Development of an infrastructure automation API platform

by

Adrien GASTÉ

TECHNICAL REPORT PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE IN PARTIAL FULFILLEMENT OF COURSE STA802 - STAGE D'ENTREPRISE ET RAPPORT TECHNIQUE

MONTREAL, AUGUST 23, 2018

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

ACKNOWLEDGMENT

I would like to personally thank:

- the team and manager of the AUTO project for welcoming me warmly in their team to work on this project, for always taking time to answer my questions and for helping me acquire a great deal of knowledge,
- Ubisoft for giving me this opportunity to work in their company for four months,
- My school, École de Technologie Supérieure, for giving advice in how to find and prepare an internship through one of their courses,
- My supervisor, Alain April, for accepting to review my internship report and being available to answer my questions,
- Last, but not least, Florian Elizagoïen, without whom I would not have gotten this internship.

DÉVELOPPEMENT D'UNE PLATEFORME API PERMETTANT L'AUTOMATISATION D'INFRASTRUCTURES POUR LES ÉQUIPES DE PRODUCTION D'UBISOFT

Adrien GASTÉ

RESUMÉ

Lors de mon stage chez Ubisoft, j'ai contribué au développement d'une plateforme offrant une API capable d'automatiser des infrastructures virtuelles. Nommée AUTO, elle est destinée aux équipes de production de jeux vidéo ainsi qu'aux administrateurs services.

Lors de mon mandat de stage, nous avons continué à améliorer la plateforme qui a été déjà déployée à mon arrivée. L'équipe s'est centrée sur l'ajout d'un service au produit qui permet d'automatiser la création d'une infrastructure avec plusieurs ressources simultanément. Nous avons notamment automatisé le déploiement d'une application de surveillance d'infrastructures sur une base de données orientée documents. Nous avons également déployé un système de gestion de bases de données.

À l'occasion de la version 1.2, nous nous sommes concentrés sur la création d'un service permettant d'automatiser la configuration d'un répartiteur de charges pour des serveurs utilisés par les équipes de production de jeux vidéo, ainsi que sa validation et sa suppression. Cette tâche nous a également poussés à mettre à jour notre plateforme dans l'environnement de production où allait être déployé notre nouveau service; cela consistait à ajouter ou mettre à jour les différents services proposés par notre produit.

Plusieurs *bugs* de taille plus ou moins importante auront été détectés et corrigés durant le stage, et la documentation officielle de la plateforme a été mise à jour tout le long du projet, afin de faciliter son utilisation par nos clients.

Adoptant la méthode Scrum, nous avions des réunions hebdomadaires et spontanées pour organiser et partager le travail, ainsi que permettre à chaque membre de proposer des idées ou donner des conseils.

Mots clés : Infrastructures, Automatisation, Infonuagique, Agile, Orchestration

DEVELOPEMENT OF AN API PLATFORM FOR AUTOMATING INFRASTRUCTURES FOR THE PRODUCTION TEAMS OF UBISOFT

Adrien GASTÉ

ABSTRACT

During my internship at Ubisoft, I have participated in developing an API platform for automating infrastructures. Named AUTO, it is destined for the teams responsible for making the company's video games, as well as service administrators.

Durant my mandate, we have continued improving the platform whose first version was already released when I arrived. The team focused on adding a service to the product capable of automating the creation of an infrastructure with more than a few resources simultaneously. We have also automated the deployment of a monitoring software, named Sensu, on a document-oriented database called MongoDB. We have notably created a database management system, called PostgreSQL.

At the next iteration, we concentrated on creating a service capable of automating the configuration of a load balancer for servers used by video game production teams, as well its validation and its deletion. This task has also forced us to update our platform in the production environment in which our new service would be released; it consisted in adding or updating the different services proposed by out product within said environment.

Bugs of size more or less important related to services or the platform itself have been detected during these iterations and have been corrected during the internship, and the API's official documentation was also updated frequently, so as to help its use by our clients.

Adopting the Scrum method, we had weekly and spontaneous meetings to organize and share our work, as well as allow each member to suggest ideas or give advice on something.

Keywords: Infrastructures, Automation, Cloud computing, Agile, Orchestration

TABLE OF CONTENTS

Page

OTHER CLASSES, AND A FUNCTION MADE UP OF TWO METHODS3535

LIST OF FIGURES

Page

LIST OF ABREVIATIONS

API	*Application Programming Interface*
ÉTS	École de Technologie Supérieure
GES801	Application Scope of Project Management (ÉTS Course)
MGL805	Software Verification and Quality Assurance (ÉTS Course)
MTI825	Information Technology Service Management (ÉTS Course)
PMBOK	Project Management Body of Knowledge

INTRODUCTION

As part of my Master's degree in Information Technology at École de Technologie Supérieure (ÉTS), I have fulfilled a four-month internship at Ubisoft Montréal, a video game developer well known for releasing several critically acclaimed video game franchises, such as Assassin's Creed and Far Cry.

With the growing number of gamers throughout the world, the company needs even sturdier and efficient services to continue delivering top quality games and continuous support to guarantee an unforgettable experience to the players. This increases the workload for the employees who find it more and more challenging to create and maintain the necessary infrastructures able to keep up with the demand. The Information Technology service of Ubisoft has therefore created a project with the goal and vision to offer within the company a true infrastructure as code platform[1]: AUTO, a platform providing an API for providing infrastructure automation services: creation of virtualized environments, domain name servers, and even the provisioning of an entire IT production infrastructure.
Having joined the team as the first version 1.0 was being released, I have taken part of its ongoing development and improvement where I have added new modules and functionalities while fixing defects that emerged as time went on.

The next parts will first describe how the knowledge acquired at ÉTS was applied during my internship. We will then talk about the tasks I have contributed to in regard to the development of the AUTO platform.

Due to a signed non-disclosure agreement, several services and technologies used within the API platform cannot not be explicitly named. I apologize in advance for the inconvenience it might cause to the reader of this document.

---

[1] Process of provisioning and managing computer data centres through machine-readable definition files, rather than physical hardware configuration of interactive configuration tools.

CHAPTER 1

THE THEORETICAL KNOWLEDGE ACQUIRED AT ÉTS APPLIED TO THE INTERNSHIP'S MANDATE

1.1 Project development and management – GES801, MGL805, MTI825

The strongest notion that I learned and worked on at ÉTS and directly involved in my internship was project development and management.

Indeed, during the course GES801 - Application Scope of Project Management, we have grasped the basic knowledge of how to manage a project, its different steps and what it entails, through the teachings of the Project Management Body of Knowledge guide (PMBOK), a standard of terminology and guidelines for project management. Most importantly, we detailed the five process groups a team has to go throughout a project:

- Initiation: during each of our project's iterations, we had a couple of meetings between team members and the project manager to define the next steps we should take to further develop our application; a base plan for the entire upcoming year was given by our project manager, but ideas and suggestions either from the team or from our clients could also come up spontaneously. These were discussed to see whether it was an urgent matter or if it could be an addition for later, to leave place to probably more important features. Each member could always give his opinion on the matter, with the project manager ultimately deciding on the final work plan we would follow.
- Planification: with the roadmap correctly defined and authorized by the project manager, we had to further refine the objectives by splitting them into smaller tasks, plan the schedule and see what could be done during the sprint (the methodology will be described in Chapter 1.3). We would then decide on the workload distribution between us: some members were assigned their tasks as they were the most fit to accomplish them successfully, however each of us could also choose which tasks they would be interested in doing, no one was forced to take something they did not feel comfortable with.

