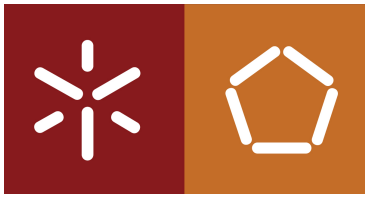
Universidade do Minho



Mestrado em Engenharia Informática

Requisitos e Arquiteturas de Software

2º Fase - 15 de Novembro de 2024

Ana Alves
PG57505

Augusto Campos
PG57510

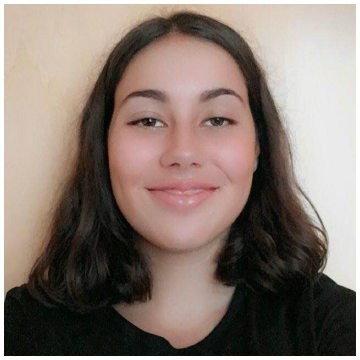
Guilherme Krull
PG55090

Iyán Riol
E12141

Tiago Silva
PG57617

Marta Gonçalves
PG55983

Francisco Lameirão
pg57542



Luna Figueiredo
PG55979

João Abreu
PG55895

Índice

1 Introduction and Goals

1.1 Overview

This document presents the architecture that will serve as the foundation for implementing the PictuRAS application.

The architectural decisions described here are based on the information collected about this application and outlined in the requirements document. This document follows the arc42 template [1], which is specifically designed for presenting and documenting application architectures.

1.2 PictuRAS

PictuRAS is a web-based image processing application that offers users a wide range of image editing tools, from basic functions to advanced capabilities.

The application supports different user profiles, each determining the level of access to its features:

- **Anonymous users**: Limited access to functionalities.
- **Registered users**: Access with some restrictions.
- **Premium users**: Full access without any limitations.

The design and functionality of PictuRAS were based on research into similar applications and the entire process of gathering requirements, enabling well-founded decisions about its operation, to allow it to compete with other similar applications.

1.3 Application Functionality

The functionalities of the application, as described in the requirements document through the functional requirements, are divided in three main categories: account management, project management and tool application.

Account Management

Account management allows users to:

- Register, creating a new account;
- Authenticate into their accounts;
- Choose the subscription plan;
- Access and update profile information;
- Perform other account-related operations.

Project Management

Projects serve as an organizational framework for managing imported images and applying tools to them. Users are able to:

- Create, access, or delete projects;
- Import images for the projects;
- Remove images within projects;
- Add and reorder tools and their respective parameters;
- Process images and save the resulting outputs.

Tool Application

The tools available in the application are categorized into:

- **Basic tools**: Used to adjust image attributes such as brightness, contrast, saturation, color, angle, and borders.
- **Advanced tools**: Usually powered by artificial intelligence, these tools enable operations such as object detection and manipulation.

1.4 Quality Goals

Non-functional requirements play a critical role in the selection of an appropriate architecture.

Among the various non-functional requirements outlined in the requirements document, five have been identified as the highest priority for architectural decisions due to their relevance to quality metrics.

While other non-functional requirements are also important, these selected metrics have the greatest impact on defining a robust architecture for the application.

Usability (Requirement 2)

Requirement: The application must be easy to use.

Fit Criterion: The time required to identify and access a functionality must not exceed 5 seconds.

Architectural Implication: The architecture must facilitate the creation of an intuitive user interface that prioritizes accessibility and minimizes user complexity, enabling users to navigate the application effortlessly, without the need for instructions.

Scalability and Elasticity (Requirement 23)

Requirement: The application must be capable of horizontal, elastic scaling to support increased users and processing volume while maintaining performance and controlling costs.

Fit Criterion: The system must support 100% user growth every 10 minutes without significant performance degradation (less than 20% variation in execution time). Resources are deallocated when no longer needed.

Architectural Implication: The architecture should support a significant increase in simultaneous users while maintaining high levels of performance. It should also be able to implement mechanisms that control costs during periods with less users on the application.

Integration with other Platforms (Requirement 25)

Requirement: The application must be integrable with other platforms and third-party services.

Fit Criterion: Well-documented APIs that enable integration with external services.

Architectural Implication: The system must be structured to allow smooth communication with external platforms, enabling PictuRAS to exchange data and functionality with third-party services. This requires designing flexible interfaces, such as APIs, that can adapt to a wide variety of integration scenarios. The architecture should facilitate secure and reliable communication between PictuRAS and external systems without introducing unnecessary complexity.

Extensibility (Requirement 28)

Requirement: The application must be easily extended with new editing tools.

Fit Criterion: The interfaces of the application's various software components must be very well defined and prepared to be complemented using different technologies, languages, and patterns.

Architectural Implication: This ensures that PictuRAS can evolve over time, accommodating different technologies and supporting different tools from the ones initially planned without being constrained by the original design. For that to happen, the architecture must be modular to allow the easy addition of new image processing tools without disrupting the existing functionalities.

Testability (Requirement 32)

Requirement: The system must be designed to facilitate the execution of tests.

Fit Criterion: Regression tests must be performed before each update to ensure that previous functionalities are not affected.

Architectural Implication: The design must support simplified execution of tests at various levels of the application, from unit tests to integration tests. It should encourage a clear separation of the concerns so that each component or service can be tested independently, without the need to run the whole application.

1.5 Stakeholders

The various stakeholders identified in the requirements document have different expectations for the project, each placing a higher value on different quality metrics.

Client

The client is the company that hired us as a team to develop the application. They aim to have a platform with optimal performance and an organized architecture, allowing easy modifications for any team that might need to implement changes, fix issues, and so on. Usability is a key factor, as it will help attract a larger number of users willing to pay for the application's services. Consequently, high scalability and good elasticity are necessary to ensure good performance for all users and that only necessary costs are incurred over time. Finally, future innovations, such as new tools or improvements to existing features, demand a system that is extensible and easy to test.

Consumers

Consumers primarily focus on the application's usability, ensuring they can easily perform their desired operations on images. However, it is also important to maintain good performance and to avoid prolonged downtime during updates or maintenance. Therefore, there is implicit interest in scalability, extensibility, and testability.

Software Engineers and Developers

Anyone who may work on the application require a clear and well-defined architecture to better understand the implementation, being particularly interested in extensibility and testability to facilitate necessary modifications or the addition of new tools without having to completely halt the application for extended periods.

Marketing Team

Finally, the marketing team does not have any direct interest in the application's architecture. However, their efforts benefit from good usability and performance, which help promote the application effectively.

2 Constraints

The following constraints share similarities with those outlined in the requirements document but are specifically focused on the architecture of the system.

2.1 Organizational Constraints

Constraint	**Description**
Project deadlines	The project must adhere to critical deadlines, including the delivery of the complete system architecture by the 15^{th} of November 2024 so the final implementation with all primary functionalities can be finished by the 17^{th} of January 2025. Timely delivery is essential to ensure stakeholder feedback and enable necessary adjustments in subsequent phases.
Budget limitations	The project must adhere to strict budget limitations, particularly regarding cloud hosting costs. Storage expenses should be closely monitored and accurately estimated to avoid exceeding the allocated budget. Additionally, costs associated with scaling the infrastructure to handle usage spikes must be carefully planned. Effective budget management is essential to ensure the financial sustainability of the project, preventing overspending on cloud resources.

2.2 Technical Constraints

Constraint	**Description**
Web Platform	Application must be developed exclusively as a web application, accessible to any user with an internet connection globally
Adaptability	Application must be designed taking into consideration that it may be used in different devices, ensuring compatibility across various screen sizes, operating systems, and input methods. This requires a responsive design approach and a user-friendly interface that remains intuitive and functional regardless of the device or context of use.
Cloud Infrastructure	Application must be deployable on a cloud platform, implementing cloud-native best practices for resource optimization, such as containerization and efficient resource allocation to ensure high availability and fault tolerance
Security	Application must implement protection against common web vulnerabilities, following security best practices for cloud deployments and focus on implementing robust authentication and authorization mechanisms
Privacy Compliance	Application must comply with GDPR requirements and implement appropriate user data management policies, providing transparency in data collection and processing
Performance	Application must optimize system response times for global access, even with an increase on users and complexity of the operations, maintaining efficient system responsiveness.
Scalability	The application must be capable of handling increasing numbers of users and growing data volumes, ensuring that resources can accommodate higher demand and usage peaks without any problems.

3 Context and Scope of the System

In this chapter, we will address the scope and context of the system, defining the system in relation to its scope, neighboring (external) systems, and users (communication partners).

To do so, we will consider two perspectives: the business context and the technical context.

The business context focuses on market needs, the value proposition of the application, and the objectives that the company aims to achieve with the implementation of the system.

On the other hand, the technical context is concerned with the technologies and challenges the development team will face in building the application in a scalable, secure, and efficient manner.

These two contexts are essential for aligning the technical development of the project with the business objectives and goals of PictuRas. They ensure that the proposed solution is not only technically feasible but also meets market needs and adds value to the company.

3.1 Business Context

In this section, we address the context of our system, defining the domain of our system within a limited set of functionalities and external services, such as the existence of an external service responsible for managing payments. It is well known that the business context should focus on the needs and expectations of the relevant audience, so it is important to clearly and generically reference the stakeholders.

