ISPGAYA
instituto superior politécnico

ADVANTAGES OF CONTAINERIZATION RED HAT OPENSHIFT FOR APPLICATION MIGRATION

Miguel Ângelo Martins Gonçalves
2020100302@ispgaya.pt

Advisor: Eng. Justino Lourenço
External Advisor: Sérgio Vale Pires

Vila Nova de Gaia, July 2023

Acknowledgements

I would like to express my deepest gratitude to my advisors, Sérgio Vale Pires at Natixis and Eng. Justino Lourenço at ISPGAYA, for their guidance and support throughout this internship. I would also like to extend my appreciation to the entire DevOps System Team at Natixis for their warm welcome and inclusive attitude, despite being an intern, I was embraced as a valuable member of the team, their kindness, patience, and willingness to share their knowledge and expertise have fostered a collaborative and supportive environment. I would like to express my heartfelt gratitude to all those who have supported me throughout this journey.

Resumo

Este estudo tem como objetivo explorar o processo de migração de aplicações executadas num ambiente virtualizado para um ambiente conteinerizado, usando o Red Hat OpenShift. A tecnologia de containers representa uma área próspera na computação em nuvem. Para muitos tipos de computação, containers são uma alternativa viável às máquinas virtuais porque muitas aplicações não exigem kernels isolados. Os containers partilham o kernel com o host, ao contrário das máquinas virtuais que têm um kernel completamente isolado. Devido a esta distinção, os containers são mais *lightweight* e têm um desempenho superior. Este estudo irá aprofundar os aspectos técnicos da migração de aplicações de um ambiente VM para uma plataforma de containers. Explora as várias ferramentas, estruturas e metodologias disponíveis para a conteinerização e fornece informações sobre a sua aplicação prática durante o processo de migração.

Abstract

This study aims to explore the process of migrating applications running on a VM environment to a containerized environment, specifically utilizing Red Hat OpenShift. Container technology represents a thriving field in cloud computing, for many types of computing, containers are a viable alternative to virtual machines because many applications do not require isolated kernels. Containers share the kernel with the host, as opposed to virtual machines which have a completely isolated kernel. Because of this distinction, containers are more lightweight and higher performing. This study will delve into the technical aspects of migrating applications from a VM environment to a containerization platform. It will explore the various tools, frameworks, and methodologies available for containerization and provide insights into their practical application during the migration process.

Glossary

Distributed system Collection of interconnected computers or nodes that work together to achieve a common goal. These systems are designed to distribute and coordinate tasks across multiple machines, enabling collaboration and resource sharing among the nodes.

System calls A way for applications to interact with the operating system and access lower-level services, such as file operations, process management, network communication, and hardware resources.

Kernel Central control program that coordinates and executes various tasks, including process management, memory management, device drivers, file system management, and network communication.

Iptables firewall utility for Linux-based operating systems that provides packet filtering, network address translation (NAT), and network packet manipulation capabilities.

Daemon Background process or service.

Network Sockets Software endpoint that enables communication between processes over a computer network acting as an interface between an application program and the underlying network protocols.

Keywords

VM	Virtual machine
LB	Load Balancing
API	Application Programming Interface
QA	Quality Assurance
CI	Continuous Integration
CD	Continuous Deployment
OS	Operating System
CPU	Central Processing Unit
SED	Stream Editor
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
DNS	Domain Name System
HTML	HyperText Markup Language
CSS	Cascading Style Sheets
NPM	Node Package Manager
UI	User Interface
UML	Unified Modeling Language
SCM	Source Code Management
URL	Uniform Resource Locator
SaaS	Software as a Service
CaaS	Container as a Service
LXC	Linux containers
OCP	OpenShift Container Platform

Contents

Figures

Introduction

Background

In recent years, containerization has emerged as a transformative technology in the field of cloud computing. Containers provide lightweight, portable, and isolated environments for deploying applications, enabling efficient resource utilization and streamlined deployment processes. As organizations increasingly embrace containerization, the need to migrate existing applications from traditional VM environments to containerized platforms becomes a critical consideration.

Problem statement

Every newcomer to the Natixis DevOps System Team follows an internal application called "DevOps Formation", which is a web based application serving as an introduction to every DevOps related tool the team uses with examples and exercises, the end goal is that person following this guide either fixes the flaws it found along the way or makes a contribution to the project, this is incentivized because the application itself has a CI/CD pipeline, so the person making the changes gets to experience first hand the process of making changes to an application that is actually in production, even if it's just an internal training tool used by the DevOps team. The proposed project is migrating the current CI/CD pipeline of the application which is currently using a virtualized approach, to a containerized environment using Red Hat OpenShift container platform. Migrating applications from VM environments to containerized platforms poses unique challenges, while VMs offer isolation and encapsulation, they often come with increased resource overhead and slower deployment times. On the other hand, containers provide higher performance, scalability, and agility, but also introduce new considerations in terms of security, orchestration, and management.

Internship entity description

Natixis is a French corporate and investment bank created in November 2006 from the merge of the asset management and investment banking operations of *Natexis Banques Populaires* (Banque Populaire group) and *IXIS* (Groupe Caisse d'Epargne). It provides financial data for the 'Markets' section on the news channel, Euronews. In February 2021, Groupe BPCE made a tender offer for all Natixis shares it did not own. The offer was completed in June 2021 and Natixis stock was delisted.

It operates in the fields of investment banking, asset management, insurance, and financial services. Founded in 2006, it is headquartered in Paris, France, and is a subsidiary of Groupe BPCE, one of the largest banking groups in France. There are three main business lines: Corporate and Investment Banking (CIB), Investment Solutions, and Specialized Financial Services.

In the Corporate and Investment Banking segment, Natixis offers a wide range of financial services to corporate clients, institutional investors, financial institutions, and public sector organizations. These services include advisory and financing solutions, capital markets activities, mergers and acquisitions, structured financing, and global transaction banking.

The Investment Solutions division focuses on asset management, private banking, and wealth management services. Natixis Asset Management, a subsidiary of Natixis, provides a diverse range of investment products and solutions to individual investors, institutional clients, and financial advisors worldwide. The Private Banking division offers personalized financial services to high-net-worth individuals and families, including wealth planning, investment management, and estate planning. Specialized Financial Services encompass activities such as insurance, consumer finance, factoring, leasing, and payments. These services cater to specific sectors and markets, providing customized financial solutions to clients in areas such as energy, infrastructure, real estate, and shipping.

With a global presence, Natixis operates in key financial centers around the world, including Europe, the Americas, Asia-Pacific, and the Middle East. It serves a diverse client base, ranging from large corporations and institutional investors to individual clients seeking financial products and services, striving to combine its financial expertise with a responsible approach to sustainable finance. The institution focuses on integrating environmental, social, and governance (ESG) factors into its business activities and investment strategies, aiming to support sustainable development and responsible investing practices.

Objectives

The purpose of this study is to evaluate the viability of migrating applications running on VM environments to a containerized environment.

Identify the benefits and challenges: Evaluate the advantages and disadvantages of migrating the DevOps application from a VM environment to a containerized one. Assess the potential benefits, such as improved performance, scalability, resource utilization, and portability. Additionally consider additional factors, including security, data management, application dependencies, and architectural changes.

Explore Red Hat OpenShift's capabilities: Investigate the features and functionalities of Red Hat OpenShift Container Platform and understand how it facilitates the migration process. Explore the platform's capabilities for container orchestration, workload management, scaling, and application deployment.

Develop migration strategies and best practices: Develop practical migration strategies and best practices for transitioning applications from VM environments to Red Hat OpenShift. Consider factors such as application profiling, containerization of dependencies, image creation, network configuration, and storage management.

State of the art

Nowadays, distributed applications and infrastructures are moving from being VM centric to container centric, container technology is strongly backed by PaaS providers, IaaS providers and Internet Service Providers, containers are also used to deploy large scale applications in challenging fields, such as big data analytics, scientific computing, edge computing and Internet-of-Things(IoT).

Container technology and the resulting transformational effects were essentially born in the cloud, where cloud providers developed much of this technology and the early adoption of containers in the enterprise was predominately in the public cloud. Containers in the public cloud were the best possible fit because developers were targeting cutting-edge cloud-native applications, public clouds provide scalable infrastructure as well as application and data services that these containerized applications consume.

Enterprises face a challenge of having to manage older gen applications, while containers can still be used to containerize traditional apps with benefits associated, larger returns come from refactoring the application over time. Enterprises are beginning a transition phase where they are learning new container skills, methodologies, and processes as they begin to both build new cloud native applications and refactor and modernize existing applications. Containers are a type of OS level virtualization that provides an isolated, resource-controlled environment to run applications, acting as a very lightweight wrapper around a single Unix process, during actual implementation that process might spawn others processes, even tho one statically compiled binary may be all that's inside the actual container. Container images define how applications are packaged and only contain the application and its dependencies such as libraries, configurations, runtimes, and tools, making a container more lightweight than a VM. The container image and runtime are standardized through the Open Container Initiative (OCI), which makes containers highly portable and universal.

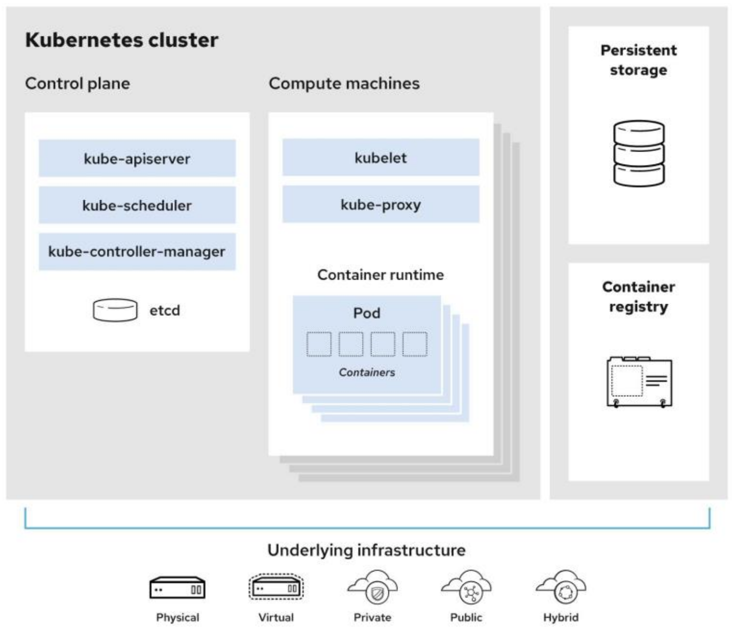
Kubernetes

Kubernetes also known as “k8s”, is an open source community project addressing container orchestration, grouping containers that make up an application into logical units making management and discovery of such components easier, originally developed by Google with more than a decade of experience deploying scalable, reliable systems in containers via application-oriented APIs. It has become the standard API for building cloud-native applications, present in nearly every public/private cloud, it's a proven infrastructure for distributed systems that is suitable for cloud-native developers of all scales.

Kubernetes architecture

At the hardware level Kubernetes is composed of multiple nodes, which can be sorted into two types:

Figure 1 Kubernetes architecture



Master node

The master node provides a running environment for the control plane responsible for managing the state of a Kubernetes cluster, and it is the brain behind all operations inside the cluster. The control plane components are agents with very distinct roles in the cluster's management. In order to communicate with the Kubernetes cluster, users send requests to the master node via a CLI tool, a Web UI Dashboard, or API.(Burns et al., 2022)

The master node is made up of the following components, **API Server**, **Scheduler**, **Controller Manager** and **etcd**.

- **kube-apiserver**, a central control plane component running on the master node. The API server intercepts RESTful calls from users, operators and external agents, then validates and processes them.
- **kube-scheduler**, schedules new objects, such as pods, to nodes. During the scheduling process, decisions are made based on current Kubernetes cluster state and new object's requirements.
- The **kube-controller-manager** are control plane components on the master node running controllers to regulate the state of the Kubernetes cluster. Controllers are watch-loops continuously running and comparing the cluster's desired state (provided by objects' configuration data) with its current state (obtained from etcd data store via the API server). In case of a mismatch, corrective action is taken in the cluster until its current state matches the desired state.
- **etcd**, is a key-value data storage that persistently stores cluster information, it is however a distributed key-value store which only holds cluster state related data, no client workload data.

Worker nodes

A worker node provides a running environment for client applications. Through containerized microservices, these applications are encapsulated in Pods, controlled by the cluster control plane agents running on the master node. Pods are scheduled on worker

nodes, where they find required compute, memory and storage resources to run, and networking to talk to each other and the outside world.
Worker nodes consist of the following components, **kubelet**, **kube-proxy** and **Container runtime**.(Chowdhury, 2020)

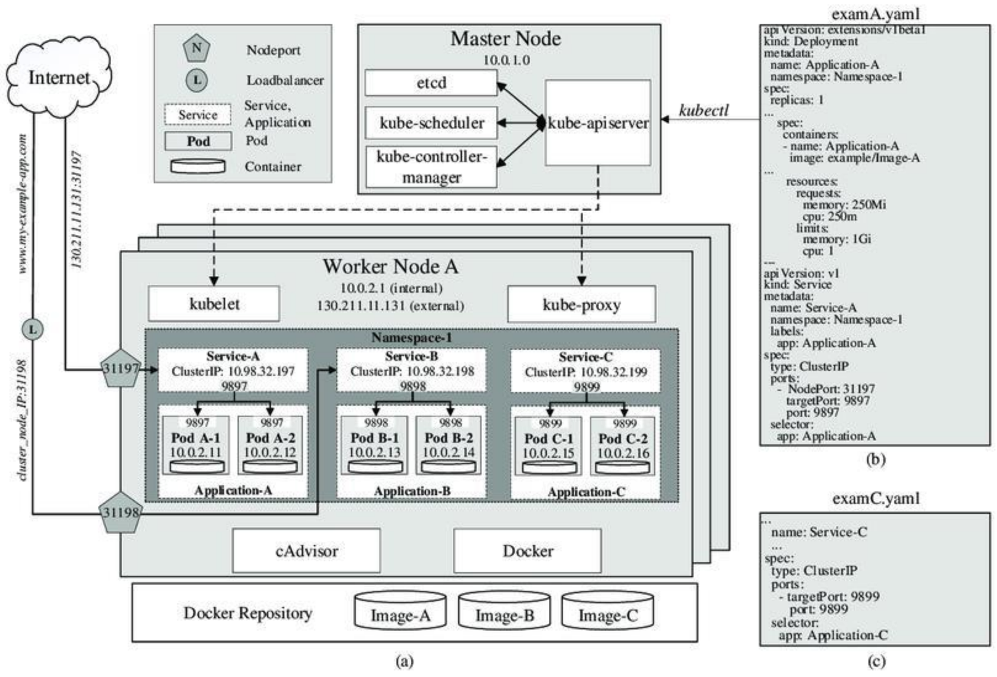
- The **kubelet** is an agent running on each node and communicates with the control plane components from the master node. It receives Pod definitions, primarily from the API server, and interacts with the container runtime on the node to run containers associated with the Pod. It also monitors the health of the Pod's running containers.

- The **kube-proxy** is the network agent which runs on each node responsible for dynamic updates and maintenance of all networking rules on the node, load balancing network traffic between application components. It abstracts the details of Pods networking and forwards connection requests to Pods.

- Even though Kubernetes is described as a ”container orchestration engine”, it does not have the capability to directly handle containers. In order to run and manage a container's lifecycle, Kubernetes requires a **container runtime** on the node where a Pod and its containers are to be scheduled. An example of a container runtime would be Docker.

Overview of Kubernetes cluster

Figure 2 Overview of Kubernetes cluster



```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: Application-A
  namespace: Namespace-1
spec:
  replicas: 1
...
    spec:
      containers:
      - name: Application-A
        image: example/Image-A
...
        resources:
          requests:
            memory: 250Mi
            cpu: 250m
          limits:
            memory: 1Gi
            cpu: 1
---
apiVersion: v1
kind: Service
metadata:
  name: Service-A
  namespace: Namespace-1
  labels:
    app: Application-A
spec:
  type: ClusterIP
  ports:
    - NodePort: 31197
      targetPort: 9897
      port: 9897
  selector:
    app: Application-A
```

```
...
  name: Service-C
  ...
spec:
  type: ClusterIP
  ports:
    - targetPort: 9899
      port: 9899
  selector:
    app: Application-C
```

A Kubernetes application description is posted to the API server after the application is packaged into one or more container images and pushed to an images registry, it includes information on the container image of the application or images that contain the application's components, how those components relate to each other, which ones need to be co-located, the number of replicas to be run, contains also information on which components provide a service to either internal or external clients and should be exposed through a single IP address and made discoverable to other components, the scheduler then allocates the specified onto the available worker nodes based on computational resources.

