INSTITUTO TECNOLÓGICO DE BUENOS AIRES – ITBA
ESCUELA DE INGENIERÍA Y GESTIÓN

Coding Evaluation Platform

Desarrollo de una plataforma online de aprendizaje para la evaluación de ejercicios de programación

Authors: Lobo, Daniel Alejandro (Leg. 51171)
Bellini, Juan Marcos (Leg. 52056)

Tutor: Meola, Franco Román

A thesis submitted for the degree of
SOFTWARE ENGINEERING

Place: Buenos Aires, Argentina
Date: December 6, 2019

Contents

Abstract

The following work is the final project of Daniel Alejandro Lobo and Juan Marcos Bellini, both students at the Instituto Tecnológico de Buenos Aires (ITBA), for the Software Engineering career.

The purpose of this document is to explain with as much level of detail as possible the different edges of this final project: a coding evaluation platform to be used by both teachers and students of the same university (or any other given institution), and also by any other developer using the platform without sitting in an exam.

The entire project consisted in planning, designing and creating an entire (micro) services platform. From a project management and software engineering perspectives, it involved: infrastructure, frontend, backend, databases and deployment.

The decisions made and the reasons behind them, the methods used and their implementation, the results obtained and their conclusions, are detailed in the following sections.

1 Motivation

How should we assess programming skills? Asking students to write code in a traditional hand-written exam can produce a wide variety of results both for the teacher and for the student. We, as students, have felt this awkward experience more than once when sitting on a written programming exam for the subjects that we have done during university.

More often than not, from the teacher perspective it is nearly impossible to meaningfully grade such code. The code might be crossed over by the student many times either on pen or pencil and the lines of it might be hardly indented as well, compared to what one could see on a typical text editor with syntax highlighting on your platform of choice. With sufficient effort one can get some idea of whether the general idea is correct, but to assess programming skills and real knowledge of the taught concepts in the subject we need much more than this.

From the student perspective, and we think one of the best gains relies here, is hard to translate the mental speed one combines when programming on the computer and typing those lines on the computer keyboard than to do it with pen and pencil. It actually requires twice the effort when studying: once while coding away on the computer (either for the exam or for personal projects) and then coding on a piece of paper simulating a real life scenario when sitting on a classroom during an exam.

At the same time, when talking about the contents of the student's submission, for example, there will almost certainly be errors in the code; how do we know whether the student would be able to correct those small errors or not? Few students (and actually programmers as well) get their code correct on the first try—testing and debugging is an integral part of the programming process or, like some people call it, 'programming flow' when solving a specific problem. To assess programming skills properly, we should provide students with an authentic programming environment in which they can develop and test their code. Only then is it fair for us to run their code and use correctness tests as our measure of their ability and understanding of computer science concepts we are evaluating them on. Not only are examinations the only kind of assessment a student has throughout their course. In introductory programming courses we usu-

ally also assess laboratory and assignment work.

From the students point of view, unlike a classic programming exam evaluated on paper, the student gains immediate feedback by submitting the code and seeing if the submitted answer works for the public test cases or not.

Some of the use cases students and teachers can give on an educational context to our platform throughout taught courses include: laboratories, tests, assignments, and even final exams. Assessment, grade recording, and distribution of course material are thus all consolidated on the same platform. Thanks to the use of an intelligent editor that supports several programming languages, our platform is very flexible in terms of the type of question that can be asked and even the form of feedback displayed. An added benefit of the consolidated approach we took is that students do not need to set up an entire environment for either Java, Ruby or C on their machines, so that the stress of online tests and examinations is considerably reduced.

We thought of a platform able to support any text-based programming language. Built-in question types are available for C, Java and Ruby. It is really simple to include other programming languages to the platform. At the same time, it offers integration with learning management systems (LMS) like Blackboard through LTI, an industry standard to communicate to such systems.

We wanted to build a platform that is particularly well suited to introductory programming courses, for which students need lots of practice with small programming problems that teach the different language concepts and techniques taught during the course.

Any student can, during an exam or just in practice mode, submit an entire file in either Java, C or Ruby and run it against a specific code sandbox created for the purpose of running code snippets of that particular language. Once the code snippet is run in the code sandbox and the resulting program is compiled and run, then the output from that run is compared with the expected output to determine if the code passes particular public and private test cases. For security reasons, we separated the execution of

the submitted code by the student on a separate machine (in our servers).

In a nutshell, having an online exam rather than a hand-written one can also provide some interesting insights. Some of these could be statistics on how students perform on each question, so we can immediately spot topic areas that might warrant further teaching effort in the future and also questions that prove problematic (e.g., that appear to discriminate against good students). Another useful statistic that might come in short notice is the ability to see which questions are answered before during the duration of the exam. At the same, any kind of data can be extracted from the database if needed once the exam (or semester) is over, in order to evaluate and iterate the way the programming topics were taught and then evaluated.

2 State of the Art: similar platforms

2.1 Introduction

At the very beginning when starting this project, one of the main things that we have done is evaluating the state of the art of the similar initiatives that we have found all over the Internet. Some of these projects have a business origin (i.e. to generate money), but most of them are academic initiatives or made for the sake of creating programming competitions.

Regardless of their genesis, they all share similar features and show a wide array of issues and different perspectives to solve the needs of the different end users. The most descriptive solutions and approaches that we have found are detailed below.

2.2 HackerRank

HackerRank is a technology hiring platform that is the standard for assessing developer skills for over 1,000 companies around the world. By enabling tech recruiters and hiring managers to objectively evaluate talent at every stage of the recruiting process, HackerRank helps companies hire skilled developers and innovate faster.

HackerRank allows solving challenges in different languages. For universities and schools it simplifies how you create, manage and grade programming assignments. It lets you invite students to test and submit their code in a true coding environment. It also lets you auto grade and evaluate student performance. They also have a plagiarism detector that will flag any submission that is > 70% similar to other students. Finally, it also provides different kinds of competitions.

HackerRank is probably one of the most famous platforms and also includes many features that universities, companies and individual software developers might look after when choosing one of these alternatives. It also offers a nice and modern user interface with proper user feedback.

One of the things that we have found interesting about HackerRank is its Leaderboard. Programmers are ranked globally on the HackerRank leaderboard and earn badges based on their accomplishments to drive competition among users. HackerRank is part of the growing gamification trend within competitive computer programming and the consumer-side of their website is free for coders to use.

Another key feature it offers are the exercises discussions. Each challenge has a Discussions tab where you can collaborate with fellow programmers who've attempted that challenge. When asking for help, be sure to provide detailed information that will enable others to understand what you're talking about.

On that regard, at the same time, exercises are organized in a way that can be used to learn or sharp a skill in particular making the most out of the discussions one can have per challenge. This also helps in a sense of community creation and we personally think it's a great value proposal for HackerRank.

At the same time, HackerRank offers a user ranking, called scoring where users can earn the following badges by solving challenges on HackerRank: either by problem solving (algorithms and data structures), language proficiency, or specialized skills, like SQL.

Another possibility HackerRank offers is the possibility to create challenges and contests. Some of the topics they cover are Algorithms, Artificial Intelligence, Distributed Systems, Databases, Mathematics, Cryptography and Security and Language Specific Domains. Some of the things one can include when creating a challenge is: its name, the description, the problem statement, the input format, the constraints, the output format and the sample Input/Output together with an explanation of them.

Most of the important features raised here are available on the free version. However, on the premium version, one can find instantly detection of code plagiarism, a dedicated account manager and personalized onboarding of candidates, integration with the Applicant Tracking System and access to a premium library of coding questions or create a personal one.

2.3 CodinGame

CodinGame is a technology company editing an online platform for developers, allowing them to play with programming with increasingly difficult puzzles, to learn to code better with an online programming application supporting twenty-five programming languages, and to compete in multiplayer programming contests involving timed artificial intelligence, or code golf challenges.

CodinGame also serves as a recruiting platform, allowing developers to get noticed by companies based on their performance on the contests.

We have found it has really great insights and resources for developers to learn and improve their skills while using the platform. It also has a great learning community that has been around for at least five years and it also counts with a wide array of different types of practices to test different skills and programming languages.

Most of the important features raised here are available on the free version. However on the premium version, one can find a library of 1500+ tasks, Junior / Senior / Expert level questions, unlimited custom questions, cheating protection, a dedicated account manager, multiple accounts / teams and ATS integration API access.

2.4 Codility

Codility is a software platform that helps tech recruiters and hiring managers assess their candidate's skills by testing their code online. It offers free global coding challenges and lessons to help candidates prepare. Codility also has a subproduct that is valuable for companies, called CodeTrain. It lets you audit the capabilities and assess skills gaps of your development team or outsourced talent. This information can be used, for example, to make promotion decisions, inform tech hiring and on-boarding strategies, and ultimately, enhance engineering workforces.

The most important features that we have found in the entire platform are the support availability (they have 24/7 for possible bugs and issues while submitting a solution), an important focus on the performance of the code sandboxes that run the code and of their entire web application. They also have a significant number of practices and challenges to offer to the user. At the same time, they have a long list of integrations and a plagiarism software.

Most of the important features raised here are available on the free version. However on the premium version, one can find a full platform access, single sign-On via SAML, full Codility task library, a custom task library, full anti-plagiarism suite, enhanced security, integration ecosystem, a developer toolkit, support and a services package.

2.5 DevSkiller

Devskiller is an automated online programming skills assessment platform. It helps to test candidates' abilities to develop in a target environment before the interviewing process.

At an overall point of view it does seem to have more complex and more variety of coding challenges that products like HackerRank. Some of the features that we have observed is the live code pair with video, the amount of different programming languages, frameworks and libraries. It also has a diverse amount of question types, for example it lets you evaluate with multiple choice, database, code review, programming task, code gap, and essay questions.

One feature that it highlights from other alternatives that we have analyzed so far is an included natural programming environment (Browser IDE). It also counts with a plagiarism system integrating with other APIs and screen customization.

Most of the important features raised here are available on the free version. However on the premium version, one can find DevOps testing, code

pair with video interview, session recording and playback, AI Benchmarking Engine, Divisions / Teams, coding contest support, a personal account manager and single sign-on (SSO).

2.6 CodeRunner

CodeRunner is a free open-source question-type plug-in for Moodle (one of the largest open-source learning platforms that can run program code submitted by students in answer to a wide range of programming questions in many different languages. It is intended primarily for use in computer programming courses although it can be used to grade any question for which the answer is text. It is normally used in Moodle's adaptive quiz mode; students paste in their code in answer to each programming question and get to see their test-case results immediately. They can then correct their code and resubmit, typically for a small penalty.