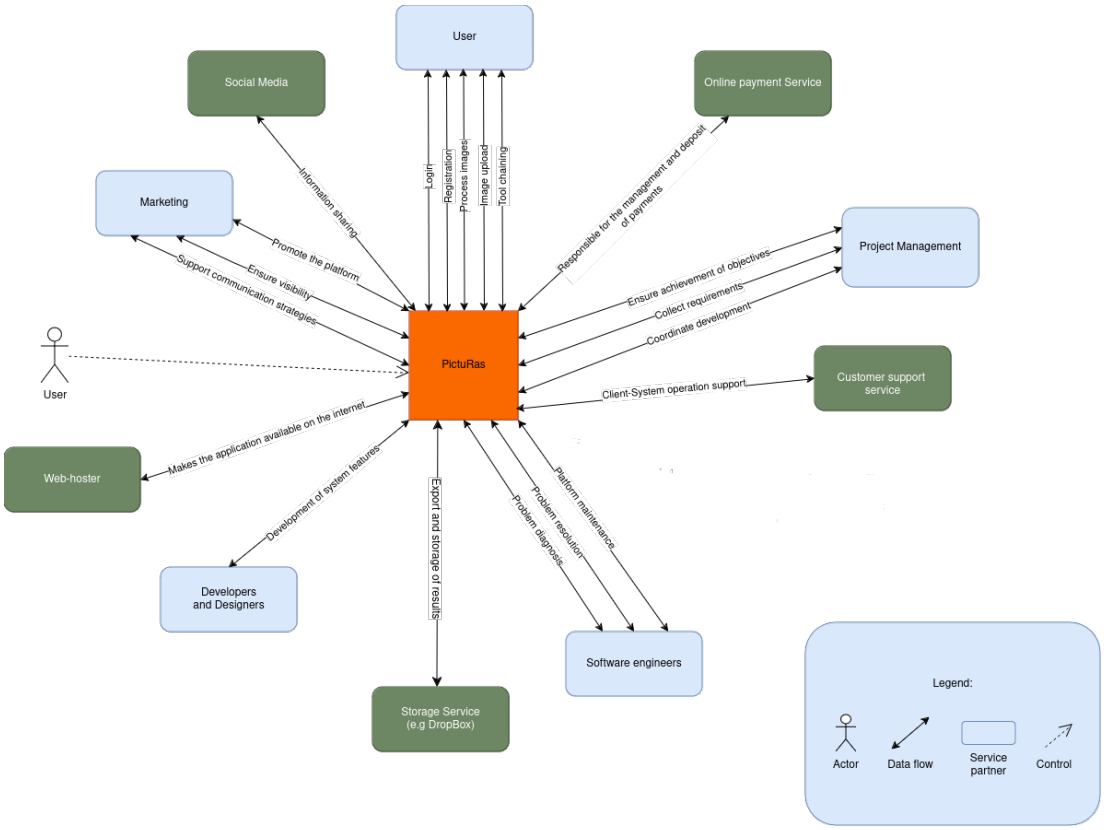


Figure 1: Context Diagram(Business)

Risks

External interfaces can pose a risk regarding the implementation of non-functional requirements and the fulfillment of quality goals. Concerning system availability, which is related to the non-functional requirement of usability, the use of external interfaces, such as the MBWay payment provider, may affect it because the system's availability becomes tied to the availability of these interfaces.

Regarding performance, the system can also become dependent on the interfaces it communicates with. For example, in the case of customer support, which should be available via chat or email with quick response times, this metric may be compromised if the external interface has worse performance or is unavailable.

On the other hand, regarding security, if sensitive data is sent or received, this could make the system less secure and more vulnerable to potential attacks.

As for system maintenance, the complexity of external interfaces could partially undermine this requirement, as they may use complex data structures, for instance, requiring reorganization and development of code designed to handle them. Furthermore, it could pose a challenge if maintaining the interface is especially difficult and requires significant manual effort.

3.2 Technical Context

In this section, we address the technical context of our system, that is, we present a technical-level context diagram that describes the interaction between the proposed system and its external components, highlighting the technical dependencies, interfaces, protocols, and interactions within the system. It provides a high-level view of the system's technical architecture, without delving into implementation details.

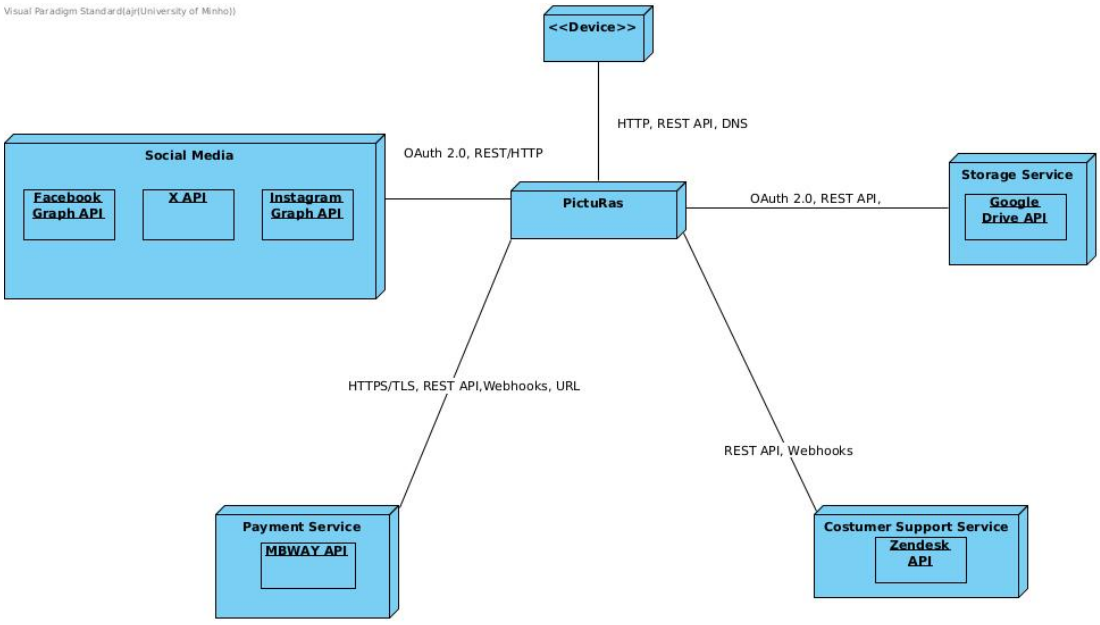


Figure 2: Context Diagram(Technical)

4 Solution Strategy

In this section, we will explore the different technical possibilities for our implementation with the aim of selecting our priorities, as well as organizing and associating the previously established requirements to the microservices context, considering the defined constraints. Thus, we will address the functional decomposition of the system, discuss the choice of architectural pattern based on the selection of non-functional requirements made in the previous section of this document, define the system's data model, and outline the technologies to be used in the implementation. Additionally, we will provide a description of the development team organization for the various system components to be developed.

4.1 Architectural Pattern

Based on the most important non-functional requirements and the ratings from the book "The Fundamentals of Software Architecture: An Engineering Approach" by Mark Richards and Neal Ford [2], we conducted the following comparison between architectures to evaluate how advantageous it would be to use a microservices-based architecture. The left table refers to classifications related to a microservices architecture, while the right table refers to a layered architecture.

Architecture characteristic	Star rating
Partitioning type	Domain
Number of quanta	1 to many
Deployability	★★★★
Elasticity	★★★★★
Evolutionary	★★★★★
Fault tolerance	★★★★
Modularity	★★★★★
Overall cost	★
Performance	★★
Reliability	★★★★
Scalability	★★★★★
Simplicity	★
Testability	★★★★

Architecture characteristic	Star rating
Partitioning type	Technical
Number of quanta	1
Deployability	★
Elasticity	★
Evolutionary	★
Fault tolerance	★
Modularity	★
Overall cost	★★★★★
Performance	★★
Reliability	★★★
Scalability	★
Simplicity	★★★★★
Testability	★★

Figure 3: Ratings from the book "The Fundamentals of Software Architecture: An Engineering Approach"

In this comparison, a one-star rating in the characteristics table signifies that a specific architectural characteristic is poorly supported within that architecture. Conversely, a five-star rating indicates that the characteristic is a primary strength within the architectural style.

ReqID	Description	Feature	Microservices	Layered
RNF04	The application must be easy to use	Usability	☆☆☆☆☆	☆☆☆☆☆
RNF23	The application must be able to scale horizontally to support increasing users and image volume while maintaining performance	Elasticity and scalability	☆☆☆☆☆	☆
RNF25	The application must be integrable with other platforms and third-party services	Scalability	☆☆☆☆☆	☆
RNF28	The application must be easily extendable with new editing tools	Modularity	☆☆☆☆☆	☆
RNF32	The system should be designed to facilitate testing	Testability	☆☆☆☆	☆☆

Table 1: Architectural Solution

Based on the table above, the "Microservices" architecture style is rated highly across most of the characteristics, particularly in usability, scalability, and extensibility, which are crucial for modern software systems that require flexibility and ease of growth. This approach aligns well with the system's need for handling an increasing number of users, integrating with third-party services, and adding new editing tools without significant rework.

4.2 Functional Decomposition

PictuRas is a software system of moderate complexity but relatively detailed. For this reason, the development team analyzed the different functional requirements, grouping them into potential microservices. This allowed them to understand which microservices would need to be implemented and which functionalities should be considered. Furthermore, the use cases were essential in understanding how the microservices could be interconnected.

IdReq	**Description**	**Microservice**
RF01	The user authenticates using their credentials	User auth
RF02	The anonymous user registers	User auth
RF03	The user chooses the Free subscription plan	User Management
RF04	The user chooses the Premium subscription plan	User Management
RF05	The user pays for the Premium subscription plan	Payment Management
RF06	The user creates a project	Project Management
RF07	The user lists their projects	Project Management
RF08	The user accesses the editing area of a project	Project Management
RF09	The user uploads images to a project	Project Management
RF10	The user removes an image from the project	Project Management
RF11	The user adds an editing tool to the project	Project Management
RF12	The user triggers the processing of a project	Project Management → Other Tools Management
RF13	The user downloads the result of a project to their local device	External
RF14	The user changes the order of images in a project	Project Management
RF15	The user changes the order of tools in a project	Project Management
RF16	The user changes the parameters of the tools	Project Management

RF17	The user cancels the processing of a project during execution	Project Management
RF18	The registered user accesses their profile information	User Management
RF19	The user edits their profile	User Management
RF20	The user with a Premium subscription changes the payment frequency to either monthly or annual	User Management → Payment Management
RF21	The Premium user cancels their Premium subscription	User Management → Payment Management
RF22	The registered user ends their session	User Management
RF23	The registered user removes their account and data	User Management
RF24	The user shares the result of an image edit directly on social media	External
RF25	The user manually crops images	Management of the Crop Tool
RF26	The user resizes images to specific dimensions	Management of the Resize Tool
RF27	The user adds a border to images	Management of the Add Colored Border Tool
RF28	The user adjusts the saturation of images	Management of the Saturation Adjustment Tool
RF29	The user adjusts the brightness of images	Management of the Brightness Adjustment Tool
RF30	The user adjusts the contrast of images	Management of the Contrast Adjustment Tool

RF31	The user binarizes images	Management of the Binarization Tool
RF32	The user rotates images	Management of the Rotation Tool
RF33	The user applies an automatic image crop algorithm based on content	Management of the Smart Crop Tool
RF34	The user applies automatic image optimization adjustments based on content	Management of the Auto-Adjustment Tool
RF35	The user removes the background of an image, keeping only the main object	Management of the Background Removal Tool
RF36	The user extracts text from images	Management of the Text Extraction Tool
RF37	The user applies an object recognition algorithm to images	Management of the Object Recognition Tool
RF38	The user applies a people-counting algorithm to images	Management of the People Counting Tool

Table 2: Association of functional requirements to microservices

Subsequently, based on the conclusions drawn from the analysis above, a block diagram was created that divides the system's responsibilities at various levels of depth. This helps in gaining a better understanding of the system, thereby facilitating its development.