Given the increase in services over the network via APIs, these APIs are often delivered by a distributed system, the various pieces that implement the API running on different machines, connected via the network and coordinating their actions via network communication. They cannot fail, even if part of the system crashes or otherwise stops working, likewise, they must maintain availability even during software rollouts or other maintenance events.

Declarative configuration

Immutability, everything in Kubernetes is a declarative configuration object representing the desired state of the system. It is the job of Kubernetes to ensure that the actual state of the world matches this desired state.

Mutable versus immutable infrastructure, declarative configuration is an alternative to imperative configuration, where the state of the world is defined by the execution of a series of instructions rather than a declaration of the desired state of the world. While imperative commands define actions, declarative configurations define state.

The effects of a declarative configuration can be understood before they are executed, making it far less error-prone. The idea of storing declarative configuration in source control is often referred to as "infrastructure as code". Combining the declarative state stored in a version control system and the ability of Kubernetes to make reality match this declarative state makes rollback of a change trivially easy.

Self-healing

When a desired state configuration is received, the self-healing system doesn't simply take a set of actions to make the current state match the desired state once. It continuously takes actions to ensure that the current state matches the desired state, this guarantees that Kubernetes initializes the system and also prevents any failures or perturbations that might destabilize the system and affect reliability. Imperative repair that involves human intervention is more expensive, often requiring an on-call operator to be available to enact the repair.

If the desired state of three replicas is asserted to Kubernetes, it does not just create the replicas, it continuously ensures that there are exactly three replicas. If a fourth replica is created manually, it will get destroyed to match the desired state with the current state, on the other hand if a replica is destroyed another replica is created to replace it.

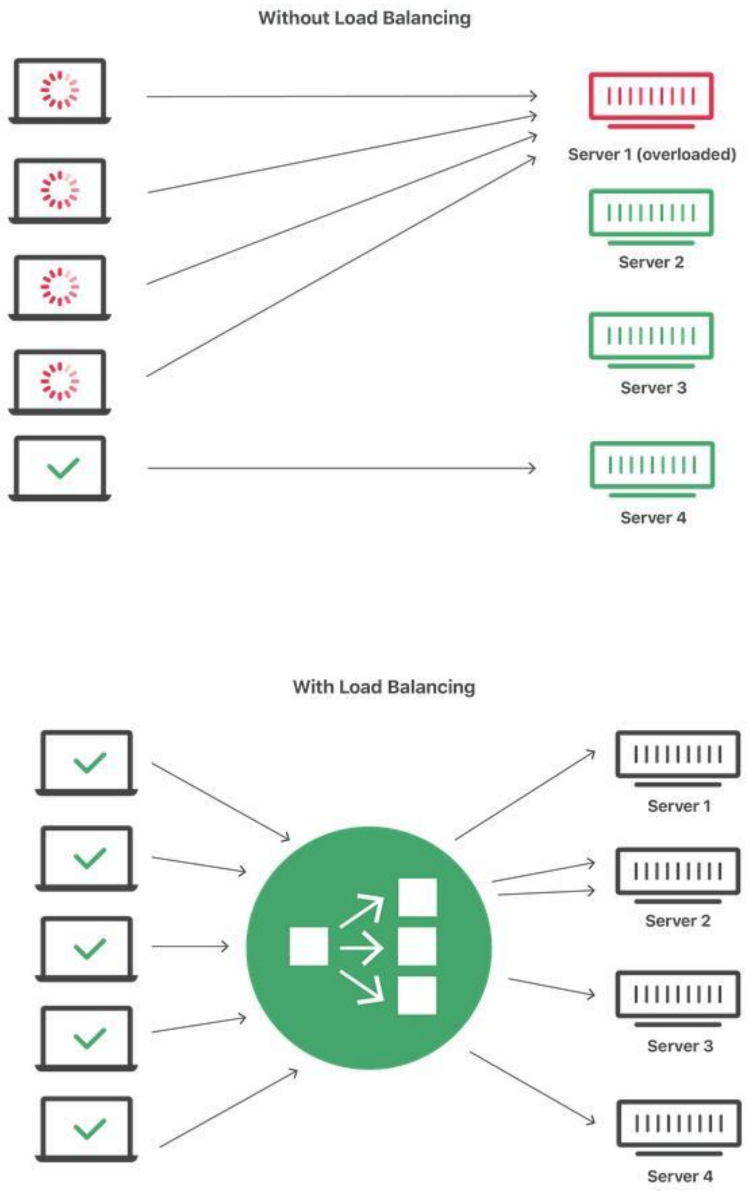
Decoupling

In a decoupled architecture, each component is isolated from other components by defined APIs and service load balancers. APIs and load balancers isolate each piece of the system from the others. APIs provide a buffer between implementer and consumer, and load balancers provide a buffer between running instances of each service.

Load Balancing

Load Balancing distributes computational workloads between two or more computers, often implemented to divide network traffic among several servers, reducing the strain on each server and making the servers more efficient, speeding up performance and reducing latency.

Figure 3 Server load with and without load balancing



A load balancer can be either hardware or software based, hardware-base requiring the installation of a dedicated load balancing device, software-based load balancers can run on a server, virtual machine or in the cloud, when a requests arrives from a user, the requests is assigned to a server by the load balancer, this process is repeated from each request.(Jafarnejad Ghomi et al., 2017) Load balancers determine which server should handle each request based on a number of different algorithms. These algorithms fall into two main categories: static and dynamic.

Static load balancing algorithms distribute workloads without taking into account the current state of the system, meaning it won't be aware of which servers are performing slowly and which servers are not being used enough, instead assigning workloads based on a predetermined plan. Round robin DNS and client-side random load balancing are two common forms of static load balancing.

Dynamic load balancing algorithms take the current availability, workload, and health of each server into account, being able to shift traffic from overburdened or poorly performing server to underutilized servers, maintaining an even and efficient distribution, these perks come at a cost of complexity at setup time, also different factors play into server availability. Decoupling components via load balancers makes it easy to scale the programs that make up your service, because increasing the size of the program can be done without adjusting or reconfiguring any of the other layers of your service.
Decoupling via APIs makes it easier to scale the development, this is because each team can focus on a single, smaller microservice with a comprehensible surface area.

Scaling

Given the declarative and immutable nature of Kubernetes, scaling is trivial to implement, since containers are immutable and the number of replicas is merely a number in a declarative configuration file, scaling upwards is a simple matter of changing a number in a configuration file, asserting this new state. This assumes that the cluster actually has resources available to be scaled up, sometimes scaling of the cluster itself is needed.

Infrastructure abstraction

Application-oriented containers APIs like Kubernetes, have two concrete benefits, the first one being that it separates developers from specific machines, making machine-oriented IT role easier, since machines can simply be added in aggregate to scale the cluster, and in the context of the cloud it also enables a high degree of portability since developers are consuming a higher-level API that is implemented in terms of specific cloud infrastructure APIs. When developers are building their applications in containerized images and deploying them in terms of portable Kubernetes APIs, transferring that applications between environments, or even running in hybrid environments, is simply a matter of sending the declarative configuration to a new cluster.

From Monolithic to microservices

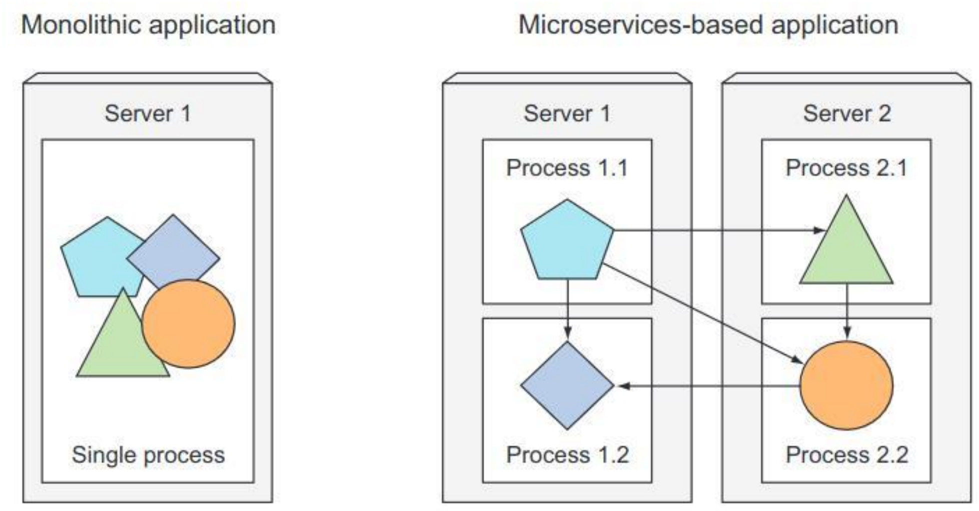
Most software applications were big monoliths, running either on a single process or as a small number of processes spread across a handful of servers, these legacy systems are still in use today, the release cycles of these softwares are very slow and infrequently, on each release cycle, developers package up the whole system and send it over to the ops team, who deploys and monitors, these big monolithic legacy applications are being slowly broken down into smaller, independently running components called microservice, since microservices are decoupled from each other, they can be developed, deployed, updated, and scaled individually. As these deployable components grow, it becomes increasingly more difficult to configure, manage and keep the system running, Kubernetes comes in to solve this problem, it introduces automation, which handles automatic scheduling of components in servers, automatic configuration, supervision, and failure-handling, this helps both developer and operations teams, developers can deploy their applications themselves without requiring any assistance from the ops team, and the ops team can automatically monitor and reschedule apps in the event of hardware failures, shifting the focus from supervising individual apps to supervising and managing Kubernetes and the rest of the infrastructure.(Hamzayev, 2023) The hardware abstraction exposes the whole datacenter as a single enormous computational resource, this is great for very large data centers, such as the ones built and operated by cloud providers, Kubernetes allows them to offer developers a simple platform for deploying and running any type of application, while not requiring the cloud provider's own sysadmins to know anything about the thousands of apps running on their hardware.

Monolithic applications consist of components that are all tightly coupled together and have to developed, deployed, and managed as one entity, because they all run as a single OS process, changes to any part of the application requires a full redeployment of the entire application, over time the increase of complexity and consequential deterioration of the quality of the whole system is inevitable because of the unconstrained growth of inter-dependencies between these parts. On the hardware side running a monolithic application requires a small number of powerful server to provide enough resources for running this application, there a two ways to manage increasing loads on the system, vertical scaling the servers(scaling up), which involves adding more CPUs, memory and other server components, or scale the whole system horizontally.(Harris, 2022)

Splitting Applications into microservices

Given the problems mentioned above regarding monolithic systems, a solution is splitting them into smaller independently deployable components called microservices, each running as an independent process and communicating with other microservices through simple, well-defined interfaces (APIs).

Figure 4 Components inside a monolithic application vs standalone microservices



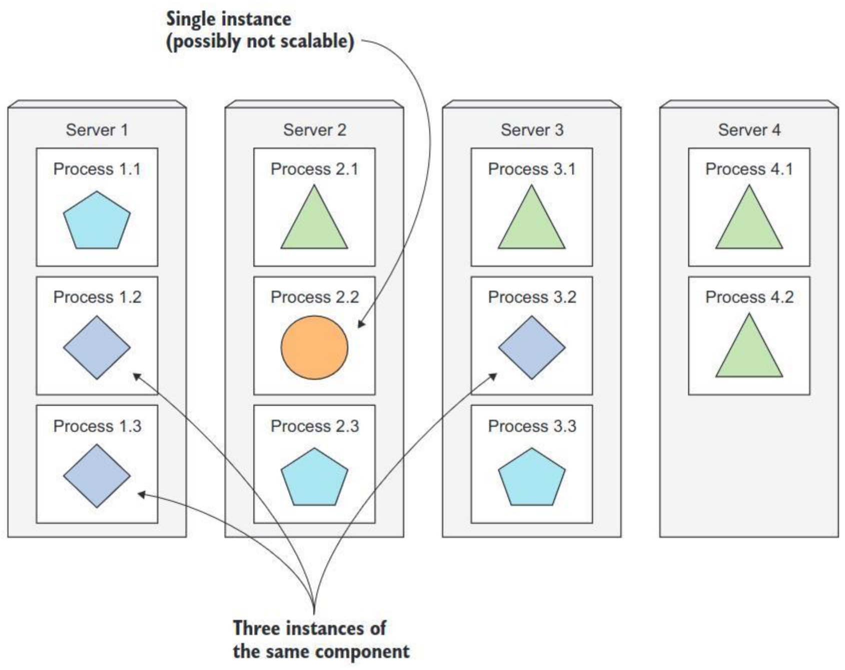
Communication is done through synchronous protocols such as HTTP, over which they usually expose RESTful APIs, or through asynchronous protocols. Since each microservice is a standalone process with a relatively static external API, it's possible to develop and deploy each microservice separately, changing one of them doesn't require changing or redeploying any other service, assuming the API doesn't change.

Scaling microservices

Microservice scaling is done on a per-service basis, meaning that there is an option of scaling only those services that actually require more resources, while leaving others at their original scale, this differs from monolithic that requires that the system as a whole is scaled.

Some components are replicated and run as multiple processes deployed on different servers, while others run as a single application process, monolithic apps that can’t be scaled out because one of its parts is unscalable, splitting the app into microservices allows you to horizontally scale the parts that allow scaling out, and scale the parts that don’t vertically instead.

Figure 5 Each microservice can be scaled individually



DevOps

The development process of an application is rapidly changing, the traditional way of deployment has the development team create the application and hand it off to the operations team, who then deploy it, tend to it and keep it running, but organizations are realizing it’s better to have the same team that develops the application also take part in deploying and managing it over its whole lifetime, this means the developer, QA and operations teams now need to collaborate throughout the whole process, this practice is called DevOps.(Erich et al., 2014) Having developers more involved in running the application in production leads them to having a better understanding of both the user's needs and issues faced by the ops team while maintaining the app. Ideally developers deploy applications themselves without knowing anything about the hardware infrastructure and without dealing with the operations team, this is referred to as NoOps, Kubernetes allows this to be achievable by abstracting away the actual hardware and exposing it as a single platform for deploying and running applications, allowing developers to configure and deploy applications without assistance from the sysadmin, in turn letting the sysadmins focus on keeping the underlying infrastructure up and running, while being abstracted from the applications running on top of it.

Health checks, self-healing and Auto-Scaling

Kubernetes monitors application components and also the nodes those components run on, automatically rescheduling them to other nodes in the event of node failure, this frees the operations team from having to do it manually.

Managing deployed applications means the operations team doesn’t need to constantly monitor the load of individual applications and react to sudden load spikes, Kubernetes can monitor the resources used by each application and keep adjusting the number of running instances of each application, scaling the infrastructure is even easier if Kubernetes is running on cloud infrastructure, adding a node can be done by just requesting them through the cloud provider’s API.

Analyzing the benefits of Kubernetes

Containerized applications already contain all the necessary dependencies to make it run, therefore the operations team doesn’t need to deal with deploying the application anymore or installing anything the application might require, as mentioned previously Kubernetes exposes all its worker nodes as a single deployment platform, so application developers can deploy application while being abstracted of the underlying infrastructure that makes up the cluster, all nodes are grouped as a single computational resource.

CONTAINER TECHNOLOGIES

Faced with the list of issues presented previously by today's development and operations teams, Kubernetes proposes a way of dealing with them which involves the use of containers, using Linux container technologies to provide isolation of running applications.

