POLITECNICO DI TORINO

Master's Degree in COMPUTER ENGINEERING

Master's Degree Thesis

DEVELOPMENT OF A WEB APPLICATION IN AN AGILE TEAM IMPROVING THE USER STORY EFFORT ESTIMATION PROCESS

Academic Supervisor

Prof. Antonio Vetrò

Industrial Supervisor

Dott. Massimo Gengarelli

Candidate

Amedeo Sarpa

October 2022

Abstract

In this master thesis work is presented my experience in a French consultant company during a six-month internship. The goal of the internship was to develop a web application in an Agile team. The main functionality of the application is the management of clients and consultants of the company. The agile team was composed by six interns developers at their first working experience, who periodically exchanged the roles of Product Owner (PO) and of the Scrum Master (SM). The thesis work extends the internship report committed at the end of the stage. Particularly, are added the chapter 4, describing an activity research made in parallel during the stage, and the sections where are described tools used for the deployment (kubernetes, terraform and helm).

The team started from scratch, defining the microservice architecture of the system and choosing the technologies that most suited each microservice. The application was deployed in a cloud environment, exploiting all its advantages: more flexibility, easier maintenance and users that do not need to directly manage resources. Developers were in charge of setting up the cloud infrastructure. The development followed the DevOps philosophy: tools like Terraform and GitLab pipelines have been used in order to automatize operations. In the thesis work are described all the steps of the project and the tools used, providing also practical examples.

The research activity, that was carried out in parallel with the development, proposes a possible approach to improve the user story estimation process. It proposed metrics and possible indicators that could badly affect this process. The possible indicators of bad estimations proposed are: 1) "New technology": when a user story requires the introduction of a new technology. 2) "Spike", user story that requires also to implement transversal tasks. 3) "New functionality", when a user story introduces a new set of functionalities. During the development were collected both quantitative data, such as effort in hours spent on each user story, and qualitative data, like developers' feedback.

These data were analysed and discussed before each sprint planning and, particularly, in a special meeting at the middle of the development in order to see if, with a more experience in the project and with the analysis of data collected, the estimation process improved. Developers' feedback, i.e. difficulties encountered during the implementation, were collected in order to identify the most common reasons of bad estimations. What came out is that the role of the Product Owner is crucial and must not be underestimated, in fact, 15 user stories out of a total of 20 identified were badly estimated because the goal of the story was not clear during the planning or client's needs were not fully understood. Furthermore, a factor that most influenced the team was the lack of experience of developers. The activity demonstrated that the effort estimation process could be improved by collecting and analysing data. The number of user stories bad estimated decreased over time, no more exceeding 2 bad estimations per sprint. Regarding the possible indicators of bad estimations, not significant conclusions can be drawn about because there were no strong correlations. However, the team thinks that developers must pay attention when estimating a user story that introduces a new set of functionalities or requires to perform transversal tasks. In the first case, the team has to make sure that the new functionality's goals are clear to every developer. In the second case, the team must discuss possible difficulties and factors that could affect the implementation of transversal tasks.

keywords: Agile methods; Effort estimation; Microservices; DevOps; Cloud computing.

Table of Contents

List of Tables

List of Figures

Acronyms

BM
: Business Manager

BU
: Business Unit

SA
: Sophia Antipolis

SM
: Scrum Master

PO
: Product Owner

POC
: Proof of concept

JWT
: Json Web Token

CDC
: Cahier Des Charges

AD
: Active directory

Chapter 1

Introduction

1.1 The company

"Alten is a French multinational consultant company founded in 1988, it has offices in more than 28 countries and works with key factors in the Aeronautics and Space, Defense and Naval, Security, Automotive, Rail, Energy, Life Sciences, Finance, Retail, Telecommunications and Services sectors" [1]. Alten's DNA combines human values, a culture of excellence and expertise at the service of its costumer. In France there are several Business Units, one in Sophia Antipolis, were I made the internship, a delivery center that manages and develops IT projects for Alten's clients. The core of the company are two figures: Business Managers and Consultants. The latter are people who provides expert advice professionally to other companies, the clients. The former instead are the people who are directly in contact with the clients to satisfy their needs, selecting the best consultants that fit their requirements. Others relevant figures inside the company are the HR, that is in charge of recruiting and retention of employees within a company, the CMC, in charge of career, which follows the career evolution of the pool of consultants of the BU and the department director, who oversees the BMs. I did the internship in the company with other nine guys, all interns. We were divided in two teams, with the goal to develop two functionalities of the same application. My team, composed by 6 developers, dealt with the front office implementation,

that is described in the next section. The second team dealt with the back office implementation, the operations accessible by the system administrator with elevated privileges. The goal of the internship was not only to develop a web application but to introduce us in a modern web development project, following the DevOps philosophy. Shortly, *DevOps* [2] indicates best practise to combine development and operational operations, with the goal to decrease the development life cycle, empathising the automation. The company has well-defined development tools that are used in every project: to help the Agile team to coordinate and manage user stories was used the Jira dashboard. GitLab is used as code repository and software development platform, that suits with DevOps projects. Due to a partnership with Microsoft, the cloud environment and its related technologies chosen is Microsoft Azure.

1.2 The project

The goal of the project was to develop a web application used manly by the HR and the BM in order to facilitate the hiring process, careers follow-up and client management. The application was developed in a multicultural team, following Agile SCRUM methodology. Unfortunately, not much detailed information can be provided on this as the project is an internal project. Peculiarity of the web app is that this will be used in all Alten's department in France, so an important aspect to took in consideration was the multi-tenancy management. Since the application is deployed as a single cloud-native Software as a service solution, each BU must be enclosed in its own tenant, the accesses must be controlled. For example, a BM that has this role in Paris' tenant, cannot access data of consultants that work for SA's tenant, even if data are stored in the same database. To manage each tenant, a new role is introduced, the tenant administrator, which is able to handle entities inside the tenant. Since several users can access the application, its functionalities act differently basing on the role of the user logged in.

Let's analyse more in deep the requirements of the main functionalities of the app:

- *Candidates Management:* A candidate is a future consultant in process of being recruited, it is an entity with several fields like general information, professional information, recruitment and interviews information. The main requirements are the possibility to display the candidate in 4 ways: as a list, as a recap card with all important information, a form to create one and a summary with the interviews planned.

- *Consultant Management:* A consultant is a current or previous employee of Alten, it is an entity with several fields like general information, contact information and Alten related information. The main requirements are the possibility to display the consultant in 4 ways: as a list of current consultant, a list of ex consultant, as a recap card with all important information, a form to create or edit a consultant.

- *Client's needs Management:* Business Managers will manage their costumers, so they need the possibility to create or edit them. Since their information are sensible, the access to the data is limited. To each customer are associated one or more needs, that correspond to the consultant they are looking for.

One particular request was to have a dashboard (fig 1.1) as the main page of the app. Once the user login is redirect to this page, where are shown some widgets basing on the role of the user logged. These widgets provide an overview of the most valuable information for the users and facilitate the access to the features of the app. Other requirements for the application were to design an application that follows the Europe's GDPR rules, accessible, fun to use, responsive and in particular with high performance. This last requirement was the most important, why? Because the application was build from scratch but already exists a first version of the application; it was developed many years ago with deprecated and old technologies, that cause the application to be very slowly and with very low performance. Also, the UX/UI is considered

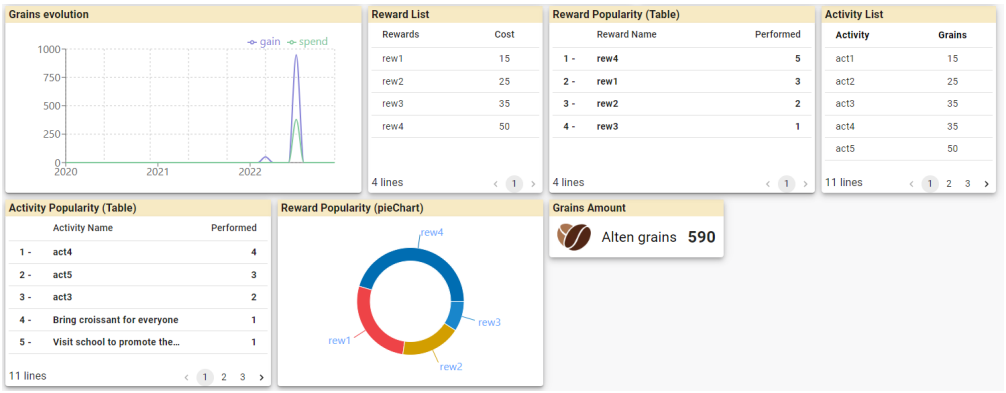

Figure 1.1: Some widgets of the dashboard

very boring and not well designed, so this is another point that we carefully took into consideration during the development. This explain why the company choose to restyle and build from scratch the second version of the product. Since the core activity of the company is to hire consultant and manage client needs, having a software that is not frustrating to use but a joy to use and with very fast performance, can only have positive effects for the company. The product thus aims to fill a need of the company, improving business processes, facilitating the work of the employees.

Chapter 2

Project structure

2.1 Microservice Architecture

Before talking about the architecture of system, I would like to introduce an important concept that was very useful for us to choose the technologies used in the project and to facilitate our works when we started implementing a new functionality: the Proof of concepts. POC [3] in general could be defined as a small realization of an idea to demonstrate its feasibility and could be used as a prototype of a solution. In our case, POC are small projects where are applied new technologies or more technologies are combined. This allows, for the first case, to start learning a new technology not in a client project and to share it with colleagues in order that they can have a starting point. For the second case, POC can show the feasibility of the combination of two or more technology (e.g. see how is possible to communicate a certain database to a back end) to determine if worth it to use together these technologies or not. I manly wrote POCs to choose the best technology for the front end. We started with the idea to use the framework Svelte, a new approach to build user interface. Since nobody knew that, was useful to see how it worked combined also with a back end. In particular, I wrote a POC for an authentication system in Svelte, then I wrote the same functionality in React. At the end, we choose React because Svelte is not well documented and there were some team members that already did some work in React. Another example was

one POC that was provided by our supervisor, that showed how to connect two Nestjs microservices through gRPC protocol. This POC helped us a lot to learn faster how gRPC works and how to apply it to the project.

The architecture of the system follows the **microservice** software design pattern. This approach enables to define an application as a suite of small services, each one in charge of implementing a functionality and communicating each other via lightweight protocols (e.g. gRPC). Each microservice must use the right tools for executing their job, it's not necessary that they all use same technologies but the ones best suited with the task to be performed. Microservices are loosely coupled: they know each other at run time and not at design time, implying a late binding, so the exchange of data is established at run time. Other properties are the autonomy, each service is responsible only of its data and functionalities, and push and pull behaviour, data exchange could be started by a service consumer that requests data or by a producer that publish messages that are received by subscribers (a message broker is needed). Another approach could be using a monolithic architecture, but although the advantages of a single code base and a central ops team, this approach is conditioned by a difficult in scalability, lack of agility and difficulties in maintain and evolve the architecture. Microservices instead has the advantages to facilitate the scaling of each individual micro-service, to facilitate the maintenance and to be more flexible and modular compared to the monolithic.

2.2 C4 model

Alten's business is not based on the product development but on the client satisfaction. Client's needs could be written or said verbally. In the latter case, the business analyst determines what is needed to do in the project, then the Product Owner defines requirements writing the CDC, document with specifications and requirements. In the former case, CDC is already provided by the client. CDC is the starting document for the software architect to define the architecture of the system. In our team, me with other two team members were in charge

of defining that structure. There are several models that could be used, like the UML, but the company suggested us to use the **C4 model**. The C4 model [4] was introduced not only to improve the graphical representation of the structure, having a more clear and readable one, but also to better communicate it to the client. As the name suggests, it's composed by 4 diagrams:

1. Context, the first level, it explains the app context and how it interacts with stakeholders and external systems. Is the one that is presented to the client, so does not contain technical details but highlights the main macroservices of the application and how they communicate with the stakeholders and between each others.

2. Containers, application design, it represents the runnable/deployable units and how they inter operate. An example of container is a database or the back end side of a microservice. Inside a macroservice there is at least one container. This level is important for the client, for the PO (that can start do the sizing) and for who has to do the setup of the infrastructure, generally the architect himself.

3. Components, used generally to do the sizing of the app, represent for each container its components. For example, could be the representation of controllers and services of a back end container.

4. Code, that is the representation of the structure of the code of a single component. Could be generated automatically by some IDE, an example is an UML class diagrams. Generally, the definition of this level is not done by the architect but is left to the developers, because they can organise the code structure as they see fit and as they usually do.

This model allows a sort of zoom in and out in the structure in order to represent it at different levels of details and to show to the reader the information he is interested on. Our supervisor gave us some guidelines and suggestions to prepare the model:

- Add title on each page and documents.

- Be consistent with names.
- Add sticky notes possibly.
- Do not use acronyms.
- Write directional lines and write the purpose of each connection
- Use icons, be explicit and use a legend.

In the figure 2.1 is showed the system context of our application. Stakeholders are identified by the human icon, the ones that most

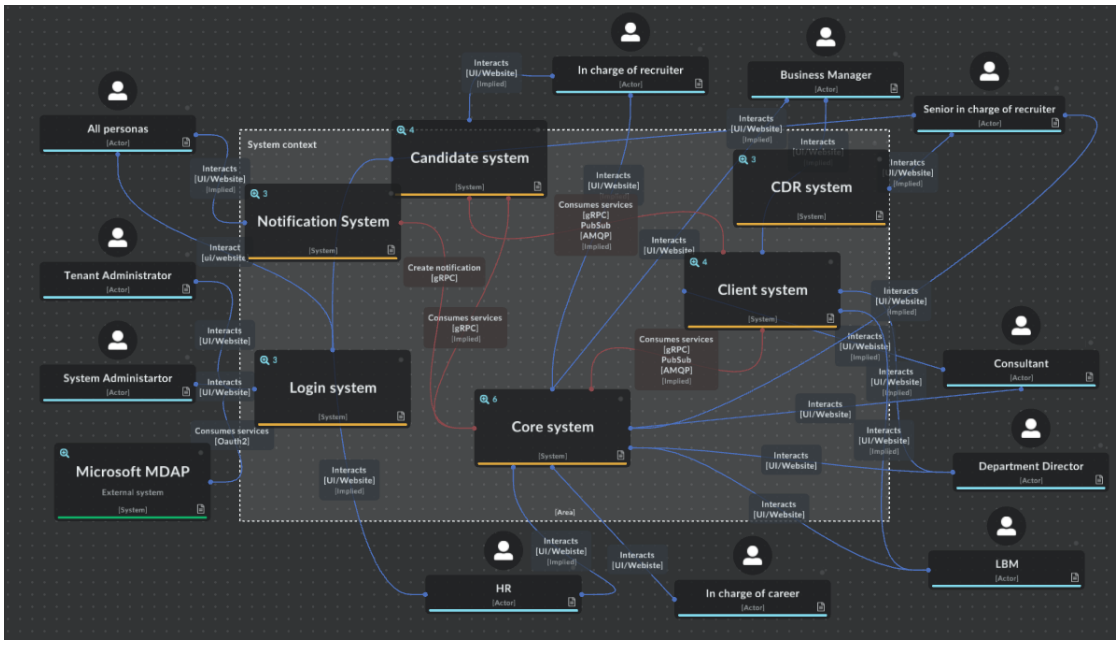

Figure 2.1: Context level

interacts with the application are already described in section 1.1, like HR and BM. Of course, they can interact with the system through the UI of the web site. In green is represented an external system, the Microsoft MDAP, that allows the user to authenticate, more details are given in section 3.1. One of the most relevant internal systems is the login system, every user interact with it and and in turn it interacts with the external system.

