janeiro 20, 2025

PictuRAS - Arquitetura do Sistema
Grupo H - PL 3

António Silva (pg57867)

Diogo Marques (pg55931)

João Andrade Rodrigues (pg57879)

João Pedro Rodrigues (pg57880)

Jorge Rodrigues (pg55966)

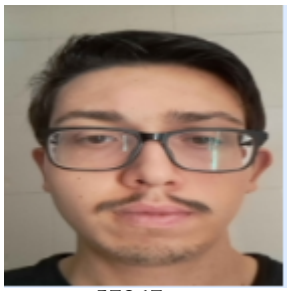
Lara Ferreira (pg57579)

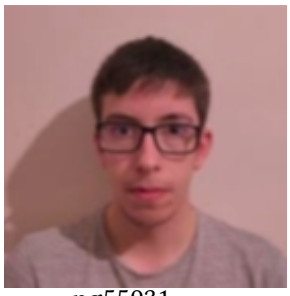
Martim Melo Ferreira (a100653)

Pedro Dantas (pg57599)

Rodrigo Gomes (pg56004)

Vasco Faria (pg57905)


pg57867


pg55931


pg57879


pg57880


pg55966


pg57579


a100653


pg57599


pg56004


pg57905

Índice

1) Introdução 4
2) Alterações Arquiteturais 5
2.1) Backend 5
2.1.1) Gestão Ferramentas 6
2.1.2) Gestão de Imagens 6
2.1.3) Gestão de Projetos 6
2.1.4) WS 6
2.1.5) Message Broker 7
2.1.6) Ferramentas 8
2.2) Frontend 8
3) Microserviços 9
3.1) API Gateway 9
3.1.1) Rotas da API 9
3.2) WS Gateway 10
3.3) WS 10
3.4) Pagamento 10
3.4.1) Rotas da API 10
3.5) Microsserviço de Gestão de Ferramentas 11
3.5.1) Rotas da API 11
3.6) Microsserviço de Gestão de Planos 11
3.6.1) Rotas da API 11
3.7) Microsserviço de Gestão de Utilizadores 12
3.7.1) Rotas da API 12
3.8) Microsserviço de Gestão de Projetos 12
3.8.1) Rotas da API 12
3.9) Ferramentas 12
3.9.1) Autocrop 13
3.9.2) Binarization 13
3.9.3) Border 13
3.9.4) Brightness 13
3.9.5) Contrast 13
3.9.6) Crop 14
3.9.7) OCR 14
3.9.8) Rotate 14
3.9.9) Scale 14
3.9.10) Watermark 15
3.9.11) Object Counter 15
3.9.12) People Counter 15
4) Grau da Solução 16
4.1) Requisitos funcionais 16
4.2) Requisitos Não Funcionais 17
5) Instalação da Implementação 20
6) Contributos Individuais na Implementação 21
7) Conclusão 23

1) Introdução

O projeto *PictuRAS* foi concebido com o objetivo de criar uma solução abrangente e eficiente para a manipulação e edição de imagens digitais, aproveitando ao máximo tecnologias modernas e uma abordagem arquitetural orientada a microsserviços. No cenário atual, onde as demandas por processamento de imagens em grande escala estão em crescimento constante, tanto na esfera comercial quanto na académica, a construção de um sistema robusto, escalável e modular tornou-se uma prioridade para atender às necessidades dos usuários.

Com isso em mente, o desenvolvimento do *PictuRAS* focou em três pilares principais: eficiência técnica, usabilidade e inovação. A arquitetura de microsserviços foi escolhida para garantir que cada funcionalidade pudesse ser desenvolvida, gerenciada e escalada independentemente, reduzindo a complexidade do sistema como um todo e permitindo uma maior flexibilidade na implementação e manutenção. Além disso, tecnologias como *RabbitMQ*, para comunicação assíncrona, *Docker*, para contentorização, e *frameworks* modernos, como Vue.js no *frontend*, foram integradas para garantir que o sistema fosse robusto e eficiente.

O projeto também abordou a necessidade de oferecer ferramentas avançadas de processamento de imagens. Bibliotecas e *frameworks* de destaque, como *FFmpeg* para manipulação de imagens e Detectron2 para detecção de objetos, foram utilizadas para implementar funcionalidades como ajuste de brilho e contraste, e até mesmo reconhecimento de objetos e contagem de pessoas. Para atender aos requisitos não funcionais, o sistema foi projetado para suportar cargas elevadas de processamento, com escalabilidade horizontal e comunicação em tempo real via *WebSockets*, proporcionando uma experiência fluida e responsiva para o usuário.

No entanto, o desenvolvimento do projeto também trouxe desafios significativos. A necessidade de adaptar tecnologias para trabalhar em um ambiente de microsserviços, a integração de múltiplos componentes com diferentes demandas e a garantia de que os requisitos funcionais e não funcionais fossem atendidos exigiram um alto nível de planejamento e colaboração da equipe. As decisões arquiteturais, detalhadas neste documento, refletem o esforço contínuo para equilibrar desempenho, escalabilidade e simplicidade.

Este relatório apresenta as etapas do desenvolvimento, incluindo as alterações arquiteturais realizadas, a implementação de funcionalidades, os desafios enfrentados e os resultados alcançados. Também são destacados os contributos individuais dos membros da equipe, que desempenharam papéis cruciais para o sucesso do projeto. O *PictuRAS* não é apenas uma solução tecnológica; é um reflexo do potencial de inovação e trabalho em equipe para resolver problemas reais de maneira criativa e eficiente.

2) Alterações Arquiteturais

Durante a implementação do projeto foi necessário realizar algumas alterações arquiteturais, quer por não concordarmos com parte da especificação apresentada pela equipa docente, quer por termos descoberto soluções que facilitavam o cumprimento dos requisitos.

Além disso, determinados pontos arquiteturais (*message broker*) não foram devidamente introduzidos pela equipa docente, tanto a nível de especificação como teórico, portanto convém dar um pouco de contexto, a fim de justificar as decisões tomadas.

2.1) Backend

Apesar da lógica para a gestão de planos e utilizadores ter-se mantido fiel à especificação da equipa docente, o mesmo não aconteceu com os restantes componentes, sendo de realçar as mudanças efetuadas sobre a gestão de projetos e o armazenamento de imagens.

Por outro lado, foi um adicionado um componente com o objetivo de modularizar a implementação do código relativo às interações com o *message broker*, este designa-se por *WS* e está inserido dentro da *Gestão de Projetos*.