Delving into the Realm of Containers

When an application is made up of only a small number of large components, it's acceptable to give a dedicated VM to each component and isolate their environment by providing each of them with their own operating system instance, however it's not acceptable from a financial point of view to give a VM to each of these components when they start getting smaller and their numbers start to grow, this leads to not only to financial and hardware resources waste but since each VM usually needs to be configured and managed individually, rising the number of VMs also leads to wasting human resources, increasing the sysadmins workload considerably. Instead of using VMs to isolate the environments of each microservice, developers use Linux container technologies, allowing them to run multiple services on the same host machine while not only exposing a different environment to each of them, but also isolating them from each other, similar to VMs but with much less overhead, a process running in a container runs inside the host's operating system like all the other processes, unlike VMs where processes run in a separate operating system, but the process in the container is still isolated from the other processes, from the 'point-of-view' of the process itself, it looks like it's the only one running on the machine/operating system.(Bernstein, 2014)

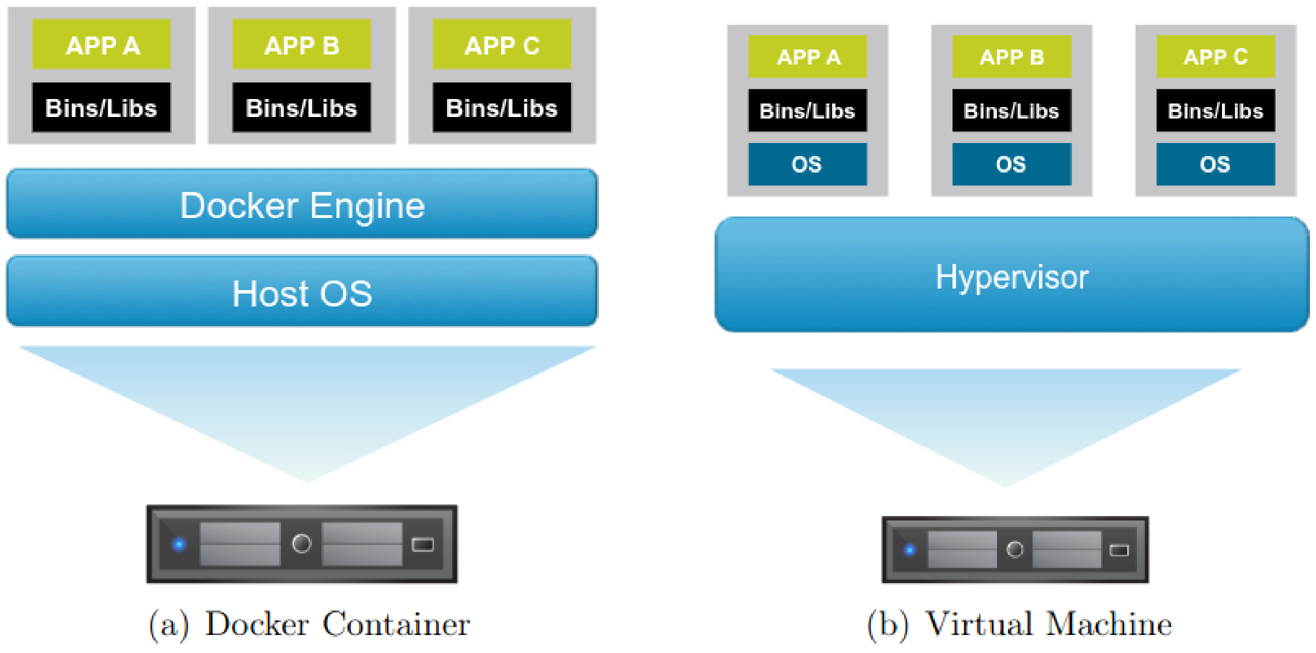
Containers vs. Virtual Machines

It is evident that virtual machines and containers are distinct products that aren't necessarily direct competitors. Each caters to a slightly different audience. Since Docker shares the kernel with the host, it only supports Linux-based containers and immediately discards support for legacy enterprise software that might need a Windows environment to run.

Central to the distinction between container and virtual machine is the tradeoff between density and isolation. Virtual machines offer the strongest form of isolation, comparable to that offered by physically separated hosts. Containers, on the other hand, share the kernel with the host, and therefore provide a much larger attack surface through which an attacker might be able to compromise another container on the same host.(Buchanan, 2023). By giving up isolation, unlike virtual machines which incur a performance overhead, containers achieve near native performance as compared to running directly on the host itself(Arango et al., 2017). Furthermore, the density of containers on a given host can be much higher than that of VMs.

Since Docker can run directly inside of a virtual machine, but not vice versa, there are interesting ways in which Docker and VMware VMs might be able to work together in the future to offer a streamlined experience that captures the benefits of each.

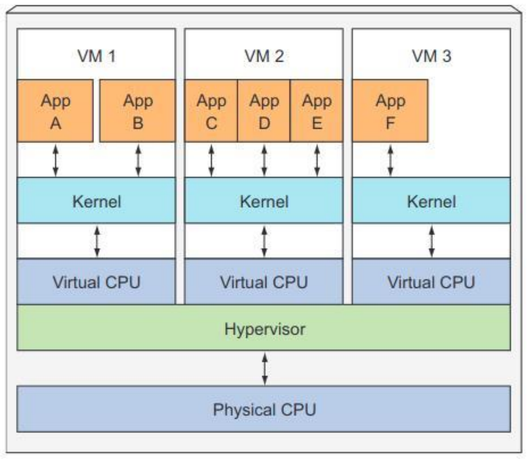
Figure 6 A comparison of the components and isolation of virtual machines as compared to Docker containers. Each container consists of only an application and its dependencies and shares the underlying kernel with the host OS

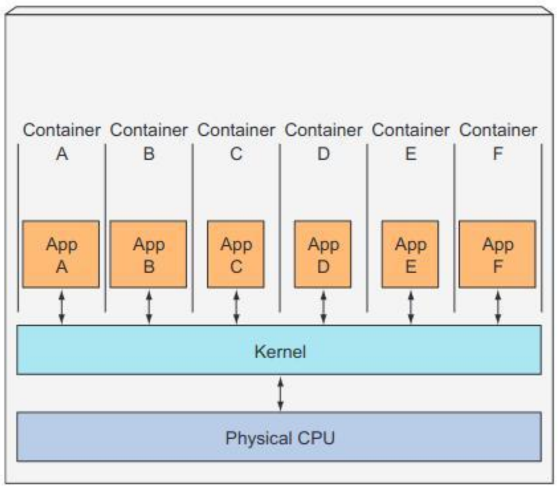


Another distinct advantage of VMs is the ability to perform live migrations from one host to another. VMware dubs this process vMotion, and it is used as a direct building block for VMware's Dynamic Resource Scheduling (DRS)(*VMware Infrastructure: Resource Management with VMware DRS*, 2019). Docker containers, unless running in a VM (and migrated with vMotion), cannot be live migrated to another host and instead must be shut down, moved, and started back up. CRIU is a project under heavy active development that attempts to bring this live migration to the container ecosystem by implementing checkpoint/restore functionality for Linux in userspace. Once completed, CRIU might be able to be used as a primitive for dynamic resource scheduling among different Docker hosts. Looking at Figure 5(a), which has a VM on a host, meaning that a completely separate OS is running and sharing the same bare-metal hardware, underneath that VM there is the host's OS and a hypervisor, which allocates the physical hardware resources into smaller sets of resources that can be used by the VM, applications running in that VM perform system calls to the guest OS kernel in the VM, and then the kernel performs x86 instructions on the host's physical CPU through the hypervisor.

The benefit VMs have over containers is a security one, since VMs have full isolation because they run their own Linux kernel, while containers all call out to the same kernel which can pose a security risk.

Figure 7 Difference between how applications in VMs use the CPU versus how they use them in containers





DOCKER

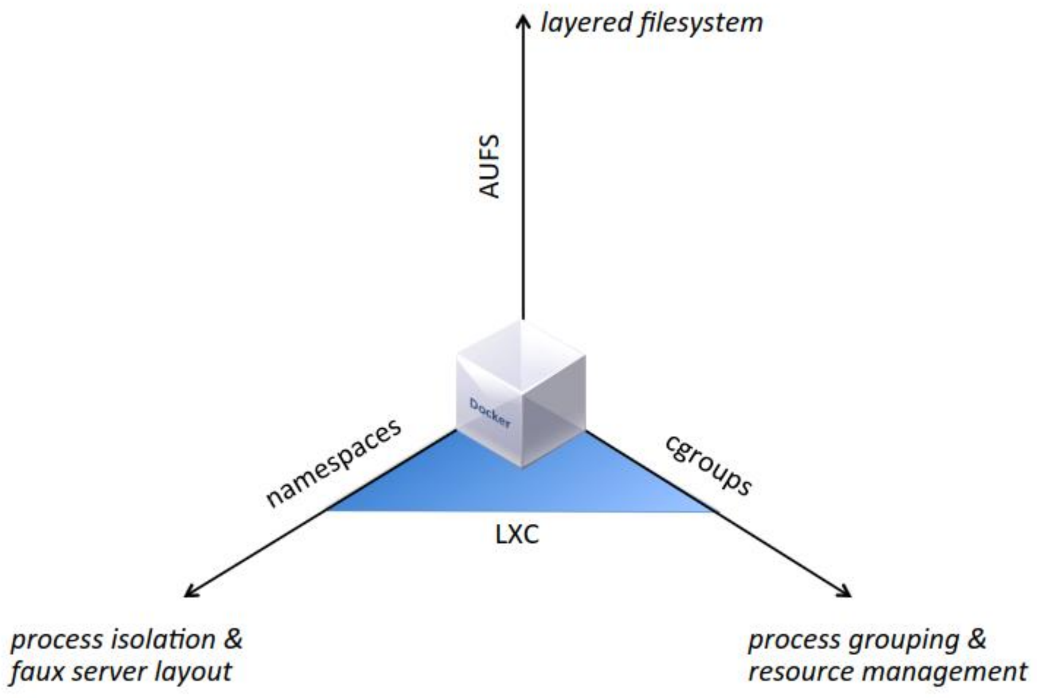
Docker is an open-source tool that allows automation of deployment, scaling, and management of applications using containerization. It provides a way to package software and its dependencies into a standardized unit called a container. Docker containers are based on the concept of operating system-level virtualization, where each container shares the host system's kernel but runs as an isolated process, enabling consistent and reproducible application deployments across different environments. Containers can be built from pre-defined images, the creation of custom images using other Dockerfiles is also possible. Docker images are read-only templates that serve as blueprints for creating containers. They can be shared and distributed through Docker registries like Artifactory, allowing developers to easily access and deploy applications in a consistent manner. It also provides a command-line interface (CLI) and a rich set of APIs that enables developers and system administrators to manage containers and their lifecycle. Containers can be started, stopped, paused, and restarted, allowing for flexible application development, testing, and deployment workflows.(Nickoloff & Kuenzli, 2016)

Containers can run on any system that has Docker installed, regardless of the underlying operating system or hardware infrastructure, this portability makes it easy to develop and deploy applications in diverse environments, from development machines to production servers and cloud platforms. By using Docker, benefits such as improved application scalability, resource efficiency, and faster deployment cycles are easily achievable. It promotes the use of microservices architecture, where complex applications are broken down into smaller, loosely coupled components that can be individually containerized and scaled independently.

Docker System

The Docker platform is divided into the Docker Engine, which supports the runtime and execution of containers, and the Docker Registry, which provides the hosting and delivery of a repository of Docker images. Each container provides a namespace, isolated environment for execution. Docker exploits filesystem layering, as well as specific features of the Linux kernel to make all of these possible.

Figure 8 Fundamental components of Docker framework. Namespaces and cgroups are provided by LXC kernel extensions, and filesystem layering is provided by AUFS


layered filesystem
AUFS
Docker
namespaces
cgroups
LXC
process isolation & faux server layout
process grouping & resource management

Docker concepts

Docker allows packaging an entire application with its whole environment, this can either be a few libraries that are requirements for running the application or all the files that are usually available on the filesystem of an installed operating system, it is possible to transfer this package to a central repository from which it can then be transferred to any computer running Docker and executed there.(Wang, 2018)

Cgroups

Control groups are a feature on the Linux kernel that provides resource limiting, in the form of memory or disk limits, as well as prioritization of CPU and disk throughput. These features are comparable to those offered by a virtual machine hypervisor to allocate a given amount of memory and CPU, network, and disk priority to a virtual machine.

Namespaces

Namespaces are the mechanism by which each Docker container is isolated from the host and other containers. There are many different namespaces that LXC supports, but perhaps the two most significant ones are the pid and net namespaces. The pid namespace is responsible for giving each container its own isolated environment for processes. A given container can only see and send signals to the processes that are running within the same container. In addition, the net namespace allows different containers to have what appears to be distinct network interfaces, thereby permitting two containers to simultaneously bind to the same port, for example.

Another Union FileSystem(**AUFS**), is the primary means through which Docker achieves both storage savings and faster deployments of containers. Each image inherits from a sequence of other images, up to the base image, and represents the set or sequence of changes on the filesystem. This layering of filesystems and images accomplishes two main benefits. First, it allows for a high degree of storage savings. If two containers are running the same OS and share some libraries and dependencies, the majority of their file systems will only be represented once on disk and are not duplicated.

Second, when downloading and deploying a container, if a host already has previous layers of the file system on which a given container depends, it need only download the incremental changes.

Docker-based container images or typically just called **Images** are what is used to package an application and its environment, containing the filesystem available to the application and other metadata, such as the executable path. A base image is a special kind of Docker image that does not have a parent image. The base image instead represents the set of files that make a given operating system unique, excluding the kernel. Examples of base images are Ubuntu 20.04. Base images do not have a parent and instead fully represent an entire OS on their own. Base images are used as starting points from which all other images can inherit.(Docker, 2023)

Registries are repositories used to store Docker images and ease the sharing of those images between different machines, when an image is built, it can either be run on the current machine or pushed to an image registry and later pulled to another machine and run there, registries can be public or private. Docker provides a public registry to which developers can push their custom Docker images and share their creations with others. It has support for creating private images, but requires the user to pay to have more than one privately hosted image. Docker also has open sourced the Docker Registry to allow for privately hosted registries.

For enterprises, this is a superior solution that allows for easy deployment and configuration of Docker containers across a wide area datacenter.

Containers are regular Linux containers created from a Docker-based container image, a running container is a process running on the host running Docker, it’s completely isolated from both the host and all other processes, it is also resource-constrained.(Moravcik & Kontsek, 2020) The Docker pipeline starts with the developer first building an image and then pushing it to a registry, the image is then available to anyone who can access the registry, the image can be pulled to any other machine running Docker and then be run, Docker creates an isolated container based on the image and runs the binary executable specified as part of the image.