Since a requirement was the possibility to receive real time notifications when some particular actions are performed, we decided to put a microservice only in charge to deal with notifications, the Notification system. In the ideal implementation, it is a subscriber of an asynchronous protocol, waiting for messages published to a message broker by one of the other microservices that act as publisher. Since this solution is expensive and required a lot of time, to simplify the process, we decided to send notifications using a REST API like, where a microservice send a request to the notification system. More details about are given in chapter 3.5, where is described the protocol used for this implementation, the gRPC. 'Core system' is the microservice with more responsibilities, except for client and candidate managements that have an own system, here there are incorporated most of the functionalities of the app. As already said, interactions between microservices happen using gPRC protocol. Internally, each microservice has more than one components, that could be back ends, front end and a database container. Interactions with database is made using http protocol with TCP at the transport layer. Communication between front end and back end happens via REST API.

Let's now discuss the chosen technologies for each container: database is a non relational one, cosmosDB. The choice of using a non relational database is due to best performances compared to a sql database, the more flexibility and scalability. CosmosDB particularly because the company has a partnership with Microsoft, the owner of Cosmos, and because we deployed our application on azure cloud. CosmosDB provides a library to query the database using an sql syntax, that helped us a lot. More details about are given at section 3.3. For the front end's technology the choice fell on React. This choice was not easy because we wanted to use Svelte, a new framework, with which we also wrote some POCs. What led us to choose React was the fact that it is more well documented and affirmed compared to Svelte, which is a younger framework, and also the fact that two out of four front end developers already knew this framework, so this helped us to be faster in the development. Regarding the structure of the front end, to make the work easier, we used a monolithic structure, all the code in the same

repository git and all the GUI defined in a single project. Is important to underline the choice of using Typescript instead of Javascript. It could seems that they are similar but there are a lot of advantages in using the first. First of all, Javascript code is fully compatible with Typescript, that points out compilation errors at development-time. As the name suggests, typescript is typed, so the developers have fully controls on type and there is a static type checking, helping a lot for the code maintenance. Typescript was used also for the back end, together with NestJs framework [5]. This framework was new to me, we chosen it after the suggestion of our supervisor. It's lightweight, simple and open source and allows to build efficient and scalable server-side applications. It's fully compatible with CosmosDB, is composed by three main components:

- Controller: component capable to handle requests, is the file where API REST are defined. It's responsible of managing errors and return the appropriate response.
- Service: The business logic of the app. Provided functions that are invoked by the controllers and interact with the database. Can be injected as dependency.
- Module: encapsulate set of capabilities, like services and controllers, or db dependency. Each app has a root module, in our case for example we have a module for each microservice.

Others NestJs advantages, that can help a software architect to chose this technology, are: 1) it is a customisable framework, allowing custom annotations. 2) Provides a lot of functionalities, like Guards. 3) Has a documentation very extensive and is widespread in developers.

Before moving to the next section, I would like to briefly describe the *customer journey* because was part of the project even though I did not work on it. It's a document produced by other two team members that dealt with the functional parts; it could be defined as the path that the client of the application will follow while using the application. Of course, since the are more roles, for each of them must be defined a customer journey, in fig 2.2 the one for the BM role. It's defined

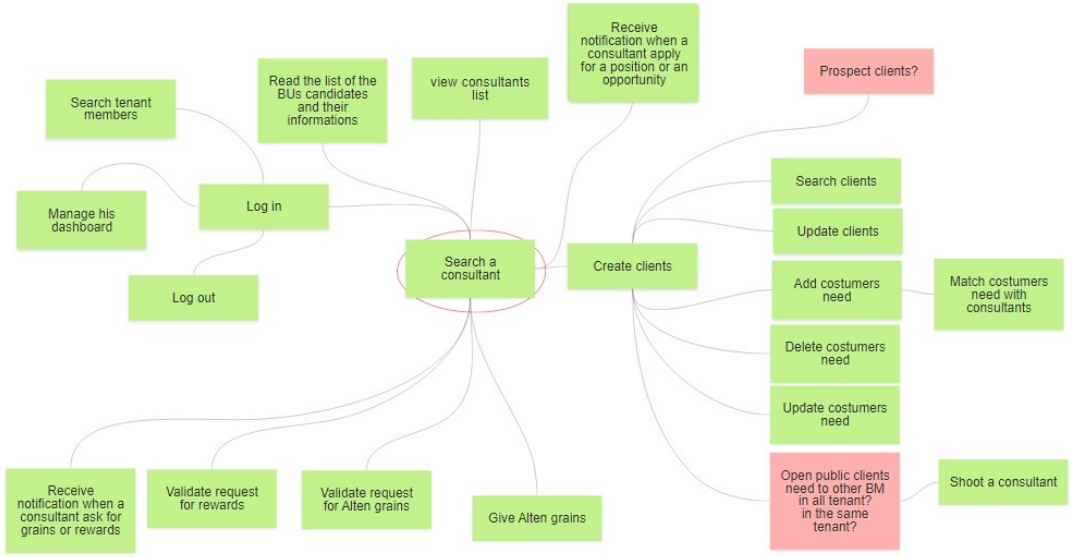

Figure 2.2: BM's customer journey

starting from the main functionality of the application that the client will use, which is circled in the diagram, then starting from that point all possible interactions and functionalities are covered. If there are some doubt about functionalities, in red rectangles is possible to add questions that the client can answer during the presentation of the document. The main goal of the customer journey is to document how the customer interacts with the application at each step during the use, in order to better understand it and possibly improve the business.

2.3 Kubernetes

In this section and in the sections 2.4 and 2.5, are explained how and which tools we used to deploy our application. First, let's simply define what does it means deploy an application: **deployment** is the process that allows to make the software available and usable by the end user. Behind this process, there are a lot of procedures and operations to perform [6], like: configure https port and virtual hosts to make the app reachable from the external through http, install the application

on the application servers, configure a firewall. The set of computer systems where the application is deployed is called *environment*. In our project two kind of environments where defined: development and production. The first is used by developers to test the application after deployed and it is where changes are developed (like if the application is running on the developer's computer). The production environment is the one used by the client to see how the application works and possibly perform user friendly test.

Nowadays, there is a new concept that allows and facilitate the deployment of an application (and other services) offering several services: **cloud computing**. Cloud computing is used to define the process of hosting and delivery services via internet. Some advantages are[7]:

- The user does not need to directly manage resources, so even a user not expert is able to use it. The goal of cloud computing is in fact to allow the user to take all the benefits of it without having an in-depth knowledge.
- Very flexible, resources could be added, removed or extended without any problems.
- Cost are reduced.
- Maintenance is easier, since the provider of the cloud maintains the cloud infrastructure, without manage hardware.
- Service providers care about the performances.
- Scalability: if for example we want to increase the power, all you have to do is change the settings.
- Availability and security reinforced, thanks to the provider's work.
- Since its main feature is the virtualization, i.e. create a virtual version of something, a virtualised resource computer can be moved on another platform without any problems.

There are three possible cloud systems used for deployments:

- Private cloud: cloud private to a single organization that is managed internally, so requires experts to manage it.

- Public cloud: there is a third part company (like Microsoft) that rents the hardware to the users. Services are provided over the network and they are shared between several users. Compared to the previous one is less secure and we don't know where hardware is located, while with the private cloud we have the fully control.

- Hybrid cloud: uses both, internal and external can communicate and are used to separate what must be inside the company from what is not needed to be enclosed.

In our project we used a kind of system public: *Azure Cloud*. As already said, Alten has a partnership with Azure, so its cloud provider is used. It remains to describe the several possible models of cloud computing[8], they are 3:

- IaaS: delivery of resources and computer infrastructure. Network equipment and servers are already provided. Generally used by the lead architect.

- PaaS: Delivery of tools used for the project development. Here the OS is hidden. Generally used by developers.

- Saas: Is an already developed application, like Dropbox, so used by the end user.

For the infrastructure of our project we used an IaaS model, illustrated on fig 2.3. Servers and Networking were already provided, we did care about OS, Middleware, Runtime and of course app development and data management. Runtime is the software that allows to execute and/or compile the code, like the the Java Virtual Machine in Java. In out project we used **Kubernetes** as runtime. Before discussing about kubernetes and see how we applied it to the project, it is important to introduce two concepts that are interrelated and are closely connected to kubernetes: Container and Docker.

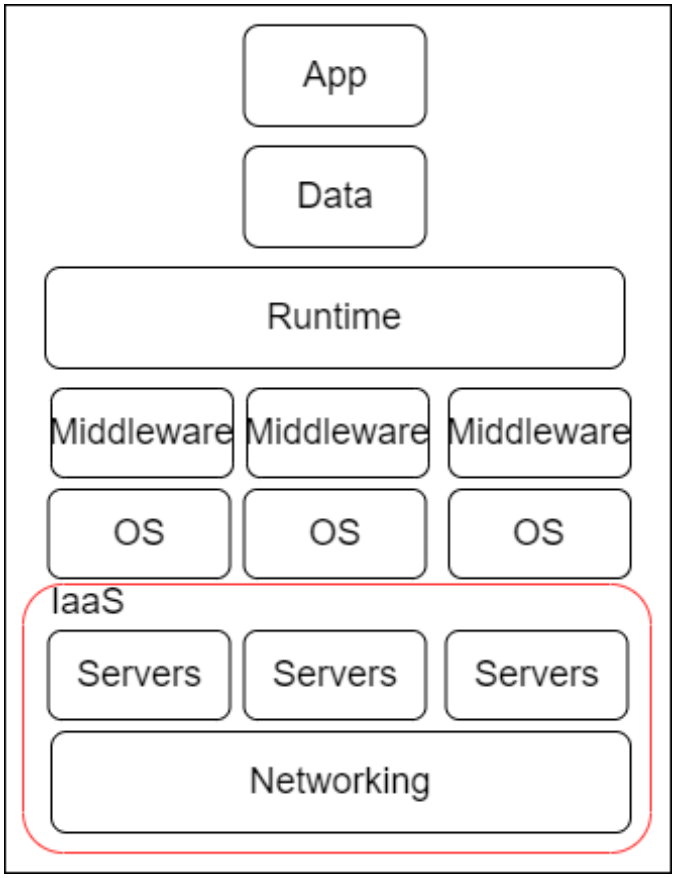

Figure 2.3: Cloud model

A *container* is a virtualization at the level of the operating system, the kernel allows the isolation of user space instances at the application layer. It could be seen like an unit of software containing code and dependencies that runs the application quickly and reliably [9]. Differently from an Operating system, they cannot see all resources of the computer but only resources assigned to them. *Docker* instead is an implementation of a container running on the Docker engine, it could be seen as a PaaS. Thanks to docker, container are more portable, flexible and lightweight because they share the kernel of the computer's OS, container are also more secure because they are isolated. From the developers point of view, using docker container allows to deploy, move, or replicate containers faster and easily if compared to a Virtual

Machine. In order to have a docker container we must build and run a docker image, assembled in the .dockerfile, a file where are defined the commands to create the image. To build the image is used the command *build*, that receives as parameter the path of the dockerfile to build. To run the container is used the command *run*, that can receive as parameter the port that the container has possibly to expose, the name of the container and environmental variables used inside the docker file, in addition to the name of the image to be run that is mandatory. In the following, an example of certain commands in the docker file used to assemble the images of the back end microservices. Note that thanks to the usage of docker environmental variables, it was possible to write a single dockerfile for all the containers:

```
1  FROM node:16-alpine as builder
2  # Which Microservice to build
3  ARG MICROSERVICE
4
5  # Microservice dist
6  COPY ${MICROSERVICE}/src /app/${MICROSERVICE}/src
7
8  RUN yarn install
9  RUN yarn build:${MICROSERVICE}
10
11 CMD ["node", "dist/main"]
```

Is possible to create a docker image from an existing one, like in our case where in the FROM we specified the image to start, alpine, that is a Linux distribution. RUN allows to execute command on the terminal, in our case we install dependencies and build the project. Thanks to the usage of a docker file is possible to define some file that must be not copied (like a gitignore) when executing the command COPY, that copies the content of the folder specified inside the folder specified of the container. Finally, CMD command is used to provide a default command for executing the container, in our case is the command's node used to run the mail file.

In a microservice architecture, a microservice could be seen as an application, so it could be implemented by a container, allowing a fast

starting with low overhead and an easier composition and replacement. So in a microservice architecture we could have several containers to manage and orchestrate, there are three ways to do that:

- Manually, for example via SSH into machine and using only docker. Approach very simple and easy to understand but not scalable and against the concept of automatization of the DevOps philosophy.

- Using scripts, that facilitate the integration with existing environments but is still not scalable and requires manual scheduling.

- Using an orchestrator, that does not requires human intervention and match containers to machines, grouping systems together to form clusters. Even if is difficult to understand and requires overhead, this approach allows to automatize and to be more scalable.

In our project we used this last approach, using particularly the **Kubernetes** orchestrator. Generally, an orchestrator has the goals to manage the complete lifecycle of a containers [10], scheduling them, handling failures and manage replication. Other appreciate features are the possibility to have a built in load balancer, cluster that auto scale them self and a provisioning storage. Kubernetes [11] is a term coming from the Greek language and it means "ship pilot", underlining the fact that we can think kubernetes as the pilot on a ship of container. Could be also referred as k8s, since between the initial and the final letter there are 8 characters. It was designed taking as idea the Google's Borg: is very rapid in adoption, modular by design and has a large scalability over thousands of machines. Its supported features are [11]:

- Automatic bin packing: basing on the availability is able to determine the resources to assign to the scheduled containers

- Horizontal scaling: scalability of applications can be done manually or automatically, where the latter case is done basing on the CPU and on the custom metrics utilization.