It has several integrations with other services that support a wide variety of programming languages. At the moment of testing we liked a lot their usage of public and private tests being run when the code is submitted and the instant feedback given to the user. The offered features on CodeRunner are free.

2.7 Conclusions

Having considered all these services and web applications we decided to focus mainly on offering a neat experience to the user to run their code snippets in, initially, three different languages: C, Java, Ruby. These languages are the ones taught in the first years of ITBA courses and are the ones known by the students that will be using the platform for the first time. They can be thought of as the first real users of this platform. The user would be able to run them anytime by using the application's playground mode without needing to install any kind of dependencies or set environmental variables or any kind of configuration.

On the other hand, we opted to enclose this same playground on an evaluation platform, where teachers can create exams with exercises of

either one of these programming languages, together with public and private test cases, in order to be run when executing the solutions provided by the students. At the same time, these exams end up having a score based on the individual score each exercise have. The score each exercise has is correlated to the test cases they passed (or not) depending if they match the program's output matches the output suggested by each test case.

Ultimately, the main goal of this project is for it to be a free application used in academic environments supporting different programming languages with the possibility of offering teachers the ability to create evaluations and evaluate exams.

3 Functional Requirements

The following sections list the functional requirements that we aimed for when starting this project as a group of features that we considered eseential for the platform as a whole and that we included in the first version of this project. The sections listed below are considered, from a product point of view, the different domains of our project at the moment of launching the first version.

3.1 Identity Management

3.1.1 Authentication

1. Allow students enter the platform when they are signed into Blackboard.
2. Allow teachers log into the platform.

3.1.2 Authorization

1. Allow teachers perform all possible operations only within their subjects and the exams they create.
2. Allow students perform all possible operations within the exam they are in and the exercises they submit.

3.2 Teacher

3.2.1 Exams

1. Allow teachers to create an exam.
2. Allow teachers to edit an exam.
3. Allow teachers to view all created exams.
4. Allow teachers to delete an exam.
5. Allow teachers to start exams.
6. Allow teachers to finish exams.

7. Allow teachers to see exam results when it is finished.

3.2.2 Exercises

1. Allow teachers to create exercises for one exam.
2. Allow teachers to edit the exercises for one exam.
3. Allow teachers to delete the exercises for one exam.
4. Allow teachers to view all the exercises of one exam.

3.2.3 Test cases

1. Allow teachers to create public test cases for one exercise.
2. Allow teachers to create private test cases for one exercise.
3. Allow teachers to edit public test cases for one exercise.
4. Allow teachers to edit private test cases for one exercise.
5. Allow teachers to delete public test cases for one exercise.
6. Allow teachers to delete private test cases for one exercise.
7. Allow teachers to view all test cases for one exercise.

3.2.4 Scoring

1. Allow teachers to mark exams according to the results on each exercise, and the test cases that are passed / failed.

3.3 Student

3.3.1 Exams

1. Allow students to enter an exam.
2. Allow students to solve exam exercises and submit them.
3. Allow students to submit an entire exam.

3.3.2 Exercises

1. Allow students to work on an exercise at any point during an exam.
2. Allow students to run the code of an exercise at any point during an exam.
3. Allow students to submit the code of an exercise at any point during an exam.

3.3.3 Test cases

1. Allow students to see the public test cases of an exercise.
2. Allow students to run the public test cases of an exercise.
3. Allow students to see the passed / failed public test cases of an exercise after running their code at any point during an exam.

3.4 Playground

1. Allow users to run the code in the programming language offered by the platform.
2. Allow users to see the output of the compiler / interpreter after the code is run.
3. Allow users to use auto-complete programming language structures as they use the editor.

4 Quality Attributes

While defining the functional requirements and the entire architecture of our application we carefully considered the quality attributes that would impact on its behavior, design, user interface, and at the same time, the development and support of the system. These quality attributes together with their raison d'être and how we they are implemented within our platform are described in the sections below.

4.1 Primary quality attributes

4.1.1 Interoperability

Interoperability is an important benchmark for us when talking about the quality attributes of our system given the fact that we chose a microservices architecture where the transmission of data and its exchange with other external systems, via the API Gateway, is a common fact in the nature our system.

Implementing the API Gateway [14] pattern facilitates integration with third-party systems in the future and also facilitates the interoperability with other systems (or new microservices within our application). It also facilitates an abstraction layer to possible outdated external systems, different formats of data in external systems (or microservices), different versions of the API and backward compatibility of the API for integration.

Thanks to ensuring interoperability mainly with the microservices architecture and the implementation of the API Gateway pattern, we evade creating additional layers for API interactions and hence, in the future, having the problem of rebuilding the entire system if new business requirements appear.

4.1.2 Usability

Usability is one of the most essential quality attributes we considered when thinking the different API endpoints of the microservices we built

and also the frontend layer of them in our user interface. We think that unlike in cases with other attributes, users can see directly how well this attribute of the system is worked out specially when performing CRUD operations with exams, exercises or test cases.

We strongly focused ourselves in evading too much interaction or too many actions necessary to accomplish a task. At the same time we iterated on different product flows with incorrect sequences of steps in multistage interfaces to perform a simple task (such as adding new test cases and setting their attributes). Our initial efforts were much more complex for us for simple operations and usage of the application.

Keeping it simple and clear from the playground, to the identity management part and also to the evaluation platform as a whole has been in our best interests since the beginning and we hope it remains the same in next versions of this project.

4.1.3 Security

Given the nature of saving the submissions of exam exercises and also entire exams before being published security for users (and also for the entire platform) is another key quality attribute we have considered when designing our system. We ensure HTTPS usage between all the microservices that are within our system as a whole. We ensure proper authentication and authorization with the usage of JSON Web Tokens (JWTs). In this way we try to reduce the likelihood of malicious or accidental actions as well as the possibility of theft or loss of information. Security measures like restrictions of user access by authentication/authorization, encryption of passwords and content and enabling secure connections between the microservices are essentials features in our entire platform.

4.1.4 Fault Tolerance

Thanks to having a microservice architecture and being able to scale horizontally on demand, it is easier for us to ensure reliability in the form of Fault Tolerance. Parts of our architecture like the microservices itself

together with the support of addons like Apache Kafka helped us build a fault-tolerant product while improving the performance, reducing the complexity of the entire architecture (and each of its components at the same time). All this led us to have simpler components, a simpler user interface (and hence simple React components) reducing the development time of the entire project.

4.1.5 Scalability

Probably the most important quality attribute of this entire system and one of the first we discussed about when starting this project is Scalability. Our main concern was how to handle load increases without decreasing performance and satisfying the different needs of the clients connected to our platform and submitting coding snippets while waiting for the code sandboxes to respond and sometimes to even share to the user if the public and private test cases would pass or not.

The scalability of our platform can be improved either vertically or horizontally by replicating the microservices that compute the code snippets submitted from different clients.

Enabling horizontal scaling was a must since the very beginning of this project. Also the effort it takes to scale it was considered when we started working with Docker images and architecturing the different microservices together with the API Gateway pattern.

4.1.6 Availability

Supported by the Scalability and Fault Tolerance attributes we also ensure proper Availability by making sure the total working time is maximized due to how we prevent the system from failing but also how easier it is for it to scale vertically and (most importantly) horizontally.

Having each feature of the product worked out on a separated microservice lets the maintainers update the software easily with different patches or minor releases and hence, a low time to apply these updates if

needed to the system as a whole.

4.2 Secondary quality attributes

4.2.1 Modularity

A quality attribute that definitely characterises our platform is Modularity, given that from the microservices architecture we focused since the very beginning on separating the functionality of a program into independent, interchangeable modules, such that each of them contains everything necessary to execute only one aspect of the desired functionality.

We also believe that this attribute helps to extend the platform with further microservices and implementation, all of them under the API Gateway pattern. It is also important to highlight that other quality attributes, such as Scalability, Maintainability and specially Interoperability, rely on this attribute.

4.2.2 Supportability

Related to Modularity, another quality attribute that we consider of importance is Supportability. We addressed this problem by having the microservices architecture and the different functions of the product working in a separated microservice. In this way, we manage to have separate logs for each microservice and it is easier to control the activity and performance of each them. Following this line of thought, one can even release different versions of the microservice (patches, minor, major releases) and identify and solve problems differently.

For us, logging and controlling the activity and performance of the system is of key importance for troubleshooting, doing system backups and even creating snapshots of the system. In this way, even auditing the system as a whole (not only one particular microservice) is a much simpler task. Health checking the system for measuring compilation time, deployment time and database size is easier to do thanks to this.

4.2.3 Maintainability

Given that this platform should easily support changes related to new business requirements or the correction of errors that can affect system components, at the moment of designing it we considered that making it easily maintainable was of paramount importance. Related to this we tried to evade excessive dependencies between components and spaghetti code.

At the same time, from an architecture point of view, we separated the responsibility of the different parts of the system with a microservice architecture and modularity. Having said this, this attribute affected not only our development processes but also how we splitted our responsibilities when developing the different product-related features.

4.2.4 Performance

We considered this quality attribute as a benchmark for our application to make sure all the actions in the system during a certain period of time occur in a proper latency (the time it is spent on responding to an event) and the channel capacity (the number of events that occur at a certain point in time). We believe performance issues very often grow into problems that can affect everything, from our server's capacity to how we built our front-end or even the efficiency of database queries.

Given all these problems that may affect the user experience but also the functioning of the system as a whole we considered Performance to be a key quality attribute to be taken into account. Also, the fact of this being an evaluation coding platform increases the importance of code submissions and their corresponding programming environments (with their environment variables, error reporting, etc) to work as one would expect.

5 Architecture

The chosen architecture for the platform is based on independent microservices, where each of them implement the corresponding features. Each microservice exposes different types of APIs, which can be consumed by other microservices, and from external clients as well. Communications can be performed both synchronously (using HTTP as protocol), and asynchronously (using a message broker).

With this approach, scalability and fault tolerance are achieved. On one hand, as components are deployed independently, when more capacity is needed to fulfill different types of requests, more replicas of a given set of microservices can be deployed. On the other hand, failure of a given component might not affect the operation of the entire system, and users might be able to continue using the platform – especially if the failed component is accessed through a message broker.

Apart from scalability and fault tolerance, this architecture helps achieve modularity. Each component is a separate piece of software that can be developed, tested and deployed independently. This will allow future extensions of the platform in an easy way. Adding new features will result in tiny modifications of existing components, or adding new microservices, which will not impact – in most cases – in the existing system.

The next subsections will give more details about the architecture.

5.1 Components

The platform is built from different components that interact between them using different protocols. These components have different tasks (implementing business logic, storing data, exchanging messages, translating protocols, etc). The following figure describes the architecture in an abstract way, giving a general overview of the system.