Figure 9 Docker images, registries, and containers

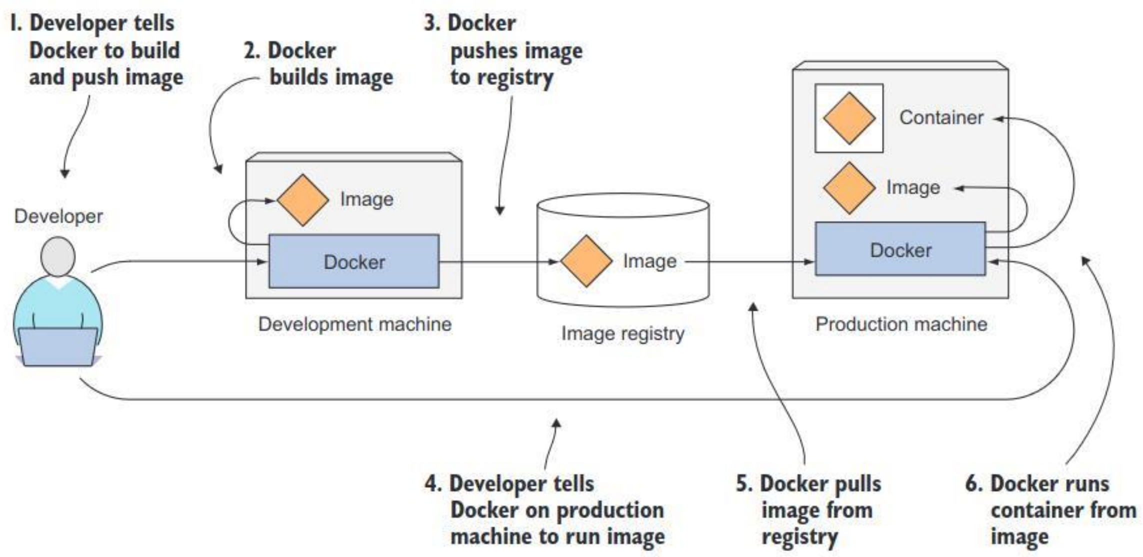
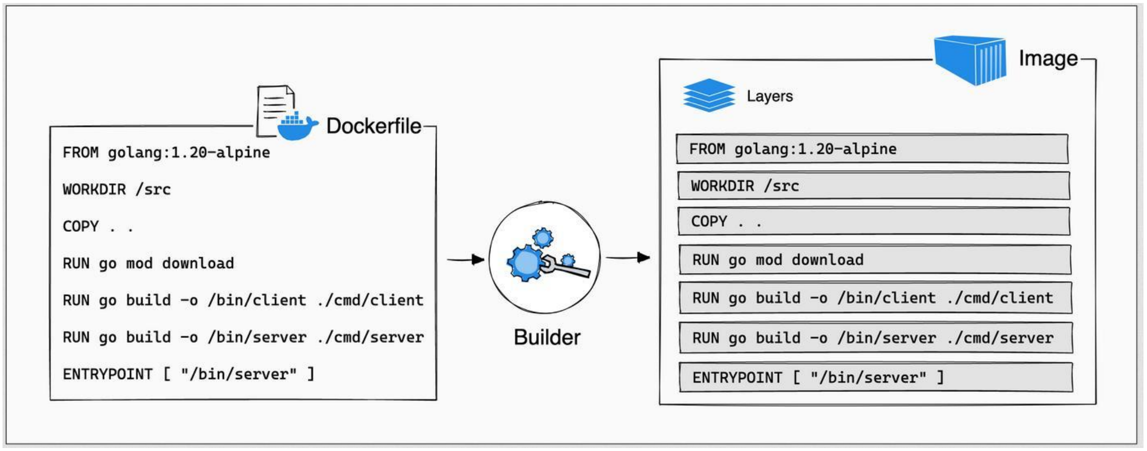

1. Developer tells Docker to build and push image
2. Docker builds image
3. Docker pushes image to registry
Developer
Image
Docker
Development machine
Image
Image registry
Container
Image
Docker
Production machine
4. Developer tells Docker on production machine to run image
5. Docker pulls image from registry
6. Docker runs container from image

Image layers

The order of Dockerfile instructions matters, a Docker build consists of a series of ordered build instructions, each instruction in a Dockerfile roughly translates to an image layer. Different images can contain the exact same layers because every Docker image is built on top of another image and two different images can both use the same parent as their base.

Figure 10 Docker image layering


Dockerfile
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
RUN go build -o /bin/server ./cmd/server
ENTRYPOINT ["/bin/server"]
Builder
Image
Layers
FROM golang:1.20-alpine
WORKDIR /src
COPY . .
RUN go mod download
RUN go build -o /bin/client ./cmd/client
RUN go build -o /bin/server ./cmd/server
ENTRYPOINT ["/bin/server"]

This helps reduce storage footprint of images, each layer is only stored once, if two containers are created from two images based on the same base layers they can therefore read the same files.

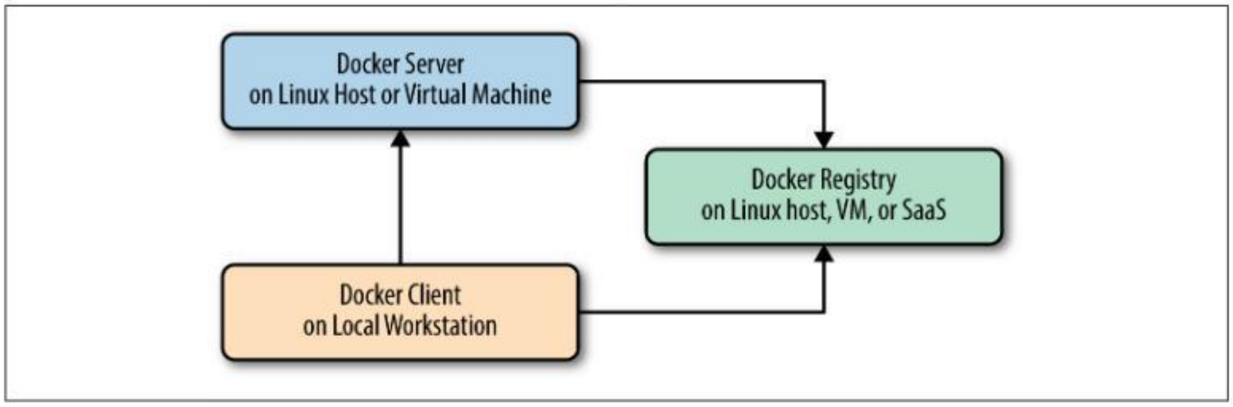
A container image in theory can be run on any Linux machine running Docker, but a small caveat exits, related to the fact that all containers running on a host use the host’s Linux kernel, if a containerized application requires a specific kernel version it might not work on every machine, this is a constraint imposed by containers on the applications running inside them, this does not happened on VMs as each VM has its own kernel, at the cost of a overhead, also a containerized application built for a specific hardware architecture can only be run on other machines that have this architecture, an example would be a containerized application built for x86 architecture and trying to run it on an ARM-based machine, this wouldn't’ be possible, and a VM would be used instead.

Docker Architecture

Fundamentally the architecture of Docker is a simple client/server model, with only one executable that acts as both components, depending on how you invoke the docker command. Underneath this simple exterior, Docker heavily leverages kernel mechanisms such as iptables, virtual bridging, cgroups, namespaces, and various filesystem drivers.

Consisting of at least two parts, the client and the server/daemon, there is also a third optional component called the registry, which stores Docker images and metadata about those images, the server does the ongoing work of running and managing the containers while the client issues commands to the server, the Docker daemon can run on any number of servers in the infrastructure, and a single client can address any number of servers. Clients drive all of the communication, but Docker servers can talk directly to image registries when told to so by the client. Clients are responsible for directing servers what to do, and servers focus on hosting containerized applications.

Figure 11 Docker client/server model

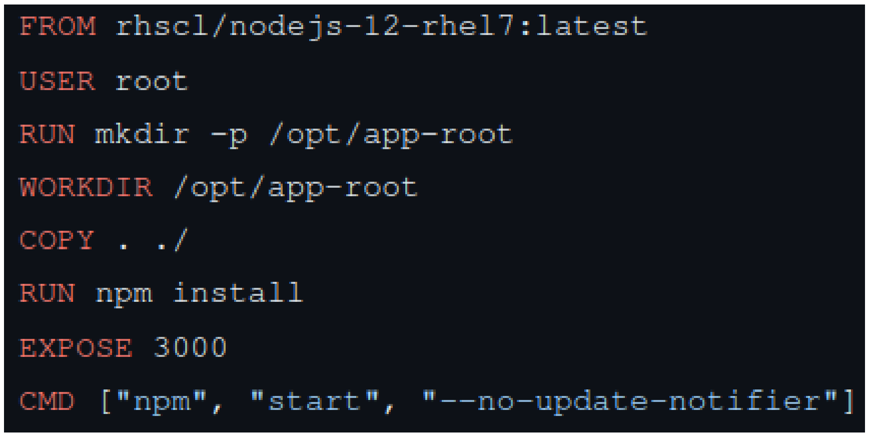


The client/server structure of Docker consists of having separate client and server executables, it uses the same binary for both components, both components are installed even though the server will only launch on a supported Linux host.(Rad et al., 2017) Launching the Docker server/daemon can be accomplished by running the **docker** command-line argument **-d**, which runs Docker as a daemon and listens for incoming connections, each host will usually have one of these daemons running that can manage a number of connections, communications between client and server can then be done via Docker CLI.

Dockerfile

A Dockerfile, similar to a Makefile, is composed of a set of instructions used by Docker to build an image. It typically starts with an inheritance line, specifying from which image to inherit. This can either be a base image, or another image that has been previously built. After the inheritance instruction, the rest of the Dockerfile consists of a combination of commands to run, files to add, environment variables to set, and ports to expose.(Zhang et al., 2018) The Dockerfile can then be passed into the Docker engine, along with an optional tag, and the resulting image is built. Each command in the Dockerfile represents a new layer on the file system. Each change is performed, copy-on-write, such that the entire image ancestry is accessible at any time.

Figure 12 Dockerfile used in this project



```
FROM rhscl/nodejs-12-rhel7:latest
USER root
RUN mkdir -p /opt/app-root
WORKDIR /opt/app-root
COPY . ./
RUN npm install
EXPOSE 3000
CMD ["npm", "start", "--no-update-notifier"]
```

Networking, Ports and Sockets

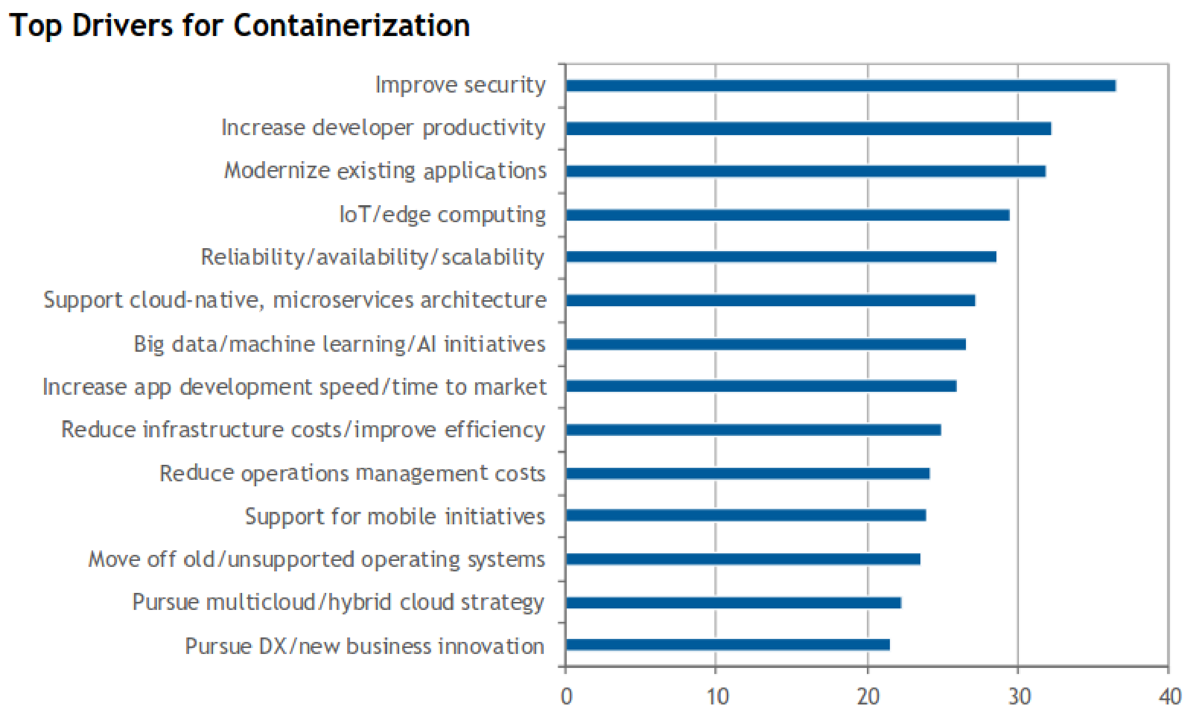
As described previously the Docker server/daemon can be launched using the -d argument on the command-line docker, the command-line tool docker and docker -d, talk to each other over network sockets, the Docker daemon can be setup to listen on one or more TCP or Unix sockets, if Docker is only being accessed from the local system, having it listen only on the Unix socket would be the most secure method, otherwise if remote access is desired having at least one TCP port listening is required.

Docker containers are mostly processes running on the host system, however they behave quite differently from other processes at the network layer, the Docker server acts as a virtual bridge and the containers are clients behind it, a bridge repeats traffic from one side to another, each container has its own virtual Ethernet interface connected to the Docker bridge and its own IP address allocated to the virtual interface, the outside world can reach the container via port binding from the host to the container, this traffic passes over a proxy that is also part of the Docker daemon before getting to the container.

Containers in production environments

The adoption speed of containers in production environments has been greater than any other computer technology, server virtualization took nearly a decade from when it was introduced to a significant number of VMs in production environments, this was due to stability and performance issues. Containers are prevalent in DevOps teams, with 95% of DevOps users already running some number of applications in containers.

Figure 13 Reasons for containerization



The stability and rising trust in containers, is due to some key factors, Linux kernel functions used by containers for execution are very 'mature', the functions are used for functionalities outside of containers, from a fundamental execution standpoint containers are inherently very stable, unlike containers server virtualization has memory overhead due to the fact that it has to emulate hardware, containers work completely differently in this regard and there is no overhead. Given that most container technologies are open source projects with a broad industry participation and high development speed, this allows for much faster software development than any single closed source project.

ADOPTION OF CONTAINERS IN ENTERPRISE SETTING

Speed of software development

With the increasing popularity of cloud native applications and microservices architectures, containers the perfect fit to encapsulate these components into portables units, the lightweight as well as the portability nature of containers makes them great wrappers to push software into these new software pipelines that are being adopted by developers and operations teams of leveraging continuous integration/deployment systems. Existing applications can be containerized without any refactoring, this leads to application portability and also being also to retrofit the application.(Chen, 2019)

Virtualization and Containerization

Virtualization operates at the hardware level, allocating server hardware resources into smaller components used to run multiple VMs, containers operate on the system level and application packaging level. These main differences inherently give VMs some advantages over containers, namely providing a more flexible OS and kernel choice, if we consider a scenario where a container host OS was installed on bare metal, all containers would share the same kernel. On the other hand using VMs this would allow a mix of OSs, be it GNU/Linux or Windows.

Another major advantage VMs have over containers is hardware provisioning, since containers don't manage the underlying infrastructure. However container's key to success are several:

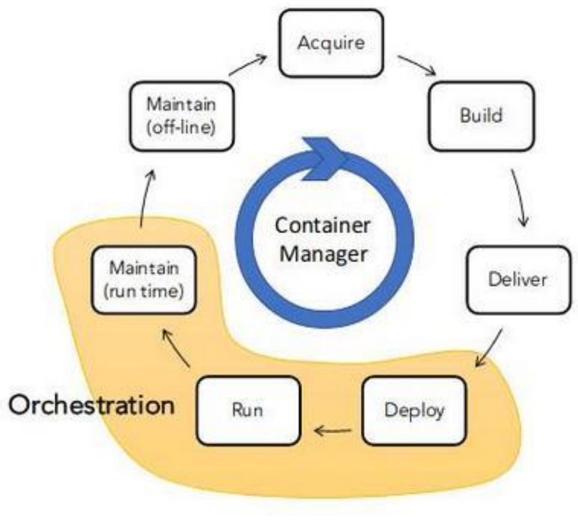
- Easier management of the life-cycle of distributed applications.
- Negligible overhead when running either on bare-metal or VMs.
- Time to start/restart and stop, which is reduced up to an order of magnitude with respect to VMs.
- Application portability, which is where hypervisor-based virtualization failed. Docker and Kubernetes are the de-facto standard container technologies, portability

is also supported by the Open Container Initiative(OCI), which defines a standard image format and a standard run-time environment for containers.

CONTAINER ORCHESTRATION

Container orchestration allows cloud and application providers to define how to select, deploy, monitor, and dynamically control the configuration of multi-container packaged applications in the cloud, the lifecycle management of container workloads, including functions such as to schedule, stop, start, and replicate across a cluster of machines are all elements of container orchestration.(Jawarneh et al., 2019) Computational resources for running workloads are abstracted, allowing the host infrastructure to be treated as a single logical deployment target, orchestration plays a critical role in our design goal of application-centricity as quality of service attributes and deployment patterns are executed by invoking Kubernetes API primitives.

Figure 14 Container lifecycle, provides an API to support at least from the acquire to the run phases, while the orchestrator allows to automate the deployment, run and maintain (run-time) phases



Compute resources for running workloads are abstracted, allowing the host infrastructure to be treated as a single logical deployment target, orchestrators usually offer the following main features:

Resource limit control

Allows allocation of a specific amount of CPU and memory for a container, useful for making scheduling decisions and limiting the resource contention among containers, while a container may use all the resources available in the underlying system, containers managers provide APIs to limit the amount of memory and CPU used.