Não obstante, o *WS* e *Gestão de Projetos* são executado por processos distintos, visto que o primeiro está focado em comunicações assíncronas e procura atingir o máximo da eferência nas comunicações através de *websockets* e *message broker*, enquanto o outro procura única e exclusivamente dar resposta a um servidor aplicacional e servir *endpoints*.

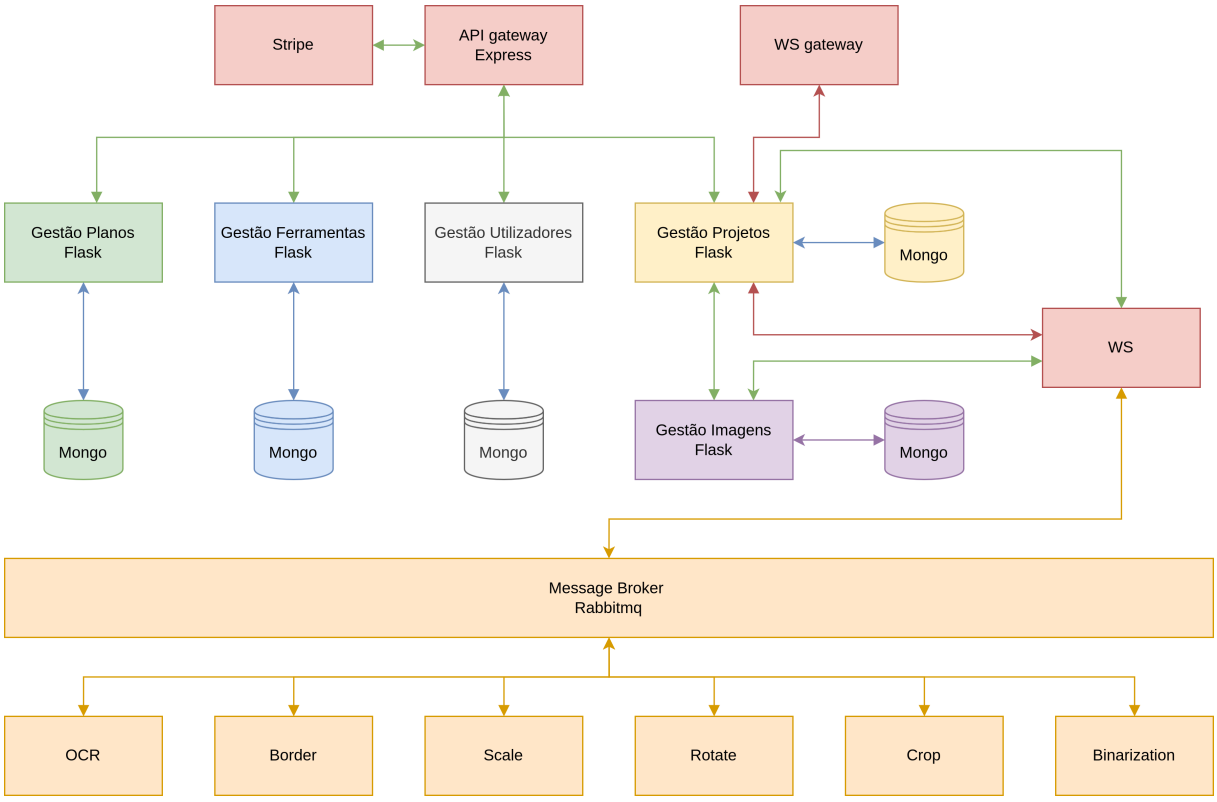


Figura 1: Organização dos componentes de *backend*

- **Azul:** comunicações internas do componente com a base de dados.
- **Laranja:** comunicações efetuadas com a publicações de mensagens no *message broker*.
- **Verde:** comunicações efetuadas através de *endpoints HTTP*.
- **Vermelho:** comunicações efetuadas através de *websockets*.

2.1.1) Gestão Ferramentas

Este microsserviço tem por objetivo fornecer todas as informações acerca das ferramentas disponíveis na *PictuRAS*, especificando os tipos de *input/output*, parâmetros e nível de acesso de cada ferramenta.

Assim sendo, trata-se simplesmente de uma servidor aplicacional que fornece conteúdos estáticos e suporta as operações de *CRUD*, algo que é necessário para atualizar as ferramentas e estender a gama de serviços disponibilizados pela *PictuRAS*, bem como eliminar aqueles que não tenham procura significativa por parte dos clientes.

2.1.2) Gestão de Imagens

Tendo em consideração que o tempo de resposta é um aspeto de elevada importância, optamos por não utilizar o sistema de armazenamento *S3* de *Amazon*, visto que isso iria requerer a execução de mais pedidos *HTTP* e consequentemente iria aumentar a latência sentida pelo cliente.

Nesse sentido os ficheiros pertencentes a um projeto são armazenados localmente da forma mais eficiente possível, ou seja, numa base de dados através do *middleware GridFS*, compatível como o *mongodb*, sendo que este possibilita o armazenamento e recuperação de arquivos que excedem o limite de um documento *BSON* (16 *MB*).

2.1.3) Gestão de Projetos

Na solução original este microsserviço era responsável por interagir com o armazenamento de imagens e as próprias ferramentas de edição ao publicar mensagens no *message broker*, porém isso não faz qualquer sentido e atribui demasiadas responsabilidades ao componente.

Como base neste pensamento, a *Gestão de Projetos* é responsável por gerir as informações dos projetos, tendo para isso uma base de dados própria, além disso também suporta *endpoints* para operações de *CRUD* sobre a *Gestão de Projetos*, visto que este componente não tem ligação com mais nenhum e portanto torna-se necessário criar um intermediário.

Por fim, esta lógica de intermediação também é aplicada para suportar a comunicação entre a *WS Gateway* e o *WS*, sendo que neste caso não se tratam de *endpoints*, mas sim de *webscokets*.

2.1.4) WS

Este componente consiste simplesmente num *ServerSocket* que está eternamente à espera de receber conexões, e a partir do momento em que deteta um pedido de conexão, atribuí um *socket* que será utilizado de modo exclusivo nessa mesma conexão.

O nosso protocolo de comunicação com *websockets* estabelece a existência de quatro tipos de mensagens, sendo que algumas delas são unidirecionais, ou seja, enviadas somente por uma das partes envolvidas.