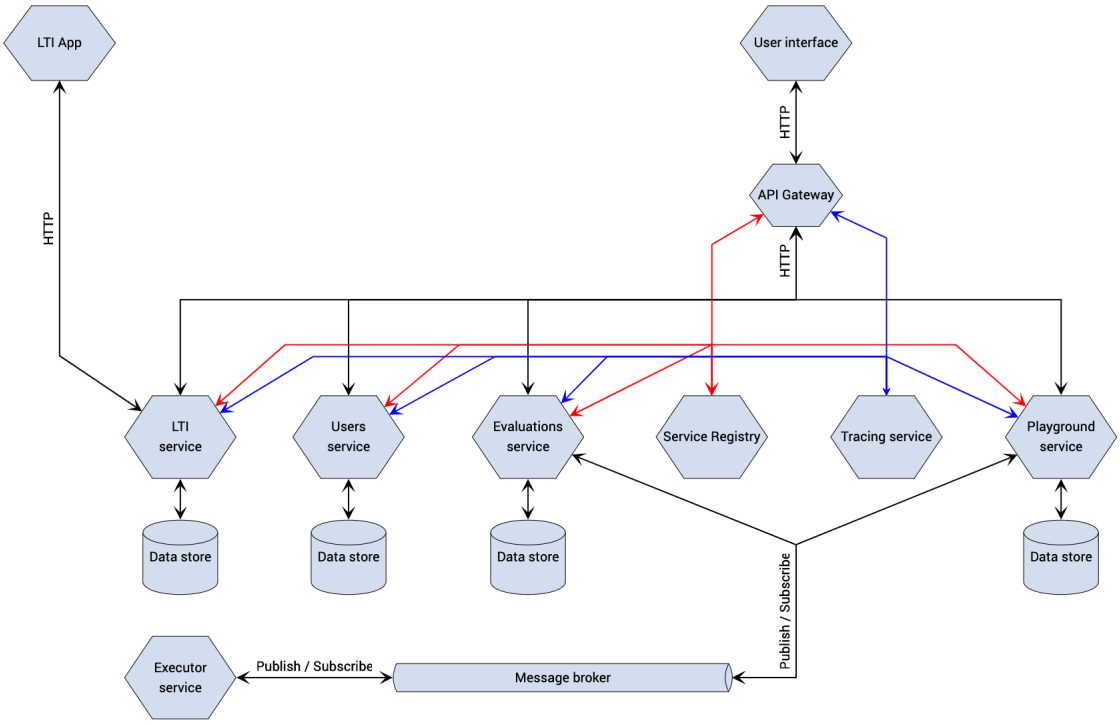

Figure 1: General overview of the architecture

5.1.1 Executor Service

The executor service is in charge of running code submitted by any other component of the platform. It can be accessed by sending a message to it via the message broker, which means that execution requests are performed asynchronously. This avoids the platform from blocking while the system is executing code, which allows handling more requests at a given time. The executor service will notify execution results in a "reply channel", also using the message broker.

5.1.2 Evaluations Service

This component is in charge of managing exams (including its exercises and test cases) created by teachers, together with solutions submitted by students, and results achieved given the said solutions.

The service can be accessed through a REST API, which exposes the different operations implemented by the component. These include creating, editing, deleting, and searching exams, exercises, test cases, submitting solutions, and querying results. In order to perform these tasks, the service relies on a relational database. It also communicates (through the message broker) with the executor service to request the execution of solutions.

5.1.3 Playground Service

The playground service allows users to interact with the executor service thought a REST API. It handles the protocol translation by storing created execution requests, sending the corresponding messages, and subscribing to the channel where the executor service publishes the execution results.

5.1.4 User Service

The users service is in charge of managing users, including permissions granted to them. It is basically the identity management software of the

platform. It allows creating, (de) activating, and deletings users, including adding and removing permissions.

For the sake of simplicity, this service is also in charge of issuing, refreshing, and blacklisting tokens which are used by the entire platform as a medium to authenticate and authorize users. The standard in which this relies is JWT (JSON Web Tokens).

This component can be accessed through a REST API which exposes the different operations implemented by the service.

5.1.5 LTI Service

This component is in charge of implementing the LTI 1.3 standard. It basically creates the messages that must be exchanged between a learning platform and this system to allow the integration with the university's online campus. These messages are actually JWTs, which means that the service must store private and public keys (the former to sign the messages being sent; the latter, to verify the received ones). The service can be accessed through a REST API.

5.1.6 LTI App

The LTI app is a web application that communicates with the LTI service through its REST API. It exposes a user interface that can be accessed from the learning platforms, allowing users to communicate with this system in the context of the said platforms.

5.1.7 Service Registry

Doing service discovery is something of key importance in the architecture. In any distributed environment, components need to know the location of other members of the platform. This concept has been around since the beginning of distributed computing.

Service discovery can be achieved by a simple approach, such as a properties file, in which all addresses of all components are stored (they must be static, which means that they will not change ever), or by a more complex system, such as a DNS server, or a UDDI (Universal Description, Discovery, and Integration) repository.

The chosen architecture makes use of a service registry. This component allows the platform perform service discovery, which is critical to microservice, cloud-based platforms for the following reasons.

On one hand, it allows to easily scale horizontally up and down. This means that the number of replicas of a given service running in an environment can be altered without impacting the operation of the platform. The consumers are abstracted away from the physical location of the services.

On the other hand, service discovery helps increase application resiliency. When an instance becomes unhealthy or unavailable, the service registry removes the said replica from its internal list of available services. The damage caused by a failed component will be minimized because the service discovery engine will not resolve to unavailable services.

5.1.8 Tracing Service

This component is in charge of storing trace data created by all the other microservices. With this information, future administrators of the platform can query how a request is spanned across the different services that handles it, including gathering timing data and troubleshooting requests. This allows achieving the required supportability.

5.1.9 API Gateway

The entire architecture is composed of several microservices, each one implementing its own set of operations, exposed by different types of APIs (HTTP routes, or messages sent/received by a message broker). This creates a very fine-grained API, often different from what clients need.

If considered from a different kind of devices point of view, different consumers need different types of APIs. On one hand, each client might need different types of data. For instance, a desktop browser version of a product details page is typically more elaborated than a mobile version. On the other hand, network latency might not be the same for each client. A mobile network will not perform the same as a wired one, which means that accesses should be reduced. Finally, a client should not need to know the location of the different components that must be accessed to fulfill a given functionality.

The API gateway pattern [14] is then considered to solve those problems. The architecture has a component (called API gateway) that implements this pattern. It allows accessing the platform from the outside world, through a REST API. It is the single entry point for all clients, routing each request to the corresponding service, based on different rules (as, for instance, HTTP method, HTTP headers, URL paths, and/or other HTTP constructs). This avoids exposing directly all the internal components that conforms the platform, allowing to exhibit a subset of operations (separating between the public and the private API).

Being a single entry point to the platform also allows enforcing several rules when a request is received. Authentication is performed in this component, analyzing the Authorization HTTP header, which must contain a valid and non-expired JWT. Also CORS is implemented in this service, adding the needed headers to the requests. Finally, it allows locating a whole request, including how it spans along the entire platform, making use of the tracing service.

Last but not least, being the entry point also avoids clients to know about the implementation of the platform. If the API gateway would not exist, clients would have to make use of the service registry to locate all the services that it wants to access, and then access them. Instead, making use of the API gateway allows clients ignore all this, which is performed by this component.

5.1.10 User Interface

Users will interact with the platform through a user interface implemented as a web application running in the user's browser. This component will communicate with the backend through the API gateway using the HTTP protocol. It implements all the features needed to fulfill the functional requirements of the project.

6 Implementation

Within the architecture, it was considered each component to be deployed as a Docker container. These containers are orchestrated using Kubernetes. Kubernetes operators [17] are used to simplify the deployment of some components. These are used to create Postgres [18] databases, which are used as data stores for some of the microservices explained in the architecture section. A Kafka cluster – together with a Zookeeper [19] ensemble – is used as a message broker. Both are also created using operators.

The next subsections will give more details about the implementation details, and how the architecture is deployed.

6.1 React for the front end

A single piece of code was decided to be used for all the screens and places where user interaction is needed. This code conforms an application to be run in the user's browser, which communicates with the backend through its API, matching the features it offers. The elected technology is React.

Originally developed by Facebook, React is a JavaScript library that builds user interfaces by dividing UI into composable components. Since it requires only a minimal understanding of HTML and JavaScript, React has risen in popularity as a front-end web development tool and currently has a lot of maintainers, tutorials and a big community behind it.

One of the reasons React was chosen is because of its learning curve. React is a very simple and lightweight library that only deals with the view layer. It is not like other MV* [11] frameworks, such as Angular or Ember. Any Javascript developer can understand the basics and start developing a web application after only a couple of days reading a tutorial.

Another reason why React was chosen is that it provides a component based structure. Components are like lego pieces. You start with tiny components like buttons, checkboxes, dropdowns etc., which are then used

to create wrapper components composed of those smaller ones. Then a higher level of wrapper components can be created, going one like that until a root component is created, conforming the web application.

Each component has its own internal logic, and decides how it should be rendered. This approach has some amazing results. For instance, components can be reused anywhere as needed, which allows the application to have a consistent look and feel and code reutilization, making it easier to maintain, grow the codebase, and develop the software.

Another reason why React was chosen is performance. When a web application that involves high user interaction, and view updates must be developed, possible performance issues must be considered. Even though today's JavaScript engines are fast enough to handle such complex applications, DOM manipulations are still not that fast. Updating the DOM is usually the bottleneck when it comes to the web performance. React is trying to solve this problem by using something called virtual DOM; a DOM kept in memory. Any view changes are first reflected to virtual DOM, then an efficient diff algorithm compares the previous and current states of the virtual DOM and calculates the best way (minimum amount of updates needed) to apply these changes. Finally those updates are applied to the DOM to ensure minimum read/write time. This is the main reason behind React's high performance.

6.1.1 Ace Editor

When choosing an editor for our end-users to use, we instantly thought of Ace Editor. It has a great community support and is updated compared to other alternatives in the market. It easily adapts to the needs we had at the beginning of the project and supported the programming languages we were thinking to use. If in future implementations, anyone wants to add more languages, that can easily done. Same thing goes for other plugins that can be added to the editor. [16]

Ace is an embeddable code editor written in JavaScript. It matches the features and performance of native editors such as Sublime, Vim and TextMate. It can be easily embedded in any web page and JavaScript ap-

plication. Ace is maintained as the primary editor for Cloud9 IDE and is the successor of the Mozilla Skywriter (Bespin) project.