Scheduling

Defines the policy used to place the desired amount of containers on the desired nodes at a given instant, this can be achieved with resource constraints, or node affinity, or both of them

Load balancing

Does the work of distributing the load among multiple container instances. Round-Robin is the default implemented policy, however more complex policies can be provided by external load balancer.

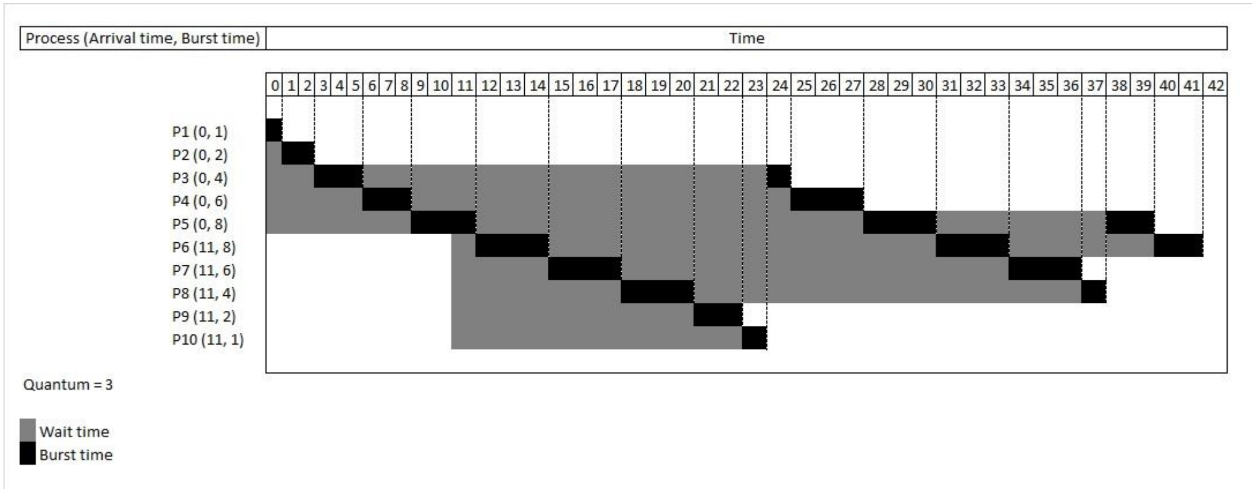
Round-Robin algorithm

This algorithm is implemented by process and network schedulers in computing. Time slices also known as time quanta are assigned to each process in equal portions and in circular order, handling all processes without priority.

A Round-Robin scheduler generally uses time-sharing, giving each job a time slice or quantum(CPU time)(Silberschatz et al., 2010), killing the job if not completed by then, the job can resume next time a slice of CPU time is assigned back to that process. If the job finishes or changes it state to waiting during its attributed time quantum, the scheduler selects the next

job in queue ready to execute, in the absence of a time-sharing approach or if the quanta were large relative to the sizes of the jobs, a process that produced large jobs would be favored over other processes(Stallings, 2015). This makes Round-Robin a pre-emptive algorithm as the scheduler forces the process out of the CPU once the time quota expires.

Figure 15 Round robin preemptive scheduling example



Fault tolerance

This can be implemented as a replica controller and/or high availability controller. Replica control allows to specify and maintain a desired number of containers. Health check is used to determine when a faulty container should be

destroyed and a new one launched to maintain the target number of replicas, the high availability controller allows to configure multiple orchestration managers to always have control on the application in case an orchestrator node fails or is overloaded, the same technique can be used to implement a scalable controller.

Auto-scaling

Allows automatic addition and removal of containers, the implemented policies are usually based on thresholds(CPU and memory usage), it is also possible to define custom auto scaling policies.

OpenShift

OpenShift is a family of containerization software products developed by Red Hat. Its flagship product is the OpenShift Container Platform, an on-premises platform as a service built around Docker containers orchestrated and managed by Kubernetes on a foundation of Red Hat Enterprise Linux.

As the migration project targets the adoption of a containerized environment, it is essential to understand the capabilities and distinctions of OpenShift in comparison to other platforms. OpenShift builds upon Kubernetes but provides additional features and functionalities to enhance container management and application deployment.

One of the primary differentiating factors between OpenShift and Kubernetes lies in OpenShift's ability to provide a complete PaaS experience. OpenShift offers additional layers of abstraction, allowing developers to focus more on the application development and deployment aspects, while abstracting away some of the underlying infrastructure management complexities, moreover Kubernetes alone does not provide any support for building the container image it runs, a build tool is needed to create an application on a separate system and push that to a registry. OpenShift builds on top of that and bundles Kubernetes to implement a PaaS environment.(IBM, 2021)

Another key distinction is OpenShift's focus on enterprise-grade security, compliance, and governance. It provides robust security mechanisms and policy enforcement, ensuring that applications and data remain secure throughout their lifecycle. OpenShift also offers fine-grained access control, allowing administrators to manage user permissions and implement role-based access control (RBAC) policies.

Methodology

The purpose of this study is to evaluate the advantages of migrating an application that is running in VM and being hosted with IBM HTTP Server Apache, to a containerized environment using Redhat Openshift powered by Kubernetes backend, this section provides an overview of the research design, data collection methods, and analysis techniques used in this study.

Project timeline

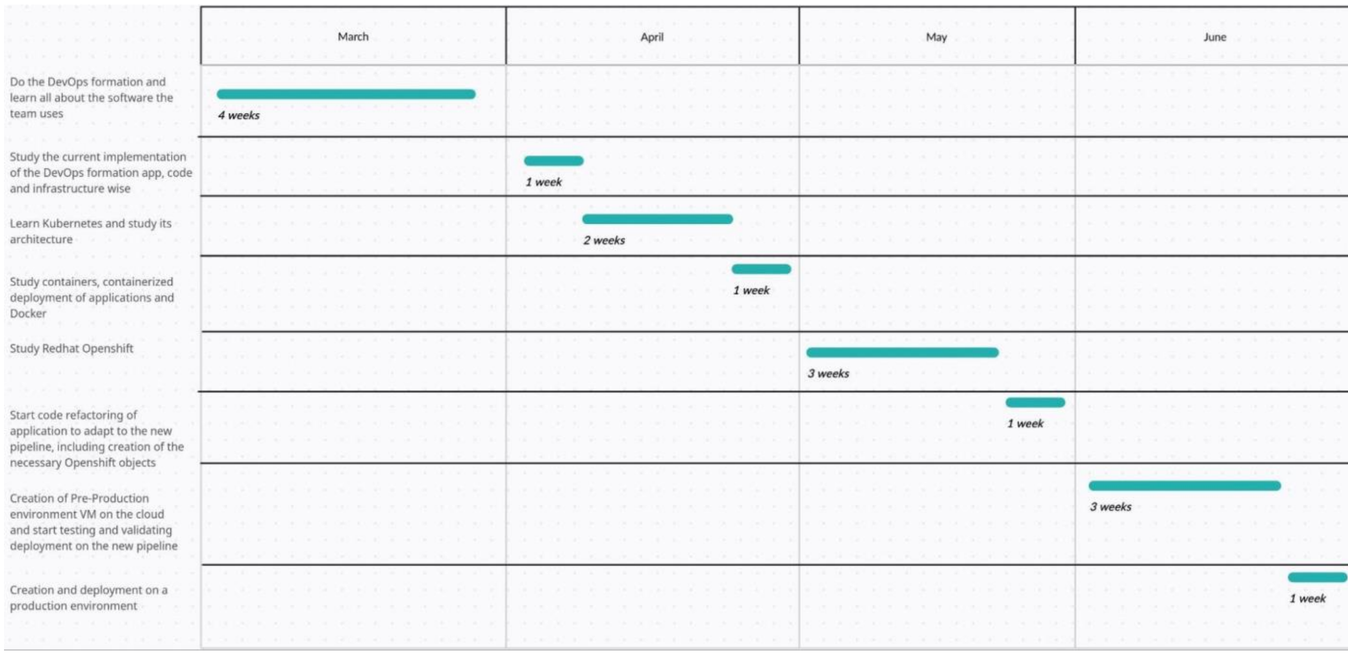
This subchapter presents the project timeline for the migration of DevOps formation application from a virtual machine (VM) environment to a containerized environment using Red Hat OpenShift. The timeline provides a visual representation of the planned schedule, outlining the various tasks, milestones, and dependencies involved in the migration process.

The project timeline, depicted in the Gantt chart shown in Figure 17, serves as a roadmap for the implementation of the migration methodology. It illustrates the sequence and duration of tasks, allowing for efficient planning, execution, and monitoring of the project.

Gantt Chart

Figure 16 showcases the Gantt chart that outlines the project timeline for the migration process. The chart highlights the key activities and their respective start and end dates, as well as the interdependencies between tasks. The duration of each task is represented by the length of the corresponding bar.

Figure 16 Gantt chart



The Gantt chart provides an overview of the migration project's major milestones and critical path. It assists in identifying potential bottlenecks, resource allocation, and tracking progress against the planned schedule.

Project Milestones

The project milestones, as identified in the Gantt chart, represent significant events or achievements in the migration process. These milestones mark key stages in the project and serve as indicators of progress and completion. The following milestones have been identified:

Going through the DevOps formation as a newcomer

As a newcomer to Natixis and to the DevOps domain, it was crucial to gain insights into the tools and practices employed by the team. To accomplish this, a milestone was dedicated to exploring the DevOps formation application, which provided valuable insights into the software utilized and the overall DevOps workflow.

The objective was to familiarize oneself with the software and practices employed by the DevOps team, enabling a better understanding of their requirements, processes, and collaborative workflows. Through this exploration, the researcher aimed to align their approach and leverage appropriate tools and practices for the successful migration project.

Version Control with Bitbucket

A deep exploration of Bitbucket, a popular version control system. Exploring Git repositories, branches, pull requests, and code review workflows, gaining a thorough understanding of how the teams managed their source code and fostered collaboration.

Private Cloud Infrastructure

Study of the private cloud infrastructure used by the teams. This involved understanding the provisioning and management of virtualized resources within the private cloud environment, ensuring efficient utilization and scalability.

Deployment Automation with XLDeploy and XLRelease

In-depth examination of XLDeploy and XLRelease, tools for deployment automation and release management, the concepts of deployment pipelines, release orchestration, and the integration of these tools with the application lifecycle.

Continuous Integration with Jenkins

Exploration of Jenkins, a popular continuous integration tool. Exploring Jenkins capabilities for building, testing, and automating the integration process, ensuring the consistent and timely integration of code changes.

Artifact Management with Artifactory

Study of Artifactory, an artifact management system. Serving as storage, versioning, and distribution of software artifacts, ensuring efficient and reliable management of dependencies and binary artifacts.

Infrastructure as Code

Analysis of Infrastructure as Code practices and tools. This involved understanding the use of tools like Terraform for automating infrastructure provisioning and configuration, enabling consistent and reproducible infrastructure deployments.

Additionally, as an active participant in the DevOps formation application, I had the opportunity to contribute to the existing pipeline. For example, by adding more information to one of the pages, correcting flaws or creating an entirely new page for the application, actively participating in enhancing the application's functionality and content, gaining practical experience in collaborative development and pipeline contribution, for future newcomers.

Analysis of current implementation

Code

The DevOps formation application is very simple code wise, using basic HTML for its pages, and the Skel Javascript framework, a lightweight framework for building responsive sites and web applications, provides a set of CSS and JavaScript modules that assist in creating flexible and adaptive layouts, the code cannot be shown as it is property of Natixis.

Infrastructure

The application has two branches, a development and a production branch, depending on which branch changes are being made the application would either be deployed on the development environment, when adding a new feature, that feature is first pushed and deployed on the development environment, it can after testing and validation be deployed on the production environment.

To accomplish this, two VMs running Linux exist, the development and production environment.

XLDeploy analysis

Managing the infrastructure and application versioning is done via XLDeploy. Two main components need to be configured here, the infrastructure and the environment. Infrastructure points to which host machine to use and also what webserver, in this case a VM running Linux is the host and the web server used is the Apache HTTP Server.
Additionally dictionaries can be created and linked to environments.

XLD Infrastructure

The infrastructure in XLDeploy encompasses various elements such as servers, operating systems, databases, network configurations, load balancers, storage systems, and other hardware or software components that are essential for application deployment and execution. It includes both the physical hardware infrastructure and the logical software infrastructure components. Allowing definition and managing infrastructure configurations as code, providing a systematic and automated approach to infrastructure provisioning and maintenance, configurations can be version-controlled, tracked, and replicated across multiple environments, ensuring consistency and reducing the risk of configuration differential. The automation of infrastructure provisioning and configuration changes, ensures that the infrastructure remains up-to-date and synchronized with the desired state defined in the deployment process.

Treating infrastructure as code, the DevOps team in Natixis can have greater control, consistency, and efficiency in their application deployment processes. They can automate the provisioning and configuration of infrastructure resources, reduce manual effort, minimize errors, and accelerate the overall deployment lifecycle.

XLDeploy Environment

An environment refers to a specific target platform or deployment destination where applications are deployed and managed. It represents a logical grouping of infrastructure resources, configuration settings, and deployment targets that collectively define a particular runtime environment for applications. In the case of the DevOps formation application two environments are defined, the pre-production also known as “bench” and the production environment, these are the target platforms where the application artifacts will be deployed, also provides capabilities to define and manage environment-specific variables, allowing for specify environment configuration values that can be dynamically resolved during deployments.

By using environments, the DevOps team in Natixis can achieve consistent and repeatable deployments across different stages of the software delivery lifecycle, being able to manage the specific infrastructure configurations and deployment settings for each environment, ensuring deployment is done to the appropriate targets with the necessary resources and configurations in place.

XLDeploy Dictionaries

Dictionaries are data structures that behave identically to how a dictionary or hashmap would in a programming language (e.g. Python), storing key value pairs as entries, this is useful because we can essential have placeholders throughout the code, and since each environment will have its dictionary when deploying an application to a certain environment XLDeploy will scan the code and replace all the placeholders of the type {{<KEY>}}, where <KEY> is the actual key in dictionary of the environment on which the application is being deployed, this placeholder is replaced with the respective value. If an application runs on port 3000 but this may not necessarily be true in future where the port number may need to be changed, so instead of having the port number hardcoded on the application's source code there can instead exist a placeholder (e.g. {{APP_PORT}}) and an entry defined on the dictionary's environment holding the value, in this case the port number, this way there is no need to change the source code, only changing the value on the dictionary is needed and re-deploying.

XLDeploy applications

An "application" refers to a software package that is deployed and managed within the deployment automation tool.

It represents the collection of artifacts, components, and configurations that make up a specific software application, this application package is created during the execution of the Jenkinsfile script of the application in the “XLD” stage.

```
stage('XLD'){
        steps{
        xldCreatePackage artifactsPath: '', darPath: 'devopsformation-openshift-${BUILD_NUMBER}.dar', manifestPath: 'deployit-manifest.xml'
                xldPublishPackage darPath: 'devopsformation-openshift-${BUILD_NUMBER}.dar', serverCredentials: '*****'
        xldDeploy serverCredentials: '*****', environmentId: 'Environments/PATH/TO/ENVIRONMENT, packageId: 'Applications/PATH/TO/APPLICATION/${BUILD_NUMBER}'
        }
}
```

Note that some code snippets have been obfuscated for privacy reasons, this will happen throughout this chapter or every time sensitive information appears in the code.

An application in XLDeploy consists of Artifacts, Components and Configuration items. Artifacts are the actual software files, packages, or binaries that constitute the application, including executable files, libraries, configuration files, scripts, database scripts, and other resources required for the application to function. Components represent logical units or modules within the application. They can be defined based on various criteria such as functional areas, modules, or services, allowing for granular management and tracking of different parts of the application during deployment and configuration processes. Configuration items refer to the specific configuration settings, properties, or parameters that are associated with an application. These can include environment-specific configuration files, database connection details, application-specific settings, or any other configuration information needed to run the application correctly.

