

## Bachelor Thesis

Bachelor Degree in Computer Engineering

Software Engineering

---

# LLM-based Automated Grading and Assessment of Student Submissions in an Academic Context

---

*Nicolás Aguado González*

### Advisor

Juanan Pereira Varela

June 10, 2025



# Acknowledgements

I want to express a profound gratitude to my parents, for supporting, guiding and being my foundation pillar throughout this entire journey. I am very grateful to have them by my side.

I want to thank my closest friends, for always being there and helping me keep my motivation throughout this whole process. I am grateful for all the good moments we've shared together, and for the many more still to come.

I also want to reserve a few words for my project advisor, *Juanan*, for his outstanding mentorship, his attentive oversight of my progress and his consistently insightful feedback and guidance.



# Abstract

Traditional Learning Management Systems (LMS) often fall short when used for providing timely and individualized feedback in the context of Project Based Learning (PBL). This bachelor thesis details the planning, design and development of *Evaluator*, a greenhouse web application that leverages Large Language Models (LLMs) for automated assessment of student assignments across multiple dimensions of submission quality.

Motivated by the need to alleviate educator workload and enhance student learning cycles, the defined application enables instructors to generate detailed rubrics and dispatch assignments, while students receive immediate, iterative feedback with diverse submission types.

Key architectural features include a fault-tolerant, multi-provider LLM integration for ensuring continuous operations, a modular system for submission type extendability, robust role-based authentication and a persistent background processing queue for grading tasks. The system is developed using the *Flask* framework (*Python*), *SQLAlchemy* as the ORM of choice and having the application containerized with *Docker* for production deployments.

*Evaluator* features a complete, production ready application containing all the expected real-world application characteristics, such as *i18n*, security hardenings and extensive documentation. Comprehensive testing, including unit testing (Pytest), load testing (Apache Benchmark), static code analysis (SonarCloud), and end-user beta testing further validated the system's reliability and performance.

The provided cost analysis also demonstrates its economic viability for educational settings, successfully delivering a comprehensive platform that augments traditional LMS capabilities and supports effective PBL methodologies.



# Sustainable Development

The *evaluator* project is expressly situated within the framework of the United Nations Sustainable Development Goals (SDGs), with a concrete emphasis on Goal 4: *Quality Education* <sup>1</sup>. By automating the assessment of student submissions, the *Evaluator* application delivers rapid, data-driven feedback that helps learners identify misconceptions and knowledge gaps at an early stage. This immediate feedback not only accelerates the learning cycle but also fosters deeper conceptual understanding, as students can iteratively refine their work through precise and concrete guidance. In parallel, the system partially relieves instructors of routine grading tasks, thereby reallocating faculty time toward the development of enriched instructional materials and the provision of targeted, individualized support for complex topics.

Furthermore, *Evaluator*'s architecture supports the principles of project based learning that underpin contemporary pedagogical research. By integrating continuous assessment into the student workflow, the application encourages active engagement, problem-solving and reflective practice; key components identified in the literature as determinants of long-term retention and higher-order thinking. Such a feedback-informed teaching cycle embodies the values of quality education by promoting inclusivity, equity and personalized learning pathways.

By coupling automation with human oversight, *Evaluator* strikes a balance between efficiency and pedagogical rigor, thereby contributing to more equitable access to high-quality feedback and fostering lifelong learning skills. Through these mechanisms, the project advances the objectives of SDG 4 by enhancing the inclusivity and sustainability of the educational experience.

---

<sup>1</sup>[www.un.org/sustainabledevelopment/education](http://www.un.org/sustainabledevelopment/education)





# Contents

<b>Contents</b>	<b>vii</b>
<b>List of Figures</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background and Related Work . . . . .	1
1.2 Motivation . . . . .	2
<b>2 Project Planning</b>	<b>3</b>
2.1 Project Scope . . . . .	3
2.1.1 Objectives . . . . .	3
2.1.2 Exclusions and Limitations . . . . .	4
2.2 Project Life Cycle . . . . .	5
2.2.1 Milestones . . . . .	6
2.3 Work Breakdown Structure (WBS) . . . . .	9
2.3.1 Dependencies Between Tasks . . . . .	11
2.4 Time Estimates . . . . .	11
2.4.1 Dedication . . . . .	11
2.4.2 <i>Tentative</i> Schedule . . . . .	12
2.5 Information and Communication System . . . . .	13
2.5.1 Storage and Backups . . . . .	13
2.5.2 GitHub Projects . . . . .	14
2.5.3 Communications . . . . .	15
2.6 Risk Management . . . . .	15
<b>3 Application Requirement Analysis</b>	<b>19</b>
3.1 Users and Stakeholders . . . . .	19
3.1.1 Stakeholders Identification . . . . .	19
3.1.2 Application Actors Identification . . . . .	19
3.1.3 Ideal Application Flow . . . . .	20
3.1.4 Functional Requirements . . . . .	21
3.1.5 Non Functional Requirements . . . . .	23

3.2	Use Cases . . . . .	25
3.3	Domain Model . . . . .	25
<b>4</b>	<b>Architectural Design</b>	<b>29</b>
4.1	Project Structure . . . . .	29
4.2	Component Overview . . . . .	30
4.3	External Providers . . . . .	32
4.3.1	Large Language Model Providers . . . . .	32
4.3.2	E-Mail and Github . . . . .	32
4.3.3	Blob Storage . . . . .	33
4.4	<i>Dockerized</i> System . . . . .	33
4.5	Production Deployment . . . . .	34
<b>5</b>	<b>Application Design</b>	<b>35</b>
5.1	Technologies . . . . .	35
5.1.1	Programming Language . . . . .	35
5.1.2	Web Framework: Flask+Jinja2 . . . . .	36
5.1.3	Object Relational Mapper: SQLAlchemy . . . . .	37
5.1.4	Testing Framework . . . . .	37
5.1.5	Redis-Queue Worker Implementation . . . . .	38
5.1.6	Internationalization (i18n) with Flask-Babel . . . . .	38
5.2	Development Tools . . . . .	39
5.3	Application Tiers . . . . .	39
5.3.1	Presentation Layer . . . . .	39
5.3.2	Application and Business Logic Layer . . . . .	40
5.3.3	Data Access and Domain Layer . . . . .	40
5.4	Application Security . . . . .	41
<b>6</b>	<b>Implementation</b>	<b>43</b>
6.1	LLM Fault-Tolerant Grading and Model Interchanging . . . . .	43
6.2	Cleanup and Reassurance Process . . . . .	47
6.3	Role Authentication Framework and Session Management . . . . .	48
6.4	Submission Type Extendability . . . . .	50
6.5	Miscellaneous Quality of Life Improvements . . . . .	51
6.6	Project Deployment . . . . .	53
<b>7</b>	<b>Cost Analysis</b>	<b>55</b>
7.1	Assumptions . . . . .	55
7.2	LLM Pricing Estimations . . . . .	56
7.3	S3 Pricing Estimations . . . . .	56
7.4	Remarks . . . . .	57
<b>8</b>	<b>Testing</b>	<b>59</b>
8.1	Unit Testing . . . . .	59

8.2	Load Testing . . . . .	60
8.3	Static Analysis (SAST) . . . . .	64
8.4	End User <i>Beta</i> Testing . . . . .	65
8.5	Metrics and Requirement Verifications . . . . .	66
<b>9</b>	<b>Monitoring and Control</b>	<b>69</b>
9.1	Change Control . . . . .	69
9.2	Time Tracking . . . . .	70
9.3	Risks and Mitigations . . . . .	73
<b>10</b>	<b>Conclusions</b>	<b>75</b>
10.1	Future Work . . . . .	75
10.2	Ethical Considerations . . . . .	76
10.3	Project Retrospective . . . . .	77
<b>A</b>	<b>Documentation and Manuals</b>	<b>79</b>
A.1	Development Setup and Useful Commands . . . . .	79
A.2	Deployment Manual . . . . .	81
<b>B</b>	<b>Code Extracts</b>	<b>87</b>
B.1	LLM Persistent Evaluation Function . . . . .	87
B.2	Decorators for Authentication . . . . .	87
B.3	Progressive Web App . . . . .	88
B.4	Fixture Initialization File . . . . .	88
B.5	Standalone Docker Compose . . . . .	88
<b>C</b>	<b>Testing Results</b>	<b>89</b>
C.1	Load Benchmark Results . . . . .	89
	<b>Bibliography</b>	<b>95</b>

# List of Figures

2.1	Project Life Cycle . . . . .	7
2.2	Work Breakdown Structure of the Project . . . . .	9
2.3	Specific Sub-Task from Milestone 2 on the GitHub Projects tool . . . . .	14
3.1	Top-Level Use Case Diagram of the Application . . . . .	26
3.2	Domain Model of the Application . . . . .	27
4.1	Filtered Overview of the Folder Structure of the Project . . . . .	30
5.1	High Level Overview of the Technology Stack for the Application . . . . .	36
6.1	High-Level Overview of the LLM Fallback Function . . . . .	44
6.2	Snippet of the Specific Tenant LLM Options in the Administrator Panel . .	45
6.3	File Structure of the Modular LLM System . . . . .	45
6.4	High-Level Overview of the LLM Selection Function . . . . .	46
6.5	Snippet of the Specific Tenant Cost and Storage Options in the Administrator Panel . . . . .	53
6.6	Snippet of some log events for the specific <i>Nico Test Course</i> in the Administrator Panel . . . . .	54
8.1	Console Output from a Successful Test Run with <i>pytest</i> . . . . .	61
8.2	Performance characteristics of the authentication endpoint under varying concurrency load levels . . . . .	63
8.3	Performance characteristics of the student submission endpoint under varying concurrency load levels . . . . .	63
8.4	Performance characteristics of the professor assignment creation endpoint under varying concurrency levels . . . . .	64
8.5	<i>SonarCloud</i> quality summary dashboard for the project at the end of the development iterations . . . . .	65
8.6	<i>SonarCloud</i> quality indicator (for each push) in the GitHub commit timeline . . . . .	65
9.1	Top Level Overview of some of the concrete Issues for Milestone 3 inside the <i>GitHub Projects</i> tracking application. . . . .	71
9.2	Granular Detail of a concrete Issue for Milestone 3 (with all its included metadata) on the <i>GitHub Projects</i> tracking application. . . . .	71

# List of Tables

2.1	Time Estimates for Each Task described in the WBS . . . . .	12
3.1	Stakeholder Identification . . . . .	20
7.1	LLM API Cost Grouped by Provider Model . . . . .	56
7.2	S3 Storage Provider Cost Grouped by Provider . . . . .	57
9.1	Time Comparison of the planned vs the real dedication for each task of the WBS . . . . .	72
9.2	Date Comparison of the planned vs the real ending for each milestone . . .	73



# Introduction

Project-based learning (PBL) has emerged as a foundation for modern educational practices, reflecting a shift from a teacher-centered instruction to a more learner-driven model. As highlighted by Blumenfield et al. [1], PBL promotes deeper understanding, critical thinking and the ability to transfer knowledge across contexts.

Despite its pedagogical advantages, the implementation of this methodology presents significant challenges for educators. Due to the individualized nature of students' work, providing this personalized guidance demands a certain non-trivial effort. In their review, Paiva et al. [2] highlight that, "it is not reasonable to consider that teachers could evaluate all attempts that the average learner should develop multiplied by the number of students enrolled in a course, much less in a timely, deep, and fair fashion."

## 1.1 Background and Related Work

In this context, automated grading mechanisms can help *lighten*<sup>1</sup> a portion of educators' workload while preserving a degree of personalized assessment. In domains such as software engineering, dynamic techniques based on unit testing have demonstrated partial effectiveness, as highlighted by Messer et al. [3]. Nevertheless, the reliance on human grading remains substantial, because, as noted by Ala-Mutka [4], enabling automated evaluation of student work requires the establishment of numerically measurable assessment targets.

In recent years, a new innovative approach to PBL has emerged. Large Language Model (LLM) based evaluation differs from usual strategies by leveraging its versatility to assess multiple dimensions of student work, including overall correctness, design rationale and documentation quality. A practical example can be seen in research by Wu and Chang [5], where an AI-based methodology for evaluating complex computer

---

<sup>1</sup>As in, not fully perform the assessment but rather help perform some high level *triage*

science projects is proposed. By adopting a meta-assessment perspective, this approach extracts generic grading variables that assess different quality aspects and are adaptable across diverse learning objectives.

Furthermore, as Kasneci et al. [6] remark, LLMs bear substantial implications for the education field. They argue that these models can generate instructional materials, enhance student engagement and tailor learning experiences to individual needs. Nonetheless, they also forewarn about the rigorous examination of potential biases and the establishment of human oversight that should be produced when integrating these kind of tools.

All together, the literature suggests that LLM evaluation offers a scalable and pedagogically aligned solution to the challenges of project-based learning.

### 1.2 Motivation

Traditional learning management systems (LMS) such as *Moodle*<sup>2</sup> provide robust infrastructures for course organization and submission delivery **but** rely predominantly on manual grading. Given an educator that wishes to apply a project based learning approach, these type of platforms can struggle to scale with all the individualized student submissions.

Therefore, a web-based application that performs LLM-based automated grading and assessment of diverse student submissions is proposed. This platform, called *Evaluator*, extends the typical LMS paradigm by building an entirely new system based on an automated grading engine. Such application enables instructors to define detailed rubrics through an intuitive interface and to dispatch assignments that immediately obtain feedback upon student submission. On the other hand, *Evaluator*'s retry mechanism allocates formative feedback at each iteration, guiding students towards profound comprehension of topics and leaving the more complex issues to a human reviewer.

---

<sup>2</sup>[www.moodle.org](http://www.moodle.org)



# Project Planning

This chapter defines the scope of the product to be developed, the tasks required for its development and its breakdown. The objective is to create an action framework that will meet the project's needs and minimize risks and incidents, as well as their possible impact. In addition, tentative schedules and dedication estimates are provided to be used as a reference for the development phase. This chapter also defines software tools and communication channels to be used for the management of the project.

## 2.1 Project Scope

The main scope of the project is to produce an application that automates the assessment of student submissions in an academic context. The system will be wrapped around Large Language Model (LLM) providers, exposing a reliable framework for obtaining feedback about student assessments and providing a methodology that can be used in an academic context useful to both students and educators.

### 2.1.1 Objectives

The main functional objectives that the produced application must fulfill can be outlined in two key points. These aspects describe the overall end goal that the system aims to achieve.

- The application must **provide a reliable feedback platform** for students to learn and improve about their assessments. The provided feedback should be tailored for each submission, so that given an assessment rubric the student can (in a detailed way) get useful insights and further internalize the aspects that the educator wants to transmit.

## 2. PROJECT PLANNING

---

- The system must **provide a supporting method for educators' assessment process**, allowing for timely reception and initial *triage*<sup>1</sup> of student submissions. The platform should aim to be specially useful in subjects with *continuous assessment* or *project-based learning* methodologies that rely heavily on multitude of assessments.

Apart from the functional objectives, this project also aims to consider several secondary implications.

- **The resulting application must be production-ready** and should behave well enough in real-world scenarios. This project starts the development of a system from scratch, but this should not mean the end result does not satisfy the expected quality, reliability and security requirements of a production system. This also means that some possible compromises could be made *feature-wise* and some related risks can arise (more on that, [later](#)).
- As the LLM field is a *rapid changing* environment, **the application should have enough modularity** to be able to adapt to these quick changes and not be (LLM) version dependant. As LLM models get better and better (and the release intervals become shorter), the objective is to not have the need of *developer* maintenance every time new advances occur in this technical field.
- **The application should be prepared to be extended** by other developers. Working in the education domain, there are a huge variety of professor needs, so the system should be designed to support technical adaptations in certain key aspects. This has direct implications on software quality and the technical design of the system.
- As the main key features of the system rely heavily on *third party* LLM providers, a hard dependency must be addressed. These external LLM providers have a cost, so a rough **economic viability analysis** of the system must also be performed (See Chapter 7).

### 2.1.2 Exclusions and Limitations

As the scope of the project is very broad and there are multiple ramifications that can take place, the need of limiting and defining concretely what things should or should not the project contain becomes evident. These are the several exclusions and limitations that are taken into account.

- As tackled in the [previous](#) section, the education field is composed of a huge variety of specific circumstances. The application must indeed have enough quality to be extended, but it remains outside of the project scope to develop a **plugin**

---

<sup>1</sup>In the sense that, when grading submissions in bulk, an educator could already know for each one what to focus on, or what to ignore.

**system** that allows for easy modular integration. For this project, it should be enough to follow clear software engineering patterns to allow for a future technical development.

- At the time of writing and executing this project, the LLM field is an area of *continuous research*. New models come out almost bi-weekly, and the system should be able to adapt to these continue changes. Nevertheless, because it is a large separate topic on its own, it remains out of the scope of the project to **analyze the quality of the LLM outputs**. The system should provide LLM feedback for the student submissions, but rather if that feedback is correct, incomplete or entirely wrong <sup>2</sup> remains out of the project scope.
- In relation to the previous exclusion, it must be reminded that the proposed system relies entirely on external LLM providers. Apart from the [risk](#) that arises from that implication, the **viability analysis and/of self-hosting these LLM models** remains out of the scope of the project. At the time of writing, most models are *not* open-source and the ones that are require non-trivial technical capabilities to be self-hosted. Because the expected quality output of the evaluation depends directly on the chosen model, it remains an entirely different task to evaluate if open-source models are good enough and another separate challenge to host those concrete models.
- The end resulting application must be production-ready, with all the expected quality, reliability and security characteristics. As the application wraps around these LLM providers, it should have some kind of control for when these services become offline and/or they experience high demand and take too long to respond. Nevertheless, it remains out of the scope to perform an **uptime and reliability analysis of the LLM providers' API interfaces**. The application will have checks in place for when the providers suffer outages, but the formal uptime and reliability analysis is excluded from the project.

Some of these exclusion points are later taken into consideration for [future work](#).

## 2.2 Project Life Cycle

The defined life cycle for the project is shown on Figure [2.1](#). An adapted version of the *iterative* life cycle has been tailored for the specific needs of the project. This bespoke design is inspired by the reasoning behind the Rapid Application Development (RAD) [\[7\]](#) approach. The milestone-based system for the project will focus on building a working application from the start and then performing successive improvement iterations.

---

<sup>2</sup>As in, that the given feedback would align with the one that the educator would *manually* provide

## 2. PROJECT PLANNING

---

When looking at a general overview, and in terms of the implementation, three main sections can be distinguished.

1. The first part outlines the starting points of the project. First, the **objectives** are defined and an exhaustive **requirement analysis** is made. Throughout this task, several meetings with the client are scheduled and the scope of the project is made clear, given the availability and communication preferences of both interested parties. Throughout those meetings an initial planning is also discussed and the general workflow is mutually agreed on. Then, that initial conversations are reflected on a more serious planning to manage the whole project.
2. The second part of the project relates to the *technical* implementation. Several *sprints* (or iterations) are designed with global starting and final tasks. Before starting doing sprints, a **technology** research is executed, such that the appropriate stack is chosen for the greenhouse project. Then, four iterations of a *pseudo-cascade* take place (those iterations expand in time  $\sim 2.5$  weeks, more on that **later**). Each sprint's tasks are planned, designed, executed <sup>3</sup> and then tested. At the end of each iteration, a meeting with the client is held to review the progress and an intermediate delivery is made. At the end of all iterations, the application is tested in its whole (load testing, etc.) and the final delivery is made.
3. The last part of the project life cycle refers to the *thesis* related work. The main key points are the thesis (this document) writing process, the defense presentation design and the actual in-person jury defense.

As with all plannings, this life cycle can change throughout the project and it is not as strict as the diagram reflects. Some non-dependent tasks might overlap, some parts of the thesis might be written before finishing development (specially those related to planning and requirements) and client meetings could be held at any point of the cycle.

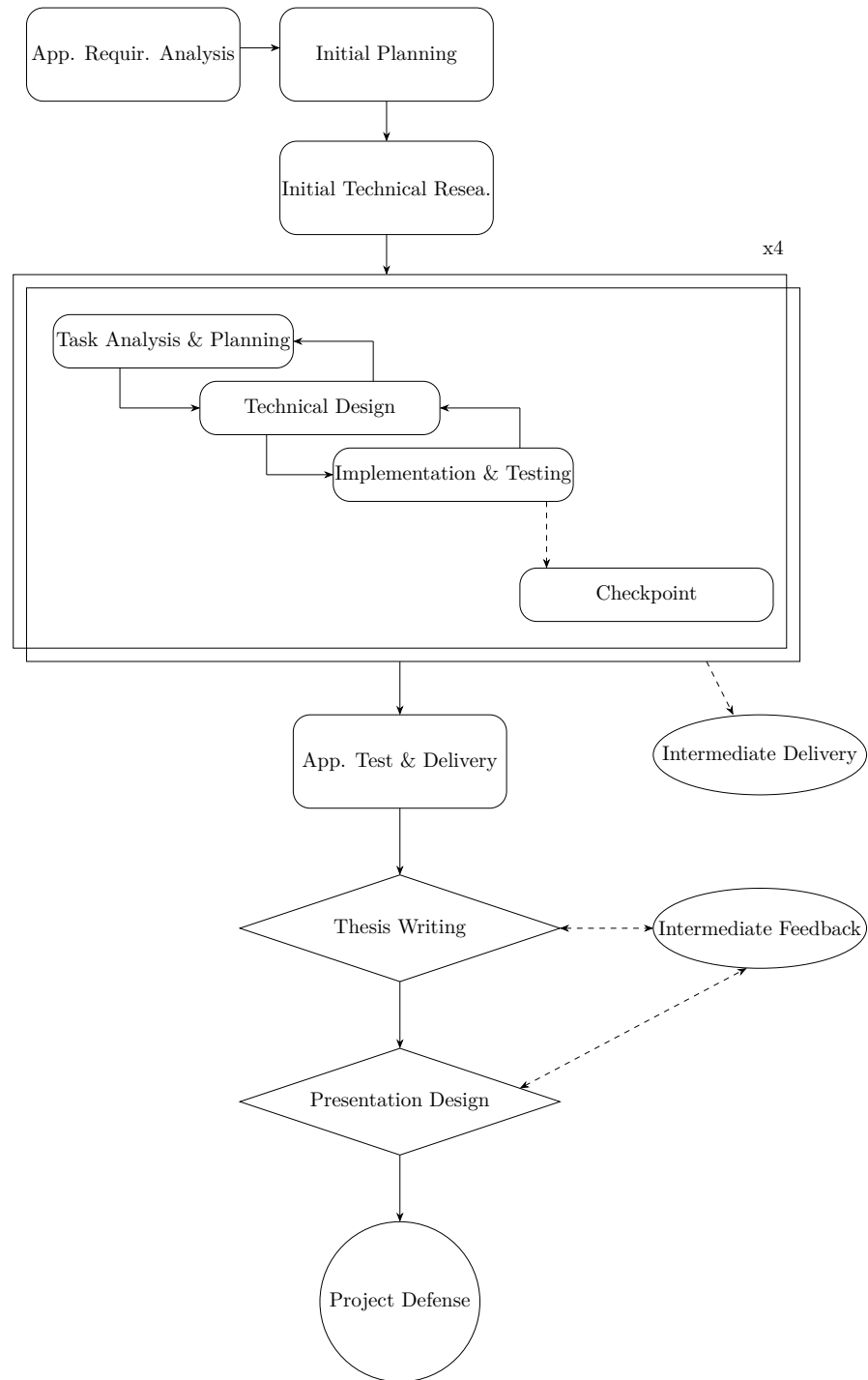
### 2.2.1 Milestones

Throughout the project life cycle, and in close relation to the *sprint* system and RAD philosophy, certain milestones can be marked to be considered along with some goals to meet for each one. The objective will be to produce a working application as quickly as possible and then perform successive refinements to meet the whole scope.