Some of the features it offers from the baseline are:

- Syntax highlighting for over 110 languages (TextMate/Sublime Text.tmlanguage files can be imported).
- Over 20 themes (TextMate/Sublime Text .tmtheme files can be imported).
- Automatic indent and outdent.
- An optional command line.
- Handles huge documents.
- Fully customizable key bindings including vim and Emacs modes.
- Search and replace with regular expressions.
- Highlight matching parentheses.
- Toggle between soft tabs and real tabs.
- Displays hidden characters.
- Drag and drop text using the mouse.
- Line wrapping.
- Code folding.
- Multiple cursors and selections.
- Live syntax checker (currently JavaScript/CoffeeScript/CSS/XQuery).
- Cut, copy, and paste functionality.

6.2 Backend built using Java 11 and Spring

Backend includes several components, as a message broker and data stores. But the most important parts are the services that implement the business logic. These (micro) services are built using Java 11 and Spring Boot. The reason for choosing Java 11 is that is the newest version with long term support.

Spring is an open-source application framework, and an inversion of control container for the Java platform. Spring Boot is Spring's convention over configuration solution for creating stand alone, production grade Spring based applications that can "just run" – the web server can be embedded, which allows an easy and rapid deployment of applications. It also provides production-ready features, such as metrics, health checks and externalized configuration. But the most important feature is that it can auto-configure the application, which means that the developer must not (in most cases) create Spring Beans for stuff such as database connections or dispatcher servlets, among other things. Spring Boot will scan the classpath in search of libraries and create those beans based on the presence of such libraries. This will result in less code, increasing the maintainability of the platform.

Spring Data is used in components that must persist information. This Spring project provides a familiar and consistent Spring-based, programming model for data access, while still retaining the special traits of the underlying data store. It makes it easy to use data access technologies, relational and non-relational databases, map-reduce frameworks, and cloud-based data services. The most important benefit is that it allows developers to create repository interfaces, defining methods for custom queries, and letting the framework create the implementation of those interfaces using name conventions. This reduces the codebase, which results in less developing time, and increases resiliency, as bugs cannot be introduced by programmers. The data mapping layer chosen is JPA, using Hibernate as the implementation (which, however, is performed by Spring Data, as it was said).

Spring Security is used as the authentication and authorization layer of all (micro) services. Final users of the platform authenticate using a JWT

which must be included in the Authorization header of the HTTP requests. Spring Security then analyzes the token, and adds the user's data (including granted permissions) in the Security Context. This information is used to allow or deny operations in the platform, such as creation of exams, or solution submission).

Glassfish Jersey – currently Eclipse Jersey – is used as the web layer of all the (micro) services. It provides support for JAX-RS APIs, and serves as a reference implementation for it. Using this approach allows changing the implementation in the future. If other solutions would have been chosen (for example, Spring Web MVC), the code would be tied to that technology. Instead, using a standard as JAX-RS, the implementation of the web layer could be changed easily.

Cloud native features are provided by Spring Cloud. These include service discovery, client load balancing, fault-tolerance and tracing, among others. These features will be explained in the next subsections. The reason for choosing all these technologies is the familiarity, and the ease of finding developers for them. The university has been teaching these technologies for several years now, which means that future students could easily maintain the platform. Also, Spring provides a rich set of features which allows building applications with a big grade of complexity, in an easy way.

6.3 Netflix Eureka and Netflix Ribbon

Eureka is a REST based service developed by Netflix used for locating services for the purpose of load balancing and failover of services. It allows components to perform client-side service discovery, in order to find and communicate with each other without hard-coding hostnames and ports. It is composed of two parts: the client and the server.

When behaving as a server, the main task is to store and localize existing services, in order to be able to report their location, state, and relevant data of each of them. To achieve these tasks, those services must communicate with the server to inform their location, every now and then (this

amount of time can be configured). If a component does not report this data, then the server will consider the instance down, and will not include it when it is queried. This approach is known as taught monitoring, in which services send heartbeats to the server to report they are alive. To achieve high availability, the server can be replicated. In this case, all the server's instances will exchange their data, in order to be consistent between them. As high availability is one of the quality attributes that must be achieved by the platform, this component is deployed with two replicas. Even though Kubernetes will always monitor the pod in which the service registry is running, having two instances of the Eureka server allows minimizing downtime.

Eureka also has its client-side components. Each component that would like to communicate with other services registered with Eureka must query the registry to get a copy of the location data. This query must be performed every a set amount of time, in order to have the most recent data. This information can then be used by Ribbon – also developed by Netflix – which implements client load balancing and fault tolerance, to allow communications with the services managed by Eureka.

6.4 Spring Cloud API gateway

Spring Cloud gateway is a project that provides a library for building an API gateway on top of Spring Web MVC. It aims to provide a simple, yet effective way to route to APIs, and provide cross cutting concern to them, such as security, monitoring, and resiliency.

It is built on top of Spring Framework 5, project reactor, and Spring Boot 2.0. It is able to match routes on any request attribute (URL, path, headers, etc.). It can be integrated with Netflix Eureka in order to know the location of each service to which requests must be routed. Ribbon can be used to perform client load balancing when routing requests.

As it was said before, Spring Cloud Gateway is built on top of project reactor, which means that the execution model is based on reactive programming. This approach is concerned with data streams and progra-

pation of change, and it's especially useful when interacting with external systems, which are unpredictable. Using reactive programming, the process does not block waiting for responses, it just react when the result is available, increasing the performance and efficiency of the component. Only one thread could be used to handle all requests.

6.5 Spring Cloud Sleuth and Zipkin

Zipkin is a distributed tracing system. It helps gather timing needed to troubleshoot latency problems in service architectures. Features include both the collection and lookup of this data. All the service in the platform are integrated with Zipkin using Spring Cloud Sleuth.

A span is the basic unit of work. For example, sending an HTTP request to a service is a new span, as is sending an HTTP response. Spans are started and stopped, keeping track of their timing information. A set of spans forming a tree-like structure is called a trace. Each span is identified by a unique 64-bit ID, together with another 64-bit ID for the trace to which the span belongs. With this information, anyone can know how a request is handled across the several components of the platform.

Spring Cloud Sleuth is the library in charge of tracking traces on the platform. It uses protocol metadata to inform span and trace id when a request is sent, or a response is received. This data is then used by the process handling a request to be aware of the identity of the request. For instance, it can be used by the logging system to report unexpected situations.

Zipkin is used as the tracing service. Sleuth communicates with Zipkin to report spans, in order to store them and then be able to query them by system administrators. It offers a simple UI, which includes a dependency diagram showing how many requests went through each component. This is especially useful to check on performance and dependency issues (for example, if an old version of a component is still consuming a deprecated service).

For the sake of simplicity, this component is deployed as an in-memory database. This means that it cannot be replicated (as it is), and that data is ephemeral. However, in a more productive environment, it could be integrated with Cassandra, ElasticSearch or MySQL to allow persistence of data, and in consequence, replication. Note that having one replica of this service is not a bottleneck as requests can be fulfilled by all services even if Zipkin is down. This communication is asynchronous, which means that it does not impact in the request time. Final users will not note the presence of Zipkin.

6.6 Docker containers

A container is a standard unit of software that packages up code and all its dependencies together. This allows an application to run quickly and reliably from one computing environment to another. Docker allows packaging the components as containers, using OS-level virtualization [20].

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, libraries, and settings. A container image becomes a container at runtime and, in the case of Docker containers, images become containers when they run on the Docker engine, which can run on several Linux distributions (CentOS, Debian, Fedora, Oracle Linux, RHEL, SUSE, and Ubuntu), and Windows Server operating systems. This enable containerized applications to run anywhere consistently on any environment, on any infrastructure.

6.7 Kubernetes cluster

Kubernetes is used in order to simplify the deployment of containerized components. Kubernetes is an open source system – originally designed by Google, now maintained by the Cloud Native Computing Foundation [21] – for automating deployment, scaling, and management of containerized applications, across a cluster of hosts running the system. It groups containers that make up an application into logical units for easy

management and discovery.

Kubernetes is spread across several nodes, conforming a cluster. There are two types of nodes: the master node, and worker nodes. The master node executes several processes needed to control the cluster. The main processes are kube-apiserver, kube-controller-manager, and kube-scheduler. Its main task is to maintain the desired state of the cluster. The master node can be replicated to achieve high availability and redundancy. The worker nodes are those executing the scheduled processes and workflows in the cluster. They are controlled by the master node. The main processes executed by these nodes to be part of the cluster are kubelet, which is needed to communicate with the master node, and kube-proxy, which implements the necessary network services on each node.

Kubernetes defines a set of building blocks ("primitives"), which collectively provide mechanisms that deploy, maintain, and scale applications based on CPU, memory, or custom metrics. Kubernetes is loosely coupled and extensible to meet different workloads. This extensibility is provided in large part by the Kubernetes API, which is used by internal components, as well as extensions and containers that run on Kubernetes.

The platform exerts its control over compute and storage resources by definings resources as Objects, which can then be managed as such. The key objects are the following.

6.7.1 Pods

A pod is a higher level of abstraction grouping containerized components. It consists of one or more containers that are guaranteed to be co-located on the host machine, and can share resources. It is the basic scheduling unit in Kubernetes. Pods can be managed manually through the Kubernetes API, or by delegating their management to a controller.

Each pod in Kubernetes is assigned a unique pod IP address within the cluster, which allows applications to use ports without the risk of conflict. Within a pod, all containers can reference each other on localhost, but a container within one pod has no way of directly addressing another con-

tainer within another pod. Instead, it has to use the pod's IP address.

A pod can define a volume, such as a local disk directory, or a network disk, and expose it to the containers in the pod, in order to store data. Note that pods are ephemeral, which means that data is lost when the pod is terminated.

6.7.2 Replica sets

A Replica set purpose is to maintain a stable set of replica pods running at a given time. They are used often used to guarantee the availability of a specified number of identical pods. To create new pods, a replica set uses a pod template.

6.7.3 Services

A service is an abstract way to expose an application running on a set of pods as a network service. Using a service, a consumer can refer the pods by a name, instead of by their IP, which are ephemeral (they will change if the pod is terminated and re created).

6.7.4 Volumes

A Kubernetes volume provides persistent storage that exists for the lifetime of the pod itself. This storage can also be used as shared disk space for containers within a pod. They are mounted at specific mount points within the container, which are defined by the pod configuration, and cannot mount onto other volumes, or link to other volumes. However, the same volume can be mounted at different points in the filesystem tree by different containers.

6.7.5 Statefulsets

A statefulset allows a pod to preserve state if it is restarted. It also allows to redistribute this state when an application is scaled up and down. Finally, it enforces the property of uniqueness and ordering among instances of a pod.

