Università degli Studi di Torino

DIPARTIMENTO DI INFORMATICA
Corso di Laurea in Informatica

Tesi di laurea

Microservice on Amazon Web Services

Candidato:
Silvestro Stefano Frisullo
Matricola 832813

Relatore:
Professore Enrico Bini

Anno Accademico 2019-2020

Contents

Ringraziamenti

Mentre scrivevo questa tesi è scoppiata la pandemia da COVID-19. Le conseguenze in Italia sono state estremamente gravi, ma questa crisi ha portato con sé degli inaspettati e favorevoli sviluppi. Internet mai più che in questo momento si è rivelato una risorsa fondamentale in quanto in tutto il paese, improvvisamente, si è imposto lo smart-working, qualcosa che fino a prima del virus sembrava impossibile. Migliaia di persone che si spostavano inutilmente ogni giorno adesso lavorano da casa, rendendo ancora più evidente la quantità di risorse e di tempo che avremmo risparmiato se queste soluzioni fossero state adottate prima e anche l'inquinamento è stato in gran parte abbattuto.

Ringrazio il Professore Enrico Bini per la sua preziosissima guida e il continuo supporto in questo percorso.

Ringrazio tutta la mia famiglia per avermi sempre sostenuto in ogni passo della mia vita, in particolare mio zio Piero che con la sua pazienza è sempre stato al mio fianco.

Ringrazio Valeria per essermi sempre vicina e motivarmi e Silvia per aver reso una quarantena qualcosa di molto più piacevole.

Ringrazio tutti gli amici, per quello che gli amici fanno.

Ringrazio inoltre Reply Storm per avermi dato la possibilità di imparare tante nuove cose e introdotto al mondo del lavoro. In particolare, un ringraziamento sentito va a Piera Limonet e Girolamo Piccinni, che mi hanno seguito e guidato scrupolosamente. Un grazie va anche ad Antonio e a tutto l'IoT team, che quotidianamente mi hanno aiutato a risolvere ogni problema.

Abstract

English

Cloud computing is one of the most important evolutions in computer science in the last decade and Amazon AWS is a prominent cloud service provider.

This thesis will follow the implementation of a microservice on AWS in Python and ElasticSearch, first exploring the concept of cloud computing and virtualization, and then explaining the services of AWS. The microservice operates both on the front end and the back end. The back end will create, manage and update product indices on ElasticSearch, and the front end exposes several APIs to query these indices.

Italiano

Il Cloud Computing è una delle evoluzioni più importanti in informatica negli ultimi 10 anni e Amazon AWS è uno dei migliori provider sul mercato.

Questa tesi segue lo sviluppo di un microservizio su AWS in Python e ElasticSearch, prima esplorando i concetti di cloud computing e virtualizzazione e poi spiegando i servizi di AWS. Il microservizio avrà sia una logica back end che una logica front end. In back end consente di creare, aggiornare e gestire degli indici di prodotti su ElasticSearch, mentre il front end espone diverse API per fare delle query su questi indici.

Chapter 1

Cloud computing and microservices

1.1 Cloud Computing

First appeared in the late 1996[1], the term *Cloud computing* is the availability over the Internet of data storage and computing power, but it became popular only in 2006 when Amazon.com launched Elastic Compute Cloud, a virtual computer renting service. Cloud computing is deployed in different models, according to the needs of the customers using it.

1.1.1 Private Cloud

Private Cloud is an infrastructure running on the internet that is managed and used only by a single organization. Usually, given the incredibly high cost and the operational complexity of running a private cloud infrastructure, this deployment model is only used by big companies or institutions.

There are some critical fields in which a private cloud is the only possible choice, like healthcare, financial services or the military sector. These companies use the highest level of security and thus using a private cloud is an additional protection, not to mention the need to satisfy strict regulations, laws, security policies or political reasons.

1.1.2 Public Cloud

Public cloud is the most used deployment model, and probably the one who made the *Cloud* so popular. In the public cloud, we have a provider offering

[1] `https://www.technologyreview.com/s/425970/who-coined-cloud-computing/`

computing services on the internet to other organizations or individuals. The most popular providers are Amazon Web Services (AWS) (more details in Section 1.6), IBM, Oracle, Microsoft, Google, etc...

This model best represents the cloud as a concept because the users do not need to manage computers and can focus solely on the service. The huge variety of services on the market allows the adoption of different business models: some services are free, others are *pay-as-you-go* and others use a classical rental business model.

1.1.3 Hybrid Cloud

Hybrid Cloud is a combination of public and private clouds, It is the richest model since it offers the benefits of both a private cloud and a public cloud. For example, companies can store sensitive data in a private cloud and rely on a public cloud for other needs, such as storing non-sensitive data or deploy a service.

Another key benefit of a hybrid cloud is its adaptiveness and scalability that can deliver results on crucial transitional moments, such as when a company is not big enough to manage or acquire the necessary infrastructure to offer a service. Similarly, when there are traffic spikes, the company can scale from a private cloud to a public cloud to meet the demand in a so-called *cloud burst*.

1.1.4 API: Application Programming Interface

Originally intended as an interface exposed by a software, now indicates every web service that is used as an extension of another one. APIs simplify the access to complex services by proving a more abstract layer that is easily understandable and more suited to the needs of other developers. The most common form of API and the type used in this project is Web API.

Web API

A web API is an API specifically designed for a web service and uses HTTP requests to get or send information. It consists of a set of *endpoints* that wait for a call and return a response once called, similarly to a method or a function in a programming language.

Both the *call* and the *response* are documents that adhere to a specific pre-defined format and are usually XML[2] or JSON files. Endpoints are ac-

[2]Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents

cessible via a URI[3] with HTTP protocol.

1.2 Virtual machines: the building blocks of cloud computing

Before explaining the different service models of cloud computing, it's necessary to introduce *virtual machines*. A virtual machine is a software that emulates a physical computer, completely independent from the hardware running it. The physical hardware running the virtual machine is called the *host*, and the virtual machine is called the *guest*. The *Hypervisor* is the software that creates the virtual machine and can be classified into two types.

1.2.1 Type-1 Hypervisors

Type-1 or bare-metal hypervisor runs directly on the host hardware, without any operating system in between. By separating the *guest* operating system from the physical hardware, Type-1 hypervisors allow running several virtual units.

This is commonly used in server virtualization to create a *virtual private server*, a virtual server usually provided by a cloud computing vendor, but at a much lower price than a physical one. Some of the most famous are the Xen Project and Microsoft Hyper-V[4].

The Xen Project

Developed at the Cambridge University by Ian Pratt[5] and Keir Fraser[6] and released in 2003, the Xen Project is free and open-source software distributed under the GNU General Public License[7] version 2.