- Execution: Task definition for one version usually started during the previous one and workload distribution at the end. The reason we do this is to gain time in management and because it also depended on our clients, who came up to us to ask for new or improved services. Once everything was set, we would then work on each task, depending on their priority. We assigned them to ourselves, and we could also pair program with another member of the team for more complex ones, however some could also come up on the way, be reported by a team member and assigned to someone else. We relied on an issue tracking software to follow the team's progress for the given objectives (it will be more detailed in Chapter 1.2).
- Monitoring and controlling: As stated before, we use a software to track the progress of each task. When was done, it would then go into review and testing, where everyone could give their opinion and check that what has been done works the way we want it to. Documentation was also done during the entire process, to give a clearer explanation of what our services offer.
- Closing: After all tasks were completed, reviewed and done, there was a final demo shown to the project manager to show him our work and what has been done. He would then give his feedback and if it was positive, we would close this iteration and officialise its release to the rest of Ubisoft by making an announcement on the company's private website. All the tasks done were then archived and we moved on to the next iteration.

These five process groups are equivalent to systems development life cycles, a term used to describe the process to follow to deploy an information system; it is a concept that has been seen in the previous course, as well as course MGL805 - Software Verification and Quality Assurance.

Another important notion learned at ÉTS is the concept of DevOps, a software engineering culture that unifies software development and software operation, learned through course MTI825- Information Technology Service Management. Our very API platform is based on the foundations of DevOps.

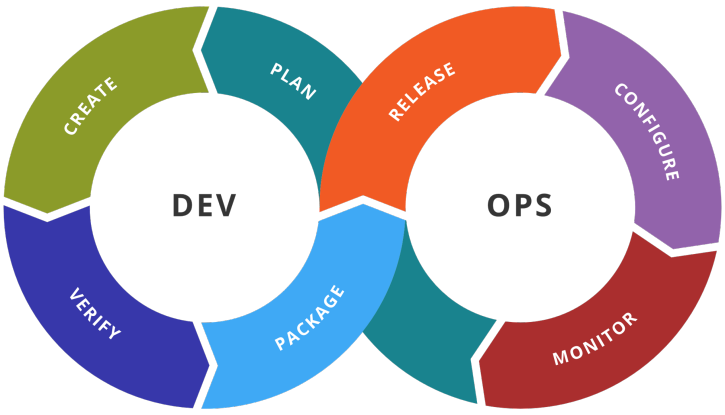

Figure 1.1.1: DevOps Toolchain, Kharnagy, 2016, Wikipedia

Indeed, our way of developing new features for our product follows the DevOps toolchain. When wanting to add a new feature to the AUTO service, we define the necessary business value and requirements through team meetings (“Plan” step).

We then proceed to the “Create” step, where the coding takes place. When a pre-release version is ready, it is tested for its performance and quality, relying on the Gitlab solution, which not only provides test automation, but also continuous integration and continuous delivery, three essential concepts of DevOps (“Verify step”); they help us in continually integrating and testing every change we make in order to improve our service when it is up for review and speed up delivery by automating the software release process.

These changes are then approved and deployed for production, always through Gitlab as it hosts all our projects, as well as a history of all the modifications made since the first release. If the deployment for some reason fails, we can rollback to the previous version in the master branch of said project (“Release” step). Monitoring is constantly done when a service is released: each instance providing a service is closely checked to make sure it runs correctly and does not use too many resources (“Monitoring” step).

Quality assurance is an essential part of any software product. Defined as the means of preventing mistakes and defects when delivering a product to a client, it is the main subject of one the courses followed at ÉTS, course MGL805.
It is one of the most important steps of the systems development life cycle, because it is what guarantees the quality of the product.

Once the coding for a specific component, it is reviewed and tested by the one who created it, but also other members of the team. This is because while it can work individually, there is no guarantee it will work when combined with the rest of the system. Therefore, before even thinking of pushing these changes to Gitlab, the one responsible for the task tests it locally by integrating it with the service he is currently working on. Once he is sure everything works fine, only then does he create a merge request on Gitlab to push his modifications; it is then up for review for the rest of the team. This step is essential because since we all have different ways of viewing things, they may come up with something we would not have thought about and vice versa. They give their opinion about the changes made, test it themselves and even try to break it to see if it should not work when it is not supposed to.

This step prevents anyone from integrating code without verification which could break down the entire system.

With automation, continuous integration and continuous delivery, monitoring integrated as well as being an infrastructure as code platform, the AUTO API is a true DevOps product.

1.2 Methodology for the project development: Agile, Scrum and Kanban– MTI825, GES801

At school, I learned new ways of how to approach a project: these are the Agile approach and its framework Scrum. I also acquired basic knowledge in task management and issue tracking, through the Kanban lean method These notions have been addressed in courses MTI825 and GES801.

The Agile approach is a methodology for project management which encourages emphasis on adaption to change, continual improvement and collaboration between cross-functional teams. It is based on 12 principles from the Manifesto for Agile Software Development to follow to improve our product's quality, most of which have been recognized and used within the AUTO project:

- As said in the previous part, some changes can come up much later after the plan has been prepared,
- We had some objectives that evolved throughout an iteration and we have consistently adapted to it,
- Each of our iterations is based on a six-week schedule, during which we propose a certain number of features that we commit to deliver on time; there has been cases where we released later, but others where it was ready before due time. These short schedules incite us to continually improve our product and suggest many more features,
- While the team and the manager had weekly and daily informal meetings, the manager always kept in touch with the client.
- Even though we were a team of nearly ten people, we all sat at desks next to or in front of each other and always had face-to-face conversations with other members when having issues with a specific task,

- Code reviews were mandatory when we finished with a task, as the approval of at least two team members was required to merge our modifications in the master branch of the project; this helped in guaranteeing continuous attention to code quality,
- Even if we sometimes wrote code in a verbose way, we made it a priority to always simplify it as much as possible,
- Like stated previously, each team member could give his opinion on architectural designs or requirements; ideas from different people leads to creating a much more optimal solution than simply proceeding with one mindset,
- Pair programming is quite frequent, as it helps work faster if it is handled well,
- One-on-one meetings were done with the manager with each member of the team so that we could reflect on how performant we are and what we wanted to achieve and how to become more performant.

More importantly, we have also adopted the Scrum framework for managing our project. Indeed, the manager had given us a list of all features that were required to have in the AUTO platform for the upcoming year. He would then concentrate on the first sprint, where he would hand us the ranked list of what he wants for the next release.

The team would therefore select all those it could actually commit to deliver by the end of the sprint. The tasks were then broken down between everyone after going through all of them, which effectively started the sprint that lasted for six weeks.

Every morning, we would have an informal stand-up meeting to see how was done the previous day and what will be done this day. It was also the opportunity to raise any blockers to get help from other when possible. Finally, there were weekly meetings on Friday to check the overall sprint advancement and see if we were falling behind the due date or not, with task priority occasionally changing and suggestions on how to solve some issues.

When the product was ready to be delivered, a sprint review and retrospective was done to see if we had done a good work and if we fulfilled the client's requirements. We then went to the next iteration and started the whole process again.

Handling the tasks progress efficiently would not have been possible without the Kanban method, a scheduling system for managing work through visualisation of a Kanban board. Because we had a high number of tasks for each iteration, this was the only method we could use to efficiently follow the sprint's progress.

We relied for this on Jira, a proprietary issue tracking software. All tasks were listed per version release, and had a priority tag, going from P4 (weakest) to P0 (strongest). They were all display on a virtual Kanban board made up of six columns: "To Do", "In Progress", "Blocked", "In review", "In testing" and "Done". Each task could be assigned to one team member (however, more than one member could work on it), who updated its status by moving the task around the different columns: this helped us know how far we had gotten compared to the time we had left. We could leave comments in a corresponding task to show the advancement but more importantly, the Kanban board was directly linked to our AUTO project on Gitlab; this allowed us to easily handle task progress with both technologies.