- **Process (Cliente → WS):** desencadeia o processamento de um projeto, ou seja, a aplicação de todas as ferramentas selecionadas e configuradas sobre as imagens nele contidas, sendo que no final os resultados são guardados na base de dados.
- **Preview (Cliente → WS):** desencadeia o processamento de uma única imagem, sendo que realçar que o resultado não é guardado no sistema de armazenamento.
- **Cancel (Cliente → WS):** permite ao cliente cancelar mensagens de *proceess* ou *preview*, o que obviamente aborta o processamento realizado em *backend*.
- **Progress (Cliente ← WS):** após enviar um *process* ou *preview*, o cliente recebe várias mensagens ao longo do tempo que indicam a percentagem de processamento realizado.

- **Error (Cliente ← WS):** caso ocorra alguma situação inesperada, o cliente é informado à cerca de erros relativos à rotina responsável por satisfazer os seus pedidos no lado do *WS*.

Como forma de melhorar a experiência do utilizador, as imagens totalmente processadas são enviadas de imediato pelo *websocket*, assim o cliente não necessita que esperar que o processo de edição termine totalmente e os resultados sejam armazenados em base de dados.

Por outro lado, o registo das imagens editadas no sistema de armazenamento só acontece no final, visto que isso possibilita ao cliente cancelar o processamento até ao último instante, e além disso elimina os risco de guardar imagens indesejadas.

Numa outra perspetiva, este componente está implementado sobre o padrão de programação assíncrona, permitindo assim servir vários clientes em simultâneo sem que para isso seja necessário suportar o custo de criar *threads*.

Neste caso em concreto a criação de *threads* não faz qualquer sentido, visto que a *workload* de cada rotina consiste essencialmente em publicar e receber mensagens de *message broker*, já para não mencionar que 99% do tempo é gasto em espera passiva.

2.1.5) Message Broker

Embora esta tecnologia nunca tenha sido mencionada nas aulas teóricas, a equipa docente solicitou a sua aplicação e forneceu uma implementação da ferramenta de *watermark*, no entanto essa mesma ferramenta apresenta algumas deficiências, entre elas a impossibilidade de suportar pedidos de vários clientes em simultâneo.

Após a aplicação da *watermark*, os resultados são escritos numa *queue* fixa, consequentemente é impossível garantir que uma determinada imagem chega ao cliente correto, visto que as leituras são efetuadas em *Round-robin*.

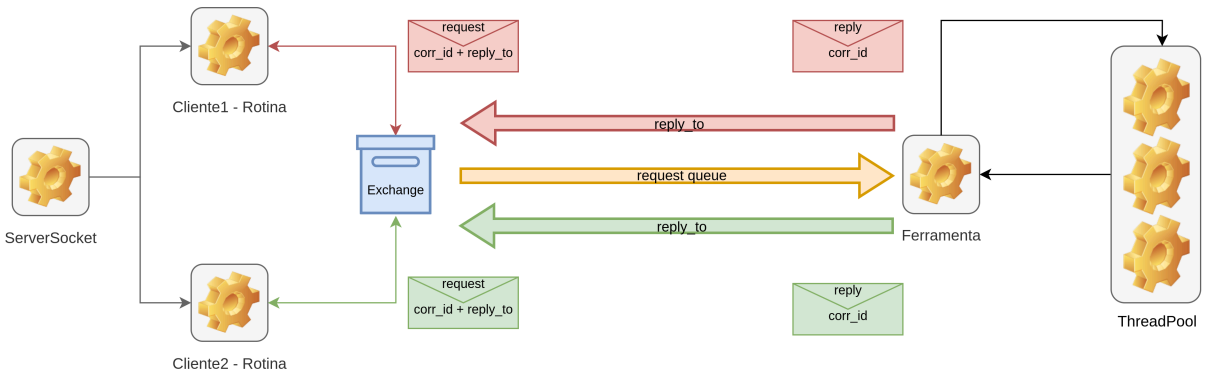


Figura 2: Organização dos componentes de *backend*

Tendo isto em mente, desenvolvemos uma solução onde cada ferramenta tem a sua própria *queue* por onde chegam os pedidos de edição, enquanto os clientes são responsáveis por criar uma *queue* temporária e exclusiva a partir da qual serão recebidas as respostas.

Posto isto, para além dos parâmetros da ferramenta e dados da imagem, um *request* contem um *correlation_id* para relacionar o pedido com a resposta, e ainda um *reply_to* que indica à ferramenta a *routing key* que deve utilizar para publicar os resultados no *exchange*.

Como forma de evitar a criação exagerada de *queues*, cada cliente cria apenas uma *queue* para receber as respostas e indica a *routing key* a todas as ferramentas que contacta, sendo que no final do processo de edição o cliente é responsável por proceder à eliminação da *queue* criada anteriormente.

2.1.6) Ferramentas

O processo de edição pode ser bastante demorado e como tal constitui um gargalo de desempenho, obrigando assim que as rotinas do *WS* aguardem durante demasiado tempo à escuta de mensagens na *queue* de respostas.

Como forma de minimizar este problema, todas as ferramentas possuem uma *thread pool* de tamanho variável, sendo este ajustado através da variável de ambiente *POOL_SIZE*, consequentemente é possível processar diversas imagens em paralelo e aumentar o desempenho do sistema.

Por outro lado, utilizar uma *thread pool* torna-se mais eficiente do que uma *thread* por pedido, visto que favorece a reutilização de recursos e evita a saturação do componente em momentos com maior frequência de pedidos.

2.2) Frontend

Na camada de apresentação, seguimos com bastante fidelidade à arquitetura inicial. A implementação manteve-se fiel ao *Vue.js*, para a *web*, com recurso ao *Vite* para a construção e produção da aplicação. Desta forma, de modo geral, consideramos que a única alteração relevante foi a distinção entre a *Landing Page* e a Página dos Projetos do Utilizador, pois acreditamos que faça mais sentido introduzir a aplicação ao utilizador, para que se possa sentir mais acompanhado na sua utilização.

Naturalmente, na arquitetura proposta, a modularidade e reutilização não foram consideradas, sendo que apenas foram especificadas as páginas e não os componentes. Desta forma, na nossa implementação, as páginas encontram-se decompostas em componentes, que podem ser reutilizados por diversas páginas de uma forma muito cómoda, fortalecendo a modularidade da aplicação. Na mesma linha de pensamento, recorremos ao uso de *stores*, pois a gestão de estado para a nossa implementação é complexa, daí a necessidade recorrer a essas bibliotecas.

Relativamte à comunicação com outros microsserviços, apenas é feita com a API Gateway, como especificado na arquitetura, e também com a WS Gateway, para possibilitar a receção assíncrona das imagens modificadas na aplicação cliente.