6.8 Postgres databases

PostgreSQL is a free and open source relational database management system (RDBMS) emphasizing extensibility and technical standards compliance. It is designed to handle a range of workloads, from single machines to data warehouses or web services with many concurrent users. Postgres features transactions with Atomicity, Consistency, Isolation and Durability (ACID) properties. This database engine is used as the data store for the components described in the architecture section (specifically, for the evaluations, the playground, the users and the LTI services).

Postgres databases are deployed in Kubernetes using Zalando's Postgres operator [22]. This tool allows creating high available Postgres clusters in a very easy way, powered by Patroni [23]. Patroni is a tool developed by Zalando that simplifies the deployment of replicated Postgres databases by managing their state, including leader election of a master node. Other features include rolling updates, volume resizing, cluster cloning, logical backups stored in AWS S3 [24] buckets, among others.

In order to create a Postgres cluster using this tool, a manifest file must be created, setting the cluster's desired state. The operator will be in charge of maintaining this state, checking the cluster's health, and performing the needed operations to get the cluster into the said state.

6.9 Apache Kafka

Apache Kafka was developed by LinkedIn in 2010, and it has been a top-level Apache project since 2012. It is a highly scalable, durable, robust, and fault-tolerant publish-subscribe event streaming platform. It is

used as the message broker of the platform.

The basic components that Kafka uses for its publish-subscribe messaging system are the producers, the consumers, and the brokers. A producer is an entity/application that publishes data to a Kafka cluster, which is made up of brokers. A broker is responsible for receiving and storing the data when a producer publishes. A consumer then consumes data from a broker at a specified offset, i.e. position. The result after connecting all these components is a multi-producer and multi-consumer structure.

A basic unit of data in Kafka is generally called a message or a record (interchangeably). A message contains the data and also the metadata. The metadata contains information such as the offset, a timestamp, compression type, etc. These messages are organised into logical groupings or categories which are called topics, to which producers publish data. Typically, messages in a topic are spread across different partitions in different brokers. Each partition contains a subset of a topic's messages. A broker can have multiple partitions. This allows increasing throughput; parallel access to the topic can occur.

Furthermore, Kafka also provides reliability and data protection using replication. If a broker fails, then all the partitions assigned to that broker would become unavailable.To resolve this issue, there is the concept of a replica, i.e. a duplicate of each partition. The number of replicas a partition has can be specified. At a given point in time, all replicas are identical to the original partition – i.e. the "leader" – unless it hasn't caught up to the most recent data. This leader is elected using Zookeeper (among other features).

What is unique about Kafka is that it keeps all the messages for a set amount of time (this can be indefinitely). Each message has an offset, or position, in this message log. Instead of Kafka managing which message a consumer is up to, Kafka delegates this responsibility entirely to the consumer itself. By doing this, Kafka is able to support many more consumers.

In order to deploy a Kafka cluster in a Kubernetes cluster, operators are used. In particular, Banzai cloud's operator [25] is used for this. This al-

lows to create a manifest file that describes the Kafka cluster, and then this tool will be in charge of maintaining the cluster's state according to the said manifest, including persistence and access. Other features include fine-grained broker configuration support, advanced and highly configurable External Access via load balancers, graceful scaling and rebalancing, monitoring, encrypted communications, automatic reaction and self healing based on alerts, graceful rolling upgrades, and advanced topic and user management via custom kubernetes resources (CRDs).

7 Infrastructure

In order to achieve the proposed architecture, and the chosen implementation, infrastructure must be deployed. This includes host machines and networking. As it was stated before, the platform consists of a set of decoupled components, deployed in a Kubernetes cluster, which means that a cloud is needed for this. Amazon Web Services [5] was chosen as the cloud provider. The following paragraphs will explain the infrastructure.

First of all, a Kubernetes cluster is needed, as it is the platform in which all the system is deployed. Deploying a Kubernetes cluster is not an easy task. Many components must be installed, including orchestration and networking software. That is why a tool like KOPS must be used.

KOPS [6] helps create, upgrade, maintain and destroy production-grade, highly available, Kubernetes cluster from the command line. AWS is currently officially supported, with Google Compute Engine [7] and OpenStack [8] in beta support, and VMWare vSphere [12] in alpha.

KOPS helps provisioning Kubernetes clusters using the command line, and by manifest files that describes the cluster (for instance, amount of nodes, names, and networking stuff can be stated in these files). It can build high available cluster spread across several availability zones in a given region, including master node replication. With just a couple of steps a production-grade cluster can be provisioned. To maintain the cluster's state (this is, the amount of desired nodes) KOPS creates autoscaling groups [13] (for master nodes, and for worker nodes) which monitors the hosts' health, and are able to scale up and down the amount of nodes in order to achieve the desired performance.

For the sake of simplicity, the platform is deployed in a single availability zone, using only one master node, and three worker nodes, without autoscaling (i.e without increasing or decreasing the amount of hosts, but only monitoring the node's state to allow autohealding in case one machine fails). It was decided this way taking into account that this is a university project, and that the computing costs might be too high.

The chosen instance type for both the master node, and for the worker nodes is m5.large [9]. These are general purpose instances, providing a balance in compute, memory and network resources. It features two virtual CPUs, 8 GiB of RAM memory, EBS-Only storage, up to 10gbps of network bandwidth and up to 3500mbps EBS bandwidth.

Another type of host exists in the platform. This is the bastion host, which exposes an SSH server to the outside world. If any remote access to any node in the cluster should be performed, this must be done through the bastion host. The chosen instance type for this is t2.micro [10]. These are low-cost general purpose instances, whose performance can be bursted to achieve higher levels of CPU performance when needed.

An AWS classic load balancer is used to access the master node from the outside in order to schedule pods (this is, in order to deploy the components). Using this technology, which combines with the autoscaling group for the master node, cluster operator users can communicate with the node in a resilient way. Apart from that, this component implements HTTPS which allow encrypted communications. Note that operators must authenticate with the cluster in order to perform actions in it. Another load balancer is used to access the bastion host. Finally, (micro) services exposed to the Internet, such as the API gateway, the frontend server, and the LTI app, can be accessed through their respective load balancers. They all use HTTPS to increase the security of the system. The certificates used to achieve encrypted communications are issued by AWS. Load balancers are accessed by their name (which match the certificates' names) assigned by Route53, the AWS DNS service.

Two types of networks exist in the provisioned cloud. There is a private network in which all Kubernetes nodes are placed. Hosts in this network are not directly exposed to the Internet (this means that they do not have a public address). There is also a public network in which components exposed to the internet are placed. These includes the bastion host, the NAT gateways, and the load balancers.

In order to allow nodes start communications over the internet with external services (such as LTI Assignment and grade services, implemented by the LMSs to which the platform is integrated), NAT gateways must be

provisioned. These components are in charge of enabling instances in a private subnet to connect to the internet, the same as a router allows connecting a laptop to the internet in a house.

EBS volumes are used to store persistent data. When a persistent volume is created in the cluster, this is reflected as an EBS volume created in AWS. These volumes are used to store databases, such as Postgres, Kafka and Zookeeper data.

KOPS uses an AWS S3 bucket to store the cluster state. This is used to backup the cluster, which means that can then be replicated in other environments (for example, in other regions). This allows rapid change if, for example, a region becomes uncommunicated with the Internet, increasing the high availability of the platform as a whole.

The following figure gives a graphic overview of the infrastructure provisioned to carry out the developed platform.

Figure 2: Graphic overview of the provisioned infrastructure used to deploy the platform.

8 Critical Points

Some of the critical points we have noticed while developing the entire application were listed in the sections below. Some of them can be considered possible entry points for future improvements or iterations of this product.

8.1 Apache Kafka

Though Apache Kafka is of key importance to the functioning of our system as a whole, it is certainly also a critical point in our architecture. One of the main weaknesses it offers is that if it fails, some services will stop working as expected while other remain working. This is something that is complemented with the health discovery service our micro-services architecture offers. However, when losing the functionality of one of the services some of the functional requirements or use cases we have thought of will not work. A clear example of this would be that a teacher could create exams but cannot run the code of those exams in the code sandboxes.

Another clear example of the importance of this tool is that, for example, the API Gateway will keep passing or accepting tokens that might have been blacklisted but as it does not have communication with a (possible) fallen users service, the API Gateway will not be able to check with it which token to block.

8.2 ZooKeeper

Another consequence of working with Apache Kafka is how reliable we are with ZooKeeper. This means, that if Apache Kafka is down, Zoo Keeper will not know which node is the leader for a specific piece of data (being Apache Kafka the one who tells ZooKeeper which node is the leader).

At the same time, ZooKeeper itself is another process that runs standalone on a completely separate machine and if it falls the entire applica-

tion will fall as a whole.

8.3 Databases

If any of the participating databases fall any of the business logic of the participating services of our micro-services architecture will not work at all preventing it from working or either given a malfunctioning of the different use cases the application is expected to support.

Going hand in hand with this aspect, any create, read, update or delete operations on the involved entities of each micro-service will not work.

8.4 API Gateway

Though we consider, and have mentioned it several times, that the API gateway pattern has been key for our entire microservices architecture we are aware that is has some drawbacks.

The API Gateway is yet another moving part that must be developed, deployed and managed alongside the other services and user interface. For future implementations or iterations of the product as a whole, developers will have to consider its maintenance like any other service we have. At the same time, changing it drastically or removing it would change the entire application's architecture as a whole. And moreover, the API Gateway is of key importance for extensibility reasons, being a key player to create new integrations with other services or third-party services.

Another collateral drawback that the API Gateway brings to the entire architecture is, as many additional parts of it, an increased response time due to the additional network hop through the API gateway itself. However, for most applications the cost of an extra round-trip is insignificant. We consider this a minor setback but worth mentioning it given that it is not completely transparent for the interactions of the services.

9 Methodology

9.1 First steps

Once chosen the different frontiers and challenges that we realized we would have after defining the functional requirements and quality attributes, we defined a methodology that would help us face and adapt the different roadblocks and technical difficulties during the entire project.

To begin with, we defined the general architecture which involves the API Gateway and different microservices that would interact between each other, while remaining scalable and data-consistent across all devices and user sessions.

Once drawn the first drafts and defined the technology that we would use for the different parts of the infrastructure we started implementing the scaffolding of each of those projects. We drew on paper the different database schemas listing each of the entities that are part of each of the microservices and the relationships between them.

We knew there was going to be more infrastructure and additional components since the very beginning so we considered them as well while building each of the microservices, but we tried to have a lean and simple approach when creating a similar and compatible architecture between all those services.

Having taken all these considerations, we gained a better understanding of the dimension of the project.

