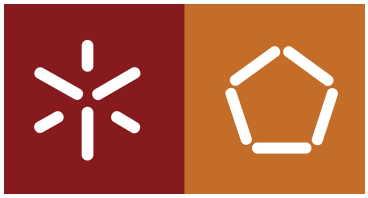
Universidade do Minho



Mestrado em Engenharia Informática

Requisitos e Arquiteturas de Software

3ª Fase - 19 de janeiro de 2025

Ana Alves
PG57505

Augusto Campos
PG57510

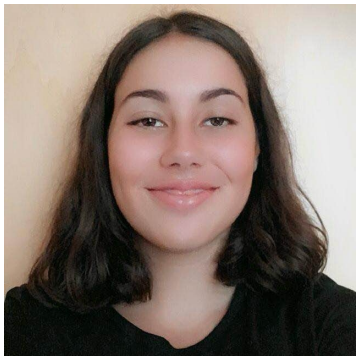
Guilherme Krull
PG55090

Iyán Riol
E12141

Tiago Silva
PG57617

Marta Gonçalves
PG55983

Francisco Lameirão
pg57542



Luna Figueiredo
PG55979

João Abreu
PG55895

Índice

1 Introduction and Goals

This document aims to cover the work carried out during the third phase of the practical assignment Picturas. It will include everything from changes made to the proposed architecture in the context of the implementation to a critical analysis of the process across the various phases. Finally, the degree of coverage of the functional and non-functional requirements defined in the first phase will also be analyzed.

2 Architectural Modifications

2.1 Absence of Payment Processing

Currently, we have not implemented any payment processing system, such as Stripe, into our application, which means that users are not able to perform financial transactions directly through the platform. The integration of a payment gateway would allow us to manage subscription payments securely and enhance the user experience, so moving forward, implementing Stripe or an equivalent payment provider would be essential to enable this more advanced feature and ensure smooth financial transactions within the platform.

2.2 Merging the Subscription and User Microservices

We have decided to merge the *Subscriptions* and *Users* microservices into a single service. This decision was primarily motivated by the fact that both services rely heavily on user-related data. By combining them, we simplify the overall architecture and reduce the complexity of maintaining two separate services that essentially interact with the same core data structures.

However, we acknowledge that separating these services might have been more appropriate in scenarios where the subscription service would need to handle additional features, such as payment related information management. A separate service would better isolate responsibilities, ensuring a cleaner and more maintainable architecture and providing scalability benefits, as the Subscription service could be independently scaled when needed, while the User service could focus solely on user-related tasks, such as authentication.

Ultimately, the current approach of merging these services was chosen to balance simplicity and functionality with the possibility of being changed in the future, especially if we decide to introduce payment processing or other features that need a more complex separation of concerns.

3 Microservices Implemented

Microservice	Description	Developer
Binarization	Converts an image to black and white by applying a threshold.	Marta Gonçalves - PG55983
Border	Adds a border or frame around an image.	João Abreu - PG55895
Brightness	Adjusts the brightness level of an image.	Augusto Campos - PG57510
Object_detection	Detects and identifies objects in an image.	Ana Alves - PG57505
Count_people	Detects and counts people in an image.	Tiago Silva - PG57617
Crop	Trims an image to remove unwanted parts.	Luna Figueiredo - PG55979
Remove_background	Isolates the main subject by removing the background.	Iyán Riol - E12141
Resize	Changes the dimensions of an image.	Guilherme Krull - PG55090
Rotate	Rotates an image by a specified angle.	Francisco Lameirão - PG57542
Watermark	Applies a watermark to an image for branding or protection.	Teaching team

Table 1: Tool Microservices Implemented

In addition to the tool microservices, other components were also developed, as listed in the following table:

Component	Description
Project	A microservice that manages all projects and orchestrates the potential tools to be applied.
User	A microservice responsible for user-related operations, such as login and user management.
Message Broker	RabbitMQ is used as a message broker to facilitate communication between microservices.
Database	MongoDB is used as the database to store application data.
Web Socket Gateway	Enables real-time communication between the application and the client through WebSockets.
API Gateway	Acts as a single entry point for all API requests, routing them to the appropriate microservices.
S3 Service	MinIO is used to provide S3-compatible object storage for image files.

Table 2: Other Components Implemented

Since we used an external service to store images, it was necessary to modify the microservice developed by the teachers regarding the loading and saving of images. This is the reason why we did not use the original docker image of the tool directly.