In the next chapter, we will describe the main tasks executed during this internship. They will be split in three sub-chapters, corresponding to the three sprints I've taken part of.

CHAPTER 2

MANDATE OF THE INTERNSHIP

2.1 The AUTO API Platform

Like presented quickly in the introduction, AUTO is an API platform developed with the goal of helping in simplifying and automating infrastructure sources within the company.

Its inception came about as production teams and service administrators faced a challenging task when dealing with infrastructures, and this solution would offer a true infrastructure as code platform[2].

When the first version 1.0 was released, several RESTful services supporting CRUD (Create, Read, Update, Delete) operations were available to use, and such as orchestration and configuration management tools. In order to use these services, the client needs to authenticate and get permissions; they are necessary headers to make the API call. The HTTP request is sent to a gateway based on Kong[3], which then sends the request to the appropriate service.

2.2 Iteration v1.1

2.2.1 Automate the installation of a Sensu client in the MongoDB service

To help in their tasks, the AUTO team has conceived some Terraform modules that can be reused in other services. One such is the MongoDB Terraform module, based on the MongoDB document-oriented database.

---

[2] Process of provisioning and managing computer data centres through machine-readable definition files, rather than physical hardware configuration of interactive configuration tools.
[3] Microservice API gateway.

To monitor such services, we have relied on a framework called Sensu. Working as a monitor for infrastructures, it can for example give checks on the health of an application (whether it is alive or not), show the live CPU usage, or even memory usage. It has already been implemented in more than a few services used by AUTO, but at that point had yet to be implemented in the MongoDB instances relied on by our product. The goal was to first successfully install a Sensu client[4] manually in an already existing MongoDB instance, check that it worked correctly (through testing the different checks available), and then automate the installation process so that when a MongoDB instance is created (the creation process will be explained later), the Sensu client is automatically configured to be installed and ready to use within it.
This task helped me learn to be familiar with some of the technologies used by our team for the AUTO platform, namely Terraform and Ansible.

The MongoDB instance is, in other words, a Mongo database orchestrated with a Terraform module.
This instance is then provisioned and configured with several services with the help of an open-source automation platform named Ansible, that allows for configuration management and application deployment. This is done through what we call Ansible roles, written in the YAML language, which consist of many Ansible playbooks, scripts that allow the configuration of complex environments through simple commands (for example, which packages are necessary to install an application, how the service is to be configured).

This is where plugging the Ansible role for the Sensu client into the MongoDB instance comes in.
After first deploying a MongoDB instance in the AUTO Staging environment[5], we pulled an already defined Ansible role capable of configuring the Sensu client service from the project's Gitlab into my local machine.

---

[4] Runs in the system requiring monitoring, and connects to the Sensu server, via a message bus. The one by default (and the one we use) is RabbitMQ.
[5] Virtual environment where all services are tested to make sure they work correctly before being deployed in the Production environment (the one being used by the clients using the AUTO platform).

The client IP address (corresponds to the instance which requires the monitoring) was hard coded in its configuration files such that it pointed to the IP address of the MongoDB instance (it was obtained by accessing the OpenStack cluster for AUTO listing all instances deployed in the Staging and Production environment).

We also updated the http and https proxy settings to allow the MongoDB instance to access Internet; indeed, the Sensu Ansible role, once installed in the instance, needed access to the Internet to fetch the necessary packages when executing its tasks defined in the Ansible role. The reason this change had to be made is because the network in which we worked has some restrictions, and the default set values prevented communication between the MongoDB instance and the outside world, thus preventing the Ansible role from completing its tasks to configure the Sensu client the right way on the MongoDB instance. The issue was discussed with teammates who helped me find the correct proxy settings allowing valid interaction between the Ansible role and deployed Terraform module.

All that was left was setting up RabbitMQ, the message bus that the Sensu client with the Sensu server. For this purpose, we edited a JSON configuration file, in which the host and

port of the Sensu server was specified, as well as the user and password necessary to log in the server.
With all these modifications, the Ansible role could then correctly install the Sensu Client service in the MongoDB instance, configure the Sensu client and link it to the MongoDB, as well as the RabbitMQ host which links the instance to the Sensu server.

To check that the service was correctly running in the instance, we used another already available service, Uchiwa, an open-source dashboard specifically conceived for the Sensu monitoring framework. Make note that the Uchiwa dashboard is the front-end user interface of the Sensu server itself. By logging in to the one in Staging, the verification was done by checking the list of all instances connected to it and see if the MongoDB instance was there.

To this end, we edited a configuration file in the Sensu client Ansible role, made up of two tasks which link the Sensu client to the MongoDB instance and the Sensu server. This file is then loaded by the Uchiwa dashboard and allows the instance to appear in the dashboard.
After confirming its presence, selecting the instance allowed me to see all the basic checks that were provided in the Sensu client Ansible role; seeing all returning an "ok" status on a regular basis proved the configuration of the Sensu client in the MongoDB instance was a success (See **Figure 2.1** for this model's architecture).

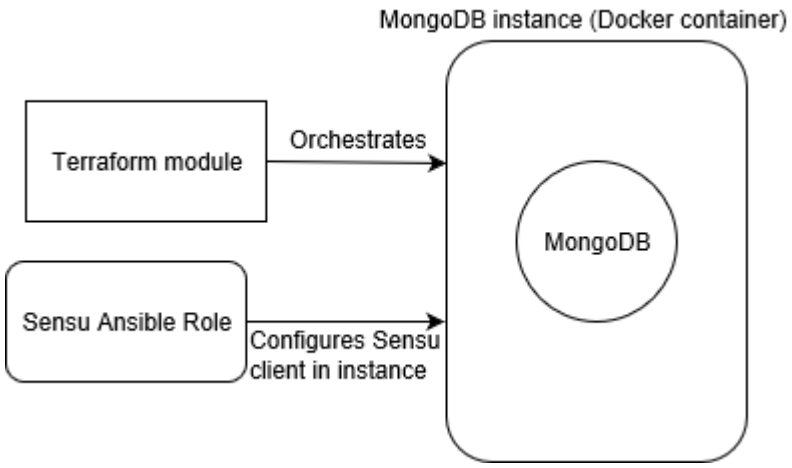

Figure 2.1: Manual addition of a Sensu client in MongoDB instance

However, this solution of manual installation was not optimal: the instance had to be first deployed, then the Sensu client Ansible role pulled locally and manually modified to be linkable to the MongoDB (this part has been automated by making the modifications directly in the role and making sure the MongoDB module correctly pulled the updated version of the role) and had to be executed manually. It was why the team and I decided to directly integrate the Sensu client configuration in the MongoDB Terraform module to gain time and efficiency.

In this aspect, we created an Ansible playbook which does the Sensu client installation and configuration at the same time as the MongoDB instance is orchestrated.

All the previously hardcoded values, such as the IP address of the MongoDB instance, were now written dynamically, gaining time and efficiency in the Sensu client configuration.

The newly updated Terraform module for the MongoDB was then tested as before, by deploying it in Staging and then checking right away the associated Uchiwa dashboard, since the Sensu client was automatically configured at the same time as the instance was being orchestrated (See **Figure 2.2** for this model's architecture).

During this task, some bugs that came up were also fixed: when testing the service locally through a Makefile, we corrected some Linting[6] -based errors that popped up. we fixed one of the tasks executed by the Sensu client Ansible role that consists in installing a gem package necessary for the Sensu configuration. Based on the nature logs we got, we could understand the source of the problem and searched the Internet to see if other developers had a similar problem and had found a solution.