Kubernetes and it's architecture

This milestone was pivotal as it provides the knowledge to leverage the power of Kubernetes in managing containerized applications effectively, to ensure a successful migration to Red Hat OpenShift, a comprehensive study was undertaken to understand Kubernetes and its architecture. The objective was to gain proficiency in deploying, scaling, and managing applications using Kubernetes' advanced features. Throughout this study, an exploration was done on the various resources and references to grasp the intricacies of Kubernetes. The focus was on comprehending the core components and concepts that constitute Kubernetes' architecture and understanding how they work together to enable efficient container orchestration.

Core Components

A detailed examination of Kubernetes' core components, including the control plane and worker nodes. The researcher learned about the roles and responsibilities of each component, such as the Kubernetes API server, etcd, kube-scheduler, kube-controller-manager, and kubelet, in managing and coordinating containerized applications.

Container Management

Deep exploration of Kubernetes' container management capabilities. The researcher gained an understanding of how Kubernetes leverages container runtimes, such as Docker, to deploy and manage containers. This involved studying concepts like pods, deployments, services, and replica sets, which enable the efficient deployment and scaling of containerized applications.

Scalability and Load Balancing

Investigation of Kubernetes' built-in mechanisms for horizontal scaling and load balancing. The researcher learned how Kubernetes handles auto-scaling of application instances based on resource utilization and implements load balancing across multiple pods to ensure optimal application performance and availability.

Service Discovery and Networking

Understanding Kubernetes' service discovery and networking capabilities. The researcher explored how Kubernetes assigns IP addresses, manages DNS resolution, and enables seamless communication among containers within a cluster. Concepts such as services, ingress, and network policies were thoroughly examined.

Containers, containerized deployment of applications and Docker

To establish a strong foundation for the migration project, a dedicated phase was undertaken to study containers, containerized deployment of applications, and the popular containerization tool, Docker. This milestone was crucial as it provided a comprehensive understanding of the key concepts and technologies underlying the container ecosystem. A systematic study was conducted to explore containers and their role in application deployment. The objective was to acquire a deep understanding of containerization principles and the benefits it offers in terms of portability, scalability, and resource efficiency. Going into the fundamental aspects of containers, including container images, container runtimes, and container orchestration. Additionally, Docker, being one of the most widely adopted containerization platforms, received significant attention throughout the study.

Containerization Concepts

A comprehensive exploration of containerization principles, including the isolation of applications, resource encapsulation, and the sharing of host kernel. Gaining insights into how containers differ from virtual machines and the advantages they offer in terms of performance and resource utilization.

Containerized Deployment

In-depth analysis of containerized deployment methods, focusing on the benefits and considerations when deploying applications within containers. The process of packaging applications into container images and deploying them using containerization platforms like Docker.

Extensive study of Docker, as a leading containerization tool, including its architecture, command-line interface, and key functionalities. Building Docker images, running containers, and utilizing Docker's vast ecosystem of tools and services. Exploration of container orchestration concepts, with a particular emphasis on tools like Kubernetes. Exploring how container orchestration platforms facilitate the management, scaling, and high availability of containerized applications.

Red Hat OpenShift Container Platform

To successfully migrate the application to Red Hat OpenShift, a comprehensive study of the platform was conducted. This milestone was particularly significant as it involved acquiring knowledge and proficiency in working with Red Hat OpenShift Container Platform, considering my initial lack of familiarity with the platform. A dedicated study phase was undertaken to understand the platform's fundamental concepts, functionalities, and best practices. The goal was to gain the necessary expertise to effectively deploy and manage containerized applications. During this study several key areas were covered.

Architecture and Components

In-depth examination of OpenShift's architecture, including its master nodes, worker nodes, and the underlying Kubernetes cluster. Studying the interaction between these components, as well as their roles in supporting container orchestration.

Application Deployment and Management

Detailed exploration of the processes and tools available for deploying and managing containerized applications. This involved understanding concepts such as pods, services, routes, and deployments, and how they facilitate the deployment and scalability of applications.

Resource Allocation and Monitoring

Investigation of resource allocation and monitoring capabilities. Studying how to effectively manage CPU, memory, and storage resources within the platform, as well as how to monitor and troubleshoot application performance.

Code refactoring and adaptation of pipeline, OCP objects creation

One of the essential milestones in the migration process involved code refactoring to enable the proper startup of the application within the containerized environment. In the previous implementation, the application relied on Apache Web Server to automatically start the application on the VM. However, as part of the migration to a containerized environment using Red Hat OpenShift, an alternative approach was required.

To address this, a code refactoring process was undertaken, leveraging Node.js as a replacement for the Apache Web Server. Node.js, a popular JavaScript runtime built on Chrome's V8 JavaScript engine, provided a lightweight and efficient framework to handle the application startup process within the container. The code refactoring involved rewriting the startup logic and integrating it with the Dockerfile's CMD instruction. This allowed the application to be properly initialized and executed when the container is launched. By utilizing Node.js, the refactored code ensured a seamless and consistent startup mechanism in the containerized environment. The other aspect of the code refactoring involved the creation of routes for the different pages of the web app. This entailed implementing routing mechanisms within the Node.js application to handle requests and direct them to the appropriate pages. By establishing these routes, the web app maintained its functionality and ensured users could access different sections or features seamlessly.

The utilization of Node.js for code refactoring brought multiple benefits to the migration process. Firstly, it reduced dependencies on external web servers, streamlining the application's startup process within the containerized environment. The creation of routes within the Node.js application enhanced the app's navigation and user experience, maintaining its functionality and ensuring smooth page transitions.

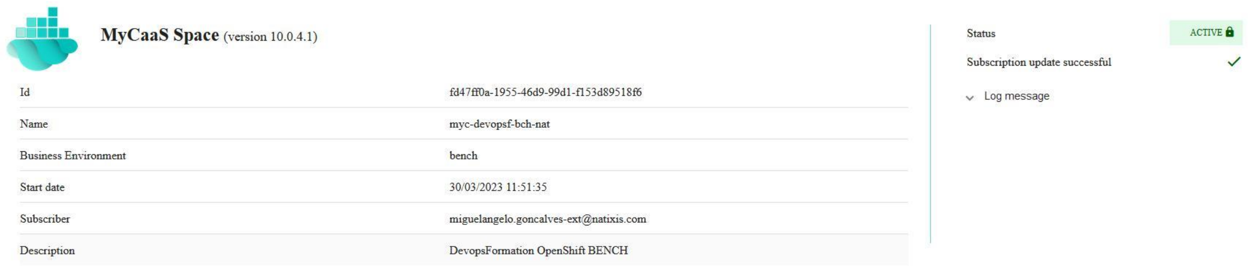
Through this code refactoring milestone, the application was successfully adapted to the container paradigm, enabling proper startup and seamless routing of the web app's different pages within the Red Hat OpenShift platform.

VM provisioning and creation of pre-production

Provisioning is the process of setting up physical or virtual hardware, installing and configuring software, such as the operating system and applications, and connecting it to middleware, network, and storage components. Provisioning can encompass all of the operations needed to create a new machine and bring it to the desired state, which is defined according to business requirements.

Since Red Hat Openshift uses a container based paradigm, where applications are packaged and run within containers, which encapsulate all the necessary dependencies and libraries required for their execution. A CaaS provider is used to host the container orchestration engine running and maintaining the infrastructure’s containers, this service can be accessed via container-based virtualization, API calls, or web portal interfaces. Instead of a VM or bare metal host, the service is offered via a container, which makes scaling easier and deployment faster.

Figure 17 CaaS provisioning



MyCaaS Space (version 10.0.4.1)

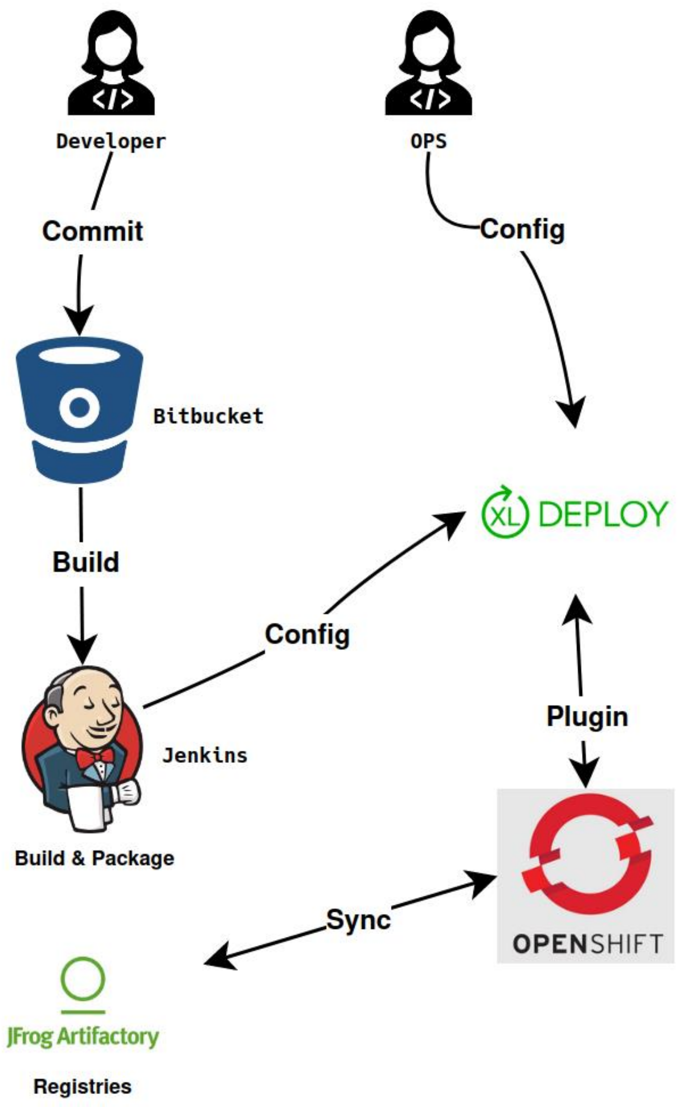
Id	fd47ff0a-1955-46d9-99d1-f153d89518f6
Name	myc-devopsf-bch-nat
Business Environment	bench
Start date	30/03/2023 11:51:35
Subscriber	miguelangelo.goncalves-ext@natixis.com
Description	DevopsFormation OpenShift BENCH

Status	ACTIVE
Subscription update successful	✓

Log message

After studying the current implementation, the goal was to plan out what the new pipeline would look like:

Figure 18 DevOps formation application pipeline



All the source code along with a Jenkinsfile and deployment manifest would still be stored on a Bitbucket repository, where a git push from a developer would trigger the Jenkins build, the build pulls the Jenkinsfile from the SCM and executes all the stages defined in the pipeline:

New Jenkinsfile

The Jenkinsfile's pipeline starts by selecting an agent node that is being assigned to this label, the label is used to specify that a particular stage or step in the Jenkins pipeline should be executed on a node with the "generic-slave" label. This allows for better control and flexibility in distributing the build jobs across different nodes based on their capabilities.

The built-in function cleanWs() is then called, it is used to clean up the workspace of a Jenkins job before or after its execution. The workspace refers to the directory on the Jenkins agent where the job's files and code are stored during the build process. It deletes all the files and directories in the workspace directory, resets the workspace back to its initial state, removing any changes or artifacts generated during previous builds, ensuring that subsequent builds start with a fresh workspace.

Cloning the repository into the Jenkins workspace, which performs a Git checkout operation from a specified repository, The git step is a built-in step provided by the Jenkins Git plugin. It allows you to clone or checkout a Git repository as part of your Jenkins pipeline, the parameters provided to the built-in git step are:

- The branch which is the master branch of the repository.

- The credentials, which specify the ID of the credentials that Jenkins should use to authenticate with the git repository. The actual credentials are configured and stored within Jenkins, and the specified ID references them.

- The specified URL of the Git repository from which the checkout should be performed. In this case, the repository is hosted on Bitbucket at the provided URL.

When this step is executed within the Jenkins pipeline, it will clone the specified Git repository and checkout the master branch using the provided credentials for authentication. This allows subsequent steps in the pipeline to access and work with the code and files from the Git repository.

On the ‘File prep’ stage, the deployit-manifest.xml as well as all the OpenShift object, inside the ocp-templates directory get all the placeholders replaced with the Jenkins built-int variable ${BUILD_NUMBER}, which has the build number of the current build, this is done using sed’s find and replace feature:

sed -i \"s/PKG_VERSION/${BUILD_NUMBER}/g\" deployit-manifest.xml

Taking a look at the above command, sed is used to find and replace PKG_VERSION(acts as placeholder for versioning purposes), on the file deployit-manifest.xml with current build number of Jenkins that is stored in the built-in variable ${BUILD_NUMBER}, /s is used to substitute the found expression PKG_VERSION with ${BUILD_NUMBER}.
/g stands for global, meaning do this for the whole line, the -i flag edits the file directly.

Docker stage, on this stage a login is done into the Docker registry at openshift.artifactory.mycloud.intranatixis.com using the ${USERNAME} and ${USERPASSWORD} variables, this is done so Docker images can be pulled and push from and into the registry, another authentication is done on the myc-docker-openshift-p.artifactory.mycloud.intranatixis.com using the same credentials, this will be the registry where the images will be stored. The image is then built using docker build command, and pushed into the registry using the docker push command, after the image is pushed the image is deleted locally as there is no need to store it in the machine, as a final step docker logout command is used to close the session from both registries, on the XLDeploy stage the dar package is created and published:

docker build -t myc-docker-openshift-p.artifactory.mycloud.intranatixis.com/devopsf:$BUILD_NUMBER .

docker push myc-docker-openshift-p.artifactory.mycloud.intranatixis.com/devopsf:$BUILD_NUMBER

docker rmi myc-docker-openshift-

p.artifactory.mycloud.intranatixis.com/devopsf:$BUILD_NUMBER -f

docker logout openshift.artifactory.mycloud.intranatixis.com

docker logout myc-docker-openshift-p.artifactory.mycloud.intranatixis.com

xldCreatePackage artifactsPath: '', darPath: 'devopsformation-openshift-${BUILD_NUMBER}.dar', manifestPath: 'deployit-manifest.xml'

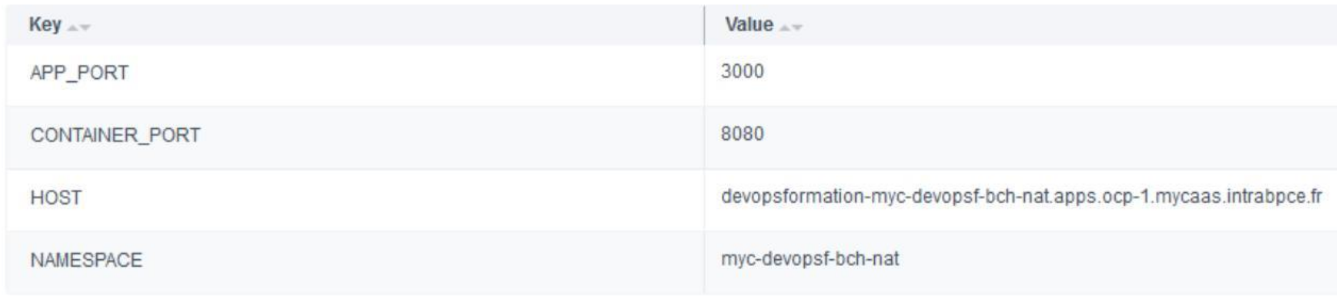
xldPublishPackage darPath: 'devopsformation-openshift-${BUILD_NUMBER}.dar', serverCredentials: '******-***'**

This is all the developer needs to worry about, the operations team is in charge of managing and deploying the application in the pre-production and production environment, as well as managing the infrastructure and dictionaries.