9.2 Docker images

For portability reasons while under development circumstances and also when launching to production each of the microservices, we decided to use Docker images as environment containers. This had the only drawback of us learning the technologies behind it. Docker containers ensure consistency across multiple development and release cycles, standardizing your environment. One of the biggest advantages to a Docker-based

architecture is actually standardization. Docker provides repeatable development, build, test, and production environments.

Having Docker images helped us to speed up development by only having to pull from a git repository the last version of the image and get it running in seconds. This would include, for instance, having an instance of the service running, together with a database and Apache Kafka running together with it. Sometimes we would only create an image of a subset of the microservices to easily carry them anywhere without having to configure each of them and the configurations between them.

For installation purposes on production, the same advantages can be seen given that we can install each of the microservices without further complexity in the installation or configuration process.

9.3 Users: authentication and authorization

We decided to work on user's authentication and authorization after finishing with the other microservices. The reason behind it is that we needed to understand not only the different entities that would be related to the users (or lack of them) and the different endpoints they would consume.

At the same time, from a product design perspective we did not want the users microservice to interfere at all in the functionality of the other services. We wanted to think of it as if it was an abstraction layer of identity management spread through the entire application and across the microservices.

From a backend perspective, we had to build the service and make it interact with the other microservices and frontend for signup and login. From a frontend perspective, we just needed to protect the routes with a proper JSON Web Token (JWT) given sufficient scopes and authorizations.

9.4 Integration with ITBA Campus

Finally, to complete the implementation, we worked on the integration with ITBA Campus (BlackBoard), which is an LMS (Learning Management System) like Moodle. Again, to simplify things we never thought any of the services to rely on this integration, not even the one related to users on the section above this one.

For this purpose, to simplify things and take a lean approach for this integration, we did not proceed in making an entire integration but only in working with the identity management of the teachers, given the fact that we only needed to know which is the teacher and the subject they were creating the exam for.

It is important to highlight that the final mark of an exam ends up being a numerical mark on the student's profile of the course on ITBA Campus. This is done thanks to the communication between the application and the LMS through the LTI protocol.

9.5 Implementation

Defining the functional requirements and quality attributes was pretty straightforward given the fact that we had very clear what to do after seeing the state-of-the-art regarding other evaluation coding platforms. On the other hand, it took us a long while to decide the implementation and corresponding methodology.

What definitely prevailed as a methodology as a whole when implementing all the services, API Gateway and interface with all their features and corresponding processes (or flows) was the Agile Methodology. We mostly based our development process on an iterative development, where requirements and solutions evolved via collaboration between the two of us. To be more specific, we incorporated iterations and the continuous feedback from friends and our tutor which successively refined what we delivered.

Within Agile methodology we implemented a lot of the famous waterfall model by doing a sequential development approach by constantly

gathering requirements analysis which, at the same time, resulted in a software requirements specification. This would be followed by software design, implementation, testing (only on the backend), integration between multiple subsystems (with other components such as databases or Apache Kafka, for instance), deployment (of Docker images to integrate them all) and maintenance of the other services of the application as a whole. As part of this modal some of these phases overlapped between each other and between the development of different services.

What also drove us between the development of each service or phase of the development was the strong emphasis on planning and target dates we set for ourselves. Some of the development phases had the luck to count with its documentation written at the same time, some had formal reviews, approvals by users and of our tutor, and code review before starting the next phase (or actually closing the current one). An explicit deliverable of each phase is of course, the written documentation of each phase.

Last but not least, we did also have a bit of Extreme programming (XP) as part of our software development methodology which is intended to improve software quality and responsiveness to changing customer/tutor requirements. Given that it is part of the agile software development methodology, we used it to advocate frequent "releases" in short development cycles, these were more frequent and formal on the backend than on the frontend. The reason behind this is that the frontend grouped (and is agnostic) of the different interactions between the micro-services of the entire application.

Following this XP methodology, we believe we managed to increase productivity and introduced checkpoints at which new iterations and requirements could also be adopted while approaching the final product release. Within this methodology, we did code review at different milestones of the development of each service, in some cases performed unit testing of the code and we also definitely tried avoiding programming of features until they were actually needed.

In order to do this last part, we used a version control system which was GIT and the chosen platform for this purpose was GitHub. In that platform we created an organization where we created several reposito-

ries for the different parts of our project and microservices. In each of them we created the different features/enhancements, bugs and nice-to-haves. For the different new big changes (and internal releases) we used pull requests and reviewed each of them (by the one of us both who did not do those pull requests).

10 Future implementations

10.1 New and improved features

10.1.1 Stats to be collected

It is considered of paramount importance to track metrics or stats about the usage of the product in order to understand how the users interact with the platform, and be able to iterate the product on the right direction. Is because of this that it is considered that tracking teachers and students usages stats would be of great use not only for them, but for the developers and maintainers of the system.

In the case of teachers, if would be useful to know how many students are sitting for an exam, how many students did pass/fail an exam, how long did each student take to complete an exam, which test cases are the most failed ones, or which is the exercise that students were able to be solved by most of them, among others. These stats seem quite basic to be implemented, but currently the platform has no section to collect them, and report them back to the teachers. With this information, teachers would be able to create improved future exams, and see how easy or hard they were for the students.

Taking into account the students, tracking metrics about what students do while sitting for an exam could help both the teacher and the student to have a better feedback for future courses or exams of the same subject. This could include the amount of runs each of the exercises have before passing a given test case. Similar to this, a feature like "rankings per exercise" – as some platforms like HackerRank have – could be included. With this functionality students would have a better idea of how hard an exercise is. With this in mind, the average time a student takes to solve an exercise (or the whole exam) could be included. All this information would be really useful for students in order to know the order to tackle the exercises.

From the developer/maintainer point of view, being able to collect and present all these stats is something not yet considered by the proposed architecture. However, it could be easily implemented creating a (micro)

service that would store all the stats, collected using event streaming (using the message broker) by all the other components.

10.1.2 Improved test cases

Even though the platform already supports some basic conditions and fields to be filled as input when creating a test case, this could be improved. For example, HackerRank offers the possibility of more editing options when describing the test case, being able to include code snippets, highlight text, underline it or put italics. It also offers the possibility of tagging the difficulty it has for that test case to pass or not, which might be helpful for the student when wondering what to expect of the exercise that s/he is working on.

This feature goes together with scoring (based on the test case difficulty). Not all test cases have the same difficulty level. While creating the test cases, the teacher must ensure that more points are assigned to the difficult test cases as compared to the easy ones. This would lead to a better distinction between outstanding, good, and average programmers. The sum of scores of all test cases would be the total score assigned to a particular question. One can even assign zero points to sample test cases if required. The overall score for a coding question could be the sum of the scores of all the test cases which are successfully passed.

10.1.3 Support for more programming languages and themes

This is translated as adding more runners to the executor service. Right now Java, Ruby and C are supported. However, adding more environment is a very simple task. The developer should only upgrade the docker image running the executor service installing the necessary runtime, and then writing a new runner program (a template is included with the project).

Together with this, the frontend should be adapted alongside with the new languages being supported. Ace editor currently supports highlighting for over 100 programming languages. By default, files are highlighted

based on their file extension.

10.1.4 Auto-complete and multiple cursors

Regarding improvements on the Ace editor itself, several upgrades or improvements can be made due to its facility to add these kind of components. Ace editor provides varying levels of intelligent and responsive autocompletion for the code. Autocompletion is based not only on the content within the code, but also standard functions and language tools that might be used. This feature can be disabled as well at anytime.

Another important feature that can be enabled/disabled is multiple cursors. They can be used to perform tasks like rename several variables or members at once, break up lists separated by commas, or insert the same text in multiple locations. Multiple selections can be copied and pasted, and entire lines can be inserted or removed in several locations. One of the best capabilities is the ability to instantly select the next instance of the currently highlighted section. This is especially useful for refactoring several parts of the code at once.

10.1.5 Integration with 3rd-party services

Given that the Playground Service is basically a place where programmers can run code against different programming environments, many of these developers would definitely benefit from integrating the entire platform to other services. One option could be something like GitHub gist service (or any other gist service out there). Another useful tool would be a tool similar to Trello to reference the code snippet and be able to run it from there. A use case for this would be, for example, if anyone is working on a feature and wants to show the result of a code snippet that belong to that feature on the Playground Service, it would be easy to implement with a short URL belonging from the domain where the service is hosted, and then easy to share or include in Trello Cards, GitHub issues or similar.

Other kinds of integrations would include the ability to add web hooks to the platform, in order to allow other systems to perform actions based

on events happening in the application. This could include stoppers and non-stoppers web hooks. For example, before uploading a grade to an LMS, the platform could be integrated with a plagiarism service (that might exist on the Internet, or developed by other teams within the university), in order to avoid student from copying between themselves. This could be easily implemented creating a (micro) service that collects events streamed by other components, and performing the corresponding actions. In case the webhook is stopper, the component streaming the event would have to wait for the response (also emitted as an event) before continuing its workflow.

10.1.6 Multi-file code support

Allowing to run code spread across several files would be a big benefit for users of the platform. Most of the alternatives that have been evaluated on the "State-of-the-art" section do not include this option. A reason behind this might be that the UIs, the correction, the test cases applied or even the different code sandboxes executing the submit- ted code tend to be more complex. Despite it all, this would be a great feature to implement as it is extremely common for any developer from any language and background to separate code in files (for example, each Java class in a separated file).

The basic idea behind this feature would be to allow all files to be sent as one single submission, for them to work together (without circular dependencies/references of course) and then be ran against a single code sandbox that would return the result of all the files taken into play. Once this is achieved in Playground mode, it will surely be easier to implement in the Evaluations Service, however this comes at a cost of a medium refactor of it.

10.2 Architecture improvements

10.2.1 Containerize runners

Right now, submitted code is run in the same environment where the executor service lives. This has several possible issues. For instance, conflict between runtimes could appear (think of different versions of a given programming language, as Java 11 or Java 8).

Up to now, execution of submitted code is performed using bash scripts that instruct the machine to (compile and) run the program that was uploaded. The executor service uses this scripts to schedule the execution of the code (using the fork and exec approach), sending input (in the form of program arguments, or as standard input for the process), and waiting for the execution to complete, receiving the data that was sent to the standard output, or to the standard error output. It also sets some environment variables in order to copy the code to the process. Finally results are read from files created by the script.

In a future version of the platform this can be improved by extracting the runner to a container, which can be scheduled by the executor service. There would be different container images (one per supported environment). In this way, it would be easier to update each of the environments and, at the same time, add more runtimes, extending the programming languages support. The runners would run separated from the executor service.

