POLITECNICO DI TORINO

Master Degree Course in Computer Engineering

Master Thesis

Definition of a Microservices-based Management and Monitoring System for Oracle Cloud

Supervisor
Prof. Fulvio Risso

Candidate
Marco Montalbano

Company Tutor
Technology Reply
Eng. Alessandro Dugo

Academic Year 2020-2021

Dedicated to
my family

Contents

List of Figures

Abstract

Oracle Cloud Infrastructure (OCI) is a highly performing IaaS (*Infrastructure as a service*) solution, however unlike other cloud operators (e.g., Aws, Azure, Google), the usage of its portal with its graphical interface is not so immediate and intuitive for an end user. Indeed, features such as workload life-cycle monitoring, as well as cost monitoring, are dispersive and are mostly designed for an expert user.

The cloud itself is also partitioned internally; in fact, each company that signs in is provided with an isolated partition called "tenancy" where it is possible to create, organize and manage cloud resources securely. Furthermore, within each tenancy it is also possible to divide the resources featured into logical groups defined as "compartments", each one protected by different policies defined by the user.

Starting from the root compartment, it is therefore easy to recognize a tree structure which rests on several levels. Whenever a user intends to carry out any search on a compartment to check the status of the resources allocated on it, the same will be always limited to returning elements belonging to the first level of the structure, without searching in depth in the various sub-levels, hence it will not even be possible have a complete view of all the existing resources in the tenancy or in a portion of it at a given time.

The thesis aims to create an application, leveraging a microservice-based architecture, to monitor the entire cloud and oriented to the end user. The idea is to immediately and promptly provide information on the life cycle, as well as on the costs, of individual resources and the possibility of managing their status in real time through a convenient graphical interface.

The choice of this type of architecture is designed in the spirit of favouring the scalability of the system because in the context of a high demand for a certain functionality (to give an example: the research for all the Autonomous Databases available in the cloud), the specific microservice affected can be individually scaled in an elastic and independent way from the application and other services used, rather than having to replicate each time the entire codebase with a consequent and useless waste of resources.

The main goals of this work can be summarized as follows:

- Design and development of a simpler and more intuitive GUI (Graphical User Interface) capable of requesting, ordering and presenting the information that the end user needs. The interface will, other than providing a preliminary authentication stage, have to recall a list of REST API developed at back-end to access the differentiated services provided by the architecture and aggregate data that currently are unrelated, such as cost and resource utilization level.
- Maintaining the system independently scalable in the face of an unpredictable number of requests from end users, by deploying it on Oracle Cloud Infrastructure Container Engine for Kubernetes (OKE). An application which disposes a number of microservices needs to be highly scalable, having to create many instances for each service and to balance these services on many hosts. All of this involves a huge level of deployment complexity for IT operations and management, as a consequence it's natural to think about using a microservice-oriented infrastructure as well as an orchestrator like Kubernetes.
- Searching for the right trade-off between the amount of resources used for system deploying (e.g., CPU, RAM, etc.) and quality of service (QoS) experienced by the end user. Parameters such as latency and number of errors must be kept to a minimum; in this perspective, where it will be possible and convenient, caching strategies will be put in place to store all the information that does not need to be shown updated in real time and which would imply a considerable decrease in response times of individual calls and consequently in total latency. For the others, such as the "Life-cycle State", which can change any minute now, it will inevitably be necessary at each research to call up the APIs provided by Oracle Cloud to request this information in real time.

Chapter 1

Oracle Cloud

Cloud is a technology solution that is growing and expanding with every passing day. The biggest advantage of cloud is the cost savings realized from its use on a "pay as you go" basis. Thus, everybody is looking for ways to store data for less money and greater facility and, consequently, under the impact of current changes the IT industry seeks to move on-premise solutions to the cloud. Cloud solutions can be applied in such sectors as health, retail, software, and among other major sectors. This basically means that cloud is a trend reaching countless people and therefore, as sectors grow, the market for cloud is growing with these sectors [1].

1.1 Introduction to Cloud computing

Oracle Cloud is a cloud computing service offered by Oracle Corporation providing servers, storage, network, applications and services through a global network of Oracle Corporation managed data centers. The company allows these services to be provisioned on demand over the Internet. When a company chooses to "move to the cloud", it means that its IT infrastructure is stored offsite, at a data center that is maintained by the cloud computing provider. An industry-leading cloud provider has the responsibility for managing the customer's IT infrastructure, integrating applications, and developing new capabilities and functionality to keep pace with market demands.

Cloud computing offers its clients more agility, scale, and flexibility. In this way, instead of spending money and resources on legacy IT systems, customers are able to focus on more strategic tasks. Without making a large upfront investment, they can quickly access the computing resources they need and pay only for what they use.

Oracle Cloud provides **Infrastructure as a Service** (*IaaS*), **Platform as a Service** (*PaaS*), **Software as a Service** (*SaaS*) and **Data as a Service** (*DaaS*). These services are used to build, deploy, integrate and extend applications in the cloud. Additionally, this platform supports open standards (e.g., SQL, HTML5, REST, etc.), open-source solutions (e.g., Kubernetes, Hadoop, Kafka, etc.), programming languages, databases, tools and frameworks including Oracle-specific, free and third-party software [2].

1.1.1 Cloud Models

Oracle offers three types of cloud services:

- `Public cloud`;
- `Private cloud`;
- `Hybrid cloud`.

Each type requires a different level of management from the customer and provides a different level of security.

In a public cloud the entire computing infrastructure is located on the premises of the cloud provider, and the provider delivers services to the customer over the internet. Customers do not have to maintain their own IT and can quickly add more users or computing power as needed. In this model, multiple tenants share the cloud provider's IT infrastructure.

The private cloud is used exclusively by one organization, which provides the highest level of security and control. It could be hosted at the organization's location or at the cloud provider's data center.

The hybrid cloud, as the name suggests, is a combination of both public and private clouds. Generally, hybrid cloud's customers host their business-critical applications on their own servers for more security and control, and store their secondary applications at the cloud provider's location.

There are essentially two kinds of public clouds, one serves individuals for personal use, and one serves businesses. Cloud Computing storage for personal use allows easy access and file sharing. Data stored on the cloud, such as photographs and music, can be shared with friends using a smart phone or a laptop, while protecting personal data from loss and damage [3].

A customer using a public cloud service can have three basic expectations. First, customers rent the services, instead of purchasing hardware and software to accomplish the same goal. Second, the vendor is responsible for all the administration, maintenance, capacity planning, backups, and troubleshooting. And finally, for many business projects, it is simply faster and easier to use the cloud. It comes with huge amounts of storage, the ability to handle multiple projects and more availability to a variety of users, simultaneously.

1.1.2 Cloud Services

The services offered by the business cloud are quite different and fall into three basic categories of service.

IaaS is the most basic service, it deals with raw computing capacity and provides a collection of servers, storage, and network infrastructure onto which the customer deploys its platform and software. This is most akin to provisioning the customer own hardware in an on-premises data center. Teams of hardware engineers, storage specialists, network specialists, system administrators, and database administrators are usually involved in installing and configuring on-premises infrastructure. IaaS customers are often tech companies that typically have a great deal of IT expertise. The goal is to have access to computing power without the responsibilities of installation or maintenance.

PaaS provides a collection of one or more preconfigured infrastructure instances, usually provided with an operating system, database, or development platform onto which the customer can deploy its software. The primary benefit of PaaS is convenience as the cloud vendor provides and supports the underlying infrastructure and platform. PaaS environments are equipped with software development technologies, such as .NET, Python, Ruby on Rails, and Java. When the code is finished, the service provider will host it, making it available to other internet users. Currently, PaaS is the smallest part of the Cloud Computing market, and has been used by businesses wanting to outsource part of their infrastructure.

SaaS provides a program, or a suite of applications, available within the Cloud. It results easier to manage rather than a computer's hard drive, and all you do is access them through a browser. This part of the Cloud is the largest and most developed and this could range from web mail to complex ERP and BI Analytic systems.

Oracle Cloud encompasses the Oracle Public Cloud, which represents a collection of infrastructure, platforms, and applications exposed as services on `http://cloud.oracle.com/`. Figure 1.1 highlights two important factors on the same spectrum when considering cloud computing models: control and convenience. The convenience offered by SaaS and PaaS comes at the cost of less control. IaaS offers the most control and complete access to the infrastructure but requires the most effort. For instance, when a compute instance is provisioned, the client has the option to choose the operating system image. With PaaS, if a database is provisioned, the client will have administrator privileges, but he has fewer options and he cannot choose the operating system on the compute instance created. SaaS offers maximum convenience, but the dependency on the cloud vendor for maintenance and support [4].

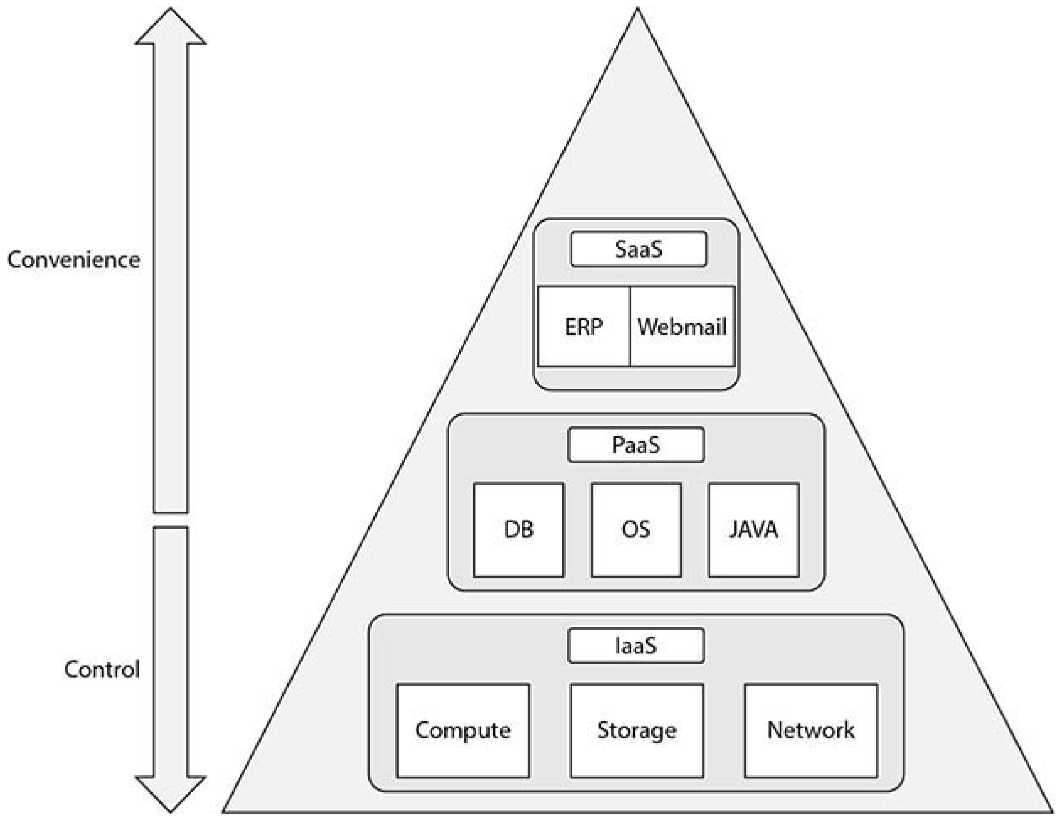


Figure 1.1: Cloud computing models (control vs convenience)

1.2 Oracle Cloud Infrastructure

Oracle Cloud Infrastructure (*OCI*) is an Iaas cloud platform that allows customers to create resources (e.g., compute instances, databases, networks, containers, functions, storage) in order to run their applications and workloads [5]. OCI offers relational, OLAP, JSON, and NoSQL databases, containers, Kubernetes, serverless functions, Spark, streaming, Jupyter notebooks, VMware and the range of cloud services necessary for nearly any workload. In 2020 alone, OCI launched nearly four hundred new services, features, and enhancements [6].

The infrastructure is designed for applications that require consistent high-performance, including stateful connections to databases, raw processing through CPUs or GPUs, millions of storage IOPS, and GB/s of throughput. Non-blocking networks guarantee that each resource gets predictable high-performance and low latency. The infrastructure leverages the latest CPUs, GPUs, networking, and storage technology like NVMe SSD drives.

1.2.1 Features and Components

OCI consists of servers, storage, and networking equipment. These reside in data centers. Data centers with resilient and redundant components, that do not have a single-point of failure, are referred to as fault-tolerant data centers. In OCI cloud-speak, a fault-tolerant data center is an **Availability Domain** (*AD*). One or more ADs located in a metropolitan area and connected with high-speed networks are grouped into a region. Figure 1.2 describes two regions, each comprising three ADs.

Availability Domains

Most OCI resources are either region-specific, such as a virtual Cloud Network (VCN), or availability domain-specific, such as a compute instance. Traffic between ADs and between regions is encrypted. ADs are isolated from each other, fault tolerant, and very unlikely to fail simultaneously. And this is because they do not share infrastructure such as power or cooling, or the internal availability domain network, so that a failure at one AD within a region is unlikely to impact the availability of the others within the same region.

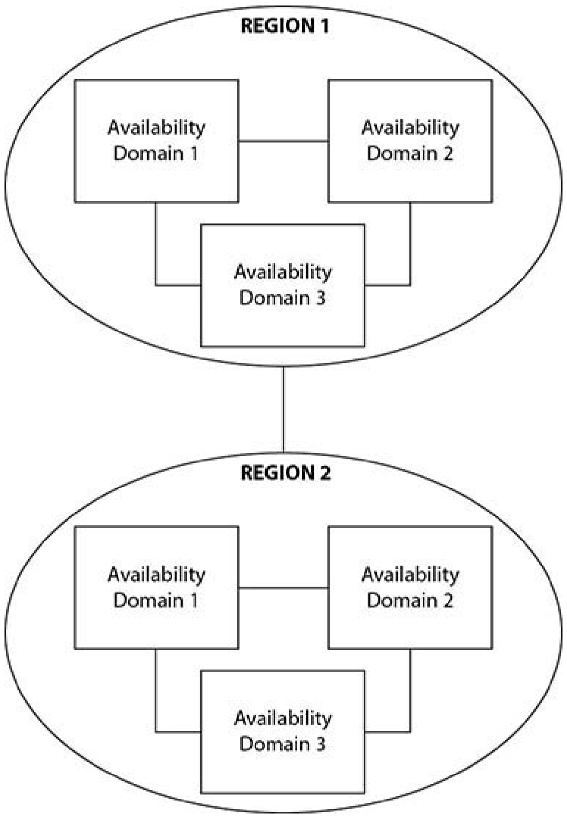


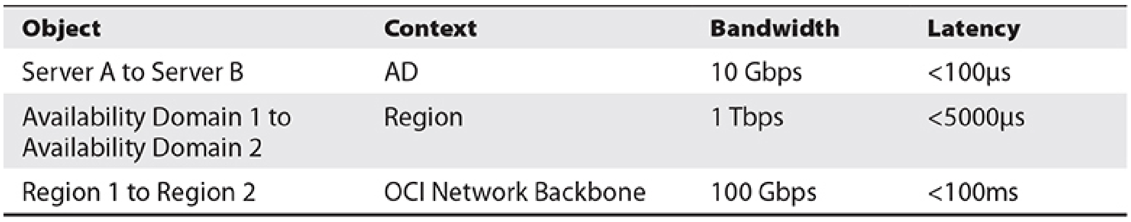
Figure 1.2: Regions and availability domains

OCI Network

The ADs within the same region are connected to each other by a low latency, high bandwidth network, which makes it possible to provide high-availability connectivity to the internet and on-premises, and to build replicated systems in multiple ADs for both high-availability and disaster recovery.

Network performance is typically measured by the metrics bandwidth and latency. Bandwidth refers to the throughput or volume of data that can be transferred over time. For example, 5 Gb/s means that five Gigabyte of data will be transferred over the network from one endpoint to another per second. Latency refers to the delay or time taken for a data packet to traverse a network and it is usually measured as the Round Trip Time (RTT) [4].

Oracle claims that the network bandwidth between servers in each AD is 10 Gbps with a latency of less than 100 microseconds. This network is a flat high-speed non-oversubscribed Clos network that provides around one million network ports per AD. Oracle promises a maximum of two network hops between Compute and Storage resources regardless of the size and scale of the estate. The bandwidth between ADs in each region is 1 Tbps with a latency of less than 5,000 microseconds. Finally, the bandwidth between regions, which are geographically vast distances apart, is 100 Gbps with a latency of less than 100 milliseconds. These network performance metrics are listed in Figure 1.3.



Object	Context	Bandwidth	Latency
Server A to Server B	AD	10 Gbps	<100µs
Availability Domain 1 to Availability Domain 2	Region	1 Tbps	<5000µs
Region 1 to Region 2	OCI Network Backbone	100 Gbps	<100ms

Figure 1.3: Network performance between servers, ADs and regions

Regions and Realms

OCI was launched in October 2016 with a single region and core services across compute, storage, and networking. Since then, Oracle has expanded to more than seventy services available in twenty-nine cloud regions worldwide with plans to reach thirty-eight total regions by the end of 2021. It is gradually adding multiple cloud regions around the world to provide local access to cloud resources for their customers. To accomplish this quickly, the company has chosen to launch regions in new geographies with one AD. For any region with only one AD, a second AD or region in the same country or geo-political area will be made available within a year, as a way of enabling further options for disaster recovery that support customer requirements for data residency, where they exist.

As shown in Figure 1.4 there are two types of regions: commercial and government.

- `Government Cloud` is dedicated to Government Organizations only (Federal compliant), therefore only government companies can access them with proper approvals and this will be assigned by Oracle;
- `Commercial Cloud` is not limited to one Organization but it is publicly available.

There is a total of twenty-three region now, every region can have up to three ADs. The first four regions (London, Ashburn, Pheonix and Frankfurt) have three ADs while all the new regions have only one AD each.

Regions are independent of other regions and can be separated by vast distances-across countries or even continents. Generally, a customer would deploy an application in the region where it is most heavily used, because using nearby resources is faster than using distant resources. However, nothing prevents customers to deploy applications in different regions for these reasons:

- To mitigate the risk of region-wide events such as large weather systems or earthquakes;
- To meet varying requirements for legal jurisdictions, tax domains, and other business or social criteria.

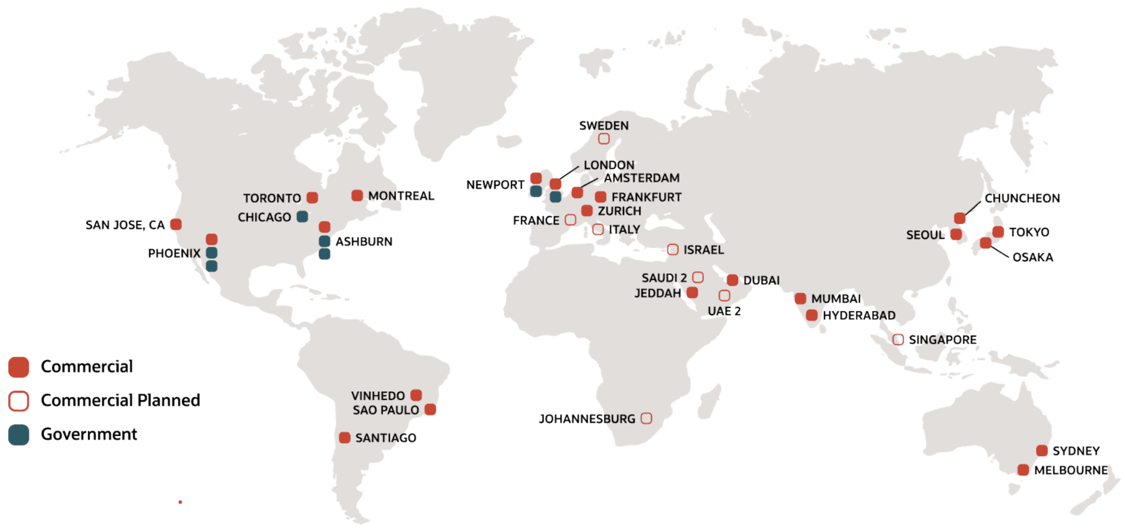


Figure 1.4: Regions map

A realm is a logical collection of regions, realms are isolated from each other and do not share any data. For a customer who subscribe to a single realm, it is only possible to access regions that belong to that realm. Currently, OCI has multiple realms, there is only one commercial realm (OC1) and three realms for Government Cloud: US Government Cloud FedRAMP authorized (OC2), IL5 authorized (OC3) and United Kingdom Government Cloud (OC4).

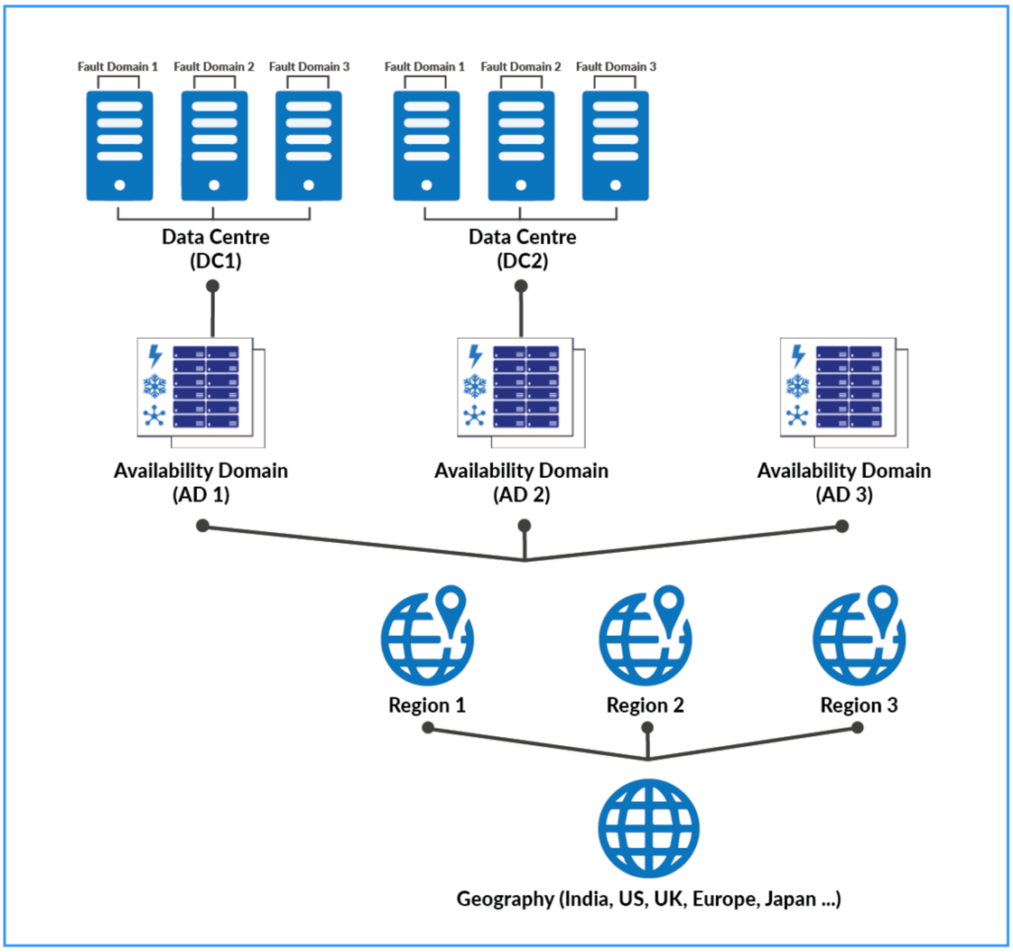


Figure 1.5: OCI structure

Fault Domains

A **Fault domain** (*FD*) is an infrastructure grouping, that allows the instances to be distributed so they do not reside on the same physical hardware within an AD. Typically each AD has three FDs, they act as a logical data center so that resources placed in different FDs will not share single points of hardware failure (e.g., same physical server, physical rack, top of rack switch or power distribution unit).

Figure 1.5 perfectly summarizes the physical and logical structure which characterizes the OCI.