The Xen hypervisor distinguishes the *dom0*(usually Linux or BSD) virtual machine, the only one that has direct access to the hardware by default, from *domU*, other unprivileged VMs that are managed from *dom0*. The Xen Project is used in enterprise-grade solutions from IBM and AWS (see Section 1.6). The latter used Xen on its EC2 (see Section 1.6.2) service from its

[3]Uniform Resource Identifier (URI) is a string that uniquely identifies a resource

[4]`https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/`

[5]`http://www.csap.cam.ac.uk/network/ian-pratt/`

[6]https://www.crunchbase.com/person/keir-fraser

[7]https://www.gnu.org/licenses/gpl-3.0.html

launch in 2006 until the shift to the Nitro Hypervisor (see Section 1.2.1) in 2017.

Nitro Hypervisor

After more than 10 years of use, AWS shifted to Nitro Hypervisor, a custom-built version of the KVM Hypervisor[8]. The team that developed Nitro [4] was working on improvements for EC2 (see Section 1.6.2) instances and addressed the problem tied to the Xen Hypervisor. The latter requires to set up a management partition, called *dom0* on each server, thus taking resources that could be used by the other VMs. On top of that, it is needed to balance the resources between the management partition and the VMs, and this task can take time and effort.

With time, hardware manufacturers like Intel[9] added virtualization handling to the CPUs, making easier to develop new technologies. Thanks to these advancements, the Nitro team introduced ASICs (application-specific interface cards) that allow virtual machines to make direct hardware invocations, thus removing the need for a management partition.

1.2.2 Type-2 Hypervisors

Also known as *hosted hypervisors*, this software runs directly on an operative system like a normal program. The *guest* operative system relies on the *host* OS for operations like memory and storage management, networking and general hardware related tasks.

In this category, we found commonly used software like VirtualBox[10], VMWareWorkstation[11], and QEMU[12]. Even if the aim of type-1 and type-2 hypervisors remains the same, it is important to notice that type-2 hypervisors introduce a significant level of latency due to the host operative system. But a significative advantage of type-2 hypervisors is the low cost and ease of use (generally they don't require more than a few clicks to run) so they are an ideal solution for small offices and test environments.

[8] `https://www.linux-kvm.org/page/Main_Page`

[9] check the white paper at `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-enabling-intel-virtualization-technology-features-and-benefits-paper.pdf`

[10] `https://www.virtualbox.org/`

[11] `https://www.vmware.com/products/workstation-pro.html`

[12] `https://www.qemu.org/`

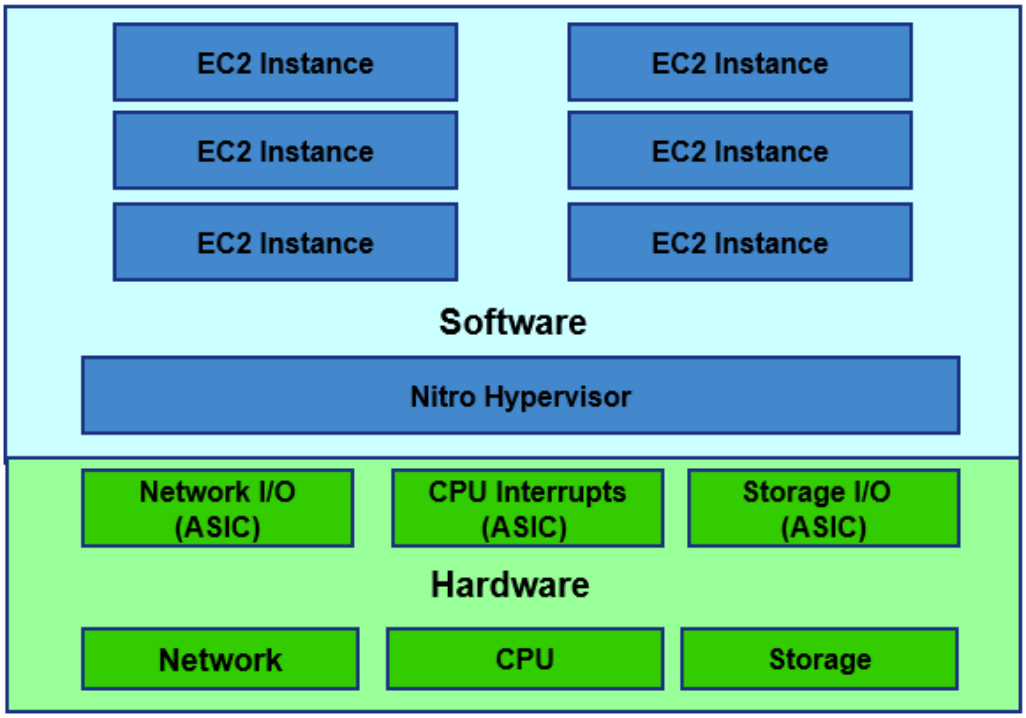

Figure 1.1: Scheme of EC2 instances running on Nitro hypervisors `https://www.metricly.com/aws-nitro/`

1.3 The service model stack

Cloud computing has different service models that operate at a different level of abstraction.

1.3.1 Infrastructure as a service (IaaS)

Infrastructure as a service is a cloud computing paradigm that provides computing resources on a virtual machine through high-level APIs (see Section 1.1.4).

IaaS vendors can offer other services like networking tools, storage, and other software bundles. IaaS can be thought of as the most rudimental level of cloud computing .

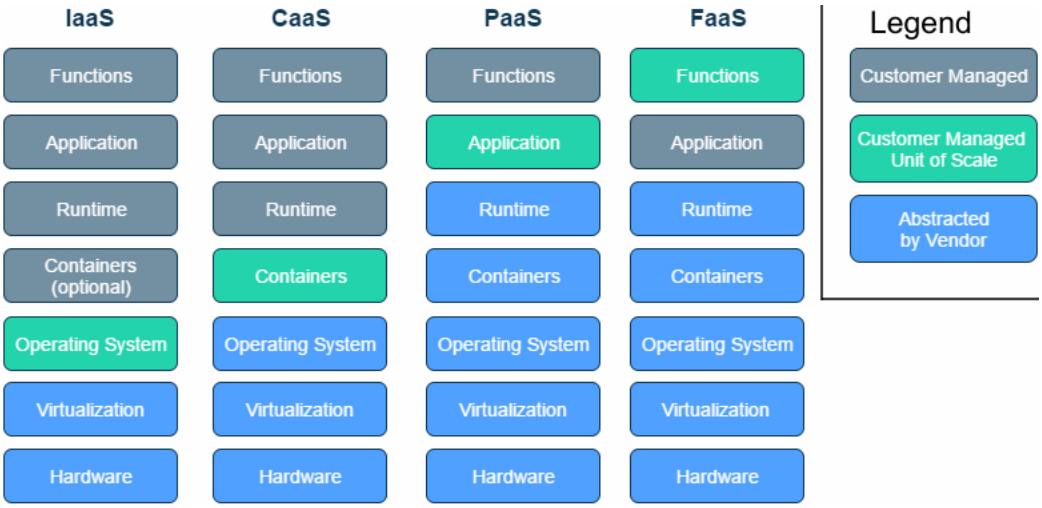

Figure 1.2: The service model stack, source `https://serverless.zone/abstracting-the-back-end-with-faas-e5e80e837362`

1.3.2 Container as a Service (CaaS)

Containers

To properly understand CaaS it's important to explain what a container is.