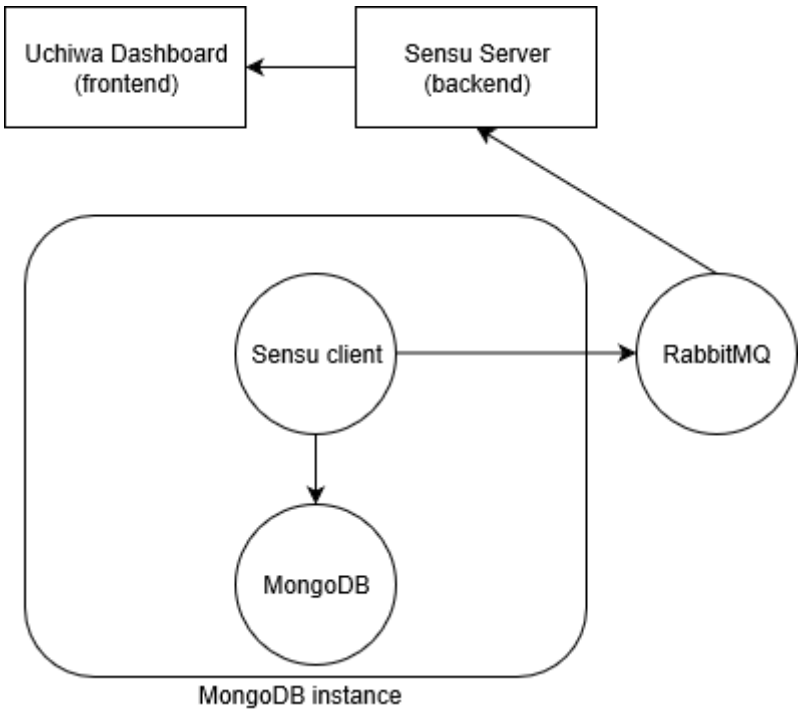

Figure 2.2: Link between the MongoDB instance and the Uchiwa dashboard

[6] Lint refers to tools that analyze code to find programming errors, bugs or stylistic errors.

With the MongoDB instance correctly running with the Uchiwa dashboard showing Sensu checks being done regularly on it, the modifications were pushed in Gitlab through a merge request to get the approval of other members of the team on this task; they also made suggestions or simply created open discussions.

2.2.2 Deploy a PostgreSQL instance in the Production environment

PostgreSQL is the database management system used by AUTO. A new instance of it had to be set up in one of the Production environments.

To that end, we first set up the orchestration of said instance, based on a Terraform module: it created an instance with a PostgreSQL server in it, as well as set up a Cinder volume, a Block Storage solution by OpenStack, and a Sensu Client for monitoring purposes. A superuser is also created (username and password variables) to connect to the server.

A network issue prevented a Sensu plugin from being set up correctly during the configuration. This error was identical to the one we encountered with the MongoDB instance, so we knew it was due to the proxies that were not set up to allow communication with the outside world, for example when trying to fetch a package outside of Ubisoft's network; we fixed it by updating the proxy settings in the PostgreSQL Ansible role that was used by the Terraform module to configure the database and the Sensu client, just like with MongoDB.
Another task was also added to execute in this Ansible role to set up an additional Sensu check for verifying the memory usage of the instance, which was missing.
After having the merge request for these modifications approved, the Sensu client could then be installed successfully during the orchestration.

The next step after installing the PostgreSQL server was configuring roles. We used another Terraform configuration file specifically for this role (the separation allows for better

troubleshooting), which connects to the server using the superuser's credentials created earlier, and then adds two databases and three PostgreSQL roles.

To test that the instance was correctly deployed, we established an SSH connection to the instance and then tried logging into the server with each user created and see if we could access each database created. We also made sure all the Sensu checks were periodically executed on the instance through Uchiwa. The changes were then pushed for a merge request in Gitlab for approval.

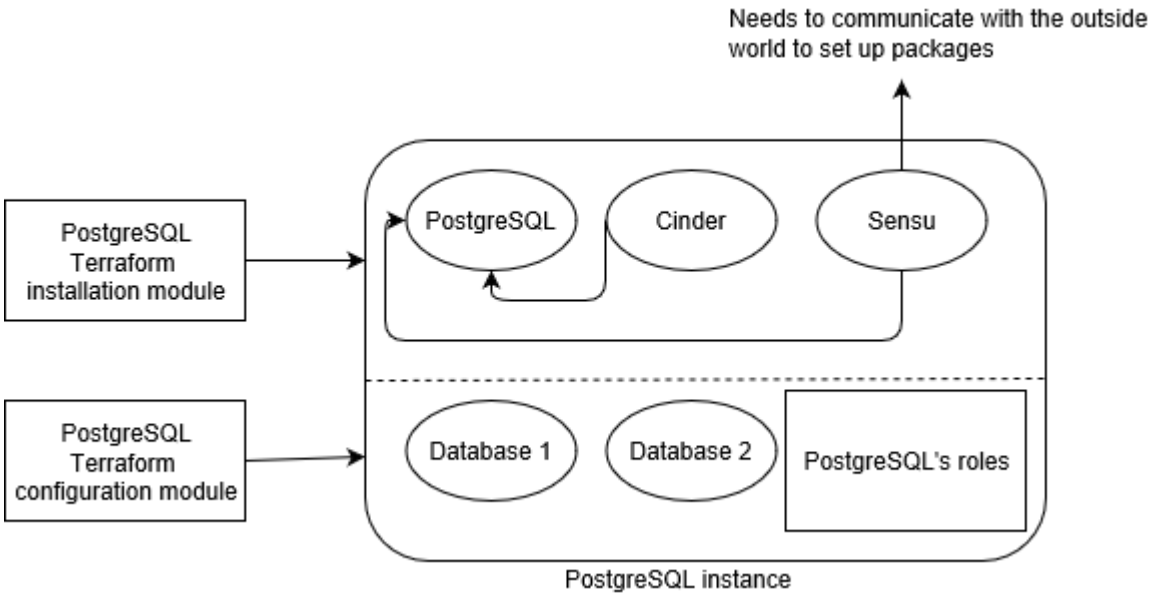

Figure 2.3: Simplified architecture of the PostgreSQL instance

All the usernames and configured earlier were stored as secret variables in Gitlab so as not to hardcode them in the Terraform files and thus increase security. They were also added to a running Vault[7] instance in Production, thus one could only access the server and databases created by connecting to the Vault User Interface to see the variables.

[7] Tool for managing secrets. It can secure, store and control access to tokens, certificates, API keys or any type of secret.

2.2.3 Fix bugs and update the Documentation website

A documentation website had been developed by the team, consisting of an API documentation showing all the available endpoints of currently deployed services, a user guide to explain how to use the platform, and a developer documentation to give more in-depth details about the services of AUTO.
As important as developing and maintaining services was important, making sure there was documentation accompanying it was just as essential. With the constant evolution of the platform, the website had to be consistently updated.

One of the tasks we did was removing all references to Gitlab repositories. Indeed, due to privacy restrictions, we could not give away those URLs to our clients.
The documentation website is brought up with a service named ‘docs’, which creates the base website and its different sections, and then pulls all the documentation from all deployed services.
To accomplish this task, we deployed a local Docker container with the docs service in it. We first sought out the pages with explicit Gitlab references. Once we saw in which service they were, the aforementioned line was removed in the Markdown documentation page of said service (it is the file that is fetched by the docs service), and ran the Docker service responsible for deploying it locally.
To test that my change was correctly done, we re-ran the docs service container who fetched the updated document of a service.
Once all Gitlab references were removed and the testing showed that the deletion was done successfully, we created a merge request to push my modifications on the Gitlab repository.