In any region, resources in at most one FD are being actively changed at any point in time, this means that availability problems caused by change procedures are isolated at the fault domain level. FDs offer additional protection at a physical server level against unexpected hardware failures and improves availability during planned outages. In addition, the physical hardware in a FD has independent and redundant power supplies, which prevents a failure in the power supply hardware within one FD from affecting other FDs [7].

To control the placement of a compute instances, bare metal DB system instances, or virtual machine DB system instances, it is optionally possible to specify the FD for a new instance or instance pool at launch time. If not specified, the system will select randomly one FD from the pool provided. OCI makes a best-effort anti-affinity placement across different FDs, while optimizing for available capacity in the AD.

1.2.2 Identity and Access Management

The **Identity and Access Management** (IAM) service, enabled by default in OCI, encompasses the three A's of security (Authentication, Authorization and Access). The IAM service allows to control who has access to cloud resources and what type of access a group of users has and to which specific resources. The service enables to enforce the security principle of least privilege by default. New users are not allowed to perform any actions on any resources until they are granted with appropriate permissions. IAM in OCI revolves around several novel concepts such as tenancy and compartments, while utilizing relatively familiar constructs such as users, groups, and policies [4].

Tenancy and Compartments

When a client signs up for OCI, Oracle creates a **Tenancy**, which is a secure and isolated partition within OCI where the customer can create, organize, and administer its cloud resources. OCI resources are collectively grouped into **Compartments**. A compartment is a logical container thought to organize and control access to the OCI resources (e.g., Compute, Storage, Network, Load Balancer, etc.), when an OCI account is provisioned, several compartments are automatically created, including the root compartment of the tenancy. A tenancy is synonymous with the cloud account and comprises a hierarchy of compartments with the root compartment at the top. There can be many compartments, and they may have child compartments nested six levels deep.

An OCI resource can only belong to one compartment, moreover resources that make up or reside on the same VCN can belong to different compartments as well as resources from multiple regions can be in the same compartment. Resources can interact with other resources in different compartments and they can also be moved from one compartment to another.

Another important consideration is that compartments can reflect an organization functional structure, where each department has its own compartment with a designated administrator. Each department compartment, in turn, can have sub compartments for different environments (e.g., Dev, Test, Prod), each one with their own administrators, if necessary. Figure 1.6 depicts this particular structure and is used as the basis for the practical example described here [8]. It may be convenient to group all infrastructure resources consumed by a specific department into their own compartment. A trend in infrastructure support is to track infrastructure usage for cost management.

These features can help a company to organize and isolate its cloud resources in a way that aligns with the data management goals of enforcing the purpose limitation of any personal information to be processed. For example, an enterprise could create a compartment for their human resources department, and another for the finance department. This would effectively separate the cloud resources, which in turn would help keep separate the data, for the two departments [9].

By default, any OCI tenancy has a default root compartment, named after the tenancy itself. The tenancy administrator (default root compartment administrator) is any user who is a member of the default Administrators group. Once compartments are created, they can be assigned their own administrators, which can then create sub-compartments and assign delegated administrators to each of them. OCI supports up to a six-level deep compartment hierarchy and allows the administrator of a parent compartment to have full power over its children compartments.

Compartments also allow resources to be secured and managed as a single entity. Once a compartment is created, it is typical to create a policy to allow appropriate access to the resources in the compartment.

Groups

OCI users are organized into groups, a user may belong to many groups at the same time. The Administrators group, created along with the OCI account, initially has a single member, that is the user created when the tenancy was provisioned. The administrator may also create additional administrator users and add them to this group or create other groups for duty separation. The administrator users have complete control over all resources in the tenancy so access to this group should be tightly regulated. The current segregation of the infrastructure and, more importantly, the current partitioning of human resources into technical teams are often good models on which to base OCI groups. These teams may support specific applications or technologies or different infrastructure layers such as OS, storage,

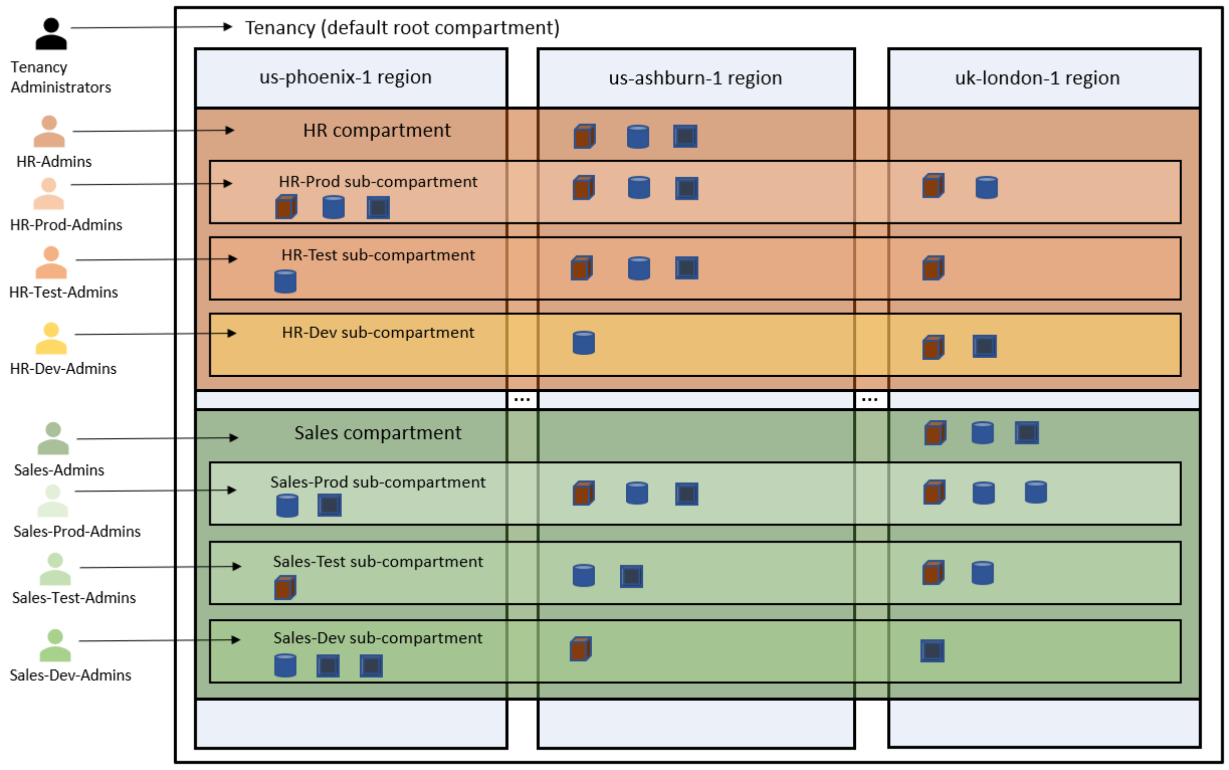


Figure 1.6: Organization multi-compartment structure

network, databases, etc. They may also support specific departments or business units. Aligning OCI group design with existing human team divisions, often simplifies the IAM nomenclature and group management strategy. As the volume of users and infrastructure grows, management of OCI resources inevitably grows in complexity. Groups are that piece of the IAM solution essential for practical, as well as auditable, user and infrastructure governance [4].

Policies

Policies are the glue that determines how groups of users interact with OCI resources that are grouped into compartments. A policy allows a group to work in certain ways with specific types of resources in a particular compartment or tenancy. Policies only allow access to groups of users, not to individual users, and they cannot explicitly deny it. It may be also possible to remove a user from a particular group of interest or delete the user entirely from the IAM service to restrict a certain access. Access is granted at the group level and compartment level, which means that exists a policy that gives a group a type of access within a specific compartment, or to the tenancy itself. If the access to the tenancy is given to a group, the group automatically gets the same type of access to all the compartments inside the tenancy [10]. Policies are inherited by their child compartments, so that if a policy is created in the root compartment it applies to all compartments, while a policy created in a child compartment with no sub-compartments applies only to the relevant resources within that child compartment.

Each policy consists of one or more policy statements that follow this basic syntax:

```
        Allow group <group_name> to <verb> <resource-type>
               in compartment <compartment_name>
```

The <verb> denotes the type of access: `inspect`, `read`, `use`, or `manage`. For example, "inspect" gives users in the group the ability to list resources without access to confidential information or user-specified metadata in the resource. The "read" verb instead, includes the permissions of the "inspect" verb and additionally provides access to user-specified metadata about the resource and access to the actual resource. This level of authorization is typically reserved for internal oversight and monitoring. The "manage" verb includes all permissions for the resource and it effectively combines the read permissions with the abilities to create and destroy the resource-type. It is the highest level of permission that can be granted on a resource-type and is generally reserved for administrator groups. The <resource-type> can be an aggregate (family) resource or an individual resource. For example, database-family is an aggregate resource-type, while db-systems and db-nodes are individual resource-types in that family.

Instance Principals and Dynamic Groups

Another important feature of IAM is the **Instance Principals**, which enables instances to be authorized actors (or principals) to perform actions on service resources. In this way, users are allowed to call IAM-protected APIs from an OCI Compute instance (virtual machine or bare metal) without the need to create IAM users or manage credentials for each instance. Each compute instance has its own identity and it authenticates using the certificates that are added to it. These certificates are automatically created, assigned to instances, and rotated, preventing the need to distribute credentials to the hosts and rotate them [11].

Oracle uses **Dynamic Groups** to implement instance principals and control them by using a policy definition. Every bare metal or virtual machine instance is deployed with an instance certificate which contains metadata about the instance. This includes the instance and the compartment identification number, along with a few other optional properties. When an API call is made, using this instance certificate as the authenticator, the certificate can be matched to one or multiple dynamic groups. The instance can then get access to the API based on the permissions granted in policies and written for the dynamic groups [12].

A logical diagram which summarizes how the OCI IAM works, it is depicted in Figure 1.7.

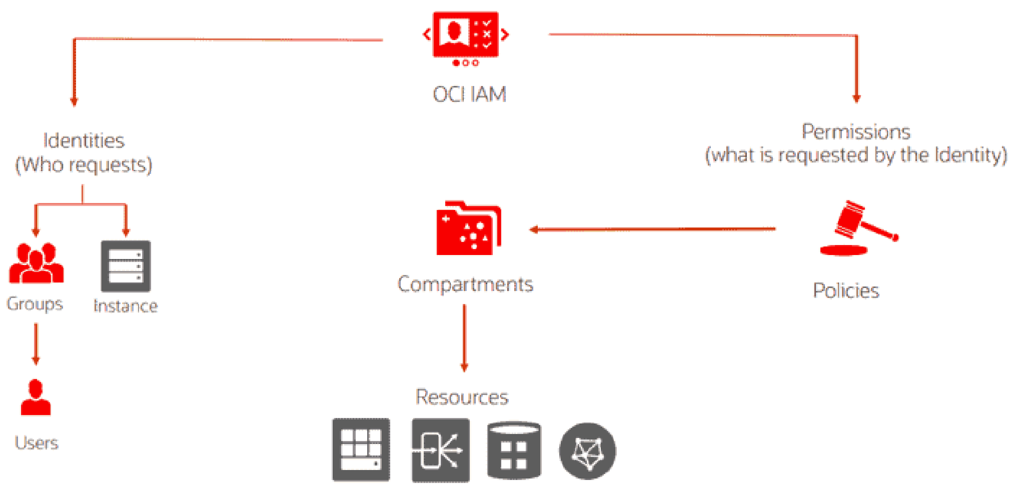


Figure 1.7: IAM logical diagram

1.2.3 Resources

OCI resources have a parallel definition and refer to artifacts, including compute instances, block storage volumes, object storage buckets, databases, file system storage, VCNs, load balancers, and dynamic routing gateways.

Global Resources

IAM resources such as users, groups, compartments, and policies are considered global resources. These resources exist in all regions and availability domains, however, they are initially created in the home region of the tenancy and the master copy of their definition resides there. When changes are made to IAM resources, they must be made in the home region and then these changes are automatically replicated to other regions. Changes to IAM resources in the home region typically take a few seconds, while it may take a few minutes for these changes to propagate to all regions.

OCID

Every OCI resource is assigned a unique identifier known as an Oracle Cloud Identifier or **OCID** (sometimes pronounced "o-sid"). All resources in OCI have been exposed through **REST** (Representational State Transfer) **APIs** and OCIDs are required to access the OCI APIs through the CLI (Command Line Interface) or the SDKs (Software Development Kits).

Developers can also use several SDKs (e.g., Java, Ruby, Python, Go) which include documentation, online sample code, and many useful tools for interfacing with OCI resources.

The OCID is based on this format:

```
ocid1.<RESOURCE TYPE>.<REALM>.[REGION][.FUTURE USE].<UNIQUE ID>
```

The following are some examples:

```
ocid1.tenancy.oc1..aaaaaaaaddqda
ocid1.availabilitydomain.oc1..aaaaaaaawyohta
ocid1.compartment.oc1..aaaaaaaazlh3iq
ocid1.vcn.oc1.iad.aaaaaaaahr6y4a
ocid1.routetable.oc1.iad.aaaaaaaapuwaa
ocid1.subnet.oc1.iad.aaaaaaaaqnyy2q
```

1.2.4 Compute Service Components

The compute service fundamentally provides compute instances as a service, which means interacting with OCI through APIs or the console to provision a computing host or instance. Instances can be either **Virtual Machines** (*VMs*) or **Bare Metal machines** (*BMs*) and reside on physical equipment localized in a data center or AD.

VM is defined as an independent computing environment, executing on physical hardware. VM compute instance runs on the same hardware as a bare metal instance, leveraging the same cloud-optimized hardware, firmware, software stack, and networking infrastructure. Multiple VMs may share the same physical hardware, as opposed to BM instances which execute on dedicated hardware, providing strong isolation and highest performance. VMs are ideal for running applications that do not require the performance and resources (e.g., CPU, memory, network bandwidth, storage) of an entire physical machine.

OCI provides a hypervisor layer that accepts API calls from the console, CLI, and other SDKs. A hypervisor is the lowest-level operating system software that is installed on bare metal servers. A compute instance runs on top of the hypervisor layer and, through it, interacts with the physical hardware. In other words, the hypervisor on a particular x86 server may host multiple VMs, while BM compute instances run directly on bare metal servers without a hypervisor. Both machine-types are available in many shapes and are based on x86 hardware, and so are capable of running a variety of Linux and Windows operating systems [4].

When a new compute instance is created, many options may be specified, including a name, the AD it resides in, the boot volume and the shape.

Shape

A compute **Shape** is a predefined bundle of computing resources, primarily differentiated by: Oracle Compute Units (OCPUs), memory, network interfaces, network bandwidth, and support for block and NVMe local storage. An OCPU provides CPU capacity equivalent to one physical core of an Intel Xeon processor with hyper-threading enabled and corresponds exactly to two hardware execution threads, known as vCPUs.

The most widely used solution is the flexible shape, which lets the client the freedom to customize the number of OCPUs and the amount of memory when launching or resizing its VM, based on the workloads that will run on the instance. Other options (e.g., the amount of memory, the network bandwidth and the number of VNICs) scale proportionately with the number of OCPUs. This flexibility lets the customer build VMs that match the workload, enabling its to optimize performance and minimize cost [13].

Another key decision, when creating compute instances, is to determine the operating system image. There is the option to choose from the following:

- A template of a virtual hard drive that determines the operating system and other software for an instance;
- Trusted third-party images published by Oracle partners from the Partner Image catalog;
- Pre-built Oracle enterprise images and solutions enabled for OCI;
- Custom images, including bring own image scenarios;
- Boot Volumes.

OCI offers several pre-built Linux and Windows images in various shape-related editions, complete with the appropriate drivers to rapidly provision the instance. It also provides a cloud marketplace where third-party vendors proffer their application software to OCI users.

Partner images are pre-built images that include an operating system and application deployment from a third-party provider. These images have been vetted by Oracle and are considered trusted images.

OCI also offers an interface for creating own images from existing compute instances and saving these as custom images, to be further used as the basis for future compute instance deployments. Custom images may be based on OCI platform or Oracle partner images, that have been customized in some way or imported from an external images, meeting several requirements.

Custom images must be launched in one of three modes:

- `Native mode`: Drivers in the image communicate directly with underlying hypervisor;
- `Paravirtualized mode`: The guest image is modified to hook directly to the underlying hypervisor for certain tasks;
- `Emulated mode`: The guest image is fully virtualized and runs without modification on the OCI hypervisor.

The launch mode is determined by the compatibility of the underlying image with the hardware hosting the virtual machines. Custom images, imported from OCI format exports, may be launched in native mode because these images already have system drivers for the underlying hardware. Images created outside of OCI may be launched in either emulated mode or paravirtualized (PV) mode, depending on whether the operating systems in these images have support for the underlying hardware. Older operating systems typically do not have drivers for modern hardware and are likely to launch in emulated mode only. The performance of virtual machines is related to its launch mode. If the compute shapes are the same, a VM launched in native mode performs better than one launched in PV mode, whereas an instance launched in PV mode will perform better than one in emulated mode. It is therefore preferable to migrate older systems to newer naively supported images.

Boot Volume

Boot Volume instead, is a special block volume that stores the operating system and boot loader required to launch the compute instance. The default boot volume size depends on the image chosen and it may be increased to provide headroom for future growth of the volume itself. Linux images usually require a significantly smaller boot volume than Windows images. Access to boot volumes is provided to compute instances through the OCI Block Volume service.

When a new VM or BM instance based on a platform image or custom image, is launched, a new boot volume for the instance is created in the same compartment. That boot volume is associated with that instance until its termination but the volume and its data can be preserved. Apart from imaging compute instances, a boot volume clone may be taken when troubleshooting a problematic instance that cannot boot up or to recover data. A useful technique is to clone the boot volume of the problematic instance and attach the clone to another instance, as another block volume. By mounting the cloned volume as a secondary volume, the file systems can be exposed on the cloned volume for further investigation. After resolving the issue, all that remains is to then reattach it to the original instance or use it to launch a new instance [14].

1.2.5 Storage

Storage is an essential ingredient in the cloud computing puzzle and unsurprisingly a comprehensive array of storage options is available on OCI. The fastest and most expensive storage options available in OCI are NVMe (Non-Volatile Memory Express) and SSD storage drives, attached locally to a compute instance. This storage is typically used in high performance computing where high IO speeds are required (e.g., important transactional database) and provides Terabyte scale capacity. This is not a durable storage at all, because it is not possible to replicate it to other ADs, unlike **Block Volumes** that are durable, fast and provide petabyte scale storage. Instead, the slowest and cheapest storage option is OCI **Object Storage**. For instance, the object storage buckets are appropriate for long-term storage when some data must be kept safely and IO speeds are not important, such as keeping several years of financial record backups for audit or compliance purposes [5].

Block Volumes

Block volumes can be used either to dynamically expand the storage capacity of a compute instance , as well as a database, or to provide durable and persistent data storage that can be used with different machines and even across multiple machines (multi-attach). Block volumes may be created, attached, connected, and detached, as needed. In fact to meet the storage performance and application requirements, a block volume may be detached from one compute instance and attached to another instance in the same AD, thereby moving the volume. After attaching and connecting a volume to an instance, it can be used like a regular hard drive, whose size go from 50 GB to 32 TB in 1 GB increments. They may be grouped with other block volumes to form a logical entity, known as volume group. Many volume groups, in turn, may be backed up together to form a consistent point-in-time, crash-consistent backup that is also useful for cloning [15].

Once a block volume has been provisioned and is in the AVAILABLE life-cycle state, it may be used by attaching it to a compute instance. There are two types of volume attachments:

- `iSCSI`: It connects the block volume to the instance using an TCP/IP network connection;

- `Paravirtualized`: This attachment type is available only on VMs and adds an extra IO virtualization layer.

Buckets and Objects Storage

Object Storage is a relatively new resilient storage-type that has become a standard for general purpose file storage in the cloud. It is not definitely suitable for high-speed computing storage requirements (such as those required to run databases), but provides flexible and scalable options for unstructured data storage and sharing, as well as being great for big data and content repositories (e.g., backups, archives, log data and large datasets) [4]. Besides, it can store an unlimited amount of unstructured data of any content type, including analytic data and rich content, like images and videos. Object storage is also not bound to an instance or an AD but is a region-level construct that resides in a compartment and, as consequence, is not tied to any specific compute instance. Data stored can be accessed from anywhere inside or outside the context of the OCI, as long an internet connectivity is provided and can access one of the Object Storage endpoints. Instances inside the tenancy may read and write to object storage through a service gateway, connecting either through the VCN or through the Internet to object storage, if sufficient permissions have been granted [16].

Bucket is a logical container for objects that reside in a compartment. As the name suggests, it is suitable for storing objects of any data type. It is possible to create up to thousand buckets per compartment per region and store an unlimited number of objects. Differently from traditional file systems, buckets may not be nested and may not contain other buckets. A single uneditable namespace is provided to a tenancy, that serves as the top-level root container for all buckets and objects. A bucket may exist at one of two tiers: `Standard tier` and `Archive tier`.

Objects stored in a standard tier bucket may be accessed frequently, and data is immediately available. This tier of storage has good performance but is more expensive than archive tier storage. Instead, archive tier is mainly used for objects that are infrequently accessed but that must be retained and preserved for a long time. Consequently, there is a longer lead time to access objects in archive tier buckets than in standard tier buckets.

1.2.6 Database

OCI provides their customers with the ability to deploy Oracle DB in the Cloud, with Oracle providing the physical storage, computing power, and tooling (e.g., backup, recovery, patching, upgrade operations) for routine database maintenance. Customers using Classic Cloud Service have full administrative privileges for the created Oracle DB [17].

The Database service offers these two Database cloud solutions:

- `Autonomous Databases` are preconfigured, fully-managed environments that are suitable for either transaction processing or for data warehouse workloads;

- `Co-managed solutions` are bare metal, virtual machines and Exadata DB systems which can be customized with the resources and settings needed.

Figure 1.8 contextualizes the OCI database services presented.

The customer has a full access to the features and operations available with the DB, but Oracle owns and manages the infrastructure.

Database Cloud Services (*DBCS*) is a PaaS which offers the possibility to choose: the compute shape, the storage, the GI and the DB version; it also lets OCI's cloud automation to complete the tedious heavy lifting behind the scenes. DBCS finally provides the customer with a fully functional and deployed Oracle DB platform on a VM, BM, or Exadata server. Exadata is Oracle's flagship engineered systems platform explicitly designed for hosting clustered highly available and high-performance Oracle DBs. DBCS significantly simplifies DB instance management, including taking backups, performing restores, and applying patches.

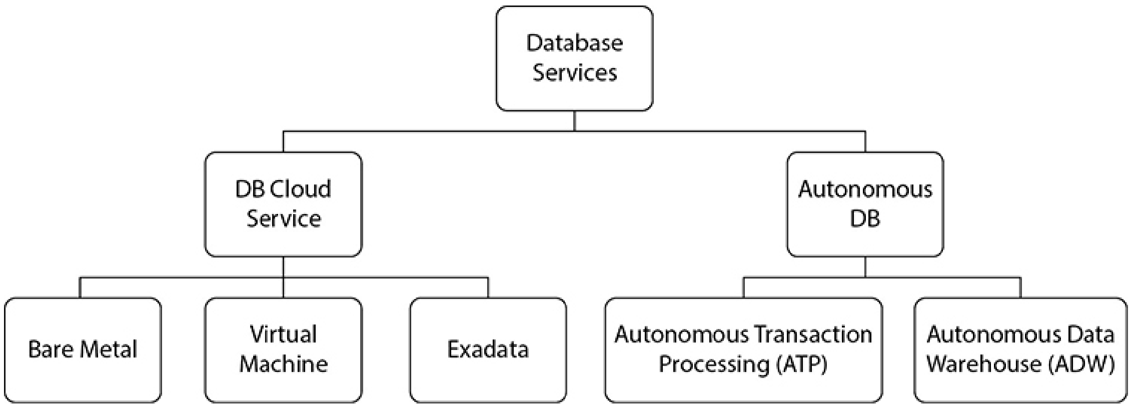


Figure 1.8: OCI Database Services

Autonomous Database

Oracle Autonomous Database (*ADB*) offers fully automated databases optimized for:

- Transaction Processing;
- Data Warehouse;
- Document-oriented Workloads (JSON).