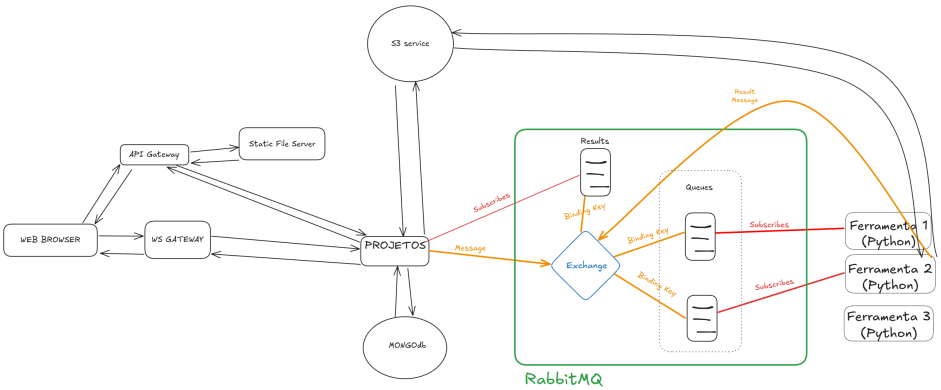


Figure 1: Simplified project overview

4 Degree of coverage of the requirements

4.1 Functional Requirements

The following are the functional requirements we consider covered by our implementation:

- **RF1:** The user can login.
- **RF2:** The user can register.
- **RF6:** The user creates a project.
- **RF7:** The user lists their projects.
- **RF8:** The user accesses the editing area of a project.
- **RF9:** The user uploads images to a project.
- **RF10:** The user removes an image from the project.
- **RF11:** The user adds an editing tool to the project.
- **RF12:** The user triggers the processing of a project.
- **RF13:** The user downloads the result of a project to their local device.
- **RF14:** The user changes the order of images in a project.
- **RF15:** The user changes the order of tools in a project.
- **RF16:** The user modifies the parameters of the tools.

- **RF25:** The user manually crops images.
- **RF26:** The user resizes images to specific dimensions.
- **RF27:** The user adds borders to images.
- **RF29:** The user adjusts the brightness of images.
- **RF31:** The user binarizes images.
- **RF32:** The user rotates images.
- **RF35:** The user removes the background of an image, keeping only the main object.
- **RF37:** The user applies an object recognition algorithm to images.
- **RF38:** The user applies a people-counting algorithm to images.

A total of 22 out of 38 functional requirements were met, resulting in a coverage rate of approximately 58%. This value is not very high, as the development team dedicated more effort and time to implementing the features requested in the MVP.

4.2 Non-functional Requirements

The following are the non-functional requirements we consider covered by our implementation:

- **RNF1:** The application must have a simple and clear interface.
- **RNF2:** The application must have clear and intuitive icons and buttons.
- **RNF3:** The application must use soft and neutral colors.
- **RNF4:** The application must be easy to use.
- **RNF5:** The project page must visually distinguish between basic and advanced tools.
- **RNF13:** The application must display a preview of the result of applying the sequence of project tools to the currently selected image.
- **RNF14:** Large or small image visualization must be assisted by a zoom utility.
- **RNF15:** Image visualization must allow navigation in all directions by dragging the mouse.

- **RNF18:** The application must be compatible with different platforms and browsers, including mobile and desktop devices.
- **RNF21:** Image editing tools must process images quickly.
- **RNF22:** The system must process up to 100 images simultaneously without noticeable performance degradation.
- **RNF23:** The application must be designed to scale horizontally in an elastic manner to support increasing users and processing volumes, maintaining performance and cost control.
- **RNF24:** The application must be available 24/7.
- **RNF25:** The application must be integrable with other platforms and third-party services.
- **RNF26:** The application must optimize the use of client computing resources (e.g., CPU, memory).
- **RNF27:** The application must support major image formats, such as JPEG, PNG, BMP, and TIFF.
- **RNF28:** The application must be easily extendable with new editing tools.
- **RNF29:** Scheduled maintenance must result in a maximum of 10 minutes of downtime per month, performed outside peak usage hours.
- **RNF32:** The system must be designed to facilitate testing.
- **RNF34:** The system ensures that only users who upload the images have access to them.

A total of 20 out of 41 non-functional requirements were met, resulting in a coverage rate of approximately 50%. This value is not very high nor very reliable since some of these requirements are hard to prove. Besides MVP considerations, the priorities of the requirements were also followed which means that some low priority requirements didn't get in the way of more important ones.

5 Critical Analysis

5.1 Phase 1: Requirements Gathering

Although we come from different groups in the first phase of the work, we all understood the importance of this phase in minimizing future complications. This phase success lies not only in the thoroughness of the documentation but also in its ability to anticipate future challenges and adapt to evolving needs. With a solid base established, the project is well-positioned to deliver a solution that meets user expectations while providing the scalability and reliability necessary for sustained growth. One thing we need to mention is the fact that, in the requirements document provided for phase 2, some non-functional requirements had limitations in their description, making it difficult to validate them post-implementation (e.g., "The application must be easy to use").

5.2 Phase 2: System Architecture

The System Architecture phase came with its challenges, as, initially, few group members had the knowledge or experience to easily arrive at an architectural solution that met the presented non-functional requirements. This phase involved extensive research and frequent meetings to ensure everyone was aligned and working towards the same goal. We decided to adopt a microservices based architecture, as it was the architectural pattern that best suited the problem.