In order to achieve this new approach, the executor service could become a Kubernetes controller. With that in mind, this component would interact with the Kubernetes API to deploy containers whose sole purpose would be to run the uploaded code, and report back the result. Another way of implementing this is to create a separate Kubernetes controller, and change the implementation in the executor service that interacts with the process API, becoming an adapter that communicates with the controller. A final approach could be to deploy each runner together with a sidecar (the executor service), which could react based on each language. In this case there would be a container for the runner, and another one for the controller. The executor service would continue receiving execution re-

quest messages, but would only react to those whose language matches the runner with which it is deployed. Note that there would be at least one pod for each of the supported language. To increase throughput in this approach, a Kafka topic for each of those pods could be created, so each one would subscribe to the corresponding channel (one for Java, another one for Ruby, etc.).

10.2.2 Error tracking software

In order to increase the supportability and maintainability of the project, it is recommended on nearly any software that is exposed to several users -and especially scale-, an error tracking tool. Such tools help developers fix bugs before they are reported to the product owner (or developers/-maintainers) and they help prioritize bug fixes because they show how many errors and which kind of users are getting on specific releases/versions of the different software pieces the application as a whole has. Every developer writes bugs and many of those bugs get shipped to production, that's unavoidable. Thanks to tools like this, one can find and fix those bugs before customers even notice a problem.

This can be seen using an example use case that might happen while using the application. A teacher might give start to an exam and then several students start sitting for it after he/she tells them so. Different code snippets are run against the corresponding services and given feedback on the different user interfaces. Some test cases may or may not pass and that feedback is given to the students as well on each of their browsers. After that, some code snippets are submitted as exercise solutions, and then the system grades them. Once the test is finished, the teacher stops it and sees the results. While this seems a common use case of the platform, there are several components involved. Different kinds of errors might appear depending on which clients are being used by the students or the inputs they have submitted against the internal APIs. This entire process can happen several times and errors might be seen or captured but not handled properly. With an error tracking software one can see how many errors happen, classify them and even know which service is the one that has the error, making easier the labour of bug fixing on each (micro) service.

One tool that can be used for this is Sentry. Sentry is an open-source error tracking software that helps developers monitor and fix crashes in real time. It allows to iterate continuously, boost efficiency in the development team and, as a result, improve the user experience.

10.2.3 Log aggregation

Carrying on with this same line of thought, centralizing all logs and being able to analyze them is definitely worth having in any (monolithic or not) application and a (micro) services application is no exception to this. A great tool for this matter is the ELK stack.

Kibana is an open source data visualization plugin for Elasticsearch. It provides visualization capabilities on top of the content indexed on an Elasticsearch cluster. Users can create bar, line and scatter plots, or pie charts and maps on top of large volumes of data. Kibana also provides a presentation tool, referred to as Canvas, that allows users to create slide decks that pull live data directly from Elasticsearch.

The combination of Elasticsearch, Logstash, and Kibana – referred to as the "Elastic Stack" (formerly the "ELK stack") – is available as a product or service. Logstash provides an input stream to Elasticsearch for storage and search, and Kibana accesses the data for visualizations such as dashboards. Elastic also provides "Beats" packages which can be configured to provide premade Kibana visualizations and dashboards about various database and application technologies.

Elasticsearch is a distributed, RESTful search and analytics engine capable of solving a growing number of use cases. As the heart of the Elastic Stack, it centrally stores all the data in one place making it easier to discover patterns or other information any developer might be missing. It allows to perform and combine many types of searches — structured, unstructured, geo or metric.

Logstash is an open source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to a given “stash.” As data is often scattered or siloed across

many systems in many formats, Logstash supports a variety of inputs that pulls in events from a multitude of common sources, all at the same time and then it can be easily ingested from logs, metrics, web applications, data stores, and various AWS services, all in a continuous, streaming fashion. As data travels from source to store, Logstash filters, parse each event, identify named fields to build a data structure, and transform them to converge on a common format for more powerful analysis and business value. Logstash dynamically transforms and prepares data regardless of format or complexity.

10.3 Auto-scaling

The way the platform is deployed (taking into account how components are packaged), and the underlying infrastructure, easily allows to horizontally scale up and down. If more replicas of a given service is needed, Kubernetes allows to schedule more pods to take the workload. Then, if more CPU power is needed, AWS could be instructed to add more nodes to the cluster.

All these operations can be automated based on metrics (amount of CPU and memory usage, amount of requests in a given period of time, etc.). This would increase the availability of the platform, as more replicas could be automatically deployed, without impacting the performance of the already running replicas. Apart from that, it would allow a quicker reaction from the platform in such events.

10.4 Infrastructure improvement

10.4.1 Multi availability zones deployment

Up to now the platform is deployed in a single availability zone, which means that, in the event of a datacenter failure, the platform would not be accessible. This issue could be tackled by deploying the Kubernetes cluster in across several datacenters (a.k.a. availability zones). So, in case one datacenter fails, the platform could continue providing services to consumers.

In case this is performed, pods could be scheduled in such a way that replicas are deployed in nodes in different availability zones, increasing the availability of the service they are carrying out. Also a master node would be needed to be replicated in each datacenter.

11 Appendix A: API documentation

The documentation of all the API endpoints for the different services used are listed here: `https://documenter.getpostman.com/view/8699794/SWDzgMx6`

It includes snippets to make requests in the following languages:

- cURL
- jQuery
- Ruby
- Python
- Node
- PHP
- Go

12 Appendix B: Infrastructure Guide

In the following link (GitHub repository where this project lives) you can find a guide to aprovision a Kubernetes cluster in AWS. Note that this guide only covers MacOS setups:

`https://github.com/coding-eval-platform/infrastructure-guides`

13 Appendix C: Deployment Guide

In the following link (GitHub repository where this project lives) you can find a guide to the usage of Kubernetes:

`https://github.com/coding-eval-platform/kubernetes`

14 Appendix D: Hexagonal Architecture

Regarding the code structure of the different services, they basically implement the *Ports and Adapters Pattern* [1], also known as the *Hexagonal Architecture*. The idea of Ports and Adapters is that the application (or service in this case) is central to the system. All the inputs and outputs reach or leave the core of the application through a port. This port isolates the application from external technologies, tools and delivery mechanics. The application itself should never have any knowledge about who is sending or receiving the input and output. This allows the system to be secured against the evolution of technology and business requirements. You do not want that an external system you use to become obsolete. And then be completely coupled to it. You want to be free from these changes, and make it easy to switch technologies or business partners.

One can consider a port like a gateway that allows the entry or exiting of data to and from the application. In code, this is what we call an interface. Ports exist in 2 types: inbound and outbound ports. An inbound port defines the exposure of the core's functionality. These interfaces define how the Core Business Logic can be used. This is the only part of the core exposed to the outside world. An outbound port defines the core's view of the outside world. This is the interface the core needs to communicate with the outside world. In short, the adapter transforms an interface into another. There are two kinds of adapters: primary and secondary.

The primary or Driving Adapters represents the UI. This can be our API controllers, web controllers and views. They are called driving adapters because they drive the application, and start actions in the core application. These adapters can use the inbound ports (interfaces) provided by the core application. The controllers then depend on these interfaces of the core business logic.

The secondary or Driven Adapters represent the connection to your back-end databases, external libraries, mail API's, etc. These adapters react to actions initiated by the primary adapters. The secondary adapters are implementations of the outbound port. Which in return depend on interfaces of these external libraries and tools to transform them, so the core application can use these without being coupled to them.

Some of the main benefits of this architecture are:

- **Agnostic to the outside world:** The application can essentially be driven by any number of different controls. You can use your inner core business logic through a Command Line Interface, another application or system, a human or an automated script.
- **Independent from external services:** When your application is agnostic to the outside world, it also means it is independent from external services. You can develop the inner core of your application long before you have to think about what type of database you are going to use. By defining the Ports and Adapters for your database, you are free to use any technology implementation. This allows you to use an in-memory datastore in the early days, and then make the decision of what type of database you want to use when you actually need to store your application's data in persistent storage.
- **Easier to test in isolation:** Now that your application is agnostic to the outside world, it is much easier to test in isolation. This means instead of sending in HTTP requests, or making requests to a database, you can simply test each layer of your application, mocking any dependencies. This not only results in quicker tests, but it also massively decouples your code from the implementation details of the outside world.
- **The Ports and Adapters are replaceable:** The role of the Ports and Adapters is to convert requests and responses as they come and go from the outside world. This conversion process allows the application to receive requests and send responses to any number of outside technologies without having to know anything of the outside world. It makes possible to replace an adapter with a different implementation that conforms to the same interface.
- **Separation of the different rates of change:** The outer most layers that typically change the most. For example, the User Interface, handling requests or working with external services typically evolves faster than the business rules of the application. This separation enables you to quickly iterate on the outer layers without touching

the inner layers that must remain consistent. The inner layer has no knowledge of the outer layer and so these changes can be made without disrupting the code that should not change.

- **High Maintainability:** Maintainability is the absence of technical debt. Changes in one area of an application doesn't really affect others. Adding features do not require large code-base changes. Adding new ways to interact with the application requires few changes. Testing is relatively easy. [2–4]

15 Appendix E: LTI Integration

This section includes the main flows of the LTI part of this project.