.

Oracle ADB is built on top of Oracle Exadata and offers shared or dedicated deployment options. The dedicated option isolates the underlying infrastructure resources to a single tenant. ADB systems offer a hosted and managed option with an underlying Exadata service and the ability to dynamically scale up and scale down both the CPUs and storage allocated to the VM. This single feature unlocks a great number of possibilities, chief among them the game-changing idea of sizing the environment for average workload, scaling up during peak periods, and scaling back down once the workload normalizes.

Relying on decades of internal automation, ADB uses advanced machine learning algorithms to balance performance and availability with cost, automating many tasks including indexing, tuning, upgrading, patching, and backing up the DB. High availability is achieved through the use of a RAC database (when scaling to more than 16 OCPUs), triple-mirrored disk groups, redundant compute and network infrastructure, and nightly backups [4].

Autonomous Transaction Processing (*ATP*) workload-type targets OLTP (On-Line Transaction Processsing) databases, and configuration parameters are biased toward high-volume random data access typical of OLTP systems. ATP databases are also suitable for mixed workloads, including some batch processing reporting, IoT, and machine learning, as well as transaction processing.

Autonomous Data Warehouse (*ADW*) workload-type targets analytic systems including data warehouses, data marts, data lakes, and large machine learning data with configuration parameters biased toward high-volume ordered data scanning operations.

ATP and ADW each support a different workload-type but they share the underlying infrastructure and tooling. Essentially, they differ in database initialization parameters and automation options. Furthermore, ADW stores data in a columnar format while ADB uses a traditional row store.

In the following chapter the problems related to the use of the OCI console, which is necessary in order to interact with OCI resources presented in this chapter, will be debated and investigated and possible solutions will be highlighted and discussed.

Chapter 2

OCI Console

The OCI console, based on a browser interface, is the main way for a company or client (who has subscribed to OCI and deployed a certain number of resources) to access their basic information on the state, cost and related metadata.

2.1 Usability issues

Many features such as life-cycle monitoring, as well as cost monitoring, are dispersive and mostly designed for an expert user, which has already experienced the portal and exploited its either basic or complex functionalities.

"Technology Reply" is a consulting firm which is part of the Reply group and which is characterized by the partnership with Oracle. It is based on Cross-Industry technology and with vertical skills in the Financial Services field, for the implementation of System Integration, Business Applications, Data Warehouse, Big Data and Machine Learning projects. During the internship, which i have carried out in this company, i have collected from many colleagues (especially the new hires) and even managers the need for having a faster and simpler way to have a complete and full monitoring over a compartment and the status of all the resources allocated on it.

2.1.1 Resources Monitoring

Entering into detail, a fairly encountered problem is the monitoring of all the resources belonging to a specific compartment within a tenancy. As explained in section 1.2.2, each company that signs in is provided with an isolated partition called tenancy, this partition is in its turn partitioned into logical container called compartments. This division is carried out, mainly based on current projects the company is working on, or on the organization functional structure. The basic scenario presents a logical structure where each department has a compartment with a designated administrator, each compartment is furthermore divided in sub-compartments, each one deployed for different environment or project. This disposition can help a company to organize and isolate its cloud resources in a privacy-oriented way, that aligns with the data management goals of enforcing the purpose limitation of any personal information to be processed.

The problem arises from the logical structure briefly described before, which can be seen as a tree structure resting on several levels (at most six), each one representing a sub-compartment. At the top of the hierarchy can be found the tenancy or root compartment, which is provided to the company after the subscription. Normally, it is considered the stage on which to develop all the structure but it can be used to store resources, as well as the other compartments. Whenever a user intends to carry out any search on a compartment, even the root one, to check the status of the resources allocated on it, the same will be always limited to returning elements belonging to the first level of the structure, without searching in depth in the various sub-levels. Hence, it will not even be possible have a complete view of all the existing resources in the tenancy or in a portion of it at a given time.

OCI can be accessed using either the console or directly the APIs, which are typical REST APIs that use HTTPS requests and responses. The console itself relies on Oracle REST API to perform all the operation, including managing resources such as VCNs, compute instances, and block storage volumes. In this particular case, we are focusing on the API to list a certain type of resources. The reason behind the problem described before is that these APIs want as parameter, to be passed, the compartment ID, which is the OCID of the compartment where the research take place, and return only the first-level child resources in the parent compartment specified. The list therefore, does not include any resources in the sub-compartments, also known as grandchildren of the parent compartment. The logic behind the console provides only a single API call for the selected compartment, whereas it should perform a series of nested API calls, one for each children compartment. Because of that, the user, in order to retrieve also the information concerning the resources in the children compartments, should run by itself a search for any sub-compartment with a considerable waste of time.

2.1.2 Cost Monitoring

In a practical use case, the company Manager (which most of the time is also the the tenancy Administrator) intends to investigate about all the resources of a given type, as for example the compute instances deployed until now by the entire company. This investigation can be carried on mainly with the purpose of knowing the exact number, as well as the relative cost which each of them contributes to add to the total billing. Using the console, this task becomes a little bit tedious and bothersome to be completed, due to the fact that, as explained before, the user must perform a number of research equal to the number of sub-compartment; in addition to that, if the user needs to know also the cost, it must be performed a separate research in a different section of the web portal called "Cost Analysis" and dedicated to the cost monitoring. This section is not so particular easy to cope with, especially for a not trained user who approaches it for the first time, since it works like a queries generator and therefore it relies on a series of filter which can be applied based on the level of detail of the request. As long as the target remains the regular monitoring of the entire compartment billing estimation, this does not appear so complicated since the compartment name can be chosen among the list of all possible compartments within the tenancy. However, if the target becomes obtaining the cost produced by a single resource, this turns out to be not so trivial because the resource OCID, which uniquely identifies a resource within a tenancy (as said in section 1.2.3), must be provided. Since this identifier is pretty long, considering that it has to guarantee the uniqueness, it is difficult to be memorised and most of the time it should be taken from the section dedicated to the specific resource and later copied.

Another issue related to cost monitoring, is that, in order to have a complete overview of the total cost produced by a certain resource-type, two components should be considered, which are respectively Computing and Storage. The Computing part (for products with compute-based pricing) depends on the computational power used, so on the number of OCPU (representing physical CPU cores, as briefly described in section 1.2.4) and is shown by the console as result of the query. The other component is the Storage part which depends on the Gigabyte storage capacity per month and is related to both block and boot volumes, that can be attached to an instance respectively to dynamically expand the storage capacity (as described in section 1.2.5) and to store the operating system and boot loader required to launch the compute instance (section 1.2.4). Unfortunately, the latter is not returned by the query performed by the API call; so, in order to obtain it, it should be carried out a separate query for each volume attached to the instance and then add up all the values returned.

Finally, a tedious problem that many users have experienced is that it is not immediately possible to access important information about the requested resources, without avoiding to access the dedicated web pages. Effectively, when all the resources of a given type are listed, in the table is missing the information regarding the owner of the resource, so the user who has deployed it. This feature can be very useful to identify immediately the person concerned, whenever some anomalies are detected, like for example a change in the life-cycle state or in the Fault Domain. Moreover, there is no possibility to order the elements of the table based on an attribute value, which could be very useful in case of the user wants to carry out a comparison based on an attribute value, as for example the amount of RAM. It might also help, the possibility to have directly embedded in the table a button to either start or stop a machine or database, avoiding to enter in the resource personal page which is accessible only from a link.

In the console, actually, there are several ways to monitor the cloud resources. One of these is the "Tenancy Explorer" present in the Governance section, which allows to list all the resources belonging to a specific compartment, as said before not the entire list but only the first level. The element are displayed in a table, this time with also the possibility to order per column attribute, however this can results a little bit dispersive due to the fact that also Global Resources (see section 1.2.3), such as users, groups and policies, are displayed along with all the other resources. Moreover, this could create a little bit of confusion, especially for the new users who have not already learnt notions regarding these type of artifacts, and additionally could lead to commit mistakes or to result in misinterpretations. Another possible solution is searching per resource-type, by using the dedicated section present in the navigation drawer (e.g., Compute, Storage, Database, Identity and Security, etc.), in this way only the desired resources are displayed but all the related issues described before remain.

The goal is to create a solution that can be realized to integrate the services provided by the Oracle cloud, by simplifying the queries and providing additional functionalities.

2.2 Integrable services

If organizations want to have insight into the use and deployment of resources and services, there are some service and platform that can be integrated with OCI. To mention one of them, ServiceNow is for sure one of the go-to suppliers to consider, it is a digitalization and workflow management platform, born in 2004 with the idea of simplifying the management of IT services for companies by proposing an Information Technology Service Management (*ITSM*) solution. It already offers integration for other cloud operators such as AWS, Azure, Google Cloud, and for container environments such as Kubernetes and OpenShift.

ServiceNow has added OCI to the ServiceNow Service Management Portal and integrated OCI into its IT operations management (*ITOM*) visibility and discovery service, so customers can inventory and analyze usage of cloud resources within their OCI tenancy. Using the OCI REST APIs, the ITOM discovery service queries for all discoverable configuration items in the entire OCI stack, across all data centers. Configuration items include Compute resources (virtual and bare metal machines), cloud network resources, storage volumes, Oracle databases, and more. All Oracle Cloud discoverable cloud resources are extracted and stored in the ServiceNow Configuration Management Database (*CMDB*) repository, which can then be used to monitor availability of those resources for IT services, operations and support level management. Additionally, combining CMDB content with ServiceNow AIOps platform enables customers to create business service health dashboards, generate customized reports, and optimize spending on cloud usage [18]. Enterprise customers that have standardized on ServiceNow and have a multi-cloud strategy, are able to more easily manage their OCI resources via their existing ServiceNow Service Management portal. The latest integration means that companies can now have a single ServiceNow dashboard to manage their public cloud resources from Oracle, AWS, Azure and Google Cloud. This solution, however, is more suited for organizations that are moving their existing on-premise workloads to the cloud and chooses to adopt a multi-cloud strategy, where buyers do not want to be tied only to one provider but to allocate resources based on differing performance needs. That means matching the platform to the service requirements, in fact each enterprise decides to adopt a different mix of traditional and cloud services to achieve its goals. In addition, some organizations pursue multi-cloud strategies for data sovereignty reasons.

Certain laws, regulations and corporate policies require enterprise data to physically reside in certain locations. Multi-cloud computing can help organizations meet those requirements, since they can select from multiple IaaS providers, data center regions or availability zones. This flexibility also enables organizations to locate compute resources as close as possible to end users to achieve optimal performance and minimal latency [19].

For a company that adopts a single-cloud approach, addressing for instance only OCI as IaaS provider, this solution could be not the optimal one to follow. One alternative, could be a service with perhaps less features and functionalities implemented (features like for instance generating report or optimize spending), but more intuitive and easy to use and capable at least of overcoming the issues mentioned in the previous section.

The idea behind the thesis is to develop a simple and intuitive web application, leveraging a microservice-based architecture, which can help a company to monitor and manage the entire tenancy in a fast and secure way. This application must be oriented to the end user and must be able to immediately and promptly provide information on the metadata, as well as on the cost, of individual resources and the possibility of managing their status in real time through a convenient graphical interface.

The choice of the architecture, as well as the tools and technologies used, will be debated in the following chapter.

Chapter 3

Microservices and related technologies

In this chapter the main features of Microservices Architecture will be presented with the aim of providing the reasons behind the choice of this type of architecture, along with a brief introduction to the technologies used to deploy this application on OCI (i.e Docker and Kubernetes).

3.1 Microservices Architecture

Microservices Architecture (*MSA*) is a cloud-native architectural style, which is inspired by Service-Oriented Architecture (*SOA*), emphasizing self-management and lightweightness as the means to improve software agility, scalability and autonomy. Typically, microservices are organized as a suite of small granular services that can be implemented (developed, tested, and deployed) on different platforms through multiple technological stacks [20].

MSA has become popular in industry because of its benefits, such as availability, flexibility, scalability, loose coupling, and high velocity [21]. According to the International Data Corporation *(IDC)*, by the end of 2021, 80% of cloud-based applications will be developed using by MSA. Another published report reveals that organizations may adopt MSA for different purposes, for example, to gain agility (82%), to improve organization performance (57%), and scalability (78%).

3.1.1 Differences with other architectural styles

For the first time microservices term was discussed at a workshop of Software Architects in Italy on May 2011, in order to describe what the participants saw as a common architectural style recently explored by many of them. In fact, the microservices have been developed as a response to the problems in Monolithic applications or SOA that have put in difficulty the part of scalability, complexity, and dependencies of the application under construction, all using lightweight communication mechanisms [22] [23]. Each module, so each microservice, is implemented and operated as a small yet independent system, offering access to its internal logic and data through a well defined network interface. This increases software agility because every microservice becomes an independent unit of development, deployment, operations, versioning, and scaling [24]. In addition, any individual service of the MSA runs on its own process and communicate with each other through, e.g., RESTful or RPC-based APIs, furthermore it has a business capability that can utilize various programming languages, as well as data stores, and is developed by a small team [25]. Migrating monolithic architectures to microservices brings in many benefits including, but not limited to, flexibility to adapt to the technological changes (in order to avoid technology lock-in), better development, team structuring around services and, more importantly, reduced time-to-market [26].

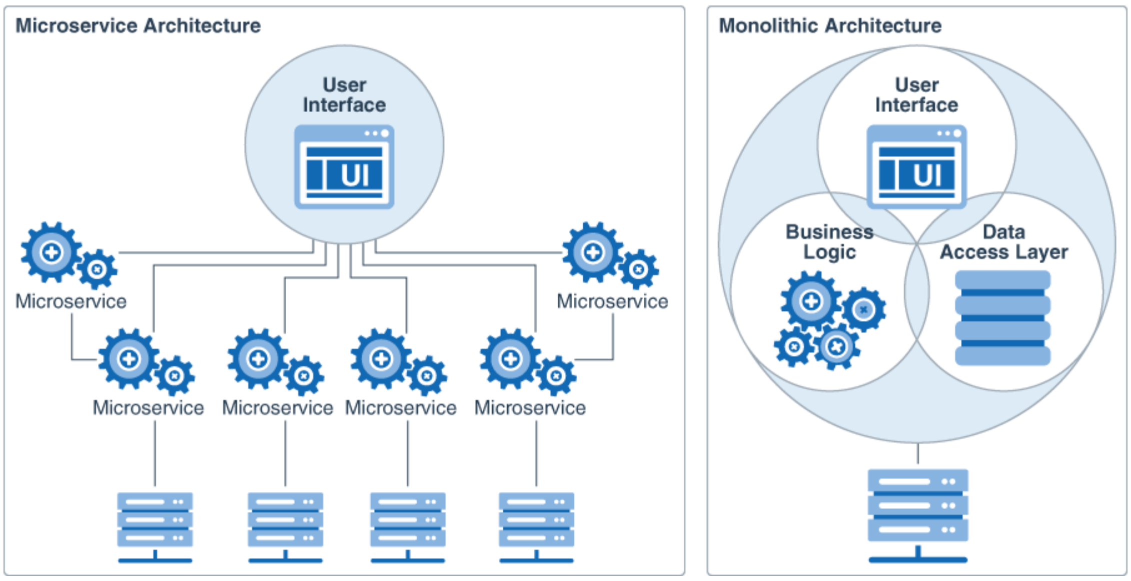


Figure 3.1: Differences between MSA and Monolithic Architecture

Monolithic Architecture

Monolithic applications can be successful but all the logic for handling a request runs in a single process, as shown in Figure 3.1, allowing to use the basic features of the language to divide up the application into classes, functions, and namespaces. However, increasingly people are feeling frustrations with them, especially as more applications are being deployed to the cloud. Consequently, a change made to a small part of the application requires the entire monolith to be rebuilt and deployed.

Since the monolith is a software application whose modules cannot be executed in an independent manner, the solution based on microservices must be regarded as the only one capable of executing independent instructions from one another [27]. The entire application is built as a single unit that contains all the business logic, on the other hand in the microservices architecture, the business logic is organized as multiple loosely coupled services.

The other relevant differences between these two architectural styles are highlighted in the following table:

Characteristic	**Microservices Architecture**	**Monolithic Architecture**
Unit design	The application consists of loosely coupled services. Each service supports a single business task	The entire application is designed, developed, and deployed as a single unit
Functionality reuse	Microservices define APIs that expose their functionality to any client, even other applications.	The opportunity for reusing functionality across applications is limited
Communication within the application	To communicate with each other, the microservices of an application use the request-response communication model. The typical implementation uses REST API calls based on the HTTP protocol	Internal procedures (function calls) facilitate communication between the components of the application. There is no need to limit the number of internal procedure calls
Technological flexibility	Each microservice can be developed using a programming language and framework that best suits the problem that the microservice is designed to solve	Usually, the entire application is written in a single programming language

Characteristic	**Microservices Architecture**	**Monolithic Architecture**
Data management	Decentralized: Each microservice may use its own database	Centralized: The entire application uses one or more databases
Deployment	Each microservice is deployed independently, without affecting the other microservices in the application	Any change, however small, requires redeploying and restarting the entire application
Maintainability	Microservices are simple, focused, and independent. So the application is easier to maintain	As the application scope increases, maintaining the code becomes more complex
Resiliency	If a microservice fails, the functionality offered by the other microservices continues to be available	A failure in any component could affect the availability of the entire application
Scalability	Each microservice can be scaled independently of the other services	The entire application must be scaled, even when the business requirement is for scaling only certain parts of the application

Table 3.1: The most relevant differences between the Microservices and Monolithic architectures.

These large monoliths become difficult to maintain in time and they evaluate with difficult due to their complexity, but a major disadvantage is that they limit the scalability of the product since the scaling of the entire application, rather than parts of it, requires greater resource (as depicted in Figure 3.2). Another problem, as it is shown in Table 3.1, comes from the fact that it does not provide fault resistance, and there is no possibility for a system component to work while another component does not work, which is possible with the microservice-oriented architecture.

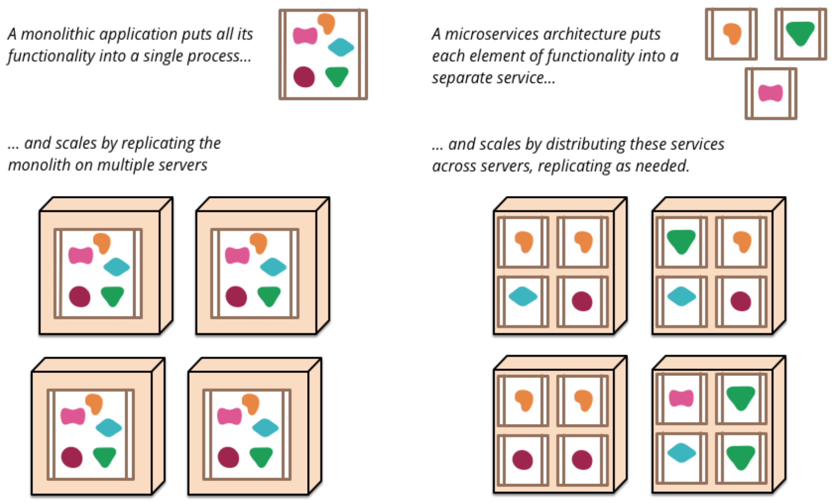

A monolithic application puts all its functionality into a single process...
A microservices architecture puts each element of functionality into a separate service...
... and scales by replicating the monolith on multiple servers
... and scales by distributing these services across servers, replicating as needed.

Figure 3.2: Scalability (Microservices vs Monolithic)

SOA

Both SOA and microservices suggest decomposing of systems into services available over a network and integrable across heterogeneous platforms. In the two approaches services cooperate to provide functionality for the overall system and thus both share the same goal, however, the path to achieving the goal is different. General use of SOA goes in the direction of preferring the design of system decomposition into simple services, emphasizing service integration with smart routing mechanisms for the entire company's IT. The smart routing mechanism provides a global governance or so called centralized management and is capable of enforcing business processes on top of services, message processing, service monitoring or even service control. Generally, the scope of the MSA is a single one application, while the scope of SOA is the set of all applications and software systems of an organization. Microservices are "tiny", this should suggest use of agile methods and practices for software development and communication through lightweight mechanisms, typically rely on REST, HTTP4, or other formats perceived as being lightweight and native for web development. SOA applications typically use heavyweight technologies such as SOAP and WSDL (Web Services Description) [28]. Finally, SOA is viewed mostly as an integration solution for large, complex, monolithic applications. As a result, a SOA application usually consists of a few large services, whereas a microservices-based application typically consists of many smaller services.

3.1.2 Decomposition into microservices

As with monolithic architectures and SOA architectures, the most difficult issue remains the decomposition of the system into services [29]. Each microservice represents a single specific business capability in a self-contained way (e.g., business logic, UI, persistent data management). A decomposition into microservices, based on technical capabilities, favours scalability and mutability better than a decomposition based on technical skills, moreover it is compatible with cross-functional development teams (or full stack) rather than with mono-functional development teams. Decomposition occurs iteratively and incrementally, by applying the **Strangler pattern**, which suggests to add an API Gateway in front of the application.

API Gateway

Microservices can provide their functions to other services through an API. However, the creation of end-user applications based on the composition of different microservices requires a server-side aggregation mechanism. The **API Gateway** emerged as a commonly recommended approach, it is the entry point of the system that routes the requests to the appropriate microservice, also invoking multiple microservices and aggregating results. It provides a tailored API to each client to route requests, transform protocols, and implement shared logic like authentication and rate limiters. In some cases, the gateway can also serve as load balancer since it knows the location and the addresses of all services. The main goal is to increase system performance and simplify interactions, thus reducing the number of requests per client. Since it acts as an entry point for the clients, it is preferable that these clients access the application through an integrated and unified access point, which is capable of routing their requests to the connected services, aggregating the required contents, and serving them to the clients. Figure 3.3 depicts a typical Microservices structure with API gateway and Data storage.

The gateway forwards requests from end users to the new microservices or to the old application, replacing iteratively and incrementally specific pieces of application functionality with new microservices, while at the same time these features are "switched off" from the original application.

Domain-Driven Design

The decomposition can also be driven by the application of **Domain-Driven Design** (*DDD*), that is an approach for building complex software applications, centered on the development of an object-oriented domain model [30]. DDD has two concepts that are incredibly useful when applying the microservice architecture: subdomains and bounded contexts. DDD defines a separate domain model for each subdomain, that is used to refer to the application's problem space. Subdomains are identified using the same approach as identifying business capabilities, so analyzing the business and identifying the different areas of expertise. DDD calls the scope of a domain model a **Bounded Context**. Bounded context is intimately linked to concepts like understanding what the service does, limiting the number of its functionalities to the strict minimum and understanding which are the entities that cannot be separated from each other. It includes also the code artifacts that implement the model. When using the microservice architecture, each bounded context is a service or possibly a set of services. It is possible to create a microservice architecture by applying DDD and defining a service for each subdomain [31].

The service is so defined as a standalone, independently-deployable software component, that implements some useful functionality and has an API that provides clients the access to its functionality. Consequently, the API consists of commands, queries, and events. A command (e.g.,*createOrder()*) performs actions and updates data, while a query (e.g., *findOrderById()*) retrieves the data. A service also publishes events, such as *OrderCreated*, which are consumed by its clients. Unlike in a monolith, a developer cannot write code that bypasses its API (encapsulating the service's internal implementation), as a result, the microservice architecture enforces the application's modularity.