In operating systems virtualization, containers are isolated user spaces or environments implemented by the kernel, in AWS usually an Amazon Linux[13] machine image.

A typical use for containers is to separate programs for better overall security because processes running inside the container can't communicate with processes inside the OS. Containers don't require an entire operating system to run, and their footprint is very small, usually a few megabytes and this allows them to have very little start-up time and dramatically reduce the costs of data centers.

Another advantage of containers is their high portability because a containerized app has all the dependencies and configuration files it needs to run. The most popular software for containers is Docker[14]

CaaS

CaaS is a cloud computing model that allows the creation and management of containers and clusters of containers, usually trough APIs and web-based

[13] Amazon Linux is a distribution similar to CentOS

[14] `https://www.docker.com/`

interfaces [12].

CaaS model helps in simplifying the management of containers but also allows container *orchestration*, automating IT functions. The most common CaaS orchestration platforms are Google Kubernetes[15] and Docker Swarm [16]

1.4 Platform as a Service (PaaS)

Platform as a service is a cloud computing model that provides the necessary infrastructure needed to run and manage applications in the cloud, thus removing the complexity associated with such tasks. Paas integrates everything already provided by IaaS (see Section 1.3.1), plus development tools, middleware, database management systems, etc....

Advantages of PaaS are:

- Reducing the cost of IT tools: developers can use the tools they need without having to purchase them.
- Manage the application lifecycle: PaaS allows us to easily build, test, deploy, manage and update applications.
- Reduce coding time: by using pre-built applications, PaaS allows coders to significantly reduce coding time.

1.5 Serverless and FaaS

One of the most popular execution models for cloud computing is Serverless computing, in which the provider takes most of the operational responsibilities by running the servers and managing the virtual machines so that the developers don't have to worry about them. One way of achieving Serverless architecture typically used when building microservices is Function as a service (FaaS), in which the developers only have to provide the code.

This rapidly emerging technology has a lot of advantages for several types of businesses. For startups companies these benefits are crucial:

1. Limit upfront capital costs — shifting infrastructure costs from CapEx to OpEx [17]

[15] `https://kubernetes.io/`

[16] `https://docs.docker.com/engine/swarm/`

[17] Capital expenditures, commonly known as CapEx, are funds used by a company to acquire, upgrade, and maintain physical assets. OpEx is the ongoing cost of running a product, business, or system

2. small time to market since most resources are spent developing the product and not maintaining the underlying infrastructure.

3. optimized distribution of labor cost: more capital to who actually build the core product

1.6 The state of the art

There are several cloud services providers in the market, such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform, IBM Cloud, etc... All of them are excellent providers offering robust infrastructures and high-quality services, so comparing them from a business perspective is beyond the intent of this thesis.

As explained earlier, in this thesis the focus is on AWS since the company that offered me an internship is an Amazon AWS Partner. AWS is a subsidiary of Amazon.com Inc. that provides on-demand cloud computing platforms and APIs on a metered pay-as-you-go basis. AWS is the market leader in IaaS (Infrastructure-as-a-Service) and PaaS (Platform-as-a-Service) for cloud ecosystems, garnering more than 30 percent of the market. The idea of AWS was conceived in 2000 when Amazon was just an e-commerce company struggling with scale problems and the solutions to these problems were the root of a business that would become AWS. The services provided by AWS cover a variety of fields, from Cloud Computing to Machine Learning and Quantum Technologies.

AWS resources are accessible both via browser with an easy graphical interface or via SDK[18] for several languages.

1.6.1 Boto3

Boto3 is the official AWS SDK for python. It offers all the services available on AWS and can be installed via *pip* [19]. Next, we will see in detail the services used within this project.

1.6.2 Elastic Compute Cloud

Elastic Compute Cloud (EC2) instances are the virtual machines of AWS. Users can choose an operative system (Linux, Windows, BSD, etc...) and

[18]SDK stands for Software Development Kit, as the name says it is a software package containing the tools necessary to use a service or a framework.

[19]pip is a package installer for Python, here the official page`https://pypi.org/project/pip/`

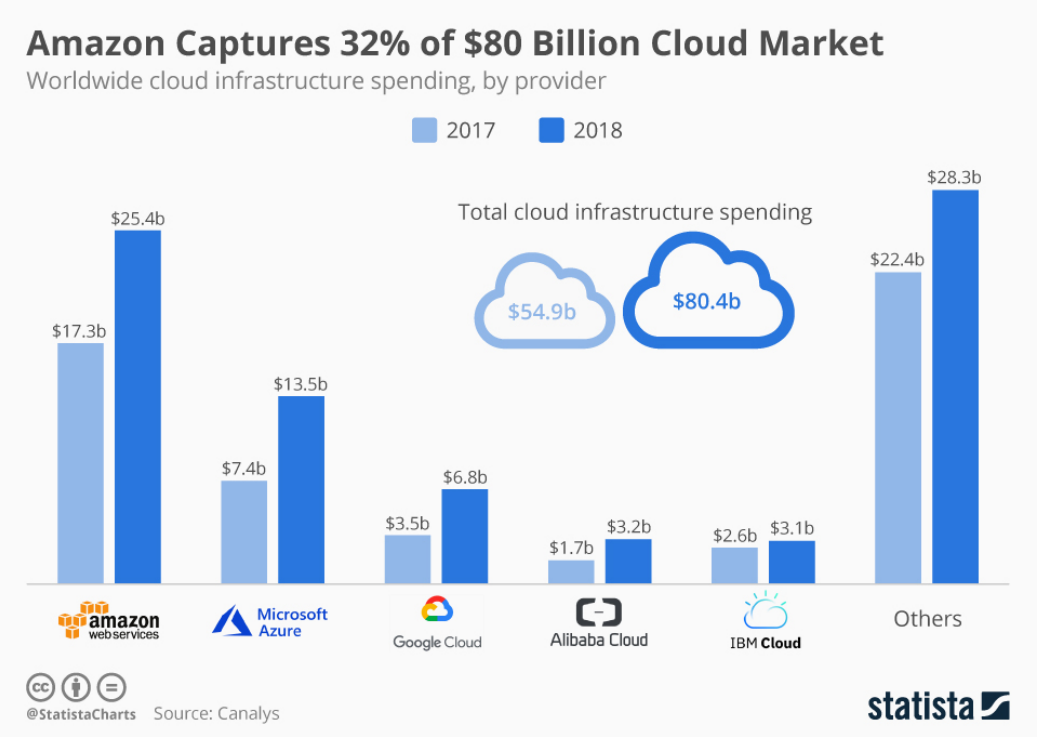

Figure 1.3: AWS captures 32% of the market in 2018 (source from [9])

launch EC2 instances via AWS console on the browser or via command line.