Assim, essencialmente, podemos dividir a estrutura do frontend em Views, que representam as diferentes páginas, Components, os componentes que constituem as páginas, Stores, onde ocorre a gestão de estado, Router, onde o *routing* das páginas é especificado, Assets, onde os ficheiros estáticos como vídeos e imagens se encontram, Nginx, onde a configuração utilizada para o deployment da componente é feita e, finalmente, a "App.vue", ficheiro principal da aplicação.

3) Microserviços

3.1) API Gateway

A API Gateway é a componente de software que atua como um ponto de entrada centralizado para a comunicação entre clientes e microsserviços, sendo que atua como intermediária, na medida em que recebe os pedidos dos clientes, redirecionando-os para os serviços apropriados e retornando as respostas ao mesmo. É também neste microsserviço que é realizada a autenticação do cliente, bem como a criação do conceito de sessão, tanto para utilizadores anónimos como registados, recorrendo a bibliotecas como *passport* e *express-session* e a uma base de dados de utilizadores.

3.1.1) Rotas da API

`POST /register` → regista um novo utilizador. Cria o utilizador no sistema local e no microsserviço de utilizador.

`POST /login` → autentica o utilizador usando o *passport*. Atualiza os dados do último acesso do utilizador.

`POST /logout` → encerra a sessão do utilizador autenticado.

`GET /profile` → obtém as informações relativas a um utilizador do microsserviço respetivo.

`GET /user/status` → verifica se um utilizador é registado ou anónimo, recorrendo ao *passport*.

`GET /tools` → obtém a lista de ferramentas do microsserviço das ferramentas.

`GET /projects` → obtém os projetos de um utilizador, através de um pedido ao microsserviço de projetos.

`POST /projects` → cria um novo projeto, através de um `POST` ao microsserviço de projetos.

`POST /projects/images` → associa imagens a um projeto de um utilizador, através de um pedido `POST` ao microsserviço de projetos.

`GET /projects/images/:id` → obtém as imagens associadas a um projeto, através de um pedido `GET` ao microsserviço de projetos.

`PUT /projects` → efetua alterações às informações de um projeto, através de um pedido `PUT` ao microsserviço de projetos.

`DELETE /projects/images` → elimina imagens associadas a um projeto, através de um pedido `DELETE` ao microsserviço de projetos.

`DELETE /projects/:id` → elimina um projeto de um utilizador, através de um pedido `DELETE` ao microsserviço de projetos.

`PUT /user/plan` → altera o plano de um utilizador, recorrendo a um pedido `PUT` ao microsserviço de utilizadores.

`GET /plans` → obtém a lista de planos disponíveis, através de um pedido `GET` ao microsserviço de planos.

`GET /plan/:planId` → obtém as informações de um plano identificado por *planId*, através de um pedido `GET` ao microsserviço de planos.

`GET /users/:email` → obtém informações de um utilizador através do seu *email*, recorrendo a um pedido `GET` ao microsserviço de utilizadores.

`DELETE /users/:email` → apaga as informações de um utilizador, através de um pedido DELETE ao microsserviço de utilizadores.

`PUT /users/:email` → altera o *email* de um utilizador, através de um pedido PUT ao microsserviço de utilizadores.

3.2) WS Gateway

Este componente é essencialmente consistido por um *ServerSocker* que está constantemente à espera de estabelecer conexão com os clientes, a partir do momento em que é detetado o pedido de um cliente, é criado um *websocket* para estabelecer ligação ao *WS*.

Assim sendo, este intermediário possui dois *websocket*, um conectado ao cliente e outro ao *WS*, sendo que a sua única função é receber mensagens de um *websocket* e enviar pelo outro, no final o cliente procede à desconexão com a *WS Gateway*, e esta faz o mesmo com o *WS*.

3.3) WS

O *WS* é o componente responsável por gerir a comunicação em tempo real entre os clientes e o *backend*, utilizando *WebSockets* para estabelecer conexões bidirecionais de forma eficiente. Este microsserviço centraliza o envio de mensagens dos clientes para o sistema de encaminhamento RabbitMQ e a receção das mesmas, garantindo uma integração fluida com o ecossistema de microsserviços.

O *WS* desempenha as seguintes funções principais:

- Encaminhamento de Mensagens: recebe mensagens dos clientes, analisa-as e encaminha-as para o *RabbitMQ*, mantendo o desacoplamento entre clientes e serviços backend.
- Processamento de Projetos: coordena tarefas complexas, como o processamento de imagens associadas a projetos, enviando atualizações periódicas de progresso aos clientes e retornando os resultados finais após a conclusão das operações.
- Subscrição em Tópicos de Interesse: Utiliza o *RabbitMQ* para subscrever tópicos relevantes e processar notificações ou eventos associados ao estado dos projetos.
- Gestão de Sessões: Através da componente `Tracer`, rastreia o estado das operações, permitindo ações como o cancelamento de tarefas ou o acompanhamento do progresso.

A arquitetura do *WS Gateway* combina um servidor *WebSocket* com integração robusta do *RabbitMQ*, assegurando comunicação escalável e eficiente, enquanto promove a modularidade e a extensibilidade, essenciais para sistemas distribuídos e dinâmicos.

3.4) Pagamento

Para efetuar o processo de pagamento será utilizada a ferramenta Stripe. Esta integração assegura que os pagamentos sejam processados de forma segura e eficiente, utilizando funcionalidades como Payment Intents e webhooks.

3.4.1) Rotas da API

`POST /create-payment-intent` → esta rota é usada para criar um Payment Intent (intenção de pagamento). O Payment Intent é um objeto controlado pela Stripe que representa o estado de uma tentativa de pagamento, garantindo que os detalhes sejam processados corretamente no fluxo de checkout.

No backend, a implementação utiliza o serviço stripe para criar um Payment Intent. O método createPaymentIntent recebe o valor do pagamento e retorna o objeto do Payment Intent com as seguintes propriedades:

- **amount:** valor total da transação.
- **currency:** moeda utilizada na transação (definida na configuração do Stripe).
- **payment_method_types:** métodos de pagamento aceitos (também definidos na configuração do Stripe).

`POST/webhook` → esta rota é usada para lidar com os webhooks da Stripe.

Webhooks são notificações enviadas automaticamente pela Stripe ao servidor para informar eventos importantes relacionados ao processamento de pagamentos. Entre os eventos gerenciados estão:

- Confirmação de pagamento bem-sucedido.
- Falha em transações.
- Processamento de reembolsos.
- Atualizações em assinaturas.