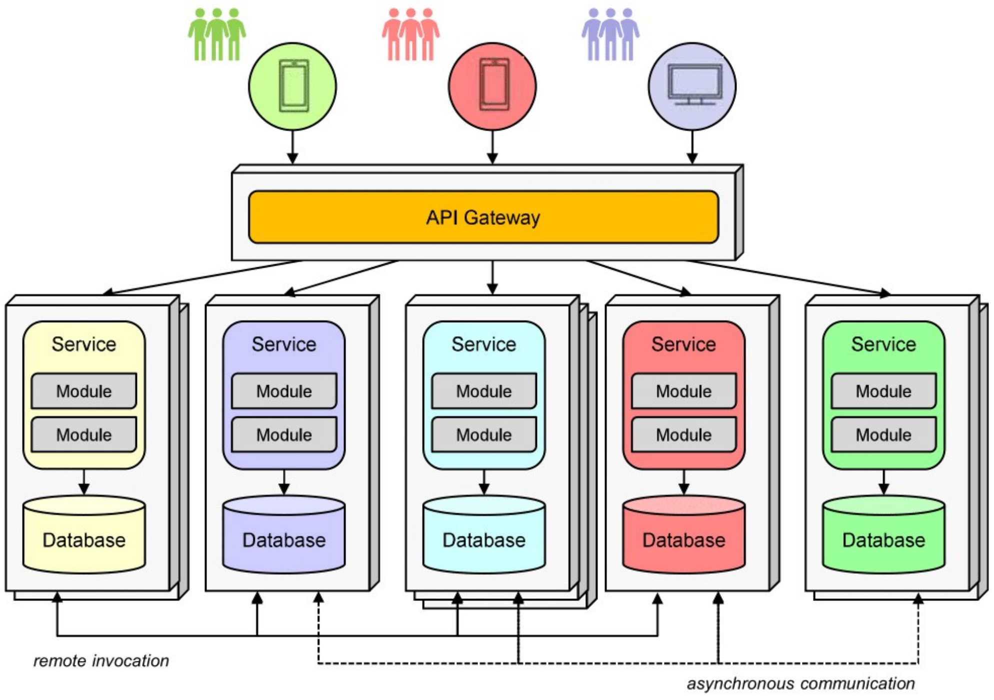


Figure 3.3: MSA with API Gateway

3.1.3 Deployment Strategies and Patterns

Typically, each microservice encapsulates storage and access to its data, using a separate database instance. Therefore, a microservice also deals directly with the data management for the business capacity it represents. The use of separate databases supports the autonomy of microservices (every microservice can manage their data using the most appropriate technology), as well as high cohesion and low coupling. However, decentralized data management also poses issues and raises challenges, if one microservice needs data from another microservice, it cannot access them directly with microservices.

A common solution is based on saga. Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions. An important benefit of asynchronous messaging is that it ensures that all the steps of a saga are executed, even if one or more of the saga's participants is temporarily unavailable.

In SOA, as well as in MSA, the main services are coordinated using two methods:

- **Orchestration**: where there is a central microservice that will send requests to other services and supervise the whole process, by sending and receiving responses;
- **Choreography**: which does not suppose any centralization, but each service knows in advance what it has to do.

Choreography distributes the decision making and sequencing among the saga participants. They primarily communicate by exchanging events, while Orchestration sends command messages to saga participants telling them which operations to perform.

Orchestration-based sagas have several benefits:

- It does not introduce cyclic dependencies;
- Less coupling;
- It improves the separation of concerns;
- It simplifies the business logic.

The saga orchestrator invokes the saga participants, but the participants do not invoke the orchestrator. As a result, the orchestrator depends on the participants but not vice versa, and so there are no cyclic dependencies. Each service implements an API that is invoked by the orchestrator, so it does not need to know about the events published by the saga participants. The saga coordination logic is localized in the saga orchestrator and the domain objects are simpler and have no knowledge of the sagas that they participate in.

In this case, an Orchestration approach is adopted in order to guarantee that all the logic is managed by a single orchestrator service, which receives the requests from the GUI or from the API gateway, and redirects these requests to the specific microservice involved through the API exposed. The bounded context for this application is based on the subset of functionalities provided for each type of OCI resource, that the application allows to monitor or to manage directly. Therefore, for each subdomain identified as a functionality or a subset of functionalities, a different microservice will be created.

Running microservices in the cloud requires a new approach, one that aligns with its core philosophy that is flexibility, and embraces new technologies and software development methodologies like cloud computing, containers, and continuous integration/delivery.

3.2 Containers

Using containers offers lots of advantages; their portability and resource efficiency provide service isolation and accelerate the development process. As you build new cloud-native applications, using microservices, or migrate existing ones to this new environment (along with building, deploying, and scaling these applications), will present its own set of challenges. Docker and are respectively the most pervasive container system and orchestration platform to address these challenges; their existence has made the tasks of managing complex service deployments and scalability, remarkably easy.

Difference with VMs

VM is best described as a software program that emulates the functionality of a physical hardware or computing system. It runs on top of an emulating software, called hypervisor, which replicates the functionality of the underlying physical hardware resources with a software environment. It can be identified as what sits between the hardware and the VM, and is necessary to virtualize the server. Within each VM, it runs a unique guest operating system. However, VMs with different operating systems can run on the same physical server, for instance a UNIX VM can sit alongside a Linux VM, and so on. Each VM may also contain the necessary system binaries and libraries to run the apps but the actual OS, however, is managed and executed using the hypervisor.

VMs operations are typically resource intensive and do not allow individual app functionality to run in isolated PC-like virtualized environments, unless a separate VM is used for different modular elements of the app. If an app workload needs to migrate between different VMs or physical data center locations, the entire OS needs to migrate along with it. The workload operation rarely consume all the resources made available to the associated VM. As a result, the remaining unused resources many not be incorporated in a capacity planning and distribution across all VMs and workloads. This leads to a an inaccurate planning and significant resource wastage, even though virtualization was developed specifically to optimize the usage and distribution of hardware resources within a data center.

Monolithic app development practices are losing popularity and organizations are pursuing infrastructure architecture solutions, to further optimize hardware utilization. This is precisely why containerization was invented and gained popularity as a viable alternative. In fact, containerization creates abstraction at OS level that allows individual, modular, and distinct functionality of the app to run independently. As a result, several isolated workload can dynamically operate using the same physical resources.

Containers create several isolated OS environments within the same host system kernel, which can be shared with other containers dedicated to run different functions of the app. Only bins, libraries, and other run-time components are developed or executed separately for each container, which makes them more resource efficient as compared to VMs. Containers do not virtualize anything and directly address the OS resources, which will be segmented and divided for each container. In the container approach, there is a Container Engine which runs on the kernel and allows to group one or more processes, dedicating them an unlimited number of resources (e.g., CPU, networking, storage) and creating an independent executive context for each container. The kernel is shared, the processes that run in a container exploit the File System dedicated to that container but run directly on the kernel of the original OS, which supports everything. A complete application component can be executed in its entirety within its isolated environment without affecting other app components or software. Conflicts within libraries or app components do not occur during execution, and the application container can move between the cloud or data center instances efficiently. To better understand these differences, Figure 3.4 can be observed. Containers are particularly useful in developing, deploying, and testing modern distributed apps and microservices, that can operate in isolated execution environments on same host machines.

3.2.1 Docker

Docker is a new container technology that has become very popular because it is suitable for building and sharing disk images and enables users to run different operating systems such as Ubuntu, Fedora, and Centos. It is often used when a version control framework is required for an application's operating system, to distribute applications on different machines, or to run code on laptop in the same environment as on the server. In general, Docker will always run the same, regardless of the environment in which it will be running, by guaranteeing that application microservices will run in their own environments that are completely separate from the operating system. Moreover, its portability and lightweight nature make it easy

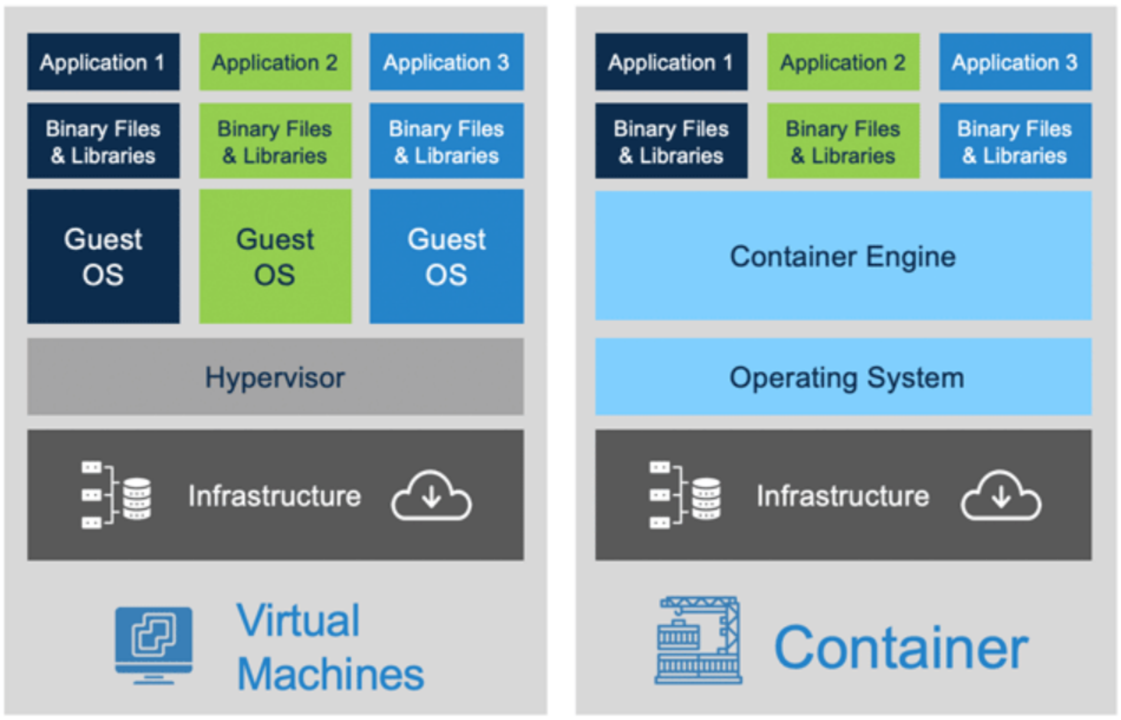


Figure 3.4: Comparison between VMs and Containers

to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate in near real time. These properties are fundamental for the purpose of this work, since the application should be available and easily reachable by anyone who needs to use it.

The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands. Docker speeds up the development life-cycle by allowing developers to work in standardized environments, using local containers which provide applications and services [32]. Docker containers are not created out of thin air, but they are instantiated from Docker images, which serve as blueprints for containers; they contain application code, libraries, tools, dependencies, and other files needed to make an application run [33]. Docker builds images automatically by reading the instructions from a **Dockerfile**, which is simply a text-based script of instructions that is used to create a container image. Usually, it is provided with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When Dockerfile is changed, and consequently the image rebuild, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies [34].

3.3 Kubernetes

Kubernetes is an open source container cluster manager, that complements and extends Docker's software encapsulation power and makes it easier to organize and schedule containerized applications across a fleet of machines [35]. Kubernetes have become the "de-facto" standard for container orchestration and has also won the race for being the most loved platforms among developers. Released in 2014 by Google, Kubernetes has come a long way, now recent reports state that out of 109 tools to manage containers, 89% of them are leveraging Kubernetes versions. This orchestrator is lightweight, modular and suited for the cloud architecture, moreover it offers several features like:

- Portability: It can be run on almost any platform with different local machine solutions;
- Scheduling: It provides the possibility to automatically schedule microservices on various machines;
- Health Check: It determines when to restart a service if an error occurs, monitoring the state of the applications by means of one of its controllers;
- Resources optimization: It increases infrastructure utilization through the efficient sharing of computing resources across multiple processes;
- Autoscaling: It automatically increases or decreases the number of a single microservice instance in a deployment, replica set or stateful set, based on custom metrics;
- Load Balancing: It efficiently distributes client requests across multiple instances of same microservice, in order to avoid overloads, optimize performance and ensure the reliability of the application;
- Service discovery: It offers the possibility to contact services with consistent DNS names instead of IP addresses.

Cluster and Control Plane

A Kubernetes cluster is a set of nodes that run containerized applications. The worker nodes are the components that run containers, they can either be VMs or physical computers, all operating as part of one system. They perform tasks assigned by the master node, which is the node controlling the state of the cluster. The master node is the origin of all task assignments, it keeps the ranks of the worker nodes with a control plane, knowing their overhead status and decides how to distribute the workloads within the cluster. Figure 3.5 shows a high level diagram of the Kubernetes cluster. Kubernetes automatically manages clusters to align with their desired state, through the Kubernetes control plane. Responsibilities of a Kubernetes control plane include: scheduling cluster activity and responding to cluster events. The Kubernetes control plane runs continuous control loops to ensure that the cluster's actual state matches its desired state [36].

kube-api-server: is the gateway to the Kubernetes cluster. It is the central touch point that is accessed by all users, automation, and components in the Kubernetes cluster. The API server implements a RESTful API over HTTP, performing all API operations, and is responsible for storing API objects into a persistent storage back-end. All requests go through the API server, including the requests that are used to configure the system. Users access the Kubernetes API using: kubectl (the Kubernetes command-line tool which allows to run commands against clusters), client libraries, or by making REST requests.

Pods: are the smallest deployable unit managed within a Kubernetes cluster. A pod can host one or more containers inside, which contribute to provide a specif service. The containers share storage and network resources and is possible to access them from the outside using the same IP address.

Service: is an abstraction which enables a group of pods to be assigned a name (label) and unique IP address. As long as the service is running, that IP address will not change. The pod instances can be exposed externally by the service, like they were a single microservice. Thereby, it is easy to identify each microservice through the label, so that the service can deal with, by balancing the requests among the different pod replicas without anyone noticing from the outside.

Components

kube-scheduler: is the default scheduler and runs as part of the control plane. It watches for newly created pods, that have no node assigned, and it is in charge of deciding the best node for that pod to run on. The scheduling decisions are based on factors like: CPU/memory usage and compatibility.

kubelet: is the primary agent that runs on each node. First, it registers the node with the API server and then, once received the order to create pods from the master node, it ensures that those pods are started and run.

kube-proxy: runs on each node and allows pods to communicate each other and with external world. It receives the calls from outside and redirects them to the correct pod.

cloud-controller-manager: is a software component, part of the control plane, in charge of interacting with the underlying cloud infrastructure. In addition, it creates a connection between the nodes that are within the cluster and the APIs of the cloud provider.

etc: is a key-value database, which is used by Kubernetes to save all cluster information and configuration.

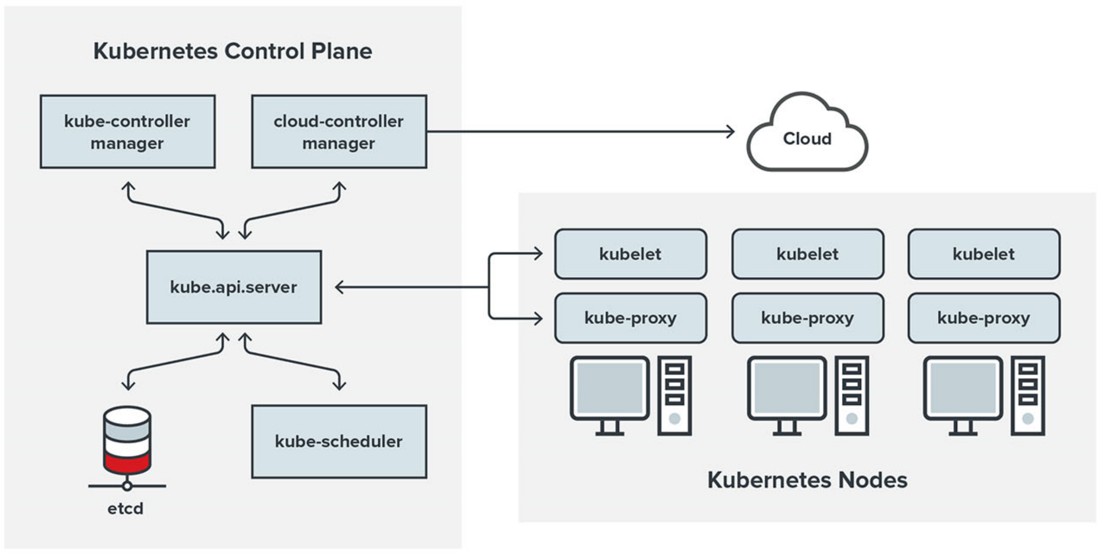


Figure 3.5: Kubernetes cluster diagram

When a cluster is created, an address range is assigned by Kubernetes, so that it can give pods their own IP addresses. Kubernetes also assigns an IP address (the pod IP address) to the virtual network interface in the pod network namespace, from a range of addresses reserved for pods on the node. For different reasons (e.g., node pool upgrade, a change in the pod declarative configuration, a change in a container's image and unavailability of a node) a pod can be destroyed. Since Kubernetes regularly tears down and recreates pods, their IP addresses cannot be reliable, so it provides stable IP addresses using Services. An arbitrary key-value pairs called labels is assigned to any Kubernetes resource. Labels are used to group multiple related pods into a logical unit, which is the service. A service has a stable IP address and ports, and provides load balancing among the set of pods whose labels match all the labels defined in the label selector [37]. Kubernetes assigns a stable and reliable IP address to each newly-created service (the ClusterIP) along with an hostname, by adding a DNS entry. The ClusterIP and hostname are unique within the cluster and do not change throughout the life-cycle of the service. Healthy pods running customer applications, can be reached using either the ClusterIP or the hostname of the service. Kubernetes spreads traffic, as evenly as possible, across the full set of pods running on many nodes, so a cluster can withstand an outage affecting one or more, but not the totality, of nodes.

By default, pods do not expose an external IP address, because kube-proxy is able to manage all traffic on each node. In order to support communication between the different pods, Cluster DNS is provided. Cluster DNS is an additional DNS server and specifically takes care of DNS records for Kubernetes services, so the service name and IP address. Consequently, microservices are made available to all the cluster nodes by means of the service name. Moreover, an endpoint resource object is referenced by a Kubernetes service, so that the service has a record of the internal IPs of pods in order to be able to communicate with them. The controller for the service selector continuously scans for pods that match its selector, and then posts any updates to the endpoint object accordingly.

3.3.1 OKE

Oracle Cloud Infrastructure Container Engine for Kubernetes (*OKE*) is a service that helps to deploy, manage, and scale Kubernetes clusters in the cloud. It enables to deploy and run highly available and scalable microservices-based applications. Furthermore, it combines the elasticity and utility of public cloud with the granular control, security, and predictability of on-premises infrastructure to deliver high-performance and cost-effective infrastructure services. When OKE launches a cluster, it creates control plane and worker nodes in a node pool along with all of the network resources needed for that cluster, including a Virtual Cloud Network.

In order to deploy the Microservices-based system, an OKE cluster with three worker nodes was created in the "tecitgen2" tenancy. This tenancy, located in the Region of Germany Central Frankfurt, is assigned and fully available to Technology Reply. Each node is a VM with the following shape (whose resources are described in section 1.2.4):

- 1 OCPU;
- 16 GB RAM;
- 1 Gbps of Network Bandwidth;
- 1 Boot Volume.

Moreover, each boot volume is provided with 47 GB size and contains a custom image "Oracle-Linux-7.92", launched in Native mode in order to communicate directly with the VM's underlying hypervisor and to better perform than a VM launched in PV mode (as explained in section 1.2.4). Each node is located in a different AD within the same region, in order to isolate a possible failure and preserve the cluster availability and integrity (see section 1.2.1). In addition, a regional subnet, with 1 Tbps of bandwidth and a latency of less than 5,000 microseconds (as outlined in section 1.2.1), was made available to allow pods on one worker node to communicate with pods on other worker nodes. Originally subnets were designed to cover only one AD in a region, so they were all AD-specific, which means the subnet's resources were required to reside in a particular AD. Now subnets can be either AD-specific or regional (as depicted in Figure 3.6) and both types of subnets can co-exist in the same VCN [38]. Oracle recommends using regional subnets, which spans all three ADs in a multi-AD region, because they make it easier to efficiently divide VCN into subnets; in fact, they are more flexible and designed for AD failure. Each subnet has a contiguous range of IPs, described in CIDR (Classless Inter-Domain Routing) notation, that cannot overlap each other. In this case, the subnet is provided with IPv4 CIDR Block: 10.0.10.0/24.

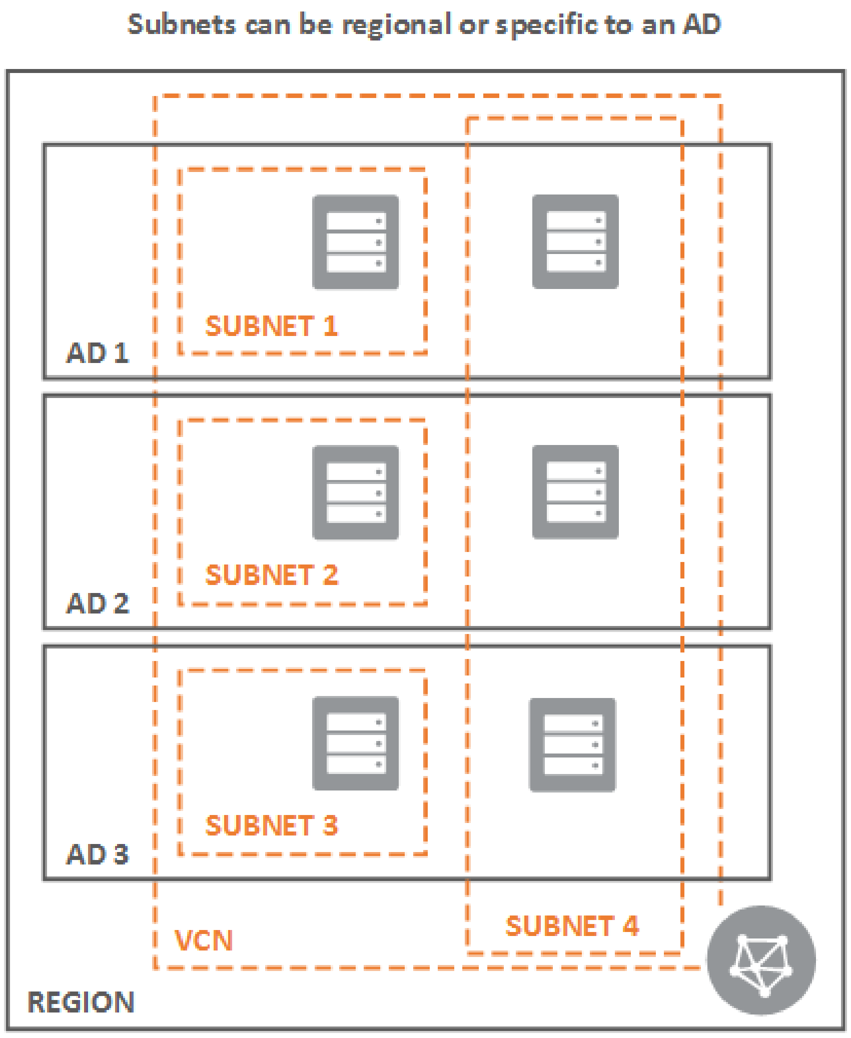


Figure 3.6: OCI subnets

Chapter 4

System implementation

The following chapter of the dissertation moves on to describe the overall architecture of the system along with a brief description of the different platforms used. Finally, the implementation details of the adopted solution will be listed.

4.1 Architecture

The idea behind the microservices-based system is to develop a full stack application based on MSA; full stack refers to an entire computer system or application from the front-end to the back-end and the code that connects the two. The back end of a computer system encompasses "behind-the-scenes" technologies such as the database and operating system, while the front-end is commonly the user interface (UI). In this implementation, the front-end side is realized with Angular and HTTP Client, while the back-end side uses Spring Boot with Spring Web MVC for the REST Controller and Spring Data MongoDB for interacting with MongoDB database. Then, to simplify the deployment of the whole application on OKE, Docker is used to create static images of both. Each image will be built using a different Dockerfile and later will be pushed to Oracle Cloud Infrastructure Registry (also known as Container Registry), using the Docker CLI. Figure 4.1 describes the full stack architecture overview.

The development stack of this application composes of the following:

- Angular (7.1);
- Node.js (8.11);
- Java +8 ;
- Spring Boot 2.4.5 (with Spring Web MVC and Spring Data MongoDB);

- Maven 4.0;
- Docker;
- Database(MongoDB).