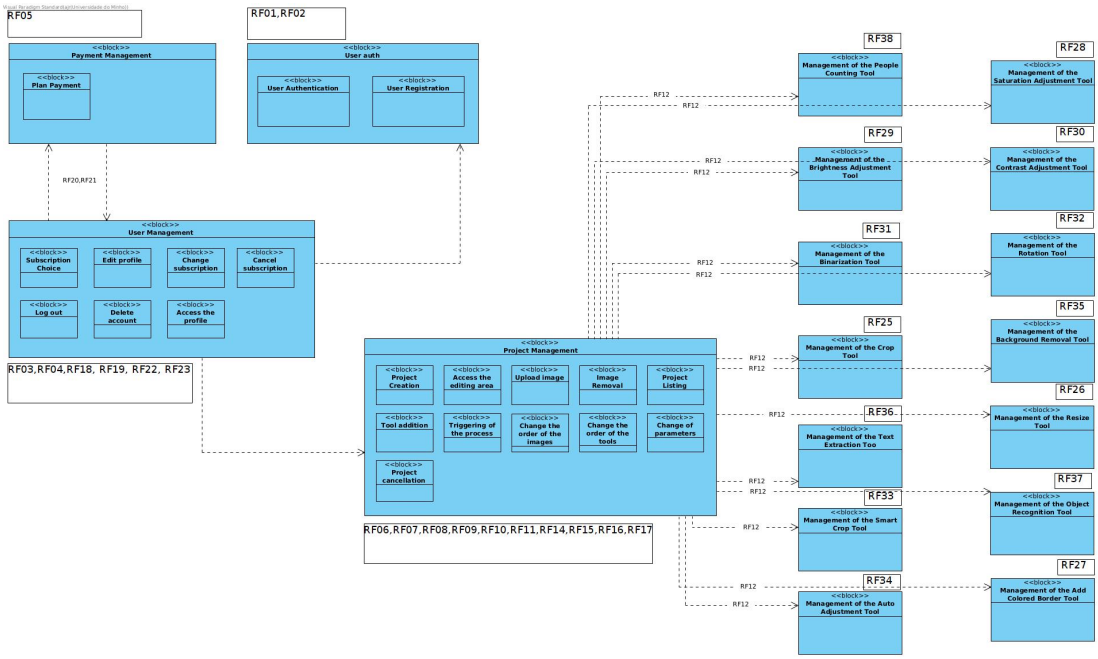


Figure 4: Block diagram - Functional decomposition

4.3 Data Models of the Microservices' Databases

In this section, we will present, for the established microservices, the data dictionaries for each of the associated databases. In the case of the tools, we believe that it would not be necessary to use databases, as they do not require long-term persistence. The tools are primarily focused on real-time image manipulation within the projects created by the users, and the data generated during the use of the tools can be processed in-memory. This way, we avoid the overhead of maintaining an independent database for each tool, simplifying the system's architecture and improving its performance.

User Auth

For user authentication, we will use a MongoDB database with a collection of data from users who have logged in.

Field Name	Type	Description
_id	String	Username of the user
email	String	Email address of the user
password	String	Encrypted User Password

Table 3: Data Model - User Auth

User Management

For user management, we will use a MongoDB database with a collection of users.

Field Name	Type	Description
_id	String	User's username
email	String	User's email
name	String	User's full name
password	String	Encrypted User Password
type	String	User's account type (Registered or Premium)
bankNumber	Integer	User's bank card number

Table 4: Data Model - Users

Payment Management

For payment management, we will use a MongoDB database with a collection of completed payments.

Field Name	Type	Description
_id	String	Payment identifier
userId	String	Username of the user who made the payment
bankNumber	String	Bank number of the user who made the payment
amount	String	Amount paid by the user
date	String	Date the payment was made

Table 5: Data Model - Payments

Project Management

For project management, we will use a MongoDB database where we will have a collection of projects completed by users.

Field Name	Type	Description
_id	String	Project identifier
userId	String	Identifier of the user responsible for the project
linksBase	Array	List of links associated with each image in the project
tools	Array	List of IDs of tools used in image transformation
tool.id	Integer	Unique identifier of the tool used
tool.name	String	Name of the tool used
tool.parameter	String	Specific parameter of the tool

Table 6: Data Model - Projects

4.4 Technologies for Implementation

In this section, we will describe the technologies selected by the development team to implement the system, outlining the reasons why they were chosen.

Microservices

In the microservices applications, the programming language JavaScript will be used, along with the Node.js runtime environment and the Express framework, which facilitates the process of creating REST APIs. For communication with the database server, we will use the Mongoose library to connect to MongoDB databases. Additionally, microservices related to tools for image processing may use other languages, such as Python, due to its wide range of specific libraries for the various types of processing we aim to implement.

Databases

The teams of each microservice, after the design of each of their respective microservices, concluded that the technology to be used for data persistence, and the Database

Management System (DBMS) for all, would be MongoDB, a document-oriented NoSQL database. This choice reflects the nature of the microservices requirements and the specific benefits MongoDB offers for microservices architectures.

The main reasons that influenced the adoption of this DBMS, and the consequent use of a NoSQL data model, are as follows:

- The flexibility provided by adopting a document-oriented database. This flexibility is evident in the ability to store different documents (related to the same entity) in a single collection.
- The document-oriented structure facilitates the development and maintenance of new features, allowing for quick adaptation to new requirements without significant refactoring.

Frontend

For the frontend development, we will use JavaScript in conjunction with the React library, which facilitates the creation of a dynamic and responsive user interface. React is widely used due to its component-based architecture, allowing the interface to be divided into small, reusable, and independent parts. This modular approach simplifies code maintenance and promotes component reuse, while also making it easier to expand the interface as the application grows more complex. For the visual layer, we will use CSS, which will be responsible for the application's styling and layout.

Docker and Containerization

To implement and manage containers that support the microservices infrastructure, the Docker platform will be used. Docker allows each system component to be isolated within a container, ensuring that dependencies and the runtime environment are consistent and independent of the host system's specifications.

With the use of containers, microservices, the API Gateway, and other system components (such as the frontend and database) can be executed in isolated environments, facilitating continuous development, testing, and deployment. Additionally, Docker simplifies horizontal scaling, enabling new containers to be instantiated for load balancing or to respond to demand spikes.

The development team will use Docker Compose to orchestrate and configure the different containers, defining the dependencies and environment variables for each service in the system.

Kubernetes and PVCs

To further enhance the deployment and management of the microservices architecture in containers, Kubernetes will be used alongside Docker. Kubernetes, an open-source container orchestration platform, was chosen for its ability to efficiently manage and scale large numbers of containers, automate deployments, and facilitate load balancing and failure recovery.

Additionally, the PVC (Persistent Volume Claim) technology will be used, a feature in Kubernetes that allows applications to maintain access to persistent data regardless of container lifecycles. PVC is especially useful in microservices architectures where some services require persistent storage for data that must survive pod restarts.

Draft of the Architecture Diagram

In conclusion, we outlined the system to structure all the technological alternatives mentioned. The architecture presented is based on a microservices system deployed in a Kubernetes Cluster, ensuring scalability and high modularity. User traffic reaches the system through browsers and passes through Load Balancers, which distribute requests to the API Gateway. The API Gateway coordinates communication between various backend services, such as User Management, User Authentication, Payment Management, and Project Management, each operating in separate Docker containers with persistent storage in MongoDB databases.

Additionally, the Project Management Service connects to specialized services for data processing, such as cropping, resizing, contrast adjustment, saturation, and rotation. These services are developed in languages such as Python, C#, and Java, and they utilize Kubernetes for internal communication via ClusterIP. The infrastructure is managed by Kubernetes, which organizes pods, persistent volumes, and load balancing, ensuring the system's efficiency and resilience.