- **Milestone 0:** In this breaking point of the project, the initial requirement analysis must have been done, the client and the stakeholders (defined **later**) must be onboard with the project and an initial *top-view* project formal planning must have been done. In this point most if not all of the technical analysis, *greenhouse*

---

<sup>3</sup>In the project context, execution can also mean writing technical documentation, researching an algorithm or deploying infrastructure, apart from pure programming



**Figure 2.1:** Project Life Cycle

## 2. PROJECT PLANNING

---

related research and decisions must have been made, so that everything is ready to kick off the *pseudo-cascade* iterations.

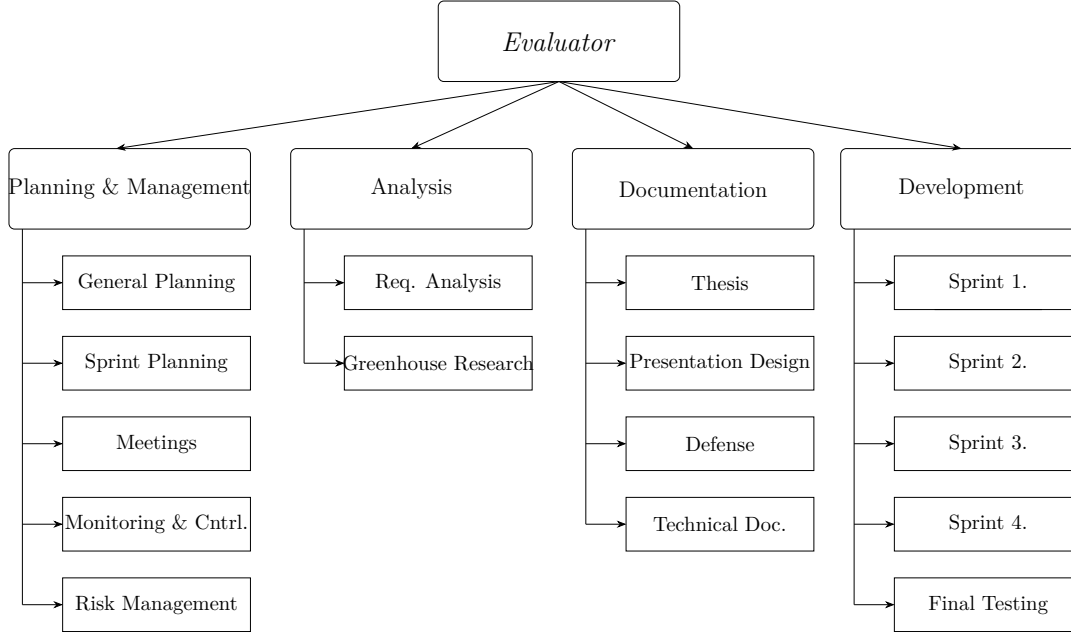
- **Milestone 1:** This stage of the project corresponds to the end of the first development iteration and reflects around having an initial *bare-bones* application working. It must have all the key technical components of the solution, but without any of the full-scoped requirements implemented. A little bit of all the parts but without detail. The application code should be *spaghetti* at this point in time, as first having a functioning application is prioritized and then in following iterations the code will be abstracted and cleaned up.
- **Milestone 2:** At this point of time in the project the second iteration must be already finishing. Some key functionality improvements must've been made and some code abstractions must have started to take place. The focus should have been put mostly on functional features, trying to add a lot of characteristics to the application while making sense of the whole functional picture. Code-wise, this time some initial refactoring must have been made and future iterations should have been envisioned.
- **Milestone 3:** At this moment of time, the third iteration of the project should be wrapping up. Key aspects of the code quality must have been addressed, and most if not all key features of the application must be functioning *reasonably well* <sup>4</sup>. All the main features of the application must have been done and the key extension points should have been outlined (and agreed with the client). The preparation of the production application should have been started, and some key aspects should have (at least) have been thought of and designed.
- **Milestone 4:** This milestone marks the end of the *coding* section of the project. This time period covers the last iteration of the *pseudo-cascade* and the final application testing. Everything should have been prepared for the application to be deployed in a production system and most bugs should have been found and stabilized. The resulting code must be of enough quality and the application should meet the defined [requirements](#).
- **Milestone 5:** This last key point marks the end of the project, with the final task being the in person jury project defense. A formal thesis must have been presented and support material for the presentation must've be done. The time frame of this milestone is variable and it's subject to change as the project advances and the other milestones develop.

---

<sup>4</sup>For a development environment, not for a production system. This does not mean that bugs won't be present

## 2.3 Work Breakdown Structure (WBS)

In line with the described [objectives](#), the project [life cycle](#) and the desired [milestones](#), the planned work for the project can be divided and structured in several *packages*. The resulting Work Breakdown Structure (WBS) is shown on Figure 2.2.



**Figure 2.2:** Work Breakdown Structure of the Project

This decomposition aligns the project tasks into four main branches: Planning and Management, Analysis, Documentation, and Development.

- **Planning & Management:** This branch encompasses all activities related to organizing and keeping up with the project.
  - *General Planning:* Corresponds to the initial high-level planning of the project. This task is planned to be performed at the beginning of the project (*Milestone 0*).
  - *Sprint Planning:* Involves the detailed, concrete planning for each of the four development iterations. Is mostly composed of defining the specific technical tasks and estimating the effort for each sprint (and each of the specific technical tasks). It might seem repetitive because (in theory) the project will already have been planned at the start of the project, but is a necessary task because the client might *tweak* <sup>5</sup> the project scope in between

<sup>5</sup>As in, not entirely change the project's objectives but demand some technical adjustments or new application features

sprints. The development of this item, as [later](#) explained, will be mostly developed using the *GitHub Projects* tool.

- *Meetings*: Covers all scheduled and ad-hoc meetings with the client and all interested parties.
  - *Monitoring & Control*: Represents the continuous effort of tracking project progress against the plan, managing risks, handling changes, and ensuring the project stays on track along its objectives. This task spans across the entire project duration.
  - *Risk Management*: Corresponds to the management, control and mitigation of all the sudden risks that might materialize during the project execution.
- **Analysis**: This branch focuses on understanding the project’s context, requirements and technical foundations.
    - *Requirement Analysis*: Involves gathering, analyzing, documenting and validating the functional and non-functional requirements of the application. This is a crucial initial step, heavily influencing Milestone 0, but may continue iteratively as needed. Most of this task will be done in correlation with the *Meetings* task, as most of the requirement gathering involves the interested parties.
    - *Greenhouse Research*: Refers to the investigation and selection of appropriate technologies, frameworks, LLM providers and architectural patterns for the project. This includes evaluating different options based on requirements, maintainability and general fitness against the project objectives within the [planned dedication](#). This task is concentrated before the development sprints begin (Milestone 0).
  - **Documentation**: This branch includes all formal deliverables for the thesis and some technical deliverables.
    - *Thesis*: The writing of *this* document, covering all aspects of the project from conception to conclusion. This task is mostly defined for the last section of the project, but some key parts of the thesis (specially those related to planning and requirements) could be written before.
    - *Presentation Design*: Preparation of presentation materials (slides, posters, videos, demos) for the defense of the project.
    - *Defense*: The actual in person, before a jury, oral defense of the project.
    - *Technical Documentation*: Creation of documentation related to the software product itself, such as deployment guides, key algorithm diagrams, developer instructions and potential user manuals. This task is performed alongside the implementation during the development sprints.



- **Development:** This is the core branch where the software application is designed built, and tested. It follows the iterative approach defined in the project [life cycle](#) and each sprint details are those aligned in its corresponding [milestone](#). A final application testing is included, such that when all the development is finished a top level approach is taken and the system is tested as a whole.

### 2.3.1 Dependencies Between Tasks

Having defined the [WBS](#) and taking into account the project's milestone system, a clear set of dependencies emerges between the various tasks. The project's progression is heavily influenced by the milestones, acting as checkpoints that validate the completion of prerequisite tasks before moving forward. Nevertheless, certain flexibility can be found in the defined planning.

Developing the life cycle mentioned idea of a three-phase project, very hard dependencies create in the initial sections of the project, specially in between the requirement analysis and the greenhouse research, and the start of the development iterations. Another hard dependency is found between the last iteration and the general project testing, as it cannot start until the development has been finished.

On the other hand, the rest of the project is (by design) more *flexible* in between tasks. In the development iterations the tasks are defined just before starting the development, so dependencies are avoided by the simple definition of the system. Regardless, the documentation tasks can overlap other chores, just as the meetings and the monitoring and control.

## 2.4 Time Estimates

The project has a hard fixed time dedication of 300 hours. The project must be done by a single person, with some occasional mentoring of an advisor.

The project executor currently is working, and expects to be working for the full time extension of the project *part-time* in a private company, so in a normal working day he has (and will have) available  $\sim 3$  hours to work on the project. Of course, with eventual peaks and troughs in availability. This schedule can also affect the possibility of conducting meetings with the interested parties (more on that [later](#)).

Based on these starting points, a task-detailed breakdown and a *tentative* schedule have been designed. As with all advance plannings, all the numbers displayed in this section are subject to change throughout the project and are sensible to the project [risks](#).

### 2.4.1 Dedication

The estimated dedication time for each task can be found on Table [2.1](#).

Task	Dedication
<b>Planning &amp; Management</b> (46,5h)	
General Planning	3 hours
Sprint Planning	3.5 hours
Meetings	20 hours
Monitoring & Control	10 hours
Risk Management	10 hours
<b>Analysis</b> (13h)	
Requirement Analysis	6 hours
Greenhouse Research	7 hours
<b>Documentation</b> (65,5h)	
Thesis	55.5 hours
Presentation Design	7 hours
Defense	0.5 hours
Technical Doc.	2.5 hours
<b>Development</b> (175h)	
Sprint Iterations	160 hours
Final Testing	15 hours
<b>Total Sum</b>	
Evaluator Project	300 hours

**Table 2.1:** Time Estimates for Each Task described in the [WBS](#)

All *sprint development* hours have been grouped up in one big chunk. The philosophy behind this sets the workflow in a *continuous refinement* mode. The concrete, granular task planning is done at the start of each sprint for each iteration, so the remaining hours and the agreed [objectives](#) are taken into account in order to define the scope and all the features that will be implemented for the application. Each planning should be done carefully so that the objectives are met within those 160 hours, possibly compromising features and/or other secondary requirements. This is an agreed upon system and as a safe measure *meetings* with the client are planned before each *Sprint Planning*.

Also, some *just-in-case* time has been scheduled in the *Risk Management* task. These hours can serve for controlling and mitigating possible risks or unplanned events that might come up during the project execution.

### 2.4.2 Tentative Schedule

The project has its fixed start date on the **last week of January 2025**. The initial tasks that are planned are those mentioned in the project [life cycle](#) and the ones corresponding

to [Milestone 0](#).

After the first week, in the initial days of **February 2025**, the project should kick off with the development iterations. These iterations have a idealistic approximated duration of  $\sim 2.5$  weeks each, starting with a client meeting and the *Sprint Planning* and finalizing with its corresponding milestone met.

Continuing with the estimations, more or less the development iterations should finish in the first-second week of **April 2025** and (giving time for the Easter festivities) the *Final Testing* should finish the last week of that same month. The thesis could be written throughout **May 2025** and the project could finish on the first-second week of **June 2025**, with the defense prepared and the final thesis text reviewed by the thesis advisor.

This generous planning estimates make room for any unexpected complications or unmanaged risks, taking into account the peaks and troughs that the daily hour availability might face, with the *tentative* objective to submit the project **before June 22nd, 2025**

## 2.5 Information and Communication System

In this section, the information and communication system that underpins the *management* of the project is described. These are the suite of tools, storage strategies, workflow definitions and communication channels that ensure consistency and traceability throughout the development life cycle.

### 2.5.1 Storage and Backups

As some flexibility is needed when developing (mainly for easily programming on the go, sending code fragments or soliciting advise), the project is primarily stored on the online *git* hosting service *GitHub* <sup>6</sup>. For the thesis, because collaboration between the advisor and the project executor is needed, the online L<sup>A</sup>T<sub>E</sub>X hosting service *Overleaf* <sup>7</sup> is used.

In terms of backups and data storage, the popular 3-2-1 rule [8] is used. Three copies are at all times available, with two of them in different types of media and one of them being offsite.

In this case, the repository is available on the *git hosting* provider, locally on the project developer's laptop and on an external hard drive. This satisfies the two types of media rule (online vs on hard drives) and the safeguards are ensured because the online *cloud* copy is not onsite.

The laptop and the online copy will be in sync at all times (simply by the how the *git* workflow development works) and the hard drive copy will be made every 2-3

---

<sup>6</sup>[www.github.com](http://www.github.com)

<sup>7</sup>[www.overleaf.com](http://www.overleaf.com)

## 2. PROJECT PLANNING

days. The project thesis will be synced at the end of each working (writing) day from *Overleaf* into *git*, ensuring the same level of protection as the code.

As some [communications](#) can take place via *e-mail*, those specially sensitive ones or ones that will need to be referenced later will also be saved on the laptop and on the hard drive. These communications will not be synced to the *git* repository and will remain private.

### 2.5.2 GitHub Projects

To maintain all information related to the project and facilitate the monitoring and control (specially from specific development iteration sub-tasks), the *GitHub Projects* <sup>8</sup> management tool has been set up.

This system is specially convenient because it integrates nicely with the existing *git* hosting (*GitHub*) and allows for organizing tasks into *issues*. These issues can have a lot of metadata attached to them, like dedication, task status, pending features, milestone control, etc.

A screenshot of a *task* in the *GitHub Projects* tool, along with all its related metadata highlighted in green rectangles can be seen on Figure 2.3.

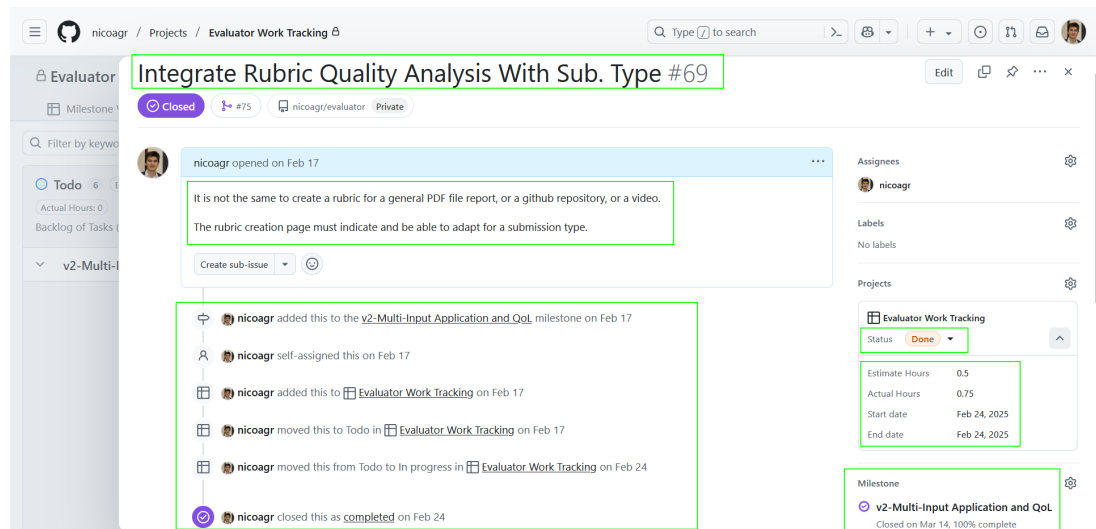


Figure 2.3: Specific Sub-Task from Milestone 2 on the GitHub Projects tool

This system also allows for displaying the repository's *issues* list in a Kanban layout, which enables quick visualization of pending, completed and in progress tasks.

<sup>8</sup><https://docs.github.com/en/issues/planning-and-tracking-with-projects/learning-about-projects/about-projects>

### 2.5.3 Communications

Communications in this project will take place using different methods and channels :

- **Email:** Main communication method. Will be used for all kinds of communications and mainly asynchronous or punctual ones. Emails will remain stored in the mail client (and sometimes be [backed up](#) in case they need to be accessed later).
- *Face-to-face* Meetings: Mainly to be held at the start of the project and specially in the requirements capture life cycle [phase](#). This method ensures the most effective and direct communication and provides clear information exchanges.
- Virtual Meetings: As for most of the duration of the project the executor will not be available for *face-to-face* meetings, communications will be held via the online video conferencing tool *Google Meet* <sup>9</sup>. Schedules and availabilities will need to be acknowledged at a very initial point in the project to ensure consistency, a correct monitoring and control and for the general success of the project.

As the defined [life cycle](#) has a very active involvement of all the interested parties, there will be different types of meetings that can be held. All of these will involve some sort of decision making and will have implications in the active development of the project.

- **Intermediate Delivery:** This type of meetings will be held at the end of each development sprint, in which the progress in the application will be showcased and the next steps will be discussed. Mostly focused on the *features* of the project and the general focus points in order to plan the next sprint.
- **Information Gathering:** This type of meetings will mostly be centered around information gathering and reaching agreements about key components or features. Specially important at the start, most of these will be held *in-person*.
- **Status Update & Feedback:** Held at any moment in the projects time span, this meetings will be all about obtaining quick feedback, making decisions and overall following the projects development. This meetings will be held when the amount of information to discuss via *e-mail* becomes unfeasible.

## 2.6 Risk Management

Effective risk management is critical in project management (and specially in software development) to proactively identify, assess, and mitigate potential threats to the project success. This section outlines the possible risks that can occur and their mitigation plan.

---

<sup>9</sup><https://meet.google.com>

### ◆ R1 LLM Reliability and Uptime

- **Description:** The main key *feature* of the application, as outlined in the [objectives](#), is centered around LLMs and the quick feedback that they can provide. These LLMs calls are made to external providers hosted online. This means that the entire application relies on *external* actors. Apart from the [monetary cost](#), this hard dependency creates a possible risk. These providers may be unreliable, have interruptions in their service or even shut down their operations.
- **Impact Level:** Medium-High
- **Prevention Plan:** To mitigate the risk of a single provider being unreliable or shutting down, the project will encompass several LLM providers and provide tools (specially automatic fail overs) to alternate between them. Also, a general, high-level testing <sup>10</sup> of the reliability of the providers will be done at the start of the project (in the *greenhouse research* task).
- **Contingency Plan:** If this risk were to break out and *all* providers would prove themselves unreliable, the project [scope exclusions](#) would have to be reevaluated and some primary objectives would be compromised. Certainly, the primary aims would switch up and a renegotiation would take place.

### ◆ R2 Production Ready Compromises

- **Description:** As one of the main [objectives](#) of the project is to produce a *production-ready* application, some compromises will have to be made in terms of the available dedication for *feature development*. Multiple aspects of software quality will have to be controlled, such as testing, *i18n*, software quality, deployment preparations or security implications. This could mean that some planned features might not make it to the final application and the product might become less complete.
- **Impact Level:** Medium
- **Prevention Plan:** As defined in the [life cycle](#), the concrete planned features and the granular tasks are defined as the project evolves (before each iteration), so that the interested parties are able to choose and prioritize the most convenient features while maintaining the production readiness.
- **Contingency Plan:** If after most of the iteration processes all interested parties meet on the fact that more *functional* features are needed and that the project needs a bit more refinement to be complete, a re-planning in time dedication might break out. Time from other tasks could be relocated into the iterations, using up the *Risk Management* time.

### ◆ R3 Personal Issues

---

<sup>10</sup>And not by any means a formal viability analysis, just a superficial comparison and some occasional monitorization

- **Description:** The project has been designed as a single-person job. The executor is mostly (with the occasional mentoring of an advisor) responsible for the whole success of the project. This reliance on only one person opens up a world of *personal-level* issues that can arise. The executor might demotivate, might catch an illness or even might have reduced availability for long periods of time.
- **Impact Level:** Medium
- **Prevention Plan:** All major life events that can occur (and that can be foreseen) will be planned and the impact into the project dedication will be analyzed and prioritized accordingly. Throughout all the meetings with the interested parties the issues will be brought up and tasks will be managed accordingly.
- **Contingency Plan:** As this project has a high level of priority, several contingencies might be evaluated in case the the project meets with too many roadblocks. The *part-time* (private company) job schedule could be subject to renegotiation and/or the project defense could be pushed back to the next available occasion (in this case, September 2025).

#### ◆ R4 Low Client Implication

- **Description:** The concrete, specific functional features that the system must have for each iteration of the life cycle remain purposely ambiguous for allowing this project to coexist in a rapid *scope-changing* environment. This in itself supposes a risk, in the terms that if the client is not implicated enough with the project, there will be no one to provide feedback and the executor will be left to develop an application without an idea of which features that application should have.
- **Impact Level:** Medium
- **Prevention Plan:** The compromise with the client will be evaluated on the initial meetings, verifying availabilities and ensuring proper guidance. If this risk is debated at an early stage of the project, it can be moderately mitigated by performing the necessary re-plannings.
- **Contingency Plan:** If the client proves insufficient implication to follow the defined life cycle at a late stage of the project, then the project executor can assume the role of the client and decide by itself the features that the application should have. This is a moderate compromise but allows for the project to continue forward. As a side note, if the client *leaves*, then it must be allowed to *come back* only if it's verified that no *180 degree* changes in scope would be provoked.

#### ◆ R5 Schedule Deviations

- **Description:** This risk refers to the possible time deviation from the [tentative schedule](#) when completing milestones. Some unexpected delays may occur and some sub-tasks might take longer than expected.
- **Impact Level:** Medium
- **Prevention Plan:** Special focus on the *monitoring and control* task will be put, so that if any delays start to appear the appropriate remediations can be applied. Nevertheless, the development iterations have some flexibility applied to them, as the granular concrete tasks are defined just before starting the iteration.
- **Contingency Plan:** If the delays start becoming significant, a *Status Update & Feedback* meeting will be scheduled in order for the interested parties to apply the necessary remediation actions and to prioritize (or de-prioritize) some tasks for the current and next iterations. This could mean *breaking* the sprint and modifying the scope in the middle of it, applying the corresponding needed flexibility.



# Application Requirement Analysis

In order to properly define the scope of the project and align the end result with the system **objectives** (and with the **scope exclusions**), a requirement analysis is presented in this chapter. Most if not all of the contents outlined here have been discussed (and agreed) with the interested stakeholders.