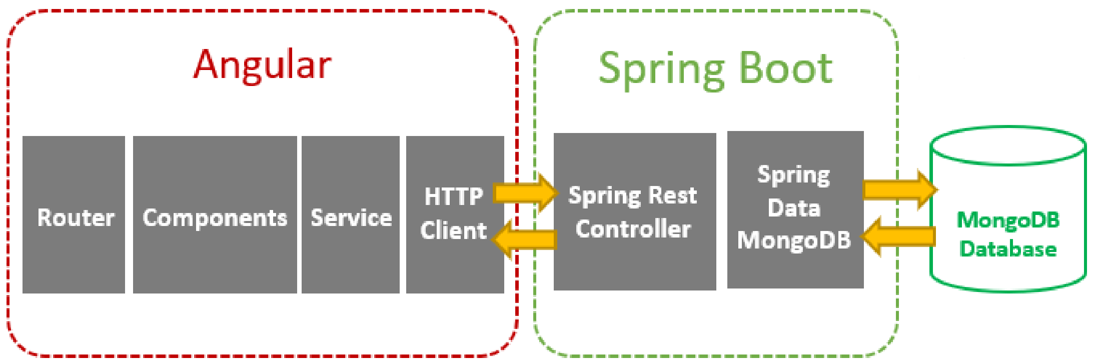


Figure 4.1: Architecture overview of proposal solution

The application workflow is the following:

Spring Boot exports REST APIs using Spring Web MVC and interacts with MongoDB Database using Spring Data MongoDB. For its part, Angular Client sends HTTP Requests and retrieves HTTP Responses using Http Client Module. In addition, it shows data on the Components and uses Angular Router for navigating to pages.

The back-end configuration is based on Orchestration saga, where (as fully explained in subsection 3.1.3) each service implements a subset of API that is invoked by the orchestrator service, which holds all the logic. The orchestrator receives all the requests from front-end, and redirects these requests to the specific microservice involved. In this case, based on the entire set of functionalities required by the system, several subsets will be spotted and for each of them a microservice will be instantiated. For each microservice a separate Spring project will be created with its own controller and will run on a different port. The front-end is simply a GUI which must request, order and present the information that the end user needs. To do this, it communicates with the back-end orchestrator service that first requests and then aggregates data from all the instances of the specific microservice involved, on behalf of the front-end app. Therefore, the interface has to call the list of REST API developed at back-end to access the differentiated services provided by the architecture. Figure 4.2 highlights the Orchestration pattern used in the microservices-based architecture. In order to speed up the response time of the REST APIs provided and keep the latency to a minimum, the back-end controller will interact with a MongoDB instance, located in the same tenancy of OKE cluster. The DB will store all the information that does not need to be shown updated in real time and which would imply a considerable decrease in response times of individual calls. For the other information, such as the Life-cycle State, which can change any minute now, it will inevitably be necessary at each research to call up the external Oracle APIs to request this information in real time.

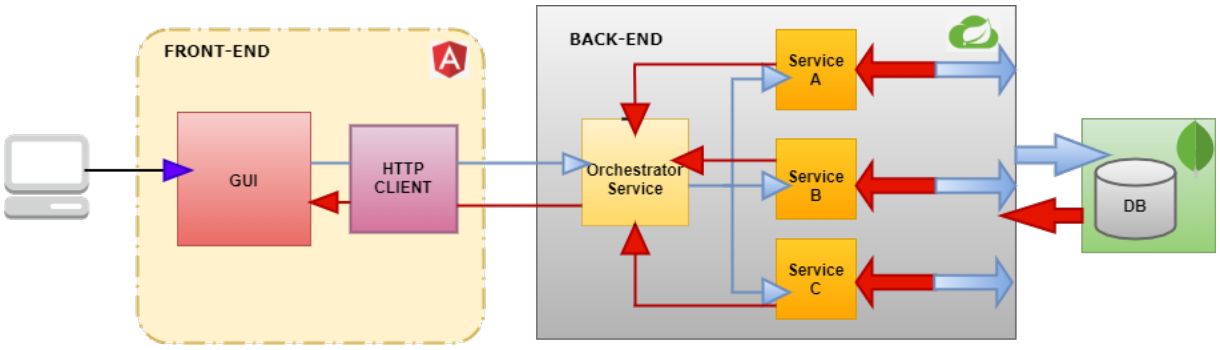


Figure 4.2: System workflow

4.2 Back-end

Spring Boot is the simplest solution to program and test applications on the Spring platform, producing stand-alone applications executable in JAVA. It is an open source framework offering a dependency injection feature, which lets objects define their own dependencies that the Spring container later injects into them. This enables developers to create modular applications consisting of loosely coupled components that are ideal for microservices and distributed network applications.

Spring Boot and Maven

Spring Boot is widely used to create Java applications that can be started by using java "-jar", it can also build a single executable jar file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development life-cycle, across different environments, and so forth [39].

Spring Boot is compatible with Apache Maven 3.3 or above, which is a Java-based software project management tool. Maven uses a construct known as the Project Object Model (POM); an XML file describing the dependencies between the project and the various library versions needed. This separates the libraries from the project directory, by using this description file to define their relationships. Maven looks for the POM in the current directory, it reads the file and gets the needed configuration information, then executes the goal. In addition, it automatically downloads Java libraries and Maven plug-ins from the various defined repositories, by downloading them locally or in a centralized repository. This allows to recover the various jar files in a uniform way and to be able to move the project independently from one environment to another, having the certainty of always using the same versions of the libraries.

For example, the "spring-boot-starter-web" dependency allows to build Spring-based web applications with minimal configuration, by adding all the necessary dependencies, such as the Apache Tomcat web server to the project. An embedded Tomcat server consists of a single Java web application along with a full Tomcat server distribution, packaged together and compressed into a single JAR, WAR or ZIP file. Moreover, it offers a way to package Java web applications that is consistent with a microservices-based approach to software development. It also makes it easier to distribute Java web applications through Docker containers and manage them through a container orchestration service, such as Kubernetes. Hence, the "spring-boot-starter-web" is a starter for building web (including RESTful) applications using Spring MVC.

4.2.1 REST Controller

A RESTFul application follows the REST architectural style, which is used for designing networked applications. RESTful applications generate HTTP requests performing CRUD (Create/Read/Update/Delete) operations on resources and typically return data in JSON or XML format. REST web service is inherently stateless, so it forgets about the session concept (it has no memory) and each request is handled independently. In a REST web service the resources are identified by URI and HTTP requests, directed to these addresses, allow to interact with the exposed resources and performing various operations on them.

As for a classic web application, also in this case Spring allows to manage requests sent to the server through objects of the controller type. In the case of a web service, however, it is possible to exploit a specific type of component that further helps in the implementation of our resource controllers. These components are identified by the annotation "*@RestController*".

Spring MVC framework is, like many other web MVC frameworks, request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. The default handler is based on the "*@Controller*" and "*@RequestMapping*" annotations, offering a wide range of flexible handling methods. When you run an application, Spring Boot will detect that a Spring MVC controller is present and start up the embedded Apache Tomcat instance, by default.

The Application class in a Spring boot project is very important, because it is used to bootstrap and launch the Spring application from a Java main method. The "*main()*" method uses "*SpringApplication.run()*" method to start the whole Spring

Framework, which in turn, starts the auto-configured Tomcat web server. The "*@SpringBootApplication*" annotation enables auto-configuration and component scanning. During the scanning process, the "*@RestController*" annotation is looked up and a Spring bean is created from the controller class.

ApiController.java

```java
@RestController
@SuppressWarnings({ "unchecked", "rawtypes" })
@RequestMapping(value = "apis", produces =
    {MediaType.APPLICATION_JSON_VALUE, "application/hal+json"})

public class ApiController {

    @Autowired
    ApiService apiService;

    @PostMapping("/listInstances")
     ResponseEntity <Instance_list> list_instances(@RequestBody
         Compartment_ids c_ids) throws ApiNotFoundException,
         IOException, InterruptedException {
        return new ResponseEntity(apiService.list_instances(c_ids),
            HttpStatus.OK);
    }

    @GetMapping("/getCompartments")
    ResponseEntity <Compartment_list> getCompartments () throws
        ApiNotFoundException, IOException {
       return new ResponseEntity(apiService.getCompartments(),
           HttpStatus.OK);
    }

    @GetMapping("/stopInstance/{id}")
     String stopInstance (@PathVariable String id) throws
         ApiNotFoundException, IOException {
                return new ResponseEntity(apiService.stopInstance(id),
                    HttpStatus.OK);
        }

    @GetMapping("/startInstance/{id}")
     String startInstance (@PathVariable String id) throws
         ApiNotFoundException, IOException {
                return new ResponseEntity(apiService.startInstance(id),
                    HttpStatus.OK);
        }
        ...
```

Listing 4.2.1 shows part of the ApiController class of the orchestrator service. "*@RestController*" indicates that the data returned by each method will be written straight into the response body instead of rendering a template. Luckily, it is not limited to returning Strings from the methods and it is possible to return objects as well. Spring will automatically convert the object to JSON and set the appropriate Content-Type header. "@RequestMapping("/api")" declares that all APIs' url in the controller will start with */api*, so "*list_instances*" method will automatically be called once a POST request to the specified path (apis/listInstances) is made. "@PostMapping(value="/listInstances")" annotation ensures that HTTP POST requests to */listInstances* are mapped to the "*list_instances*" method, similarly for @GetMapping and so on.

"*list_instances()*" method is called to return all the information concerning the compute instances existing within a specific compartment. "*c_ids*" is a parameter passed in the request body of the HTTP POST request, simply a Java class, part of the Model, having as attribute the list of all the compartments IDs (so the OCIDs) of the sub-tree (having as its root the compartment indicated by the client). The controller will return a new instance of the "*Instance_list*" class, which has a nested list of class as parameter. Each class represents a single compute instance and will contain all the metadata concerning that instance, such as: shape, Life-cycle state, amount of RAM, number of OCPUs, owner, creation date, computing cost, storage Cost, etc.

A key difference between a traditional MVC controller and the RESTful web service controller, is the way that the HTTP response body is created. Rather than relying on a view technology to perform server-side rendering of the compute instance data to HTML, this RESTful web service controller populates and returns a "*Instance_list*" class object, that will be written directly to the HTTP response as JSON.

"*getCompartments()*" method instead, has to look for all the compartment currently available within the tenancy and it returns in the response body the "*Compartment_list*" object, which substantially is a list of Compartment classes. This class has as attributes the name and the ID of the compartment along with the parent-compartment name (which is useful to understand where the compartment is located in the tree structure). The list of names will be later displayed in a Dropdown menu placed in the GUI, in order to allow clients to select the compartment for further researches.

"*stopInstance()*" and "*startInstance()*" methods are called to either start or stop a compute instance (whose ID is passed as part of the query string in the GET request), so to change its Life-cycle state. In this case, the responses will contain only a String to either indicate if the operation was successful or not.

As can be easily noted, an "ApiService" class is declared and tagged with "*@Autowired*" notation. This notation is used to inject ApiService bean to local variable "*apiService*". By declaring all the bean dependencies in a Spring configuration file, Spring container can autowire relationships between collaborating beans; this is called Spring bean autowiring. As a result, when a Spring Boot application run, it will automatically scan the components in the current package and its sub-packages. Thus, it will register them in Spring's Application Context, and allows to inject beans using "*@Autowired*" [40].

4.2.2 REST Service

REST Services are the class file which contains "*@Service*" annotation. These class files are used to write business logic in a different layer, on behalf of the REST controller. For each method of the controller, a method in the service class is implemented in order to process the request and returning to the controller the requested data. In order to retrieve the information requested by the client, the back-end must necessarily rely on Oracle APIs.

Oracle REST API

All OCI APIs use standard HTTP requests and responses. Each response includes a unique Oracle-assigned request ID in the opc-request-id response header, that can be provided to Oracle in order to contact it about a particular request.

Call an Oracle API is not so trivial, because the request must be signed for authentication purposes. Normally, the steps required to sign a request are:

- Form the HTTPS request (SSL protocol TLS 1.2 is required);
- Create the signing string, which is based on parts of the request;
- Create the signature from the signing string, using the private key and the RSA-SHA256 algorithm;
- Add the resulting signature and other required information to the Authorization header in the request.

In the Authorization header of the request, there must be set the signing string which has the following format:

```
keyId="<TENANCY OCID>/<USER OCID>/<KEY FINGERPRINT>"
```

As consequence, a configuration file is required in order to retrieve these information. The basic entries of the configuration file are:

- OCID of the user calling the API;
- Fingerprint for the public key that was added to the user;
- Full path and filename of the private key (in PEM format);

- OCID of the user tenancy;
- OCI region.

The following piece of code shows all the steps to carry out a call to an external Oracle API in the proposal solution.

`ApiService.java`

```java
this.requestSigningFilter = RequestSigningFilter.fromConfigFile
                          (this.configurationFilePath, profile);

this.client = ClientBuilder.newBuilder().build().
             register(requestSigningFilter);

WebTarget target = this.client
                  .target("https://database.eu-frankfurt-1.oracleclo.com")
                  .path("20160918")
                  .path("autonomousDatabases")
                  .queryParam("compartmentId", ids[i]);

Invocation.Builder ib = target.request();
Response response = ib.accept(MediaTyp.APPLICATION_JSON).get();
```

In order to perform this authentication procedure on a Java application, two third-party external libraries must be added to the Spring boot project, by including respectively the following dependencies in the POM file: "javax.ws.rs" and "com.oracle.oci.sdk".

The latter library allows to create a new "*RequestSigningFilter*" instance, which is able to compose the signing string and to create the digital signature, using the private key (whose path is specified in the configuration file provided) to encrypt the string. The Filter will add the authentication header to each following request. The next step is to create a Jersey Client defined in the "javax.ws.rs.client" package, which provides a high-level API for accessing any REST resources.

Then, a WebTarget object is created through the Client interface "*target()*" method, providing the URI of the target REST resource. WebTarget has additional methods to extend the URI originally constructed, by adding path segments or query parameters. After having applied the configuration options to the target, the "*request()*" method is called to begin creating the request. This last method returns an instance of "*Invocation Builder*", which is a helper object that provides methods for preparing the client request. Finally, after setting the accepted media response type, it invokes the request, by calling one of the methods of the "*Invocation Builder*" instance that corresponds to the type of HTTP request, the target REST resource expects.

Listing 4.2.2 is an example of what said previously, the "*RequestSigningFilter*" and the Client instances will be used for all the API calls performed by the service class. In the "*target()*" method is specified an endpoint URI to invoke an external REST API, this API is designed to retrieve all the metadata concerning the Autonomous Databases available within the compartment, whose ID is specified in the query string of the target instance. In this case, the target REST resource is for an HTTP GET request, so the Invocation Builder calls the "*get()*" method. The return type of the returned entity is set to JSON because it must correspond to to the entity returned by the target REST resource.

In the proposal solution, the service implements several methods, each one mapped to a controller API and in charge of accomplish a specific tasks related to one of the functionalities provided by the GUI. In many of these methods, shall be carried out many API calls in order to retrieve the metadata concerning a specific resource. In fact, as explained in section 2.1, the Oracle APIs allow only to supply data concerning the first-level child resources in the parent compartment specified. So, each method should perform a series of API calls, one for each sub-compartment in the hierarchy. In the most complicated scenario, that is when a user intends to list all the resources throughout the entire hierarchy of the tenancy, so when the root compartment is provided, a single method should perform at least twenty or more calls in order to retrieve all the data. This leads essentially to a considerable waste of time, because each call has no negligible response time (in the order of seconds) so the total API latency of the Controller (the total amount of time that it is taken by an API system to respond to an API call) takes an unacceptable value, so it provides a bad QoS to the end user.

The MSA helps us in this situation, because the orchestrator service, as shown in Figure 4.2, can simultaneously make several calls (one for each sub-compartment) to the same API, that each instance of the specific microservice involved exposes, but passing a different compartment ID each time. Once they have all returned, it can aggregate the results and return to the client the complete list. In the developing phase no effect can be seen, because there is only one instance of the service that runs locally (in the port specified in the "application.property" file); but once the system will be ready for being deployed on OKE, all the API calls should be directed towards the specific Kubernetes Service which, as explained in section 3.3, will balance the requests across the multiple instances of same microservice, so the different pod replicas, that will run on the cluster. This will bring a tangible reduction in the total response time and will allow to fully exploit all the potentialities of the MSA.

Another scenario that leads to increase the total latency, is that, in order to supply also the cost produced by a cloud resource along with the other metadata, a separate call should be made to a different API endpoint for each sub-compartment in the hierarchy. This will produce a JSON which lists the cost produced by each single resources that belong to that compartment (identified by the OCID) and when a match is found between the two OCIDs, the value is inserted in the resource metadata.

In addition (as in mentioned in subsection 2.1.2), for the cloud resources as the compute instances, that have both a Computing and a Storage cost, two additional calls need to be made in order to retrieve all the Block and Boot volumes that are attached to those instances.

4.2.3 Interaction with MongoDB

In order to quickly and promptly retrieve the desired information, the idea is to adopt a caching strategy, by adding an interaction with an external DB provided by the company. The DB will store all the information that does not need to be shown updated in real time, including the costs (due to the fact that, through the Oracle APIs, it is only possible to obtain cost information updated at most to the day before) and will supply that information to the method that will require it. In order to keep data updated within the DB, a Scheduler class will be created with the task of periodically refreshing all the data stored in the DB, at a specific date and time.

MongoDB is a document-based NoSQL database, providing high performance and high availability. It is quite fast and can handle large amounts of structured and unstructured data, making it a database of choice for web applications. Data is stored as documents in BSON format, making their retrieval quite easy [41]. These documents are stored in a collection, which, in turn, are held in databases. Spring Data MongoDB automatically maps the model (so the Object class) with the collection only when the name of the model and collection are same.

The Spring framework provides powerful connectors to easily perform database operations with MongoDB. The interaction is based on the **MongoRepository** interface, which is used for basic queries that involve all or many fields of the document. It is therefore necessary to create a Repository class, extending the MongoRepository, for each type of resource you want to store in the DB. Moreover, the Repository must be provided with the corresponding Java class that is associated with that resource and that will be mapped to the respective collection in the DB. This interface comes with many operations, including standard CRUD operations. It is also possible to define custom queries by declaring their method signatures, like the following one:

```
List<Autonomous_Database> findByCompartmentId(String compartment_id)
```

This query essentially seeks resources of type Autonomous Database and finds the ones that match on "*compartment_id*". This results very useful, because it allows to each microservice, which has to return only data concerning a specific compartment, to retrieve all the records in the corresponding collection that meet this condition. However, there are metadata, such as the Life-cycle state, which can change any minute now, so it will inevitably be necessary to still call up the external Oracle API to request this information in real time, and then aggregate it with the remaining metadata.

In order to make the system fault tolerant and to guarantee availability of service, each method features two different version, one with DB interaction and one without. In fact, the DB could go down or become unreachable for any reason, so the idea is to check periodically if the connection with the DB is still alive and, accordingly, configure a global Boolean variable to indicate either if the DB is up or down. Based on the value of this variable, it is possible to switch between the two version of the method and guarantee, also in presence of a connection problem, that the service is still provided, even with a consequent increase of the response time. The connection with MongoDB is established by means of the MongoClient interface, which must be provided with the following information: username, password, IP address, port and database name.

Scheduling

Spring Boot Scheduling is a practical feature that allows to schedule jobs in a Spring Boot applications. It is mainly used to perform some task after a fixed interval or based on some schedule and, naturally, it is a great tool for automating lots of processes, which otherwise would require human intervention.

It also works on the principle of a typical **Cron job**, which is simply a job scheduler on Unix-like operating systems. Cron is most suitable for scheduling repetitive tasks, to run periodically at fixed times, dates, or intervals. The "*@Scheduled*" annotation is used to trigger the scheduler for a specific time period and generally is provided with a Cron expression, which is a string consisting of six or seven sub-expressions (fields) that describe individual details of the schedule. As mentioned before, a Scheduler class was implemented in order to keep update all the DB records and to verify if the connection with the DB is still alive. Based on the amount of resources existing in the provided tenancy, it was considered reasonable to schedule the update task twice a day, at noon and midnight, whereas the connection checking task every five minutes.

4.3 Front-end

Angular is an open source web application development framework powered by Google, which provides support for developing dynamic and single page web applications. It allows to use HTML as a template language and to extend its syntax to express the components of an application in a clear and succinct way. Since the framework is entirely scripted in Typescript, it is necessary to compile all the developed code, to transform it into a language understandable by current browsers.

Therefore, the compilation is done in plain JavaScript framework and is required the use of Node.js and Node Package Manager (NPM) to accumulate them into JavaScript files, so that deployment can be done in the process. Node.JS is a cross-platform run-time environment for running JavaScript applications outside the browser and it offers a rich library of various JavaScript modules that can simplify coding.

4.3.1 Angular framework

The architecture of an Angular application relies on certain fundamental concepts, the most basic UI building blocks are undoubtedly the **Angular Components** [42]. Typically, an Angular app structure is based on a tree of components, where each component consists of:

- An HTML template that declares what renders on the page;
- A Typescript class that defines its behavior;
- A CSS selector that defines how the component is used in a template;
- An optional CSS styles applied to the template.

Directives are classes that add additional behavior to elements in an Angular applications. Angular components are a subset of directives, always associated with a template and, unlike other directives, can be instantiated for a given element in a template. Angular applications can be written by creating HTML templates along with mark-up, a component class that administers these templates, in addition the application logic is incorporated in services, both components and services are boxed into modules.

A template looks like regular HTML file and can use a mechanism, called data binding, for coordinating the parts of a template with the parts of a component. Data binding plays an important role in communication between a template and its component, and is also important for communication between parent and child components. Angular supports two-way data binding, a mechanism for coordinating the parts of a template with the parts of a component, by adding a binding markup to the template HTML to tell Angular how to connect both sides. Property binding allows to define properties in a component class, and communicate these properties to the template, also setting the properties and attributes of various HTML elements. Event binding, instead, allows to define events that occur in the template (user-initiated events) and communicate to the component class.

A component must belong to an **NgModule** in order to be available to another component or application. NgModules are containers for a cohesive block of code that can be dedicated either to an application domain, a workflow, or a closely related set of capabilities. Modules are a great way to organize an application and extend it with capabilities from external libraries. Hence, many third-party libraries are available as NgModules (e.g., Material Design, Ionic and Syncfusion JavaScript). Angular internal libraries are NgModules, such as FormsModule, HttpClientModule, and RouterModule. NgModules consolidate components, directives, and pipes into cohesive blocks of functionality, each focused on a feature area, application business domain, workflow, or common collection of utilities.

Every application has at least one Angular module, the root module (conventionally named AppModule), which must be present for bootstrapping the application on launch. Then, the framework takes over and presents the app content in the browser and it responds to the user interaction in accordance with the instructions that are provided.

Components normally should only focus on presenting data, so they utilize services which supply particular functionality and are not directly connected to views. Services are singleton objects that get instantiated only once during the lifetime of an application and they contain methods that maintain data throughout the life of an application. In order to provide the metadata, that allows Angular to inject a service into a component as a dependency, the Injector design pattern is needed. This in order to requests dependencies from external sources, rather than creating them inside the same module. The main objective of a service is to organize and share business logic, models, or data and functions with different components of an Angular application. Services generally have functions to make API call, these functions can be used by any component in order to interact with the back-end. Also, components have no longer to perform the task of fetching the data, as services take care of this, thus achieving the objective of Separation of Concerns.

Figure 4.3 shows the diagram of Angular architecture that describes the main building blocks of the application, that have been presented. Here, services and components are merely classes, at times with decorators that mark its type and give metadata that inform the framework how to employ them.