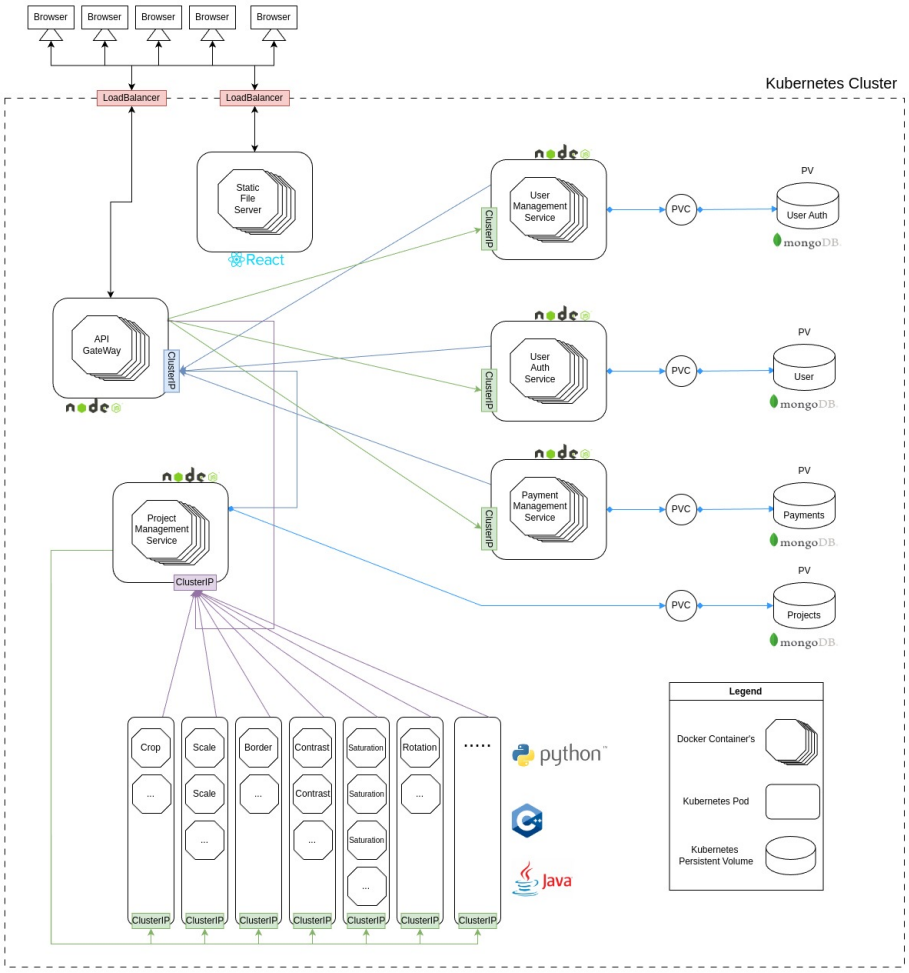


Figure 5: Draft of the Architecture Diagram

4.5 Organizational Decisions

The development team is composed of 9 members, and it was decided that all team members should be distributed across the various microservices. The team was strategically organized, with more members allocated to more complex microservices, such as Project Management, since it requires a greater number of processes to consider. This organization also facilitates the specialization of each subgroup within the team in their respective microservice, promoting a more focused development on specific objectives, accelerating the development process, and strengthening collaboration, while minimizing points of failure and maximizing efficiency and productivity, as each group gains a deeper understanding of their own service.

Microservice	**Number of team members**
User Auth	2
User Management	3
Payment Management	2
Project Management	4
Crop Tool Management	1
Resize Tool Management	1
Add Colored Border Tool Management	1
Adjust Saturation Tool Management	1
Adjust Brightness Tool Management	1
Adjust Contrast Tool Management	1
Binarization Tool Management	1
Rotation Tool Management	1
Smart Crop Tool Management	1
Auto Adjustment Tool Management	1
Background Removal Tool Management	1
Text Extraction Tool Management	1
Object Recognition Tool Management	1
People Counting Tool Management	1

Table 7: Distribution of Team Members Across Microservices

Regarding the frontend, each microservice team will be responsible for the microservice they developed (for example, the User Management microservice team will be responsible for developing the entire frontend part related to the functionalities of that microservice), as they have a clearer understanding of their microservice's context and how it is implemented. This strategy encourages autonomy, and in the future, if a microservice requires specific changes or adjustments to the interface, only the team responsible for it will need to intervene, simplifying the maintenance process.

5 Building Block View

5.1 Scope & Context (Level 0)

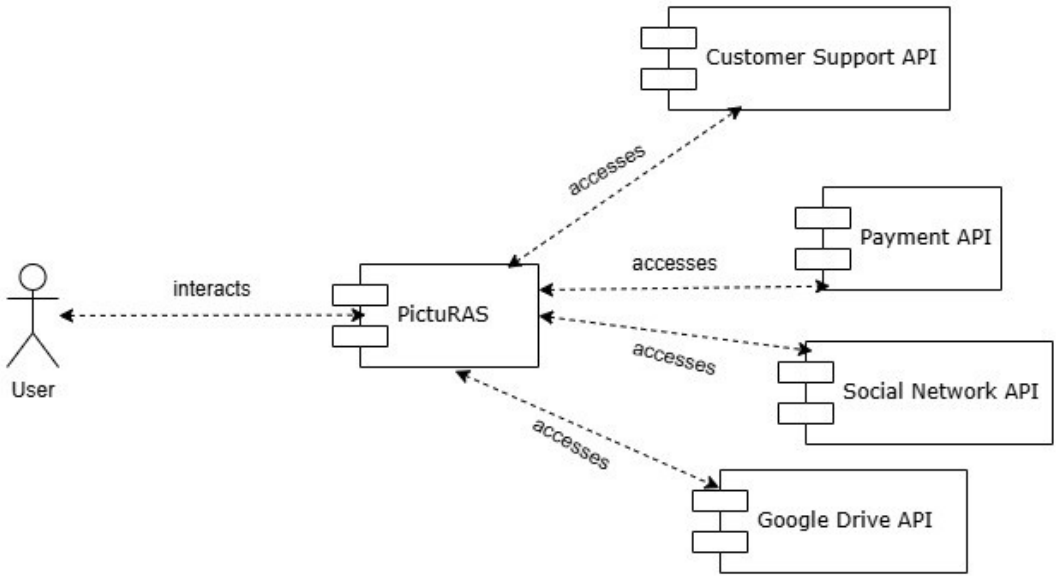


Figure 6: System Level 0

Component	**Responsibility**
User	Interacts with the application, initiating requests and consuming features
PictuRAS	Core image editing application that handles user operations and coordinates with external services
Customer Support API	Provides a channel for users to get help with issues, including FAQs and support ticket management
Payment API	External service that processes and manages user payments securely
Social Network API	External service that allows users to share edited images directly to social media platforms
Google Drive API	Allows users to save and retrieve edited images from their Google Drive account seamlessly

Table 8: Level 0 components responsibilities

5.2 Level 1 - PictuRAS

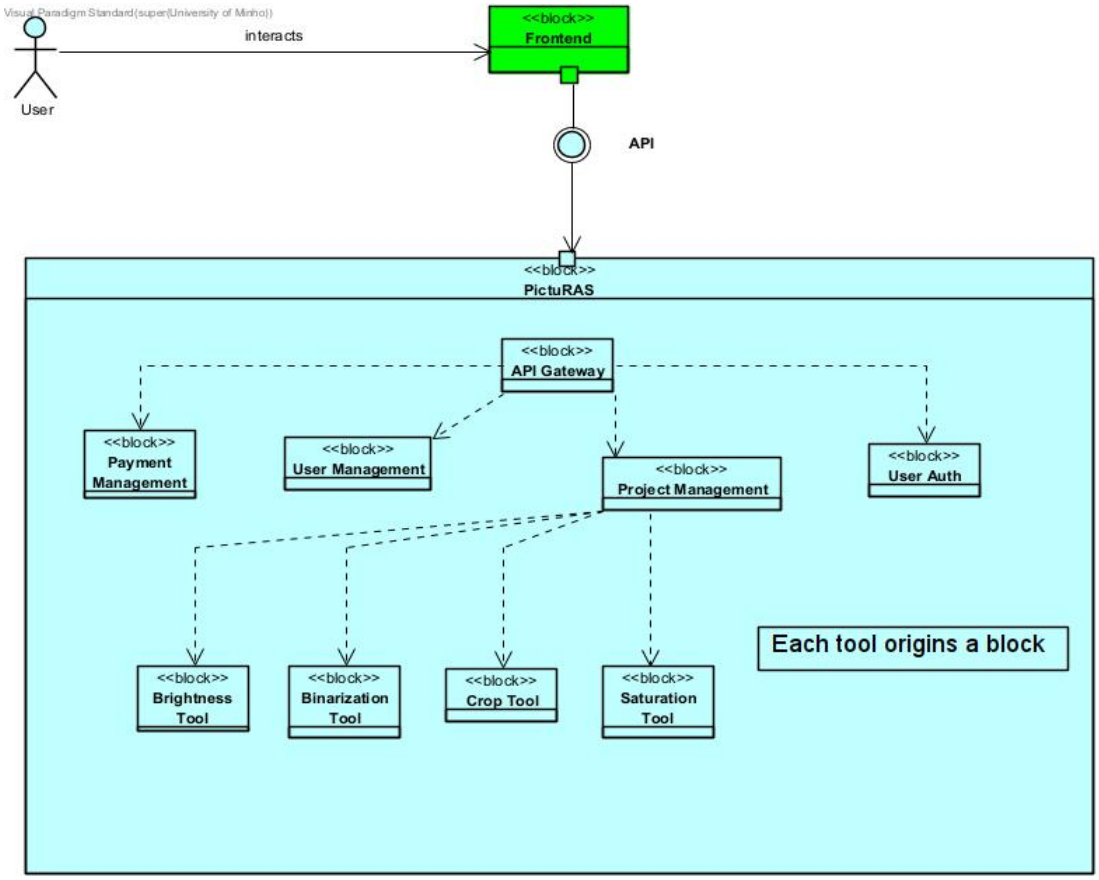


Figure 7: White Box Block View - PictuRAS

Component	**Responsibility**
API Gateway	Acts as the entry point for all client requests, routing them to appropriate services and managing communication between components
Payment Management	Tracks the payments within the app
User Management	Manages the app user base
User Auth	Authorizes users to perform specific actions within the app
Project Management	Manages the entire workflow of image processing
Tool	Applies a specific image transformation