- Load balancing: Each container receives an IP address, then k8s assigns a single DNS name to a set of container to aid load balancing of requests inside the set of containers.

- Self-healing: In case a node fails, it is automatically replaced and rescheduled. Furthermore, a traffic is prevented from being routed to an unresponsive container.

- Storage: is able to mount automatically storage solutions like external cloud providers to the containers.

- Both version of IP, v4 and v6, are fully supported.

- Secret management: Sensitive data, like the key to sign a JWT token, are managed separately from the container image, in order to avoid a rebuild of that.

Note that all the appreciated feature of an orchestrator are all in the just aforementioned features of kubernetes. In kubernetes, a group of one or more containers, together with disk volumes, are called *pods*, the minimum elements in k8s. These containers are fully in relation each other, sharing a 'namespace'. A pod is executed on a single node since elements inside of it are highly coupled, allowing the scheduler to avoid to start them on different servers. Some pods may require some privileges, like access the bare OS; is possible to create a privileged pod setting the permission in the "deploy" request of the pod. Pod could be defined like a general kubernetes resource, starting from a defined template containing the following information:

- apiVersion and kind, that specify the resource and define uniquely an object. An example of a possible kind value is 'Pod'.

- metadata, info about the object, there could be of two type: labels, information useful for the kubernetes cluster and annotations, information used to enrich the basic info. An example of a label that for convention in used in Alten is: 'app.alten-dcx.dev/<name of the project>'.

- spec, description of the goal of the object provided by the programmer, specifying also the expected state.
- status, current state of the application.

Let's now see the main k8s resources of our project and see also their definition. *Deployment* is the resource that is in charge of manage the life cycle of a Pod and that create object of type 'ReplicaSet'. ReplicaSet is an object that implements the 'cattle pattern': if a server goes down, it is destroyed and a new replica is created, allowing to not affect the whole system in case of a failure of a single replica. When a new version of deployment is provided, a new ReplicaSet is created. Following, I will show step by step the deployment resource used for the template of our front end, adding comments:

```
apiVersion: apps/v1
kind: Deployment
```

Simply specify the version and the kind of the resource. I will skip the metadata, that I will discuss later, moving directly to the spec attribute

```
selector:
  matchLabels:
      {{- include "chart.selector" . | nindent 6 }}
  spec:
    containers:
      - imagePullPolicy: Always
        env:
          - name: TZ
            valueFrom:
              configMapKeyRef:
                name: {{ include "chart.name" . }}
                key: APPLICATION_TIMEZONE
        image: {{.Values.image.name}}:{{.Values.image.tag}}
        name: {{ include "chart.name" . }}
        ports:
            - containerPort: 80
    restartPolicy: Always
```

The thing that may immediately jump out is the values passed to some attributes, that are not 'hard coded' but passed inside brackets. This is a functionality of *helm*, a tool that will be described later. Inside 'matchLabels', is specified the pod to manage; Inside the spec are written some pod specifications: 1) the restart policy, i.e. when to restart the container 2) the specifications of the containers, that are: the port to expose; the image pull policy, i.e. when pull the image of the pod (always in this case); the name of the environmental variables received from the ConfigMap, another resource that will be described. In the example is not shown, but through the 'replicas' attribute is also possible to specify the number of replicas we want in order to have more than one instance in execution. After defining the deployment resource, is important to introduce the resource that acts as a load balancer and manages the traffic in ingress: *service*. It exposes the application in a set of running pods, its implementation is very simple, following the one for the front end template:

```
1 apiVersion: v1
2 kind: Service
3 spec:
4   ports:
5     - name: http
6       port: 80
7       targetPort: 80
8       protocol: http
```

Also here, metadata are omitted. It specifies the port through which be reached and the protocol to use, giving a name. In our case we gave the name of the protocol with which communicate with. If we want to have a load balance internally (reachable only internally), we must add the attribute 'type' with the value 'ClusterIP'. Note that, even if we did not do that, there is a resource that allows to limits the traffic between pods, the 'NameSpace', because by default there is no firewall between them.

It's now the moment to describe the resources *ConfigMap* and *Secret*. ConfigMap is a sort of .env from where environmental variables are

inserted. Then, as already shown for the deployment resource, these values can be injected into pods. Secret has the same goal, but as the name suggests, is used to hide some values to the developers like passwords or keys, encoding them in base64. Following an example of a configMap for the front end template:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "chart.name" . }}
data:
  APPLICATION_ENVIRONMENT: {{ .Values.global.env }}
  APPLICATION_TIMEZONE: {{ .Values.global.timezone }}
```

Last resource described is the *ingress*, it allows to a service to be reached from the external through an url. Following, the ingress for the front end template:

```
kind: Ingress
apiVersion: networking.k8s.io/v1
metadata:
  annotations:
    kubernetes.io/ingress.class: traefik
    traefik.ingress.kubernetes.io/router.entrypoints: websec
    traefik.ingress.kubernetes.io/router.tls: 'true'
    traefik.ingress.kubernetes.io/router.tls.certresolver: le
spec:
  rules:
  - host: {{ .Values.ingress.host }}
    http:
      paths:
      - backend:
          service:
            name: {{ include "chart.name" . }}
            port:
              name: http
        pathType: Prefix
        path: /
```

Inside spec is defined a load balancer, in its service attribute is indicated the corresponding service to expose, in our case, as seen above while describing it, is specified the http service. Host is the url, that is configured through helm. Under paths are listed the services, back end redefines the references service. In the annotations attributes, are specified some options to controls the traffic routing.

2.4 Helm

In the previous section is shown how templates were created, what came up is that the manifest file contains some configurable variables. **Helm** is a package manager for Kubernetes, used to make templates configurable. It allows to create, configure and deploy applications and service to k8s clusters [12]. We can say that helm is for kubernetes what apt is for Ubuntu. Applications are packaged and structured into *charts*. A chart contains information about versioning and type of the application. Following an example:

```
1 apiVersion: v2
2 name: bo-frontend
3 description: A Helm chart for front end
4 # A chart can be either an 'application' or a 'library' chart
5 type: application
6
7 # This is the chart version. This version number should be
     incremented each time you make changes
8 version: 1.0.2
9
10 # This is the version number of the application being
     deployed.
11 appVersion: "1.0.0"
```

So, Helm charts are collection of files describing k8s resources. In fig 2.4 is illustrated the directory's structure of an helm chart used in our project.

In addition to the chart file, there is a "values" file, where are listed all the variables that will be injected for the configuration of templates.

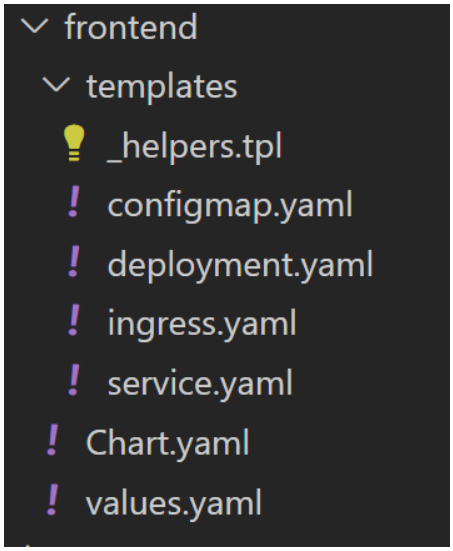

Figure 2.4: Helm's folder structure

We have then the templates folder, where are defined k8s' templates. Inside this folder, there is the file *helpers*, where are described data structure that can be used inside the manifest using the keyword *include*. Following, the content of the helpers file of the front end chart. After, is illustrated how the 'chart.name' structure is invoked inside the matchLabels attribute of the deployment resource:

```
1  {{- define "chart.name" -}}
2  frontend
3  {{- end }}
4
5  {{- define "chart.labels" -}}
6  app.alten-dcx.dev/name: frontend
7  app.alten-dcx.dev/project: mia2
8  {{- end }}
9
10 {{- define "chart.selector" -}}
11 app.alten-dcx.dev/service: frontend
12 {{- end }}
```

```
matchLabels:
      {{- include "chart.selector" . | nindent 6 }}
```

Helm facilitate to manage k8s resources, facilitating the maintenance of the kubernetes' manifests packaging information into the aforementioned charts. Other reason to use Helm is that it guarantee repeatability and consistency [13].

Note that kubernetes and helm were used only for deployment on the cloud. In order to run and test the application in locally, is enough to use *docker compose* without the two technologies just mentioned. For each functionality like front end, back end or the database itself, we have a docker image that could be run; we need a tool that allows to define and run them all in once, and docker compose is what we need [14]. Here an example of the docker compose file to execute our back end monorepo, for simplicity are reported only two services:

```
version: '3.9'
services:
  # DATABASE EMULATOR
  cosmosdb:
    image: 'mcr.microsoft.com/azure-cosmos-emulator'
    ports:
      - '${COSMOS_PORT}:8081'
    environment:
      AZURE_COSMOS_EMULATOR_ENABLE_DATA_PERSISTENCE: 'false'

  login-backend:
    ports:
      - '5000:5000'
    environment:
      - TZ=${APPLICATION_TIMEZONE}
    build:
      context: .
      args:
        MICROSERVICE: login-backend
    restart: unless-stopped
    depends_on:
      - cosmosdb
```

The first service specified allows to run the database; inside are specified the ports to expose, while environment is used to pass environmental variables to the container. For the database, the image to execute is specified through the image attribute, while for the second service, the login back end, a specific build section is defined. Inside this build section is specified the path of the image to build and run and possible arguments to pass. In our case, we put the dot because the docker compose is inside the same folder of the dockerfile and, as already said, the dockerfile is unique for each back end, it's only necessary to pass as argument the name of the folder of the back end we want to build and execute. Finally, we can note that we can specify some dependencies for the container in order to not start it before the container specified in the "depends on" attribute starts.

2.5 Terraform

Terraform is an infrastructure as code tool that allows to define cloud resources in human-readable way, configuring files that could be versioned, reused or shared [15]. It is able to perform these operations thanks to a *provider*, exploiting the Application Programming Interfaces (API) of the cloud platforms. For example, we can use the provider to create and manage resources of Azure Cloud, IBM Cloud, Amazon Web Services and more and more. The core of Terraform in enclosed in three commands [15]:

- Init: Resources definition, check of provider and data upload.
- Plan: Is a sort of compilation, the output is an execution plan that illustrates the infrastructure that will be created and which resources will be destroyed or updated.
- Apply: It's the effectively connection to the cloud, the operations described in the execution plan are performed.

Note that the first command is executed once the resources are defined by the programmer, the other two can be inserted into a pipeline

(explained in the next chapter), like in our case: each time there is a push on the main branch, the 'infra pipeline' is triggered and the two commands are executed automatically in sequence. Together with these two, at the end, is executed another command part of Kubernetes, the 'rollout'. This is the command that replaces previous versions of the application with new versions of an application, recognizing if there are new images and in the affirmative case, creating them. Note that resources are recreated if there are changes in the 'version' field of the corresponding helm chart. Before illustrate how to create a resource and describe the main resources in our project, I will illustrate some other useful terraform files:

- *do not modify*: it contains basic configuration info necessary for terraform, for example here is the place where is specified the provider, Azure cloud in our case. Inside of it is also defined the resource group, that will be described later.

```
data "azurerm_resource_group" "main-rg" {
    name = var.rg-name
}
terraform {
    required_providers {
        azurerm = {
            source  = "hashicorp/azurerm"
            version = "~> 2.99"
        }
    }}

```

- *deployments YAML files*: These are template modules could be defined for each resource and thanks to is possible to pass values in the definition of the resource. In our project they are initialized by Helm thanks to its values file.

- *variables*: here are defined reusable variables with its attributes that are the type, the default value and a description.

Now, after having cited these files, we can describe our main resources. To have clear in mind what is a terraform resource, we can say that it is an infrastructure object, i.e. the elements showed in fig 2.3. The first resource that has been created is the *resource group*, a sort of package created manually where all the other resources are contained. Let's now defined the *storage* resource: we have a resource for each back end since each one works with a specific container. We have then a general resource, called "azurerm cosmosdb account", where general information like the location, resource name and consistency policy (more details in the CosmosDB's section) are provided. Following an example for the notification container:

```
resource "azurerm_cosmosdb_sql_container"
    "notification-container" {
    name = "Notification"
    resource_group_name = data.name
    account_name = azurerm_cosmosdb_account.name
    database_name = azurerm_cosmosdb_sql_database.name
    partition_key_path = "/id"
    }
```

Inside the resource are specified its name, the database account the container refers and other useful information to create the container. Note that to each terraform resource are attached two tags, one relative to the project's name and the other is an internal tag of Alten, that must be always written. Note also that for each resource me must specify a type and a name, in the example are respectively "azurerm cosmosdb sql container" and "notification-container". Let's now describe a resource that allowed us to store and manage docker containers pushed by the pipeline: the *container registry*.

```
resource "azurerm_container_registry" "acr" {
    resource_group_name = data.main-rg.name
    location = data.azurerm_resource_group.location
    name = "acrweudevmia2001"
    sku  = "Premium"
    tags = var.alten-tags
```

```
7     }
```

In the 'sku' attribute is specified the version, we selected the Premium. Into 'location' we specify where there are the resources, the Azure location.

Looking at fig 2.3, is still missing the creation of the resource related to the runtime, that as already said is kubernetes. So it's the moment to create a Kubernetes cluster; its creation is a little bit more complicated to the other resources seen so far, so I will show step by step the creation: First, let's start with basic info, specifying the resource group, location, name and the dns prefix, defined in another resource, that is the one used when the cluster is created.