## 3.1 Users and Stakeholders

### 3.1.1 Stakeholders Identification

There are several interested parties in ensuring the success of the project. The main stakeholder is the **project executor** itself, as this project is going to be submitted as a thesis for partial completion of the Bachelor's degree in Computer Engineering. Because of that, this is a supervised project by a professor in the Faculty of Informatics. That makes the **advisor to the project** also an interested stakeholder in the success of the project.

Apart from the academic context, one of the main objectives of the application is to create a *production-ready* application, a system that is be able to be used by real users. The main reasoning behind this objective comes from the existence of an interested **client** that will offer the application to some users, and will be looking out for a good project conclusion. That offered system will be deployed on a *production* machine; therefore a **system configurator** that appropriately configures the application and maintains the system will also want to see through the project.

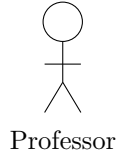
The stakeholders are identified in Table 3.1.

### 3.1.2 Application Actors Identification

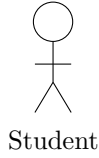
The application has three main actors, or *user roles*, that can operate with it.

Stakeholder Role	Person
Project Executor	Nicolás Aguado
Project Advisor	Juanan Pereira
Client	Juanan Pereira
System Configurator	Nicolás Aguado

**Table 3.1:** Stakeholder Identification



Professors create assignments and review student submissions. They provide rubrics for the LLM to automatically grade submissions and have tools to review and generate automatically those same rubrics.



Students deliver assignment submissions and obtain feedback about them. They can perform a finite number of retries and have different ways of sending in their submissions.



Administrators manage the application setting cost limits, configuring LLM models, generating credentials and organizing courses.

#### 3.1.3 Ideal Application Flow

The system should be designed to provide the following top-level *event flow* <sup>1</sup>. This should be the usual way the application is used, following by how the normal operations are designed and in what order does each action trigger.

1. First, the **administrator** provisions the system and performs an initial configuration (LLM API keys, email configuration, course creation, etc)
2. After that, the **administrator** generates the credentials for the professors and assigns them to their respective course.
3. Then, *for each course* the **professor** (or possibly the **administrator**) generates the credentials for the students, and assigns them to his/hers course.

---

<sup>1</sup>Not in the software engineering meaning of the word, but rather a general interactive *application usage* approach

4. The **professor** has a solution to an assignment that he/she wants to publish, so it uploads it to the application and a rubric that can evaluate works similar to the solution is generated automatically. In the case the **professor** already has a rubric, it can be reviewed using an LLM to check for correctness, ambiguity, etc.
5. Then, the **professor** creates an assignment for its **students** to deliver, using the generated rubric, the reviewed rubric or a different one.
6. After that, each **student** delivers their submission to the assignment, and gets feedback. This should be possible to be repeated for as many times as the professor has specified when creating the assignment in Step 5.
7. Then, when the assignment deadline hits, the **professor** reviews the students' submissions and has already available for him/her the LLM provided *triage* for grading the assignments.
8. Repeat from Step 4 for each assignment in a course and from Step 3 for each academic year.

#### 3.1.4 Functional Requirements

As outlined in the main [objectives of the project](#) and taking into account the target [users of the application](#), the functional requirements of the application can be divided in four main blocks (the general requirements, and specific ones for each application actor). These are the agreed upon requirements that the resulting application must fulfill in order to meet the project's scope.

In a **general** case, the application must provide

1. Authentication that allows for a guest to *login* into the system. A registered guest (user, from now on) must be able to have one or more *roles*. Each role will correspond to each identified application [actor](#) and it will have a set of possible actions to perform.
2. A framework to be able to have separate *courses* <sup>2</sup>, each with different assignments, grades, *email alerts* and users. Each user can belong in zero or more subjects, and he/she must be able to switch between the assigned courses using the application interface.
3. Localization *support* <sup>3</sup> for three languages: Spanish, English and Basque. Guests and users must be able to switch language at any time and the application must update accordingly. Also, if a user selects a language, from that point onward, all communications with them must be done in that language (emails) too.

---

<sup>2</sup>As in, separate academic subjects

<sup>3</sup>As in, just supporting the different languages, not taking into account the quality of the translations

### 3. APPLICATION REQUIREMENT ANALYSIS

---

For the **students**, the platform must have

1. A background process *queue* that collects all ungraded submissions and uses LLM API calls to evaluate them. Following the *production ready* objective, this background process should be fault-reliant and generally persistent. This means that if at time of evaluating a submission a provider is down, the application must try to evaluate the submission with another provider, and if that provider is also down, with another one. The system must *always*<sup>4</sup> grade a submission, even when compromising on evaluation quality<sup>5</sup>.
2. A form that allows for submitting assignments. That form must have several controls that validate due date, retry attempts and user permissions. A submission can be of different types, with the default implemented ones being *text* submissions, *PDF files* and *GitHub* repositories. When submitting an assignment, an student must be able to view the rubric that the LLM will use to grade that assignment, along with a professor provided description.
3. An *indicator* that describes in one word for each submission if their work has enough quality, according to the rubric criterion. In either case, the indicator must be accompanied of a text justification with references to the rubric.
4. A centralized view that allows students to check their grades and their submissions in each course. The view will provide download options, information about the evaluated assignment and metadata for each submission.

For the **professors**, the system must provide

1. A intuitive tool that allows them to generate evaluation rubrics given a solution to an assignment. That solution can be of any of the types accepted in the application. The LLM prompt must ensure that the generated rubric covers the solution correctly and addresses<sup>6</sup> quality aspects such as Completeness, Clarity, Fairness and Structure.
2. A framework that enables an analysis of rubrics. Given an already created rubric, an LLM will analyze it, provide feedback, and suggest improvements. It is assumed that the professor has already thought out all the relative *evaluation*-like guidance but maybe how it is expressed is not as ideal as it could be. So, this tool checks for issues in four categories: Clarity, Completeness, Fairness and Structure. That way, the professors can check for ambiguities and/or expression errors that could otherwise lead to an incorrect evaluation.

---

<sup>4</sup>Except when all free providers are unavailable and the course has reached its spending limit

<sup>5</sup>This is a debatable choice, but taking into account that the LLM evaluation is meant to be just a *supporting material* for the professor's grading process, this compromise is deemed acceptable

<sup>6</sup>In the sense that the rubric text covers the solution without lacking in any of the mentioned quality attributes

3. A form for creating new assignments. Information such as the assignment title, the submission type and the due date must be customizable. The form must also allow for importing a generated (1) or reviewed (2) rubric, along with a manually inputted rubric.
4. A centralized view that enables professors to check their students' grades for each assignment. The view will provide download options, excel exporting, the main submission grade *indicator* and a link to a more detailed view for each submission. In the more detailed view, the professor must be able to view the LLM feedback and provide extra comments.

For the **administrators**, the application must contain

- A user management interface that provides credential generation and profile management. This system must allow for email communication of credentials, single and bulk user creation (for example, importing the users' details in an excel file). A platform that allows for managing the courses should also exist. Users must be able to be added, removed and added in bulk to courses.
- A logging platform that saves a record for every action that happens in each course (and specially every LLM call that is made). Metadata must be saved about the user performing the action, a timestamp and the monetary cost of that action.
- A LLM management interface that allows for system administrators to provide global API keys to enable certain paid LLM models. The system should also provide a toggle to choose which model is used for which submission type, for each tenant.
- A cost management interface, that allows for tracking the spending for each tenant and hard limits that disable spending at all. As a side note, the planning currently contemplates two areas of possible recurring costs: LLM API calls and S3 cloud storage.

### 3.1.5 Non Functional Requirements

Non-functional requirements define the quality attributes and operational characteristics that the application should fulfill to (among others) achieve an overall success in a real-world production environment. A division in five key categories for this type of requirements has been made.

#### Performance

##### 1. Form Submission Times

### 3. APPLICATION REQUIREMENT ANALYSIS

---

- **Metric:** Submission form response times under increased user load.
- **How to measure:** Monitor time taken to process a *text* submission in peak load time. Subtract from that time the actual upload time of the file (inspecting the network requests or via debugging) and the submission upload time is obtained.
- **Objective:** Processing time of  $< 1s$ .

#### Security

##### 1. Logging

- **Metric:** Audit log completeness.
- **How to measure:** Review logs for all critical actions (logins, submissions, grading, config changes).
- **Objective:** 100% of critical actions are logged with user, timestamp, action details and cost.

##### 2. Tenant Isolation

- **Metric:** Course and student data isolation.
- **How to measure:** Penetration testing and code review for cross-tenant and cross-student data access.
- **Objective:** No data leakage between tenants.

#### Reliability

##### 1. LLM Calling Robustness

- **Metric:** LLM provider fail-over success rate.
- **How to measure:** Log and analyze incidents where the primary LLM provider fails and system switches to backup.
- **Objective:** 100% of failed LLM requests are retried with more than one alternative provider.

#### Maintainability

##### 1. Quality Scanning

- **Metric:** Code quality (linting, test coverage).
- **How to measure:** Automated code analysis tools (SCA) and security analysis tools (SAST) test reports.
- **Objective:** 0 critical linting and security errors; Default quality gate passing

### Non-Measurable Requirements

These requirements have no quantitative metric; but they are nevertheless necessary to the project. They are listed below along with their descriptions.

1. **Dockerization:** The application must be able to run in a Docker container environment. Docker is required to simplify the management of different components and make deployments easier. In addition to easing deployment, using Docker containers allows for stricter security policies (more on that, [later](#)).
2. **Documentation:** The project must include *some level* <sup>7</sup> of documentation for developers and system maintainers for outlining the system functionality and specially deployment and development operations.
3. **Unit Testing:** The project should include a suite of unit tests to ensure, at least, the correct basic operations of the service and good maintainability thereafter.
4. **Load Testing:** The service must undergo load tests to analyze its performance under different workloads. This analysis enables (a rough) estimation of the computing resources required to run the service.

## 3.2 Use Cases

Following a *top level* <sup>8</sup> overview, the use case diagram for the application is shown on Figure 3.1. This diagram follows along the already mentioned [functional requirements](#) integrated within the [application actors](#).

## 3.3 Domain Model

The project's domain model is presented on Figure 3.2. Because the project will utilize an Object Relational Mapper (ORM), the actual end resulting `models` file will be very similar to the presented diagram (more on that, [later](#)).

For the schema design an hybrid modeling strategy is adopted. A clear differentiation is made between high-cohesion domain concepts (each materialized as its own relational table) and auxiliary or per-object configuration data, which is persisted as semi-structured JSON or verbatim text within an existing table.

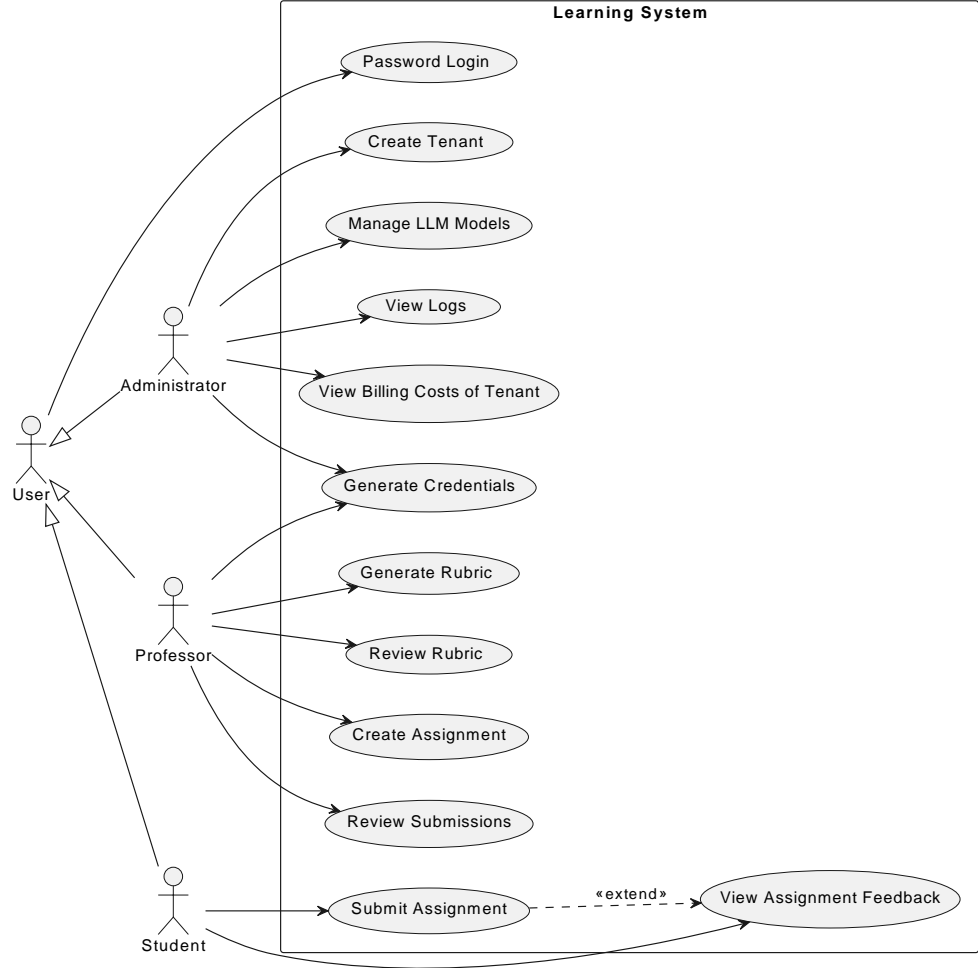
1. **First-Class Entities:** The `User`, `Tenant`, `Log`, `Assignment`, `Rubric`, `Submission` and `Settings` entities are modeled as discrete tables, each with a well-defined set of custom *tailored* <sup>9</sup> columns. Nevertheless, the `Settings` table will be purposely

---

<sup>7</sup>As in, not a detailed guide, but rather informative practical schemas

<sup>8</sup>Referring to the lack of detail when describing features

<sup>9</sup>Minimizing each attribute length



**Figure 3.1:** Top-Level Use Case Diagram of the Application

ambiguous, allowing for several settings to be saved and providing the flexibility of a *key-value* store but with the added properties (ACID) of a relational database.

2. **Semi-Structured Attributes:** Following along the idea of flexible attributes, certain properties have been modeled to exhibit some domain variability that would otherwise provoke onerous migration burdens or impractical *join* chaos if promoted to full tables. Rather than introduce narrow *configuration* entities (for example, `submission_models` or `mailverbosity` in a tenant), these flex-fields are embedded as `VARCHAR` columns holding JSON payloads.

By using normalized tables for core entities alongside JSON/text columns for flexible attributes, this design ensures strong relational integrity while accommodating for evolving domain requirements.



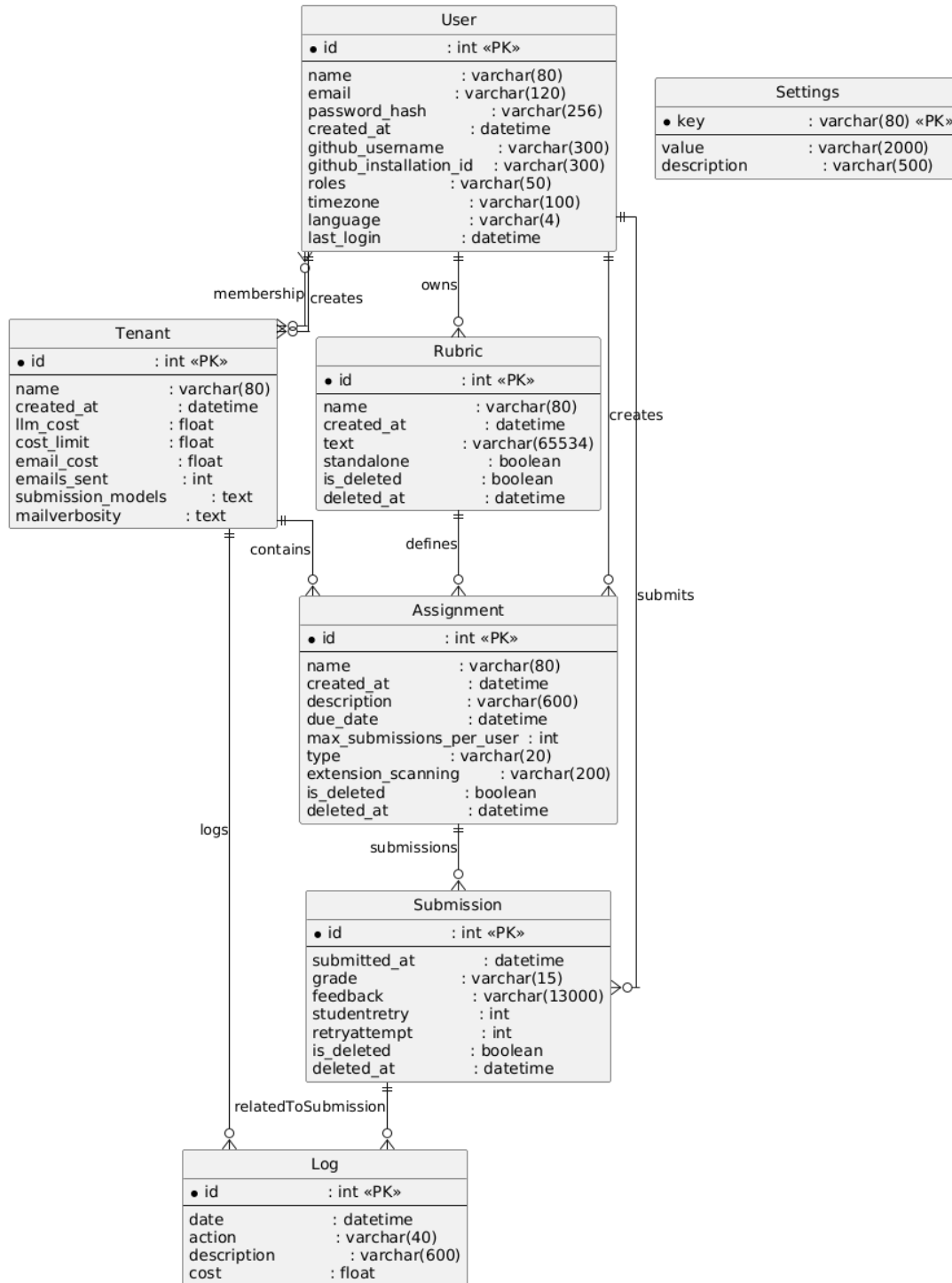


Figure 3.2: Domain Model of the Application



# Architectural Design

This architectural blueprint of the system delineates the primary components, the relied on external providers and the interactions between them. The chapter also describes the different environments the system can be in and its different configurations.

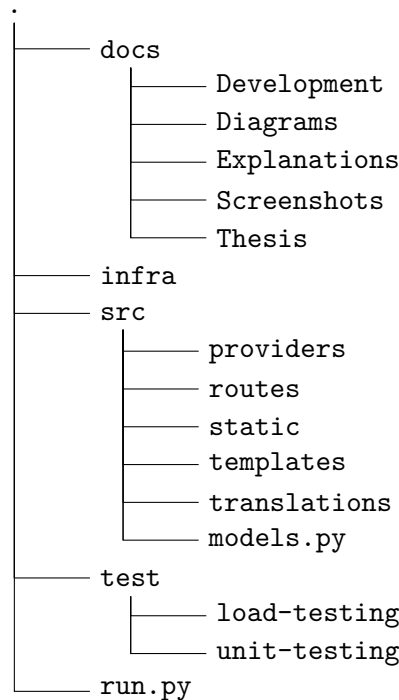
## 4.1 Project Structure

The project is contained within a file system structure. The defined folder structure is shown on Figure 4.1.

The **docs** folder centralizes all written artifacts that support the project's lifecycle, including requirements, design rationales, user manuals and the thesis itself; serving both as a repository for formal documentation and as a living record of technical specifications and revisions. The **infra** directory contains everything needed to deploy and manage the project: scripts, configuration files, and possible *docker* templates for provisioning servers.

The **src** folder houses the core application code: the business logic, web-service *routers*, front-end assets, templates, database logic and any modules that power the system's functionality (explained in detail in Section 5.3). Finally, the **test** directory groups all automated validation suites (unit tests to verify individual components and performance tests) providing the quality assurance backbone that underpins continuous integration and delivery.

On the root directory of the project (.) some code related and miscellaneous files can be found, such as translation configuration files, the main code entry point, *.gitignores* and dependency files.



**Figure 4.1:** Filtered Overview of the Folder Structure of the Project

### 4.2 Component Overview

In an architectural sense, there are several components that make up the whole application system. Components marked with an asterisk (\*) are *optional* <sup>1</sup> and may be omitted depending on the environment configuration.

- **Web Application:** The main system. This component refers to the front and back end parts of the application, as well as the basic web server listening for requests. It's what users interact with and the main platform orchestrator (communicates with queues, databases and storage). In the development environment, the web server bundled with the framework is used for quick testing. On the other hand, aquatinting to the [chosen framework](https://flask.palletsprojects.com/en/stable/deploying/) and its production recommendations <sup>2</sup>, a reliable and performant basic web server for serving the `wsgi`-typed application in a production environment is `uWSGI` <sup>3</sup>.
- **Relational Database:** The place for storing application data. Primarily holds information implementing the described [domain model](#), while not containing user submission data. In a development environment materializes as a `SQLite`

---

<sup>1</sup>To the system operations

<sup>2</sup><https://flask.palletsprojects.com/en/stable/deploying/>

<sup>3</sup><https://uwsgi-docs.readthedocs.io/en/latest/HTTP.html>

<sup>4</sup> database and in a production environment is represented by a *PostgreSQL* <sup>5</sup> database. This flexibility is made possible by using a database-agnostic **ORM**, allowing for an easy debugging experience for developers and a robust battle-tested platform for production systems <sup>6</sup>.

- **Background Workers:** To prevent the main application from becoming unresponsive, computationally expensive or asynchronous operations (such as grading submissions and dispatching notification emails) are offloaded to background worker processes <sup>7</sup>. By delegating these tasks, the application preserves its responsiveness in the foreground and enables horizontal scaling across multiple worker instances. All background jobs are processed by the workers in FIFO order from a queue.
- **Redis In-Memory Database:** Main volatile storage for the application. *Redis* <sup>8</sup> has a lot of applications, but in this specific context the main implementations refer to caching the users' web sessions (K-V storage) and as a message queue for transmitting jobs to the background workers. It's an ideal use case for both features, as for web sessions speed is chosen in favor of persistence (users can just log back in) and for message queues *Redis* provides a solid reliable implementation. Volatility for the message queues is handled through a background job that checks for any unhandled messages (more on that, [later](#)).
- **\*\* Reverse Proxy:** For production environments, having a central application entry point enhances performance, scalability and security. The usual provider of choice in the industry tends to be *nginx* <sup>9</sup>, but for this specific project, given the [chosen framework](#), a reverse proxy that speaks the *uwsgi* protocol (specifically spoken by the application's web server) seems appropriate. *Lighttpd* <sup>10</sup> is a well-tested high performance proxy that adequately corrects to the project's demands and has some added performance benefits, thus making it an ideal choice for the project configuration.
- **\*\* S3-like Storage:** Depending on the deployment, the system configurator can choose where the students' submissions are saved. In addition to the local file system, the application supports saving data in an *S3-like* cloud storage provider (more in detail, in [next section](#)).