A non-negligible bug also came up during this iteration: the API documentation section in the documentation website deployed in one of the Production environments, which provides the deployed service endpoints, as well as showed the request and response example bodies,

went down. Indeed, when trying to access it, an error message by ReDoc[8] popped up. Because it specified that it was an error with ReDoc failing to render a spec, that meant the generated data for the API documentation was corrupted.

We deployed locally the docs service in a Docker container, accessed it and extracted all its data under the JSON format.

We then installed and deployed a Swagger validator Docker image on my local machine, in which we tried to validate the JSON data from earlier; indeed, the API documentation is based on the Swagger specification. The validator was deployed locally for security purposes, as sensitive information was in the JSON data, thus using a public validator online could risk exposing secrets.

It was then that the Swagger validator[9] raised some semantic (multiple body parameters not allowed) and schema (parameters having additional properties, expected values different from actual ones) errors in the JSON data.

Being more than one working on it, we fixed the problem by correcting the syntax, adding or modifying values in the validator.

Once all raised errors were gone, we ported the same modifications done in the editor into the services which raised the errors and redeployed for testing the docs service. Seeing the API documentation page was now clean of errors, the updates were therefore pushed for a merge request in the Gitlab repository.

2.3 Iteration v1.2

The second iteration, version 1.2, is centered around two deliverables:

- Creation of a new service to handle load balancer configurations. For understanding purposes, let us call it LB-Redbox,
- Update our services in one of the Production environments.

[8] API Reference documentation based on Swagger.
[9] Service for validating a Swagger-based specification

We will first talk about my involvement with the LB-Redbox service, and then what was done regarding the Production environment update.

2.3.1 The load balancer configuration service: Introduction

The objective was to conceive a service capable of creating, validating and deleting a load balancer configuration for servers and load balancers: for easier understanding, we will name the service encompassing both the load balancers and the servers a Redbox. It would encompass the functionalities of two already existing services, which were already responsible for doing the validation check on the configuration, as well for configuring and managing a load balancer.

Considering Redbox which is made up of a set of Windows servers, the load balancer must be configured such that the incoming load is equally distributed among these servers.

I mainly took part in creating the CREATE logic of the service.

2.3.2 The load balancer configuration service: Implement the CREATE logic

The way the Load balancer configuration works is as follows: given Redbox server nodes (i.e. the Windows servers), resources are created for each of them and are bind together under a service group. Given the load balancer itself, a resource for it is also created and is bind to the previously created service group (See **Figure 2.4**).

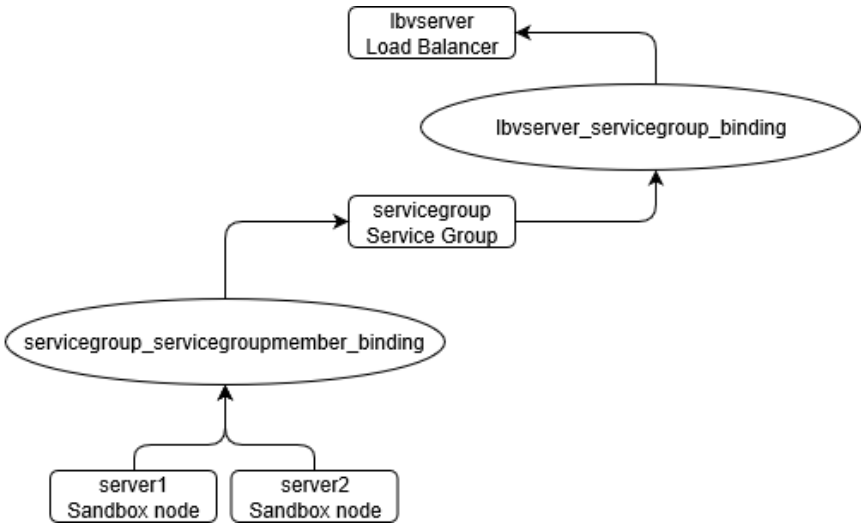

Figure 2.4: Load Balancer configuration architecture

For the implementation of this service, the team agreed to rely on the single responsibility principle in 3 layers (See **Figure 2.5** for a better understanding of the architecture):

- Presentation layer: the API; corresponding to the entry point of the service, it is responsible for receiving the request payload and sends it to the next layer, the Controller,
- Business logic layer: the Controller; receiving the request payload from the API layer, it is responsible for handling the domain logic and redirects the payload to the Adapter depending on the /endpoint he received,
- Data Access layer: the Adapter; the final layer, it is the one responsible for making the external library calls and CRUD operations.

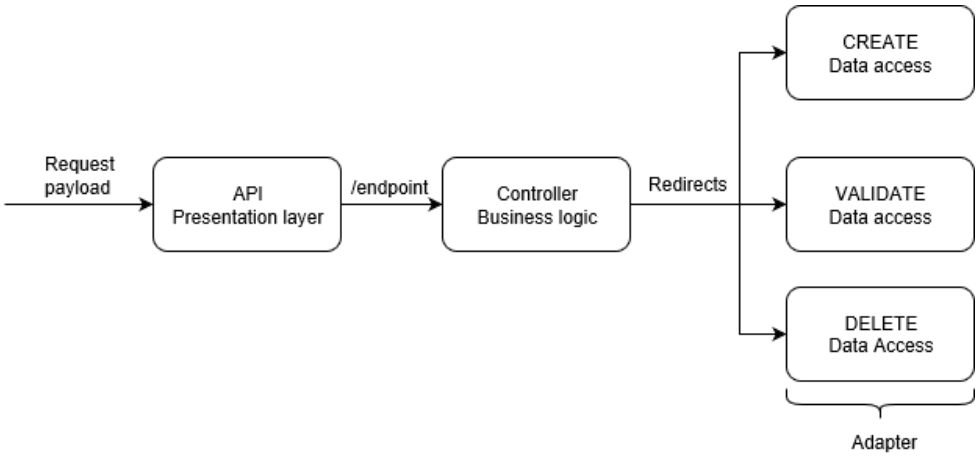

Figure 2.5: Single Responsibility Principle architecture for the LB-Redbox service

To create a load balancer configuration, a user sends a valid request payload (contains information about one or more Redbox's load balancers and servers) to the service, via the Kong gateway. After the authentication and authorization process, the service retrieves credentials from Vault, which are used to transform the load balancer's public IP into a private one, which is then used to create the configuration payload using an updated load balancer service developed by a member of the team. The configuration payload is then sent back to the user.

We started with the lowest layer of the architecture, namely the adapter for the CREATE logic, as it is where all the work is done.

At this stage, the adapter should have received the request payload and the private IP address of the Load balancer to configure from the controller.

We defined a class (suppose it is called "lbClient") which corresponds to the load balancer service we are using and will be the class that will create the load balancer configuration payload at the end.

We first crafted an intermediate payload, that we will call Redbox load balancer definition payload. It gives information about the client Redbox load balancer server to configure, the associated Redbox servers and the service group resource to create: they make up the Redbox.
This payload is built using a configuration template we created serving as the skeleton, a Redbox instance in found in the load balancer service given the private IP, as well as the cluster host in which is located that Redbox.

The load balancer configuration is then created using the Redbox load balancer configuration definition, for each Redbox defined. It is done through a method from an external library. It creates an empty dictionary and fills it with the data coming from the Redbox load balancer configuration definition payload. The advantage of knowing whether this payload was

correct is because of a sub-dictionary named “errors”, that was not empty so long as errors were raised.

The CREATE logic for the adapter being done at this point, we now had to add unit tests to make sure all the functions we created worked like they are supposed to and failed when they needed to.
With the other members working on the other endpoints, it was decided we would rely on three technologies: fixtures from the pytest framework, the mock object library for testing in Python and the flexmock library.