There share several types of instances[20], based on the computing resources available, storage and specific applications:

- General use: instances that provide a balance of computing power, memory and networking resources are A1, T3, T3a, T2, M6g, M5, M5a, M5n, M4
- Compute optimized: instances for applications that need high-performance processors are C5, C5n, C4
- Memory-optimized: instances for application with huge datasets in memory are R5, R5a, R5n, R4, X1e, X1, High Memory and z1d
- Accelerated computing: ideal for floating point number calculations are P3, P2, Inf1, G4, G3, F1
- Storage optimized: instances for an application that needs to access large datasets are I3, I3en, D2, H1

[20] `https://aws.amazon.com/ec2/instance-types/?nc1=h_ls`

1.6.3 API Gateway

Amazon API Gateway is an AWS service for creating, monitoring and maintaining, REST[21] and WebSocket APIs[22] at any scale. The APIs created by API Gateway are HTTP based, thus offer standard HTTP verbs such as POST, GET, PUT and DELETE and conform to the REST protocol. With API Gateway it's possible to define *stages*, namely paths to different environments or different versions of an API. In this project, the APIs can be logically divided into two sets, the backend (details in Section 2.3) and the frontend (details in Section 2.4): the first one manages the interaction between ElasticSearch and SAP Hybris and the latter manages the client's requests to ElasticSearch.

The execution flow of an API invocation usually follows these steps:

1. The client invokes the API
2. a Method Request is made with the chosen HTTP verb and the required parameters
3. then the Integration Request maps the parameters in a JSON[23] format expected by the Lambda
4. Lambda executes the function
5. Integration Response maps back from Lambda's response to HTTP statuses and formats, usually application/JSON
6. Method Response returns the HTTP response

1.6.4 AWS Lambda

AWS Lambda is a computing service that runs code without provisioning or managing servers. Programmers only need to provide the code but can choose the software stack, the platform, and all the dependencies and then Lambda will execute the functions in *containers*.

The Lambda functions can perform any kind of computing task and support many popular programming languages such as Node.js, Python, Java, C#, C++, Go, Rust and Ruby.

The benefits of using Lambda functions can be synthesized in [8]:

[21] Representational state transfer (REST) is a software architectural style that defines a set of constraints to be used for creating Web services.

[22] A WebSocket API is a technology that allows a two-way communication session between the user's browser and a server.

[23] JSON is a data-interchange format https://www.json.org/json-en.html

- Fully managed infrastructure: no need to think about the servers, so the focus is totally on the software
- Pay per use: just pay for the running time and the memory used
- Automatic scaling: instances of the function are created as they are required

When a function is created, the programmer must specify

1. the configurations, such as the maximum memory size,
2. a timeout after which the execution should stop and
3. the permissions of the function, managed by an IAM policy (see Section 1.6.8 for an explanation of IAM).

The computing power given to lambdas is directly proportional to the memory, for example, a function with 128MB of memory will have half of the CPU resources than one with 256MB.

Lambdas can be invoked[24]:

- *synchronously*, in which case the Lambda runtime waits until the handler has finished its execution and then returns to the caller, and
- *asynchronously*, that is when lambda returns immediately to the caller without a result.

A Lambda function is *stateless*, meaning that no session information is stored by AWS, so if you need status information then you must use services like DynamoDB (details in Section 1.6.5) or an Amazon S3[25] bucket.

Cold Start

Lambda functions are executed in containers and containers must be started prior to the execution [5]. This operation introduces latency issues when the function is called after a prolonged period of time. To avoid this problem, known as *cold start*, it's a common practice to ping the function before its environment is deleted, so the function is kept *warm*. Cold start timing can vary based on the language, the memory, and the number dependencies in the code. For example Java and C# runtimes are much slower than Python or Node.js.

[24] `https://blog.symphonia.io/posts/2017-08-18_learning-lambda-part-6`

[25] Amazon S3 is an object store with a REST API, in which objects are stored in uniquely identified buckets

This was true until 4th December 2019, when AWS announced *Provisioned Concurrency*. With this advancement, it's possible to have a predictable start time by keeping *your functions initialized and hyper-ready to respond in double-digit milliseconds at the scale you need.* [1] A *provisioning* process assures that the execution environment is ready *before* the Lambda execution, thus eliminating the delay associated with the creation of it.

1.6.5 DynamoDB

Amazon DynamoDB is a managed NoSQL key-value and document database. DynamoDB is scalable both in storage and throughput and it's accessible via an HTTP API and performs authentication and authorization via IAM (details in Section 1.6.8) roles, making it a perfect fit for building Serverless applications. Tables in DynamoDB must have a primary key.

1.6.6 Elastic Search and Amazon Elasticsearch Service

Elasticsearch (details in Section 1.8) is a search engine based on the Lucene library. It allows us to store, search, and analyze big volumes of data quickly and in near real-time. It is commonly used for applications that require complex search featured. Since the project is built on AWS, the service used is Amazon Elasticsearch Service, which provides ElasticSearch without the need to manage servers and clusters.

1.6.7 CloudWatch

Although not present in the architecture scheme, Cloudwatch is intensely used to collect logs of each event happening in AWS with 1-second granularity, thus making it necessary to have a complete understanding of issues and effectively troubleshooting them.

1.6.8 AWS Identity and Access Management (IAM)

When working on an important project with several developers and different teams it is extremely important to define the roles and the permissions that allow or deny access to AWS resources. This is what IAM does, providing an easy interface to create and manage AWS users and groups each of them with an attached IAM policy.

Also, when a user calls an API, AWS must know who the user is trough *authentication* and what the user is allowed to do to with *authorization.* IAM allows the creation of *users, gropus and roles.*

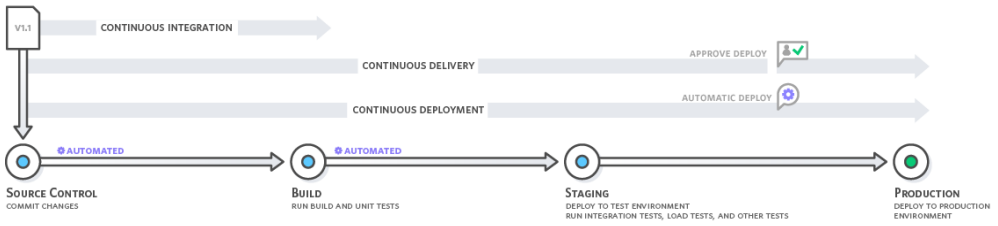

Figure 1.4: Continuous delivery makes the release process automatic, but the deploy is approved by a human.

The difference between a group and a role is that users can be *added* to a group, while a role is *assumed* by a user or other AWS services. The permissions for users, groups and roles are defined by *policies* and by default at least one must be attached to them. In other words, policies have effects on actions targeting specific resources [26]. There are three types of policies:

- *resource-based policies*: define what can be done on a specific resource
- *user-based policies*: what users, groups or roles can do
- *trust-policies*: define who is allowed to assume a certain role

Moving to the authentication problem, we need to introduce *security credentials*. Security credentials can be permanent or temporary and are composed by by an access key ID and an by a secret access key. The first one is added to every API call and the latter is used to sign the call.