```
resource_group_name = data.azurerm_resource_group.name
location = data.azurerm_resource_group.location
name = "aks-weu-dev-mia2-001"
kubernetes_version = "1.22.6"
node_resource_group = "rg-weu-dev-mia2_nodepool-001"
dns_prefix = "aks-weu-dev-mia2-001"
```

Now, a default pool must be created. In each node runs a docker runtime where several container can be executed:

```
default_node_pool {
    name                = "default"
    vm_size             = "Standard_B2ms"
    enable_auto_scaling = false
    node_count          = 1
    max_pods            = 120
    vnet_subnet_id      = azurerm_subnet.subnet.id
    tags = var.alten-tags
  }
```

120 is the maximum number of containers can run in a node while 1 is the initial value. In the 'vnet subnet id' is written the id of the subnet resource, that I will describe later. The auto scaling could be set through a boolean variable, false in our case. Last but not the least, in 'vm size' we have to specify which 'sku' to use for the Virtual Machines.

Let's see and describe two possible Azure Virtual machines we could choose, in order to complete the OS level of the cloud structure 2.3:

- Av series: the more stable and are more suitable in developing and testing environments.
- B series: the one used by us, we pay RAM memory and CPU we use. They are suitable for Web Servers, where the workload does not require full CPU performance.

Note that if we want to have multiple node pools we must use a Virtual Machine of kind "Scale Sets". Also, note that this choice is very very important, because once decided is not possible to come back to the decision.

There is still missing one last step: associate a Virtual Network to the runtime. Is needed to setup the components of the High Level Networking that in Azure are called Virtual Network (router and network controller), Subnet (Switch) and private endpoints (cables). I will illustrate just the first two. Virtual network is fundamental if we want resources communicate between them. It's the core of the private network, guaranteeing isolation, availability and scalability. The definition is very simple:

```
resource "azurerm_virtual_network" "vnet" {
    resource_group_name = data.azurerm_resource_group.name
    location = data.azurerm_resource_group.location
    name = "vnet-weu-dev-mia2-001"
    address_space = ["10.0.0.0/8"]
    tags = var.alten-tags
    }
```

Note that the name is not given randomly but following the convention: <resource name> - <location (west Europe)> - <deployment phase (dev, prod..)> - <project name> - <version>. The IP address space is given without following a specific criteria, just keeping in mind one constraint: do not use address starting with 172 because these are used by Docker. The resource just illustrated allows to manage a virtual

network including its subnets. The reason of having subnets inside the virtual network is to have more security and organization. Following, an example, noting that the address space is a range of the IP address assigned to the virtual network:

```
resource "azurerm_subnet" "subnet" {
  name = "kubernetes2"
  resource_group_name = data.azurerm_resource_group.name
  virtual_network_name = azurerm_virtual_network.vnet.name
  address_prefixes = ["10.42.0.0/16"]
  enforce_private_link_endpoint_network_policies = true
  enforce_private_link_service_network_policies  = true
  }
```

To conclude the chapter, I will describe an optimization we did during the project, using the *edge router*. In order to do not keep services on the board and expose them, is possible to expose a single edge router that receives all the requests. The possibles edge router specific for kubernetes are HAProxy, Ambassador or traefik, the one chosen by us. To add traefik on a kubernetes cluster we can use helm, using the *helm release* resource, we just need to ensure that the cluster has the permission to modify resources. Note that the helm release resource requires a TLS certificate; we can create and insert a fake one, for example using 'LetsEncrpyt'.

Chapter 3

Development technologies

3.1 Microsoft MSAL for the login system

After having defined the system architecture and the cloud infrastructure, we started with the development. The first functionality implemented was the **authentication system**, in order to enable users to log in the application. I will list the protocols and the tools used and then show how they work together; in fig 3.1 the flow of the login process, that will be described. First, is important to underline that the company has a partnership with Microsoft, so in all its applications the company uses the Microsoft Active Directory. AD [16] is a directory service used for authenticate and authorize all users and computer in a Windows domain network (Alten in our case). This system allows to enforce the security, checking if the user exists or not and determining also the privileges of the user. The protocol used for the authentication is **OAuth 2.0** [17], a standardized protocol that simplifies the process to get the authorized access to the protected resources via HTTP. Using OAuth 2.0, the application acts as a user-agent that redirects the user to the Microsoft Authorization server (Identity platform) to insert its company credentials and then get an access token where user info are

stored. Note that these info are taken from the Microsoft Active Directory. The access token is a JSON Web Token that will be saved locally (in our project into cookies) and will be included in every request's header sent to the server. JWT is composed by three parts:

- header: contains metadata about the token type.
- payload: contains user's useful data, like permissions or roles.
- signature: contains a signature of the parts above.

From the security point of view, in the payload must not be written sensible information because it is sent in clear, it's not encrypted. The signature guarantee data integrity; protection by sniffing attacks is provided by default only if sensible information are not written in the payload.

Now that we have introduced the main concepts, let's have a look on how they cooperate: When the user clicks on the login button, the

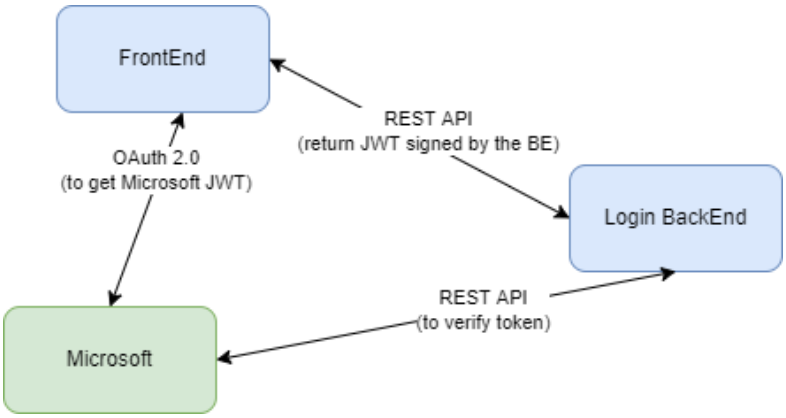

Figure 3.1: Login structure

Front End will start implement the Oauth2.0 protocol communication with Microsoft Active directory and the dialog in fig 3.2 is shown. If credentials are incorrect (email and password do not match any user in the AD), an error will be shown in the dialog. If the authentication succeed, Front End will get a JWT, the access token signed by Microsoft. Now, since the Microsoft's JWT does not contains useful information for

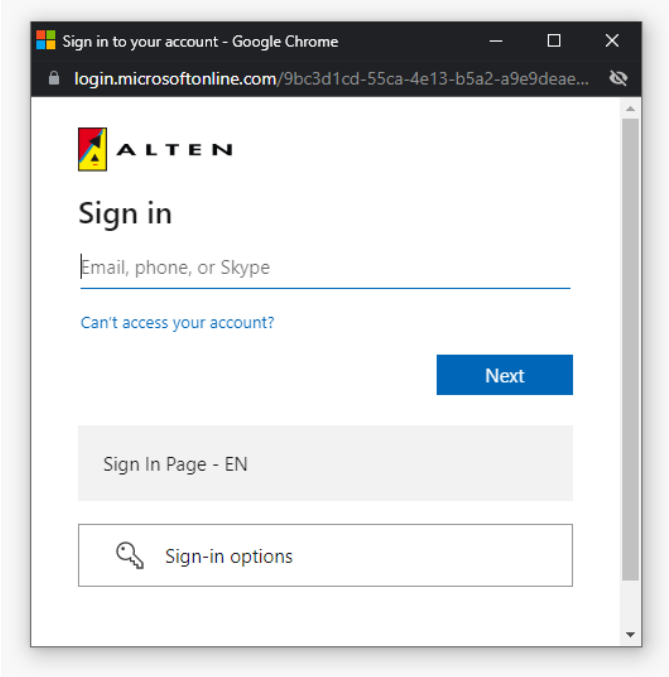

Figure 3.2: Login dialog

us and since the back ends, if we will use this token at each request, must ask Microsoft if the user is authorized or not, the received Microsoft's JWT is sent to the login back end. Login back end that will validate the token providing a new one, with the correct info. In particular, through a POST request, the front end sends a request to the login back end that, through a GET request sending the token received at the url graph microsoft url, is able to determine if the token is valid and if the user that requests to login is part of the active directory. This additional step was made to increase the security and to verify the token. Once Microsoft's JWT is validated (if not, an error 403 is sent), the back end checks, using the mail contained in the JWT, if the user already exists in the database or not. If it not exists, login back end will create the user assigning no role to it. If the user already exists, login back end will retrieve his info from the database. At the last step, the back end generate a new JWT with its signature and will add the token in the cookies of the response. The front end can get and include it in each request sent to the back ends. In fig 3.3 is a possible payload

of our token: There are user's mail, display name and most of all, the

PAYLOAD: DATA


```
{
  "email": "amedeo.sarpa@alten.com",
  "roles": {
    "CONSULTANT": [
      "SA"
    ]
  },
  "displayName": "Amedeo SARPA",
  "iat": 1660137862,
  "exp": 1660224262
}
```


Figure 3.3: JWT Payload

roles the user has and the list of tenants where he has that roles. This was a crucial step for the multi tenancy request of the project. The others two info are about the issue time and the expiration time.

In the front end, to implement the OAuth 2.0 communication, we used a library provided my Microsoft: *MSAL* [18]. The name stands for 'MicroSoft Authentication Library' and is the one that allows to communicate with Microsoft in order to acquire the token. It acts like a sort of wrapper, that allows to not directly use the OAuth 2.0 libraries and most of all can be used as an hook in React. The name of the hook is *useMsal*, from which the object *instance* could be taken. This object contains several useful methods; in particular, the method *acquireTokenSilent*, once invoked will shows in the application the dialog in fig 3.2. After the user is authenticated, in the code is returned the access token.

It only remains to describe how the token is checked at each request. We must remember that our architecture is a microservice one, the operations performed by the login back end are not known by the others back end as for the signature's key. So, even if the user is authenticated by the login back end, there is loosely coupling of information, each

back end must check authentication and authorization at each request. A possible solution could be to check the permission in each route of the controller, but this implies a lot of code replication for the same check. The solution used was to define the roles that can request an operation and put a *Guard* for each request. Guard is a class that determines if a given request will be handled by the controller or not, depending on some conditions. It is like a filter that make pass all authorized requests. Guard class is a functionality provides by NestJs frameworks; it implements the *CanActivate interface*, an interface with only one method with the same name of the interface. The 'CanActivate' method returns a Boolean indicating if the request is authorized or not, receiving as parameter the ExecutionContext, that was fundamental in our project because from there it could be extracted the JWT sent in the request header. We combined guards with the possibility to create personal decorators in NestJs. We created a personal decorator called 'RolesCheck', that receives as parameters the list of roles that are allowed to perform the operation requested. This decorator was added in all the controller's routes that required roles check. Inside the decorator, roles received are set as metadata that will be used by the 'AuthenticatedGuard', a guard defined by us. Inside that class, in the 'canActivate' method, from the token are extracted the roles and the return value is 'true' if at least one of the authorized roles (received as metadata) is present in the user's roles. Note that the tenant check is not implemented there but this check is performed by each service, if needed.

3.2 Pipelines

One of the DevOps principles is to automatize things, eliminating the need of manual operations during the life cycle of a software development process. Pipeline [19] are a powerful tool for DevOps developers, allowing to automatize processes utilized by software engineering team like run test, compile, build and deploy. This allows to be fast in the processes, save developer time since he no longer has to deal with these tasks and to avoid possible human errors improving the quality of the

code. For creating a DevOps pipeline there exists many tools like CircleCI, Azure DevOps Pipelines or GitLab CI/CD, the one we choose in our project. GitLab pipelines consists in: 1) *jobs*, the operation to execute. 2) *stages*, that indicates when to execute the job. In a stage there could be executed more than one job, in parallel, moving to the next stage only if all current jobs pass. Jobs are executed by GitLab runners, an application that could also be installed on a local machine. In fig 3.4 are illustrates the stages of one pipeline defined in

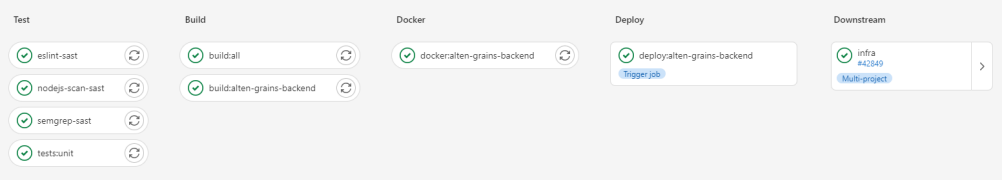

Figure 3.4: GitLab pipeline

our projects. The stages are:

- Test: executes tests and stop the pipeline in case not all test pass or the coverage is under the threshold defined in the Definition of Done.
- Build: that builds the project; for the back end, for example, it simply runs the 'build' command of yarn.
- Docker: builds docker images and push them to the azure repository.
- Deploy: this is a particular stage because it triggers the infra pipeline, which is in charge of deploying the project, executing in order the terraform commands 'plan', 'apply' and 'rollout'. Their execution result is summarized under the 'downstream' column.

The image also illustrates that we can have at each stage several jobs executing their task.

Until now we have discussed about pipelines and the stages defined in our project, but how to setup a GitLab pipeline? They are configured using a version-controlled YAML file, called *.gitlab-ci.yml*, within the root of the project. In that file can be defined, aside for jobs and stages,

other parameters like: when trigger pipeline, what to do if a stage pass or not. I will now show step by step the content of one *.gitlab-ci.yaml* file defined in our project, used to manage the pipeline of the cronjob scripts' repository:

```
stages:
  - test
  - compile
  - docker
```

stages allows to define stages of the pipeline, executed in order. In our case we have just 3 stage since the pipeline manages scripts to run in a cronjob, so is not needed to deploy. These stages are: 1) test, that executes tests and check coverage. 2) compile, that compiles the project. 3) docker, that builds and push in the cloud's images repository the images. Now we can define each step:

```
compile:
    image: node:16-alpine
    before_script:
        - yarn install
    stage: compile
    script:
        - yarn compile
    artifacts:
        paths:
            - src
```

Here, as suggested by the field stage, the compile stage is defined. *image* represents the docker image from which start to execute jobs, in this case the version 16 of node alpine. In *before script* are defined the scripts to be executed before the job starts; scripts contains the command executed by the job, both commands are ordinary shell commands, is like typing in the system terminal. In this case, first libraries and dependencies are installed, then the project is compiled. *artifacts* is very important if the output of the current stage will be used by the following stage. This because the files with which the pipeline works

are the ones present in the repository, so if some file are ignored, like in our case with the dependency values (node modules), the next stage, without using artifact, will not have that files and as a consequence will not work. So, artifact allows to overwrite the folders specified in the paths field with the one produced by the current stage. In this case, test stage will have in the paths folder the files produced by the compile stage. Test stage will be not shown because does not add more information compared to the compile one, the difference is that does not produce artifacts and the script that executes is the one that runs test. What is important to show is the docker stage, that has the peculiarity to be executed only if some conditions are verified. These conditions are called *rules*, and could be defined at the beginning of the file as follows:

```
.default_docker_rules:
    rules:
        - if: $CI_COMMIT_BRANCH != $CI_DEFAULT_BRANCH
        when: never
        - changes:
            - dockerfile
            - package.json
            - .gitlab-ci.yml
            - .gitignore
            - tsconfig.build.json
            - .dockerignore
            - tsconfig.json
        when: never
```

Rules are evaluated in order until we have a match, then the stage is executed or not basing on the *when* condition, that by default is on success. If there are no match, the pipeline is not executed. In our case, the first condition indicates that, if the branch on which we have committed is not the main branch, do not execute the stage because of the never conditions. Note that the name of the current branch and the default branch are GitLab variables that could be set manually in the variables section of the GitLab CI/CD pipeline. The second condition indicates that, even if we are in the main branch (because we skip the

first condition) but the changes of the current commit are only on the listed files, the pipeline must not be executed.

Once defined the rules, let's apply them to the docker stage. This stage was split in two because of the number of cronjob scripts we have, adding a tag after the name of the stage. Let's see the docker stage for the notification cronjob:

```
1  docker:notification-cronjob:
2  variables:
3      DOCKER_HOST: tcp://docker:2375
4      DOCKER_TLS_CERTDIR: ''
5      DOCKER_DRIVER: overlay2
6      NOTIFICATION_CRONJOB_IMAGE: ${REGISTRY_URL}/${CI_PROJECT_PATH}/
       notification-cronjob
7  stage: docker
8  image: docker:20
9  rules:
10     - !reference [.default_docker_rules, rules]
11     - changes:
12         - src/notification-cronjob/*
13
14 services:
15     - name: docker:20-dind
16     alias: docker
17     command: ['--tls=false']
18     script:
19       - docker build -t ${NOTIFICATION_CRONJOB_IMAGE}:latest -f
       dockerfile --build-arg FOLDER=notification-cronjob .
20       - echo "${REGISTRY_PASSWORD}" | docker login --password-stdin -u $
       {REGISTRY_USER} ${REGISTRY_URL}
21       - docker push ${NOTIFICATION_CRONJOB_IMAGE}:latest
```

If the job uses ENV variables, these could be defined under the *variables* field. Under *rules* section are listed the rules to decide if execute or not the stage. Together with the rules defined before, like default docker images rules, there is a third condition that indicates that if there are changes in the specified path, the stage must be executed only if the previous stages passed correctly. This because the default value of when condition, if not specified, is *on success*, that executes the job if there are no failures in the previous stages. The commands executed are the building of the image, then a print to log the operations and finally the push on the cloud's repository. Also here, the notification cronjob image variable is defined in the variables section of the GitLab

CI/CD web site. To conclude, *services* defines a docker instance that runs during the execution of the job. It is needed in order to execute docker commands like build and push.

These kind of pipelines defined by us where *basic pipelines*, the simplest ones. They are not the most efficient but the easiest to maintain. To increase the efficiency, GitLab proposes another kind of pipeline: *Directed Acyclic Graph Pipelines* [20]. They allows to define dependencies between jobs, allowing to run everything as fast as possible without waiting to conclude each stage step by step. The outcome of the pipeline could be checked directly on the main page of the repository git. There is a badge that indicates if the pipeline has passed (in green) or has failed (in red). Then, clicking on the badge, it redirects to the pipeline page where details are given and where is possible to restart the whole pipeline (or just a single stage).

3.3 CosmosDB

CosmosDB is the database technology chosen by the team. It is a *non relational database*, so before see in details aspects related to Cosmos, let's analyze the differences from a relational database and why we preferred a noSQL one. Relational database's data are organized in structure called table. If there are dependencies within data, it provides cascade operations that allows data consistency and, most of all, is possible to perform JOIN operations to merge tables with attributes in common. All these concepts are not valid for non relational databases. Data in a non relational database are not structured in tables but in *documents*, that belongs to a certain collection; join operation is removed; non relational database do not use SQL to query the database but custom query languages. What are the advantages? More scalability, flexibility and adaptability. Our decision was to use a non relational database because the data to be collected are not very structured and because there are no much relationships between data stored in different collections (not a lot of join operations). There are several noSQL technologies could be used, we decided to use CosmosDB [21] because:

- Is owned by Microsoft; thanks to the partnership with the company we had a reductions in the costs and more functionalities not present in the basic version.

- There is a very powerful library, *azure cosmos*, available in typescript, that facilitates operations. It allows to query the database using SQL syntax, so allowing us to have a sort of hybrid approach.

- It has very high performance in terms of availability and operations execution's time, that was a requirement for the project.

In CosmosDB documents can be written only in JSON data format and all together can be stored in containers. In our case, each back end microservice uses one container. In each container could be stored heterogeneous documents, not all belonging to the same type. Good performance are achieved because each field of the document is indexed; data integrity is ensured thanks to an unique key constraint that could be assigned automatically or manually also. The internal data model not only provides SQL API (used by the typescripts' library aforementioned) but offers also others five API to guarantee compatibility with MongoDB, Gremlin, Cassandra, Azure table Storage and etcd [22]. To give an idea about performances, latency for all kind of requests is guarantee to be below than 10ms, reserving the needed resources to guarantee the constraint. Another feature of CosmosDB is that can be configured in all the Microsoft Azure regions, dynamically adding or removing a region. By the way, we did not exploited this functionality because our application will be used only in France. However, within the region we can have several databases distributed. One problem not mentioned for CosmosDB, but in general for all non relation database, is that is impossible to guarantee the coexistence of the following three properties: 1) consistency, all distributed database contains the same data. 2) availability, even if one database has a failure others one continue their job. 3) partition, the system continue to operate even if some data are lost. As there is that problem, CosmosDB allows to configure the data consistency among five different different levels. Some limitations of cosmosDB are that it only supports JSON data

format and lack of support for representation of date and time. Another limitation is that, even if could be queried through SQL, not all functionality, like GROUP BY, can be used. This last limitation could be overcome using *User-defined functions* [23]. They allows to extends the SQL syntax adding complex business logic in the queries. Once defined, they could be used by the developers as they are SQL keywords. Following, an example of a user defined function written by us, named 'GET MERGED VALUES', that is very simple and allows to retrieve in an array all the values of a document, a sort of mapping:

```
function mergedObjectValues(object) {
 return Object.values(object).reduce((a, b) => a.concat(b),
    []);
}
```

Note that they are written in Java code, and could be used in a 'whery' condition in this way:

```
AND udf.ARRAY_CONTAINS_SOME(udf.GET_MERGED_VALUES(c.roles),
    @tenants)
```

where 'ARRAY CONTAINS SOME' is another user defined functions that checks if at least one object of the array received as second parameter is present in the array received as first parameter.

From the application's code is possible to connect to the database using HTTP over TCP, using the aforementioned *@azure/cosmos* library. It provides the 'CosmosClient' class, thanks to is possible to open the connection just passing the endpoint and the key to access the database (proved to us by the company).

Using CosmosDB for all development steps, despite the partnership, can lead to increased costs for the company. For testing and development purposes, the best solution is to use the *CosmosDB emulator*. It allows to emulate in local an Azure CosmosDB instance, providing the same interface of the real one, accessible in local host at port 8085, without requiring an Azure subscription. In the next section, is shown how to combine the emulator with test container, in order to run the emulator

in a docker container for testing purposes. Cosmos Emulator data are not loss, but they could be migrated in a real Azure CosmosDB service using the Azure data migration tool. Following, some small differences from the emulator to the real service, that is important to underline:

- Emulator does not provide multi-region replication.
- The consistency levels configurable are not five like for the real service but less.
- Emulator is not scalable and only a limited number of containers can be added.

However, these limitations do not affected our development.

3.4 Jest and Supertest

In this section are described tools and technologies used to **test** the application. Let's first describe what does it means 'test an application' and which kind of test we performed.

Testing is a Validation and Verification technique, we want to test the reliability and the effectiveness of the system and its efficiency and correctness. Test is a dynamic technique that requires the execution of the application or of a single part; given a certain input, test detects if the correspondent output is the one expected. Tests must be exhaustive, to reach this property, several kind of test must be performed:

- Unit test: test the correctness of a single unit; two possible techniques to write an unit test are the white box and the black box (the former was used by us). If one module depends on another, the dependency could be *mocked*, in the sense that we assume that the module that creates the dependency is correct and returns a specific value.
- Integration test: test the correctness of the interactions between two or more module. Generally is performed after the unit test.

- System test: all the system is tested to check if it satisfies the requirements.
- Acceptance test: in an agile environment could be performed by the PO or in general by the end client, where he tests the system according to a given scenario.
- E2E test: specific for the software development cycle, similar to the system test, emulate a scenario and validates the system.

In our project we performed just the first two, the others will be performed by the next team of interns that will replace our team. One indicator that could be used to determine if test are exhaustive is the coverage: it indicates the percentage of code tested. In the project we decided to have a coverage of at least 90%.

Now, having a clear idea of what is test's purpose and what are the techniques, let's have a look into the technologies used in our project. Unit and integration test were performed by developers using *Jest* framework. It is a Javascript Testing framework that runs test in parallel; it facilitate mocking functions and is very well documented. It allows to define a test case using the keyword *describe*, that receives in input two arguments: the name of the use case scenario and the function where internally are executed the unit tests. Unit tests are introduced by the keyword *it* and the definition is similar to the describe since the two arguments are the name and the body of the test. Inside each test are made the assertions, using the keyword *expect*. For example, if we want to test an asynchronous function that resolves the integer value 3, the code is the following :

```
1 it('test example', async () => {
2   await expect(function()).resolves.toBe(3);
3 });
```

Is important to put the 'await' keyword before the test since we are testing a promise function. 'Resolves' means that the promise returns the value; if the promise rejects we must use *rejects*, addressing the error expected with the keyword toThrowError.

```
it('test example', async () => {
  await expect(function()).rejects.toThrowError(new Error());
});
```

One powerful feature of Jest is that it provides simple functions that can be executed after or before all tests, or after or before each tests, allowing to perform some operations of initialization or of cleaning (e.g. restore the database after the execution of the test case). Some of these useful functions are *afterEach*, *afterAll*, *beforeEach*, *beforeAll*; the latter, in particular, is used to create the testing module: a jest module where could be specified the services to inject or where could be opened the connection with the database. Mocking a function is very easy with Jest, of course we must have in our testing module the service on which the component to be tested depends. Let's for example test a certain controller that uses methods provided by a service class. First, we must initialize an instance of the service class, that we will call service. The Jest function spyOn allows to mock a method of the service class:

```
jest.spyOn(service, method).mockResolvedValue();
```

The type of the returned value depends on whether the method is asynchronous or not. If it is, as in our case, it can be mocked to resolve or reject a value. It's a good practise to clear all the mocked functions at the end of each test, exploiting the function 'beforeEach', using the Jest method clearMocks. About controllers, since they handle HTTP requests, to test their behaviour is needed an additional libray: **supertest** [24]. It allows to test API, specifying the http method and the URI we want to test, and specifying also the expected result, testing both message status and body of the response. Let's analyze the following code:

```
await request(app.getHttpServer).
    get('/data').
    set('Content-Type','application/json').
    expect(HttpStatus.OK).
```

```
5     expect([]);
```

Here we are testing that, if we invoke the '/data' API with a get method, the response is an empty array and then of course the status is 'ok' (200). Since our data format is JSON, this must be specified in the Content-Type header, as in line 3. If we need to send data, this is possible using the *send* method, that has as parameter the data to send. If it's necessary to include some data in cookies, we can use, as for the application context, the *set* method, were the first parameter is the string 'Cookie' and the second is the list of cookies in the form 'name=value'. The request method of supertest requires the http server to test. Is possible to have it using an 'INestApplication' instance, that provides the method *getHttpServer*. As already discussed before, the INestApplication instance could be initialised in the 'beforeAll' method, where first the module is defined and then the instance is created using the method createNestApplication. In the following a possible implementation:

```
1  let app: INestApplication;
2  beforeAll(async () => {
3  const fixture = await Test.createTestingModule({
4      imports: ...,
5      controllers: ....,
6      services: ...
7  })
8  .compile();
9
10 app = fixture.createNestApplication();
11 await app.init();
12 }
```

It remains to describe the last tool used: **testcontainer** [25]. This is a fundamental tool to emulate a database or something else running in a docker container. This was useful for the team to test services class using an emulated version of the database, CosmosDB Emulator. The container is initialised in the testing module; there are provided information about the id of the database, the name of the container, the name of the endpoint and the key of the cosmos database emulator

client.

3.5 gRPC protocol

gRPC is a protocol owned by Google that could be used for the communication between two microservices. In this section I will explain how it works and what are the advantages compared to REST API. gRPC was used for the communication between several microservices and the notification back end container, in order to request to send a notification. This protocol was used also for the interaction with the login microservice in order to create,read,update or delete users in its corresponding database container. There will be examples considering the first case, the one for sending notification. gRPC allows to perform these operations as if methods of the other services to invoke are in the same application of the one that consumes them. Client and server could be implemented using different technologies thanks to the proto file concept. gRPC [26] is based on the idea that a microservice provides a service with its corresponding methods, return values and method's parameters. The server, then, implements this interface running the gRPC server while the client has a *stub* that provides the methods of the server. The interface that the service implements is described in the **proto file**, a file with the .proto extension. Data structure defined in the proto are then serialized since gRPC is based on protocols buffers [27], an extensible mechanism for serializing data in a forward-compatible and backward-compatible way; it is similar to JSON but smaller and faster. Following an example of a proto file:

```
1 syntax = "proto3";
2 service NotificationSenderService {
3   rpc SendNotification (NotificationSenderRequest) returns (
    NotificationSenderResponse) {}
4 }
5 message NotificationSenderRequest {
6   string message = 1;
7   optional string notifiedUserId = 2 ;
8 }
```

```
9  message NotificationSenderResponse {
10   bool notificationSent = 1;
11 }
```

- At the first line is specified the syntax used, 'proto3'. In the official documentation are described all the possible data types the protocol buffers supports and also, how to compose an object with multiple fields.
- 'service' introduces the name of the service we want to define, in our case our service is used to send notifications.
- 'rpc' introduces the definition of methods using the syntax: <method name> (<argument>) returns (<return value>) .
- 'messages' represents the data structure. In this case, the arguments require the message to show in the notification and the notified user, that could be optional. In case we want to send a list, the keyword *repeated* precedes the type. The return value, instead, contains a Boolean indicating if the operation was performed correctly or not.

One main feature of gRPC is that, internally, it uses a built-in compiler, protoc. Thanks to this compiler, data defined in the proto file are transcribed automatically in code, targeting the chosen programming language. gRPC is supported by the TypeScript programming language and by NestJs, providing a package that could be installed via yarn or npm. To better understand how to implement a gRPC connection in

Figure 3.5: gRPC schema

NestJs, let's consider the sub system illustrated in fig 3.5, where the gRPC server is the notification back end while the client is the 'Core system back end', which internally has the hub.

In the main file of the Notification back end, the gRPC service must be initialized,

```
app.connectMicroservice({
    transport: Transport.GRPC,
    options: {
      package: 'notification',
      'proto-messages/dist/notification.proto',
      url: process.env.NOTIFICATION_GRPC_URL
});
```

selecting the gRPC transporter mechanism and specifying several parameters, that are: 1) the url of the service. 2) the package name. 3) the location of the proto file. Then, we move to the Notification controller. There, thanks to the *GrpcMethod* decorator, is possible to define the gRPC service method:

```
@GrpcMethod('NotificationSenderService')
async sendNotification(data: NotificationSenderRequest):
   Promise<NotificationServerResponse>{
  //Implementation
}
});
```

The decorator receives as parameter the name of the service; the name of the method is the same of the one defined in the proto file. The argument and the return file, that are defined as message in the proto file, here could be defined as interfaces. On the client side (core system back end in our case), we must obtain the Stub. One common technique is to use the *register* method of the 'ClientsModule' class, that binds a package of the proto file into an injected token.