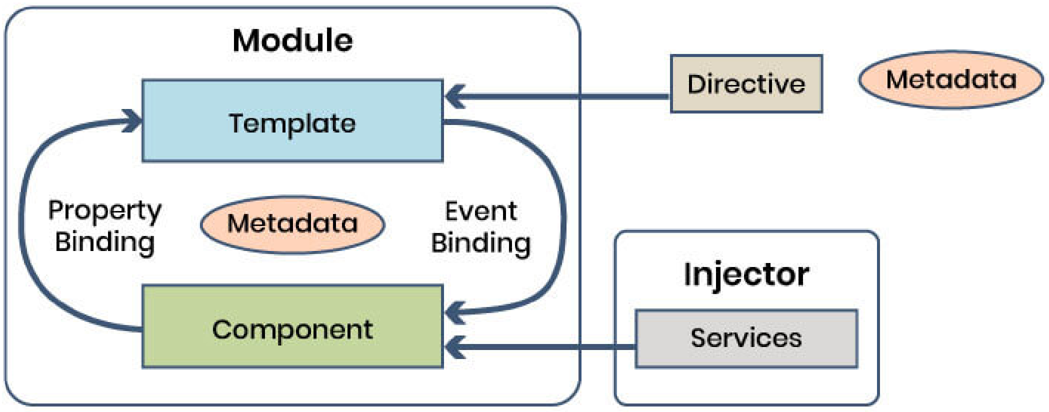


Figure 4.3: Angular application architecture

4.3.2 GUI features

In order to implement the functionalities needed in the GUI, the first step was to generate a starting component associated with a basic template, which immediately defines that component's host view. The application can take action at each moment in the component life-cycle, that starts when Angular instantiates the component class and renders the component view along with its child views. The life-cycle continues with change detection, through optional life-cycle hooks, like "*ngOnInit()*". This hook is called immediately after the component creation, to execute custom initialization logic after the directive's data-bound properties have been initialized.

The idea is to render a Dropdown menu (a GUI component that lets users choose from a list of item), where the whole list of compartment belonging to the "tecitgen2" tenancy is provided. Therefore, inside the "*ngOnInit()*" life-cycle hook, the "*getCompartments()*" service method, responsible of making an HTTP request to the specif back-end API endpoint, is called.

Services and HTTP Client

There is only one and unique service, visible throughout the application and in charge of making all the HTTP API calls to the endpoints exposed at back-end, that enable the API to access resources. This can happen by means of the **HTTP CLient Module**, which is an Angular's mechanism for communicating with a remote server over HTTP, in order to download or upload data and access other back-end services. The service method invokes the "*get()*" method to fetch data from a server. The asynchronous method takes the endpoint URL from which to fetch and returns to the component, that have called the method, an Observable object that emits the requested data when the response is received. The return type varies based on the observe and response type values that is passed to the call. By default, it returns the body of the response as an untyped JSON object, however, it is possible to manually instantiate a class object for the type of response that is returned, and it automatically parses the JSON server response into the object type supplied.

If the HTTP request will be successful, then the object will emit only one value and complete, otherwise it will emit an error. The component, at this point, stores the requested data in a local variable of the same type and bind it to the HTML view for being displayed. In this case, the returned object is a list of *Compartment* classes with the same attributes of the class defined in the back-end (subsection 4.2.1). The "*name*" attributed will be mapped in the text property of the Dropdown menu to be displayed in the list, the "*id*" will be mapped in the value property, whereas the "*compartmentName*" attribute will be mapped in the groupby property in order to logically group together all the compartment with the same parent, so at the same level of the hierarchy.

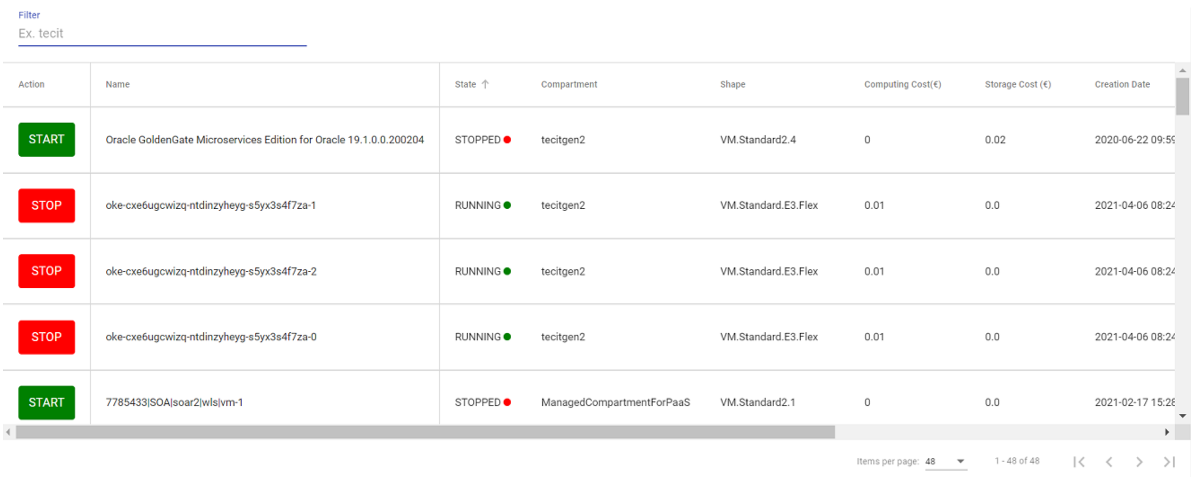
Once the entire list is displayed in the menu, the user can now select the desired compartment and click the button in order to delegate the service to start another API call. This time, the HTTP Client will perform "*post()*", passing the selected compartment id (which is the *value* property of the menu item) along with the ids of all the child compartment. So an HTTP POST request is sent to the corresponding back-end endpoint in order to retrieve all the metadata concerning the requested resources. As observed before, the data will be returned and mapped in another class object but this time the local variable will be passed as input to a child component, in charge of render a table where to present all the resource metadata.

In fact, a component can also define a view hierarchy, which contains embedded views, hosted by other components. The child component has its own business logic and design, that can act as a small unit of functionality for the whole component. Every child component associated with a parent component is called nested component. Moreover, one use of nested components is to send data from the child component to the parent, so that an action can be triggered by the parent based on the instruction provided by the child component. The nested component are recognizable through selector tag, that can be inserted as normal tag in an HTML template, and by using such a selector it is possible to render the view part of that child component into the parent component. Once the data are available, the child view is rendered and the table is finally displayed.

Material Table and Histogram Chart

Angular Material data tables provide a quick and efficient way to create tables of data with common features like pagination, filtering and ordering. Moreover, the cell templates are not restricted to only showing simple string values, but are flexible and allows to provide any template, like in this case a button to trigger an API call. Figure 4.4 shows a screenshot of the material table implemented in the GUI, in order to display metadata related to all compute instances within the tenancy. This snapshot clearly pints out some of the features that are available, such as the button to either stop or start a compute instance, the sorting behavior and styling to a set of table headers with ascending order first and then descending, the input field provided with a custom function to filter data and the paginator in the footer section of the table.

The simplest way to provide data to a table is by passing a data array but a more flexible approach can be used by by encapsulating the data source logic into a DataSource class. It is necessary to get data from the back-end via service and push it to the DataSource array, which is meant to serve as a place to encapsulate any sorting, filtering, pagination, and data retrieval logic specific to the application. This approach is very useful, because it allows to dynamically change the table content based on the interaction between the three features implemented. For instance to sort data related only to a page or to a portion of filtered data, instead of sorting the entire list. This can be done by simply overriding the default sorting behavior, thus implementing a custom sorting method and applying the sorting strategy to the correct portion of data. For the purpose of keeping data updated, if the user does not perform another research, a timeout is set in the parent component to periodically refresh the table content by making another request to the same service method. Whereas, the child component by means of a special class, called EventEmitter, is in charge of emitting an event, so that it can communicate to the parent component to refresh the table when the button is pressed.


Filter
Ex. tecit

Action	Name	State ↑	Compartment	Shape	Computing Cost(€)	Storage Cost (€)	Creation Date
START	Oracle GoldenGate Microservices Edition for Oracle 19.1.0.0.200204	STOPPED ●	tecitgen2	VM.Standard2.4	0	0.02	2020-06-22 09:5
STOP	oke-cxe6ugcwizq-ntdinzyheyg-s5yx3s4f7za-1	RUNNING ●	tecitgen2	VM.Standard.E3.Flex	0.01	0.0	2021-04-06 08:2
STOP	oke-cxe6ugcwizq-ntdinzyheyg-s5yx3s4f7za-2	RUNNING ●	tecitgen2	VM.Standard.E3.Flex	0.01	0.0	2021-04-06 08:2
STOP	oke-cxe6ugcwizq-ntdinzyheyg-s5yx3s4f7za-0	RUNNING ●	tecitgen2	VM.Standard.E3.Flex	0.01	0.0	2021-04-06 08:2
START	7785433|SOA|soar2|wls|vm-1	STOPPED ●	ManagedCompartmentForPaaS	VM.Standard2.1	0	0.0	2021-02-17 15:2

Items per page: 48 1 - 48 of 48

Figure 4.4: Material Table view

Another important feature which this GUI provides, is that, in order to promptly and clearly supply an overview of the cost produced by each compartment regarding a specific resource type, an Angular Histogram Chart is therefore displayed. This module is a bar column chart used for frequency distribution, where the widths of bars are proportional to classes into which variables have been divided (the compartment to which the resources belong) and the heights of the bars are proportional to class frequencies (the total cost produced by each resource). Figure 4.5 is a screenshot taken from the GUI, showing the Histogram Chart related to the cost (EUR) produced by each compartment in "tecitgen2" tenancy. This type of chart is a good choice when the data is larger than could be plotted on a bar chart, and can be used to visual display large amounts that are difficult to understand in a tabular or spreadsheet form. In addition, a series of label (one for each compartment) is provided below the chart, with the purpose of letting the user to choose the desired compartments to be analysed in the chart.

Route

Router is the last building block from the architecture overview (Figure 4.5), which remains to be mentioned. It is an Angular module that allows to create applications

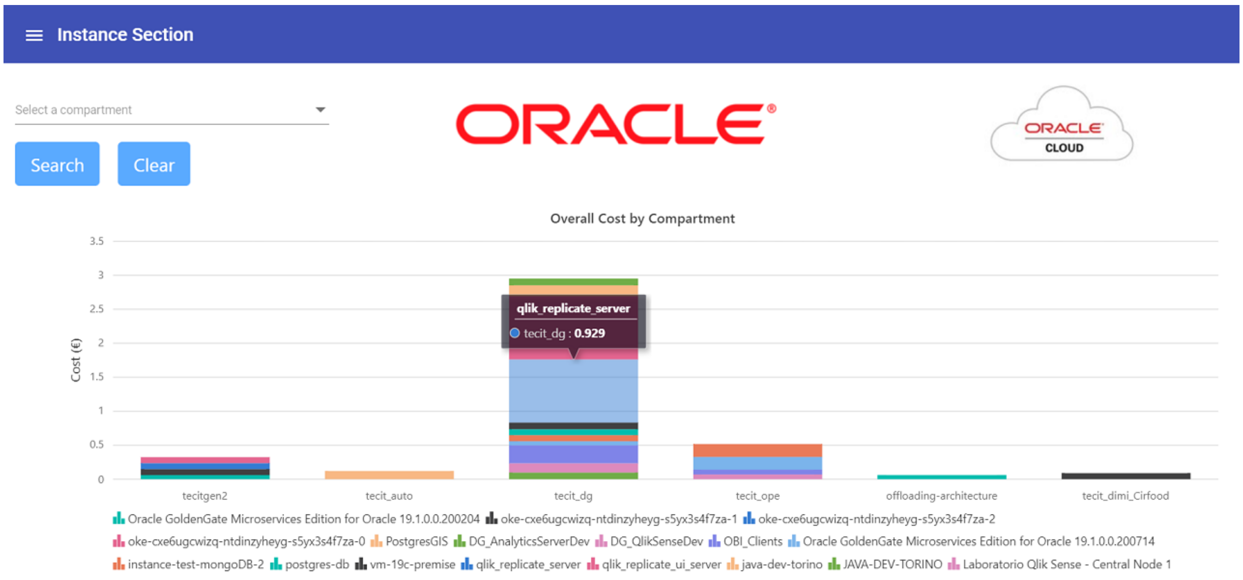


Figure 4.5: Histogram Chart view

with multiple views and also enables the user to navigate between the different sections without having to reload the pages through the browser. In order to build a Single Page Applications with multiple views and allow navigation between these views, it is of primary concern to define the different paths of the application inside the "*app.module.ts*" file, where is created an array of JavaScript objects that contains two properties. Each object has a "*path*" property, which is the URL path for the route, and a "*component*" property, defining the component which Angular should render for the corresponding path.

The Router, having to decide which component to show at a certain path, starts scrolling the array from the first object and stops as soon as it finds one with a "*path*" property that matches the path. The "*forRoot*" static method is the method that configures the root routing module for the application, asking Angular to provide an instance of the Router class globally. Finally, it is requested to update the component template to include "*<router-outlet>*". This element informs Angular to update the application view with the component for the selected route.

4.4 Authentication

The information that the GUI presents and manages are confidential and restricted only to the users who have subscribed to Oracle. In order to prevent unauthorized users from accessing such information, an authentication stage was integrated to the system according to the IAM policy described in subsection 1.2.2. Organizations need to securely manage access and entitlements across a wide range of cloud and on-premises, in view of this, Oracle provides a cloud-native Identity-as-a-Service (*IDaaS*) platform that addresses these needs. **Oracle Identity Cloud Service** (*IDCS*) is a cloud-native service, managing user access and entitlements across a wide range of cloud applications and services with flexible authentication options. It supports the three-legged authentication flow for several SDKs, with the purpose of allowing the users to directly interact with IDCS. The user can login with his credential and provide his consent to allow application to access his data, so all authorization and authentication stuff is handled through IDCS. Figure 4.6 illustrates a data flow diagrams, describing the calls and responses between the web browser, the web application, and Oracle Identity Cloud Service for each use case.

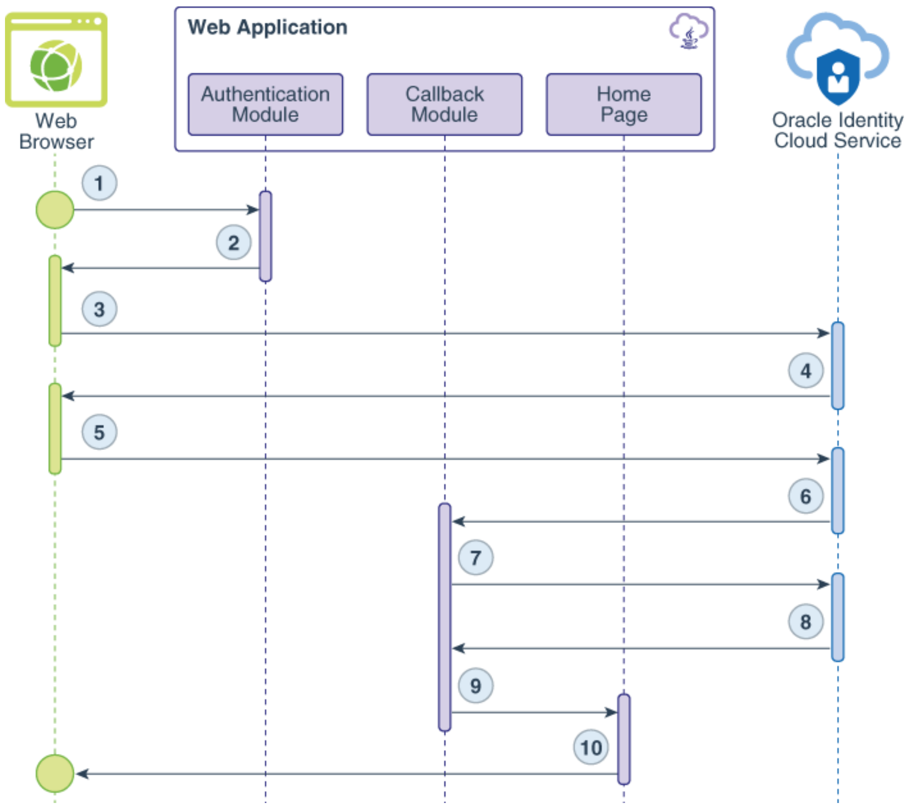
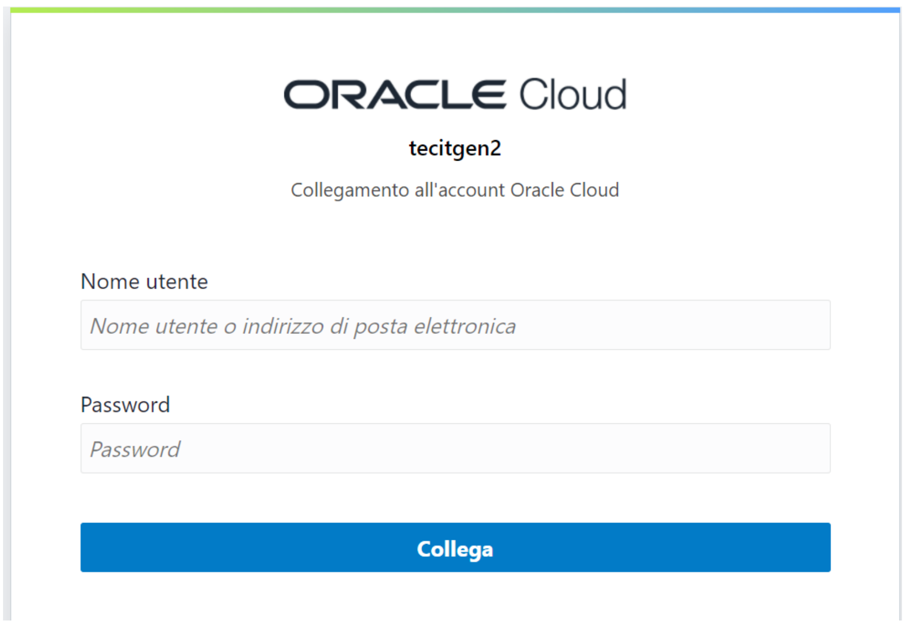


Figure 4.6: Three Legged Authentication with Oracle Identity Cloud Service

The workflow is the following; the user requests a protected resource, so the authentication module uses the SDK to generate a request-authorization code URL for IDCS, and send this URL as a redirect response to the web browser. At this point, the browser calls the URL and the IDCS Sign In page (depicted in Figure 4.7) appears. The user has now to submit its credentials to sign in, and consequently IDCS creates a private session issuing an authorization code. Once the code is arrived, the web application can make a back-end call to exchange it for the access token. IDCS issues the access token and finally establishes the session, so the user can be redirected to the Home page of the web application.

In the proposal solution, the web application was registered with Oracle Identity Cloud Service to establish communication through an ID and a Secret provided. In addition a Node.js SDK was used to be integrated with Angular framework. This strategy is based on the passport framework, which is requested to forward the access token as a header, and use the "*passport.authenticate()*" method to create a user session, passing the user attributes in the request as a json object. Moreover, several information are needed to be provided to IDCS:

- Client ID, the value generated after the application have been registered in the Identity Cloud Service console;
- Client Secret, generated in the same way;
- ClientTenant, the domain prefix of the IDCS instance under consideration;
- IDCSHost, the domain suffix of the IDCS instance under consideration;
- Redirect URL, to redirectthe web browes after user signs in;
- The OAuth scope of authentication;
- State, used by OAuth protocol to check whether communication can be established or not.


ORACLE Cloud

tecitgen2

Collegamento all'account Oracle Cloud

Nome utente

Nome utente o indirizzo di posta elettronica

Password

Password

Collega

Figure 4.7: Oracle Identity Cloud Service Sign In web page

These parameters are used for the authorization code URL, that is generated by the SDK. The web browser makes a request to */oauth/oracle* URL route and the SDK, as consequence, triggers a function designed to build the URL. Furthermore, a promise is used to either redirect the user's web browser, when the authorization code URL is generated, or to render an error. At this point, the SDK uses the authorization code to request an access token, which is stored as a cookie to be later sent to the web browser for future use.

As mentioned before, the "*passport.authenticate()*" method creates the application's session and the user's web browser is redirected to a protected URL, which is the web application GUI Home page. Finally, when the user wants to sign out from single sign-on between the application and Oracle Identity Cloud Service, it has simply to call the */logout* route, which is implemented by the Node.js SDK. This route invalidates the user session, removing any cookies set previously, and then redirects the user's web browser to a log out URL, set in the configuration file.

.

Chapter 5

Cloud deployment

In this next chapter, it will be illustrated how the whole system was deployed on OKE. It will follow a brief description of the Docker files used to create the static images and the Kubernetes deployment files, necessary to manage the pods and the services creation. Finally, the results of a stress test will be presented in order to monitor the behavior of the system and analyse its performance.

5.1 Docker images

The goal now is to move the whole system, with both front-end and back-end that working flawlessly on the local computer, to a cloud environment. To accomplish this, it is required the use of Docker in order to build the "pictures" of the system that will then be uploaded to OKE container registry. As briefly said in subsection 3.2.1, the picture or image is assembled utilizing a Dockerfile, which is located in the same directory where the source code resides and characterizes how the image ought to be constructed.

Listing 5.1 shows the content of the file used to build the back-end image, so to "Dockerize" a Springboot application. This file contains all the commands executed by the Docker daemon, once the docker build command is issued. The first instruction is simply meant to build the image upon java 8 image from Docker Hub, the source code and all the necessary files are copied, then it is added a VOLUME pointing to "/tmp" because that is where a Spring Boot application creates working directories for Tomcat by default. The effect is to create a temporary file on the host under "/var/lib/docker" and link it to the container under "/tmp". The third instruction provides to the container the folder with the configuration file and the private key used in order to perform a call to an external Oracle API (as explained in section 4.2.2). Then, it copies the the project JAR file, built with Maven, into the container as "app.jar", which is run in the ENTRYPOINT. Finally, after having exposed the port 8080, the ENTRYPOINT instruction (the executable to start when the container is booting) is called. It runs Java and sets the Spring Mongo property, in order to reach the DB within the container, along with quick additional property to speed up the Tomcat startup time, and then points it at the "app.jar" defined before.

Dockerfile back-end

```
FROM java:8
VOLUME /tmp
ADD /src/main/resources/.oci /.oci
ADD target/ *.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-Dspring.data.mongodb.uri=mongodb
            ://TechnologyIT:******130.61.38.219:27017/fca_test",
            "-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
```

The following step is to assemble the image also for the front-end. Listing 5.1 shows the instructions from the second Dockerfile. This time, a multi-stage build is adopted, using multiple FROM statements. Each instruction allows to use a different base, and each of them begins a new stage of the build. In addition, it is possible to selectively copy artifacts from one stage to another, leaving behind everything it is not necessary in the final image.

In the first stage, it is initialized a new build stage and set the Base Image for subsequent instructions, the chosen image define is based on the popular Alpine Linux project, which is much smaller than most distribution base images (5MB), and thus leads to much slimmer images in general. Then, a directory is created to hold the application code inside the image, this will be the working directory. This image comes with Node.js and NPM already installed so the next thing to do is to install all the app dependencies using the npm binary. It is important to point out that, rather than copying the entire working directory, only the "package.json" file is copied. This allows to take advantage of cached Docker layers, in this way it is not necessary to rebuild the modules each time the container is rebuilt. If the "package.json" file changes then the modules will be rebuilt, otherwise Docker will use its cache and skip that part. Inside the app folder, the dependencies are installed by running the "npm install" command and copying the other contents of the project folder to the app folder. The app's source code is then bundled inside the Docker image and the generated build artifacts are copied to the "app/dist" folder.

In the second stage, the Angular application has to be deployed in an NGINX web server. Therefore, the next instruction sets the latest nginx image as the base image for executing subsequent instructions relevant to nginx configuration. An nginx configuration file must be created in the project directory and copied in the folder specified in the first COPY instruction. Then, all the build output generated in the first stage is copied to replace the default nginx contents. In fact, by default, nginx looks in the "/usr/share/nginx/html" directory inside the container for files to serve, so it is necessary that the files reside into this directory. Finally, with the CMD command the nginx container is started in background and listens on network port 80 at runtime.

```
Dockerfile front-end
```

```
### STAGE 1:  Build ###
FROM node:12.7-alpine AS build
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN $(npm bin)/ng build --prod --output-path=app/dist

### STAGE 2:  Run ###
FROM nginx:1.17.1-alpine
COPY nginx.conf /etc/nginx/conf.d/default.conf
COPY --from=build /usr/src/app/dist/oracle-web /usr/share/nginx/html
CMD ["nginx", "-g", "daemon off;"]

EXPOSE 80
```

Listing 5.1 shows the nginx configuration file used to create a web server that listen on port 80. The server declaration indicates which server block is used for a given request and defines a specific virtual host. The root directive indicates the actual path on the hard drive where this virtual host's assets (e.g., HTML, images, CSS, etc.) are located (the default location is "/usr/share/nginx/html"). The index setting tells nginx what file or files to serve when it is asked to display a directory. If none of the files listed are found, nginx will either reply with a listing of all the files in that directory or with an error. That will make it return the index.html for any other URL it receives, so, all the Angular router URLs will work, even when going to the URL directly in the browser.