No backend, o método constructWebhookEvent do serviço da stripe é utilizado para validar e construir o evento recebido do webhook, garantindo que a origem seja confiável (verificada por meio do webhookSecret). Caso a validação falhe, é gerado um erro para evitar processamentos indesejados.

Após a confirmação de um pagamento bem-sucedido, será utilizada a rota PUT /user/plan para atribuir ao utilizador o estatuto correspondente ao plano adquirido. Isso garante que os benefícios relacionados ao plano sejam ativados automaticamente após o pagamento.

3.5) Microsserviço de Gestão de Ferramentas

Este componente é responsável por servir conteúdo estático no formato *JSON* acerca das informações das ferramentas, apresentando dados como os parâmetros necessários para configura a ferramenta, os tipos de *input/output*, e o nível de acesso (ser *premium*).

3.5.1) Rotas da API

- `GET /tools` → devolve as informações das ferramentas contidas no sistema de armazenamento.
- `GET /tools/:toolid` → devolve as informações da ferramenta identificada por *toolid*.
- `POST /tools` → regista uma ferramenta no sistema de armazenamento.
- `PUT /tools/:toolid` → altera o registo da ferramenta identificada por *toolid*.
- `DELETE /tools/:toolId` → elimina o registo da ferramenta identificada por *toolid*.

3.6) Microsserviço de Gestão de Planos

Este componente é responsável por gerir as informações dos planos de subscrição, o que contém uma tabela de preços e os períodos de pagamento, tais como diário, mensal e anual.

3.6.1) Rotas da API

`GET /plans` → devolve as informações dos planos disponibilizados pelo sistema.

`GET /plans/:planid` → devolve as informações do plano identificado por *planid*.

`POST /plans` → adiciona o registo de um plano ao sistema de armazenamento.

`PUT /plans/:planid` → atualiza o registo do plano identificado por *planid*.

`DELETE /plans/:planid` → elimina as informações do plano identificado por *planid*.

3.7) Microsserviço de Gestão de Utilizadores

O microsserviço de gestão de utilizadores guardar as informações relativas aos indivíduos registados no sistema, ou seja, os utilizadores anónimos não são abrangidos por este serviço.

3.7.1) Rotas da API

`GET /users` → devolve as informações dos utilizadores registados no sistema.

`GET /users/:username` → devolve as informações do utilizador identificado por *username*.

`POST /users` → adiciona o registo de um utilizador ao sistema de armazenamento.

`PUT /user/:username` → atualiza o registo do utilizador identificado por *username*.

`DELETE /user/:username` → elimina as informações do utilizador identificado por *username*.

3.8) Microsserviço de Gestão de Projetos

O microsserviço de gestão de projetos é o mais complexo. Este é constituído por dois elementos. Uma base de dados que guarda as informações relativas aos projetos *Load balancer* que gere as edições de imagens após o desencadeamento do processamento de um projeto

3.8.1) Rotas da API

Tendo em consideração que o microsserviço de *Gestão de Imagens* não está ao alcance dos restantes, este componente apresenta *endpoints* que permitem uma intermediação, assim sendo existem duas rotas principais, uma direcionada para os projetos, e outra que serve os propósitos das imagens.

3.8.1.1) Rotas de projetos

`GET /projects` → devolve a lista de todos os projetos registados no sistema de armazenamento.

`GET /projects/:projectid` → devolve as informações do projeto identificado por *projectid*.

`GET /projects/owner/:username` → devolve a lista dos projetos pertencente ao utilizador *username*.

`POST /projects` → cria um novo projeto, sendo o proprietário um utilizador autenticado ou anônimo.

`PUT /projects/:projectid` → atualiza um projeto existente.

`DELETE /projects/:projectid` → remove um projeto (somente acessível a utilizadores autenticados).

3.8.1.2) Rotas de imagens de projetos

`GET /projects/images/:projectid` → retorna uma lista de URIs das imagens associadas a um projeto.

`GET /projects/images/data/:id` → retorna os dados binários de uma imagem identificada por *id*.

`POST /projects/images/:projectid` → faz upload de uma imagem para um projeto específico.

`DELETE /projects/images/:id` → remove a imagem identificada por *id* do sistema de armazenamento.

3.9) Ferramentas

Nas ferramenta, o servidor recebe um objeto do tipo *Tool_* Message_Request, que contém os dados da imagem em base64. Através da *Tool_*Message_Reply encapsula a imagem processada ou o resultado da ferramenta aplicada. Há também um *Tool_*worker que é uma aplicação que consome mensagens de uma fila RabbitMQ, aplica uma ferramenta as imagens e envia as respostas de volta.

3.9.1) Autocrop

Esta ferramenta aplica um processo de remoção de fundo em uma imagem fornecida no formato base64, utilizando a biblioteca **rembg**. Ela retorna a imagem processada, também codificada em base64, juntamente com seu tipo MIME.

Bibliotecas utilizadas:

- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Codifica e decodifica dados no formato base64.
- ***rembg:*** (ferramenta externa): Biblioteca usada para remoção de fundo em imagens.

3.9.2) Binarization

Esta ferramenta converte uma imagem fornecida em formato base64 para escala de cinza (binarização) usando o FFmpeg. Ela retorna a imagem processada, também em base64, junto com seu tipo MIME.

Bibliotecas utilizadas:

- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Codifica e decodifica dados no formato base64.
- ***subprocess:*** Executa o comando FFmpeg para aplicar o filtro de escala de cinza.
- ***FFmpeg:*** (ferramenta externa): Software de linha de comando usado para manipulação de arquivos multimídia, como conversão e edição de vídeos/imagens.

3.9.3) Border

A ferramenta *Border* adiciona uma borda personalizada a uma imagem fornecida em formato base64. A largura, altura e cor da borda são especificadas no pedido. A imagem com a borda aplicada é retornada também em base64, acompanhada do tipo MIME.

Bibliotecas utilizadas:

- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Codifica e decodifica dados no formato base64.
- ***subprocess:*** Executa o comando FFmpeg para aplicar a borda.
- ***FFmpeg:*** Software de linha de comando usado para editar e manipular arquivos multimídia.

3.9.4) Brightness

A ferramenta Brightness ajusta o brilho de uma imagem fornecida em formato base64. O nível de brilho é especificado na solicitação. Após o processamento, a imagem ajustada é retornada em base64, junto com seu tipo MIME.

Bibliotecas utilizadas:

- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Codifica e decodifica dados no formato base64.
- ***subprocess:*** Executa o comando FFmpeg para aplicar o ajuste de brilho.
- ***FFmpeg:*** (ferramenta externa): Software de linha de comando usado para manipular multimídia, neste caso ajustando o brilho.