```
ClientsModule.register([
  name: 'NOTIFICATION_GRPC_SERVICE',
```

```
3   transport: Transport.GRPC,
4   options: {
5     package: 'notification',
6     protoPath: 'proto-messages/dist/notification.proto',
7     url: process.env.NOTIFICATION_GRPC_URL
8   }
9 ])
```

It is dual to the initialization of the server: are specified the transport method, the proto package, the path of the proto file and the url of the server. Once registered, through the annotation *Inject()*, is possible to inject the client as an object with the proto file defined methods. It remains to underline one last important thing: the value that the method invoked returns through the gRPC client is an *observable* [28] value. For example, injecting the *client* object in the core system back end, invoking the method sendNotification, we will get the following instance: Observabe<NotificationSenderResponse>. Observable objects allows to handle events [29], registering and then processing the value get. Inside the observable object three different callbacks could be defined:

- A success handler function, that can work with the value returned.
- An error handler function, that receives the error and manages it.
- A completion handler function, that gets called only if the stream completes.

Since the results of the function call is asynchronous, subscribing only could make the risk to continue the code execution without having the result. A possible solution to overcome this problem is to wrap the subscription inside a promise that will be awaited; if the subscription fails, the promise rejects, if not, the promise resolves the value.

To conclude, let's highlight the differences between REST and gRPC [30] and why prefer the latter. Proto buffer, data format used by gRPC, is smaller, faster, highly packed and efficient compared to JSON, the one used by REST. The second advantage of gRPC is that it is faster in transmission because REST API uses HTTP and so all the 7 level

of the ISO/OSI stack are involved; gRPC stops at transport layer. The code generation is faster with gRPC due to the usage of the protoc built-in compiler. However, gRPC is not supported universally like REST API, so this limits its diffusion for the moment and makes this protocol suitable for building internal systems, services that are closed to the external users. Note the gRPC is only a temporary solution, the best solution is to use a message broker using an asynchronous protocols communications. This because as the system now is defined, the microservices are coupled and this is against one principles of the microservice architecture.

3.6 Cronjob

A cronjob is a task scheduled to be executed periodically, for example to perform backup operations. In our project, since we manage requests sent by the consultant to the BMs, we used cron jobs to send notifications to the BM if he/she has not accepted or approved that request after 3 days. Another application of cronjob was to send a notification to the consultant when the BM reviews the request. Another cronjob was used to clear the notifications reads by the users older than 5 days. To implement a cronjob there were two opportunities: 1) using the 'cronjob' library provided by nest. 2) using kubernetes cronjobs [31]. The choice fell on the latter because since we are using a microservice architecture, it could be possible to add a new microservices in the future that runs a new back end technology, so Nest cronjob provided a solution working only with this framework. Further, since kubernets orchestrate our pods, using its jobs is a more high level and scalable solution. First, we defined in the notification controller a new set of API to invoke in order to execute the job required (e.g. clear the notification container). Then, we decided to create a new GitLab repository where we added all the scripts to execute periodically, scripts that are simple typescript files. These scripts merely invoke the newly defined API, with the feature that the request does not 'pass' via the Internet but reaches the pod of interest via the local endpoint. The docker images of the scripts file are built and pushed in the cloud registry by a GitLab

pipeline. Now it's the moment to modify the notification chart, adding in the template the *cronjob.yaml* file, containing the cronjob object. I will show its content and then analyse it:

```
apiVersion: batch/v1
kind: CronJob
metadata:
   name: {{ include "chart.name" . }}
   labels: {{-include "chart.labels" .|nindent 4}}
spec:
   schedule: "0 0 * * *"
   jobTemplate:
    spec:
     template:
       spec:
        containers:
         - name: notification-cronjob
         image: <docker image>
         imagePullPolicy: Always
         env :
          - name : NOTIFICATION_BACK_END_ENDPOINT
           value : http://{{include "chart.name".}}:3000
         restartPolicy: OnFailure
```

- apiVersion: defines the versioned schema of the representation of the object.
- Kind: string value that represents the resource of the object.
- metadata: standard object's metadata.
- schedule: part of the spec object, represents the schedule in the cron format. Clicking here is possible to access an online converter to get quickly the cron format. Schedule time are based on the timezone of the master where job is initialised.
- jobTemplate: specifies the job that will be created when executing the cronjob. Internally, under the containers object, is possible to define: the name of the job, the image to execute (in our case

the ones pushed in the Azure repo), possible env variables of the docker images, and the ImagePullPolicy, i.e. when and how pull the docker image (in this case every time the job is going to be executed). If the jobs fails, there is the possibility of doing nothing or restart the job; this behaviour could be set in the 'restartPolicy' field.

The docker env variable is the url of the notification back end. One important feature of jobs is that they should be idempotent, it could happen that two jobs might be created simultaneously. In our case, if this happen is not a problem for the job that requests to clear the notifications while, for the jobs that request to send notifications, these will be sent two times.

Of course, cronjob scheduled can fail. Is possible to check the cronjob scheduled through the command *kubectl get cronjob* and through the command *kubectl get jobs* is possible to see the job scheduled, showing the complete list with the name, the completion state (0/1 uncompleted or 1/1 completed), the duration and the time elapsed since its execution, as illustrated in fig. 3.6.