In the second location section, nginx proxies a request, sending the request to a specified proxied server. If the "proxy_pass" directive is specified with a URI, then when a request is passed to the server, the part of a normalized request URI matching the location is replaced by the URI specified in the directive. This results in passing all requests processed in this location to the proxied server at the specified address. This address can be specified as a domain name or an IP address and it may also include a port. In this case, all the traffic targeted to */apis* will be redirected to "backend-service", which is the internal DNS name for the Kubernetes Service in charge of serving the back-end pod exposing the REST API endpoints and listening at port 8081.

Nginx.conf

```
server {
    listen 80;
    server_name frontend;

    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
        try_files $uri $uri/ /index.html =404;
    }

    location /apis/ {
        proxy_pass http://backend-service:8081;
    }
}
```

In order to deploy the system to OKE, the images must be previously pushed to the OCI Registry, that enables to store, share, and manage development artifacts like Docker images. If the registry has been already created in a tenancy, then it is possible to login from the Docker CLI, by entering an authentication token generated by the registry owner. Once authenticated, Docker must give a tag to the image that is going to be pushed to the registry with the following format:

```
<region-key>.ocir.io/<tenancy-namespace>/<repo-name>:<tag>
```

"<region-key>" is the key of region where the registry resides on, "ocir.io" is the OCI registry name, and "<tenancy-namespace>" is the auto-generated Object Storage namespace (see section 1.2.5) string of the tenancy to which is desired to push the image. Finally, "<repo-name>" is the name of the target repository to which push the image and "<tag>" is an image tag (e.g, latest). The last step is pushing the Docker image from the client machine to Oracle Cloud Infrastructure Registry, by means of "docker push" instruction followed by the tag chosen by the user.

5.2 Pods Deployment

Once the images are available on OCI Registry, it is now possible to reference them explicitly in a Pod. For making easy the Pods deployment, it is necessary to create a Kubernetes **Deployment** objects, which are resources that provide declarative updates to applications.

A deployment describes the life cycle of the application, for example specifying the images to be used, the number of pods needed, and how to update them. Moreover, it automatically replaces any instances that fail or become unresponsive, ensuring that at least one instances of the application is available to serve the user requests. Listing 5.2 shows the YAML deployment file used to create and run the pod replicas responsible of listing the compute instances.

service-deployment.yaml

```
spec:
  replicas: 1
  selector  :
    matchLabels:
      app : service-deployment
  strategy : {}
  template :
    metadata :
      labels :
        app : service-deployment
    spec :
      containers :
      - image : eu-frankfurt-1.ocir.io/fr99gki0bywn/
              docker-oracle-backend/oracle-service:latest
        name : service
        resources :
          requests :
            cpu : "250m"
            memory : "250Mi"
          limits :
           cpu : "250m"
           memory : "250Mi"
        ports :
        - containerPort : 8091
---
apiVersion : v1
kind : Service
metadata :
  name : instance-service
  labels :
    app : service-deployment
spec :
  ports :
  - port : 8090
    protocol : TCP
    targetPort : 8091
  selector :
    app : service-deployment
```

The deployment file contains the metadata with a "spec" section to define the replicas and configurations related to the deployment. It also specifies the "matchLabels" selector, which is a required field that specifies a label selector for the pods targeted by this deployment, and tells the resource to match the pod according to that label. The template section defines the container image, in this case is provided the Springboot image built with Listing 5.1 and pushed to OCI Registry, and the container port as 8091. In addition, for each resource type in a container, it is possible to specify a "requests" and a "limits" value. Requests generally is used for scheduling and is the minimum amount of resources a container needs to run. Whereas, the limits is the maximum amount of this resource that the node will allow the containers to use. Limits and requests for cpu resources are measured in cpu units. One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers and 1 hyperthread on bare-metal Intel processors. The memory is measured in bytes and can be expressed as a plain integer or as a fixed-point number using one of these suffixes: E, P, T, G, M, k (or the power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki).

Finally, it is specified the service associated with the deployment. The service routes traffic to pods which match the label selectors, and identify those pods as member pods. It is possible to specify the type of service, ClusterIP is the default value. This kind of service exposes an internal IP, which is accessible from any of the Kubernetes cluster's nodes but not routable outside of the cluster. In addition, the service must define one or more ports, in order to listen on with target ports to forward TCP/UDP traffic to containers. The use of virtual IP addresses for this purpose makes it possible to have several pods exposing the same port on the same node.

In the proposal solution, the idea is to create a pod running the nginx front-end microservice, described in Listing 5.1. In order to enable the communication with the back-end pods, the front-end microservice is configured to send traffic to the ClusterIP service, which exposes the orchestrator service pod's API endpoints. In fact, the pod in the front-end deployment run a nginx image that is configured to proxy requests to the orchestrator back-end service (as seen in nginx configuration file Listing 5.1), by using the DNS name. The orchestrator service, in turn, communicates with the specific ClusterIP service managing one of the back-end microservice, running in multiple instances.

An important difference to notice between the back-end and front-end services, is that the configuration for the front-end service has type "LoadBalancer", so a load balancer service object is created to expose the front-end microservice outside the cluster. The load balancer has a stable IP address that is accessible from outside and forwards the packets with no change to the source and destination IP addresses. While the actual pods that compose the back-end set may change, the end clients should not need to be aware of that, nor should they need to keep track of the set of back-ends themselves.

Autoscaling

The solution adopted needs to exploit Kubernetes full potential, so it is necessary to automate the scaling of the microservices running in the cluster. In Kubernetes the autoscaling is possible thanks to the **Horizontal Pod Autoscaler** (*HPA*). This resource has to periodically query the resource metrics on the pods to understand when to increase or decrease the number of pod instances between a minimum and a maximum. When these metrics are available, the HPA verifies if the threshold defined has been exceed or not, and scales up or down accordingly. In the proposed solution, an HPA is associated with each microservice and the cpu utilization is monitored to understand when the number of instance for the specific microservice must be increased or decreased maintaining an average cpu utilization across all Pods of 25%.

The minimum number of replicas is set to one in the debugging phase but it would be more efficient to start with at least three instances, while the maximum number is set to one hundred. If the cpu usage goes beyond 25%, a new replica will be created up to the maximum, instead if the percentage goes below the target the exceeding pods are immediately killed by the HPA. In this way is always guaranteed that at least the minimum number of pods is always running, avoiding to waste the resource footprint of individual pods and resulting in significant cost savings.

QoS

It is possible to make sure that a pod has a fixed or minimal amount of node resources by specifying the container configuration. **Quality of Service** (*QoS*) class is a Kubernetes concept that the scheduler uses for deciding the scheduling and eviction priority of the pods. The configuration provide different QoS classes for pods running in Kubernetes:

- Guaranteed: Pods are considered top-priority and are guaranteed to not be killed until they exceed their limits;
- Burstable: Pods have some form of minimal resource guarantee, but can use more resources when available;
- Best Effort: Pods will be treated as lowest priority and they are the first to get killed if the system runs out of memory.

Cluster pod allocation is based on requests (cpu and memory). If a pod claims a request larger than available cpu or memory in a node, the pod cannot be run on that node. Moreover, if none of the cluster nodes have enough resources to run the pod, it will remain pending of schedule until there are enough resources. For a pod to be placed in the Guaranteed QoS class, every container in the pod must have a cpu and memory limit. Kubernetes will automatically assign cpu and memory request values, equal to the cpu and memory limit values, to the containers inside this pod and will assign it the Guaranteed QoS class. Pods with explicit and equal values for both cpu requests and limits and memory requests and limits are consequently placed in the Guaranteed QoS class.

The Kubernetes scheduler assigns Guaranteed pods only to nodes which have enough resources to fulfil their cpu and memory requests. The Scheduler does this by ensuring that the sum of both memory and cpu requests for all containers (running and newly scheduled) is lower than the total capacity of the node.

For what concerning Burstable Pods, it is a different matter; in fact, the Kubernetes scheduler will not be able to ensure that Burstable pods are placed onto nodes that have enough resources for them. This class is used when container has more memory or cpu limit than request value, so when pod requires a range of cpu or memory usage and not necessarily a specific value. If there is no BestEffort class pod, these pods are killed before Guaranteed class pods when they reached their limit.

Pod is labeled as BestEffort when it has no memory or cpu request or limit definition, and because of that, these pods can only get memory or cpu that node has. They are, however, able to use any amount of free CPU and memory resources on the node but processes in these pods are the first to get killed if the system runs out of memory. Figure 5.1 perfectly summarizes the characteristic, as well as the Kubernetes scheduling policy, related to QoS classes.

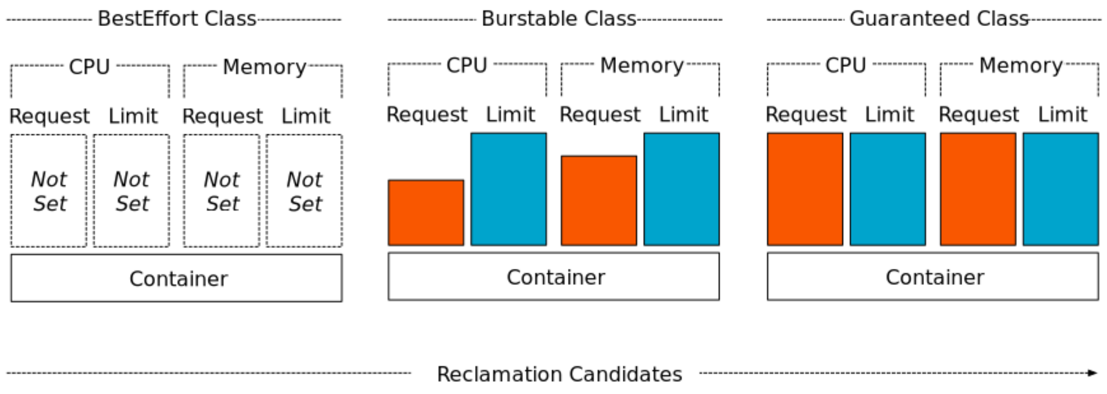


Figure 5.1: Kubernetes QoS Classes

In an over-committed environment, it is possible that the pods on the node will attempt to use more compute resource than is available at any given point in time. When this occurs, the node must give priority to one pod over another. The QoS class does affect the order in which pods are chosen for eviction by the kubelet. Kubelet first evicts BestEffort and Burstable pods using resources above requests, while Guaranteed pods are supposed to be safe in case of eviction. The idea here, is to protect the critical pods, so the orchestrator service and all the microservice instances listing the resource's metadata within the tenancy. This can be done by carefully setting reasonable values, so that they can be classified as Guaranteed. These pods must have the priority over the other but since they are managed by the HPA previously described, they will be automatically evicted once the request is over. This will help to adjust cluster capacity and improve pod evicted issues.

5.3 System monitoring

The next step in this dissertation is to monitor the performance of the system, especially the latency and the response time of individual calls, as well as the overall CPU and RAM utilization level in the cluster.

Black box monitoring means treating the application as a black box, by sending it various inputs and observing it from the outside to analyse the response. All of this does not require necessarily instrumenting the code and can be implemented from outside the application. This kind of monitoring can give a simple picture of performance that can be standardized across multiple applications. When implemented in a microservice architecture, black box monitoring can give an operator a similar view of services as the services have of each other.

Kong API gateway

Kong is an orchestration microservice API Gateway, which provides a flexible abstraction layer that securely manages communication between clients and microservices through APIs [43]. Kong allows users to easily implement black box monitoring because it sits between the consumers of a service and the service itself. This allows it to collect the same black box metrics for every service it manages, providing uniformity and preventing repetition.

The Kubernetes Ingress Controller manages external access to HTTP services in a Kubernetes cluster, using the Kong API Gateway and supporting the exposure of metrics. It is useful to give visibility into how the services in the cluster are responding to the inbound traffic. In addition, Ingress routing rules must be defined to configure Kong to proxy traffic destined for the services correctly. Listing 5.3 shows the Ingress configuration YAML file used to deploy the Kong Ingress Controller in the cluster. In this case, all the inbound traffic directed to the Ingress is proxied to the orchestrator service (whose DNS and port number are specified in the "back-end" section of the configuration file).

Nowadays **Prometheus** is the "de-facto" standard for monitoring and retrieving metrics. Kong offers the possibility to integrate a new Prometheus plugin. This kind of system empowers users to easily track performance metrics for upstream APIs, exposing them in Prometheus exposition format and providing the backbone for implementation of robust monitoring and alerting. The plugin records and exposes metrics at the node level but is not associated to any service or route. It can be set as "global", with the purpose of collecting information about all incoming calls regardless of the specific microservice involved.

kong-ingress.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: kong-ingress
  annotations:
    konghq.com/strip-path: "true"
    kubernetes.io/ingress.class: kong
spec:
  rules:
  - http:
     paths:
     - path: /apis
       backend:
         serviceName: backend-service
         servicePort: 8081
```

Listing 5.3 is used to define the Prometheus plugin within the cluster. Furthermore, the Prometheus server must be configured to discover all Kong nodes via a service discovery mechanism, before consuming data from each node's configured */metrics* endpoint. To do this, the Kong-proxy deployment must be provided with an appropriate annotation in order to ask Prometheus to scrap metrics from the default port 8100.

prometheus-plugin.yaml