---

<sup>4</sup><https://sqlite.org/about.html>

<sup>5</sup><https://postgresql.org/about>

<sup>6</sup>This does not mean that *SQLite* is not well versed for a production environment, but rather that between the available options, and as a subjective opinion, *PostgreSQL* was selected for real-world use

<sup>7</sup>That have the same database and storage connection capabilities of the main application

<sup>8</sup><https://redis.io>

<sup>9</sup><https://nginx.org>

<sup>10</sup><https://lighttpd.net>

### 4.3 External Providers

For both development and production environments the system depends in external actors for its normal operations in a substantial way. The project dependencies are here acknowledged and detailed.

#### 4.3.1 Large Language Model Providers

As already outlined in the project [objectives](#) and [risks](#), the main *key* component of the application is the LLM grader.

To avoid having a central point of failure, the application is configured with 5 LLM providers: *Google* <sup>11</sup>, *Anthropic* <sup>12</sup>, *GroqCloud*, <sup>13</sup>, *OpenAI* <sup>14</sup> and *together.ai* <sup>15</sup>. These providers will be called via an HTTP API and the relevant authentication is done via *API Keys*. For the models without a free tier, the system provides Administrators <sup>16</sup> an easily configurable UI framework for inputting and swapping the corresponding *API Keys*. In the case the provider has a library for the chosen framework language, it will be the one used instead of the HTTP API call.

Nevertheless, using most of these dependencies involves a monetary cost. A financial analysis is later provided in Chapter 7.

#### 4.3.2 E-Mail and Github

The system also relies on *e-mail* sending for notifying students of actions that occur or that they trigger on the platform. For maximizing compatibility on a variety of environments, the platform has been configured to support two types of mail configurations. The preferred option is sending messages directly through an *SMTP* server providing an account's credentials and the secondary method relies on using the external closed-source provider *MailJet* <sup>17</sup>. Both solutions work exactly the same and are just differentiated by the environment variables that have to be set-up (at deployment) when executing the main application server.

Another dependency we can find on the system is the *GitHub* <sup>18</sup> auth integration. This is a necessary component for sending GitHub repositories as assignment submissions. This integration is used both to verify that a student is an actual collaborator of the code project and to download the repository contents (for the cases that the code is closed-source). A *zero-trust* model approach has been developed, only saving the minimal amount of information from the user, requesting the minimum required

---

<sup>11</sup><https://ai.google.dev/gemini-api/docs>

<sup>12</sup><https://anthropic.com/api>

<sup>13</sup><https://groq.com>

<sup>14</sup><https://platform.openai.com/docs/models>

<sup>15</sup><https://docs.together.ai>

<sup>16</sup>As in, the application actors administrators

<sup>17</sup><https://mailjet.com/products/email-api/>

<sup>18</sup><https://docs.github.com/en/apps>

*GitHub* permissions and following the intended flow, with access tokens, user codes and installation ids.

### 4.3.3 Blob Storage

As mentioned earlier, the application can support either local storage or a remote *cloud* storage for saving students' submissions. In both cases a framework for managing the used up storage is provided and a *hard limit* can be set up to control operating system corruption and/or unanticipated monetary charges.

On the one hand, for saving students' data in the local server storage the folder route `/opt/evaluator_nicoagrdata/` is used as a primary mounting point. A *UNIX-like* system is assumed and the permissions handling is done automatically when executing the (*docker* deployment) production scripts.

On the other hand, the *de-facto* standard protocol for application storage in *cloud* providers is *S3*, so this protocol has been the one selected for implementation. Specifically, to protect the students' privacy and guarantee proper security controls, the system uses *pre-signed* URLs that allow for download only under specific conditions and from trusted origins.

Notable *S3-like* storage providers that provide the necessary security features (URL pre-signing) are *Amazon AWS* <sup>19</sup>, *Cloudflare* <sup>20</sup> and *BackBlaze* <sup>21</sup>.

Using a *cloud* storage provider can have some benefits, like providing fault tolerance, but can also have some downfalls, like the created vendor dependency and possible **cost overheads**.

## 4.4 Dockerized System

For the development environment and the normal *maintenance* workflow, it is enough to configure the local machine following Appendix A.1. Nevertheless, to simplify deployment, enforce strong isolation between components and enable easy horizontal scaling, the entire application stack has been containerized using *Docker* and *Docker Compose* <sup>22</sup>. Two distinct **production** configurations are provided.

- A **standalone** deployment, in which the web server container obtains and renews its own **SSL** certificates (using *Certbot* + *Lighttpd*) and is designed for listening in port 80 and 443 of the machine. The complete `docker-compose.yml` file for the system can be found on Appendix B.5.

---

<sup>19</sup><https://aws.amazon.com/s3>

<sup>20</sup>[www.cloudflare.com/developer-platform/products/r2](https://www.cloudflare.com/developer-platform/products/r2)

<sup>21</sup>[www.backblaze.com/cloud-storage](https://www.backblaze.com/cloud-storage)

<sup>22</sup><https://docs.docker.com/compose/>

- A **colocated** deployment, where all application containers sit behind an external reverse proxy (for example, an on-site Nginx or *cloud* hosted load balancer), and thus do not manage TLS themselves. This configuration is most useful when the machine is shared with other services and other unrelated containers are running alongside the evaluator system.

Both configurations share the same core services: application, database, Redis, background workers; they differ only in whether the Certbot service for encryption is present and how the front end ports are exposed.

Regardless, by using *Compose*, a network segmentation security advantage is achieved. Virtual networks between each of the different components and its recipients are defined and internet access is restricted to the containers that actually need it. For example, if the database needs to communicate with the web application but not with the internet, the *db-web* internal network would be created for exclusive communication between those two services and the database container would not have the *public-internet* network attached to it. This segmentation ensures that, for example, the background worker replicas cannot directly reach the web server's internal port, and the database is never exposed to the public gateway.

To support high throughput of background tasks (grading, email dispatching), the *background worker* service is declared using `mode: replicated` and `replicas: 3` (or any desired replica count). Each worker replica attaches to the same database and *Redis* queue, and can be increased or decreased within the system simply by simply updating the *replicas* count. The replicas can be accessed with its *Compose*-defined DNS name and *Compose* itself acts as a load balancer. This mechanism also works for the web server instances and provides easy horizontal scaling without changing application code.

### 4.5 Production Deployment

The primary method for deploying the application to a production system is *Docker Compose*. For the system to function adequately, a machine with minimum 2 GB of RAM, 5 GB of storage and a *UNIX-like* operating system with *Docker* and *Docker Compose* is required.

Using the provided configuration templates, adequate security protections come bundled with the system and all the components come in a *production-ready* configuration state.

The reader might refer to the **Deployment Manual** described in Appendix A.2 for a reference in setting up all the environment variables and deploying the system in a production machine.



# Application Design

As the project presents a purely greenhouse development, this chapter presents the core technologies and design decisions, remarking the development of the web application, the selection of database and other key components and libraries. It then exposes the tools and environments employed in the implementation workflow and describes the tiered architecture, while also outlining the security measures integrated at the application level.

## 5.1 Technologies

The chosen tech stack for the project is a **full-stack** *Flask* application (using simple *Jinja2* templates for the front-end), with *SQLAlchemy* as ORM, *Pytest* as the testing framework of choice, *redis-queue* as the framework for worker jobs and *python-babel* for the `i18n` provider. A high level overview of the project is simplified on Figure [5.1](#)

### 5.1.1 Programming Language

The initial architectural decision concerned the choice of an implementation language. Given the system's roots in the artificial-intelligence domain (limited for now to API-driven operations rather than in-house algorithm development) and the project's inherent need for repeated prototyping and iterative refinement before reaching feature completeness, **Python** was adopted as the foundational development language.

Its straightforward and expressive syntax reduces language complexity and accelerates development, while its flexibility allows seamless adaptation to evolving requirements. Moreover, the extensive standard library and rich ecosystem of third-party packages facilitate rapid integration of different domain components.



Figure 5.1: High Level Overview of the Technology Stack for the Application

### 5.1.2 Web Framework: Flask+Jinja2

When choosing web technologies, several choices and compromises have to be made taking into account the specific project needs. Sometimes a flexible, *front-end* complex state management is needed, while other times a performant complex backend has more importance. The chosen technologies most of the time arrive to a good in-between point in that balance. For the evaluator project's specific needs, the application is not as interactive and simple HTML forms are more than enough. Regardless, the backend does indeed need a good foundation and is going to handle complex processing.

On the **backend** side, *Flask* <sup>1</sup> stood out among Python frameworks by striking the right balance between minimalism and extensibility. *Django* <sup>2</sup>, while robust and extensively used, proved too opinionated with its pre-made components, middleware layers and routing conventions. It could affect future customizations and made implementing the project's requirements too complex. By contrast, Flask's blueprint system offers modular route organization, its middleware hooks allow for fine-grained request handling and response processing, and its ecosystem of extensions (for forms, authentication, database integration, etc.) lets the application grow organically. Ultimately, Flask delivers a complete GET/POST-oriented solution that remains agnostic to specific libraries, empowering rapid prototyping today and confident scalability tomorrow.

Following along the chosen backend technology, the *Jinja2* <sup>3</sup> templates form an ideal **front-end** *combo* with Flask for a largely form-driven application where rich interactivity is minimal. By relying on server-rendered HTML with vanilla JavaScript for any dynamic behavior, the evaluator platform avoids the overhead of bundling and

---

<sup>1</sup><https://flask.palletsprojects.com>

<sup>2</sup><https://djangoproject.com>

<sup>3</sup><https://jinja.palletsprojects.com>

state-management logic inherent to modern single-page frameworks. Instead, templates are populated directly with data on each GET or POST, keeping the front-end codebase maintainable while ensuring rapid page loads and predictable performance under heavy assignment-submission workloads. This approach also simplifies debugging and reduces client-server synchronization, since all business logic remains on the server and the front-end serves just as a simple facade.

### 5.1.3 Object Relational Mapper: SQLAlchemy

Object-relational mappers (ORMs) help fill the gap between the designed [domain model](#) and the actual relational database implementation through object oriented programming. These systems enable developers to interact with data through language constructs rather than raw SQL queries, improving developer productivity and maintaining high-level schema definitions alongside business logic (resulting in the `models.py` file being very similar to the implemented domain model). Therefore, its use in the system was a clear decision to make. Another quality feature that distinguishes ORM is the **database agnosticism**, allowing for easy transitions between different backends (for example, *SQLite* for development and *PostgreSQL* for production) without noticeable code modifications.

Following along the language and [framework](#) choice, *SQLAlchemy* <sup>4</sup> stands out as a robust and well-tested solution. Many frameworks use it as an internal tool, it has been thoroughly proven effective for production environments and it integrates nicely with the *Flask* framework. Its support for schema migrations and dialects for virtually every major relational database further support the decision for considering *SQLAlchemy* as the ORM of choice.

### 5.1.4 Testing Framework

In the Python ecosystem there are several approaches to (unit) testing web applications. The scope of available solutions range from built-in frameworks like *unittest* <sup>5</sup> to end-to-end solutions such as *Selenium* <sup>6</sup>. These browser-testing based tools excel at testing real UIs in real environments, but they are quite hard to maintain afterwards and usually imply mocking back-end services externally. This extra layer complicates test setup, and distances the tests (specially in such a *backend* heavy application) from the code paths that actually implement the Flask routes and the *SQLAlchemy* models.

Taking this into account, *pytest* <sup>7</sup> sits in the middle ground and offers native **unit testing** Python support and easy integrations for *Flask* <sup>8</sup> and *SQLAlchemy*. The web application context can be easily spun up and the ORM functions are available

---

<sup>4</sup><https://sqlalchemy.org>

<sup>5</sup><https://docs.python.org/3.3/library/unittest.html>

<sup>6</sup><https://selenium.dev>

<sup>7</sup><https://docs.pytest.org>

<sup>8</sup><https://flask.palletsprojects.com/en/stable/testing>

with minimal boilerplate. *Pytest*'s *fixture* system allows for keeping tests concise and expressive, while its parametrization features enable for covering multiple scenarios in a single function.

Apart from the usual unit testing, **load testing** is crucial for uncovering performance bottlenecks and ensuring system stability in production systems. *Apache Benchmark (ab)* <sup>9</sup> is the *de-facto* standard for load testing, offering a comprehensive feature set (customizable concurrency levels, detailed performance metrics and extensible reporting), serving as a valuable insights source for performance tuning the application.

Nevertheless, **SCA and SAST** scanning are essential for catching security vulnerabilities and code quality defects early in the development process, thereby reducing technical debt and improving maintainability. *SonarCloud* <sup>10</sup> is the *de-facto* standard for static (python, among others) analysis, it offering a variety of quality related features and proper *GitHub* CI/CD integration, making it an easy addition for the project.

### 5.1.5 Redis-Queue Worker Implementation

By offloading resource-intensive and repetitive tasks to background workers, the main *Flask* server can maintain low-latency response times and improve the overall system efficiency. *Redis-queue* <sup>11</sup> is a well-supported Python library that provides a straightforward job framework for en-queuing, scheduling and executing asynchronous tasks. This framework (as its name implies) **conveniently** uses *Redis* as the message queue. It is a mostly minimal configuration that allows for *background* and *foreground* code to live in the same codebase and utilize the same helper functions (and **ORM**); thus making it an ideal fit for the system.

### 5.1.6 Internationalization (i18n) with Flask-Babel

Internationalization is a fundamental requirement for the evaluator application because it will operate in a diverse, multilingual environment where students, professors and administrators interact in different native languages. The objective is to be able to decouple the user-facing texts from the source code, reducing duplication and improving flexibility.

*Flask-Babel* <sup>12</sup> is an ideal choice because it provides the standard `__()` translation function within the *Flask* request context and *Jinja2* templates, centralizes all message strings in isolated files and offers built-in support for forcing locales for when outside the request scope (on background workers, for example). Its command-line utilities for extracting, updating, and compiling translations ease by a notable margin the localization pipeline, making it simple to keep translations in sync with evolving application texts.

---

<sup>9</sup><https://httpd.apache.org/docs/2.4/programs/ab.html>

<sup>10</sup>[www.sonarsource.com/products/sonarcloud/](https://www.sonarsource.com/products/sonarcloud/)

<sup>11</sup><https://python-rq.org>

<sup>12</sup><https://python-babel.github.io/flask-babel/>

## 5.2 Development Tools

These development tools have helped not only accelerate coding, debugging, and testing, but also have improved code quality and maintainability throughout all the life cycle.

In terms of the primary IDE for developing the project, the application does not have any specific requirements that dictate for an specific platform or another. Regardless, the chosen IDE has been *Visual Studio Code* <sup>13</sup>; whose lightweight footprint and extensible plugin system (specially the debugger and the python IntelliSense support) provide for a *friction-less* coding experience.

For quick database modeling and inspection, *DB Browser for SQLite (BD4s)* <sup>14</sup> has been employed to visually explore table structures and data rows in the local development database (that *conveniently* is just an SQLite file). For visualizing the volatile *Redis* database its family product *Insight* <sup>15</sup> has been used. Also, application-level testing has been made using the *Mozilla Firefox* <sup>16</sup> browser, whose built-in Developer Tools enable for easy JavaScript debugging and direct network inspection.

## 5.3 Application Tiers

When designing the target structure of the application and how the code was going to be organized, several approaches were considered. The Model-View-Controller (MVC) approach is a very common pattern, specially for web applications. Regardless, it was not deemed a good fit because it lacked the extendibility with *controllers*, in such a way that such a backend heavy application would need a more strict separation with all the processing components and not only with the actual *web request* scope. Specially, the need was to distinguish and extend *HTTP layer* processing vs *core business logic* processing.

In this context, a three-tier application [9] seemed like a good fit. This approach splits the system into presentation, application (or Business Logic), and data tiers. This separation improves modularity, testability and allows each layer to scale or evolve independently. When applying this pattern, the resulting folder structure for the project (in the `/src` folder) is shown in Figure 4.1.

### 5.3.1 Presentation Layer

The presentation tier is responsible for all UI-related functionality. In the project, the contents of the `templates/` and `static/` directories implement this tier.

- In `templates/` the *HTML* pages using *Jinja2* syntax that compose pages are defined. The application layer will directly inject the necessary data and the

---

<sup>13</sup><https://code.visualstudio.com>

<sup>14</sup><https://sqlitebrowser.org>

<sup>15</sup><https://redis.io/insight>

<sup>16</sup><https://mozilla.org/firefox>

composer will put together all the different templating parts to spit out a final *HTML* for the client-side.

- The `static/` folder holds *CSS*, *JavaScript*, images and other assets for the front-end logic. In other contexts this section alone could be an entirely separate project, but in this specific project very little interactivity is needed from the front-end side and just simple stylings and basic *JavaScript* functionality is provided.

### 5.3.2 Application and Business Logic Layer

The application tier orchestrates the route handling of the application and the related business logic that is called from it. In the Flask application this is embodied by the `src/routes/` blueprints, the `src/providers/` modules and the `run.py` startup point.

- Each file in `src/routes/` (`auth.py`, `student.py`, `professor.py`, etc.) defines one Flask blueprint and a *high-level* application part. These functions parse incoming requests, decide which business logic routines to invoke and gather data for final rendering to pass onto the presentation layer.
- The `src/providers/` folder contains the *core* backend functionality. These files will be the ones that allow for the heavy processing and that will utilize software engineering patterns that'll allow for modularity. Routes call into these providers rather than performing the logic inline, ensuring each concern remains isolated and that the application the necessary extensible points.
- The `run.py` file in the root of the project serves as the initial entry point for the application. It is the main orchestrator and thus it contains all the setup logic that enables or disables certain aspects of the application at startup. The initial database connections (both relational and volatile), the internationalization defaults, the security connections, the blueprint registering and the environment variable management are all handled in this initial startup file.

Because providers may themselves invoke auxiliary utilities (for example, for submission types may be categorized in folders), the application structure layers dependencies in a downward direction: routes  $\rightarrow$  providers  $\rightarrow$  submission types. This structure makes it easy to add new endpoints or swap implementations (for example, replacing one LLM service with another or evaluating a PDF or GitHub submission) without touching the routing layer.

### 5.3.3 Data Access and Domain Layer

The data tier manages persistence of the application users' data and enforces schema constraints. In the application, the core of this tier resides in `src/models.py`, where the *SQLAlchemy* ORM models are defined and each model object contains auxiliary functions to perform pure data operations within that object.

- Each class in `models.py` maps directly to a database table (a first-class domain entity), with relationships and constraints expressed declaratively. Each class also contains direct methods to functions (class operations) that are used across all other blueprints and business logic providers.
- As referenced in the [domain model](#), the relational database sometimes stores JSON information directly in text-structured fields inside tables. In this file an abstraction is provided for using these fields, such that the internal JSON schema is not altered and the data handling operations are eased out and centralized in one place.
- As with the usual advantages when using an ORM, some direct access to the schema is made along all the business logic and within the routes themselves. For that matter, this file has to be consistent and with a certain fixed schema, so during testing phases extra care is put here.

## 5.4 Application Security

Multiple layers of security best practices have been implemented in the *Flask web application* to reduce the attack surface. First off, all secrets and configuration parameters (database credentials, API keys, cryptographic salts) are obtained from environment variables stored in `.env` files, ensuring proper controls with *Git* and CI/CD integrations.

Then, the *flask-talisman* library <sup>17</sup> is used to enforce HTTPS (in production environments) and to inject a strict Content Security Policy (CSP) that restricts the origins of scripts and styles. As or front-end CDNs will be used (libraries will be served locally) and no external API providers are used (in the front-end), the security policy is mostly restrictive to only allow local domain code execution and local domain network requests.

Session data is stored server-side in *Redis*, and cookies are marked (again, in production environments) as `Secure` and `HttpOnly` to guard against session hijacking and client-side script access.

Error handlers are centralized and flask is configured in such a way so that stack traces and sensitive internal details are never exposed to end users. Also, all database tables are automatically created within the application context (in the startup script) to avoid ad-hoc migrations that might expose schema details.

Cross-site request forgery (CSRF) protection is enabled via *flask-seasurf* <sup>18</sup>, which automatically generates and verifies per-session CSRF tokens. Extra care has been taken to include this token in every *HTML* form and in the `X-CSRFToken` header for AJAX calls; with failing to do so resulting in 403 rejections, effectively blocking unauthorized form submissions.

---

<sup>17</sup>[www.github.com/wntrblm/flask-talisman](https://www.github.com/wntrblm/flask-talisman)

<sup>18</sup>[www.github.com/maxcountryman/flask-seasurf/](https://www.github.com/maxcountryman/flask-seasurf/)





# Implementation

This chapter covers the *core* of the system implementation. Even though the full system contains many more pieces, only the most impactful modules are shown here. The entirety of the code and implementation details mentioned here are in its *final product* state, and reflect the end resulting productive application.

## 6.1 LLM Fault-Tolerant Grading and Model Interchanging

Large-language models (LLMs) serve as the *key* grading engines for the evaluator system. Yet, these models and their hosting providers form a highly *dynamic* ecosystem: New models come out, pricing tiers shift and outages can interrupt the evaluation pipeline at any moment. On the other hand, the application [requirements](#) mandate that every student submission is graded (within budget) regardless of which LLM is available, even if doing so compromises grading quality.