XLDeploy dictionary entries

Dictionaries in XLDeploy are a data structure that behave exactly as a dictionary in Python would, by storing key value pairs of entries, then on the deployment every file that contains a placeholder for the key of the dictionary is replaced by the respective value of that key. Let's consider the dictionary for the devopsformation application:

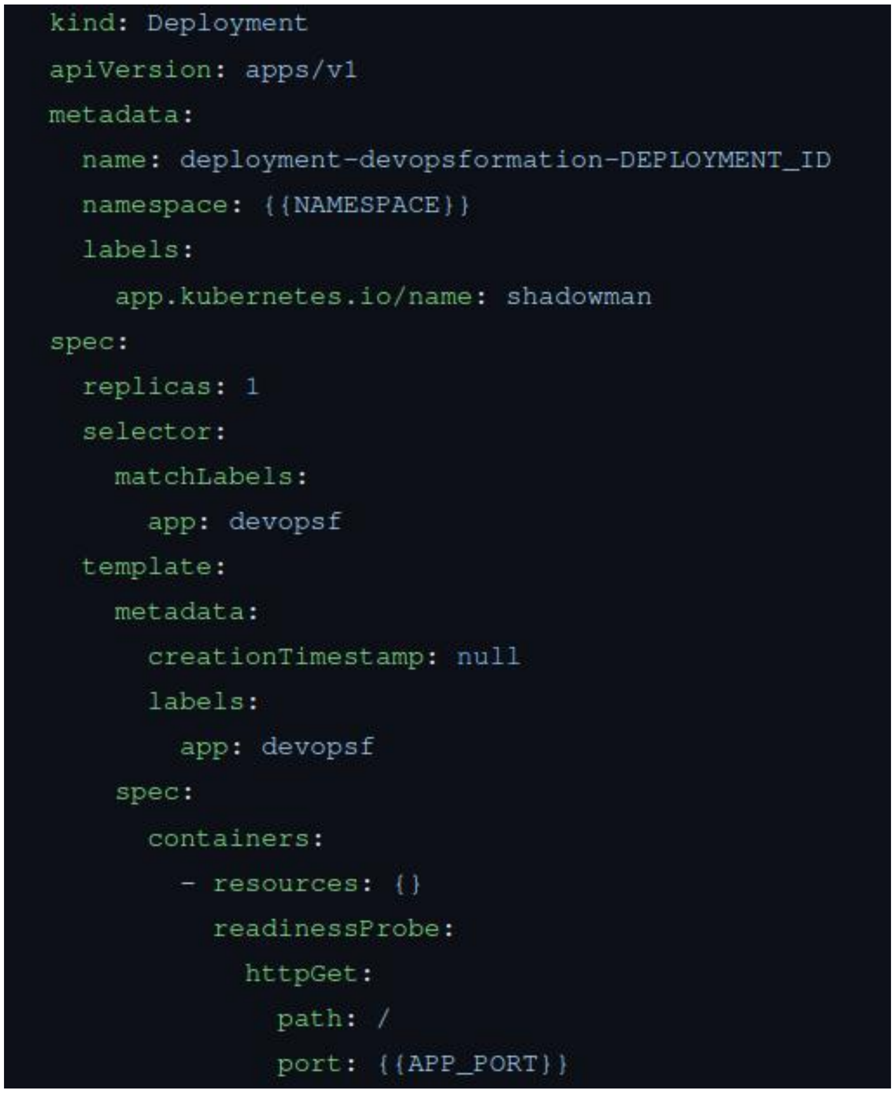
Figure 19 XLDeploy dictionary



Key	Value
APP_PORT	3000
CONTAINER_PORT	8080
HOST	devopsformation-myc-devopsf-bch-nat.apps.ocp-1.mycaas.intrabpce.fr
NAMESPACE	myc-devopsf-bch-nat

Taking the first entry as an example, every file in which a placeholder of the type {{<KEY_NAME>}}, in this case {{APP_PORT}} would get replaced by the value of APP_PORT which is 3000.
The XLDeploy dictionary of the application is used to populate the values in the Openshift object configuration files, such as the deployment yaml file.

Figure 20 OCP-Deployment yaml



```
kind: Deployment
apiVersion: apps/v1
metadata:
  name: deployment-devopsformation-DEPLOYMENT_ID
  namespace: {{NAMESPACE}}
  labels:
    app.kubernetes.io/name: shadowman
spec:
  replicas: 1
  selector:
    matchLabels:
      app: devopsf
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: devopsf
    spec:
      containers:
        - resources: {}
          readinessProbe:
            httpGet:
              path: /
              port: {{APP_PORT}}
```

Jenkins job creating and Webhook configuration

Here is outlined the process of creating and configuring Jenkins jobs to enable automated builds triggered by repository events, specifically git pushes to the application's Bitbucket repository. Setting up webhooks on Bitbucket to facilitate seamless integration between the source code repository and the Jenkins CI/CD system.
To establish the automated build process, a Jenkins job is created tailored to the needs of the project.

Job Definition, the created job is a pipeline job, a pipeline job in Jenkins is based on the concept of a pipeline, which represents a series of stages through which the code progresses before reaching the final deployment or delivery. Each stage in the pipeline represents a phase of the software delivery process, such as building, testing, deploying, and monitoring. The pipeline is defined using a Jenkinsfile, which is a text file written in either Declarative or Scripted syntax. The Jenkinsfile serves as the pipeline's definition and can be stored in the source code repository alongside the application code. It allows definition of the different stages of the pipeline and specifies the actions or steps to be executed within each stage.

Job Configuration, the configuration of each Jenkins job involved specifying the source code repository location, branch filters, build triggers, build steps, and post-build actions. Job configurations were designed to facilitate the smooth integration with Bitbucket and ensure the desired build behavior upon git push events. To trigger the automated builds on Jenkins whenever a git push event occurs on the Bitbucket repository, we configured webhooks on Bitbucket. The following steps were taken for webhook configuration:

Webhook Creation, webhooks were created on the Bitbucket repository settings to send notifications to the Jenkins CI/CD system upon specific repository events, such as pushes to the master and dev branches.

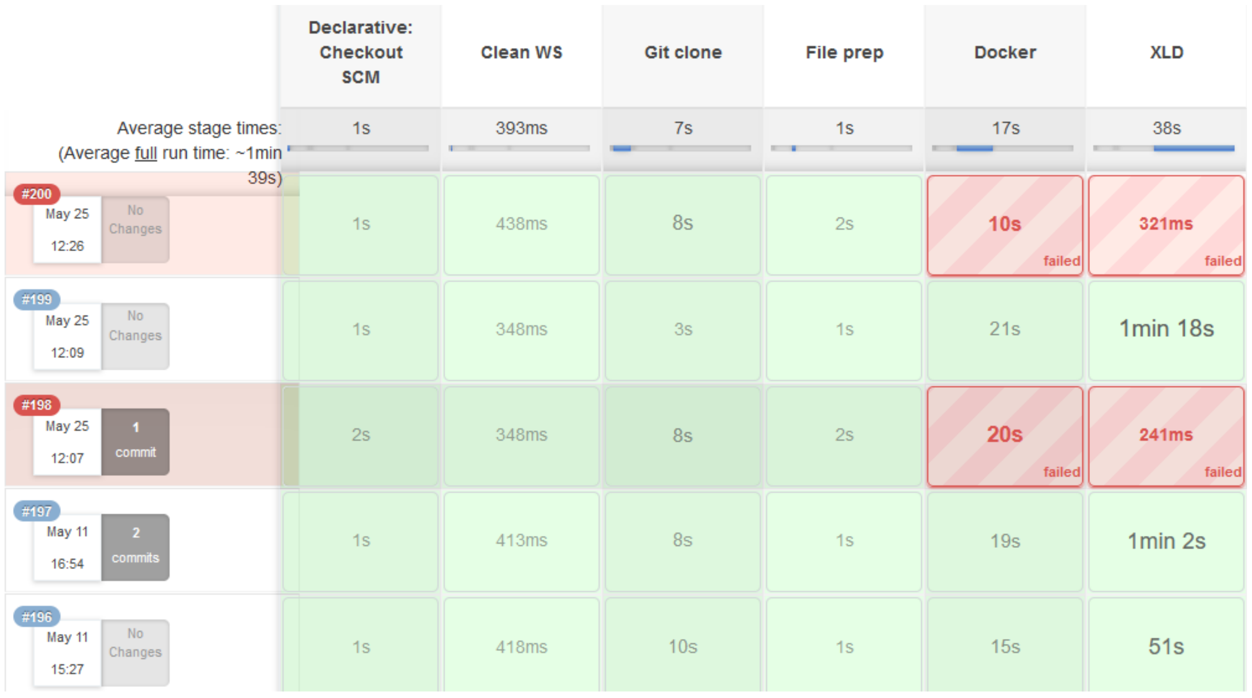
Webhook URL Configuration, the Jenkins webhook URL was configured as the target URL in the Bitbucket webhook settings. This ensured that Bitbucket could send the relevant repository event information to the Jenkins system.

Payload Configuration, the webhook payload was customized to include the necessary details for triggering the corresponding Jenkins jobs. This included information such as the branch name, commit details, and repository URL.

Testing and Verification, testing is done to ensure that the webhooks were properly configured and triggered the corresponding Jenkins jobs upon git push events. This involved pushing code changes to the repository and verifying that the Jenkins jobs were automatically triggered and executed as expected.

The Jenkins job creation and configuration, along with the webhook configuration on Bitbucket, established an automated build process that seamlessly integrated the source code repository and the Jenkins CI/CD system. This ensured that code changes pushed to the dev and master branches triggered the appropriate Jenkins jobs, facilitating continuous integration and enabling rapid and reliable deployments to the respective environments.

Figure 21 Jenkins builds execution status



	Declarative: Checkout SCM	Clean WS	Git clone	File prep	Docker	XLD
Average stage times: (Average full run time: ~1min 39s)	1s	393ms	7s	1s	17s	38s
#200 May 25 12:26 No Changes	1s	438ms	8s	2s	10s failed	321ms failed
#199 May 25 12:09 No Changes	1s	348ms	3s	1s	21s	1min 18s
#198 May 25 12:07 1 commit	2s	348ms	8s	2s	20s failed	241ms failed
#197 May 11 16:54 2 commits	1s	413ms	8s	1s	19s	1min 2s
#196 May 11 15:27 No Changes	1s	418ms	10s	1s	15s	51s

OpenShift Namespace Creation and Configuration

In this section, we delve into the crucial steps involved in setting up the OpenShift namespace and configuring the necessary objects to facilitate the deployment and accessibility of our containerized application. By creating a dedicated namespace and defining the required OpenShift objects, we establish a robust environment for hosting and managing our application within the OpenShift Container Platform.

OpenShift Namespace Creation

The first step in our methodology involved creating a dedicated namespace within the OpenShift cluster to isolate and manage the resources associated with our application. The namespace creation process included the following steps:

Namespace Definition

The name and purpose of the namespace is defined, aligning it with the application's requirements and organizational conventions. This namespace would serve as the containerized environment for hosting our application components.

- devopsf-prd-nat
- devopsf-dev-nat

Devopsf: Name of the application

prd/dev: Refers to the environment type, Production or Development respectively.

Namespace Creation: Using the OCP command-line tools or web console, the namespace is created within the OpenShift cluster. This process involved specifying the desired namespace name and ensuring appropriate access controls and resource quotas were set.

Namespace Configuration: The newly created namespace is configured, setting resource limits, access permissions, and any specific policies required for the application. This ensured that the namespace operated within the defined boundaries and adhered to the desired security and resource allocation constraints.

Creation of OpenShift Objects

To deploy the application within the OpenShift namespace, various OpenShift objects were created, including deployments, services, and routes. These objects were essential for managing the lifecycle of the containerized application and enabling external access.

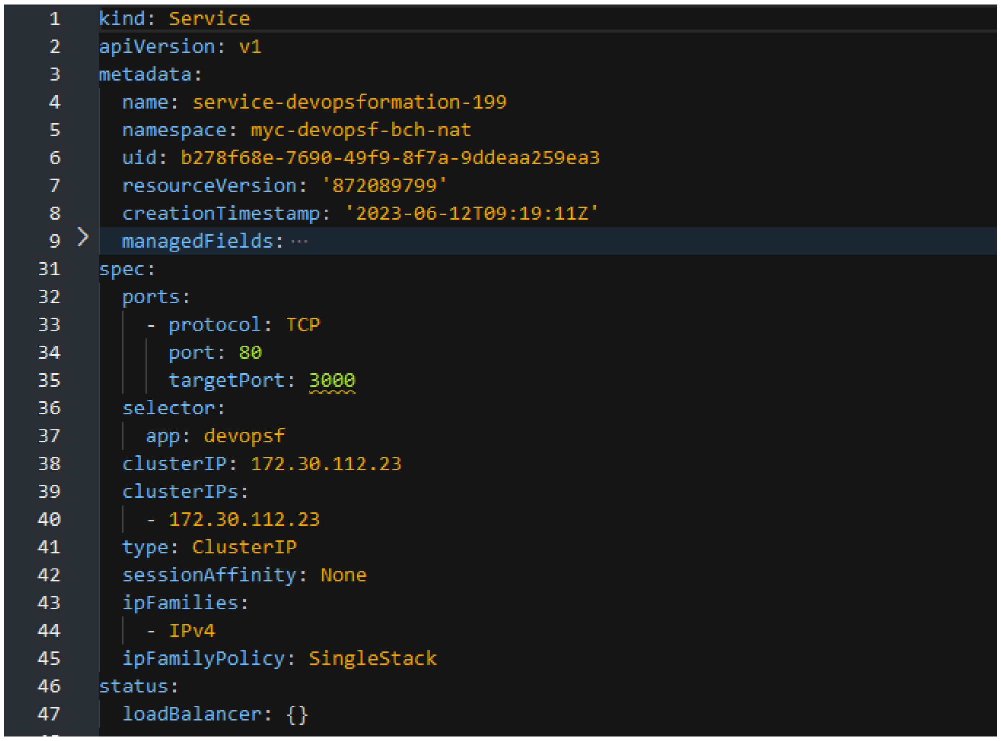
Deployment Configuration

Defines the deployment configuration for the application, specifying the container images, environment variables, resource requirements, and any necessary initialization or post-deployment actions. This configuration ensures the proper deployment and execution of the application within the OpenShift cluster.

Service Creation

Services expose the application internally within the cluster. Services allowed other components within the cluster to access the application using a stable, internal network endpoint, defining service ports and selectors to establish the connection between the service and the application's deployment.

Figure 22 OCP-Service yaml

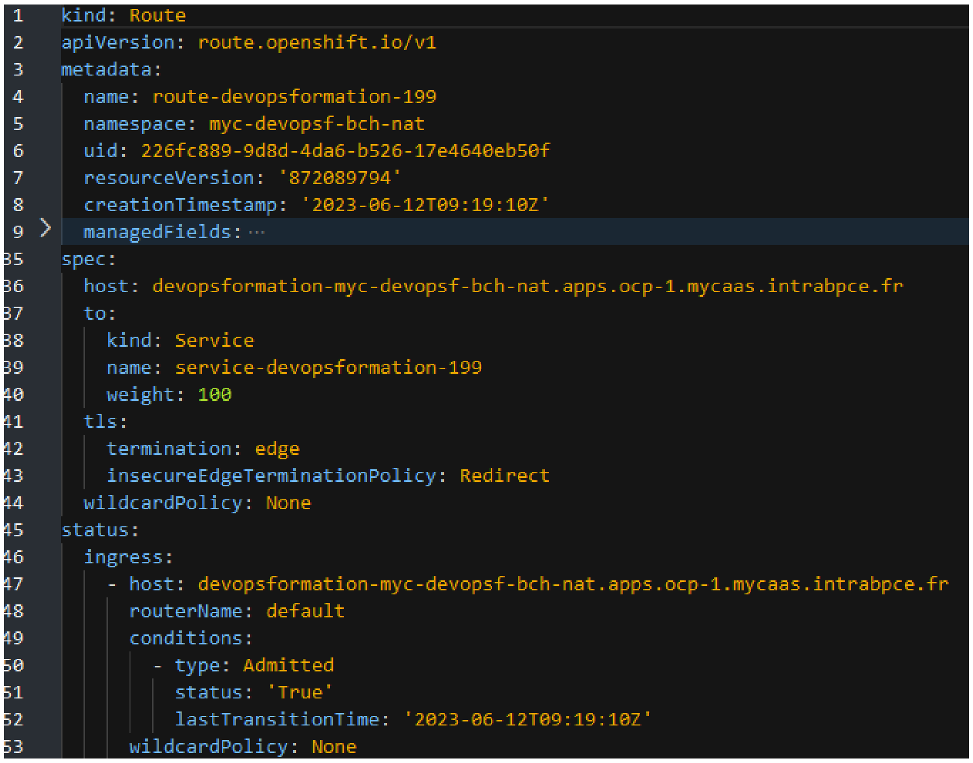


```yaml
kind: Service
apiVersion: v1
metadata:
  name: service-devopsformation-199
  namespace: myc-devopsf-bch-nat
  uid: b278f68e-7690-49f9-8f7a-9ddeaa259ea3
  resourceVersion: '872089799'
  creationTimestamp: '2023-06-12T09:19:11Z'
  managedFields:…
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 3000
  selector:
    app: devopsf
  clusterIP: 172.30.112.23
  clusterIPs:
    - 172.30.112.23
  type: ClusterIP
  sessionAffinity: None
  ipFamilies:
    - IPv4
  ipFamilyPolicy: SingleStack
status:
  loadBalancer: {}
```

Route Configuration

Routes enable external access to the application from outside the cluster. Routes provide a secure and accessible URL for accessing the application via HTTP or HTTPS protocols, specifying the service and the desired hostname to map incoming requests to the application's service.