3.9.5) Contrast

A ferramenta *Contrast* ajusta o contraste de uma imagem fornecida em formato base64. O nível de contraste é especificado na solicitação. Após o processamento, a imagem ajustada é retornada em base64, acompanhada do tipo MIME.

Bibliotecas utilizadas:

- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Codifica e decodifica dados no formato base64.
- ***subprocess:*** Executa o comando FFmpeg para aplicar o ajuste de contraste.
- ***FFmpeg:*** Software de linha de comando utilizado para manipulação de arquivos multimídia, neste caso para ajustar o contraste.

3.9.6) Crop

A ferramenta Crop realiza o recorte de uma imagem fornecida em formato base64. A largura, altura e as coordenadas do canto superior esquerdo do recorte são especificadas na solicitação. Após o processamento, a imagem recortada é retornada em base64, junto com seu tipo MIME.

Bibliotecas utilizadas:

- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Codifica e decodifica dados no formato base64.
- ***subprocess:*** Executa o comando FFmpeg para aplicar o recorte.
- ***FFmpeg:*** (ferramenta externa): Software de linha de comando usado para edição multimídia, neste caso para recortar a imagem.

3.9.7) OCR

A ferramenta OCR realiza o reconhecimento de texto em uma imagem fornecida em formato base64 utilizando o Tesseract OCR. O texto extraído é retornado como uma string codificada em base64, acompanhado de um tipo MIME que indica texto simples.

Bibliotecas utilizadas:

- ***base64:*** Decodifica a imagem de entrada e codifica o texto extraído.
- ***subprocess:*** Executa o comando Tesseract OCR para realizar a extração de texto.
- ***Tesseract OCR:*** (ferramenta externa): Software de OCR utilizado para identificar texto em imagens.

3.9.8) Rotate

A ferramenta *Rotate* aplica uma rotação em uma imagem fornecida em formato base64. O ângulo de rotação é especificado na solicitação e convertido de graus para radianos antes de ser aplicado. Após o processamento, a imagem rotacionada é retornada em base64, acompanhada de seu tipo MIME.

Bibliotecas utilizadas:

- ***math:*** Converte o ângulo de rotação de graus para radianos.
- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Decodifica a imagem de entrada e codifica a imagem rotacionada.
- ***subprocess:*** Executa o comando FFmpeg para aplicar a rotação.
- ***FFmpeg:*** (ferramenta externa): Software de linha de comando usado para manipular multimídia, neste caso para rotacionar a imagem.

3.9.9) Scale

A ferramenta Scale redimensiona uma imagem fornecida em formato base64 para as dimensões especificadas (largura e altura). Após o processamento, a imagem redimensionada é retornada em base64, acompanhada de seu tipo MIME.

Bibliotecas utilizadas:

- ***magic:*** Determina o tipo MIME do arquivo resultante.
- ***base64:*** Decodifica a imagem de entrada e codifica a imagem redimensionada.

- ***subprocess:*** Executa o comando FFmpeg para aplicar o redimensionamento.
- ***FFmpeg:*** (ferramenta externa): Software de linha de comando utilizado para manipulação de multimídia, neste caso para redimensionar a imagem.

3.9.10) Watermark

A ferramenta Watermark aplica uma marca d'água em uma imagem fornecida em formato base64. A marca d'água é uma imagem adicional (que pode ser configurada via caminho de arquivo) que será aplicada na imagem de entrada, com a opacidade ajustada. A marca d'água é posicionada aleatoriamente dentro da imagem e redimensionada conforme o tamanho da imagem original. Após o processamento, a imagem com a marca d'água é retornada em base64, acompanhada de seu tipo MIME.

Bibliotecas utilizadas:

- ***magic*** : Determina o tipo MIME do arquivo resultante.
- ***base64:*** Codifica e decodifica dados no formato base64.
- ***random:*** Gera posições aleatórias para a marca d'água na imagem.
- ***io:*** Manipula os fluxos de entrada e saída de dados, necessário para a conversão da imagem.
- ***PIL:*** Biblioteca para manipulação de imagens.
- ***Image:*** Carrega, manipula e guarda imagens.
- ***ImageEnhance:*** Ajusta a opacidade da marca d'água.

3.9.11) Object Counter

Esta ferramenta realiza a contagem de objetos em uma imagem fornecida no formato base64, utilizando o Detectron2 para detecção e classificação. Ela retorna a contagem de objetos identificados, codificada em base64, junto com seu tipo MIME.

Bibliotecas utilizadas:

- ***base64:*** Decodifica a imagem de entrada e codifica a resposta no formato base64.
- ***numpy:*** Converte os dados da imagem decodificada em um array processável.
- ***OpenCV:*** Decodifica o array de imagem para um formato utilizável.
- ***Detectron2:*** Realiza a inferência para detecção e classificação de objetos na imagem.
- ***Counter:*** Conta a ocorrência de cada classe de objeto detectada.

3.9.12) People Counter

Infelizmente não fizemos *deployment* do *People Counter*, dado a imagem *docker* ser extremamente pesada e requerer a utilização de *CUDA*, no entanto a ferramenta *Object Counter* também devolve o número de pessoas, logo caso alguém pretenda contar pessoas pode simplesmente utilizar essa ferramenta como alternativa.

4) Grau da Solução

Nesta secção indicamos quais os requisitos funcionais e não funcionais que foram possíveis de ser cumpridos na solução de modo a avaliar o grau de completude do produto final.

4.1) Requisitos funcionais

Nesta subsecção, apresentamos os requisitos funcionais definidos para a solução, avaliando a sua implementação, cobertura e que microsserviços estão associados. Para esse efeito, recorremos ás seguintes tabelas.