```
notification-backend-27660960    1/1    3s    2d10h
notification-backend-27662400    1/1    3s    34h
notification-backend-27663840    1/1    2s    10h
```

Figure 3.6: Cronjob Scheduled

Chapter 4

Research activity

4.1 Description and motivation

During my internship I carried out a research activity with the main goal of **improving the user stories estimation process** within the agile environment. I will first do a little recap of Agile and what estimate user stories means, then I will describe our estimation process and the metrics collected, analyzing the results obtained. This work is inspired to a work done by my thesis supervisor a few years ago in a German company [32].

Agile is an iterative software development practise that breaks the work into small increments in order to deliver faster the value to the customer. It's based on short iterations, each of these produce a software functionality subset showed, at the end of the interaction, to the customer, who can monitor progress and can give feedback. *Scrum* is the Agile methodology mostly used, its pillars are transparency (everything done is visible), inspection and adaption. Iterations are called *sprints*, the duration goes from 1 to 4 weeks; in our team we choose a sprint duration of two weeks. Main roles are the team member, the Product Owner and the Scrum Master. Team members self-organize the work choosing tools and conventions that best fit them. Generally a team is composed by 5-9 members, in our case we were five for the first half of the project then another developer joined us. Product Owner is the one in directly contact with the client, he has to ensure that

customer needs are well understood by the team. His main tasks are: define user stories and prioritize items in the backlog, that is owned by him. The last role, the Scrum Master, as the name suggests, is an agile expert that has to guide the team ensuring that Scrum principles are not violated and removing impediments inside the team. It is like a coach. In our team there were no fixed PO and SM, but the company gave us the change to play those roles in turn, exchanging these two roles approximately every two sprints.

The effort estimation of user stories is a process done by the scrum team during the sprint planning. Briefly, a user story is a description of a functionality required; it must be estimable. Sizing the effort of a user story means assigning points to it. This process could be done in two ways: 1) absolutely, assigning the exact effort to the story. 2) relative, comparing the estimating story to a target user story. We used this latter approach, belonging to the Fibonacci series scale to assign points. I will not consider the first two sprints since we considered them as test sprints. Starting from sprint 3, after the advice given by our company supervisor, we used only the Fibonacci series subset [1,13]. We estimated user story's effort comparing each story to the target one, a user story with effort 1 implemented during the first sprint. If a user story was for example estimated 5, it would mean that it would require 5 times more effort to target one, estimated 1. Note that user stories estimation process must take into account not only the development effort but also other processes, like testing. It's not a deterministic process, it could happen that the estimation given to a user story during the sprint turns out to be inaccurate. Particularly, two kind of errors could happen when a user story is bad estimated: over estimation and under estimation. Over estimation means that the user story requires less effort than the one estimated. If that happens, developers can relax and reduce their productivity, thus taking longer to perform tasks that they would be able to do in less time. The opposite case, under estimation, happens when a user story requires more effort than the one estimated. When developers recognize that a user story could be under estimated, in order to match the estimated effort, they can reduce the quality of the code for example writing less unit tests or

performing a not detailed code review. The effort estimation process should not be underestimated, in the following sections it is shown that by monitoring and discussing data collected about estimations, it is possible to improve this process and obtain all the benefits of having well-estimated stories.

4.2 Research questions and metrics

The goals of this activity were: improve the effort estimation process of user stories, collect the most common reasons of inaccurate estimates and try to identify some indicators of inaccurate estimations. First we tried to identify a metric to determine if a user stories was badly estimated. After a suggestion given by my thesis supervisor, in order to determine if a user story was under/over estimated, the team followed these steps: 1) we took track of the effort in hour spent for each user story. 2) For each story point category (from 1 to 13 in the Fibonacci series) we computed the median and the standard deviation. A user story of a certain category is bad estimated if its effort in hours is outside the interval centered in the median of the category and large two standard deviation. For example, if for the user stories estimated 8 the median is 20 hours and the standard deviation is 2, a user story with an effort of 25 hour is bad estimated because the effort outside the range [18,22]. Note that the effort in hours takes in consideration all kind of efforts (e.g. testing) except for the code reviews on merge requests. To collect the most common reasons of inaccurate estimations we simply collected and counted the feedback provided by the developer. To identify some possible indicators of a user story than can lead to inaccurate estimations we computed the *phi correlation coefficient* between three indicators identified by us and the list of user stories. These three proposed indicators are:

- *New technology*: user story requires the application of new technologies not used so far, like a new communication protocol. Since we are a team of interns, we could not know how to apply new technologies, receiving a lot of code reviews on merge requests

reviewed by our supervisors.

- *New functionality*: user story starts a set of new functionalities (like start implementing a new microservice). It could be an indicator because if the new functionality is not completely clear for the developer, the first implementations could not be totally aligned with the requirements.
- *Spike*: spike stories are user stories that not only have to produce something 'valuable' but also require additional tasks, in our case the study of a new technology.

Note that there is a strong overlap between 'spike' and 'new tech': often if a user story requires the use of a new technology it is also a spike story, because before applying these new technologies we had to study them. There are only a few times when the user story required a new technology and some team members already knew it (so it gone straight into application without the study, implying the user story is not a spike). The correlation coefficient was computed between each indicator (list of boolean values) and the boolean values indicating which user stories were bad estimated. We chosen the phi correlation coefficient because it is the one suitable to compute the correlation between boolean variables. This measure could be interpreted like the Pearson correlation coefficient, in fact, if we compute the Pearson correlation coefficient between two binary vectors, it will return the phi coefficient [33]. We interpreted this value referring to the *Cohen's classification*, for which a correlation coefficient between 0.10 and 0.29 is thought to represent a weak or small association, a correlation coefficient between 0.30 and 0.49 is considered a moderate correlation and a correlation coefficient larger than 0.50 is thought to represent a strong correlation. Since the metric aforementioned, to determine if a user story is bad estimated, could be prone to errors like of bias selection, for the computation of the correlation coefficients the team decided to consider a user story as bad estimated following this approach: We took as reference the target story used for the estimation process (the one estimated 1), seeing if, for example, a story that was estimated 5 actually required 5 times more effort than the reference user story. After making this comparison and

especially discussing possible difficulties and various implementation feedback (also taking in consideration the output of the metric about median and std), we decided whether or not the user story was badly estimated. In the next section will be presented the differences between this approach and the one proposed in the first metric defined.

Data collected were analyzed during each sprint planning and discussed more in depth during a special meeting at the end of sprint 6 (middle of the development), in order to see if, with a more experience in the project and with the knowledge of the errors, the user stories estimation process improved.

4.3 Results analysis

Let's now have a look on data collected comparing the one obtained at the end of sprint 6 and the one obtained at the end of development. Let's start with the first metric, the one used to determine if a user story is bad estimated basing on the median and standard deviation of its story point category. Data are shown using the variance in estimates chart: on the x axis we have the story point category while on the y axis the effort in hours. The red point represents the median of the story point category while the green lines define the interval large two standard deviation. User stories, represented by crosses, are considered well estimated if they are inside this interval. We chosen the median and not the mean because the median could protect against compensatory effects. In fig 4.1 the variance estimation chart at the middle of the development. It is relevant that the median of user stories estimated 5 was higher compared to the one of the category 8. This was an alert for the team, indicating that something was going wrong. Normally, the trend of the median points must follow an increasing trend. Category 5 was also the category with the most user stories bad estimated. The chart underlines that the team did not well catch the difference between the category 2 and the category 3, the median is about the same and the high value of the standard deviation for both the category can lead to bias selection errors. About that, one significant outcome is that the two user stories closest to the median

of the category 5 where considered as underestimated for the team (following the metric aforementioned) while for the median + standard deviation metric were estimated correctly. In fig 4.2 the variance in

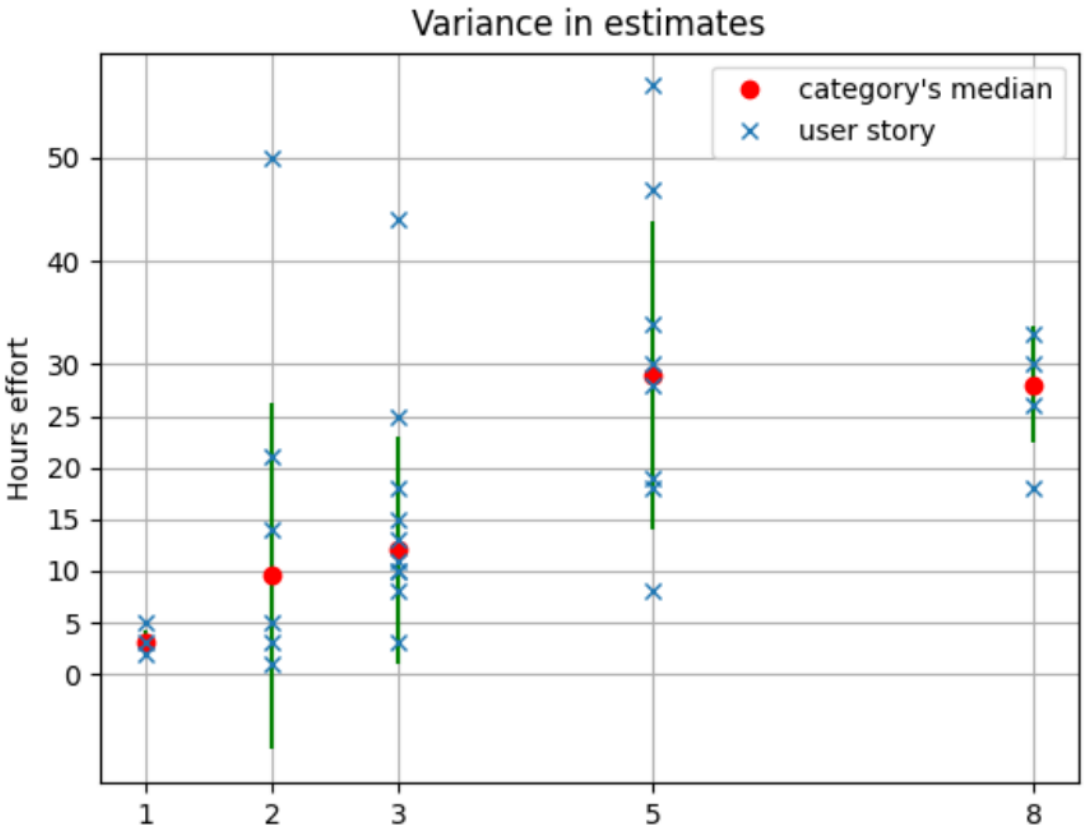

Figure 4.1: Variance in estimates chart at sprint 6

estimation chart after the last sprint. Compared to the previous one, the most important result is that now the median points follow an increasing trend, the median of the 5th category is correctly decreased under the one of the category 8. The standard deviation of the 2th category decreased significantly and there is now a clear difference between the category 2 and the category 3, where the two median values differ of almost 5 hour. There are still some inconsistencies with the category 5 and the category 8: the median values differ by just two hours. Also the fact that the standard deviation of the category 8 is increased while the one of the category 5 is almost the same compared

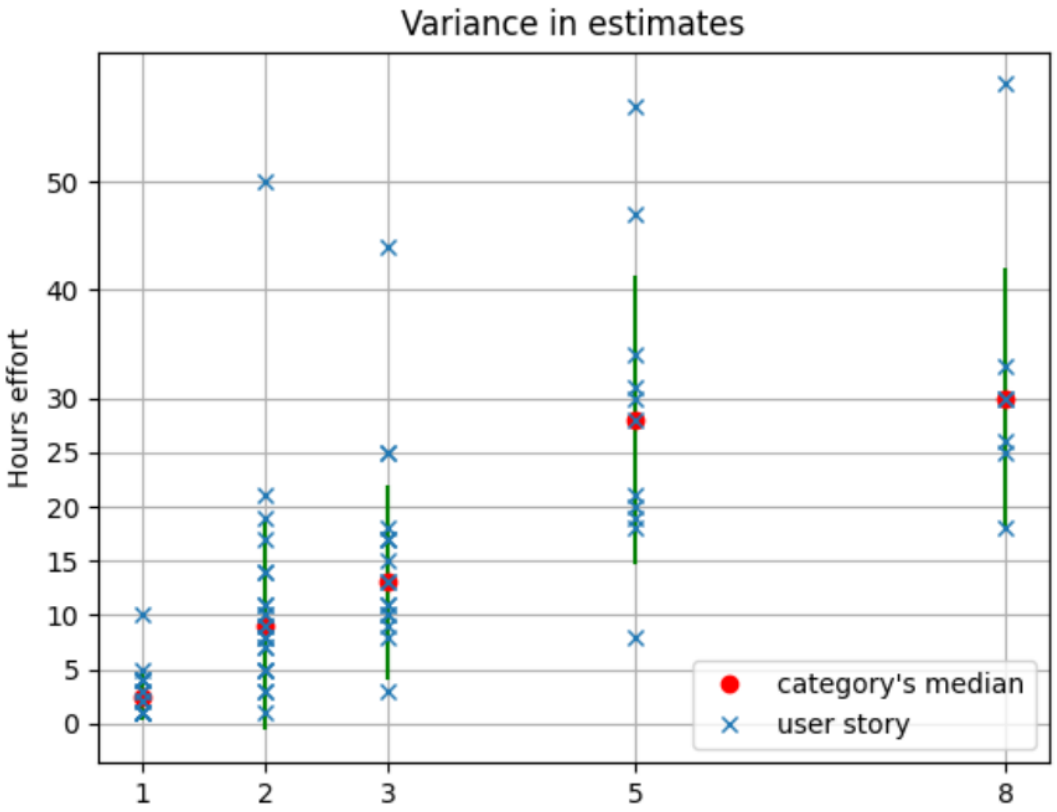

Figure 4.2: Final variance in estimates chart

to the previous chart, it indicates that the team did not conceive very well the difference between the two categories. This explains also why most user stories bad estimated for the team and not for this metric belong to the 5th and 8th category. In the table 4.1 are showed the most relevant differences between the two metrics: the most relevant results are that 8 user stories (6 after the first period and 2 in the second period) were considered as bad estimated for the team but not by the metric, while 1 user story (implemented in the first period) was considered bad estimated for the metric but not for the team. So, the metric tends to produce some false negatives because, as already said, most of this 8 stories belonged to the the 5th and 8th story point category, where the standard deviation is very high. So, the differences could be due to the choice of the arbitrary threshold for the median + standard deviation metric. These inaccuracies could be eliminated by

reducing the standard deviation of some categories or defining an upper bound for this value. Note that metric based on the team's feedback is also a non deterministic metric, it is not completely correct. However, we think it is more reliable because is based on team discussions and analysis, and developers are directly involved in the project.

Time	Median + standard deviation	Team
After 4 sprints	8	13
At the end	13	20

Table 4.1: Differences of the number of user stories identified as bad estimated by the two metrics

In fig 4.3 is shown the number of user stories bad estimated according to the team during the 8 sprints analysed. In particular, this number decreased during the time, no more exceeding the value of 2 user stories after sprint number 6. This improvement could be mainly due to two factors: 1) a better understanding of the project. 2) the analysis and discussion of data collected at the end of each sprint and during the special meeting. This helped us making decisions during the sprint planning in order to not make the same mistakes as in the past. Benefits

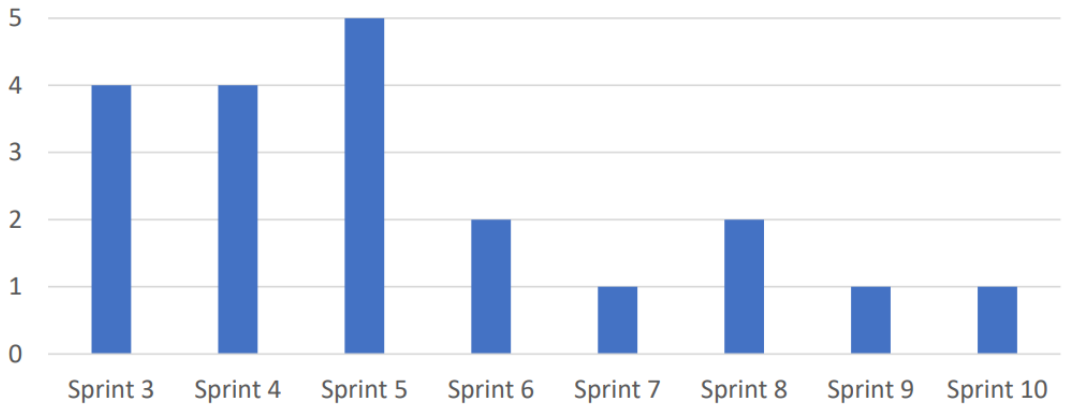

Figure 4.3: Bad estimations number trend

of more accurate planning can also be verified with the trend of the

ratio between story points done and story points planned in fig 4.4. However, even before the sprint 6 this value was not completely wrong, in fact there are only two sprints where the ration was lower than 1. In sprint 3 we think we were not able to complete all the story points

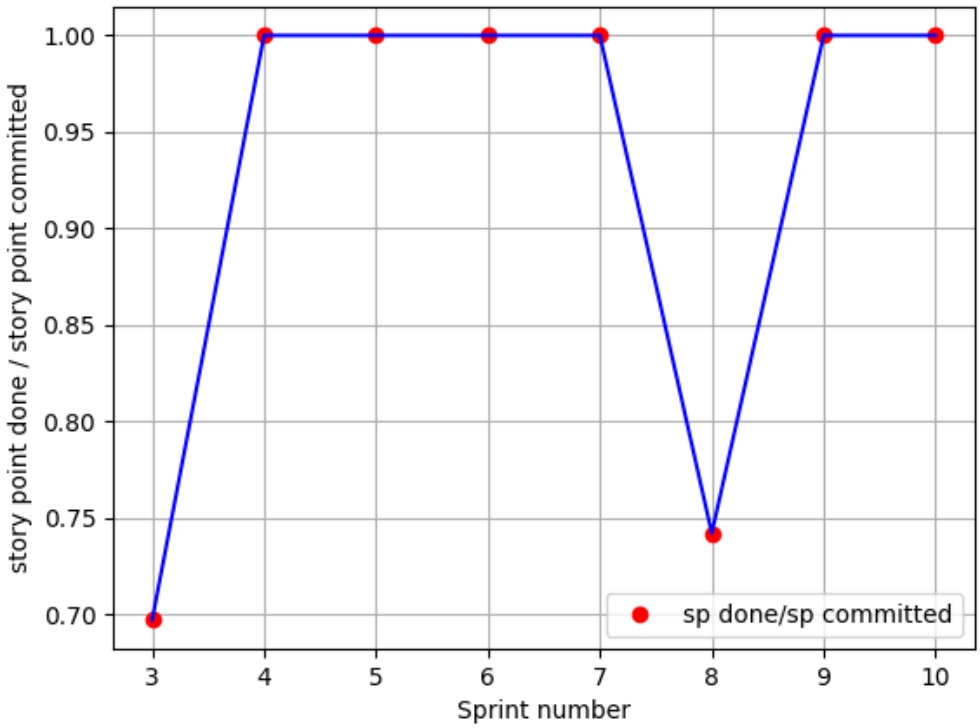

Figure 4.4: Trend story points done over story points committed

committed because it was the first sprint where we started estimating user stories using the range [1,13] of the Fibonacci series. In sprint number 8 the ratio is not 1 because during this sprint many team members didn't work for a few days because of university exams, a public holiday for the French national day, and also the last Tuesday and Wednesday of the sprint the team attended a workshop on UX/UI that occupied us the whole day. So, the main reason of the mistake made at sprint 8 was a *lack of organization*. The histogram in fig 4.5 better shows the differences between the story points done versus the

ones committed. What is relevant is that the story points committed at Sprint 6 where higher compared to the average because in this sprint the sixth developer joined us and also because during this sprint all the team members were available for working at the project, free from other tasks like interviews with the future clients or work commissioned by our supervisors for client projects. Team velocity was not perfectly

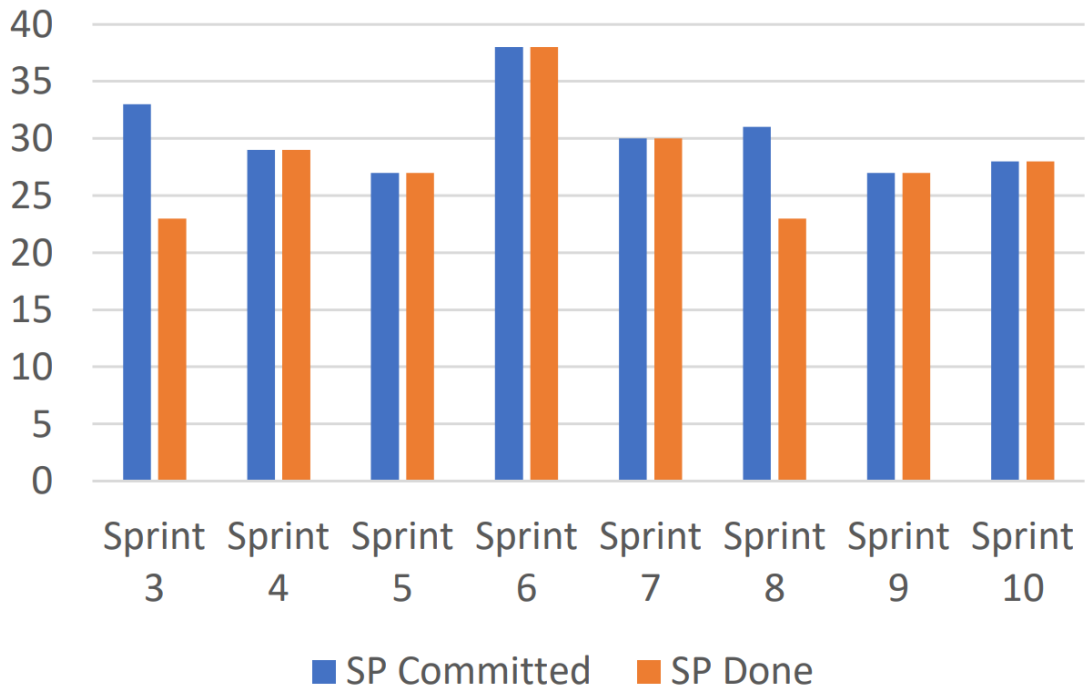

Figure 4.5: Histogram comparing story points done versus committed

constant because data are bad affected by the errors in sprint 3 and sprint 8. In table 4.2 are shown the mean, the median and the standard deviation for story points committed and done. If we remove the two inaccurate sprints (3 and 8) from the computation of these values, of course we will get the same medium and mean and also the standard deviation of story points done decrease by 1 unit.

So far we have analyzed data concerning user stories bad estimated; it remains to analyze developers' feedback and the correlation coefficient of the possible indicators. Let's start with the results obtained from the phi correlation coefficient. In table 4.3 are shown the correlation

	Mean	Median	Standard deviation
Sp committed	30.3	29.5	3.4
Sp done	28.1	27.5	4.4

Table 4.2: Mean, median and standard deviation of story points done and committed

coefficients computed between user stories and the indicators 'new technology', 'new functionality' and 'spike story'. Interpreting the results using the Cohen's classification, for the 'spike' indicator there is a *moderate* correlation while for the other two the correlation is *small*. About the 'new functionality' indicator, although the correlation