Mock objects are simulated objects that mimic the behaviour of real objects. Based on the “action → assertion pattern” (run some tasks, then assert the results), they can be very useful when using a real object is impossible or too difficult to integrate in a unit test.
Flexmock is an improved testing library for Python and is also a way to generate fake objects on the fly. What makes it very useful and simple to use is that, given an object with several methods, it can stub the methods and replace them with fake ones. You can even dictate the behaviour of a function, from the methods it calla to the values it should return, making it very intuitive to use.
Fixtures help set up the system by providing the necessary code to initialize it. For example, using the context of the LB-Redbox, a fixture could be setting up valid Vault credentials for use by different methods requiring it, such as for when getting the Redbox instance in the load balancer service using the private IP. They not only allow reusability, but also prevent us from using real data and modifying them; a fixture’s data will always have the same setup every time it is being run.

We started from the beginning of the CREATE adapter to build the unit tests: The first thing defined is the “lbClient” class itself, and because it would be reused often since it is called by all its methods, a fixture instantiating an instance was written, with all its attributes being flexmocked. This fixture could then be used in any test functions involving a “lbClient” instance.

Mock payloads were created for test purposes and became fixtures for reusability when doing assertions. We created the request, Redbox load balancer definition and Load Balancer configuration payloads, both valid and invalid (by removing a key-value for example).
Every single function defined in the adapter was tested, taking all possible results we could expect, successful scenarios and failed ones alike.
See **Annex I** for an example of how the problem was tackled when considering a function made up of two methods.
To help in raising the expected errors in the tests, we created Exceptions specific to the type of error we intend to get (for instance, if it is due to a Vault connection issue or a load balancer issue). One exception would equal to one fail test. They would later inherit the APIException class to raise the classic HTTP errors, as they will be required by the middleware to show the type of HTTP error raised when a client will use the service, and something goes wrong.

Running the tests using Pycharm[10] 's integrated unit tester, the logs would help me fix any test that did not finish how it should.

After making sure that all the functions in the adapter were covered by a unit test, we added docstrings to describe each function's role, their parameters and the return value for documentation purposes, and then started to work on the CREATE logic for the controller.

Made up of the "lbController" class, which has the load balancer and IP translation services as attributes, the controller has 3 functions corresponding to the CREATE, VALIDATE and DELETE operations of the service. I only concentrated on the first one.
Only two methods were called with the CREATE logic:

- Get the load balancer's private IP by passing the public one in parameters,
- Create the Load Balancer configuration payload.

[10] Integrated Development Environment for the Python language.

The unit tests for each method were already done through the adapter, in consequence there was no need to test them again here. The only testing that was to be done was the CREATE logic encompassing the two functions. This meant 4 possible scenarios:

- the Load Balancer configuration creation succeeds,
- the private IP fetch fails (therefore the remaining method is not tested),
- the private IP fetch succeeds, but the Load Balancer configuration creation fails, and because there are two possible exceptions that can be raised (load balancer service issue or configuration creation issue), that meant one fail test for each.

The mocking and usage of fixtures followed the same principle as what was done with the adapter.

Finally, the last part to do was working on the API layer for the CREATE logic. The implementation was the simplest of them, as you simply had to fetch the controller, and then make a call of the CREATE function on it.
The main difference was regarding the tests. Instead of doing unit tests, we transitioned to doing functional tests, which instead of examining the internal structure of an application, considers it as a black box; we only send it an input and expect an output. This is because the API is at such a high level compared to the controller and adapter (see **Figure 2.6**).

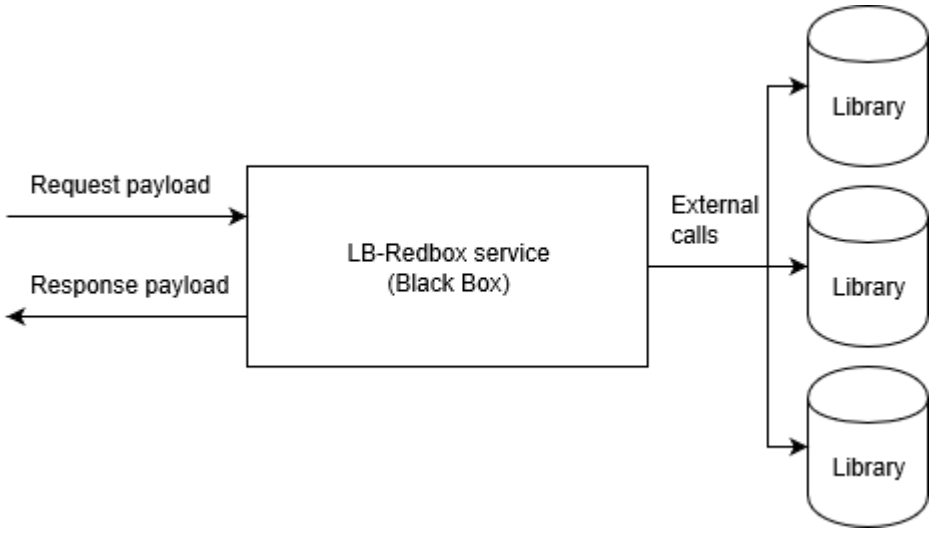

Figure 2.6: Black Box concept for the API layer functional tests

The tests that we made consisted in seeing if a Load Balancer configuration payload was successfully created, and all the failed creation scenarios, which consisted of all external calls made by the whole service that raised an error. These were:

- Not finding any cluster in the load balancer service to which the private IP belonged to,
- Finding an invalid cluster in load balancer service in the same context as the previous scenario,
- Finding a valid cluster but not the associated traffic domain, an essential parameter for the service group,
- The generation of the Netscaler Redbox definition payload fails,
- The Load Balancer configuration has the "errors" dictionary in its payload.

The fixtures that were created all mocked the external services (such as Vault) used by the LB-Redbox service.

With this final step done, the CREATE logic of the LB-Redbox service was essentially done. All the changes had been progressively pushed into merged request over time.

2.3.3 Test the LB-Sandbox service, documentation and bug fixes

Once the VALIDATE and DELETE logic were also done by other members of the team, we could finally deploy the service in Staging and start testing it.

We used the Swagger service, as well as an external software called Postman[11], to test each endpoint.
Using a valid request payload with real data, we tested all 3 routes, making sure that it returned a 200 or 201 HTTP response with a response payload to confirm it. We also modified the request payload to make sure that the right exception was raised for each of the

[11] API development environment for testing APIs CRUD operations.

test cases that we created, such as if the service could note change the load balancer's public IP to a private one, or if we missed the required permissions.
We also checked some unusual behaviors that could happen and that could not be really tested, such as what would happen if we tried to create the configuration with only one parameter being changed each time (would it create a new configuration? Would it raise an "already exists" error?). The service had never given an unexpected answer; therefore, the service could be considered as fully functional, ready to be deployed in Production for actual use by the client who requested this service to us.

We also updated the service's documentation to display it in the website generated by the 'docs' service, mainly explaining how to use the different endpoints in the User Guide.
The API documentation in Staging went down at one moment. When we checked the website and read the error, we understood it was due to a JSON component, "schema", that could not be loaded properly in the loadbalancer service; 3 endpoints were impacted by it. This bug was similar to the one that was mentioned in chapter I.4., therefore the same process was executed: deploying locally the "docs" service, extracting its data in a JSON format and analyzing it in a locally deployed Swagger Validator; this showed there were semantic and schema errors. To solve these errors, we removed the schema and their parent component and instead specified a predefined payload.

2.3.4 Updating AUTO services: Validate backward compatibility in AUTO

In AUTO, there were two services with up to date and deprecated routes still working:

- api/<service_name>/<version>/<endpoint>: correct, updated route
- api/<version>/<service_name>/<endpoint>: deprecated route

The issue here is that the deprecated routes are still being used by our services. To this end, we worked mainly with a service named Discovery, which is responsible for dynamically finding all running services, along with their definition, health, documentation and routes, in

a Docker swarm. It would then store them in a Consul[12] store and finally send the updated list of running services to the Kong gateway, that takes care of updating its own list.