Table 9: Responsibilities of PictuRAS' Black Boxes

5.3 Level 2 - Tool microservice composition

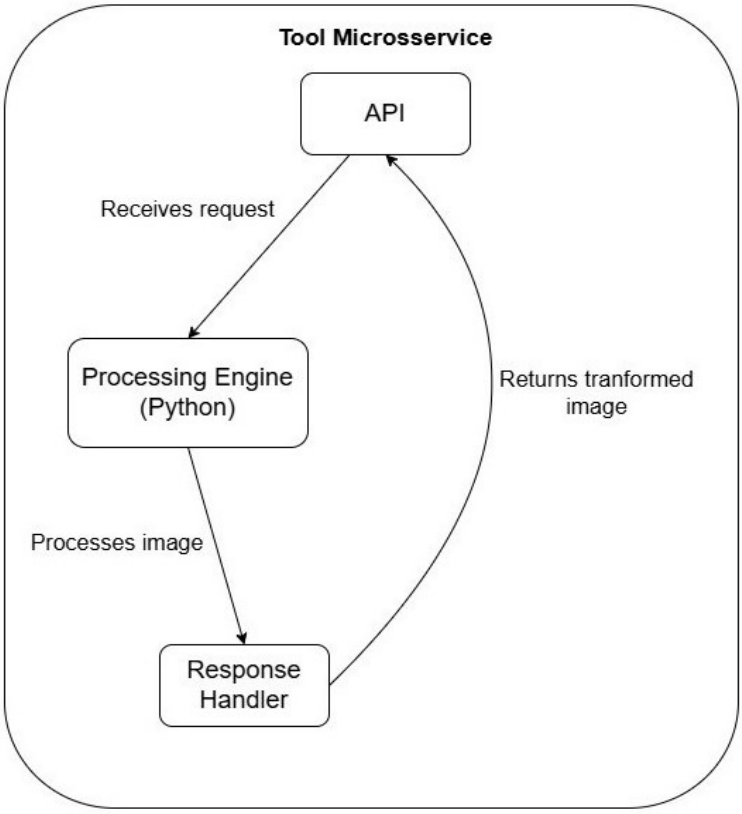


Figure 8: White Box Block View - Tool

Component	**Responsibility**
API	Serves as the entry point for requests, receiving input images and transformation parameters, and passing them to the processing engine
Processing Engine	Executes the image transformation logic (e.g., resize, apply filter) based on the received parameters
Response Handler	Prepares and returns the transformation output to the client in the desired format

Table 10: Responsibilities of the Tool Microservice Components

6 Runtime View

The aim of this section is to illustrate the behavior of the objects defined in the Block view from a dynamic perspective. To achieve this, we will use sequence diagrams to model the key use cases that best represent the interactions and processes within our system. The use cases covered will include user login, user registration, upload images, and applying tool chaining to a set of images.

6.1 User login

This Use Case describes the sequence of events in the functionality in which the User logs in.

The API Gateway will first display the login page to the user. The user will then enter their credentials, including email and password, into the login form. After submission, the API Gateway sends these credentials to the User Auth Service, which checks whether the email and password both exist and match. If either the email or password is incorrect, or if the credentials do not exist, the Auth Service will return an error message, which the API Gateway will display to the user. If the credentials are valid and correct, the API Gateway will proceed to display the PictuRas homepage.

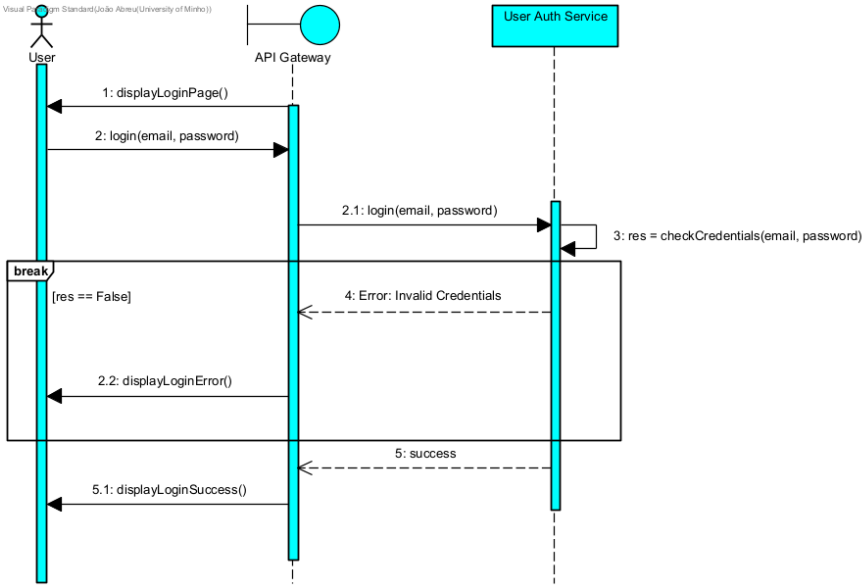


Figure 9: Sequence Diagram for Use Case 6.1 - User Login

6.2 User registration

This Use Case describes the process of registering a user in PictuRas. The user starts by clicking the registration button, which opens the Registration Page. They enter their email and password, and the API Gateway sends this information to the User Auth Service to verify if the credentials are valid and if the email is already registered.

If the credentials are incorrect, an error message appears. If they are valid, the user proceeds to the subscription selection page. Choosing the free plan saves their information in the User Management Service, showing a success message. For monthly or yearly plans, payment details are required.

The user submits payment information, and if incorrect, the Payment Manager requests corrections. The Payment Manager, through the external payment service API, asynchronously validates the provided payment details. Once the payment information is validated successfully, a success message confirms the registration and subscription

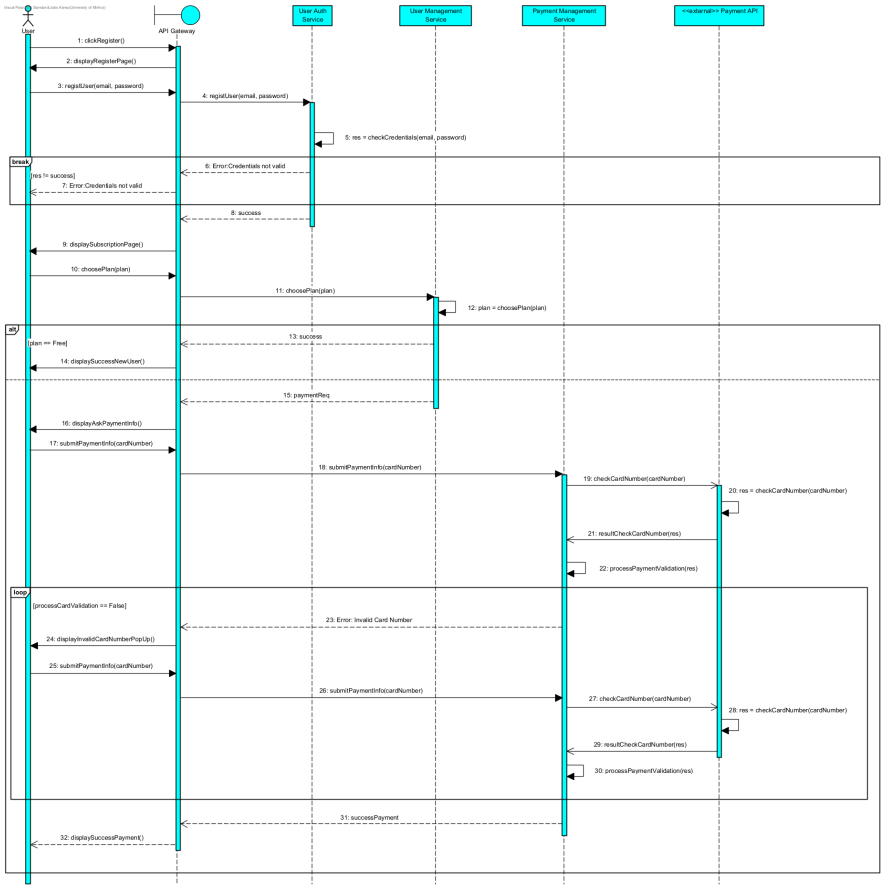


Figure 10: Sequence Diagram for Use Case 6.2 - User Registration

6.3 Upload Images

This Use Case describes the sequence of events for uploading images to new or existing projects.

First, the user uploads the image(s) through the API Gateway. The upload is then verified by the Project Management Service. If successful, the user is presented with an optional form to enter a project name to associate with the image(s).

After entering the project name, there are two options: i) add the image(s) to an existing project, or ii) create a new project and add the image(s).

In both cases, the system checks if the user can upload the images and if they meet the required dimensions. If there are any issues, an error message is displayed. If everything is correct, the image(s) will be successfully associated with the project.

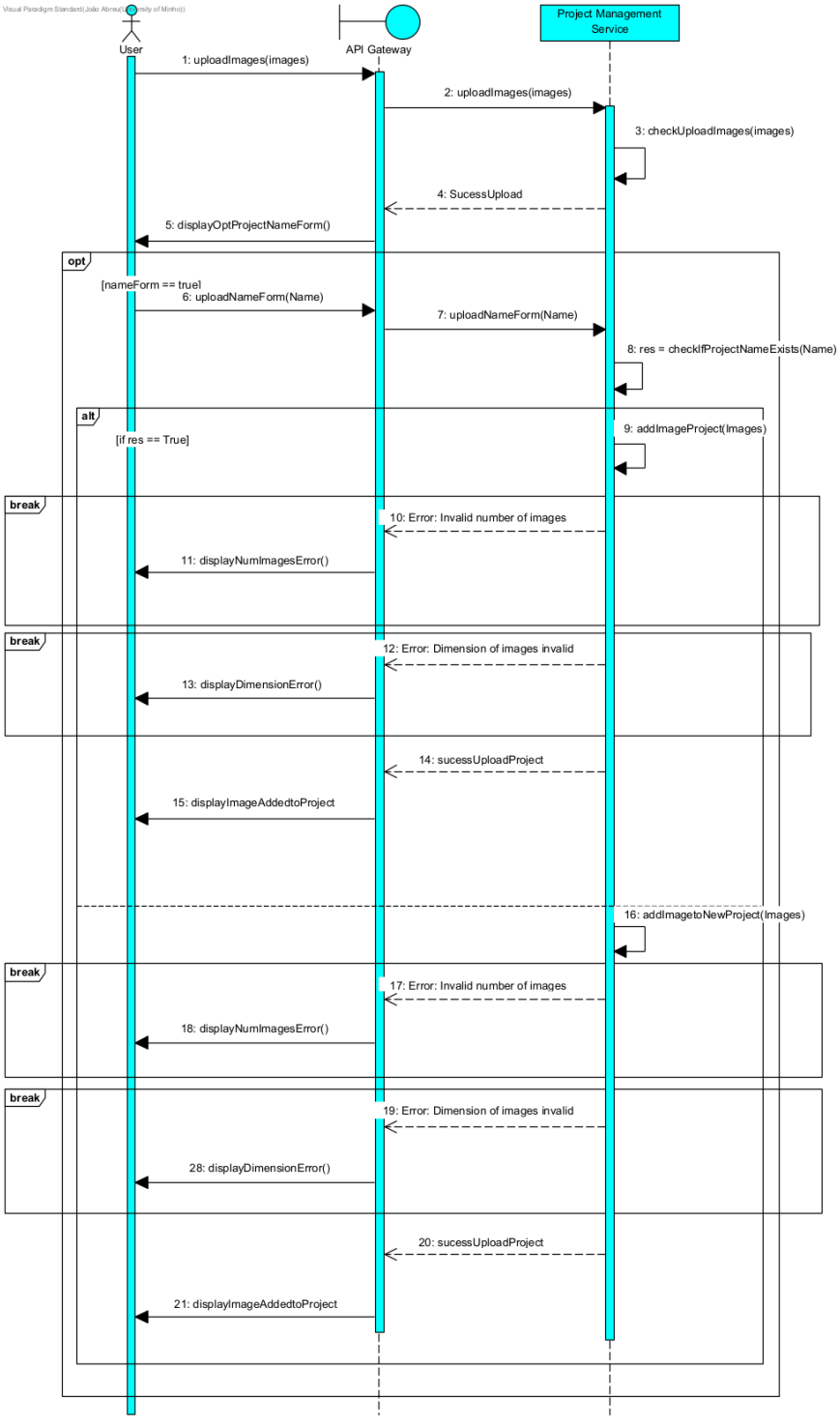


Figure 11: Sequence Diagram for Use Case 6.3 - Upload Images

6.4 Applying tool chaining to a set of images

This Use Case describes the sequence of events required to apply tool chaining to a set of images.