1.6.9 AWS Developer Tools

AWS provides some development tools: CodeCommit, CodeBuild, CodeDeploy and CodePipeline. These tools are fundamental to implement continuous integration and continuous delivery (CI/CD), two successful DevOps software development practices that are necessary for a modern application development.

CodeCommit is source control service for git-based repositories, it provides the code that is compiled and tested in CodeBuild, producing a ready to deploy package that is deployed with CodeDeploy, while all of this is controlled and automated with CodePipeline.

[26] resources are specified by Amazon Resource Names (ARNs) that are unique strings. Some of them, like S3 bucket ARNs, are unique globally

1.6.10 Cloudformation

Cloudformation is a free service that allows configuring and automatically deploy the resources of the AWS infrastructure with a simple YAML file [3]. Deploy a cloudformation file can be difficult considering the complexity involved, but in case of errors the deploy stops without affecting the existing configuration. Cloudformation is an expression of the concept of *Infrastracture as a code*, in which a declarative approach in a machine-readable file is used to configure complex cloud computing configurations. Cloudformation automatically creates almost every AWS resource like databases, networks, load balancers, etc....

Another tool called Cloudformation designer further simplifies this process providing a graphical interface that creates resources with a simple drag and drop.

1.7 Microservices

Microservices are an architectural pattern which divides the project into small, decoupled, independent services responding to a particular need. Teams can write microservices in almost every language and them can deploy independently of one another.

As Peter Sbarski[27] explains [10]

> *"a serverless architecture leverages a serverless implementation for each of its components, leveraging FaaS like AWS Lambda for custom logic. This means each component is built as a service with utility, pay-per-use pricing and incurs cost only when used. Each component is a service and exposes no configuration or cost related to the infrastructure it is running on, which means these architectures don't rely on direct access to a server to work. By making use of various powerful single-purpose APIs and web services, developers can build loosely coupled, scalable, and efficient architectures quickly. Moving away from servers and infrastructure concerns, as well as allowing the developer to primarily focus on code, is the ultimate goal behind serverless."*

[27] Read the freely available chapter of this citation at `https://livebook.manning.com/book/serverless-architectures-on-aws-second-edition/chapter-1/v-4/point-7923-47-47-0`

1.8 ElasticSearch

ElasticSearch is a distributed search and analytics engine written in Java[28] that provides a web interface for scalable and real-time search [2]. The fundamental element of ElasticSearch is an index, indexes can be divided in *shards* and each shard can have several replicas. The following scheme can help to understand the terminology of ElasticSearch by making a comparison with a standard relational database:

- Relational DB : Databases : Tables : Rows : Columns
- ElasticSearch : Indices : Types : Documents : Fields

ElasticSearch consumes JSON data and automatically stores the original document and adds a searchable reference to the document, making every field indexed and searchable. ElasticSearch is free and open-source software and its currently in use in several scenarios, the following examples are taken from the official guide:

- GitHub[29] uses it to search in more than 130 billion lines of code
- Wikipedia[30] uses Elasticsearch to provide full-text search with highlighted search snippets
- Stack Overflow[31] combines full-text search with geolocation queries and uses more-like-this to find related questions and answers.

The power of ElasticSearch is such that it can run on a small laptop and scale easily to a cluster of servers handling petabytes of data. In ElasticSearch scale can occur in two ways:

- Vertical: buying more powerful servers
- Horizontal: adding new servers to the cluster

Of the two, the latter is more powerful because provides computing power, spreads the load and increment overall reliability.

A *node* is an instance of ElasticSearch , and a cluster consists of one or more nodes. One node is selected to be the Master Node, thus it can add or remove nodes from the cluster. When working with clusters, it's very important to constantly monitor the cluster health and to understand the symbolism associated with different levels of health:

[28] `https://www.java.com/it/`
[29] `https://github.com/`
[30] The free encyclopedia `https://www.wikipedia.org/`
[31] `https://stackoverflow.com/`

- green: all primary replicas and shards are active
- yellow: all primary shards active, but not all replicas.
- red: at least one primary shard is not active

Document

In ElasticSearch a document is an entity or an object serialized into a JSON file. Documents have keys that represent the name of the fields of the document and values that can be standard types such as numbers, strings, booleans or other complex objects.

A document contains also *metadata*, information about the document itself:

- *index*: the index in which the document is contained. The index name cannot contain commas, cannot begin with an underscore and must be lowercase
- *type*: the class of object that the document is representing. Each object has it's class and each class has it's *mapping*, a schematic definition of its properties.
- *id*: the id of the document that combined with the index and the type uniquely identifies it in ElasticSearch

1.9 The scenario

Before going into the details, I will offer a quick overview of the solution discussed in this thesis.

The customer uses a platform called SAP Hybris[32] from which the clients have to extract a considerable amount of data. The platform currently in use by the customer cannot sustain high traffic originated from multiple clients.

The request of the customer was to have a faster, high availability, fault-tolerant and scalable solution. On the back-end, the customer wants to periodically bulk data transfer to the new database, while the clients can access the same database to retrieve information. So the new solution will be accessed from both sides: back end by SAP Hybris and front end by the clients.

The platform on which the solution will be deployed is Amazon AWS. The back end APIs are:

[32] An e-commerce and product content management software

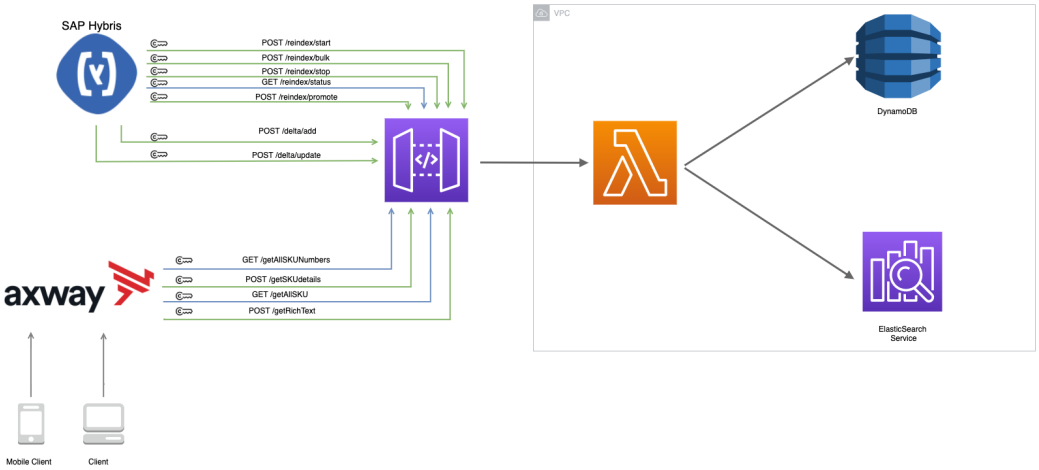

Figure 1.5: Scheme of the solution's architecture

- /reindex/start
- /reindex/bulk
- /reindex/stop
- /reindex/status
- /reindex/promote
- /delta/update

The front end APIs are:

- /getAllSKUNumbers
- /getSKUdetails
- /getAllSKU
- /getRichText

The characteristics of each of these APIs will be discussed in the corresponding sections in Chapter 2.