```yaml
kind: KongClusterPlugin
metadata:
  name: prometheus
  annotations:
    kubernetes.io/ingress.class: kong
  labels:
    global: "true"
plugin: prometheus
```

Now that the metrics are available, it is possible to supply a graphical representation by means of **Grafana**, which is an open-source visualization software, mainly used to create dashboards for the monitoring data with customizable visualizations. Grafana does not interact directly with Kong but is in charge of regularly querying Prometheus, which stores internally the data obtained by scraping the Ingress Controller for requests proxied via Kong. In order to access the Grafana dashboard, it is necessary to have a service with external IP or load balancer, since the pods are deployed on OKE.

Stress Test with Apache JMeter

The next step is to generate some traffic in order to verify the stability and reliability of the system. Therefore, it should be performed a test, mainly to determine the system on its robustness and error handling under extremely heavy load conditions. Stress testing ensures that the system would not crash under crunch situations and that it does not breakdown under heavy loads. It tests beyond the normal operating point and evaluates how the system works under those extreme conditions.

Apache JMeter is an open source testing based on Java for stress and performance testing. Thread group elements are the beginning points of any test plan. Hence, a Thread Group must be created and provided with an HTTP request containing: the url, the port and the method along with the body of the request. The most important property to set are:

- Number of Threads (representing the number of users connected to the target website);
- Loop Count (number of time to execute testing);
- Ramp-Up Period (how long to delay before starting the next user).

All controllers and samplers must be under a thread group. The Parallel Controller can be used to create parallel requests, that are not executed one after the other, but simultaneously. For this purpose, the thread number was set to eight with a ram-up period of one second, based on the number of nodes in the cluster and on the number of potential users which can use simultaneously the application in the "tecitgen2" tenancy. Finally, a Listener was added inside the thread group to show the status of the test that has taken place. One of the most important element to take into account is the response assertion, which must be inserted inside the thread group to help asserting the response of request in the software load test plan. In this case, for asserting the response, it was used the response code 200, representing the success of HTTP request.

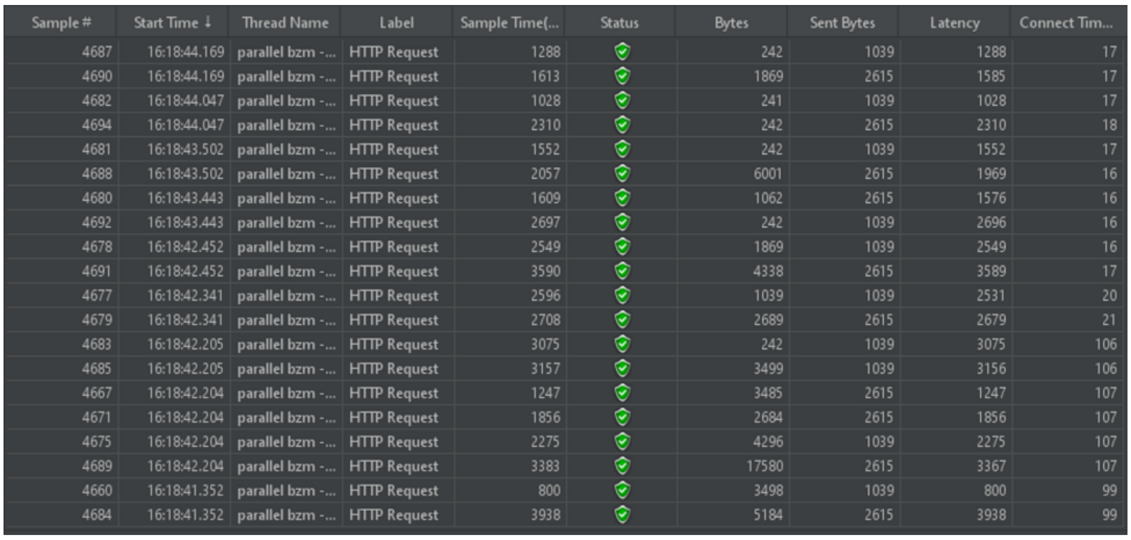
The most relevant metrics which is possible to collect through the use of Prometheus plugin are :

- Status codes: HTTP status codes returned by upstream services;
- Total number of requests per second;
- Latencies Histograms:
  - Request: Total time taken by Kong and upstream services to serve requests.

  - Kong: Time taken for Kong to route a request and run all configured plugins.
  - Upstream: Time taken by the upstream service to respond to requests.

- Bandwidth: Total Bandwidth (egress/ingress) flowing through Kong.

Initially, the worst scenario was evaluated, so when the entire hierarchy of compartments is explored in order to list all the resources of a given type. In this case, the orchestrator service has to call simultaneously a number of microservice instances equal to the actual number of compartments within the tenancy (more than twenty parallel call per HTTP request). Figure 5.2 shows a screen shot taken from Apache JMeter interface, displaying the listener table with the test results.



Sample #	Start Time ↓	Thread Name	Label	Sample Time(...	Status	Bytes	Sent Bytes	Latency	Connect Tim...
4687	16:18:44.169	parallel bzm -...	HTTP Request	1288	✓	242	1039	1288	17
4690	16:18:44.169	parallel bzm -...	HTTP Request	1613	✓	1869	2615	1585	17
4682	16:18:44.047	parallel bzm -...	HTTP Request	1028	✓	241	1039	1028	17
4694	16:18:44.047	parallel bzm -...	HTTP Request	2310	✓	242	2615	2310	18
4681	16:18:43.502	parallel bzm -...	HTTP Request	1552	✓	242	1039	1552	17
4688	16:18:43.502	parallel bzm -...	HTTP Request	2057	✓	6001	2615	1969	16
4680	16:18:43.443	parallel bzm -...	HTTP Request	1609	✓	1062	2615	1576	16
4692	16:18:43.443	parallel bzm -...	HTTP Request	2697	✓	242	1039	2696	16
4678	16:18:42.452	parallel bzm -...	HTTP Request	2549	✓	1869	1039	2549	16
4691	16:18:42.452	parallel bzm -...	HTTP Request	3590	✓	4338	2615	3589	17
4677	16:18:42.341	parallel bzm -...	HTTP Request	2596	✓	1039	1039	2531	20
4679	16:18:42.341	parallel bzm -...	HTTP Request	2708	✓	2689	2615	2679	21
4683	16:18:42.205	parallel bzm -...	HTTP Request	3075	✓	242	1039	3075	106
4685	16:18:42.205	parallel bzm -...	HTTP Request	3157	✓	3499	1039	3156	106
4667	16:18:42.204	parallel bzm -...	HTTP Request	1247	✓	3485	2615	1247	107
4671	16:18:42.204	parallel bzm -...	HTTP Request	1856	✓	2684	2615	1856	107
4675	16:18:42.204	parallel bzm -...	HTTP Request	2275	✓	4296	1039	2275	107
4689	16:18:42.204	parallel bzm -...	HTTP Request	3383	✓	17580	2615	3367	107
4660	16:18:41.352	parallel bzm -...	HTTP Request	800	✓	3498	1039	800	99
4684	16:18:41.352	parallel bzm -...	HTTP Request	3938	✓	5184	2615	3938	99

Figure 5.2: Listener Table with Test results

Figure 5.3 shows the first graph representing the total number of requests per second, which Kong handles and forwards to the orchestrator service. Another metric that Kong keeps track of is the amount of network bandwidth (kong_bandwidth) being consumed. This gives an estimate of how request/response sizes correlate with other behaviours in the infrastructure. Figure 5.4 represents the total amount of network bandwidth along with specific information like minimum, maximum, average and current value. Whereas Figure 5.5 displays separately the incoming and outgoing band used by the service. Another important metric to track is the rate of errors and requests which the service is serving. The timeseries kong_http_status collects HTTP status code metrics for each service. Figure 5.6 shows the of the status codes returned by the HTTP requests sent and, as said before, the status code 200 was inserted for asserting the success; while the default code 400 is still used to indicate that the server cannot or will not process the request.

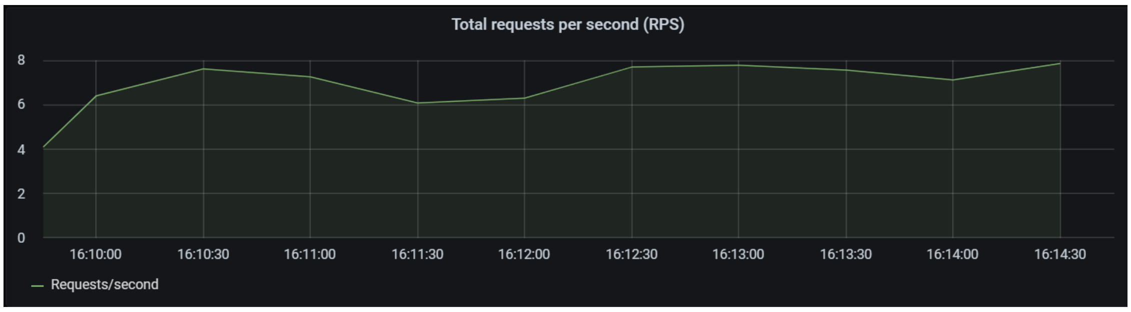
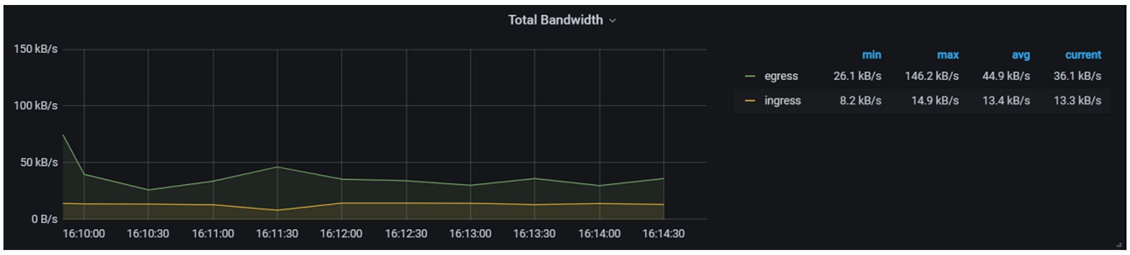


Figure 5.3: Total number of requests per second



	min	max	avg	current
egress	26.1 kB/s	146.2 kB/s	44.9 kB/s	36.1 kB/s
ingress	8.2 kB/s	14.9 kB/s	13.4 kB/s	13.3 kB/s

Figure 5.4: Total Bandwidth

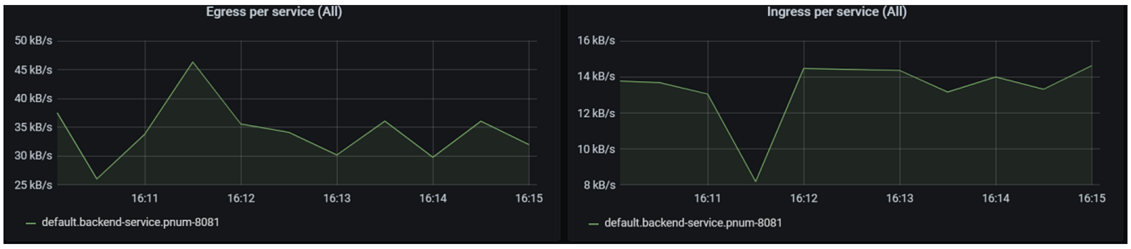


Figure 5.5: Ingress and Egress Band

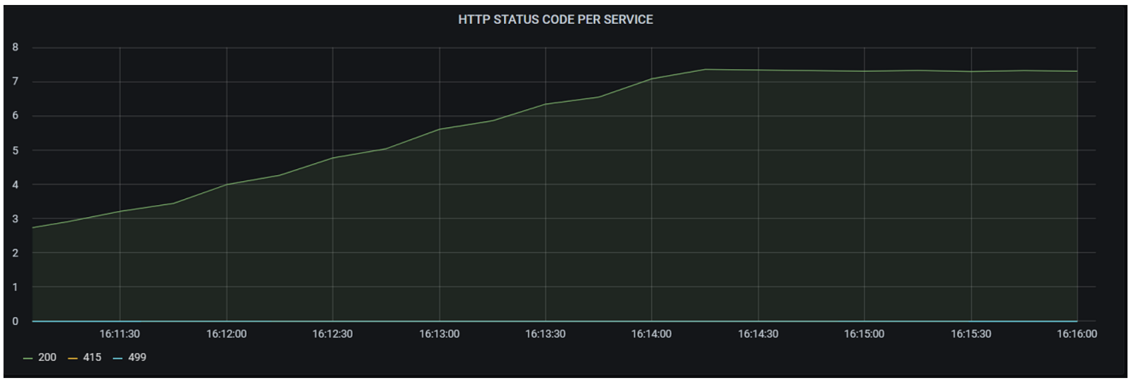


Figure 5.6: HTTP Status Code

Kong collects also latency data, measuring the time that elapses between the request and the following response. Figure 5.7 shows latency introduced by Kong, so the time in milliseconds between Kong receiving the request from the client and sending the request to the upstream service. Figure 5.8, instead, shows in the first row the total time taken by Kong and upstream services to serve requests; while in the second row is considered only the time taken by the upstream service (in this case the difference between the two is negligible). Prometheus relies on percentiles in order to supply reliable and accurate data rather than the average value or the maximum value.

"A percentile is a measure used in statistics indicating the value below which a given percentage of observations in a group of observations fall". Therefore, it is not relevant to display the percentage of requests served, but instead the 95th percentile (i.e. the maximum value of 95% of the observations). A typical way to measure percentiles from continuous monitoring data is to use histograms (also called, distributions). Prometheus queries calculate the so called ϕ-quantiles, where $0 \leq \phi \leq 1$. The ϕ-quantile is the observation value that ranks at number ϕ * N among the N observations. Making an example, the 0.95-quantile is the 95th percentile. In the below graphs, the latency is measured by estimating the quantiles (P90, P95, P99) from a set of client side observations.

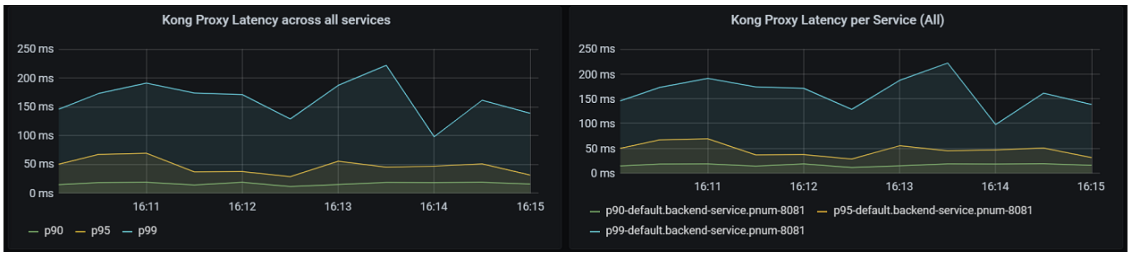


Figure 5.7: Kong Proxy Latency

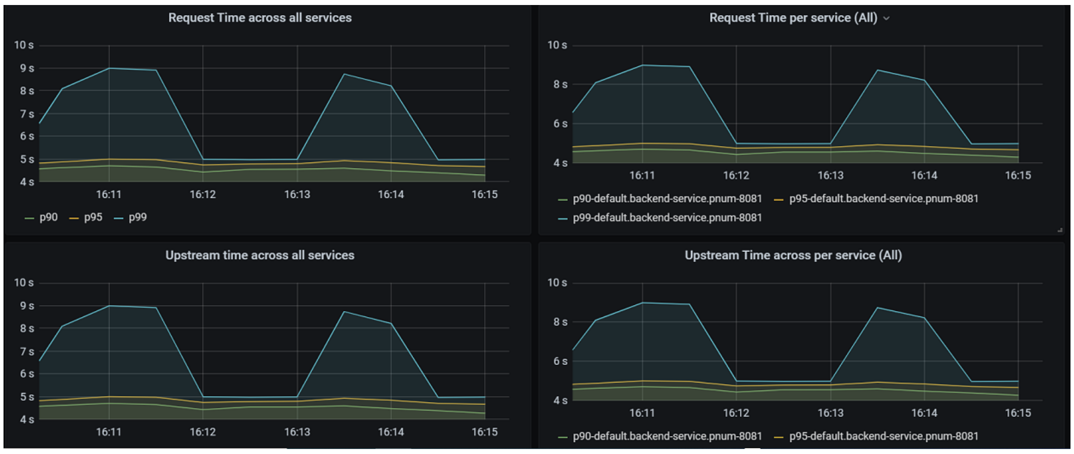


Figure 5.8: Request and Upstream Time

Figure 5.9 shows the request time considering only the *p90* and *p95* percentiles. The latency values are kept below 5 s; this can be deemed as a pretty interesting value considering that, for each HTTP request performed, the orchestrator microservice has to call roughly twenty microservice instances (in the "tecitgen2" tenancy case). Each instance, in turn, has to call the external OCI REST API (described in section 4.2.2) with a not insignificant response time. Unfortunately, it is not possible to compare the results obtained with the OCI console's ones because, as widely cited in chapter 2, the console does not offer the possibility to explore the entire hierarchy but it is limited to perform only one single API call for the specified compartment.

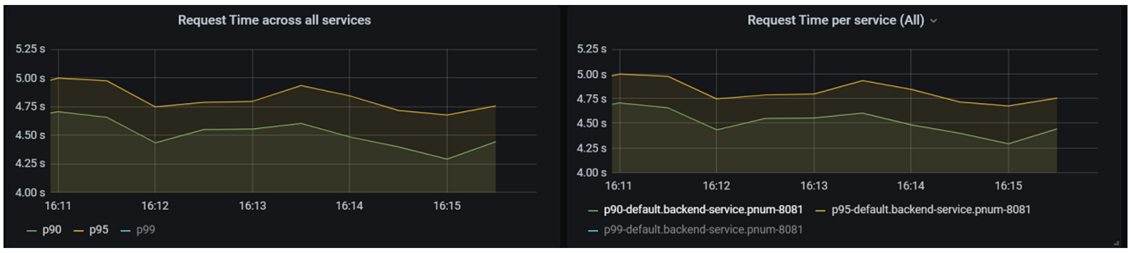


Figure 5.9: Request Time with percentile (P90,P95)

Considering instead a more realistic and common scenario, the test was moved to analyse the performance concerning only a randomly selected part of the hierarchy and not anymore the entire tenancy. Figure 5.10, Figure 5.11, Figure 5.12 respectively show the total number of requests per second, the kong proxy latency and the request time concerning this second scenario.

Here, it is interesting to observe that the total number of requests per second has grown, because the system is now able to fulfill a larger number of requests per second. Moreover, both the kong latency and the total response time decreased consequently, especially the response time which now has the *p90* percentile even below 2 s, that can be considered an excellent result also compared to the OCI console performances related to the response time of the individual OCI REST API calls (350-400 ms for a single API call).

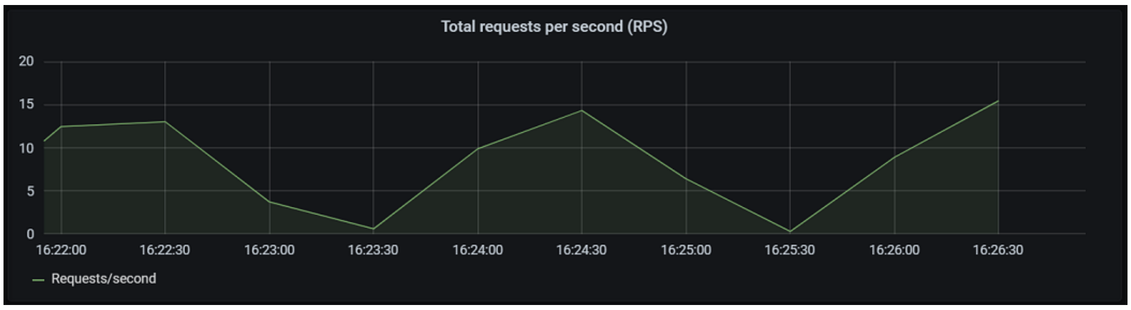


Figure 5.10: Total number of requests per second (2nd Scenario)

Another aspect which is interesting to note is the level of saturation of the cluster's nodes. Figure 5.13 depicts both the CPU Usage and Memory Usage graphs on the Kubernetes dashboard. These are the aggregated CPU and memory usage metrics for all pods belonging to the cluster, including also the nginx front-end, the authentication module, and all the remaining pods concerning Prometheus, Grafana and Kubernetes dashboard. As evidenced from the graphs, the overall CPU usage level is around 2 CPU cores (i.e. 2 VCPU), which means that each node does not use more than 1 VCPU, so it is close to the limit but no exceeds it. Whereas the memory usage is holding steady, due to the fact that most of the read/write operations involve the external MongoDB. This represents a good trade-off between the QoS experienced by the users and the amount of resources used to deploy the entire system.

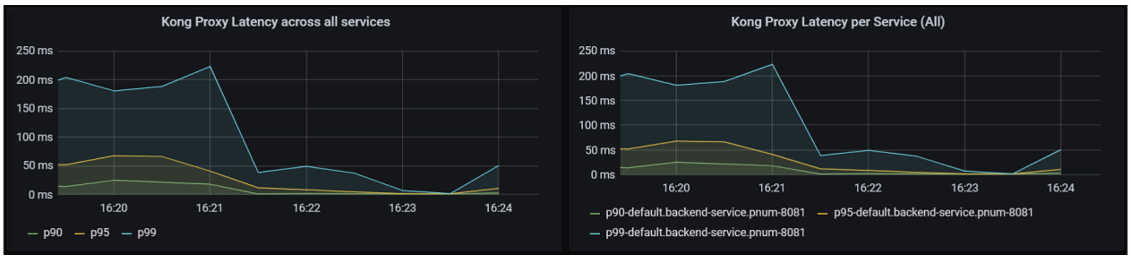


Figure 5.11: Kong Proxy Latency (2nd Scenario)

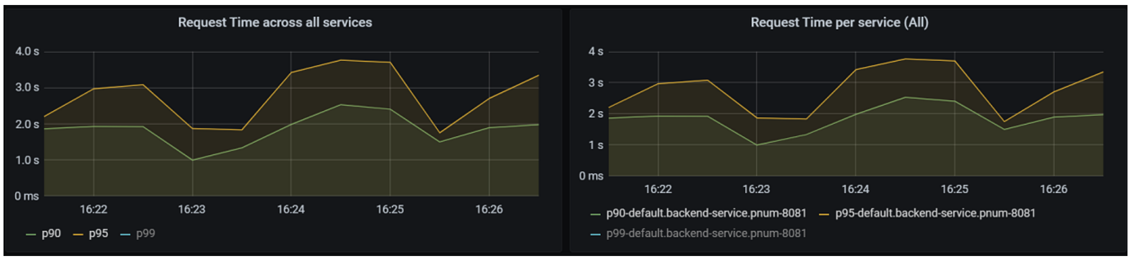


Figure 5.12: Request Time (2nd Scenario) with percentile (P90,P95)

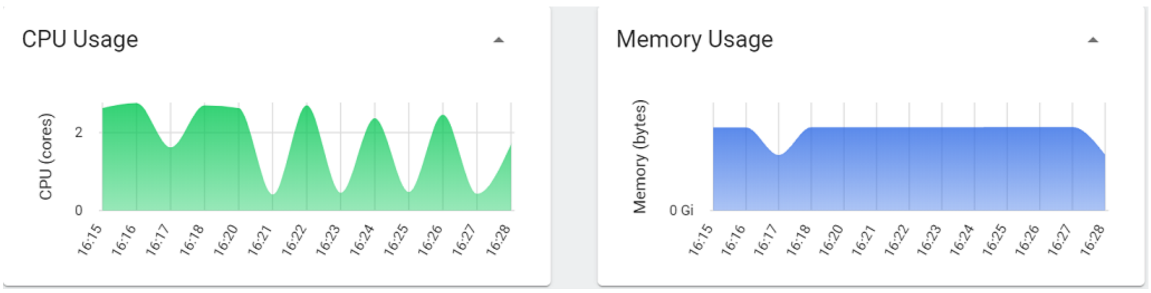
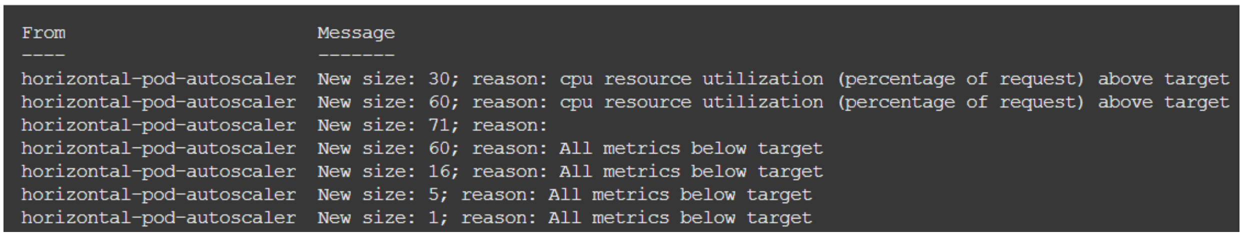


Figure 5.13: CPU and Memory usage

Finally, Figure 5.13 shows the HPA workflow during the test execution. The HPA, which continuously collects the metric regarding the CPU usage, as soon as noticed that the level has exceeded 25%, it has immediately extended the replica set by adding new instances. Since the percentage has gone beyond the 100%, the number of newly-created replicas has become very high until reaching the stability with seventy-one. Then, once fewer requests have begun to arrive, the percentage has dropped below 25% and the HPA has consequently reduced the replica's number up to only one (which was the initial defined number).



```
From                       Message
----                       -------
horizontal-pod-autoscaler  New size: 30; reason: cpu resource utilization (percentage of request) above target
horizontal-pod-autoscaler  New size: 60; reason: cpu resource utilization (percentage of request) above target
horizontal-pod-autoscaler  New size: 71; reason:
horizontal-pod-autoscaler  New size: 60; reason: All metrics below target
horizontal-pod-autoscaler  New size: 16; reason: All metrics below target
horizontal-pod-autoscaler  New size: 5; reason: All metrics below target
horizontal-pod-autoscaler  New size: 1; reason: All metrics below target
```

Figure 5.14: HPA workflow

Chapter 6

Conclusions

In this final chapter interpretations and insights are debated, conclusions are drawn, and future research directions are highlighted.

This study aimed to analyze in detail an ever expanding sector like the cloud computing with a special glance to the microservices world, deeply in touch with technologies such as Docker and Kubernetes. The solution tries to meet both the company and the OCI users needs, by integrating the research with newly adopted technologies and growing trends, which nowadays are very popular and widely demanded by the market.

The current work comes with the need of mitigating a commonly encountered issue related to the use of the OCI console and the monitoring of cloud resources. Subsequently, the proposal solution relies on a scalable and flexible system, leveraging a microservice-based architecture and oriented to the end user. Throughout this dissertation, the various aspects of the development and deployment process have been outlined, as well as the overall architecture of the system has been fully described and analysed.

The GUI has met with support from OCI users within Technology Reply, due to its intuitive, immediate and simple nature. The resource-related metadata are presented in a clear and compact way through the use of Material Table, while the colors and the icons are kept consistent between screens of similar function. Moreover, the authentication stage guarantees that the access is kept secure and the information are reserved only to the trusted users. The results of this research support the idea that also the QoS experienced by the end user (in terms of latency, number of errors and served requests per second) is quite good, compared to the OCI console's one. As can be seen from section 5.3, the system is able to accommodate an adequate number of requests per second, by keeping the number of errors to a minimum and maintaining the response time under acceptable thresholds. But for the moment it is not meant to be a ready-to-use application, but rather a prototype that is suggested to be used in a more restricted context, such as the Technology Reply headquarter in Turin.

However, this study settles the groundwork for future researches and turned to be a good starting point to possible improvements and extensions. The most evident limitation is of course the cluster dimension; in fact the the number of worker nodes, as well as the amount of CPU provided to each node, should be increased in order to accommodate more requests per second and provide a lower latency (obviously with a consequent and consistent increase in operating costs). Furthermore, each microservice should interact with a separate database instance rather than sharing the same DB. In this way individual data stores cannot be directly accessed by other microservices, and persistent data is accessed only by APIs. This would ensure the removal of a single point of failure and, as consequence, a better resiliency of the overall application. Another possible improvements could involve the scaling system, which relies only on CPU usage. In fact, the HPA should be able to evaluate multiple metrics in the decision mechanism in order to efficiently choose the correct number of replicas to run.

Further research is needed to extend the number of functionalities requested by the OCI users and that can be integrated in the system. A natural progression of this work could be the definition of a cost optimization strategy by, for instance, trying to automate the process of starting/stopping a machine based on certain user-defined policies.

Finally, an additional improvement could go in the direction of automating the software development, as much as possible, and letting the different stages of the implementation process interact with each other. Therefore, it should be possible to deliver changes more quickly, applying any updates in the system by means of a Continuous Integration (CI) pipeline. Additionally, it can be adopted a Continuous Deployment (CD) pipeline to deploy the system continuously into production environment. Under such circumstances, the CI or CD pipeline could execute every change introduced into the system but they should be combined with a testing strategy that includes both automated tests and monitoring in production. This behavior works best when organizations adopt a DevOps culture. Hence, many practitioners and researchers advocate that MSA has a natural progression of embracing DevOps, due to the fact that it brings additional productivity through the use of tools chain and fast feedback mechanisms.

Bibliography

[1] Okcan Yasin Saygili, “Oracle IaaS: Quick Reference Guide to Cloud Solutions“, Istanbul, 2017, ISBN-13: 978-1-4842-2831-9, doi:10.1007/978-1-4842-2832-6

[2] Safonov, O.Vladimir, “Trustworthy Cloud Computing“, John Wiley & Sons, 2017, ISBN: 9781119113515

[3] Keith D.Foote, “A Brief History of Cloud Computing“, June 22, 2017

[4] Roopesh Ramklass, “Oracle Cloud Infrastructure Architect Associate“, McGraw-Hill Education, 2020, ISBN: 978-1-26-045260-0

[5] Prasenjit Sarkar, Guillermo Ruiz, “Oracle Cloud Infrastructure for Solutions Architects“, Packt Publishing Ltd., August 2021, ISBN: 978-1-80056-646-0

[6] “Oracle Cloud Infrastructure Platform Overview“, Oracle and/or its affiliates, April 2021

[7] `https://docs.oracle.com/en-us/iaas/Content/General/Concepts/regions.htm`

[8] Andre Correa Neto, “Oracle Cloud Infrastructure Compartments“, May 9, 2019

[9] “Oracle Cloud Infrastructure Privacy Features“, Oracle and/or its affiliates, October 26, 2020

[10] Changbin Gong, “Best Practices for Identity and Access Management (IAM) in Oracle Cloud Infrastructure“, March 2018

[11] `http://docs.oracle.com/en-us/iaas/Content/Identity/Tasks/callingservicesfrominstances.htm`

[12] `http://docs.oracle.com/en-us/iaas/tools/oci-cli/2.9.2/oci_cli_docs/cmdref/iam/dynamic-group.html`

[13] `https://docs.oracle.com/en-us/iaas/Content/Compute/References/computeshapes.htm#dvhshapes`

[14] `https://docs.oracle.com/en-us/iaas/Content/Block/Concepts/bootvolumes.htm`

[15] `https://docs.oracle.com/en-us/iaas/Content/Block/Concepts/overview.htm`

[16] `https://docs.oracle.com/en-us/iaas/Content/Object/Concepts/objectstorageoverview.htm`

[17] Jyoti Shah, “Simplifying Oracle Database Management on OCI with Cloud Service“, 2019

[18] Sander Almekinders, “ServiceNow adds support for Oracle OCI“, April 2021

[19] A.R.Earls, “Multi-cloud strategy“, 2019

[20] X.Larrucea, I.Santamaria, R.Palacios, and C.Ebert, “Microservices, IEEE Software,“ pp. 96-100, 2018

[21] W.Hasselbring and G.Steinacker, “Microservice Architectures for Scalability, Agility and Reliability in E-Commerce,“ in *Proceedings of the 1st International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden*, pp. 243-246, 2017

[22] E.Akentev, A.Tchitchigin, L.Safina and M.Mzzara, “Verified type checker for Jolie programming language”, *arXiv preprint arXiv:1703.05186*, 2017.

[23] T.Černý, M.J.Donahoo and M.Trnka, “Contextual understanding of microservice architecture: current and future directions”, in *SIGAPP Applied Computing Review* , pp. 29-45, 2018

[24] Lewis and M. Fowler, “Microservices”, March 25, 2014

[25] A.Balalaie, A.Heydarnoori and P.Jamshidi, “Microservices Architecture Enables Devops: Migration to a Cloud-Native Architecture, IEEE Software,“ pp. 42-52, 2016

[26] A.Balalaie, A.Heydarnoori and P. Jamshidi, “Migrating to cloud-native architectures using microservices: An experience report,” in *In Proceedings of the 1st International Workshop on Cloud Adoption and Migration*, September 2015

[27] X.Larucces, I.Santamaria, R.Colomo-Palacios and C.Ebert, “Microservices”, in *IEEE Software*, pp. 96-100, 2018.

[28] R.T. Fielding, “Architectural Styles and the Design of Network-Based Software Architectures”, doctoral diss., Univ. of California, Irvine, 2000.

[29] V. Lenarduzzi and D. Taibi, “MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product”, at the *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2016

[30] Eric Evans, “Domain-driven design“, Addison-Wesley Professional, 2003

[31] Chris Richardson, “Microservices Patterns“, Manning Publications Co., ISBN: 9781617294549

[32] Available: `https://docs.docker.com/engine/`

[33] A.Sviatoslav and C.Sergey, “Advantages of Using Docker for Microservices“, February 2021

[34] Available: `https://docs.docker.com/get-started/overview/`

[35] Deepak Vohra and Massimo Nardone, “Kubernetes Microservices with Docker“, April 2016

[36] Available: `https://kubernetes.io/it/docs/concepts/overview/`

[37] Available: `https://cloud.google.com/kubernetes-engine/docs/`

[38] `https://docs.oracle.com/en-us/iaas/Content/Network/Tasks/managingVCNs_topic-Overview_of_VCNs_and_Subnets.htm`

[39] Available: `https://spring.io/guides/gs/actuator-service/`

[40] Available: `https://www.baeldung.com/spring-autowire`

[41] Available: `https://www.mongodb.com/compatibility/spring-boot`
[42] Available: `https://angular.io/guide`
[43] Faren Faren, “KONG, The Microservice API Gateway“, July 20, 2018