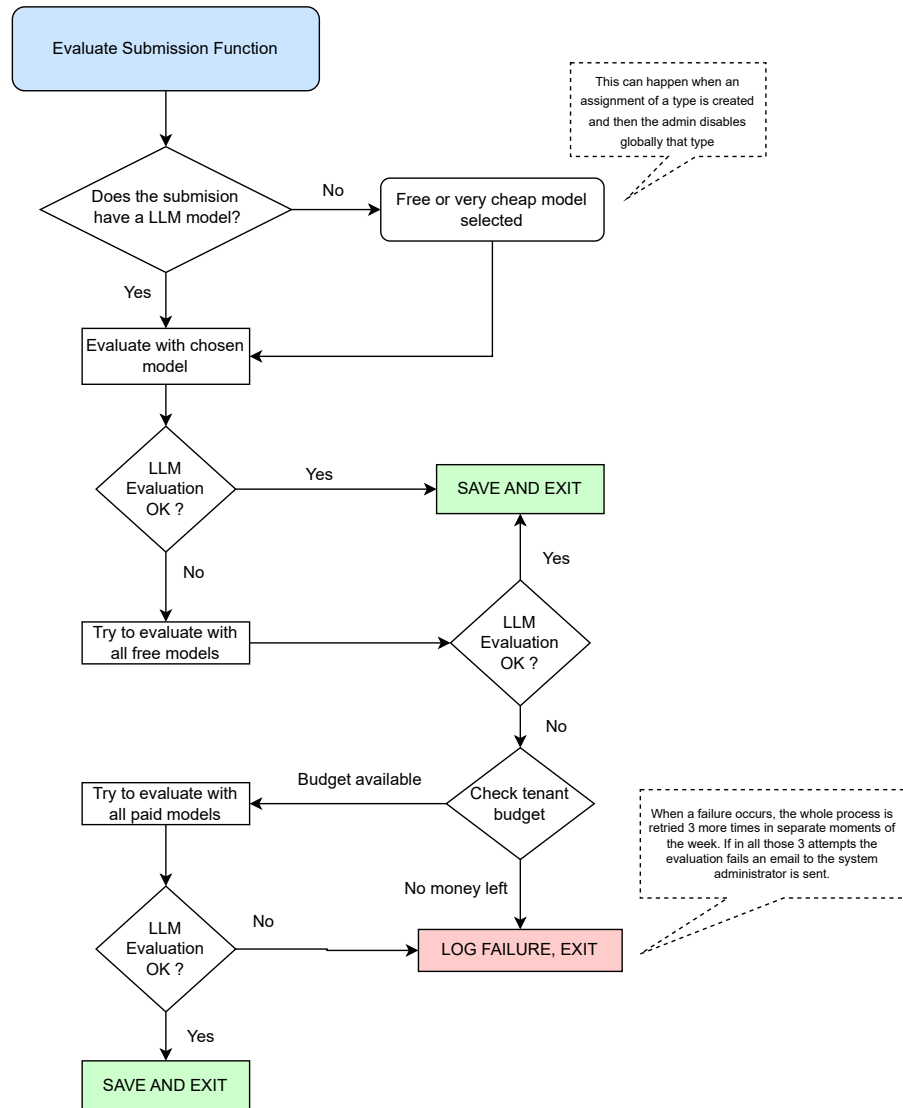
To satisfy these constraints, two complementary subsystems have been designed. First, a Multi-LLM Evaluation Tool that orchestrates multiple model providers, automatically retrying and rerouting requests to alternative endpoints upon failure or budget exhaustion. Second, an Ultra-Modular LLM Model System that allows for replacing LLM models or pricing tiers without changes to the core application (while seamlessly integrating with the grading process).

For the **Multi-LLM Evaluation** tool, the following key design points have been taken into account.

- Each submission type (PDF, GitHub, etc.) within a course can be assigned a *preferred* LLM model. The administrator panel (see Figure [6.2](#)) exposes these per-course settings, so that in most operations the grader will just invoke the designated model without further intervention.

## 6. IMPLEMENTATION

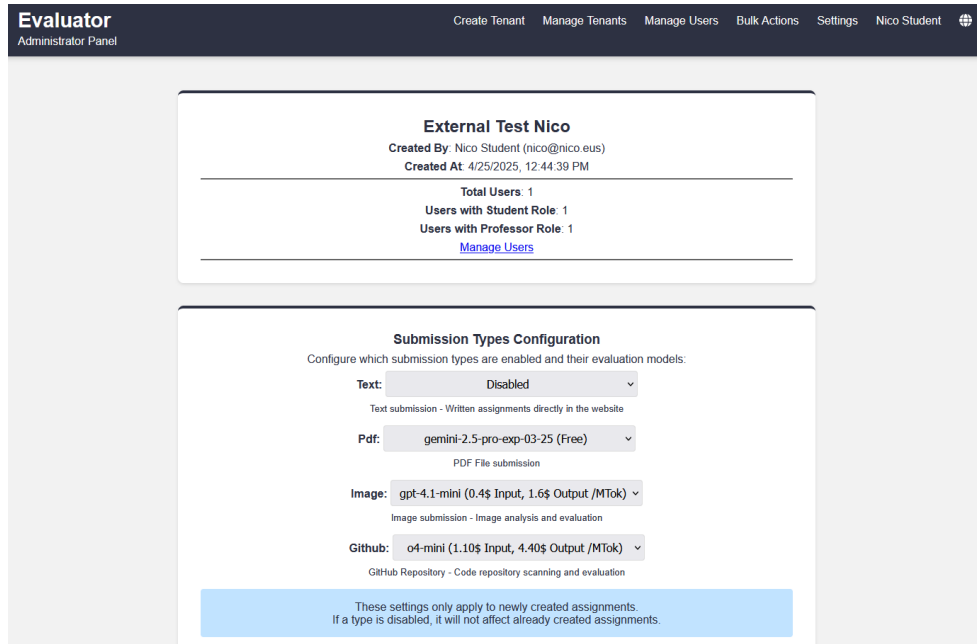
- If a request to the preferred LLM fails (for example, due to an API error) or the course's budget is exhausted, the tool automatically triggers the fallback selection routine. This routine consults a ranked list of alternative models and providers, retrying or rerouting the request until one succeeds or all options are depleted. That way, it is ensured that every submission receives a grade (even though compromising the evaluation quality). The high-level flow of this logic is shown in Figure 6.1.



**Figure 6.1:** High-Level Overview of the LLM Fallback Function

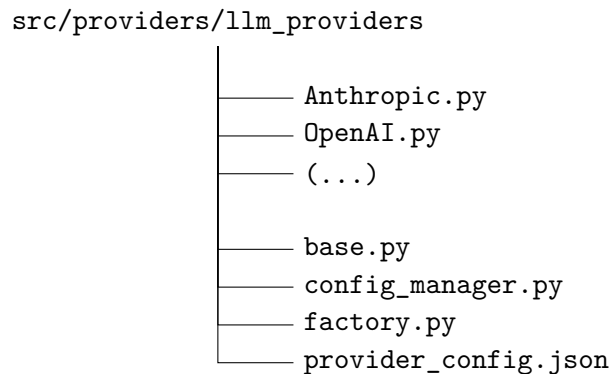
## 6.1. LLM Fault-Tolerant Grading and Model Interchanging

The core *persistent evaluation* function, responsible for orchestrating retries and budget checks, is implemented as detailed in Appendix B.1. An extended explanation on the specific implementation details is also contained in that Appendix.



**Figure 6.2:** Snippet of the Specific Tenant LLM Options in the Administrator Panel

In order to complement the high-availability guarantees of the Multi-LLM Evaluation tool, an **Ultra-Modular** LLM model system has been designed to handle provider changes using an extensible interface. Applying software engineering practices and an adapted *Factory* design pattern, every individual LLM model offered by the providers can be added, replaced, or reconfigured without touching the core grading logic. The directory layout for this subsystem appears in Figure 6.3.

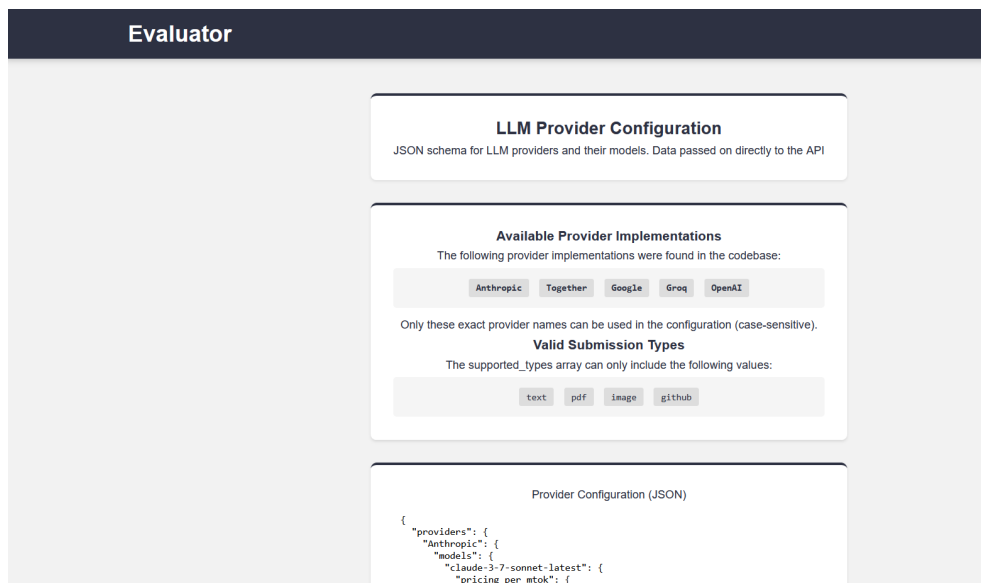


**Figure 6.3:** File Structure of the Modular LLM System

## 6. IMPLEMENTATION

---

- The `base.py` file defines the abstract base class that declares the common LLM interface for all providers to implement and also provides shared helper methods useful to every provider.
- The `factory.py` file functions as the model registry and runtime router: it reads the active model and pricing-tier selection from `provider_config.json`, instantiates the corresponding provider subclass, and dispatches all LLM requests. Because all provider related actions go through this single *factory*, seamless provider interchange is guaranteed.
- The `provider_config.json` file serves as the single source of truth for the available models, provider routing endpoints and cost tiers. The `factory.py` file is a wrapper around it and all logic comes directly from the defined JSON schema. This file is directly editable via a web UI interface (see Figure 6.4).
- Because the JSON schema is a sensitive piece for the application, managing all models, the providers and the pricing, a helper file attached to the web ui interface (`config_manager.py`) has been created. It validates `provider_config.json` against a predefined schema and coherence rules, preventing misconfiguration and providing controlled changes.



**Figure 6.4:** High-Level Overview of the LLM Selection Function

All along, thanks to this modular organization, the Multi-LLM Evaluation tool can confidently rely on the factory to select and fallback to any configured model, while the platform administrator retains full control over the provider roster without requiring code changes.

## 6.2 Cleanup and Reassurance Process

Following along one of the mentioned [requirements](#), the system is responsible for making sure that every student submission gets processed, even when compromising in evaluation quality. On the first evaluation attempt, every model within budget is tried. If all the available models cannot successfully produce a result, the submission is marked as *pending*. Regarding the same use case, the *Redis* queue may become unavailable or a bug in the containerization process may occur. Thus, some submissions can actually become stuck *pending* for grading.

To that end, a **cleanup routine** has been defined. This job is wired both into the application's startup sequence (so that if the server was down at the scheduled time the pending work is picked up) and into a daily scheduler that fires at 3 AM (CEST). The objective of these periodic executions is double:

- Ensure that the grading process for any *pending* submission is re-tried.
- Alert the administrator about submissions that have been retried several times and have failed.

Code-wise, this cleanup process is directly engraved onto the *Flask* application constructor framework (`run.py` file) and has several helper functions around it. The library that implements the workers (*Redis-Queue*) already has some built-in scheduling functions available, but for this specific scenario a tailored wrapper around it has been designed.

Both an immediate execution path and a recurring schedule for the cleanup routine have been implemented on top of *Redis-Queue*. To simplify operations within the library, each cleanup run enqueues the next one as soon as it completes. To prevent duplicate or recursive scheduling (particularly when crossing day boundaries in certain timezones, daylight savings times or invoking the job manually) the system first inspects the *Redis-Queue* job registry and filters out any existing entries before enqueueing. The implementation of the helper functions is shown in Listing 6.1.

```

1  def run_cleanup_and_retry():
2      job = queue.enqueue(worker_cleanup_and_retry)
3      return job
4
5  def schedule_cleanup_and_retry():
6      from src.providers.timeops import get_next_3am
7
8      # Check if there's already a scheduled cleanup job
9      registry = ScheduledJobRegistry(queue=queue)
10     scheduled_jobs_ids = registry.get_job_ids()
11     for job_id in scheduled_jobs_ids:
12         job = queue.fetch_job(job_id)

```

## 6. IMPLEMENTATION

---

```
13         if job.func_name ==  
14             'src.providers.queue.worker_cleanup_and_retry':  
15                 return job  
16  
17         # Schedule new job only if none exists  
18         print("Scheduling cleanup and retry job at: ",  
19             get_next_3am())  
20         job = queue.enqueue_at(get_next_3am(),  
21                               worker_cleanup_and_retry)  
22         return job
```

**Listing 6.1:** Helper Functions Around the Cleanup Process

That way, when booting the application from a fresh start (Listing 6.2) and after each cleanup (Listing 6.3) the process is again scheduled for another run.

```
1 def create_app(run_cleanup=True, web_app=True):  
2     load_dotenv()  
3  
4     (...)  
5  
6     if run_cleanup:  
7         app.schedule_cleanup_and_retry()  
8         app.run_cleanup_and_retry()
```

**Listing 6.2:** Cleanup Process on Fresh Application Starts (run.py)

```
1 def worker_cleanup_and_retry():  
2     from run import create_app  
3     app = create_app(run_cleanup=False, web_app=False)  
4  
5     with app.app_context():  
6  
7         (...)  
8  
9     # Schedule the next cleanup job  
10    current_app.schedule_cleanup_and_retry()
```

**Listing 6.3:** Cleanup Scheduling for the Next Day (worker)

### 6.3 Role Authentication Framework and Session Management

Following along the clearly defined and separated [application actors](#), a custom made authentication framework has been designed for the application, structuring the components around a modular **role system**. A registered user of the application can have one or more *roles*, with each of them having separate menus, functionalities and texts. Practically, this is like creating different separate applications while maintaining the

same backend. Apart from that, a user with a role can also have several *courses* <sup>1</sup>. Multiple functionalities throughout the application are intended to have a course level separation (for example, when submitting an assignment for a specific course) and benefit of having this information set up at an authentication level. This two-step *pseudo-permission* system allows for easy separation of concerns and improved benefits for the common application grounds.

In a more technical deep-dive, user roles are defined at the data-model level and they are loaded (with the rest of the user's information) into the current session when the user logs in. Upon successful authentication, if the user has one or more roles <sup>2</sup>, the user is redirected to a dedicated role selector endpoint, where he/she can select one out of the available roles that he/she has. The chosen role is then stored in the session object and the user is redirected to the specific *blueprint* <sup>3</sup> that corresponds to the selected role's application. If that blueprint is configured to require a course <sup>4</sup>, then the user is redirected to the course handler. In a similar fashion, if the user contains one or more courses then a selector is shown, if not a transparent login is made.

Because of the whole range of permission requirements an individual *HTTP* route may have, the method for personalizing access to each specific route has been designed in a small suite of decorators in the file `decorators.py` (Having the complete file showcase in Appendix B.2). These helpers (and the possibility of stacking them) allow for checking all the required types of permissions that a possible application route may need.

- The `@login_required` decorator simply checks if a user has logged in into the system before allowing access onto the *HTTP* route.
- The `@course_required` decorator similarly verifies the presence of a chosen course, issuing a redirect to the course-selection view if the check fails.
- The more complex `@role_required(*roles)` decorator allows for each function to restrict the access of a user for a subset of the available roles. For example, if the function is decorated with `@role_required(['student', 'professor'])` then that *HTTP* route is only going to be available to users with the active role `student` or the active role `professor`.

Each decorator is implemented with `functools.wraps` to preserve endpoint meta-data, and their minimal, single-responsibility design yields high *reusability* across the entire application.

---

<sup>1</sup>As in, academic subjects

<sup>2</sup>If the user only has one role, that role is selected and the whole *permissions selection* process is transparent

<sup>3</sup>Blueprints refer to specific secondary http routers in the Flask framework

<sup>4</sup>For example, the administrator role does not have a course selected globally *per se*

This custom made authentication system has also been designed to provide some scalability measures. The *flask-session* <sup>5</sup> library is used for storing all the internal user session data into the *Redis* key-value store. This shifts the session storage from the client to the server, improving in application security measures and benefitting scalability.

```
1 app.config['SESSION_TYPE'] = 'redis'
2 app.config['SESSION_KEY_PREFIX'] = 'evaluatorsession:'
3 redis_host = os.getenv('REDIS_HOST', 'localhost')
4 redis_port = int(os.getenv('REDIS_PORT', '6379'))
5 redis_password = os.getenv('REDIS_PASSWORD', None)
6 app.config['SESSION_REDIS'] = redis.Redis(
7     host=redis_host,
8     port=redis_port,
9     password=redis_password,
10    db=0,
11    decode_responses=False
12 )
```

**Listing 6.4:** Redis-Session implementation in the run.py Startup Process

So, by externalizing session state into Redis, each application process remains stateless and interchangeable, allowing any server behind a reverse proxy to access and mutate session user objects, roles, etc. This system in combination with [Docker Compose](#) provides nearly immediate horizontal scaling (via the `replicas` parameter), guaranteeing fault tolerance and consistent session delivery.

All things considered, this complete implementation allows for the authentication subsystem to improve scalability and also to be maintained in isolation: routes (blueprints) and auth reside in separated modules, facilitating both unit testing of auth logic and incremental extension of roles or policies.

## 6.4 Submission Type Extendability

In the Evaluator application, a submission type denotes a specific classification of a student assignment solution (for example, a PDF document submission, or a GitHub repository submission); each of which contains distinct requirements for storage, validation, and grading. As performing operations with each type for each requirement is not a trivial task, the *Factory* design pattern has been applied to correctly structure and organize the *type system*.

To manage these variations systematically, the system defines three core components for every submission type: a *SubmissionEncoder* responsible for transforming incoming data into a storable format (and vice versa), a *SubmissionHandler* which enforces pre-submission checks and manages the upload or download of files and a *SubmissionEvaluator* that encapsulates the type specific logic for issuing verdicts and feedback.

---

<sup>5</sup><https://flask-session.readthedocs.io/en/latest/introduction.html>



By isolating these concerns using well-defined classes, this architecture achieves a high degree of modularity and provides clear separation of responsibilities within the whole evaluation flow of events. `Base.py` and `factory.py` files are provided for the type system for managing a single-point of truth for types in the application. This approach is similar of what has been implemented for the [Ultra-Modular LLM model system](#).

- The *SubmissionEncoder* interface cleanly separates how submission content is stored and retrieved from the higher-level logic in handlers and evaluators. For example, in PDFs and images, the *DirectBlobEncoder* implements this interface by storing raw bytes in the local storage or s3 and reconstructing them as file-like streams on download. This uniform contract makes it easy to introduce new encoders, such as an example *ZipRepositoryEncoder* for GitHub assignments.
- The *SubmissionHandler* component centralizes all pre- and post-processing rules for intake and delivery of submissions. For example, *PDFSubmissionHandler*'s `process_submission` method verifies the `.pdf` extension, confirms the file is a genuine PDF with extractable text, enforces the 15 MB size limit and then persists its bytes via the encoder. Its download method uses a shared helper to prepare HTTP headers and either redirect to a signed URL (s3 storage) or stream the file directly (from local storage). This combination of specific checks and shared helpers allows for maximizing re-usability and fostering quick development.
- The *SubmissionEvaluator* component handles the conversion of submitted work into pedagogical feedback. For example, in the PDF file format, the evaluator re-extracts text, assembles the correct prompt and then calls the LLM service with relevant metadata. Its `evaluate` method parses the returned JSON into a verdict and explanatory feedback, and (in another example) the image evaluator follows the same process using vision-capable prompts on binary image data. Again, the same mixed use of shared helper functions and internal type-related management.

Looking ahead for [future work](#), this *BaseSubmissionType* class abstraction could evolve into a lightweight plugin system: third-party or custom submission formats would simply implement and register their own encoder, handler, and evaluator classes (maybe discovered at runtime via entry points or a configurable plugins directory). Such a mechanism would allow institutions to introduce bespoke assignment types (for example, audio transcripts or specific document grading) without altering the core codebase, so long as each plugin adheres to the well-defined interfaces for encoding, handling, and evaluating submissions.

## 6.5 Miscellaneous Quality of Life Improvements

This section presents a collection of extra implementation details that, while not part of the core architecture, significantly improve the user experience and the system

management.

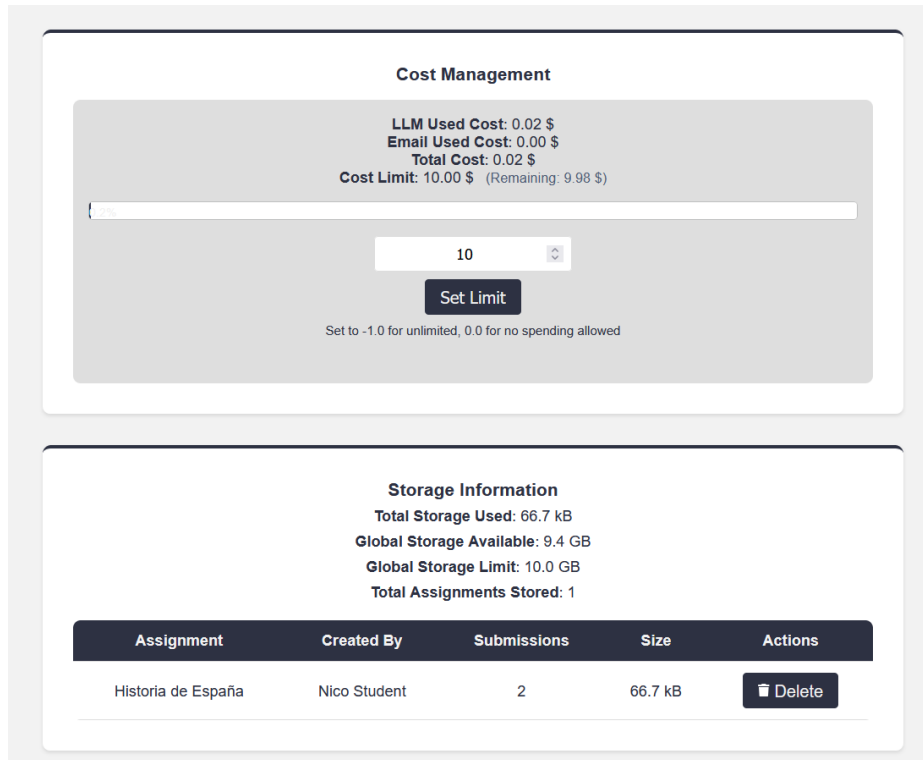
1. Nowadays, smartphones are increasingly used. Because the system aims to appeal to a wide variety of users, the styles of the app have been adapted to correctly work on smaller, vertical screens (**responsiveness**) and the possibility to generate a **Progressive Web App (PWA)** has been implemented. Using the *Workbox* <sup>6</sup> library for service worker tooling, an *Install this web as an app* button has been materialized (with the complete source code in Appendix B.3) inside the profile section. This allows for using the web application as a native app inside the Android or iOS operating systems, appealing to the increasingly large user base that wants to operate with the application using their smartphones.
2. Although the application has S3 storage support, the usual deployment method relies on the local (operating system) storage for saving the students' submission data. With the right number of users, the application can quickly fill up the storage space on the server it relies on, provoking fatal *UNIX* failures and halting normal server operations. To that end, **storage space management** interfaces have been implemented. Firstly, professors and administrators have been enabled to delete assignments. When an assignment is removed, grading data on the database is *soft-deleted* <sup>7</sup> and the actual submission data is completely removed. This is implemented with a background process <sup>8</sup> so to not block the main thread for user-heavy assignments. Apart from that, the administrator has an *storage* management interface for each course, allowing for deeper management on the assignments. As a final measure, a storage *hard limit* environment variable has been created. When the storage space of the complete application hits the defined threshold (in gigabytes), all submissions are paused and an email to the administrator is sent alerting of the problem.
3. As mentioned in previous sections, the LLM grading service for the submissions and the rubrics can incur in monetary costs. When considering a large amount of users and several courses, the necessity to implement a **cost management** system becomes clear. This management interface allows for setting fixed cost limits for each specific course, integrating deeply into the grading and assignment submission systems to control when the limits are exceeded and perform the necessary restrictions. An snippet of the administrator panel for a specific course showing this feature and the one mentioned in the previous point (2) can be found on Figure 6.5.
4. As the application is *meant* to be used in a production environment, there will exist some moments in which debugging of issues and traces are important.