Chapter 2

API specification and implementation

2.1 Methodology

The definition of a correct set of APIs (as explained in Section 1.1.4) is crucial to the success of the project, because a wrong interpretation of the customer's needs will have a dramatic impact on time, efficiency, costs, and frustration of the IT team. This process is called Requirement Analysis and can be defined as *"the software engineering practice that, at the top level of the software architecture, translates stakeholder needs and expectations into a viable set of software requirements."* [11]

The best tool to design and present the APIs is the Swagger Editor[1] with OpenAPI Specification[2]. It provides an easy to use interface and a great data visualization that helps in a process prone to errors. The file usually consists in YAML [3] code and the output is shown in Figure 2.1.

The swagger file must specify the type of the HTTP request, the format for both the parameters and the responses and the HTTP status code. Once that the swagger file is written and approved by the customer, the implementation of APIs can start.

[1] `https://swagger.io/tools/swagger-editor/`

[2] an API description format for REST APIs

[3] recursive acronym for YAML Ain't Markup Language What It Is, it's a human-friendly data serialization standard

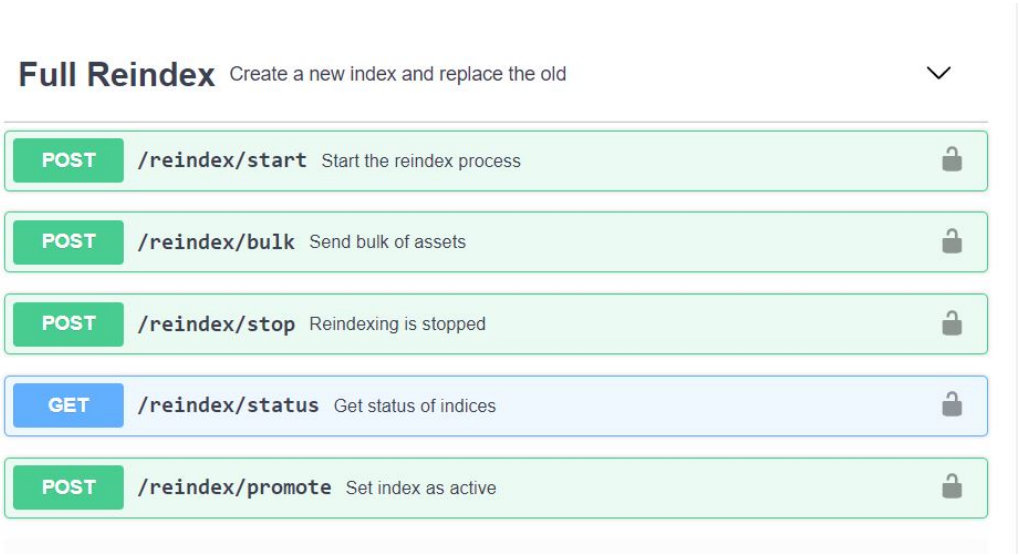

Figure 2.1: The APIs are readable and well presented

2.2 The APIs

As explained in the introduction (details in Section 1.9), the APIs can be divided into two groups, back end and front end. The task accomplished by the back end group is a full reindex of a product catalog, so we need an API to create a new index [4] on ElasticSearch, another one to add hundreds of products to the freshly created index, and finally, an API to stop the process. On top of that, we want an API to monitor all the available indexes, one to mark an index as active (the one that will be used) and one to update existing products.

These APIs must be executed in the right order: first create a new index with /reindex/start, then add data to it with /reindex/bulk and finally stop the process with reindex/stop. Changing the order or using the same call twice (more consecutive reindex/start for example) will result in an error response.

The front end APIs are conceptually simpler since they don't require to be called in a precise order, but are harder to implement, since they execute fairly complex queries on the active index.

[4] An Elasticsearch index is a collection of documents that are related to each other.

Code	Explanation
400	Invalid Request: data not readable or index does not exists
409	Invalid state: reindexing not started or started twice
413	Payload too large: too many assets requested
500	Internal server error, please retry later
200	Success!

Table 2.1: Error codes of interest.

2.3 The back end APIs for Full Reindex

2.3.1 /reindex/start

This API is for starting the reindex process. No parameters are required, but a condition must be satisfied: the reindexing process can't be already started. The function checks with *check_started()* if the reindexing process was already started, and returns error 400 (details in Section 2.1) if it was.

The status of the reindexing process is saved in a boolean value in DynamoDB, so this function checks the status [5] before creating a new index and if it is false then create an index with the UNIX timestamp as a name.

2.3.2 /reindex/bulk

As said before, this API aims to load up to 1000 products in the active index. This value is hardcoded in a global value in the lambda body, however, it is the maximum value that will operate safely, avoiding breaking the lambda payload limit. Decreasing it will not make any change but increasing it will cause troubles, so better leave it as it is. In technical terms, a *bulk operation* is an action performed on large scale, processing a huge number of items.

ElasticSearch offers a Bulk API call that *"performs multiple indexing or delete operations in a single API call. This reduces overhead and can greatly increase indexing speed."* [7] ElasticSearch is a powerful tool, but AWS Lambda has a payload limit of 6MB[6] so for the sake of simplicity and to respect this limit the number of products that can be processed in a single API call is set to 1000.

The function require a *body* parameter containg the index name generated by /reindex/start (details in Section 2.3.1) and an array of assets. The error codes used returned by the API are explained in Table 2.1. A schematic presentation of this endpoint can be →

[5] Check this value the auxiliary function check_started()

[6] https://docs.aws.amazon.com/lambda/latest/dg/limits.html

1. data is not readable → error 400, check error table 2.1.
2. provided index doesn't exists → error 400 check error table 2.1
3. reindexing process was not started or index is not the valid target → error 409, check error table 2.1
4. !(number of assets <1000 && <size 6MB) → 413, check error table 2.1
5. now can add assets
6. return code 200, check table 2.1
7. if any exception return error 500, check table 2.1

2.3.3 /reindex/stop

This endpoint requires the name of the index on which the reindexing was performed. Obviously the function checks that the index provided is the same generated by /reindex/start (details in Section 2.3.1), but since ElasticSearch allows the use of aliases for indexes' names, it also checks that provided alias is equal to the alias in use and that the alias in use points to the index generated by /reindex/start (details in Section 2.3.1).

If these checks are successful, the reindexing status is updated in DynamoDB and the function returns a confirmation message with the number of assets present in the index at the moment.

2.3.4 /reindex/status

This simple function retrieves information from ElasticSearch about all the available indexes, and returns the *active_index* and the array *available_indices* with their respective names, number of assets and creation date.

The simplicity of this endpoint is made possible by using different auxiliary components such as *get_active_index()*, a simple function that returns the active index.

2.3.5 /reindex/promote

This endpoint will set the provided index as active, so it will become queryable by the clients. The returned values of this function are the same of /reindex/status (details in Section 2.3.4), because after promoting the index it calls the status function to give both a confirmation message and an understanding of the situation.