Requisito	Descrição	Cobertura	Microsserviço
RF1	O utilizador autentica-se utilizando as suas credenciais.	Total	API
RF2	O utilizador anónimo regista-se	Total	API e Gestão de Utilizadores
RF3/4	O utilizador escolhe o plano de subscrição	Total	Gestão de Planos e Stripe
RF6	O utilizador cria um projeto	Total	Gestão de Projetos
RF7	O utilizador lista os seus projetos	Total	Gestão de Projetos
RF8	O utilizador acede à área de edição de um projeto	Total	Gestão de Projetos, Gestão de Ferramentas e Gestão de Imagens
RF9	O utilizador carrega imagens para um projeto.	Total	Gestão de Imagens
RF11	O utilizador adiciona uma ferramenta de edição ao projeto.	Total	Gestão de Projetos
RF13	O utilizador transfere o resultado de um projeto para o dispositivo local	Total	Gestão de Imagens
RF15	O utilizador altera a ordem das ferramentas de um projeto.	Total	Gestão de Projetos
RF16	O utilizador altera os parâmetros das ferramentas	Total	Gestão de Projetos
RF17	O utilizador cancela o processamento de um projeto durante a sua execução	Total	WS
RF18	O utilizador registado acede às suas informações de perfil.	Total	Gestão de Utilizadores
RF19	O utilizador edita o seu perfil.	Total	Gestão de Utilizadores
RF21	O utilizador premium cancela a sua subscrição premium.	Total	Gestão de Planos e Stripe
RF22	O utilizador registado termina a sua sessão.	Total	API

Tabela 1: Cobertura dos Requisitos funcionais

Requisito	Descrição	Cobertura	Microsserviço
RF25	O utilizador recorta manualmente imagens.	Total	Ferramentas e Gestão de Projetos
RF26	O utilizador escala imagens para dimensões específicas.	Total	Ferramentas e Gestão de Projetos
RF27	O utilizador adiciona borda a imagens.	Total	Ferramentas e Gestão de Projetos
RF29	O utilizador ajusta o brilho a imagens.	Total	Ferramentas e Gestão de Projetos
RF30	O utilizador ajusta o contraste a imagens.	Total	Ferramentas e Gestão de Projetos
RF32	O utilizador roda imagens.	Total	Ferramentas e Gestão de Projetos
RF35	O utilizador aplica um algoritmo de recorte automático de imagens com base no seu conteúdo.	Total	Ferramentas e Gestão de Projetos
RF36	O utilizador extrai texto de imagens	Total	Ferramentas e Gestão de Projetos
RF37	O utilizador aplica um algoritmo de reconhecimento de objetos em imagens.	Total	Ferramentas e Gestão de Projetos
RF38	O utilizador aplica um algoritmo de contagem de pessoas em imagens.	Parcial	Ferramentas e Gestão de Projetos

Tabela 2: Cobertura dos Requisitos funcionais

Das tabelas 1 e 2, consideramos que 25 em 26 requisitos foram implementados na totalidade, com a excessão do requisito funcional "RF38". Sobre esta ferramenta, infelizmente, não conseguimos testar própriamente devido às necessidades de hardware da biblioteca encontrada. Contudo, o código foi corretamente adapatado à solução, assim como dockerizado, apenas foi excluido do *deployment* final da solução. Além disso, a ferramenta que conta objetos, é também capaz de contar o número de pesosas.

4.2) Requisitos Não Funcionais

Para os requisitos não funcionais, avaliamos a nossa solução, destacando a sua cobertura, para compreender se a implementação garante as características de qualidade.

Requisito	Descrição	Cobertura	Microsserviço
RNF5	A página de um projeto distingue visualmente entre as ferramentas básicas e avançadas.	Total	Frontend
RNF7	O utilizador cria um projeto implicitamente ao arrastar ficheiros para o dashboard da aplicação.	Falha	Frontend
RNF8	O utilizador carrega imagens arquivadas num único ficheiro .zip.	Total	Frontend
RNF9	O utilizador é mantido informado acerca do estado de processamento de um projeto em tempo real.	Total	Frontend
RNF14	A visualização de imagens grandes ou pequenas é auxiliada pelo utilitário zoom.	Falha	Frontend
RNF15	A visualização de imagens permite que se navegue em todas as duas direções arrastando o rato.	Falha	Frontend
RNF18	A aplicação é compatível com diferentes plataformas e browsers, incluindo de dispositivos móveis e desktop.	Parcial	Frontend
RNF19	A aplicação fornece feedback visual claro e imediato ao utilizador em caso de erro ou falha durante um procedimento demorado.	Total	Frontend
RNF22	O sistema deve processar até 100 imagens ao mesmo tempo, sem quebras percetíveis no desempenho.	Falha	Ferramentas e Bus
RNF23	A aplicação deve ser capaz de escalar horizontalmente de forma elástica para suportar o aumento de utilizadores e volume de processamentos, mantendo o desempenho mas também os custos controlados.	Total	Aplicação
RNF25	A aplicação deve ser integrável com outras plataformas e serviços de terceiros.	Total	Aplicação
RNF28	A aplicação deve ser facilmente estendida com novas ferramentas de edição.	Total	Aplicação
RNF32	A aplicação deve ser facilmente estendida com novas ferramentas de edição.	Total	Aplicação
RNF33	A aplicação deve realizar backups automáticos dos dados e imagens dos utilizadores.	Falha	Aplicação

Tabela 3: Cobertura dos Requisitos não funcionais

Face ao panorama observado nos requisitos funcionais, deparamos uma situação oposta, na medida em que diversos aspetos apenas foram parcialmente implementados, ou falham.

A nível da camada de apresentação da aplicação, alguns aspetos foram esquecidos dado que o desenvolvimento foi, na grande maioria, feito em *displays* maiores, como portáteis ou *desktops*. Desta forma, a necessidade de fazer *zoom* e navegar pelas imagens, descritas nos *RNF14* e *RNF15*, ficou para o fim do desenvolvimento, onde infelizmente faltou tempo. O mesmo pode ser dito para a adaptação para *displays* pequenos, escrita no *RNF18*, apesar de ser parcial, visto que testamos em diversos *Browsers*, como o Brave, Google Chrome, Opera, Firefox, Edge e Safari.

Relativamente aos requisitos *RNF22* e *RNF33*, novamente a falta de tempo impediu que alguns procedimentos como o *Backup* dos dados não fosse corretamente implementado. No caso do requisito *RNF22*, existem vários fatores que devem ser considerados e necessários para responder a este requisito. Um deles é relativo à capacidade do *hardware* que deve ser capaz de suportar os recursos necessários para processar as 100 imagens em simultâneo. Apesar de acharmos que o nosso desenho foi um sucesso, é difícil de realizar um *benchmark* que possa ditar definitivamente se o requisito foi ou não cumprido.

5) Instalação da Implementação

Como objetivo do projeto, resta configurar a instalação da aplicação na máquina virtual disponibilizada pela equipa docente. Sobre esta máquina, sabemos que existe uma única porta à escuta, a porta 80, e também que a gestão de certificados é feita pela equipa docente. Posto, isto, restou ao grupo optar por utilizar uma estratégia, preferencialmente de contentorização, que permitisse uma instalação eficiente.