However, the deprecated routes were not dynamically found, and so we had to statically configure them in the Discovery service, so that they would always be recognized by Discovery, and the gateway would therefore always have these deprecated routes.

We edited a JSON file which is called by Discovery and holds all static configurations for the service; we looked for a dictionary corresponding to the deprecated services to keep in the gateway, and added there the two services: We specified their name, their deprecated URI, as well as the upstream URL associated to the deprecated routes; a specific port also had to be mentioned.

To test that the modifications were correctly done, we deployed a local Discovery service with those updated modifications, as well as a Consul and Kong instances. We also added some logs to the route of each service to make sure that calling either the new or deprecated endpoint would lead to the same result. Deploying them to the same stack, we then applied a curl command to the gateway calling each endpoint for the verification.

With the tests being successful, we pushed the modifications to the Gitlab repository for a merge request approval.
This task has helped me better understand the way how the services are registered in the Kong gateway, and it also aided me in learning how to use the Discovery and Kong services.

2.3.5 Updating AUTO services: Configure the Staging environment

One of the remaining main tasks we did for this iteration was configuring the majority of the Staging environment.

---

[12] Key-value store used to register the running services and their properties found by Discovery. Removes unused ones.

To accomplish this, there was a breakdown of the different tasks to do in order:

- Create a Terraform file to orchestrate the Staging environment,
- Create new floating IPs mapping for the new the Staging environment,
- Create a HAProxy for the Staging environment.
- Create blue and green clusters in the Staging environment and test the switching between both.

Knowing we relied on the blue-green deployment solution[13], we first had to create two Terraform files that orchestrate the cluster in Staging, one that would be the Blue environment, while the other would be Green.

In each Terraform file, we orchestrate the deployment of one manager and two workers for load balancing purposes when switching between the blue and green clusters. We specified the name, dependant on the blue-green environment, the environment in which the cluster were to be created (in other terms, Staging in this case), as well as defined a Cinder volume. We also needed a way to know which color was currently active, to be able to do the switch successfully: The color was saved in a file, which would then be updated every time a switch happened.

The next step was then to create 6 floating IP addresses and map them to the blue and green cluster's managers and workers. They were created using a Terraform module for generating these.
The values were then stored in a file and assigned variables, themselves used to parameterize the IP address of the managers and workers: once deployed, the clusters would have those floating IPs.

[13] Technique that reduces downtime and risk by running two identical environments called Blue and Green. At any time, one of the environments is live, being the actual production environment, while the other is idle and can be used to develop and test a new version.

At that point we tried to deploy the cluster in Staging (the one to deploy depended on the current colour present in the state file mentioned earlier) to test whether they were correctly orchestrated with the right floating IPs assigned to them, however we encountered at that point an SSL certificate verification error. With some help from a team member, we found out it was due to the CentOS Docker image used as a basis for the cluster. Even though the Dockerfile related to the parent image correctly specified the necessary certificates and added them in the Docker image corresponding to the blue/green cluster, nothing was done with them; in other words, while they were there, the container did not do anything with them and thus did not use them for the SSL certificate verification.
To solve this problem, we manually force updated the authentication store within the blue/green image. This fixed the issue and we were able to correctly deploy one of the two clusters. It is however a temporary fix, and it has been discussed that the CentOS Dockerfile image had to be modified with the authentication store update, so that we do not have to redo the operation manually in the future.

With the deployment of the cluster happening as wanted, and after checking that the managers and workers had their floating IPs correctly assigned to them, we proceeded to the next step: creating the HAProxy, an open-source software providing a high availability load balancer, for the 2 clusters, which will be required to safely do the switching.
This was simply done by creating another Terraform file as well, which relied on an already defined Terraform module to set up a HAProxy.

Finally, the last step to set up the cluster in the Staging environment is to create the cluster and test the switching; this would also test the correct configuration of the HAProxy, as the load balancer would help in making the switch correctly.
The script file for switching the clusters was already done. Another team member had added a solution on top of it which consisted in dockerizing the blue/green switch: Instead of manually deploying a cluster and then call manually the switch-cluster script, the cluster switching is also done at the same time as the cluster deployment, automatically.

We therefore made the test by using the blue-green dockerization method, and then checked the currently active cluster by fetching the state file showing the current colour of the cluster. After fixing a quick issue causing the switch to not work if the cluster did not exist in the first place (an Ansible task called by the switching cluster bash file responsible for cloning the state of one cluster in the other before the cluster switch failed if neither of the blue or green cluster were first deployed; solving it would simply require skipping the task if this situation happened), the switch was successfully done, and all modifications were then pushed into a merge request in the Gitlab repository for approval.

2.3.6 Updating AUTO services: Deploy in the Staging environment

With the new Staging environment now up and running, the remaining thing to do was redeploying the basic services of AUTO.

To deploy each service in the Staging environment, we configured a file named ".gitlab-ci.yml", which is the file used by the Gitlab runner to manage and execute the project's job. We added a stage for deployment the Staging environment which would first fetch the IP address of the cluster's current manager to know where the AUTO stack, if it exists, is and update it with by deploying this service.

The IP get is done by making a curl command to an AUTO service which gives a state of a particular environment in a JSON format and also lists the running services, as well as the IP address of the manager of the currently active cluster. It would simply create the stack and then add the service otherwise.
It would then export a Transport Layer Security verification to check that the communication over the network is secure, export the path to the certificates in Staging necessary when trying to fetch the cluster manager's IP address (without those certificates, you cannot get this IP address as a trust issue would be raised) before actually deploying the service in the AUTO stack present in the Staging environment, using the manager's IP address to find the

stack's location. We would also specify the environment to have a description of where the service is being deployed.
The deployment in the Gitlab CI stage is done by doing a 'docker stack deploy' command, which creates or updates an existing stack, and using a docker-compose file for deploying it in the Staging environment.

To test the correct deployment of a service, we would specify in the ".gitlab-ci.yml" file to run the pipeline with 2 stages only, one that builds the Docker image, and one which deploys the service with it in the Staging environment. This has been done by mentioning that it should only run in the working branch in which we did my current modifications:
As such, when pushing my branch to Gitlab, the runner would then run the pipeline for this branch and do the deployment. After the pipeline succeeded, checking the success of the deployment usually relied on connecting to the Kong gateway in the Staging environment and seeing whether the service was there; an extra verification could be made by connecting to the currently active manager (through SSH) and list all the running services and check if the one we are looking for is there.
The actual final deployment would then be made after my working branch would be merged with the master branch, which runs its own pipeline; in other words, after having my modifications approved.

While adding the new stage for deployment in the Staging environment in the ".gitlab-ci.yml" file, we also updated the global variables set for making the Transport Layer Security and certificate verification in Docker as their old values were deprecated and had to be updated with the new ones that they should be pointing to: this was done by creating new secret variables stored in the Gitlab repository with the correct values (for the certificate verification, this corresponded the new path where the certificates were stored in the Gitlab runner used to run all pipelines for our projects).

2.4 Iteration v1.3

2.4.1 Generate Gitlab pages for AUTO services

The last task done before the end of the internship was generating Gitlab pages for the AUTO services for documentation purposes.
Gitlab pages are simply stage websites for Gitlab projects. For our case, they are documentation on all functionalities of a given service and are generated using “sphinx”, a documentation generator.