2.3.6 /delta/update

This endopoint updates the specified index with the provided items and each asset is mapped with its SKU number[7] as key. This endpoint has the same limits of the bulk function (details in Section 2.3.2), so a maximum of 1000 assets and 6MB payload, and if they are not respected the function returns error 413, check the error in Table 2.1.

Similarly, if the input data is not readable an error 400 (check Table2.1) is thrown, but the similarities end here since the function loads the items one by one in ElasticSearch calling *es_client.save()*, an auxiliary function in the generic ElasticSearch client that either creates or updates an existing item.

2.4 The front end APIs

These APIs are going to be called by the clients, as shown in 1.5 and offer different endpoints to retrieve data.

An important concept to explain is pagination, since it's frequently used in these APIs due to already mentioned (details in Section 2.3.2) limits of Lambda functions. Pagination is *the process of dividing a document into discrete pages, either electronic pages or printed pages* [8] and it's implemented in every front end endpoint of this project. Solving this problem when working with lambdas is crucial and even very simple tasks can become challenging due to the implementation of pagination.

In order to implement pagination, two parameters are required and extensively used in the code: *from* and *size*. The former identifies the first element returned in a ordered list and the latter the number of elements returned in a single lambda call.

2.4.1 /getAllSKUNumbers

This endpoint requires a boolean parameter *allSKUs* an optional *from* and *size* with default values respectively 0 and 5000. The value 5000 is hardcoded but it is a reasonable amount since the values returned are integers, it's better to use the default value to avoid hitting the lambda's payload limit. If *allSKUs* is set at True returns all the available SKU Numbers, else if set to False returns just the products with the availability group array empty.

For clarification, the *availability group* is an array where are stored the country in which the product is available, for example a product X will have

[7] SKU is the acronym for Stock-Keeping Unit

[8] https://en.wikipedia.org/wiki/Pagination

availability group=["GB","RU","FR"]. The specification for this endpoint is arguable, because it's not a clean design to have an API called *getAllSKUNumbers* and then ask if you want to get all SKU Numbers, but that's what the customer wanted.

The endpoint returns the *total_sku_numbers* and the array *all_sku_numbers*

2.4.2 /getSKUdetails

This API returns an array with detailed product information given an array of product IDs. The input parameter is *sku_numbers* and the data returned is *skus*, in which every product is mapped with its ID as a key. Pagination is not required because there is a limit of 100 products.

If this limit is not respected, the endpoint will return error 413 (check Table 2.1) and if one or more IDs can't be found in ElasticSearch the result is an error message "SKU not found" with the SKU that wasn't found and the classic code 404,check the error Table 2.1.

2.4.3 /getAllSKU

The input parameters are the *availability_group* array (an array of string called "availabilities" in the product JSON) and *language array*, another array of strings that defaults to en-EN. Returns just products containing at least one of the availability groups passed as input in the array called 'availabilities', grouped per availability group.

Let's say we have a product X and a product Y. X has three availability zones (countries in which the product is available) GB, RU and DE, while product Y has GB, RU and FR. If we request availability_group= "GB, FR", ignoring the language for now, we have a result like [X, Y, Y].

The real data format returned with status 200 (check Table 2.1) is

```
{
"total_items": "string",
"from": "string",
"size": "string",
"skus": [
    "0001": {product one info},
    "0002": {}
]
}
```

After these considerations, we must reintroduce the *language_array*, its fallback mechanism, and localizable fields. The localizable fields are pieces

of information inside a product JSON that are available in several languages. This is the algorithm as requested by the customer:

- check if the language parameters are valid
- if none of the input languages is valid(contained in the language table) do not return any product.
- If at least one language is valid, use the language table in the following way
  - For all the fields to be localized
  - show only the values that are matching the languages passed as parameter
  - for each value:
    - if one of the languages passed as a parameter does not match any value
      - try with a fallback language (some languages have fallback languages)
      - The localized values should be always included in a JSON object representing the matched language even in case of multiple languages provided as input input: "Es,LV" Es: "Espagnol", LV: ""

If languages or their fallbacks do not match any localized value, these values won't be localized. The language table is very long (more than 300 lines) and pasting it here wouldn't be helpful, let's just say that for every language, there is an array of fall-back languages that can be occasionally empty, and to make it usable in this python project I loaded it in a dictionary using a script.

Like other APIs, getAllSKu implements pagination (details in Section 2.4) using the parameter *next_token*. To properly implement pagination, we divide the results into a set of finite pages and then assign to each one a token ID. When calling the API for the first time we just provide the parameter for the query and get as a result the first page and *next_token*. From the second call to the last, we just provide the *next_token*, following the chain of tokens until the last page.

When started, the function establishes the connection to DynamoDB using Boto3 (details in Section 1.6.1) methods. After a successful connection to DynamoDB, the function checks if the input is valid and differentiate between two situations:

- The first call, so the client provided the parameters for the query ()
- 2nd to n-th calls: the client provided *next_token*, so the query was already executed and the function must retrieve the result from DynamoDB.

Let's start with the first case. The idea is:

1. Load all results from the elastisearch query in *all_ids_zone* and calculate *total_items*.
2. write*asset_array_to_dynamo* to DynamoDB, in blocks of lenght = *pagination_size*,giving each of them an unique token id= request_id+from.
3. return first asset array and *next_token* for pagination

To complete step 1 we must understand how many results the query will return and to accomplish this we need an auxiliary function *get_total_hits* that will perform the same query, but returns just the number of total hits, so it's very light and quick. With this number, we can calculate how many iterations we need to extract all the results while respecting the pagination limit, that still applies since we are accessing ElasticSearch with a lambda function.

With a for cycle, we extract all the products matching the query and put them inside an array.

The second step is a nice for cycle over a number calculated in a very un-pythonic [9] fashion

```
for repetition in
    range(int(math.ceil(len(all_ids_zone)/pagination_size)))
```

We have a huge array with all the product IDs, so we need to split it in small arrays that respects the pagination limit, by extracting the elements in each iteration from *dynamo_from* (which is 0 in the firt iteration) to pagination_size*(repetition+1). Then each of the small arrays must be sent to DynamoDB with its token as ID, and the time to live set to half an hour to prevent DynamoDB to became overpopulated with useless data. After, *dynamo_from* is set to (repetition+1)*pagination_size and the cicle starts again.

In step 3 we have a DynamoDB table full of entries with unique keys containing groups of product IDs (SKUs) grouped by language availability. These groupings are not precisely contained in a single entry but are

[9] https://docs.python-guide.org/writing/style/

shared across multiple ones. Now we need to return the first page of results, but before doing it we must localize the language for each product. To do it, we need the auxiliary function *localize_fields(asset, request_languages)*, which takes one product and a list of languages to localize in about 50 different properties. These fields are generally in the form *property-X/property-Y/property-Z*, they are already defined in the requirements specified by the customer and are stored in *field_to_localize*.

For each ID in the array in DynamoDB, the function retrieves the complete product from ElasticSearch, runs *localize_fields* on it, adds it to an array and finally returns the array.