---

<sup>6</sup><https://developer.chrome.com/docs/workbox>

<sup>7</sup>Referring to *marking* the submission as deleted, not actually deleting it. This ensures that in case of accidental deletion, at least the grading and feedback data (although not in a very friendly way) can be recovered

<sup>8</sup>Using the previously mentioned [queue system](#)



**Figure 6.5:** Snippet of the Specific Tenant Cost and Storage Options in the Administrator Panel

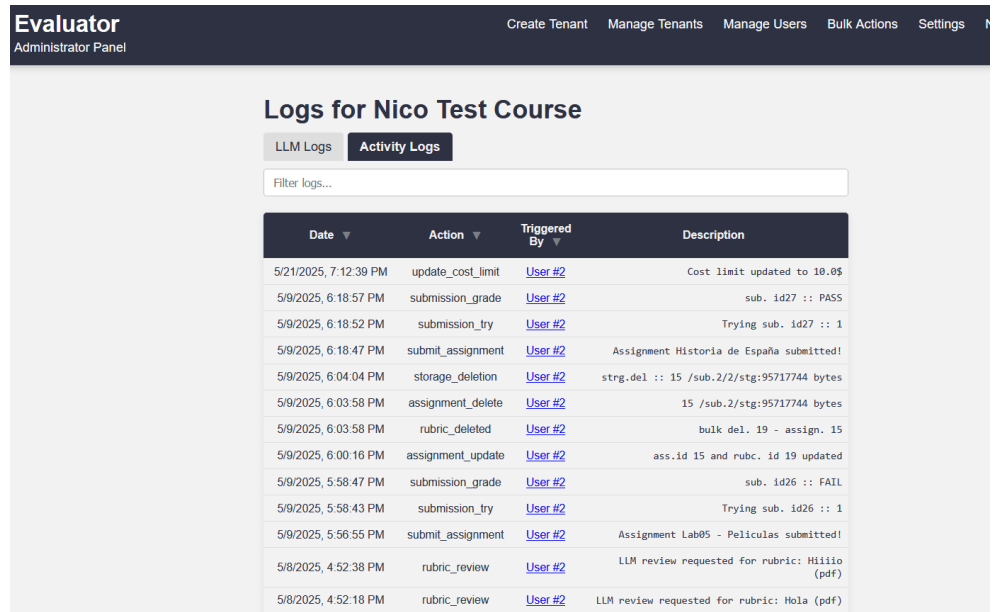
Because of that, a complete **logging framework** has been implemented for the system. Internally, every time *anything* happens inside of a course (assignment submission, LLM model changes, rubric generation, etc.) a log is saved onto the relational database (with the logged information being the one described in the [domain model](#)). Then, the administrator can use the bespoke *log viewer* to check out if an action went well, the pricing cost of a specific LLM call, how the user credentials generation went, and many other events. This is the only interface menu that has not been adapted for a mobile view, given the amount of the information that has to be displayed. A snippet with some examples of the kind of events that can be viewed is found on [Figure 6.6](#).

## 6.6 Project Deployment

After the end of the [life cycle](#) development iterations, the *Evaluator* application has been deployed on a dedicated production server. At the moment of submitting this document, the system is already being tested and used by real users in some specific courses in the Faculty of Informatics of the University of the Basque Country (more on that, [later](#)).

For the reader to freely explore and examine the described system, the production

## 6. IMPLEMENTATION



The screenshot shows the 'Evaluator Administrator Panel' interface. At the top, there is a navigation bar with links: 'Create Tenant', 'Manage Tenants', 'Manage Users', 'Bulk Actions', and 'Settings'. Below this, the main heading is 'Logs for Nico Test Course'. There are two tabs: 'LLM Logs' and 'Activity Logs', with 'Activity Logs' being the active tab. A search bar labeled 'Filter logs...' is present. Below the search bar is a table with the following columns: 'Date', 'Action', 'Triggered By', and 'Description'. The table contains 15 log entries, all triggered by 'User #2'.

Date	Action	Triggered By	Description
5/21/2025, 7:12:39 PM	update_cost_limit	User #2	Cost limit updated to 10.0\$
5/9/2025, 6:18:57 PM	submission_grade	User #2	sub. id27 :: PASS
5/9/2025, 6:18:52 PM	submission_try	User #2	Trying sub. id27 :: 1
5/9/2025, 6:18:47 PM	submit_assignment	User #2	Assignment Historia de España submitted!
5/9/2025, 6:04:04 PM	storage_deletion	User #2	strg.del :: 15 /sub.2/stg:95717744 bytes
5/9/2025, 6:03:58 PM	assignment_delete	User #2	15 /sub.2/stg:95717744 bytes
5/9/2025, 6:03:58 PM	rubric_deleted	User #2	bulk del. 19 - assign. 15
5/9/2025, 6:00:16 PM	assignment_update	User #2	ass.id 15 and rubc. id 19 updated
5/9/2025, 5:58:47 PM	submission_grade	User #2	sub. id26 :: FAIL
5/9/2025, 5:58:43 PM	submission_try	User #2	Trying sub. id26 :: 1
5/9/2025, 5:56:55 PM	submit_assignment	User #2	Assignment Lab05 - Peliculas submitted!
5/8/2025, 4:52:38 PM	rubric_review	User #2	LLM review requested for rubric: Hiiiio (pdf)
5/8/2025, 4:52:18 PM	rubric_review	User #2	LLM review requested for rubric: Hola (pdf)

**Figure 6.6:** Snippet of some log events for the specific *Nico Test Course* in the Administrator Panel

deployment is available at the <https://evaluator.ikasten.io> web address. Upon successful authentication (using the credentials provided in the accompanying documentation), all the implemented features may be freely explored using the different [application roles](#).

# Cost Analysis

The following cost analysis examines the financial implications of invoking large language models for grading assessments. In the evaluator application, *S3* storage is provided as an optional feature, but to honor the truth and provide a worst-case representative scenario, all comparisons are also made taking the storage costs into account. All of the detailed amounts are in US Dollars (\$) <sup>1</sup>, arithmetic rounding has been applied throughout the calculations and the point (.) is used as the decimal separator.

## 7.1 Assumptions

In order to obtain approximate estimates, a set of simplifying assumptions is first introduced. The usage patterns presented here correspond to the upper bound of possible values <sup>2</sup>. Concretely, all costs are here presented related to an example university-level 5 month course modeled as follows:

- The course has 60 students.
- Each student sends in 8 assignments.
- Each submission allows for 3 retries. It will be assumed that all students exhaust the possible retries.
- The submissions are GitHub repositories and each one occupies 75 Mb (each having the correct *.gitignores* in place).
- The submissions contain 15 000 tokens each and the LLM grading output for each one of them is of at least 1000 tokens.

---

<sup>1</sup>For a conversion to each local currency, the *Visa* or the *MasterCard* online calculators are recommended for accurate exchange rates information

<sup>2</sup>More realistic scenarios would certainly represent lower figures

## 7. COST ANALYSIS

---

- The submissions are downloaded 3 times from the storage provider, and they are retained for a minimum of 6 months.

Adding up the math, 1440 API calls are meant to be made to the LLM providers for evaluating student work and 108 gigabytes of storage space are needed for storing all the submissions for a course.

### 7.2 LLM Pricing Estimations

These LLM pricing estimations include, as of the 1st of June of 2025, the newer flagship models from every major LLM provider. The per-provider breakdown can be found on Table 7.1. These prices are calculated for 1440 API calls total (24 for each student), with each having 15 000 input tokens and 1000 output tokens.

Provider	Model	\$ / Call	\$ / Student	Total Cost
Google	gemini-2.5-flash	\$0.00285	\$0.07	\$4.10
Deepseek	r1	\$0.0105	\$0.25	\$15.12
Groq	llama3-70b	\$0.01424	\$0.34	\$20.51
OpenAI	o4-mini	\$0.0209	\$0.5	\$30.10
Google	gemini-2.5-pro	\$0.02875	\$0.7	\$41.4
OpenAI	gpt-4.1	\$0.038	\$0.91	\$54.72
Anthropic	claude-sonnet-4	\$0.06	\$1.44	\$86.4
OpenAI	o3	\$0.285	\$6.84	\$410.4

**Table 7.1:** LLM API Cost Grouped by Provider Model

Please note that the model roster implemented in the final application may vary from the models listed here. This comparison is populated with the flagship models for each provider, but thanks to the [application LLM modularity](#) a variety of other (cheaper, faster or simply better suited for the job) models could be included.

### 7.3 S3 Pricing Estimations

In addition to Amazon S3, this pricing comparison includes remarkable *S3-compatible* providers, since they share identical core features and APIs. Because the system implementation only requires *presigned URLs* for secure access we can directly compare their pricing structures on equivalent service offerings. In this specific scenario, 108 gigabytes of storage are saved and 324 gigabytes of outgoing bandwidth is consumed. To, again, compare on the higher-end of the spectrum, the pricing will be pulled from the *always-hot* blob storage offerings.

It should be observed that because the application has implemented the provider agnostic *S3* protocol, the system administrator has full discretion to select among

Storage Provider	\$ / Month	Total Cost
iDrive e2	\$0.54	\$3.24
BackBlaze b2	\$0.65	\$3.9
Cloudflare r2	\$1.62	\$9.72
OVH Blob	\$2.58	\$15.48
Scaleway Obj	\$3.97	\$23.82
Azure Blob	\$7.56	\$45.34
Amazon S3	\$7.9	\$47.39

**Table 7.2:** S3 Storage Provider Cost Grouped by Provider

the functionally equivalent, protocol-compliant offerings. Consequently, the pricing disparities evident in Table 7.2 reflect intangible considerations and such provider elections are recommended to be handled on a case by case basis.

## 7.4 Remarks

Because it is [out of the scope](#) to evaluate the quality of both LLM and *S3-like* providers, a mid-range selection from Table 7.1 and Table 7.2 has been selected for making the final estimations.

*OpenAI's o4-mini* LLM model is priced at \$0.0209 per API call, amounting to \$0.50 per student and \$30.10 in total for the whole course. The blob storage from the *OVH* french provider is priced at \$2.58 per month, yielding a total of \$15.48 over six months and \$0.26 per student. Accordingly, the combined cost is \$0.76 per student and \$45.58 in aggregate for the whole course. It should be noted that the *S3-like* storage is an optional feature, as local storage may be employed without incurring in any additional fees. Under a *cloud* storage-free configuration, the sole expense arises from LLM invocations, corresponding to \$0.50 per student or \$30.10 in aggregate.

Considering both approaches, the financial commitment remains modest and well within viable limits for a single course offering.

Moreover, the majority of Google's and Groq's models (such as *gemini-2.5-flash* or *llama3-70b*) have very generous free-tier allowances with notably hard to reach rate limits<sup>3</sup>. Consequently, some system deployments (in which the quality of the *free* llm grading is deemed enough) may effectively operate at *near zero* LLM cost, further reducing the overall economic cost.

Nevertheless, it must be emphasized that the LLM market is highly dynamic and subject to frequent pricing and policy changes. These estimations reflect the market state as of June 1, 2025. Additionally, the presented figures constitute conservative,

<sup>3</sup>More information on the following website: [ai.google.dev/gemini-api/docs/pricing](https://ai.google.dev/gemini-api/docs/pricing)

## 7. COST ANALYSIS

---

upper-bound scenarios. Most practical deployments would certainly obtain overall lower figures.

In conclusion, the evaluator application results **economically viable** for educational contexts, with the understanding that actual costs will vary according to provider selections, usage patterns, and future market developments.



# Testing

Comprehensive quality assurance processes are indispensable to ensure the reliability and maintainability of any *production grade* system. As per the project [objectives](#), the guarantee of core functionality (with a well tested and performant product) is prioritized over the abundance of features. This chapter presents several quality measures implemented throughout the whole project [life cycle](#).

## 8.1 Unit Testing

A comprehensive unit-testing strategy has been adopted in the evaluator system to ensure correctness of both the *Flask* logic layer and the underlying data models. The Python testing framework *pytest* was chosen (See Section [5.1.4](#)) for its support of *fixtures* and parameterization.

First, a top-level `conftest.py` fixture file was created to centralize common setup, tear-down and intermediate step logic. The file, found in its entirety in Appendix [B.4](#), was designed to take into account several re-usability cases.

- The Flask application is instantiated in *testing* mode and its database schema is dropped and recreated at the start of each session (ensuring test isolation).
- *Fixtures* for the Flask test client (`client`), CLI runner (`runner`), and authenticated user sessions (`logged_in_professor`, `logged_in_student`) are defined to be used across most of the tests.
- A mock email-sending function replaces real **SMTP** calls with a redirection to `stdout` to avoid any possible (real-world) side effects.

After the initial configuration, the various test cases are developed, with each functionality and each role being separated into its own independent file. Regardless,

the provided test cases refer to the *core* system features. Non-critical *nice-to-haves* (for example, Excel-exporting and bulk credential management) are not covered in unit tests by deliberate scope decision, rather the basic features and functionalities are prioritized for a more complete coverage instead.

- The `test_auth.py` file provides complete coverage for login, logout, role-selection and guard conditions (wrong password, missing fields, invalid HTTP methods, etc).
- The `test_models.py` file is in charge of verifying consistency and coherency within the [domain model](#). Password hashing, role assignment and validation, timezone handling, cost accounting, mail-verbosity controls, submission-type management and type-validation are examples of the multiple domain checks that this file performs. This test suite has relative importance because it ensures that both the HTTP interface and the ORM-layer remain in sync.
- In the `test_assignment.py`, `test_grading.py` and `test_rubric.py` files the professor workflows are tested. Everything from rubric generation (both standalone and from solution) to assignment creation and grading (both manual and automatic) is controlled.
- The `test_submission.py` file validates all the aspects related to the student role. The whole suite of possible actions is tested, ranging from text- and *GitHub*-based submission endpoints, deadline and maximum-submission guards to download endpoints and error handling for invalid payloads or deleted assignments.

All in all, the totality of these defined tests can be executed in *bulk* by using the `pytest` shell command from the root of the project. With that single command, a clear pass/fail summary and coverage report is produced, detailing all the points of failures and a quick glance of what is failing (and where). A snippet of the `stdout` output from a successful test run can be found on [Figure 8.1](#).

Finally, by structuring tests around fixtures (application context, database, authenticated clients) and grouping assertions by functionality, these tests allow for extensions when new system characteristics are developed, speeding up the development life cycle and providing a fault-proof approach for modifying code confidently.

## 8.2 Load Testing

In order to validate the Evaluator application under heavy load environments, a dedicated load-testing layer has been developed on top of the *Apache Benchmark* (`ab`) tool (see [Section 5.1.4](#) for tool selection rationale). Three specially sensible endpoints have been designated for careful examination under test.

- `POST /auth/login` is the primary entry point to the application, handling all the related authentication logic for each specific user and its different roles. It is

```

(venv) nico@laptop:~/Downloads/evaluator$ pytest
===== test session starts =====
platform linux -- Python 3.11.2, pytest-8.3.5, pluggy-1.5.0
rootdir: /home/nico/Downloads/evaluator
configfile: pytest.ini
testpaths: test
plugins: anyio-4.9.0
collected 84 items

test/unit-testing/test_auth.py ..... [ 11%]
test/unit-testing/test_models.py ..... [ 29%]
test/unit-testing/test_professor_assignment_rubrics.py ..... [ 36%]
test/unit-testing/test_professor_assignments.py ..... [ 48%]
test/unit-testing/test_professor_grading.py ..... [ 55%]
test/unit-testing/test_professor_rubric_generation.py ..... [ 63%]
test/unit-testing/test_professor_rubrics.py ..... [ 75%]
test/unit-testing/test_professor_submission_types.py ..... [ 86%]
test/unit-testing/test_student_submissions.py ..... [100%]

===== 84 passed in 99.79s (0:01:39) =====
(venv) nico@laptop:~/Downloads/evaluator$

```

**Figure 8.1:** Console Output from a Successful Test Run with *pytest*

the bottleneck that handles session variables and the primary permissions logic. Without this endpoint, users can't login (and thus, can't interact with the system). For that reason, special care has to be put to verify that this endpoint handles adequate load under high pressure moments.

- `POST /evaluate/new` is the professor's final destination for sending an assignment to a group of students. Whether an assignment gets created, or not, depends for the most part in this endpoint. Although the circumstance on which this specific route would be overloaded is *rare* <sup>1</sup>, it is vital to guarantee that in moments of high demand professors can still access its *core* endpoint and that their service is not affected.
- `POST /learn/submit` is the endpoint for students to submit their assignments. This is the most critical point of the application and the one that is most likely to be overloaded (specially near the end of the deadline of the assignments). Although an entire queue system (Sections 6.1 and 6.2) has been developed specifically to not overload this route, extra care has to be taken and it has to be verified that the main thread dispatcher is efficient enough to handle all student workload.

To perform this testing, first, helper scripts have been created to automate the capturing of valid session cookies for both authenticated roles (professor and student) and unauthenticated users (login). Subsequently, a master script (`run_load_tests.sh`) orchestrates a matrix of ab invocations across a spectrum of request volumes and

<sup>1</sup>Usually this would not be the first route to fail under high load simply because it is not a frequently accessed use case

concurrency levels (having `REQUEST_VOLUMES` and `CONCURRENCY_LEVELS` as adjustable parameters). Each invocation of the script targets one of the three mission-critical endpoints.

Additionally, some predefined data is available in several external files (for example, the assignment data to submit), thus invoking `ab` (for example) as shown in Listing 8.1.

```
1 ab -n 2500 -c 50 \  
2 -p create_assignment_data.txt \  
3 -C "$(cat professor_cookie.txt)" \  
4 -T "application/x-www-form-urlencoded" \  
5 http://localhost:80/evaluate/new
```

**Listing 8.1:** Sample *Apache Benchmark* Invocation with 50 concurrent requests and 2500 total request count

The raw `ab` outputs are captured in per-scenario files and subsequently parsed by the master script to extract key metrics (requests / second, mean time per request, 50th/90th percentiles, failure rate, etc). Finally, all metrics are aggregated into a unified report to facilitate comparison across endpoints and configurations.

Just before finishing writing this document, the unified load testing *script* has been run and its results are here presented. For reference, the complete parsed script output can be found in Appendix C.1.

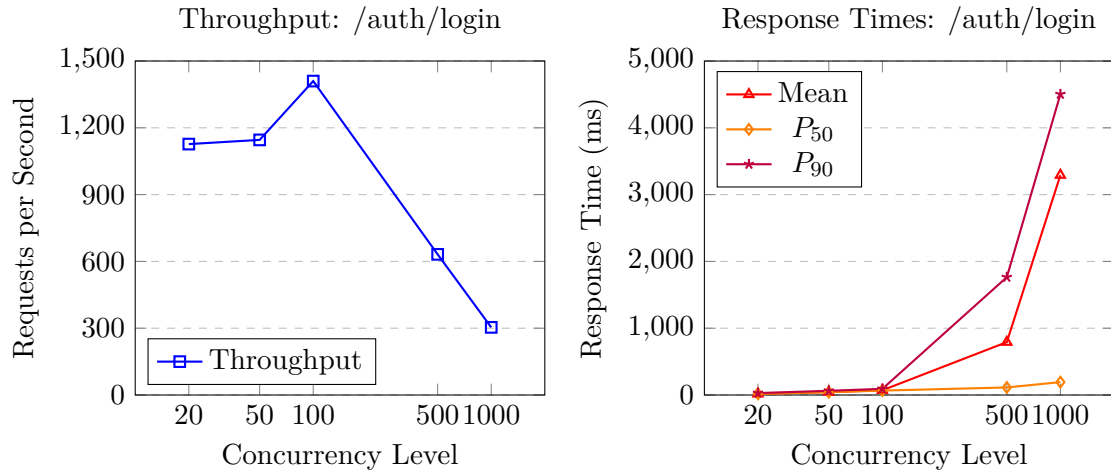
For the **authentication endpoint**, exceptional performance characteristics have been observed across all tested concurrency levels, as seen on Figure 8.2. Peak throughput of 1 409 requests per second and mean values under 100ms have been achieved at 100 concurrent users. Notably, no request failures have been observed across the entire testing spectrum, demonstrating the performance characteristics of this key bottleneck even under substantial load conditions.

As for the **student assignment submission** endpoint, remarkable performance stability with zero request failures across all concurrency scenarios has been observed. As the charts on Figure 8.3 emphasize, a maximum throughput of 1 275 requests per second and sub-100ms at 100 concurrent users really remark the queue-based architecture. It truly makes worth the implementation overhead by successfully preventing system overload and by maintaining consistent service availability in this critical, variable load endpoint.