First, the user requests a list of projects, which is returned by the Project Management Service through the API Gateway. The user then selects the project they want to work on, and the Project Management Service presents the editing page for the chosen project.

On this page, the user can select the desired tools and set their parameters, which are sent to the Project Management Service. The service applies each tool and returns a preview to the user. The user then has two options: i) reject the preview and modify the tools or parameters, or ii) proceed with the processing.

In option ii), the system checks if the user has available daily processing slots. If not, an error message is displayed. If they do, the process begins.

During the process, the user receives updates on the image processing progress and has the option to cancel the process if desired.

Once the process is complete, the final image(s) are displayed to the user, who can then download them.

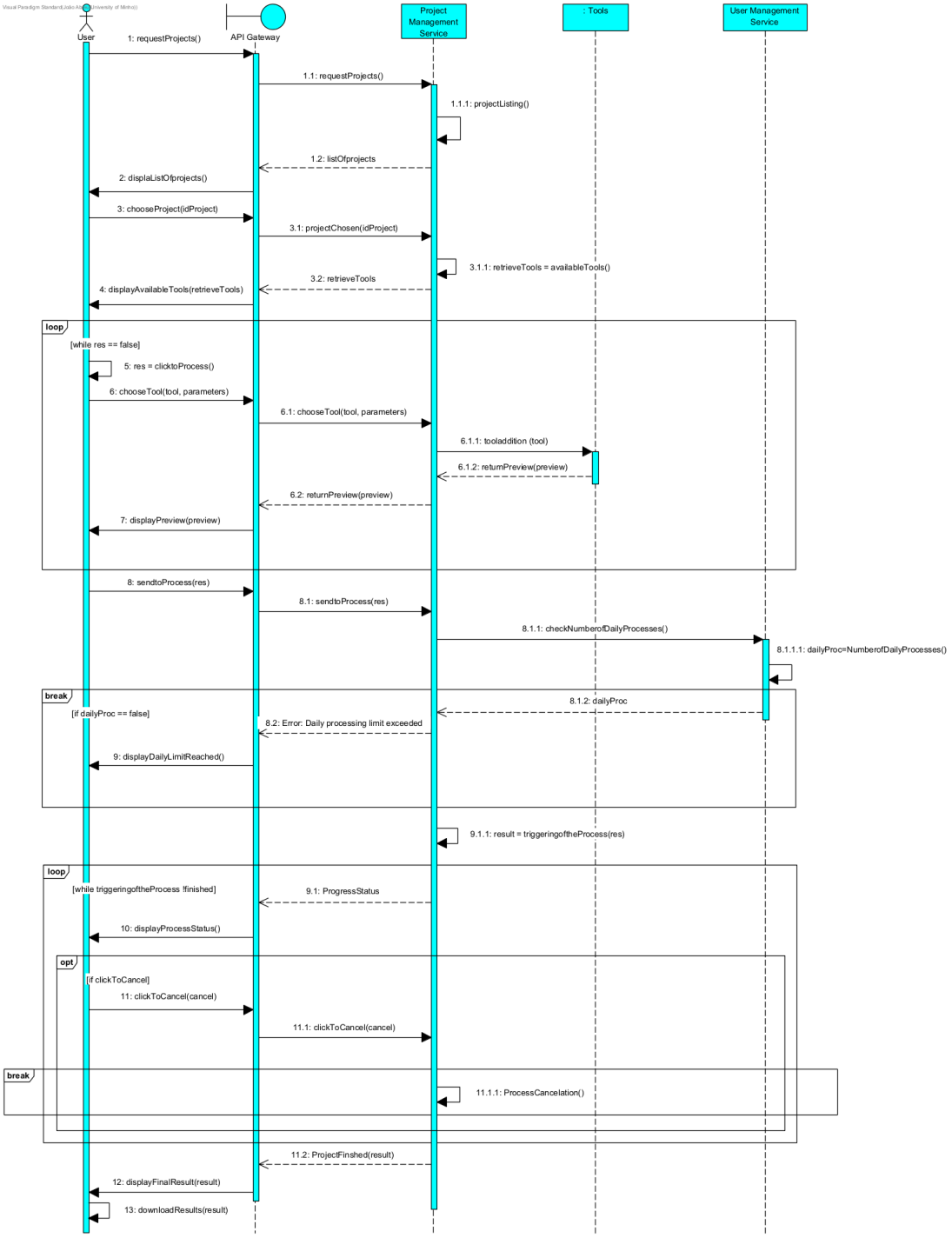


Figure 12: Sequence Diagram for Use Case 6.4 - Applying Tool Chaining to a Set of Images

7 Deployment View

7.1 Technical Infrastructure

The chosen architecture is a Kubernetes cluster. This type of architecture allows the app to scale fast and easily to keep up with the user demands. Kubernetes supports both horizontal pod auto-scaling (adjusting the number of pod replicas inside a node) and cluster auto-scaling (adjusting the number of nodes in the cluster) based on real-time metrics like CPU utilization. Kubernetes also automatically replaces failed containers and ensures that the application is always running as expected. If a pod fails or a node goes down, Kubernetes can redistribute the workload and launch new pods on healthy nodes.

Inside the PictuRAS cluster there are several Kubernetes pods that accomodate all the services this app will need, namely the Web App service, that receives the requests from the user and presents the results, the Payment Management service that tracks the payments within the app, the User Management service that manages the app user base, the User Auth service that will be in charge of authorizing the users to perform specific actions within the app and the Project Management service, that will manage the entire workflow of image processing.

For the Web App service, a minimum of two nodes should always be running, even when the app is in low demand, to minimize the possibility of a single point of failure and avoid the entire app becoming unavailable to the users. When the app has high demand, more replicas and nodes will automatically be created by the auto-scaler.

Regarding the image processing tools, every single one of them is considered a micro-service as well. This gives a finer control of the individual processing tools allowing for independent scalability, portability and overall isolation. These micro-services, however, do not require data persistence.

For data persistence, static Kubernetes Persistence Volumes are used to allow the services to keep track of the state of the app, more specifically the registered users, the payments received and all the users projects.

All the app's micro-services are available through known ClusterIP's. This ensures that services can interact with each other without any of them knowing how pods are configured internally.

Finally, to better isolate the entire infrastructure from the user and to balance the load of the user requests between the Web App Service pods, an Amazon Load Balancer Controller was placed between them.

7.2 Deployment Diagram

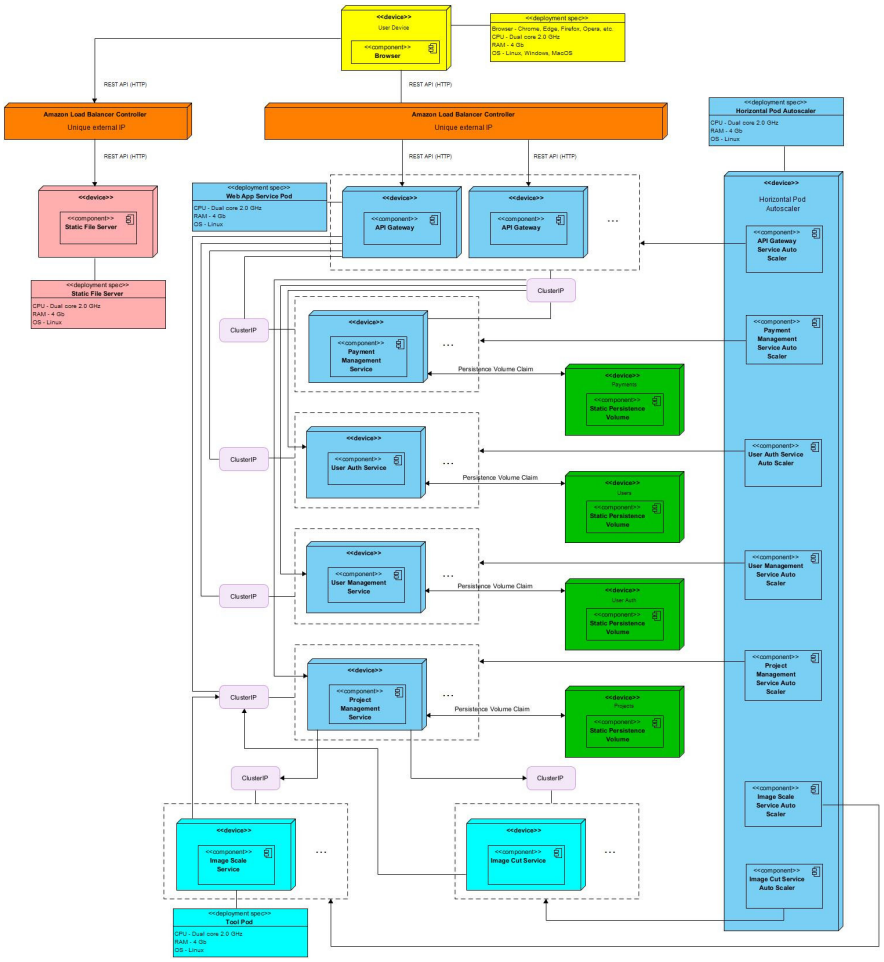


Figure 13: Deployment Diagram

8 Architectural Decisions

8.1 Microservices Division

Decision: We decided to assign every functional requirement to a logical block, and each of these blocks will represent a microservice in the application.