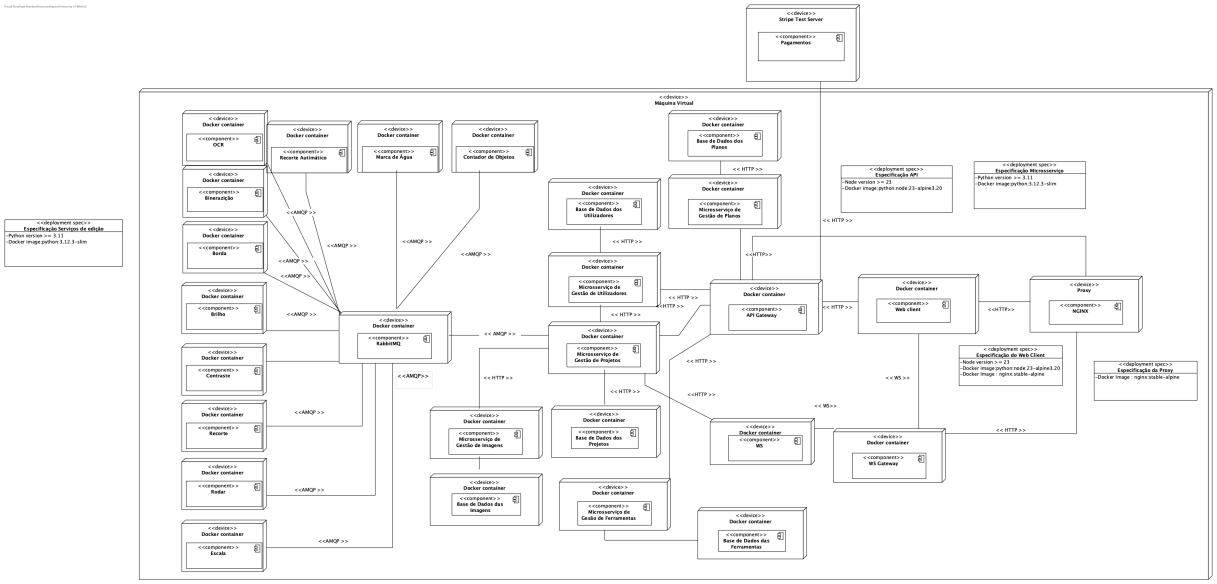


Figura 3: Instalação na Máquina Virtual

Como representado na figura 3, cada microsserviço fica encapsulado no respetivo contentor, e comunicam através da rede virtual montada pela ferramenta Docker. Para orquestração, optamos pelo *docker compose*, que apesar de não ser tão eficiente quanto o Kubernetes, para deployments num único nó, é suficiente. Outras opções consideradas, na base do Kubernetes, envolviam o uso do *minikube*, que simula um cluster Kubernetes num único nó.

Um cliente, observa todos os seus pedidos a serem reencaminhados pelo *nginx*, de forma a que tanto o cliente *web* e a Api estejam públicas.

6) Contributos Individuais na Implementação

Elemento da implementação	Número	Contributos
API Gateway	pg57880	Endpoints relacionados ao perfil, pagamentos e planos
API Gateway	pg56004	Autenticação geral
API Gateway	pg55966	Endpoints relacionados aos projetos e imagens
API Gateway	pg57905	Endpoints relacionados aos projetos

Tabela 4: Contributos individuais da *API Gateway*

Elemento da implementação	Número	Contributos
WS Gateway	a100653	Geral

Tabela 5: Contributos individuais da *WS Gateway*

Elemento da implementação	Número	Contributos
Autenticação	pg57905	Frontend de login e registo

Tabela 6: Contributos individuais da Autenticação

Elemento da implementação	Número	Contributos
Pagamentos	pg57880	Funcionamento backend da stripe
Pagamentos	pg57880	Frontend dos pagamentos

Tabela 7: Contributos individuais dos Pagamentos

Elemento da implementação	Número	Contributos
Gestão de Utilizadores	pg55931	Backend geral
Gestão de Utilizadores	pg57879	Frontend de perfil

Tabela 8: Contributos individuais da Gestão de Utilizadores

Elemento da implementação	Número	Contributos
Gestão de Planos	pg55931	Backend do microsserviço
Gestão de Planos	pg56004	Frontend dos planos

Tabela 9: Contributos individuais da Gestão de Planos

Elemento da implementação	Número	Contributos
Gestão de Projetos	pg55931	Backend do microsserviço
Gestão de Projetos	pg57905	Frontend da página de projetos
Gestão de Projetos	pg55966	Frontend da página de projeto

Tabela 10: Contributos individuais da Gestão de Projetos

Elemento da implementação	Número	Contributos
Ferramentas	pg57867	Ferramentas de IA
Ferramentas	pg55931	Ferramenta de border
Ferramentas	pg57879	Ferramenta de Binarization
Ferramentas	pg57880	Ferramenta de Brightness
Ferramentas	pg55966	Ferramenta de Contrast
Ferramentas	pg57579	Ferramenta de Crop
Ferramentas	a100653	Ferramenta de OCR
Ferramentas	pg57599	Ferramenta de Rotate
Ferramentas	pg56004	Ferramenta de Scale
Ferramentas	pg57905	Ferramenta de Watermark

Tabela 11: Contributos individuais das Ferramentas

Elemento da implementação	Número	Contributos
Docker	pg55966	Frontend e API
Docker	pg55931	Backend
Docker	pg57867	Ferramentas

Tabela 12: Contributos individuais do *Docker*

Elemento da implementação	Número	Contributos
Deployment	pg55966	Deployment geral

Tabela 13: Contributos individuais do *Deployment*

7) Conclusão

Concluímos que o projeto desenvolvido representa um trabalho bem-sucedido, cumprindo de forma satisfatória os requisitos inicialmente propostos. Ao longo do processo, esforçamo-nos por garantir que cada funcionalidade essencial fosse implementada de forma eficaz, assegurando um resultado sólido e funcional.

Embora os objetivos principais tenham sido atingidos, reconhecemos que há margem para melhorias no futuro. Em particular, o frontend poderia ser aperfeiçoado, tanto ao nível do design como da usabilidade, de forma a proporcionar uma experiência de utilizador mais intuitiva e apelativa.

Para além disso, poderiam ser exploradas novas funcionalidades que acrescentassem ainda mais valor ao projeto e ampliassem o seu potencial de aplicação.

De forma geral, estamos satisfeitos com o trabalho realizado. Acreditamos que o projeto reflete o nosso empenho e capacidade de desenvolver soluções práticas e eficazes, demonstrando o nosso progresso ao longo da sua execução.