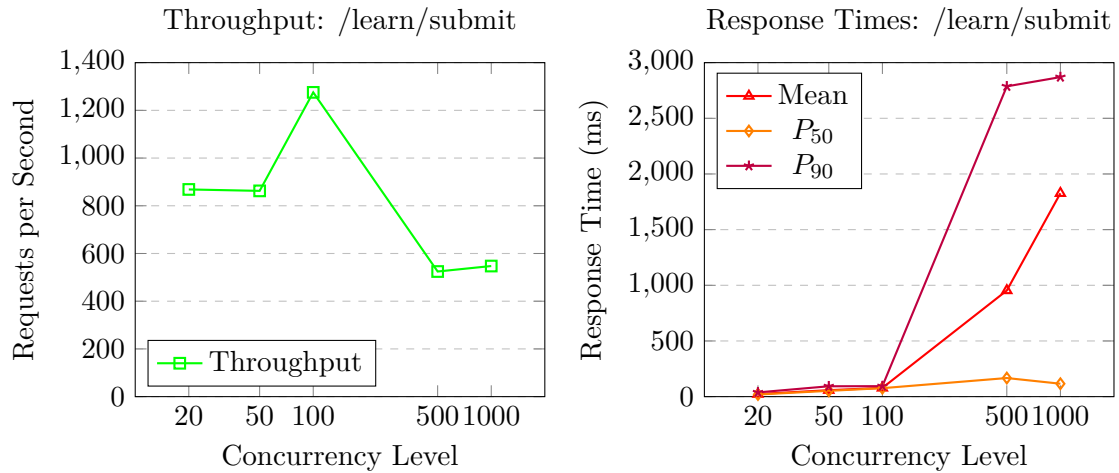
Finally, the **professor assignment creation** endpoint demonstrates *acceptable* performance under realistic conditions. As the results plotted on Figure 8.4 indicate, for typical usage scenarios (concurrency levels 20-100), the system supports the load with no request failures, indicating good stability for *realistic* professor workloads. Nevertheless, the *complexity*<sup>2</sup> of creating assignments results in higher response times compared to other endpoints. The failure thresholds observed at high concurrency

---

<sup>2</sup>Referring to performing validations, student checks, and the required database operations

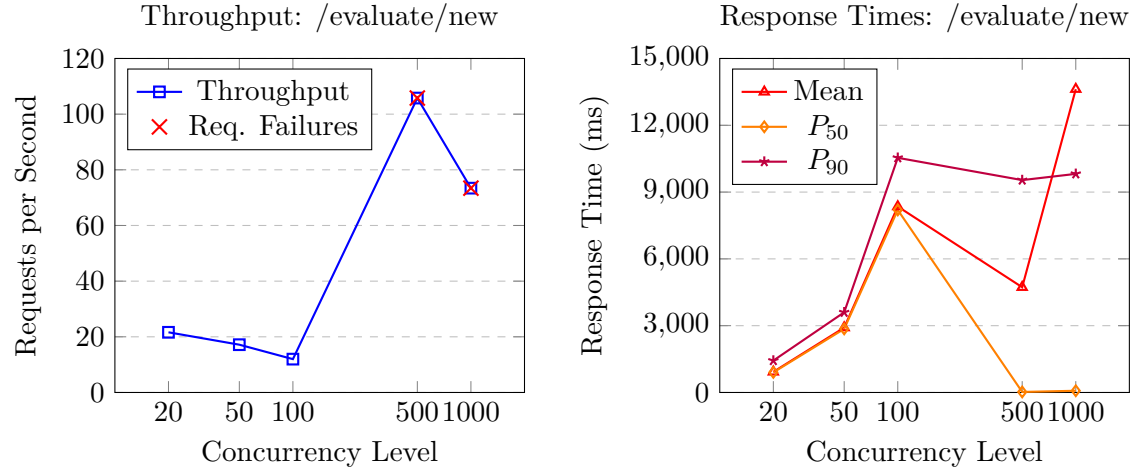


**Figure 8.2:** Performance characteristics of the authentication endpoint under varying concurrency load levels



**Figure 8.3:** Performance characteristics of the student submission endpoint under varying concurrency load levels

levels (500+ users) represent stress conditions beyond realistic scenarios, and thus are considered out of the scope.



**Figure 8.4:** Performance characteristics of the professor assignment creation endpoint under varying concurrency levels

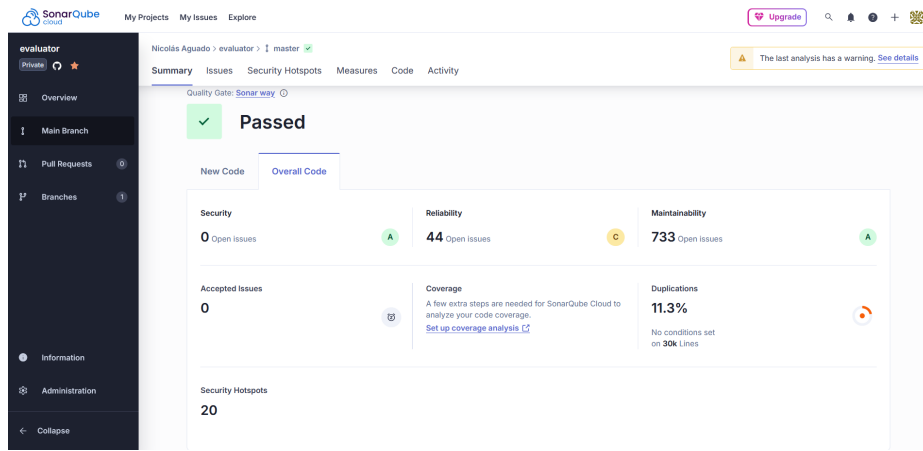
In general, it can be concluded that the Evaluator application remains stable and adequate for [production use](#). No significant failures or performance degradations were observed within the expected concurrency ranges, confirming its readiness for deployment.

### 8.3 Static Analysis (SAST)

Static Application Security Testing (SAST) provides an automated way of identifying security vulnerabilities, code smells and maintenance issues early in the development life cycle. The chosen SAST tool for the project (as explained in Section 5.1.4) is *SonarCloud*. Concretely, this tool offers a free-tiered version and a more customizable paid version. For this specific context and taking into account the project [requirements](#), the free tier has been deemed well-versed enough.

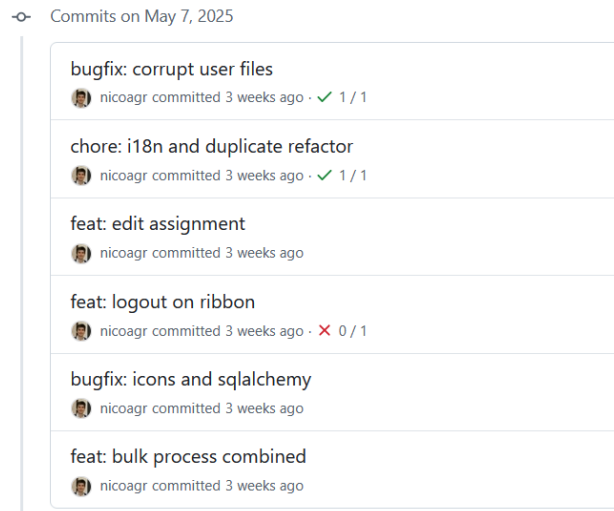
*SonarCloud* provides several metrics that represent several quality features. The comprehensive list of specific measurements can be grouped up in five categories: Security, Reliability, Maintainability, Duplications and Coverage. After comparing all of these indicators, a global grouped up *score* (called *Quality Gate*) is provided. In Figure 8.5 the state of the project as of finishing the development iterations on the *SonarCloud* panel can be observed.

This SAST scan has been configured to integrate with the [selected git hosting](#) service. The implementation works in a way such that the scan is triggered on every push to the repository, ensuring that each code change is validated against these metrics. Immediate feedback is thus provided to developers, allowing defects (and bad software



**Figure 8.5:** *SonarCloud* quality summary dashboard for the project at the end of the development iterations

quality practices) to be remedied before they proliferate. This integration also provides pass/fail indicators directly in pull requests and commit statuses (see Figure 8.6), providing a *quick glance* for the security and maintainability of the project.



**Figure 8.6:** *SonarCloud* quality indicator (for each push) in the GitHub commit timeline

## 8.4 End User *Beta* Testing

The evaluator application has been developed to provide an automated mechanism for grading student assessments by leveraging large language models. Because of its usefulness, after wrapping up the development iterations and before finishing writing this report, the system has been deployed for early *beta* testing on some specific users.

More concretely, the system is already being tested in the Faculty of Informatics of the *University of the Basque Country* <sup>3</sup>.

The system has been deployed in a machine (that meets the minimum [system requirements](#)) rented to the *Hetzner GmbH* virtual server provider and this beta testing experience has been proven useful to correct and refine a handful of *sharp edges*.

Mostly all of the encountered bugs and miscellaneous issues were related to the deployment phase. Thanks to this final application testing, the following improvements were identified and fixed.

- The *deployment guide* (referenced in Appendix [A.2](#)) was updated to better explain the GitHub configuration steps. When configuring its *OAuth*, there are some non-trivial configurations that need to take place when creating the *connector* <sup>4</sup> for the system to correctly fetch repository contents.
- Some bugs were found and resolved when testing the *local storage* mode of the system. Depending on the machine configuration and the UNIX users' permission, there are some non trivial steps that need to be taken (now automated) for storing submissions correctly within the corresponding file system.
- The *Docker* deployment guide and some internal tools were extended to support the system coexisting alongside other containers (for example, inside a machine that is used for more than one purpose). In this new *collocation* mode, the HTTP requests go through two reverse proxies (the one orchestrating the whole system, and the one belonging to *Evaluator*). The corresponding *Compose* files were updated to make sure that there were not any conflicts between other packages and/or networks and the helper scripts were changed to support both deployment modes.
- Some system functionalities, interfaces and menus were *redesigned* <sup>5</sup> (specially those in the Administrator panel) to be more *user friendly* and several strings were changed (with its translation being reviewed) to better explain the motive for each occasion.

## 8.5 Metrics and Requirement Verifications

This section aims to collect all of the mentioned characteristics of the system and compare them onto the initially defined [non-functional requirements](#), performing the relevant *due diligence* on the project scope.

---

<sup>3</sup>[www.ehu.eus/if](http://www.ehu.eus/if)

<sup>4</sup>Or, as GitHub calls them, *applications*

<sup>5</sup>As in, not a major functionality overhaul but rather a graphical design change



## Performance

### 1. Form Submission Times

- **Metric:** Submission form response times under increased user load.
- **Objective:** Processing time of  $< 1s$  when submitting a *text* assignment.
- **Result:** As seen on the data provided on Section 8.2 and in Appendix C.1, even when under unrealistic peak scenarios (1000 concurrent requests) the submission processing times remain under 200 ms, well below the defined 1s threshold.

## Security

### 1. Logging

- **Metric:** Audit log completeness. Critical actions describe logins, submissions, grading, config changes, etc.
- **Objective:** 100% of critical actions are logged with user, timestamp, action details and cost.
- **Result:** As described on Section 6.5, a complete logging framework has been developed. The system logs all the required actions within all the critical sections and with exhaustive verbosity, deeming this metric as a *pass*.

### 2. Tenant Isolation

- **Metric:** Course and student data isolation.
- **Objective:** No data leakage between tenants.
- **Result:** After completing the development iterations, a thorough review of each relevant method was performed to check for data leakage. Also, part of the defined unit tests (referenced in Section 8.1) cover this concrete aspect. Combining both reviews results in deeming this metric as a *pass*.

## Reliability

### 1. LLM Calling Robustness

- **Metric:** LLM provider fail-over success rate.
- **Objective:** 100% of failed LLM requests are retried with more than one alternative providers.
- **Result:** As described on Section 6.1 and with the code in Appendix B.1, a reliable LLM selection algorithm has been developed, taking into account per-course costs and API Key availabilities.

### Maintainability

#### 1. Quality Scanning

- **Metric:** Code quality (linting, test coverage), using automated code analysis tools (SCA) and security analysis tools (SAST).
- **Objective:** 0 critical linting and security errors; Quality gate passing
- **Result:** As seen on Section 8.3, the application development cycle is integrated with a static code analysis and its quality gate is passing at the end of the development iterations (see Figure 8.5).

### Non-Measurable Requirements

1. **Dockerization:** The application has indeed been containerized with *Docker* and *Docker Compose*, as seen on Section 4.4.
2. **Documentation:** The project does include some technical documentation. This guides are (partially) reflected on Appendix A and technical diagrams are scattered all around the technical explanations in this report (Chapters 4, 5 and 6).
3. **Unit Testing:** As per Section 8.1, the application contains the necessary unit tests for ensuring system maintainability.
4. **Load Testing:** As reflected in Section 8.2, the application has indeed undergone a load testing process and optimized subsequently.

# Monitoring and Control

This chapter documents significant deviations and changes in planning, as well as delays and modifications to system decisions relative to the [project plan](#).

## 9.1 Change Control

This particular project, with the defined [life cycle](#), benefits from a certain *ambiguity* in terms of the general scope planning. The specific features of each development iteration are (deliberately) left vague, so that with the high implication of the client the system evolves along the project needs. As one of the planned risks reflects (risk [R4](#)), the specific feature scope of the project relies a lot on the approach the client gives to this project. After finishing all the development iterations and verifying the high level of communication that has been achieved with the client, this specific risk can be considered as *mitigated*. Meetings were successfully held every  $\sim 2.5$  weeks and asynchronous communication was constant via e-mail.

Nevertheless, there was one specific change in scope at the early stages of the project that significantly changed the approach for the project. Throughout development processes, in general, there usually are functional requirements that must be taken into account at the very early stages of the application. If not, refactoring an existing model for a specific breaking change can transform into a very tedious and time consuming task. So, in *Evaluator*, at the *in-person* meetings phase, a *multi-course* application that could integrate with the *Moodle* <sup>1</sup> learning platform was envisioned.

After the second milestone, it was decided that this was a very ambitious goal and that it would be discarded in favor of creating a reliable and complete web application (as aligned with the [project objectives](#)). This meant that some *refactoring* <sup>2</sup> was done

---

<sup>1</sup>[www.moodle.org](http://www.moodle.org)

<sup>2</sup>In this specific case, more of a code simplification

to adapt the application to the *not* integration of *Moodle* and to just having a web only evaluation.

This apparently big in scope change in scope was taken as an opportunity and instead some other extension points were identified to leverage the task at hand. As a result, the extensible (submission) [type system](#) was created and the foundations for a possible future *plugin* system were laid. As a counter measure for preventing these type of changes in the future, a list of *extra non-required* features was created for the ideas that implied too big of a (technical) change in the application. These ideas were meant to be consider for future work or for spare time in the development iterations. Most of these aspects remain for [future work](#), but an example of one that did make it to the production system is the [PWA Support](#) for installing the application in mobile devices.

Furthermore, some time had to be reserved for an unexpected task. At the end of the development iterations, while writing the thesis for the project (this document), the opportunity to deploy the application in a real world *beta* testing environment arose. Because of the nature of the project's life cycle, most of the allotted time (more concretely, the risk management *extra* time) was already consumed or allocated to other tasks. Regardless, the benefits of this offering outweighed the cons and the deployment [was indeed performed](#). This opportunity was proven worth of the extra effort (as reflected on Section 8.4) for refining the application and for squashing some of the remaining bugs. As a consequence of this extra time spent, a delay in the events development occurred and the final milestone was instead completed a week and a half later.

All in all, the scope management throughout the project can be considered to have been efficiently and correctly managed. Coming from a good flexible foundation with the [life cycle](#), the more unexpected changes have been successfully *steered* <sup>3</sup> in order to meet the project [objectives](#) within the allotted time and deadlines.

## 9.2 Time Tracking

In this section a comparison between the planned time allocations and the actual devoted time for each task has been done, in accordance with the schedule defined in Section 2.4.1 (and in contrast to Table 2.1). This assessment has been conducted to highlight the differences between the forecasted deadlines and the actual effort invested in the whole project (not only in the development phases) of the *Evaluator* application. The preciseness in the reflected data corresponds to the granular management performed via *GitHub projects* (referenced in Section 2.5.2). A snippet of the third milestone, with a general view of all the *issues* <sup>4</sup> (taken after the end of the iteration) can be found on Figure 9.1. In the snippet on Figure 9.2 a concrete task from milestone 3 with all its detail and metadata can be examined.

---

<sup>3</sup>Redirected or taken advantage of for the project

<sup>4</sup>As in, GitHub *issues*, referring to concrete project tasks

## 9.2. Time Tracking

**Figure 9.1: Top Level Overview of some of the concrete Issues for Milestone 3 inside the *GitHub Projects* tracking application.**

Title	Estimate Hours	Status	Start date	Actual Hours	End date
1 Feedback and Planning #77	3	Done	Feb 26, 2025	3	Feb 26, 2025
2 Bug fixes and refactoring #80	0.5	Done	Feb 26, 2025	0.5	Feb 26, 2025
3 Email Verbosity Configuring #78	1.5	Done	Feb 27, 2025	1.25	Feb 27, 2025
4 Integrate other LLM providers #52	2	Done	Feb 27, 2025	8	Mar 3, 2025
5 API Key providing #79	3	Done	Feb 27, 2025	4.75	Mar 3, 2025
6 Image Submission Type #37	1	Done	Feb 28, 2025	3.25	Mar 1, 2025
7 Error control for grader #58	1.5	Done	Feb 17, 2025	5.5	Mar 3, 2025
8 Web Security #50	2	Done	Mar 4, 2025	2	Mar 4, 2025
9 Storage Control #82	2	Done	Mar 4, 2025	2	Mar 4, 2025
10 I18n #49	3.5	Done	Mar 5, 2025	7	Mar 6, 2025
11 Generic Evaluators #81	3	Done	Mar 7, 2025	2.5	Mar 7, 2025
12 Grades Export to Excel #61	1	Done	Mar 7, 2025	0.5	Mar 7, 2025

**Figure 9.1:** Top Level Overview of some of the concrete Issues for Milestone 3 inside the *GitHub Projects* tracking application.

**Figure 9.2: Granular Detail of a concrete Issue for Milestone 3 (with all its included metadata) on the *GitHub Projects* tracking application.**

Field	Value
Issue Title	API Key providing #79
Status	Closed
Assignees	nicoagr
Labels	No labels
Projects	Evaluator Work Tracking
Estimate Hours	2
Actual Hours	4.75
Start date	Feb 27, 2025
End date	Mar 3, 2025
Milestone	v3-Refactoring and Correctness
Milestone Status	Closed on Mar 14, 100% complete

**Figure 9.2:** Granular Detail of a concrete Issue for Milestone 3 (with all its included metadata) on the *GitHub Projects* tracking application.

A side-by-side comparison of the allotted time for each task defined in the [work breakdown structure](#) can be found on Table 9.1.

This comparison reveals some notable variations across different task categories, with the total project requiring **326 hours** versus the initially planned 300 hours, representing a manageable  $\sim$  one week overrun. The most significant deviation occurred in the *Documentation* phase, which exceeded its planned allocation primarily because of the thesis writing process. On the other hand, the *Planning & Management* phase was over estimated, remarking the overall good efficiency and the positive communication processes with the client. Within the *Development* phase, there have been noticeable variations between the iterations, but as the overall sum reflects, the scope flexibility

Task	Planned	Real
<b>Planning &amp; Management</b> (46,5h   42h)		
General Planning	3 hours	2 hours
Sprint Planning	3.5 hours	3 hours
Meetings / Comms	20 hours	17 hours
Monitoring & Control	10 hours	10 hours
Risk Management	10 hours	10 hours
<b>Analysis</b> (13h   14h)		
Requirement Analysis	6 hours	5 hours
Greenhouse Research	7 hours	9 hours
<b>Documentation</b> (65,5h   86h)		
Thesis	55.5 hours	71 hours
Presentation Design	7 hours	10 hours
Defense	0.5 hours	1 hours
Technical Doc.	2.5 hours	4 hours
<b>Development</b> (175h   184h)		
First Iteration	40 hours	23 hours
Second Iteration	40 hours	34 hours
Third Iteration	40 hours	53 hours
Fourth Iteration	40 hours	39 hours
Final Testing	15 hours	18 hours
Deployment	-	17 hours
<b>Total Sum</b>		
Evaluator Project	300 hours	326 hours

**Table 9.1:** Time Comparison of the planned vs the real dedication for each task of the [WBS](#)

allowed for a more *foreseeable* <sup>5</sup> time management. Additionally, for this phase, the unplanned *Deployment* task emerged (as already explained in Section 9.1), consuming that extra week that was not accounted for in the tentative schedule.

Furthermore, looking back to the initial planning performed on Section 2.4.2, a comparison of the milestone completion dates <sup>6</sup> for the project can be found on Table 9.2.

This completion timeline demonstrates the strong momentum that the project maintained throughout all its execution. It can be observed that between the third and fourth milestones more than the scheduled  $\sim$  two and a half weeks passed. This corresponds to the unexpected deployment and beta testing phase, while also performing

<sup>5</sup>Referring to the adjustment of specific functionalities within the scope to match the planned dedications

<sup>6</sup>For reference, all dates correspond to the year 2025

Milestone	Planned Date	Real Completion
0 - <i>Planning &amp; Initial Research</i>	Late Jan.	Feb. 3
1 - <i>Bare bones Application</i>	Mid-Late Feb.	Feb. 12
2 - <i>Functional Base</i>	Early Mar.	Feb. 25
3 - <i>Code Quality</i>	Mid-Late Mar.	Mar 14.
4 - <i>Production Ready</i>	Early Apr.	Apr. 7
5 - <i>Memoir and Defense</i>	Early-Mid Jun.	18 Jun.

**Table 9.2:** Date Comparison of the planned vs the real ending for each [milestone](#)

the necessary checks for verifying the stability of the application. Nevertheless, all milestones were achieved **on time**, with efficient [scope and change](#) management.

### 9.3 Risks and Mitigations

Throughout the project’s execution, none of the risks identified during the [planning phase](#) materialized.

1. Regarding risk [R1](#) and following the project [objectives](#), the implementation for [combining all LLM providers](#) for fault-tolerant grading was deemed of enough quality in order to fully rely on it for the main key point of the application. Some time to time monitoring was performed <sup>7</sup> and overall the providers achieve good uptime and the whole system can almost always provide a grading result.
2. On the topic of risk [R2](#), as already explained in Sections [9.1](#) and [9.2](#), the scope was correctly managed and the application was deemed of enough quality (and specially with enough functional features) to be shipped to real users. Regardless, this application can be extended in many ways and additional features can improve the system in [future work](#).
3. Regarding risk [R3](#), all personal issues were appropriately discussed during communications between all the interested parties and no significant modifications to the scope or schedule were deemed necessary.
4. As already discussed in Section [9.1](#) and [9.2](#), risks [R4](#) and [R5](#) have been proven mitigated.

<sup>7</sup>Not a formal investigation, but rather an occasional check





## Conclusions

After completing the project, there are some final conclusions that can be taken into account.

### 10.1 Future Work

In light of all the possible expansion points presented throughout this report, a consolidated set of future work recommendations is now exposed.

- In relation with the [extensible type](#) and the [extensible LLM](#) systems, a possible **plugin system** could be implemented for the application. There are pieces of functionality that can be encapsulated, grouped up together into single *modules* that could be installed or managed separately. The application already applies the necessary software engineering patterns to allow for using these two points of extensions, so through some possible *plugins*, different LLM providers could be implemented and/or custom logic for grading special typed assignments could become available.
- The key component of the system is the automated LLM evaluation. But, the **quality** of that LLM evaluation is unknown. Of course, it will vary depending on which model does indeed grade the submission, but a more formal and complete **analysis** is needed. In order to assess the educational impact of the *Evaluator* system, it is left as future work to compare and analyze the quality of the LLM results and its usefulness in the educator submission grading process.
- In relation to the extensible *plugin* system, another proposed functionality for future work refers to specific **ZIP file repository submissions**. Instead of grading coding assignments using only GitHub repositories, code could be delivered through compressed archives and evaluated using the already implemented file

## 10. CONCLUSIONS

---

scanning methods. This could allow for exiting *git* hosting provider dependencies and for providing feedback in environments where the *git* versioning system is not available.