Justification: This approach increases the modularity of the application, making it easier to maintain and to add or remove functionalities without significant effort.

State: Approved

Consequences: Each functional requirement must be mapped to a logical block, and the corresponding functionality must be implemented in the designated microservice.

8.2 Separation of Authentication and User Microservices

Decision: We decided to implement two separate microservices: one for authentication (e.g., login and logout) and another for user-related tasks.

Justification: Separating these concerns ensures better modularity, especially if a third-party authentication service, like Google, is integrated in the future.

State: Approved

Consequences: The functionalities related to user profiles and authentication must be divided and implemented in their respective microservices.

8.3 JavaScript

Decision: We selected JavaScript as a core technology for both the back-end and front-end of our application.

Justification: JavaScript was chosen because the entire development team is proficient in it. It is also a versatile language that works on both the front-end and back-end, enabling faster development, code sharing, and access to a wide range of tools for modern, scalable web apps.

State: Approved

Consequences: The back-end and front-end will both be partially built in JavaScript.

8.4 MongoDB

Decision: We chose MongoDB as the database for our application.

Justification: MongoDB is a flexible, document-based, schema-less database optimized for large datasets. Its integration with JavaScript/Node.js and adaptability to changing requirements make it a strong candidate for modern applications.

State: Approved

Consequences: MongoDB will manage the application's data storage, providing scalability and flexibility.

8.5 React

Decision: We chose React as the front-end framework for our application.

Justification: React is a component-based library that enables the development of modular, interactive, and high-performance user interfaces. Its large ecosystem of libraries and tools, along with the team's familiarity, supports its adoption.

State: Approved

Consequences: The user interface will be built with React, ensuring ease of management, modularity, and a smooth user experience.

8.6 Node.js

Decision: We selected Node.js as the back-end framework for our application.

Justification: Node.js allows JavaScript to be used server-side, ensuring language consistency across the stack. Its ability to handle asynchronous operations efficiently makes it suitable for scalable, real-time applications.

State: Approved

Consequences: The back-end will leverage Node.js for efficient request handling and seamless front-end integration.

8.7 Kubernetes

Decision: We chose Kubernetes for container orchestration in our infrastructure.

Justification: Kubernetes simplifies the deployment, scaling, and management of containerized applications. It automates tasks and ensures application reliability and performance, making it ideal for our needs.

State: Approved

Consequences: Kubernetes will be used for container orchestration, enabling dynamic scaling, resource optimization, and high availability.

8.8 Docker

Decision: We decided to use Docker as the primary tool for containerizing applications, in conjunction with Kubernetes for orchestration.

Justification: Docker provides lightweight, consistent, and portable containers, while Kubernetes manages deployment, scaling, and self-healing. Together, they simplify microservices implementation and ensure high availability, load balancing, and efficient resource use.

State: Approved

Consequences: The infrastructure will use Docker for packaging and Kubernetes for managing containers. This will provide:

- **Dynamic Scalability:** Kubernetes will automatically adjust container instances based on system load.
- **High Availability:** Kubernetes will reschedule failed containers and balance the load.
- **Consistency:** Docker will ensure uniform environments across development and production.
- **Efficiency:** Kubernetes will optimize CPU, memory, and storage resources.

9 Cross-Cutting Concepts

In this part we give a brief overview the intersection of key concepts that shape the project, demonstrating how their integration drives innovative and cohesive design solutions.

9.1 Domain Concepts

Image Editing and Processing:

The domain of the application is the specific field that it should serve. In this situation, it revolves around image editing and processing.

9.2 Architecture and Design Patterns

Microservices Architecture:

The architecture of this project is based on microservices, which is a software design pattern that uses a collection of independent services that communicate over APIs to structure an application. As a result, the application becomes more scalable, flexible, and resilient.

9.3 User Experience (UX)

Intuitive:

The application should cater to both beginners and professionals, with the help of features like preview and organized tool categorization.

User Interface:

The user interface is the point of communication between humans and computers, so it should be functional and user-friendly.

Ergonomics:

Ergonomics refers to the science of enhancing user satisfaction when using an application, adapting it to the user and not the other way around. This is achieved by making interactions smooth and efficient.

Internalization:

The application should be multilingual in order to be used by people globally.

9.4 Development Concepts

Continuous Integration and Continuous Deployment (CI/CD):

PictuRAS uses a CI/CD pipeline by automating the integration, testing, and deployment of code progress, ensuring that the application is updated and reliable.

Modularity:

The application code should be structured into small, independent services, in order to be easier to develop, test and maintain.

Version Control:

The project should use a version control software to help with code management, enabling collaboration and easier code review.

9.5 Security and Safety

Data Protection:

The project complies with the GDPR regulations and has strong user permissions that are essential for data safety.

Regular Updates:

To reduce the risk of security threats, the application should be regularly updated.

Password Encryption:

In case of an unauthorized access, sensible data, specifically users passwords, should be encrypted.

9.6 Operation Concepts

High Availability and Reliability:

PictuRAS is a SaaS application, so it should ensure that it has high uptime and solid data recovery measures.

Scalability:

To maintain performance as the demand increases, the application should be capable to scale horizontally, adding more instances of each microservice, and/or vertically, increasing resources on a specific microservice.

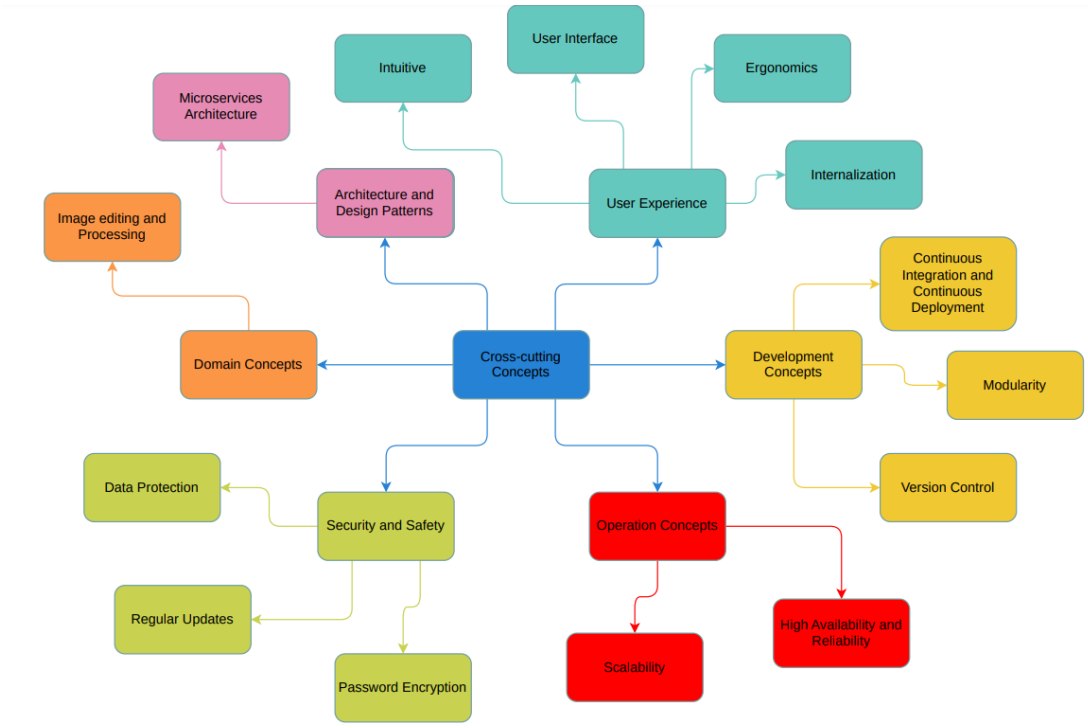


Figure 14: Concepts Map

10 Quality Requirements

In this section, we will present some of the quality requirements of the project, excluding those that were already elaborated on the first chapter (Quality Goals). In contrast to those requirements, these are not as fundamentally necessary, but are still very important for a good user experience and a better functioning application overall.

10.1 Quality Tree

To better organize these requirements, we created the following quality tree, which groups up the quality requirements in five categories or qualities: **Performance**, **Security**, **Usability**, **Scalability and Availability** and **Compatibility**.