Figure 23 OCP-Route yaml



```yaml
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  name: route-devopsformation-199
  namespace: myc-devopsf-bch-nat
  uid: 226fc889-9d8d-4da6-b526-17e4640eb50f
  resourceVersion: '872089794'
  creationTimestamp: '2023-06-12T09:19:10Z'
  managedFields:…
spec:
  host: devopsformation-myc-devopsf-bch-nat.apps.ocp-1.mycaas.intrabpce.fr
  to:
    kind: Service
    name: service-devopsformation-199
    weight: 100
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Redirect
  wildcardPolicy: None
status:
  ingress:
    - host: devopsformation-myc-devopsf-bch-nat.apps.ocp-1.mycaas.intrabpce.fr
      routerName: default
      conditions:
        - type: Admitted
          status: 'True'
          lastTransitionTime: '2023-06-12T09:19:10Z'
      wildcardPolicy: None
```

By creating the necessary OpenShift objects, it is ensured that the proper deployment, availability, and accessibility of the containerized application within the designated namespace. These objects facilitated the management of our application's lifecycle, network connectivity, and external access requirements.

Results

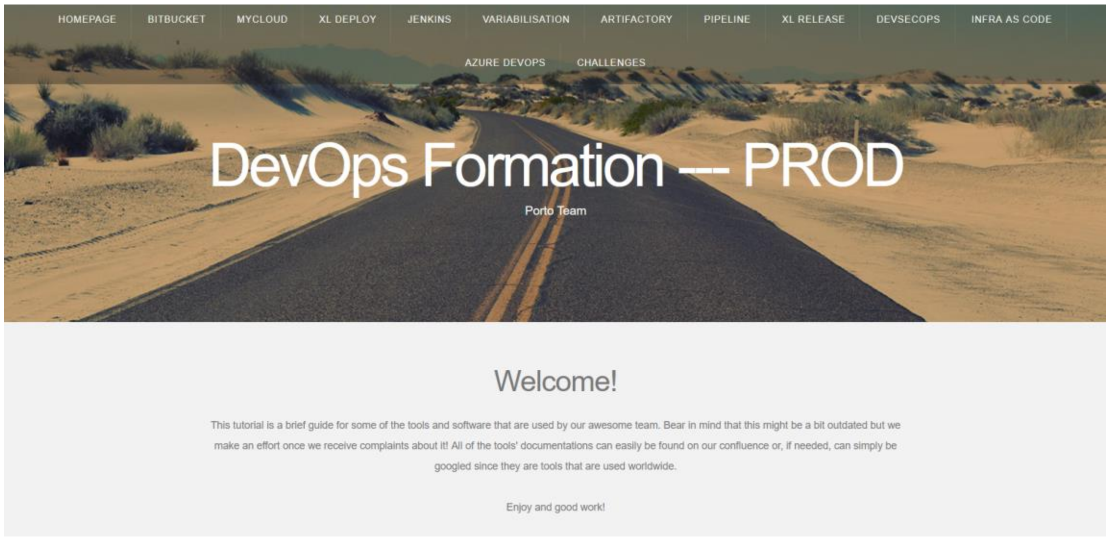
Here is presented the results and findings obtained from the migration of the DevOps formation application from a VM environment to a containerized environment using Red Hat OpenShift, the successful completion of the migration process yielded valuable insights and outcomes that contribute to the understanding and evaluation of the benefits and challenges associated with such a migration.

Migration Process Results

The migration process involved several milestones, including code refactoring, learning Red Hat OpenShift, and configuring the necessary components. The results obtained from each milestone are as follows:

The code refactoring process was successfully carried out, allowing for the adaptation of the application to run on the Red Hat OpenShift platform, the transition from the previous implementation, which relied on Apache Web Server, to using Node.js for starting the application in the Dockerfile was accomplished seamlessly. Additionally, the creation of routes for different pages within the web app was successfully implemented, enabling proper navigation and access to various sections of the application.

Figure 24 DevOps Formation landing page



DevOps Formation — PROD

Porto Team

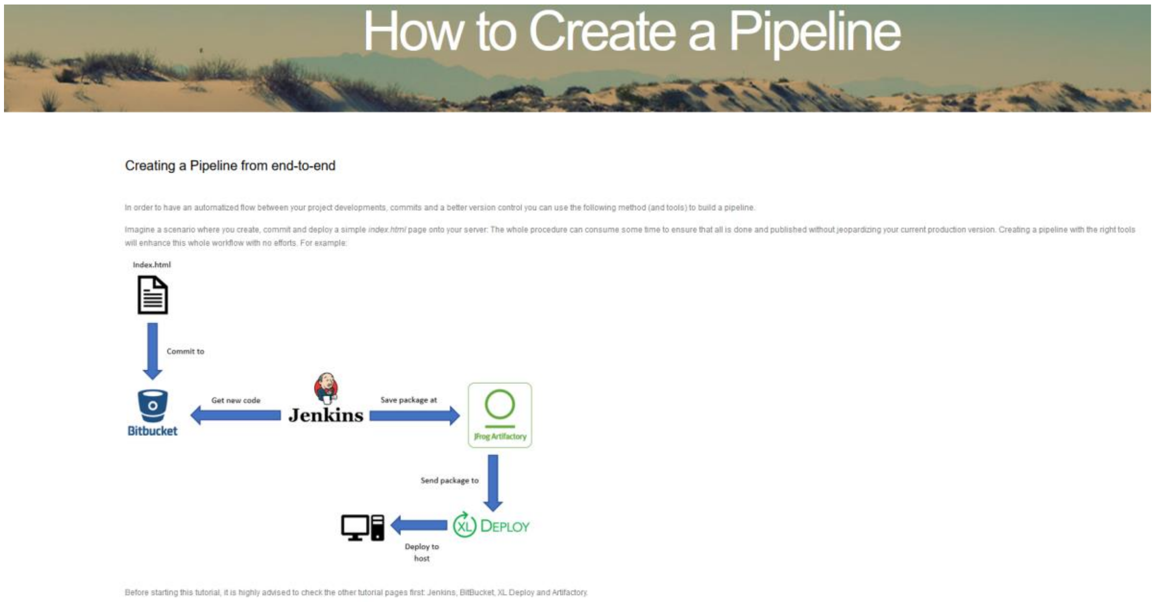
Welcome!

This tutorial is a brief guide for some of the tools and software that are used by our awesome team. Bear in mind that this might be a bit outdated but we make an effort once we receive complaints about it! All of the tools' documentations can easily be found on our confluence or, if needed, can simply be googled since they are tools that are used worldwide.

Enjoy and good work!

Despite being a newcomer to Red Hat OpenShift, a comprehensive study and exploration of the platform were conducted. This allowed for gaining a solid understanding of its architecture, features, and functionalities. The study involved learning about OpenShift's container-based paradigm, the creation and configuration of namespaces, and the creation of essential OpenShift objects such as deployments, services, and routes. The knowledge gained during this process proved instrumental in successfully deploying and managing the application within the OCP environment. The configuration and integration of various tools and technologies played a significant role in the migration process. Creation and configuration of Jenkins jobs enabled automated builds triggered by git pushes to the Bitbucket repository. The setup of webhooks on Bitbucket facilitated seamless integration between the source code repository and the Jenkins CI/CD system. Furthermore, the creation and configuration of the necessary OpenShift objects within the designated namespace ensured the proper deployment, accessibility, and management of the containerized application.

Figure 25 . Example of a page in DevOps Formation



How to Create a Pipeline

Creating a Pipeline from end-to-end

In order to have an automatized flow between your project developments, commits and a better version control you can use the following method (and tools) to build a pipeline.

Imagine a scenario where you create, commit and deploy a simple *index.html* page onto your server: The whole procedure can consume some time to ensure that all is done and published without jeopardizing your current production version. Creating a pipeline with the right tools will enhance this whole workflow with no efforts. For example:

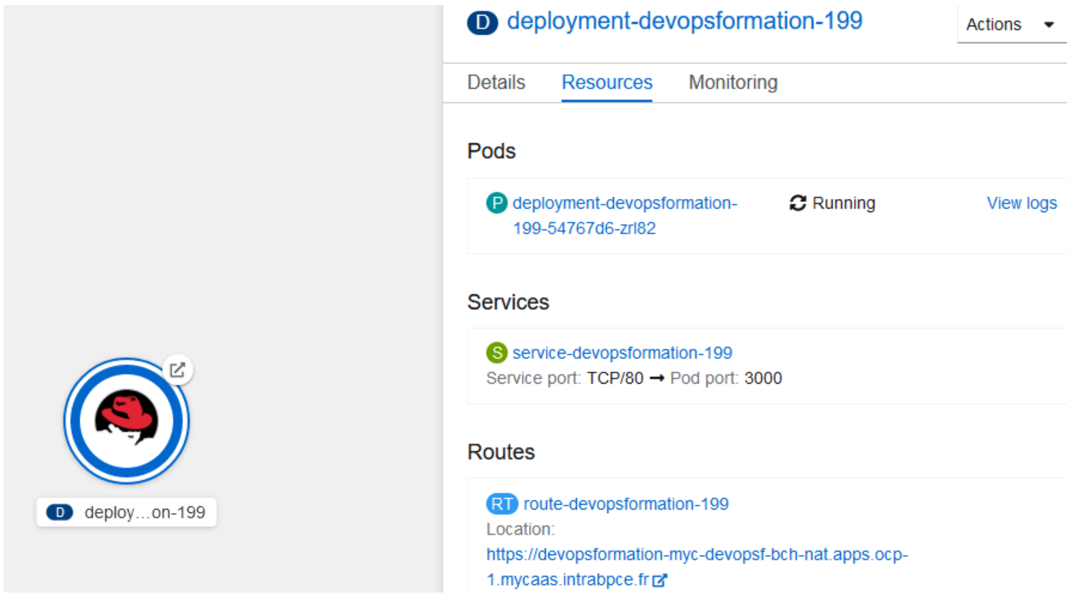
Before starting this tutorial, it is highly advised to check the other tutorial pages first: Jenkins, BitBucket, XL Deploy and Artifactory.

Evaluation and Findings

The migration of the DevOps formation application to a containerized environment using Red Hat OpenShift yielded several key findings and evaluations.

Improved Scalability and Resource Efficiency, the transition to a containerized environment provided notable benefits in terms of scalability and resource utilization. By leveraging containerization, the application could be easily scaled up or down based on demand, allowing for efficient allocation of resources and improved performance.

Figure 26 OCP Pod configuration overview



Pods

P deployment-devopsformation-199-54767d6-zrl82 Running View logs

Services

S service-devopsformation-199
Service port: TCP/80 → Pod port: 3000

Routes

RT route-devopsformation-199
Location:
https://devopsformation-myc-devopsf-bch-nat.apps.ocp-1.mycaas.intrabpce.fr

Enhanced Deployment and Continuous Integration, the use of OCP and associated tools such as Jenkins enabled streamlined deployment processes and facilitated continuous integration and delivery practices, automated builds triggered by repository events and the seamless integration between different tools improved the efficiency and reliability of the application deployment pipeline.

Remarks and Reflections

The successful migration of the DevOps formation application to a containerized environment using Red Hat OpenShift highlights the benefits and possibilities of adopting containerization technologies. The process revealed that careful planning, code adaptation, and configuration were essential to achieve a seamless transition.

Conclusions

The goal of migrating the DevOps formation application from a VM environment to a containerized environment was achieved successfully. This migration was designed to accommodate future changes to the application's CI/CD pipeline as well as to the OCP objects themselves, granting the ability to future proof this work, leveraging the capabilities of Red Hat OpenShift. The migration process involved code refactoring, creation of routes, and adapting the application to run on the containerized platform. The application now benefits from enhanced performance, scalability, and resource utilization compared to its previous VM-based deployment. Containerization has provided the DevOps formation application with enhanced scalability and resource management capabilities, dynamic scaling and workload management features, the application can easily handle varying workloads and adjust resource allocation based on demand.

In conclusion, the successful migration of the DevOps formation application from a VM environment to a containerized one using Red Hat OpenShift Container Platform has yielded significant benefits in terms of performance, scalability, resource management, and security.

References

Arango, C., Dernat, R., & Sanabria, J. (2017). *Performance Evaluation of Container-based Virtualization for High Performance Computing Environments* (arXiv:1709.10140). arXiv. http://arxiv.org/abs/1709.10140

Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, *1*(3), 81–84. https://doi.org/10.1109/MCC.2014.51

Buchanan, I. (2023). *Containers vs. Virtual machines*. https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms

Burns, B., Beda, J., & Hightower, K. (2022). *Kubernetes: Up and Running*.

Chen, G. (2019). *Modernizing Applications with Containers in the Public Cloud*.

Chowdhury, F. (2020). *Kubernetes Handbook*. https://www.freecodecamp.org/news/the-kubernetes-handbook/

Docker. (2023). *Create your own base image*. https://docs.docker.com/build/building/base-images/

Erich, F., Amrit, C., & Daneva, M. (2014). *Report: DevOps Literature Review*.

Hamzayev, S. (2023). *Monolith vs Microservices architecture*. https://medium.com/spring-boot/microservices-vs-monolith-architecture-af89b06c2c02

Harris, C. (2022). *Microservices vs. Monolithic architecture*. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith

IBM, C. E. (2021). *OpenShift vs. Kubernetes: What's the Difference?* https://www.ibm.com/cloud/blog/openshift-vs-kubernetes

Jafarnejad Ghomi, E., Masoud Rahmani, A., & Nasih Qader, N. (2017). Load-balancing algorithms in cloud computing: A survey. *Journal of Network and Computer Applications*, *88*, 50–71. https://doi.org/10.1016/j.jnca.2017.04.007

Jawarneh, I. M. A., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., & Palopoli,

A. (2019). Container Orchestration Engines: A Thorough Functional and Performance Comparison. *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 1–6. https://doi.org/10.1109/ICC.2019.8762053

Moravcik, M., & Kontsek, M. (2020). Overview of Docker container orchestration tools. *2020 18th International Conference on Emerging ELearning Technologies and Applications (ICETA)*, 475–480. https://doi.org/10.1109/ICETA51985.2020.9379236

Nickoloff, J., & Kuenzli, S. (2016). *Docker in Action, Second Edition*.

Rad, B. B., Bhatti, H. J., & Ahmadi, M. (2017). *An Introduction to Docker and Analysis of its Performance*.

Silberschatz, A., Galvin, P. B., & Gagne, G. (2010). *"Process Scheduling". Operating System Concepts (8th ed.)*.

Stallings, W. (2015). *Operating Systems: Internals and Design Principles*.

VMware Infrastructure: Resource Management with VMware DRS. (2019).

Wang, W. (2018). *Demystifying Containers 101: A Deep Dive Into Container Technology*. https://www.freecodecamp.org/news/demystifying-containers-101-a-deep-dive-into-container-technology-for-beginners-d7b60d8511c1/

Zhang, Y., Yin, G., Wang, T., Yu, Y., & Wang, H. (2018). An Insight Into the Impact of Dockerfile Evolutionary Trajectories on Quality and Latency. *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, 138–143. https://doi.org/10.1109/COMPSAC.2018.00026