Chapter 3

The phases of the internship

In this chapter, I will describe the stages of the work developed during the internship. I began this internship with no knowledge about AWS, Python, and ElasticSearch. This sounds drastic for someone who is about to work on a project entirely based on these technologies, but what motivated me was exactly the fact that I knew nothing about the most important technologies in use today. This was also my first experience in a real IT company. My internship was articulated in several phases that will be explained in the next sections.

3.1 Studying AWS

First, I needed an understanding of how AWS works, so I was asked to watch several educational videos on LinuxAcademy.com[1] a popular learn-by-doing cloud training platform. The course gave a basic understanding of AWS services and cloud computing in general and lasted about 3 days, because it was composed of both video lessons and exercises to complete alone. After this initial phase, I began the study of Python. The first task that was assigned to me was to complete a function from a previous project. This had the intent to both let me understand the fundamentals of Python and also to let me do something useful.

The development of this function took over ten days because it was a bit complex and at the same time I needed to understand the code that the function interacts with in order to properly address some problems. This was also my first real-world release since the function was tested by the dedicated team and, after some back and forth due to some bugs, it was approved and staged in pre-production.

[1] `https://linuxacademy.com/`

After this inial phase, I was able to move around AWS fairly quickly for a beginner, but I had no knowledge about ElasticSearch.

3.2 API definition

The first real task was to discuss the API definition. It was a challenging experience because we needed to interact with a team in Germany, who also conducted the tests, with video conferences on Webex[2]. This was a delicate phase, we needed to analyze and understand the request of the customer, propose a solution and then validate it with the other team in Germany and the IT team of the customer. Of course the expertise of the managers provided the necessary support to properly address these problems with a quick and effective solution.

After drafting a definition, I began transcribing the APIs in the Swagger format (Figure 2.1). This was an easy but delicate phase, since once written and approved the definitions could no longer be changed.

Finally, after a couple of days of work, the swagger file was ready and sent to the other team for approval.

3.3 Implementation

Usually, the practice is to wait until the approval and then start implementing it, but we decided to start the implementation right after the swagger file was complete. From a management point of view, this was a *risk* because the customer can make changes to the APIs thus making the implementation useless. But since the approval required almost a week, the managers were willing to take this risk, so I began implementing the APIs. When we received the approval, we were enthusiastic because the risk taken was worth it.

The implementation phase was the hardest because I had to learn Python and ElasticSearch. Learning Python was not so difficult, but ElasticSearch was more challenging, mostly because there are very few answers on StackOverflow[3]. So I began the implementation of the APIs.

ElasticSearch is a complex software, and even if it provides a good python client, I had to use an intermediate level of simplified APIs. Luckily, a lot of them were already implemented by the company, but I needed to implement a couple more to work efficiently.

The most important auxiliary functions are:

[2]Cisco Webex is a leader video conferences platform `https://www.webex.com/`

[3] A popular Q&A community `https://stackoverflow.com/`

- index(): index a new document
- bulk(): Bulk a list of documents
- status(): retrieves the status of the indices
- list_indexes(): get the list of all the indexes and their aliases

3.4 Testing the implementation

It's important to point out that because of the internal structure of the company, I was assigned to the coding, and another team is in charge of testing, but for completeness, in the next section I will offer a theoretical overview of what testing is.

3.4.1 What is software testing?

Software testing is a series of activities which aim is to check the quality of the software and find bugs or other defects. Depending on the approach, testing can be done when the software is implemented or like in the agile approach[4] concurrently with programming.

Since the number of possible tests is almost infinite even for trivial programs [6], software testing aims to conduct just a meaningful portion of them, but in various environments and under a lot of different conditions. Usually, the testing team is separate from the developer team. Testing cannot determine all the bugs in a program but can compare the output of the program against an expected value or behavior. For this reason, software testing cannot assure that the software will operate successfully under all conditions, but can prove that it doesn't work in a specific condition.

Errors are not always located inside the software's code, sometimes they can be generated by a requirement absence. There are different approaches to software testing.

3.4.2 Black-Box testing

We must imagine the program as a black-box we know nothing about, except for the specification. Under these circumstances, we must proceed with an *exhaustive input testing* because there could be some conditions under which the program fails.

[4]Agile is a software development approach `https://agilemanifesto.org/`

For example, let's suppose that we have a program for checking equilateral triangles that accept three input values, and it returns true if the values represent an equilateral triangle, false otherwise. Since we are facing a *balck-box* software, we can't tell if inside the code there is a function that, for some unknown reasons, returns *false* if we provide the values 123, 123, 123. So we should try every possible value allowed by the variables used to represent the input values. This is a very difficult task, almost impossible in a more complex software, so there are two implications [6]:

1. *You cannot test a program to guarantee that it is error-free*
2. *Since exhaustive testing is out of the question, the object should be to maximize the yield on the testing investment by maximizing the number of errors found by a finite number of test cases*

So in a real application, we cannot guarantee that the software will work under all conditions, but we can be reasonably confident that it will work under conditions similar to those tested successfully.

3.4.3 White-Box testing

White-Box testing or *logic-driven*, is a testing technique that examine the logic flow of a program. Like in the black-box testing (details in Section 3.4.2), it's impossible to test every possible flow of the program, so we cannot proceed with *exhaustive* testing. Even if exhaustive testing could be done, several problems will still not be discovered. For example, the program could still not satisfy the specifications, or could have fewer paths than necessary (unchecked conditions).

One white-box testing technique is API testing, in which an API endpoint is tested to check if it returns the correct result.

Bibliography

[1] James Beswick. News for aws lambda – predictable start-up times with provisioned concurrency. https://aws.amazon.com/blogs/compute/new-for-aws-lambda-predictable-start-up-times-with-provisioned-concurrency/, 2019.

[2] Clinton Gormley and Zachary Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.

[3] Stefan Luber. Was ist aws cloudformation? https://www.cloudcomputing-insider.de/was-ist-aws-cloudformation-a-857705/, 2019.

[4] Mike Mackrory. Aws nitro—what are aws nitro instances, and why use them ? https://www.metricly.com/aws-nitro/, 2017.

[5] J. Manner, M. Endreß, T. Heckel, and G. Wirtz. Cold start influencing factors in function as a service. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 181–188, Dec 2018.

[6] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[7] ElasticSearch official documentation. Bulk api elasticsearch. Official documentation https://www.elastic.co/guide/en/elasticsearch/reference/current/docs-bulk.html, 2019.

[8] Danilo Poccia. *AWS Lambda in Action: Event-driven serverless applications.* Manning Publications Co., 2016.

[9] Felix Richter. Amazon captures 32% of $80 billion cloud market. available at https://www.statista.com/chart/7994/cloud-market-share/, 2019.

[10] Peter Sbarski. *Serverless architectures on Aws: with examples using Aws Lambda*. Manning Publications Company, 2017.

[11] Richard F Schmidt. *Software engineering: architecture-driven software development*. Newnes, 2013.

[12] IBM Services. What is containers as a service (caas)? `https://www.ibm.com/services/cloud/containers-as-a-service`, 2019.