The first service to update was the Cookie Cutter service, which serves as a boilerplate to create an AUTO microservice; all AUTO services follow the same architecture.
There were previously 2 ways to accomplish this task: either generate the pages within a Docker container running on a sphinx Docker image, or within a tox environment in a Docker container running this time on the service’s own Docker image. We ended up choosing the latter, as it was easier to set up.

We followed the same architecture for the other services requiring the Gitlab pages.
These modifications were then tested locally by first building the image and then running a Makefile rule executing the sphinx commands in the tox environment within the container. It was then tested in the Staging environment by running the Gitlab runner and checking that the URL for Gitlab pages for this service was up in said environment.

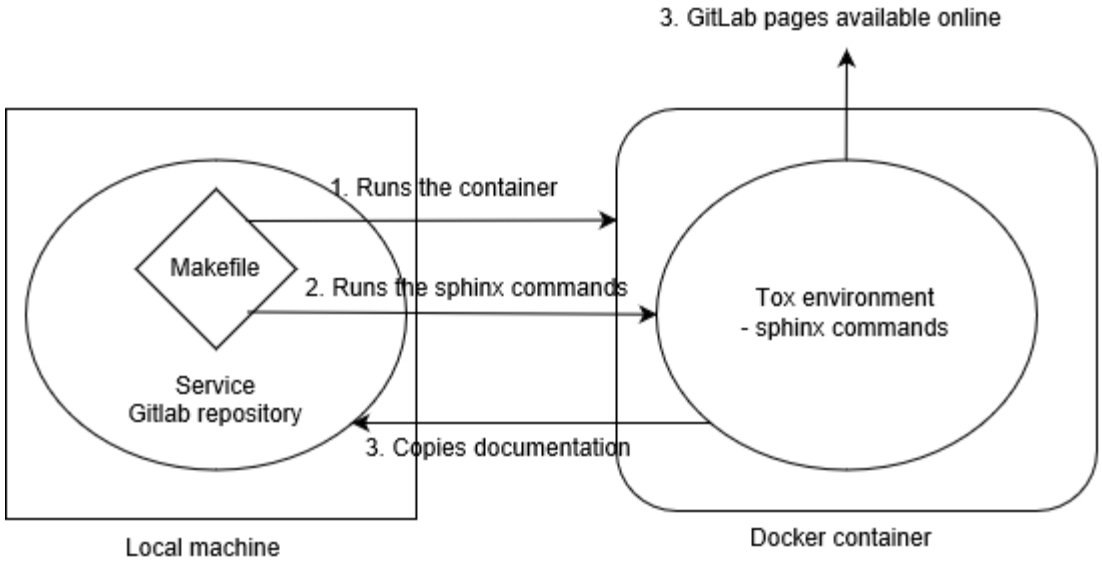

Figure 2.7: Generating Gitlab pages using a tox environment in a Docker container

CONCLUSION

During this internship at Ubisoft Montréal, I have taken part in developing an infrastructure automation API platform, called AUTO. With the version v1.0 being released at my arrival, I started working on the next iteration.

Several features were already requested to be added to the service throughout a year long plan. The next step was checking which ones we could commit to deploy in our API. Relying on Agile methods such as Scrum and Kanban to plan and track task progress during an iteration, we have released several new additions to our product:

I have added automated the installation and configuration of a monitoring solution to a document-oriented database system, with the automation process being reusable for any other service which may be in need for such a thing. I have also deployed a database system management in the Production environment to replace the old one and updated the basic AUTO services in the Staging environment that has also been reconfigured to adapt to the new gateway that allows access to using AUTO's various services.

One of the most important tasks realised during the internship was creating a new service capable of creating, validating and deleting load balancer configurations, for which I implemented and tested the CREATE logic.
Throughout the three iterations I have been part of, I have also fixed bugs that occasionally came, whether they came from the services or the platform itself and have created documentation for the API platform's services.

I have applied all the theoretical knowledge I acquired during my courses at ÉTS for the internship, such as project management, DevOps, the Agile methods and Scrum framework, as well as Quality Assurance.
More importantly, however, is the huge amount of knowledge I assimilated during the internship, learning a great deal about cloud computing in general, several orchestration and configuration tools, and even higher-level notions, like new methods of developing and testing a software application.

All in all, this internship has brought so much, and I will definitely use what I learned in the foreseeable future.

ANNEX I

EXAMPLE OF TESTING A FUNCTION IN THE CONTEXT OF THE TESTING DONE FOR THE CREATE LOGIC ADAPTER; WE CONSIDER A CLASS WITH TWO ATTRIBUTES ASSOCIATED TO OTHER CLASSES, AND A FUNCTION MADE UP OF TWO METHODS

```
### src/adapter.py file ###
class Class1(): # Main class of the adapter
    def __init__ (self, arg1, arg2)
    self.attr1 = arg1
    self.attr2 = arg2

    def function1(arg1, arg2): # Main function of adapter
        res1 = self.attr1.function2(arg1, "arg2")
        return self.attr2.function3("arg2", res1[0])

    def function2(arg1, arg2):
        [...]
        return result

    def function3(arg1, arg2):
        [...]
        return result

### tests/units/conftest.py file à where all fixtures and mock objects are created ###
from package1 import ObjectClass
from package2 import ObjectClass2

@pytest.fixture
```

```
def fixture_class_instance(): # We mock an instance of the main class, along with its attributes
   [...]
   return Class1(flexmock(ObjectClass1), flexmock(ObjectClass2));

@pytest.fixture
def fixture_input_payload(): # We create a fixture of the input payload
   [...]
   return input_payload;

@pytest.fixture
def fixture_output_payload(): # We create a fixture for the expected output payload
   [...]
   return output_payload;

### test/units/tests.py file : where all tests are written ###
def test_function1_successfully(fixture_class_instance, fixture_input_payload, fixture_output_payload):
   (flexmock(fixture_class_instance.attr1)
    .should_receive("function2")
    .with_args(fixture_input_payload, "valid_value")
    .and_return(result1))

   (flexmock(fixture_class_instance.attr2)
    .should_receive("function3")
    .with_args("valid_value", result1[0])
    .and_return(fixture_output_payload))

   final_result = fixture_class_instance.function1(fixture_input_payload, "valid_value")
```

```
    assert final_result == fixture_output_payload

def                 test_function1_fails_at_function2_invalid_payload(fixture_class_instance,
fixture_output_payload):
    (flexmock(fixture_class_instance.attr1)
     .should_receive("function2")
     .with_args(corrupted_payload, "valid_value")
     .and_raise(Error1, "The input payload is corrupted"))

    with pytest.raises(Error1) as exc:
        fixture_class_instance.function1(corrupted_payload, "valid_value")

    assert type(exc.value) is Error1

def                 test_function1_fails_at_function2_invalid_value(fixture_class_instance,
fixture_input_payload, fixture_output_payload):
    (flexmock(fixture_class_instance.attr1)
     .should_receive("function2")
     .with_args(fixture_input_payload, "INVALID_VALUE")
     .and_raise(Error2, "The value given is invalid"))

    with pytest.raises(Error2) as exc:
        fixture_class_instance.function1(fixture_input_payload, "INVALID_VALUE")

    assert type(exc.value) is Error2

def      test_function1_fails_at_function3(fixture_class_instance,      fixture_input_payload,
fixture_output_payload):
    (flexmock(fixture_class_instance.attr1)
     .should_receive("function2")
```

```
    .with_args(fixture_input_payload, "valid_value")
    .and_return(result1))

(flexmock(fixture_class_instance.attr2)
    .should_receive("function3")
    .with_args("valid_value", result1[0])
    .and_raise(Error3, "result1 is missing parameters"))

with pytest.raises(Error3) as exc:
    fixture_class_instance.function1(fixture_input_payload, "valid_value")

assert type(exc.value) is Error3
```