Indicator	Coefficient value
New functionalities	0.279
New technology	0.219
Spike story	0.321

Table 4.3: Phi correlation coefficients

coefficient is low, we think that the team should pay attention when estimating a user story introducing a new functionality, assuring that all team members have a clear idea of the story goal and of the new set of functionalities. The correlation coefficient of the 'new technology' indicator is also low and we do agree with this result. We noticed that the user stories bad estimated that required new technologies were implemented only at the beginning of the project, when we lacked of experience and we did not know best practices. If the team was composed by more experienced developers, probably, we would have a lower coefficient value. Furthermore, POCs could help reduce the effort while applying a new technology. So, we think that the 'new technology' cannot be considered as a good indicator. About 'spike story' indicator, we think it could be an indicator of inaccurate estimates. Except for one, all spike stories were bad estimated. What more influences bad estimation for spike stories is the fact that they require additional tasks

that are difficult to estimate because often too subjective (e.g. study a new technology); this could be the reason why is the indicator with the highest coefficient value, that can be interpreted as moderate.

In addition to the possible indicators, we collected and counted the occurrences of additional reasons of inaccurate estimations according to the developers. These additional reasons were:

- Difficult in testing: what badly influenced the team at the beginning was that only few of us already used test's frameworks like jest or generally tested an application. Testing the features of our project was not easy at all since it required the introduction of new tools and concepts new for everyone. This, especially at the beginning, influenced a lot the user story estimation, mainly because who never wrote a test had difficulties on giving an estimation since they did not know the effort required. This is a demonstration that our lack of experience impacted our estimates.

- Difficult in the implementation: user story was clear, well understood by developers, but due to technical problems or unexpected code behavior, has proved to be difficult to implement. When this happened, we asked for help to our supervisors to reduce the effort in the implementation but it was not enough to have an under estimation of the user story.

- User story not clear during the planning: PO has the important role to define and prioritize user stories. Especially at the beginning, when the functionalities of the project are not very clear, if a user story is not well defined, it could happen that team members perceive different difficulties. Team had not clear the goal of the user story, implying that during the implementation, when the functionality to implement become clear, the implementation time needed is more or less than expected. It's important that during the planning a user story is well discussed in order to be clear for all the team, before giving an estimation.

- Client needs not clear: PO defined the user story in a clear way, following the structure "As a <role> I want to <perform action>

so that <value created>". However the PO was not quite sure of what the customer wanted and during the sprint, after a discussion with the client, the story requirements were changed, thus leading to a slowdown in implementation.

- Code review took a lot of time: especially at the beginning or when implementing a user story that required the use of a new technology, code review on a merge request implemented by our supervisors took a lot of time since we received a lot of comments due manly to our lack of experience.

Let's now see and discuss the occurrences. In table 4.4 the values at the end of the project, the symbol '-' means that there where no occurrences. The most common reason is "User story not clear during the implementation", that is also the only reason that caused over estimations. This is followed by "Difficult in the implementation", other reasons occurred once and only caused under estimations. Having 14 occurrences for the "User story not clear during planning", underlines the fact that the role of the PO is fundamental; these errors could be avoided if: 1) user stories and their acceptance criteria were defined better 2) during the planning it was ensured that each team member had a clear idea of the purpose of the story and of all the aspects that would influence its implementation. In table 4.5 are shown the occurrences collected at the end of sprint 6. These data were analyzed during a special meeting. Let's compare them to the one collected at the end of the project to see if there where improvements. Despite we knew that an unclear user story could lead to bad estimates, we made this error another five times. With the increasing of the knowledge and the experience, we had no more long code reviews and difficulties in testing. We have twice more experienced "difficult in the implementation", for reasons not due to unclear user stories but to problems that developers may encounter every day.

Description	Total occurrences	Under estimations	Over estimations
Difficult in testing	1	1	-
Difficult in the implementation	3	3	-
User story not clear during the planning	14	11	3
Client needs not clear	1	1	-
Code review took a lot of time	1	1	-

Table 4.4: Developers' feedback occurrences after last sprint

Description	Total occurrences	Under estimations	Over estimations
Difficult in testing	1	1	-
Difficult in the implementation	1	1	-
User story not clear during the planning	9	6	3
Client needs not clear	1	1	-
Code review took a lot of time	1	1	-

Table 4.5: Developers' feedback occurrences after sprint 6

4.4 Conclusions and future works

In this section I have described the research activity carried out during the development of the project. The main goal was to improve the user story estimation process after collecting and analysing data in a special meeting; others goals were to identify the most common reasons of bad estimations and possible indicators of bad estimates. We tried to individuate a metric that could identify automatically if a user story was under/over estimated; comparing it to a metric based on developers' feedback, the differences were not so huge. The metric tended to produce some false negatives, these kind of errors can be reduced by decreasing the standard deviation of some story points category, possibly limiting the value. The most important result is that we *were able to improve the estimation process*, decreasing the number of inaccurate estimations. This result is obtained thanks to a bigger experience on the project and to the discussion and the analysis of data collected. Not only data regarding this metric were discussed but also feedback provided by developers, trying to identify the most common reasons of inaccurate estimations. About feedback collection, it is noticeable that some mistake happened once, never happened again. The most common reason of errors was "User story not clear"; even if after sprint 6 it had a significant value of occurrences, was not enough for the team to avoid this error, causing other bad estimations. However, since this is mostly a PO responsibility, we think that this is an expected behaviour: the company gave us the change to play in turn the PO and SM roles, so exchanging the PO role lead to possible inconsistencies and different points of view. From our feedback, also, came out that one factor that mostly impacted the team was our lack of experience; after some sprints the team gained a bit more of experience and some mistakes were no longer made. The lack of experience was an expected problem since we had all different academic backgrounds and for most team members it was the first working experience in an Agile environment. Regarding the possible indicators of inaccurate estimations identified by us, computing the phi correlation coefficient between the indicators and the user story data, we did not get strong

evidences to draw significant conclusions. However, we think that "New functionality" indicator and especially the 'Spike' indicator could be good indicators for inaccurate estimations. It's interesting that our data collected are close to data obtained from a survey made in 228 companies working in 10 countries, about the contemporary problems practitioners encounter [34]. For a future work, a machine learning algorithm could be used to determine if the estimation given to a user story could be prone to errors. We tried to use a logistic regression model to determine if the user stories of the sprint 10 were recognized as good or bad estimated according to developers' feedback, labelling them with the three indicators aforementioned. The interesting result is that the confusion matrix has an accuracy and precision of 1; the only user story bad estimated was identified by the model. Of course, using the list of user stories until sprint 9 as train set, is a limitation since it is very small and unbalanced. However, this work could be considered in the future, where more data from different projects could be available and used.

The results of this research activity could be used by the company for internal projects and for the next team of interns that will continue the development of the project. Data cannot be generalized since only the first sprints of a single project are analyzed and, for example, other companies could not use the concept of 'Spike story', using transversal tasks instead.

Chapter 5

Conclusions

In this master thesis work is presented my experience in a French consultant company while developing a web application in an Agile team. The main functionality of the application is the management of clients and consultants of the company. The thesis work extends the internship report committed at the end of the stage, particularly, it is added the last chapter describing the activity research made in parallel during the stage.

During the internship, we had the possibility to take part in the initial steps of the project, defining the microservice architecture of the system. We deployed the application in a cloud environment, exploiting all the advantages of cloud computing. The development of the project followed the DevOps philosophy, using interesting tools like Terraform and Pipelines in order to automatize operations. The Minimum Viable Product was delivered on date 12/08/2021. Right now, are implemented 3 application functionalities: 1) Management of entities inside a tenant. 2) Consultant management by HR and Business Managers. 3) Consultant's rewards management. The entities that could be managed by the application right now are the employee of the Sophia Antipolis Business unit, that are around 100 people. The development of the project will be continued by a new group of interns. Momentarily, users are performing a "friendly user test" in order to collect suggestions and feedback, all positive for the moment.

The next team can be guided by the outcomes of the research activity

carried out during the 6 months. The activity consisted in collecting both qualitative and quantitative data and analyze them in order to improve the user story estimation process. We tried also to identify most common reasons of inaccurate estimations and possible indicators of bad estimations. The activity highlighted the importance of the PO role: there were a lot of inaccurate estimations (15 out of 20) because user stories where not clear or client needs not well understood. Another factor that influenced negatively the development was our lack of experience. The activity demonstrate that gaining a bit of experience, knowing more the environment, the project's requirements and analysing data and errors, can improve the user story estimation process. Analyzing data collected helped also to not have the problems encountered at the beginning, like difficulties in testing and long code reviews, the only exception is for the "User story not clear" error. However, we think that we had this problem because we exchanged the PO role during the development, involving to expected inconsistencies and different points of view.

We cannot draw significant conclusions regarding the possible indicators of inaccurate estimations, the values of the correlation coefficients do not indicate a strong correlation and also data are related to a single project. However, we think that the "New functionality" and the "Spike" indicators should be taken in consideration by the next team when estimating user stories implementation effort. The first indicator indicates a user story that introduces a new set of functionalities, it is important to make sure that each developer has well understood the new functionality goals. The second indicator indicates that the user story requires also to implement further tasks (like study a new technology) that increase the difficulty of the estimation, so it's important that the team discuss about possible difficulties and factors that could affect he implementation of these transversal tasks.

Bibliography and Sitography

[1] https://en.wikipedia.org/wiki/ALTEN. visited on March 2022 (cit. on p. 1).

[2] https://en.wikipedia.org/wiki/DevOps. visited on March 2022 (cit. on p. 2).

[3] https://fulcrum.rocks/blog/proof-of-concept. visited on March 2022 (cit. on p. 5).

[4] https://c4model.com/. visited on March 2022 (cit. on p. 7).

[5] https://docs.nestjs.com/. visited on April 2022 (cit. on p. 10).

[6] https://xebia.com/blog/so-what-is-a-deployment-really/. visited on April 2022 (cit. on p. 11).

[7] https://en.wikipedia.org/wiki/Cloud_computing. visited on April 2022 (cit. on p. 12).

[8] https://www.geeksforgeeks.org/difference-between-iaas-paas-and-saas/. visited on April 2022 (cit. on p. 13).

[9] https://www.docker.com/resources/what-container/. visited on April 2022 (cit. on p. 14).

[10] Tim Hockin. *A Crash Course on Container Orchestration*. May 2017 (cit. on p. 16).

[11] https://manish-bannur.hashnode.dev/kubernetes-k8s-kybernhths. visited on April 2022 (cit. on p. 16).

[12] `https://phoenixnap.com/kb/what-is-helm`. visited on April 2022 (cit. on p. 21).

[13] `https://harness.io/blog/what-is-helm`. visited on April 2022 (cit. on p. 23).

[14] `https://docs.docker.com/compose/`. visited on April 2022 (cit. on p. 23).

[15] `https://www.terraform.io/intro`. visited on May 2022 (cit. on p. 24).

[16] `https://en.wikipedia.org/wiki/Active_Directory`. visited on April 2022 (cit. on p. 30).

[17] `https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-auth-code-flow`. visited on April 2022 (cit. on p. 30).

[18] `https://docs.microsoft.com/en-us/azure/active-directory/develop/msal-overview`. visited on April 2022 (cit. on p. 33).

[19] `https://www.pagerduty.com/resources/learn/what-is-a-pipeline-in-devops-and-how-to-build/`. visited on May 2022 (cit. on p. 34).

[20] `https://docs.gitlab.com/ee/ci/pipelines/`. visited on May 2022 (cit. on p. 39).

[21] `https://azure.microsoft.com/en-us/services/cosmos-db/#overview`. visited on May 2022 (cit. on p. 39).

[22] `https://en.wikipedia.org/wiki/Cosmos_DB`. visited on May 2022 (cit. on p. 40).

[23] `https://docs.microsoft.com/en-us/azure/cosmos-db/sql/sql-query-udfs`. visited on June 2022 (cit. on p. 41).

[24] `https://www.testim.io/blog/supertest-how-to-test-apis-like-a-pro/`. visited on April 2022 (cit. on p. 44).

[25] `https://www.testcontainers.org/`. visited on April 2022 (cit. on p. 45).

[26] https://grpc.io/docs/what-is-grpc/introduction/. visited on May 2022 (cit. on p. 46).

[27] https://developers.google.com/protocol-buffers/docs/overview. visited on May 2022 (cit. on p. 46).

[28] https://blog.logrocket.com/using-observables-transform-data-typescript/. visited on May 2022 (cit. on p. 49).

[29] https://blog.angular-university.io/rxjs-error-handling/. visited on May 2022 (cit. on p. 49).

[30] https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/. visited on May 2022 (cit. on p. 49).

[31] https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/. visited on July 2022 (cit. on p. 50).

[32] A. Vetrò M. Conoscenti V. Besner and D. Méndez Fernàndez. *Combining Data Analytics and Developers Feedback for Identifying Reasons of Inaccurate Estimations in Agile Software Development*. 2019 (cit. on p. 53).

[33] https://en.wikipedia.org/wiki/Phi_coefficient. visited on August 2022 (cit. on p. 56).

[34] M. Kalinowski D. Fernández M. Felderer and S. Wagner. *Naming the pain in requirements engineering*. 2017 (cit. on p. 68).