However, the previously mentioned lack of experience resulted in some necessary components not being included in the plan we proposed for phase 2. Since a standardized architecture was provided to all groups for phase 3, the omission of elements like a message broker and the need for WebSockets for certain interactions did not negatively impact the subsequent phase.

5.3 Phase 3: System Implementation

In this phase, the main goal was to complete the MVP. As previously mentioned, the remaining functionalities played a secondary role in adhering to the proposed architecture.

The team opted for technologies they felt comfortable with, including Python for the tools, Node.js for the backend, and Vue.js for the frontend. RabbitMQ was chosen as the message broker, MinIO for image storage, and MongoDB for the database.

The integration of the various tools went smoothly, especially after the teaching team provided their tool. However, we faced several challenges, including issues with WebSocket communication. Overall, we consider this phase to have been less successful than the previous one, although it was an interesting experience from which we learned a great deal, ranging from technical skills to managing and working in large teams.

6 Future Improvements

6.1 Payments

One of the key improvements we plan to implement in the near future is the integration of a payment processing system. Adding a payments solution, such as Stripe or another equivalent provider, will allow us to introduce subscription-based services and financial transactions into our platform. This would be particularly useful for monetizing our service by offering premium features or extended user access based on subscription tiers. Additionally, the implementation of such a system will enhance the overall user experience by enabling secure, automated billing and providing users with convenient payment management options, such as changing payment methods and managing recurring payments.

With this improvement, we recognize the importance of dividing our current microservices architecture for scalability, maintainability, and clearer responsibility segregation. The merging of the *User* and *Subscription* microservices into one service was a practical choice at the time of development but once we integrate a payment system and introduce the new features explained, it would be beneficial to separate the *User* and *Subscription* microservices.

6.2 Kubernetes and Scalability

As our platform continues to grow, ensuring its scalability will become increasingly important. Currently, we are working with a relatively simple architecture, but as the number of users, transactions, and data processing requirements increase, we will need a more robust solution to handle the growing demand. Kubernetes, a powerful container orchestration tool, will play a vital role in improving the scalability and efficiency of our infrastructure.

Our microservices are already containerized and this helps to deploy them on a Kubernetes cluster, which will earn us several advantages. Kubernetes will allow us to automatically manage, scale, and monitor our application services in real time. This will enable us to handle increases in traffic more effectively by automatically scaling services up or down based on current demand. Furthermore, Kubernetes provides built-in support for load balancing, which will ensure that our platform remains responsive and reliable even under heavy load conditions.

Additionally, Kubernetes will facilitate the deployment and management of new services or features, ensuring that we can iterate quickly and deploy new versions of services with minimal downtime.

By leveraging Kubernetes, we will not only improve the platform's ability to scale

horizontally but also ensure that the underlying infrastructure can support future enhancements, such as the addition of new tools or more advanced payment systems. Ultimately, Kubernetes will help us maintain high availability, resilience and fault tolerance, ensuring a seamless user experience as we scale.

6.3 Automation with Ansible

Automation plays a crucial role in maintaining the efficiency and consistency of deployment processes, particularly in complex microservices architectures like ours. To streamline deployment, configuration management, and application updates, we plan to incorporate Ansible into our development workflow.

Ansible is a powerful automation tool that allows us to define infrastructure as code using simple, human-readable YAML files. By integrating Ansible, we can automate repetitive tasks such as provisioning servers, deploying microservices, and configuring dependencies across various environments. This will significantly reduce the likelihood of human error while saving valuable development time.

The use of Ansible will also enhance the scalability of our platform. As we expand to include Kubernetes for container orchestration, Ansible can be used to set up and maintain Kubernetes clusters, ensuring a seamless transition to this new architecture. Furthermore, Ansible's idempotent nature ensures that repeated runs of automation scripts will result in consistent configurations, even as the infrastructure evolves.

Another advantage of Ansible is its compatibility with various cloud providers and on-premises setups, allowing us to maintain a flexible and portable deployment pipeline. This aligns well with our long-term goals of supporting multiple environments and ensuring high availability.

In summary, by incorporating Ansible, we aim to achieve greater efficiency, reliability, and scalability in our deployment and management processes. This will lay a solid foundation for future enhancements while enabling our team to focus more on innovation and less on operational overhead.

7 Conclusion

This report provides a comprehensive overview of the system's development, covering its objectives, architectural choices, implementation, and evaluation of requirements.

We examined the microservices implemented and analyzed how well the system meets functional and non-functional requirements, ensuring it aligns with the project's goals. The critical analysis of the development phases highlighted successes and lessons learned, offering insights into the system's lifecycle, from requirements gathering to implementation.

Future improvements were also outlined, including areas such as scalability, automation, and architecture refinements, to ensure the platform evolves to meet user needs effectively. Overall, the project demonstrates significant progress toward building a robust and maintainable system, while identifying clear pathways for future growth and optimization.