- On another note, as mentioned on section 9.3, a top-level, not exhaustive monitoring has been performed on the LLM providers' reliability and uptime. It would be useful to actually perform a formal analysis to check on the exact numbers for these key pieces of the *Evaluator* system. If the providers are not deemed reliable enough (and the available open-source models provide enough quality), some local-hosting alternatives could be explored (acknowledging that merely self-hosting an *state of the art* LLM model is an entire project by itself).
- It would be out of the scope regarding the *plugin* system, but the core application can also be extended to support **submissions in groups of students**. Project based learning often hinges on collaborative, team-based work, so, when submitting an assignment, the possibility of doing so in *groups* and allowing all members to view and iterate on the same feedback results specially useful.
- In terms of being a more complete learning management system, two specific time related features could be implemented: the possibility of allowing **late submissions** and the choice for **scheduling assignments**. In the first case the system would allow for students to keep submitting their assignments even when the deadline expires (under a penalty or a note or some sort) and in the second case the system would allow the instructors for scheduling their assignments to be *opened* <sup>1</sup> in a later date in the future. These both functional features result in a more complete learning experience and provide *quality of life* features for both students and educators.

### 10.2 Ethical Considerations

The integration of large language models (LLMs) into educational assessments, as implemented in the *Evaluator* application, necessitates careful consideration of ethical implications. The main concerning aspect relates to the entire replacement of the human grading process <sup>2</sup>, with the derived educational grading quality and tutoring feedback implications.

The decision to have the LLM provide a binary PASS/FAIL grade, rather than an overall numerical score, is a deliberate design choice to mitigate potential biases or misinterpretations inherent in current AI capabilities [10]. By limiting the LLM's output to a preliminary assessment, the system is positioned as a *triage* tool. Its primary function is to offer an initial screening of student submissions against professor-defined rubrics, thereby alleviating a significant portion of the lecturer's workload by identifying

---

<sup>1</sup>As in, opened for student submissions

<sup>2</sup>Whatever the reasons or ulterior motives for this choice may be, regardless of them

submissions that clearly meet or do not meet basic criteria, rather than attempting to replace the educator’s comprehensive judgment.

Furthermore, to address the limitations and potential risks of uniquely relying on an automated evaluation, a submission retry mechanism has been implemented. This system ensures that submissions are not only judged by a single LLM assessment and allows for human intervention or alternative evaluation pathways, particularly for complex or borderline cases.

This design highlights the intention for the system to augment, not supplant, the educator’s role, ensuring that the lecturer remains the ultimate judge of student performance and can focus their expertise on providing in-depth feedback where it is most needed.

## 10.3 Project Retrospective

One of the key factors that has played an important role for the realization of this project has been the initial planning. From the beginning, a more flexible approach was [designed](#) and some rough milestones were marked. This organization had a clear differentiation with the more traditional, stricter planning and left most of the concrete scope management to be defined throughout the project development. Because one of the *stricter* <sup>3</sup> limits was the budgeted time, this approach has proven highly effective to produce an end result (of considerable [quality](#)) and its scope flexibility has provided valuable intermediate rearrangements.

The client implication is also a point to be remarked on. As the risk [R4](#) reflects, the project could not have had such a high degree of success without the consistent, timely devotion of the client throughout the entire [life cycle](#). All in all, this has led to an **on time** project development and a positive adjustment to all of the defined milestones (Section [9.2](#)).

The project has concluded at an overall success, meeting both the main two primary [objectives](#) and all of the secondary ones. A **reliable feedback platform** for students has been built, providing timely feedback and a personalized evaluation that covers several aspects of submission quality (Sections [6.1](#), [6.2](#) and [8.1](#)). The platform also provides a **supporting method for educators’ assesment process**, allowing for the already defined *triage* to occur, enabling for a more specialized human feedback and an overall improvement in project based learning methodologies.

The *Evaluator* application has been deemed economically viable (Section [7.4](#)), its modularity allows it to be extended by other developers (Section [6.4](#)) and to be adapted to a rapid changing large language model environment (Section [6.1](#)). In general terms, and because of all the testing that has been made (Sections [8.1](#), [8.2](#), [8.3](#) and [8.4](#)), the application can be confidently deemed **production ready**.

---

<sup>3</sup>As in, the one that mattered most and had to take special care of

## 10. CONCLUSIONS

---

Ultimately, the planning has gone well and the greenhouse application has been positively developed (and deployed!), meeting all of the objectives and achieving an overall **success** for the project.

# Documentation and Manuals

## A.1 Development Setup and Useful Commands

For developing and executing the project, a python interpreter and a **UNIX-like system** are required. Also, a **REDIS** server is required and the ability to spawn redis “workers”.

```
git clone [repository-url]
cd evaluator
```

Here, modify the `.env.example` file with the correct values for the environment variables and rename it to `.env`.

```
python3 -m venv venv
```

( If the previous `venv` command gives an error, skip the following comand. If not, execute it )

```
source venv/bin/activate
```

```
pip install -r requirements.txt
```

```
python3 run.py
```

Now, the development FLASK server will be running. Usually this server supports *hot reloading* for fast development.

To start the REDIS workers, run the following command, from a terminal already *activated* with the `venv`.

```
rq worker evaluator-working-queue --with-scheduler
```

(If you are running multiple workers, you can only use the `--with-scheduler` option in one of them.)

### A.1.0.1 HTTP Deployment (Not Recommended for Production)

If you need to deploy without HTTPS (for testing purposes only):

1. Set `FLASK_TALISMAN_FORCE_HTTPS=false` in your environment

Note: This is not secure for production environments as it disables important security features.

### A.1.1 To initialize a new language

```
pybabel extract -F babel.cfg -k _l -o messages.pot .
pybabel init -i messages.pot -d src/translations -l <language>
```

And update the “`enabled_languages`” dict in `run.py` to include the new language.

### A.1.2 To update the translations

```
pybabel extract -F babel.cfg -k _l -o messages.pot .
pybabel update -i messages.pot -d src/translations
```

### A.1.3 To compile the translations

```
pybabel compile -d src/translations
```

## A.2 Deployment Manual

This document outlines the deployment process for the Evaluator application, a web application that automatically grades student assignments using LLMs.

### A.2.1 Environment Setup

Before deploying, configure the environment variables in the `infra/.env` file:

```
# Required settings
SECRET_KEY=your-secure-secret-key
STORAGE_HARDB_LIMT=10
ADMIN_EMAIL=your-admin-email
ADMIN_PASSWORD=secure-admin-password
DOMAIN=evaluator.domain.com

# API Keys (Required for rate limited, free tiered applications)
GROQ_API_KEY=your-groq-api-key
GEMINI_API_KEY=your-gemini-api-key

# Email Configuration
SMTP_HOST=your-smtp-host
SMTP_EMAIL_USERNAME=usually-sender-email
SMTP_EMAIL_PASSWORD=usually-sender-password
or
MAILJET_API_KEY=get-your-own-in-mailjet-com
MAILJET_API_SECRET=get-your-own-in-mailjet-com
MAILJET_SENDER_MAIL=sender-configured-in@mailjet.com

# GitHub Integration (if needed)
GITHUB_CLIENT_ID=your-github-client-id
GITHUB_CLIENT_SECRET=your-github-client-secret
GITHUB_PRIVATE_KEY_PATH=./private-github-key.pem
GITHUB_PUBLIC_APP_LINK=https://github.com/apps/your-app-name

# Important GitHub App Requirements:
# - The GitHub application MUST be set to "public" (not private)
# - The application MUST have "read access" to the
#   "contents" permission in the repository tab
# If these conditions are not met, GitHub authentication will not
#   work, even if the key file is set correctly.

# S3-Like Storage (if using -if not, will store locally)
S3_BUCKET_NAME=your-bucket-name
```

```
S3_ACCESS_KEY=your-s3-access-key
S3_SECRET_KEY=your-s3-secret-key
S3_ENDPOINT_URL=your-s3-endpoint-url

# Security Configuration
FLASK_TALISMAN_FORCE_HTTPS=true # Set to false for development
DISABLE_CSRF=false              # Set to true only for development
ENVIRONMENT=production          # Options: production, development
CERTBOT_FIRST_RUN=true          # Set to true for initial cert acquisition
```

### A.2.2 Architecture Overview

The application consists of several Docker containers:

1. **evaluator-app**: Flask application serving the main web interface
2. **evaluator-webserver**: Lighttpd web server as a reverse proxy
3. **evaluator-queue-single**: Main worker. Manages scheduling
4. **evaluator-queue-worker**: Extra background workers
5. **evaluator-redis**: Redis for session caching and queue
6. **evaluator-db**: PostgreSQL relational database

### A.2.3 Deployment Options

The Evaluator application supports two deployment modes:

#### A.2.3.1 Standalone Deployment

The standalone deployment is designed to run as an independent application, with its own web server listening on ports 80 and 443. This is the default deployment mode.

##### Key characteristics

- Manages its own SSL certificates via Certbot
- Listens on standard HTTP/HTTPS ports (80/443)
- Fully independent operations
- Uses `docker-compose.yml` for deployment
- Uses a shared user data volume

#### A.2.3.2 Colocation Deployment

The colocation deployment is designed to run alongside other applications on the same server, without requiring dedicated access to ports 80 and 443. This is useful when you already have other web services running on the same machine.

##### Key characteristics



- Listens on a custom port (21322 by default)
- Doesn't handle SSL termination (should be managed by your main web server)
- Designed to work behind a reverse proxy
- Uses `docker-compose-colocation.yml` for deployment
- Uses a shared user data volume

## A.2.4 Deployment Steps

### A.2.4.1 Prerequisites

- Docker and Docker Compose installed
- Domain name pointed to your server
- For standalone mode: Ports 80 and 443 open on your firewall
- For colocation mode: Port 21322 accessible
- Root access on the server

### A.2.4.2 Setup

Clone the repository and navigate to the project directory:

```
git clone [repository-url]
cd evaluator
```

### A.2.4.3 Configuration

1. Copy the example environment file and modify it:

```
cp .env.example infra/.env
nano infra/.env
```

2. Ensure all required environment variables are set

### A.2.4.4 Launching the Application

The application includes convenient startup scripts that handle directory creation, permissions, and container startup:

#### For Standalone Deployments:

```
cd infra
sudo ./docker-standalone.sh
```

This script:

- Creates the `/opt/evaluatornicoagrdata/userdata` directory if it doesn't exist

## A. DOCUMENTATION AND MANUALS

---

- Sets proper permissions (user ID 1000)
- Starts the application with `docker compose -f docker-compose.yml up -d --force-recreate`

### For Colocation Deployments:

```
cd infra
sudo ./docker-colocation.sh
```

This script:

- Creates the `/opt/evaluatornicoagrdata/userdata` directory if it doesn't exist
- Sets proper permissions (user ID 1000)
- Starts the application with `docker compose -f docker-compose-colocation.yml up -d --force-recreate`

Ensure that your main web server is configured to proxy requests to port 21322 for the Evaluator application domain or path.

**Manual Startup (Alternative):** If you prefer not to use the startup scripts, you can manually create the directories and start the containers:

For standalone deployment:

```
# Create directory and set permissions
sudo mkdir -p /opt/evaluatornicoagrdata/userdata
sudo chown -R 1000:1000 /opt/evaluatornicoagrdata/userdata

# Start containers
cd infra
docker compose -f docker-compose.yml up -d
```

For colocation deployment:

```
# Create directory and set permissions
sudo mkdir -p /opt/evaluatornicoagrdata/userdata
sudo chown -R 1000:1000 /opt/evaluatornicoagrdata/userdata

# Start containers
cd infra
docker compose -f docker-compose-colocation.yml up -d
```

#### A.2.4.5 HTTPS Configuration for Standalone Deployment

When deploying in standalone mode, using HTTPS is strongly recommended. This application uses secure cookies by default, which will only work over HTTPS connections.

The application uses Certbot to automatically obtain and renew TLS certificates from the *Buypass* provider ACME server. To configure HTTPS:

1. Set the following variables in your `.env` file:

```
DOMAIN=your-domain.com
ADMIN_EMAIL=your-email@example.com
FLASK_TALISMAN_FORCE_HTTPS=true
CERTBOT_FIRST_RUN=true
```

2. For the initial deployment, set `CERTBOT_FIRST_RUN=true` to trigger the certificate acquisition process.
3. After the initial deployment, you can set `CERTBOT_FIRST_RUN=false`.

If you encounter the error: “Cookie ‘session’ has been rejected because a non-HTTPS cookie can’t be set as ‘secure’”, it means you’re trying to access the application over HTTP while secure cookies are enabled. Solutions: Set `FLASK_TALISMAN_FORCE_HTTPS=false` temporarily during development, ensure you’re accessing the application via HTTPS and/or check that your SSL certificates are properly configured

#### A.2.4.6 Certificate Management (Standalone Only)

The standalone deployment includes scripts for managing SSL certificates:

- **manage-certs.sh**: Handles certificate acquisition and renewal
- **check-ssl-certs.sh**: Configures Lighttpd based on certificate availability

To manually renew certificates (by default they are renewed automatically every two months):

```
docker exec evaluator-webserver /usr/local/bin/manage-certs.sh renew
```

#### A.2.4.7 Scaling Workers

To scale the number of queue workers:

For standalone deployment:

## A. DOCUMENTATION AND MANUALS

---

```
docker compose -f docker-compose.yml up -d --scale evaluator-queue-worker=3
```

For colocation deployment:

```
docker compose -f (...) - colocation.yml up -d --scale evaluator-queue-worker=3
```

### A.2.4.8 Monitoring and Logs

View application logs:

For standalone deployment:

```
docker compose -f docker-compose.yml logs -f evaluator-app
```

For colocation deployment:

```
docker compose -f docker-compose-colocation.yml logs -f evaluator-app
```

### A.2.5 Security Considerations

1. Always use a strong SECRET\_KEY
2. In production, ensure FLASK\_TALISMAN\_FORCE\_HTTPS=true
3. Set DISABLE\_CSRF=false in production
4. Replace default admin credentials
5. Store sensitive API keys securely

### A.2.6 Backup and Recovery

Database data is stored in `infra/dbdata/`. To backup the database:

```
docker exec evaluator-db pg_dump -U evaluatoruser evaluator > backup.sql
```

#### A.2.6.1 User Data Storage

In both deployment modes, user uploads are stored in the `/opt/evaluatornicoagrdata/userdata` directory, which is mounted as a Docker volume.

To backup the user data:

```
sudo tar -cvzf userdata-backup.tar.gz /opt/evaluatornicoagrdata/userdata
```

## Code Extracts

In this appendix all code extracts referenced in the thesis are gathered, organized by sections to facilitate quick navigation and review.

### B.1 LLM Persistent Evaluation Function

Attached along this lines is the hyperlink to the production code for the LLM `try_to_evaluate_persistent` function.

<https://pastebin.com/SUfZq9ai>

As to provide a bit more context, the `evaluator.evaluate` function (and mostly, the subsequent functions that it calls) routes the data and rubric to the corresponding provider API of the passed along model. It also is in charge of building the prompt, keeping track of costs and emails and updating the database with the correct or incorrect results.

The `get_categorized_models` function returns a list of all the models that are available to evaluate that submission type (because there are some models that may not be able to - for example, some models cannot process images).

### B.2 Decorators for Authentication

In the hyperlink provided below the complete Python *Flask* implementation for the authentication decorators is available.

<https://pastebin.com/VnTWg7be>

These mentioned decorators can be applied to every route (every function inside a blueprint) and they allow for easily checking access to the application and the corresponding role permissions.

### B.3 Progressive Web App

Attached along this lines is the URL to the complete implemented code for showing the native *PWA* installation prompt in all mobile operating systems and in the desktop browsers that support it.

<https://pastebin.com/qdRFLK9S>

The source code for the service worker registration that makes the *PWA* possible throughout the pages of the application is also listed in the web address immediately below.

<https://pastebin.com/F2XcvYiK>

### B.4 Fixture Initialization File

The `conftest.py` file (attached via hyperlink below) refers to the automated [unit testing](#) initializations.

<https://pastebin.com/zwV1LpMX>

This file serves as a common ground for all testing files and provides useful *logged in* professors and students for testing. For operating the tests in an isolated environment, a forceful table *drop* and *recreate* is done before the whole test suite.

For the more curious readers, this file also serves as a *sneek peek* into the actual implementation of the database schema and the various *helper* methods that are defined internally.

### B.5 Standalone Docker Compose

Following these lines, a web URL to the production use *standalone* docker-compose file is provided.

<https://pastebin.com/bAgey8uu>

The effective network isolation can be verified by checking how the different networks have been setup between containers.



## Testing Results

### C.1 Load Benchmark Results

This is the output of the load testing script, having each *apache benchmark* file parsed and the results per endpoint combined.

The script was run for the production *dockerized system* in a machine with only the benchmarking script and docker running, with no other containers.

#### Load Testing Summary

=====

mar 27 may 2025 16:34:10 CEST

=====

Results saved to: results

Debug files saved to: results/debug

Total execution time: 0h 13m 42s

=====

Endpoint: /evaluate/new

Description: Professor Create Assignment

Method: POST

Concurrency Level: 20

Total Requests: 2500

Requests per second: 21.62/sec

Mean time per request: 924.93 ms

50th percentile (median): 902 ms

90th percentile: 1432 ms

Failed Requests: N/A

Failures started at percentile: N/A

## C. TESTING RESULTS

---

-----  
Endpoint: /evaluate/new  
Description: Professor Create Assignment  
Method: POST  
Concurrency Level: 50  
Total Requests: 2500  
Requests per second: 17.17/sec  
Mean time per request: 2912.55 ms  
50th percentile (median): 2855 ms  
90th percentile: 3600 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /evaluate/new  
Description: Professor Create Assignment  
Method: POST  
Concurrency Level: 100  
Total Requests: 2500  
Requests per second: 11.98/sec  
Mean time per request: 8350.31 ms  
50th percentile (median): 8197 ms  
90th percentile: 10539 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /evaluate/new  
Description: Professor Create Assignment  
Method: POST  
Concurrency Level: 500  
Total Requests: 2500  
Requests per second: 105.78/sec  
Mean time per request: 4726.93 ms  
50th percentile (median): 24 ms  
90th percentile: 9538 ms  
Failed Requests: 2223  
Failures started at percentile: 80%  
-----

Endpoint: /evaluate/new  
Description: Professor Create Assignment



Method: POST  
Concurrency Level: 1000  
Total Requests: 2500  
Requests per second: 73.41/sec  
Mean time per request: 13621.90 ms  
50th percentile (median): 77 ms  
90th percentile: 9817 ms  
Failed Requests: 2170  
Failures started at percentile: 66%

-----

Endpoint: /auth/login  
Description: Public Login Page  
Method: POST  
Concurrency Level: 20  
Total Requests: 2500  
Requests per second: 1126.88/sec  
Mean time per request: 17.75 ms  
50th percentile (median): 17 ms  
90th percentile: 29 ms  
Failed Requests: N/A  
Failures started at percentile: N/A

-----

Endpoint: /auth/login  
Description: Public Login Page  
Method: POST  
Concurrency Level: 50  
Total Requests: 2500  
Requests per second: 1145.73/sec  
Mean time per request: 43.64 ms  
50th percentile (median): 42 ms  
90th percentile: 63 ms  
Failed Requests: N/A  
Failures started at percentile: N/A

-----

Endpoint: /auth/login  
Description: Public Login Page  
Method: POST  
Concurrency Level: 100  
Total Requests: 2500  
Requests per second: 1409.89/sec

### C. TESTING RESULTS

---

Mean time per request: 70.93 ms  
50th percentile (median): 66 ms  
90th percentile: 92 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /auth/login  
Description: Public Login Page  
Method: POST  
Concurrency Level: 500  
Total Requests: 2500  
Requests per second: 631.61/sec  
Mean time per request: 791.63 ms  
50th percentile (median): 113 ms  
90th percentile: 1764 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /auth/login  
Description: Public Login Page  
Method: POST  
Concurrency Level: 1000  
Total Requests: 2500  
Requests per second: 303.75/sec  
Mean time per request: 3292.15 ms  
50th percentile (median): 193 ms  
90th percentile: 4503 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /learn/submit  
Description: Student Submit Assignment  
Method: POST  
Concurrency Level: 20  
Total Requests: 2500  
Requests per second: 868.46/sec  
Mean time per request: 23.03 ms  
50th percentile (median): 19 ms  
90th percentile: 38 ms  
Failed Requests: N/A

Failures started at percentile: N/A

-----  
Endpoint: /learn/submit  
Description: Student Submit Assignment  
Method: POST  
Concurrency Level: 50  
Total Requests: 2500  
Requests per second: 862.49/sec  
Mean time per request: 57.97 ms  
50th percentile (median): 52 ms  
90th percentile: 93 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /learn/submit  
Description: Student Submit Assignment  
Method: POST  
Concurrency Level: 100  
Total Requests: 2500  
Requests per second: 1275.40/sec  
Mean time per request: 78.41 ms  
50th percentile (median): 76 ms  
90th percentile: 94 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /learn/submit  
Description: Student Submit Assignment  
Method: POST  
Concurrency Level: 500  
Total Requests: 2500  
Requests per second: 524.44/sec  
Mean time per request: 953.39 ms  
50th percentile (median): 167 ms  
90th percentile: 2787 ms  
Failed Requests: N/A  
Failures started at percentile: N/A  
-----

Endpoint: /learn/submit

### C. TESTING RESULTS

---

Description: Student Submit Assignment

Method: POST

Concurrency Level: 1000

Total Requests: 2500

Requests per second: 547.42/sec

Mean time per request: 1826.74 ms

50th percentile (median): 116 ms

90th percentile: 2870 ms

Failed Requests: N/A

Failures started at percentile: N/A

-----

# Bibliography

- [1] Phyllis C. Blumenfeld, Elliot Soloway, Ronald W. Marx, Joseph S. Krajcik, Mark Guzdial, and Annemarie Palincsar. Motivating project-based learning: Sustaining the doing, supporting the learning. *Educational Psychologist*, 26:369–398, 1991. See page [1](#).
- [2] J.C. Paiva, J.P. Leal, and Á. Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Transactions on Computing Education*, 2022. See page [1](#).
- [3] Marcus Messer, Neil C. C. Brown, Michael Kölling, and Miaoqing Shi. Automated grading and feedback tools for programming education: A systematic review. *ACM Transactions on Computing Education*, 2024. See page [1](#).
- [4] K. Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer Science Education*, pages 83–102, 2005. See page [1](#).
- [5] Tao Wu and Maiga Chang. Application of generative artificial intelligence to assessment and curriculum design for project-based learning. In *2023 International Conference on Engineering and Emerging Technologies (ICEET)*, pages 1–6, 2023. See page [1](#).
- [6] Enkelejda Kasneci, Kathrin Sessler, Stefan Küchemann, Maria Bannert, et al. Chatgpt for good? on opportunities and challenges of large language models for education. *Learning and Individual Differences*, 103:102274, 2023. See page [2](#).
- [7] James Martin. *Rapid Application Development*. MacMillan, 1991. See page [5](#).
- [8] Peter Krogh. *The DAM Book: Digital Asset Management for Photographers*, page 207. O’Reilly Media, Inc, Cambridge, 2nd edition, 2009. See page [13](#).
- [9] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns*. Wiley Publishing, 1996. See page [39](#).
- [10] F.J. García, M.J. Casañ, Marc Alier, and J.A. Pereira. The ethics of generative artificial intelligence in education under debate. a perspective from the development of a theoretical-practical case study. *Revista Española de Pedagogía*, 83:281–293, 2025. See page [76](#).