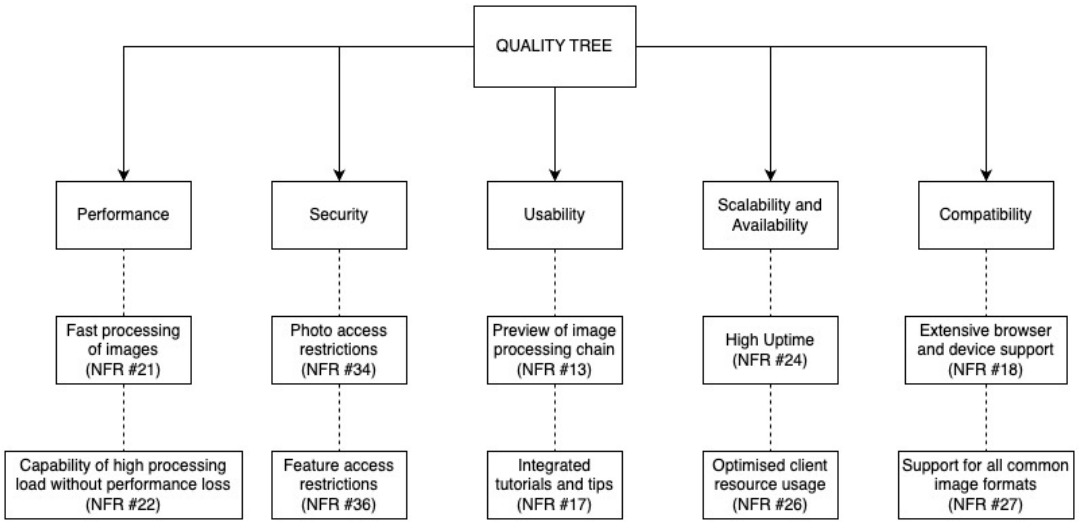


Figure 15: Quality Tree

10.2 Quality Scenarios

For each of the quality requirements present in the tree, we created a corresponding quality scenario, which elaborates on how the program should function in order to meet each requirement. In this section, we will elaborate on each one of these scenarios:

- **Fast processing of images**: The application takes, at max, 3 seconds and 5 seconds to process images with basic and advanced tools, respectively. Most processing tasks only take 1 second and 3 seconds for basic and advanced tools, respectively.
- **Capability of high processing load without performance loss**: The application can handle the simultaneous processing of around 100 images with a lower performance loss than 20
- **Photo access restrictions**: The application does not allow, under any circumstances, the viewing of another users' photo, guaranteeing the privacy of each user.
- **Feature access restrictions**: The application does not allow, under any circumstances, the usage of any feature that the current user does not have access to (for example, a processing filter that is exclusive to paying users).
- **Preview of image processing chain**: The application can show a preview of the result of the current processing chain. This preview is generated in less than 1 second, to allow the user to quickly make any changes they desire.
- **Integrated tutorials and tips**: The application has tutorials for most of its features, allowing any user to use any tool they have access to. The application will also show tips that teach tricks to more experienced users.
- **High Uptime**: The application can only be down (for maintenance) 40 minutes per month, being online and working for the rest of the time.
- **Optimized client resource usage**: The application uses the least client memory it can, letting any type of device use the app without performance issues.
- **Extensive browser and device support**: The application supports most browsers (especially the more popular ones, like Chrome, Safari, Edge, etc.) and also has no issues when being accessed in a common operating system.
- **Support for all common image formats**: The application offers support for the most popular image formats without any issues (.jpg, .png, .gif, etc.).

11 Risks and Technical Debts

11.1 Risks

1. **Risk: MongoDB Scalability and Performance**

   - **Description:** MongoDB is used as the primary database for critical services such as User Auth and User Manager. MongoDB can face performance and scalability issues as the dataset grows or as query complexity increases.
   - **Priority:** High
   - **Impact:** As the application scales, MongoDB may struggle to handle large amounts of concurrent transactions, which could lead to slow response times, data inconsistency, or downtime.
   - **Mitigation Measures:** Implement indexing and sharding for large datasets, monitor database performance, and consider caching frequently accessed data. Regularly review and optimize database queries.

2. **Risk: Persistent Storage Configuration and Data Integrity**

   - **Description:** Persistent Volumes (PVs) are used for storing user, payment, and project data. Any misconfiguration in PVCs can lead to data loss or integrity issues.
   - **Priority:** High
   - **Impact:** Data loss or corruption in Persistent Volumes would impact critical application areas, leading to potential user dissatisfaction, data recovery costs, and loss of service continuity.
   - **Mitigation Measures:** Implement backup and recovery solutions for Persistent Volumes. Regularly monitor storage utilization and ensure proper access controls to prevent accidental data loss.

3. **Risk: Exposure of Web Application via LoadBalancer**

   - **Description:** The web application is exposed using a LoadBalancer service, which improves security and efficiency compared to a NodePort service but still lacks advanced routing and security features.
   - **Priority:** Medium
   - **Impact:** While LoadBalancer provides a dedicated IP for external access, it may not offer the full range of security and routing options needed for complex applications.

- **Mitigation Measures:** Consider implementing an Ingress controller for more granular control over traffic, SSL termination, and enhanced security configurations to further protect the application.

4. **Risk: Microservices Logging and Monitoring**

   - **Description:** With many microservices deployed, consistent logging and monitoring across services may become challenging.
   - **Priority:** High
   - **Impact:** Without centralized logging and monitoring, tracking down issues across multiple services can be time-consuming and lead to prolonged downtimes.
   - **Mitigation Measures:** Implement a centralized logging and monitoring solution (e.g., ELK stack, Prometheus/Grafana). Set up alerts for key metrics to detect and respond to issues quickly.

5. **Risk: Scalability of Web Application in Node.js and React**

   - **Description:** The Node.js and React web application may face scalability challenges under heavy concurrent loads.
   - **Priority:** Medium
   - **Impact:** High traffic could lead to performance degradation, affecting user experience.
   - **Mitigation Measures:** Implement horizontal scaling for the web application in Kubernetes. Use load balancing to distribute traffic evenly, and consider caching mechanisms to reduce the load on the server.

6. **Risk: Security of User Data in MongoDB**

   - **Description:** MongoDB contains sensitive user information managed by the User Auth and User Manager services.
   - **Priority:** High
   - **Impact:** Security vulnerabilities could lead to unauthorized access to sensitive user data, resulting in compliance violations and damage to the organization's reputation.
   - **Mitigation Measures:** Apply strict access control policies, ensure data encryption both in transit and at rest, and conduct regular security audits.

11.2 Technical Debt

1. **Technical Debt: Inter-Service Communication Complexity**

   - **Description:** Communication between services within the Kubernetes cluster relies on **ClusterIP** services. Managing inter-service calls, especially for a high number of services, may become complex and prone to issues.
   - **Priority:** Medium
   - **Impact:** Misconfigurations or failures in service discovery can lead to downtime or degraded performance. Debugging network issues across multiple services could become time-consuming.
   - **Mitigation Measures:** Consider implementing a service mesh (e.g., Istio or Linkerd) to handle service discovery, load balancing, and monitoring, which can streamline inter-service communication and provide observability into traffic flows. Alternatively, if inter-service communication can be asynchronous, use a message broker like RabbitMQ to manage communication between services, which can decouple services and provide reliable messaging.

2. **Technical Debt: Dependency on Kubernetes for Deployment and Orchestration**

   - **Description:** Heavy reliance on Kubernetes for deployment and orchestration introduces a steep learning curve and increases operational complexity.
   - **Priority:** Medium
   - **Impact:** Requires specialized knowledge for maintenance and troubleshooting, potentially leading to slower response times during incidents or outages.
   - **Mitigation Measures:** Ensure team members receive adequate training in Kubernetes. Consider managed Kubernetes solutions that handle some of the complexity, and automate routine maintenance tasks where possible.

3. **Technical Debt: Limited Flexibility in Scaling Microservices**

   - **Description:** Scaling different microservices independently may be challenging due to their interdependencies and the language variability across services.
   - **Priority:** Medium
   - **Impact:** Inefficient scaling could lead to resource waste or insufficient resources for certain services under high demand, impacting performance.

- **Mitigation Measures:** Set up autoscaling policies for each microservice in Kubernetes. Use metrics-based autoscaling to ensure that each service has the resources it needs without over-provisioning.

11.3 Risks and Technical Debt

1. **Risk / Technical Debt: Language Diversity in Microservices**

   - **Description:** Microservices (e.g., Crop Tool, Rotation Tool, Resize Tool, Binarization Tool etc.) will be built in multiple languages, which are currently unknown.
   - **Priority:** High
   - **Impact:** Increased complexity in deployment and maintenance due to differences in libraries, dependencies, and runtime environments. It may lead to inconsistent performance and higher learning curves for developers.
   - **Mitigation Measures:** Establish guidelines for microservice development, including preferred language choices and standardized practices for building, deploying, and monitoring services. Utilize containerization best practices to isolate each service.

12 Glossary

Term	Definition
AI-powered Tools	Advanced image editing tools in PictuRAS that use artificial intelligence for features like object detection and content-based adjustments.
API	Application Programming Interface; a set of rules allowing different software applications to communicate.
Authentication	The process of verifying the identity of a user, typically through credentials such as a username and password.
Auto-scaling	A feature in cloud environments to automatically adjust the number of resources allocated to an application based on current demand.
Backend	The server-side component of an application, responsible for logic, database interactions, and serving client requests.
Containerization	A method of packaging an application and its dependencies into a lightweight, portable container for consistent deployment.
Continuous Integration/ Deployment (CI/CD)	A methodology to automate the integration, testing, and deployment of application updates to ensure reliability and speed.
Docker	A platform for developing, shipping, and running applications using containerization to ensure consistency and scalability.
Elasticity	The ability of the application to scale resources dynamically based on demand to maintain performance.
Frontend	The user-facing part of an application, responsible for the user interface and experience.
GDPR	General Data Protection Regulation; a legal framework ensuring data protection and privacy in the European Union.
Horizontal Scaling	Adding more instances of services to handle increased load without affecting performance.

Kubernetes	An open-source system for automating deployment, scaling, and management of containerized applications.
Load Balancer	A system that distributes incoming network traffic across multiple servers to ensure availability and performance.
Microservices	A software architecture pattern where an application is structured as a collection of small, independent services communicating over APIs.
MongoDB	A NoSQL, document-oriented database, used for data storage in a flexible, schema-less manner.
Node.js	A JavaScript runtime built on Chrome's V8 JavaScript engine, used for building scalable network applications.
NoSQL Database	A type of database designed to handle unstructured or semi-structured data, allowing for high flexibility and scalability.
Persistent Volume (PV)	A Kubernetes resource for managing persistent data storage independent of container lifecycle.
PictuRAS	A web-based image processing application offering tools for image editing, categorized into basic and advanced functionalities.
Project Management	The feature within PictuRAS for organizing, managing, and processing images across multiple projects.
React	A JavaScript library for building user interfaces, known for its component-based architecture.
Regression Testing	A testing practice ensuring that new updates or features do not negatively impact existing functionality.
Tool Chaining	A method where multiple tools are applied in sequence to process images in a project.
Usability	The ease with which users can navigate and interact with the application, measured by metrics such as response times and accessibility.

Bibliography

[1] arc42 template overview. https://arc42.org/overview. Accessed: 2024-11-06.

[2] Mark Richards and Neal Ford. *The Fundamentals of Software Architecture: An Engineering Approach*. O'Reilly Media, Sebastopol, CA, 2020.