] *GitHub Actions Documentation*. URL: <https://docs.github.com/en/actions> (cit. on p. 28).
- [20] Scott Chacon and Ben Straub. *Pro Git book*. URL: <https://git-scm.com/book/en/v2> (cit. on p. 29).
- [21] Harsh Binani. *What is Devops? The complete guide to DevOps*. 2019. URL: <https://medium.com/cuelogic-technologies/what-is-devops-the-complete-guide-to-devops-with-examples-13db789dd1c> (cit. on p. 31).
- [22] *Roadway to IT Revolution: The History of DevOps*. URL: <https://www.appknox.com/blog/history-of-devops> (cit. on p. 31).
- [23] Meredith Courtemanche, Emily Mell, and Alexander S. Gillis. *What is DevOps, the ultimate guide*. 2020. URL: <https://searchitoperations.techtarget.com/definition/DevOps> (cit. on p. 31).
- [24] *What is Infrastructure as Code*. URL: <https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code> (cit. on p. 32).
- [25] URL: https://miro.medium.com/max/1050/0*n57zykBMd0dmUg7C.png (cit. on p. 33).
- [26] *Guide To GitOps*. URL: <https://www.weave.works/technologies/gitops/> (cit. on p. 35).
- [27] *Push Based Pipeline*. URL: <https://www.gitops.tech/images/push.png> (cit. on p. 37).

- [28] *Pull Based Pipeline*. URL: <https://www.gitops.tech/images/push.png> (cit. on p. 38).
- [29] *ArgoCD official website*. URL: <https://argoproj.github.io/argo-cd/> (cit. on p. 39).
- [30] Rafael Portela and Ádám Sándor. *Comparing GitOps tools*. 2020. URL: <https://blog.container-solutions.com/fluxcd-argocd-jenkins-x-gitops-tools> (cit. on p. 41).
- [31] Aditya Soni. *ArgoCD: GitOps Continuous Delivery Approach On Google Kubernetes Engine*. 2020. URL: https://miro.medium.com/max/1050/1*3nPB4xER4aC2iXVs40K4Pw.jpeg (cit. on p. 44).
- [32] Vincent Driessen. *A successful Git branching model*. 2010. URL: <https://nvie.com/posts/a-successful-git-branching-model/> (cit. on p. 51).
- [33] Vincent Driessen. *A successful Git branching model*. 2010. URL: <https://nvie.com/img/git-model@2x.png> (cit. on p. 52).
- [34] *Ligo official website*. URL: <https://liqo.io> (cit. on p. 94).
- [35] Aurore Malherbes. *Observability for CD with Argo*. 2021. URL: <https://www.padok.fr/en/blog/cd-argo>.
- [36] *The history of Cloud computing*. URL: <https://about.gitlab.com/topics/gitops/>.
- [37] *The history of Cloud computing*. URL: <https://www.cloudbees.com/gitops/what-is-gitops>.
- [38] *The history of Cloud computing*. URL: <https://www.scality.com/solved/the-history-of-cloud-computing/>.
- [39] Fulvio Risso. *Cloud Computing*. University Lecture. 2020.
- [40] Giovanni Malnati. *Internet Application*. University Lecture. 2020.
- [41] *Git in a Nutshell*. 2019. URL: <https://kancane.nl/git-in-a-nutshell/>.
- [42] Florian Beetz, Anja Kammer, and Simon Harrer. *GitOps*. URL: <https://www.gitops.tech>.
- [43] *DevOps: Principles, Practices, and DevOps Engineer Role*. 2021. URL: <https://www.altexsoft.com/blog/engineering/devops-principles-practices-and-devops-engineer-role/>.