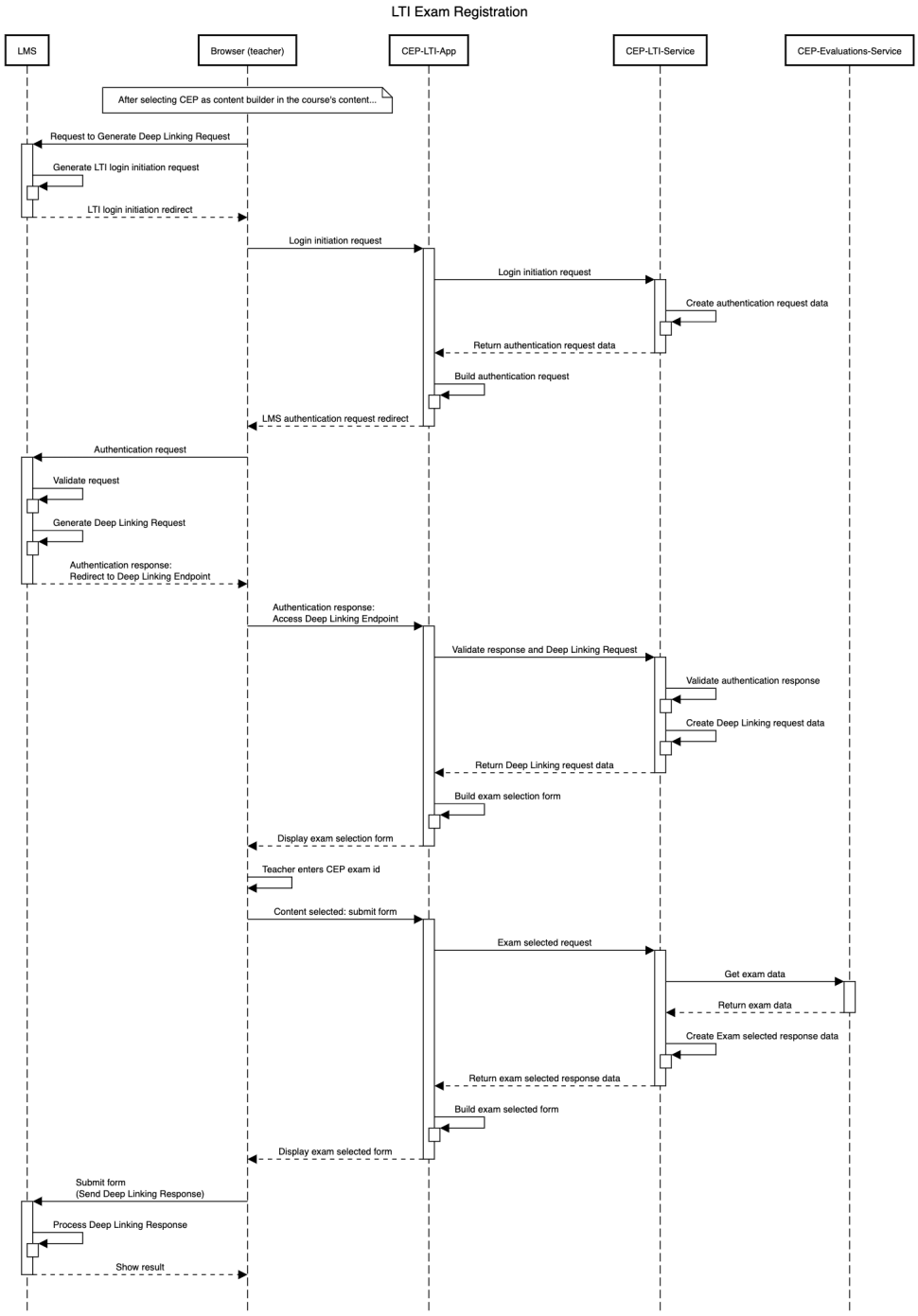

Figure 3: Exam registration flow

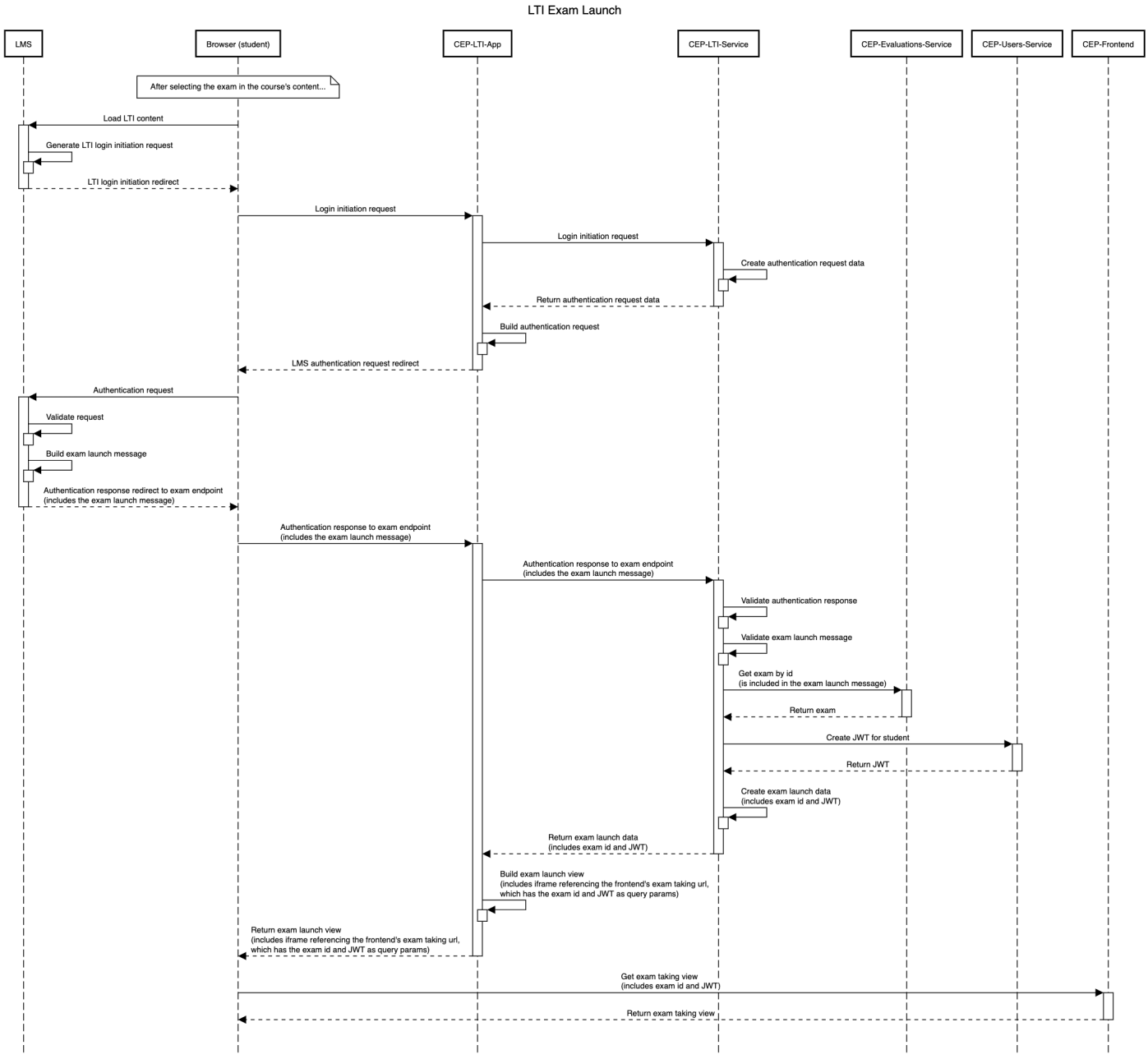

Figure 4: Exam launch flow

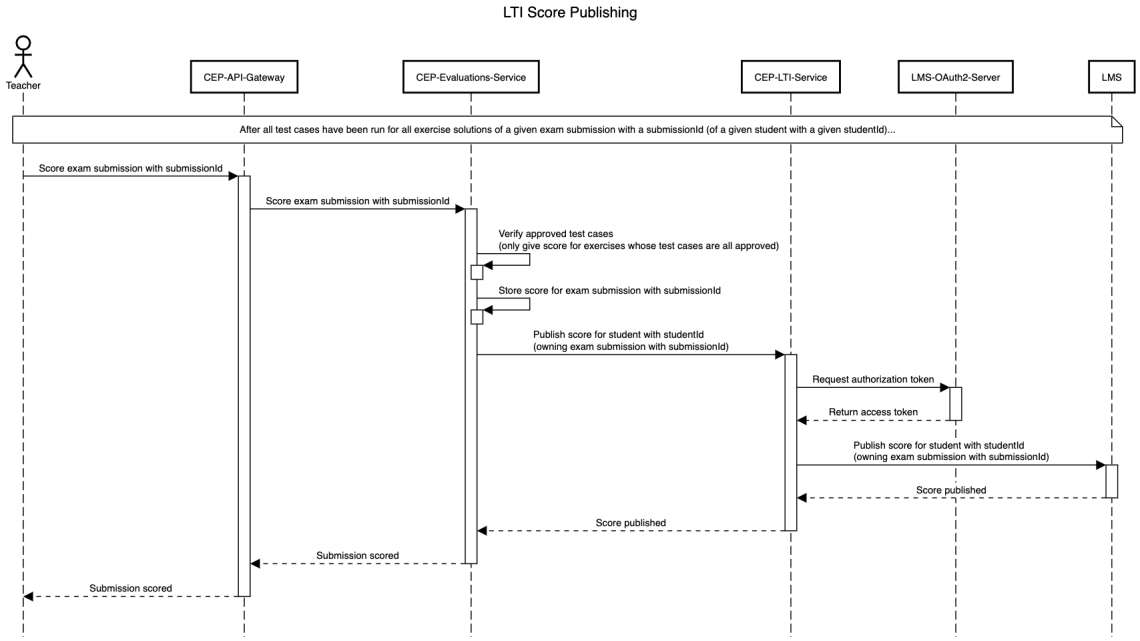

Figure 5: Score Publishing flow

16 Bibliography

References

[1] Ports and Adapters.
https://softwarecampament.wordpress.com/portsadapters/

[2] Microservices
https://dzone.com/articles/quick-guide-to-microservices-with-spring-boot-20-e

[3] Hexagonal Architecture
https://medium.com/@nicolopigna/demystifying-the-hexagon-5e58cb57bbda

[4] Hexagonal Architecture
https://apiumhub.com/tech-blog-barcelona/hexagonal-architecture/

[5] Amazon Web Services
https://aws.amazon.com

[6] KOPS
https://github.com/kubernetes/kops

[7] Google Compute Engine
https://cloud.google.com/compute/

[8] OpenStack
https://www.openstack.org/

[9] m5.large *https://aws.amazon.com/es/ec2/instance-types/m5/*

[10] t2.micro *https://aws.amazon.com/es/ec2/instance-types/t2/*

[11] MVWildcard

[12] VMWare vSphere
https://www.vmware.com/ar/products/vsphere.html

[13] Autoscaling groups
https://aws.amazon.com/autoscaling/

[14] API Gateway Pattern
https://microservices.io/patterns/apigateway.html

[15] Service Discovery / Registry
https://en.wikipedia.org/wiki/Web_Services_DiscoveryUniversal_Description_Discovery_and_Integration

[16] Ace Editor
https://ace.c9.io/

[17] Kubernetes operators *https://kubernetes.io/docs/concepts/extend-kubernetes/operator/*

[18] Postgres *https://postgresql.org/*

[19] Zookeeper *https://zookeeper.apache.org/*

[20] OS-level virtualization *https://en.wikipedia.org/wiki/OS-level_virtualization*

[21] Cloud Native Computing Foundation *https://www.cncf.io/*

[22] Zalando's Postgres operator *https://github.com/zalando/postgres-operator*

[23] Patroni *https://github.com/zalando/patroni*

[24] AWS S3 *https://aws.amazon.com/s3/*

[25] Banzai cloud's operator *https://github.com/banzaicloud/kafka-operator*