Universidade do Minho
Escola de Engenharia
Mestrado em Engenharia Informática

Requisitos e Arquiteturas de Software

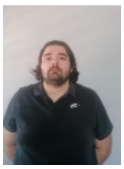
Ano Letivo de 2024/2025

PictuRAS

Eduardo Cunha
PG55939

Rodrigo Ralha
PG56005

Délio Alves
a94557

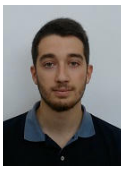


Rafael Peixoto
PG55998

Rui Gonçalves
PG56011

João Coelho
PG55954

Flávio Silva
PG57539



Filipe Oliveira
a80330

November 22, 2024

RAS

Índice

Lista de Figuras

1. Introdução e Objetivos

Este documento foi concebido no âmbito de uma proposta para desenvolver o sistema *PictuRAS*, que visa oferecer uma plataforma de processamento e edição de imagens, acessível a diversos tipos de utilizadores. A motivação principal para a criação do *PictuRAS* reside na necessidade de uma ferramenta intuitiva e versátil que permita tanto a amadores como a profissionais explorar um vasto conjunto de funcionalidades de edição de imagem de forma eficiente e acessível, suportada por uma arquitetura escalável e moderna na *cloud*.

Nele, descrevem-se diversas etapas relacionadas com o planeamento da implementação do sistema. Inicialmente, abordamos o conjunto de restrições impostas à equipa de desenvolvimento, englobando limitações orçamentais, temporais e relacionadas com a implementação da solução. Adicionalmente, destacamos os benefícios esperados do sistema, incluindo a capacidade de processamento em lote e a extensibilidade para incorporar novas ferramentas.

Seguidamente, procedemos à análise dos requisitos não funcionais apresentados no documento, selecionando aqueles com maior impacto na escolha da arquitetura do sistema a desenvolver, tais como a escalabilidade, a usabilidade e a segurança. Posteriormente, delimitamos o contexto do sistema, construindo uma decomposição funcional para compreender as fronteiras e capacidades das funcionalidades oferecidas.

De seguida, estabelecemos a arquitetura que sustentará o sistema, decidindo-a com base na análise dos requisitos não funcionais e na sua devida priorização, com especial atenção ao suporte a serviços externos e à modularidade.

Por último, descrevemos detalhes relativos à implementação do sistema. Começamos por descrever os componentes principais do sistema e a forma como se comunicam entre si, bem como a implementação de todas as funcionalidades previstas. Esta fase foca-se na tradução dos requisitos em soluções técnicas robustas que garantam a sustentabilidade e a eficiência do sistema.

2. Requisitos Não Funcionais

2.1. Diagrama de Use Case

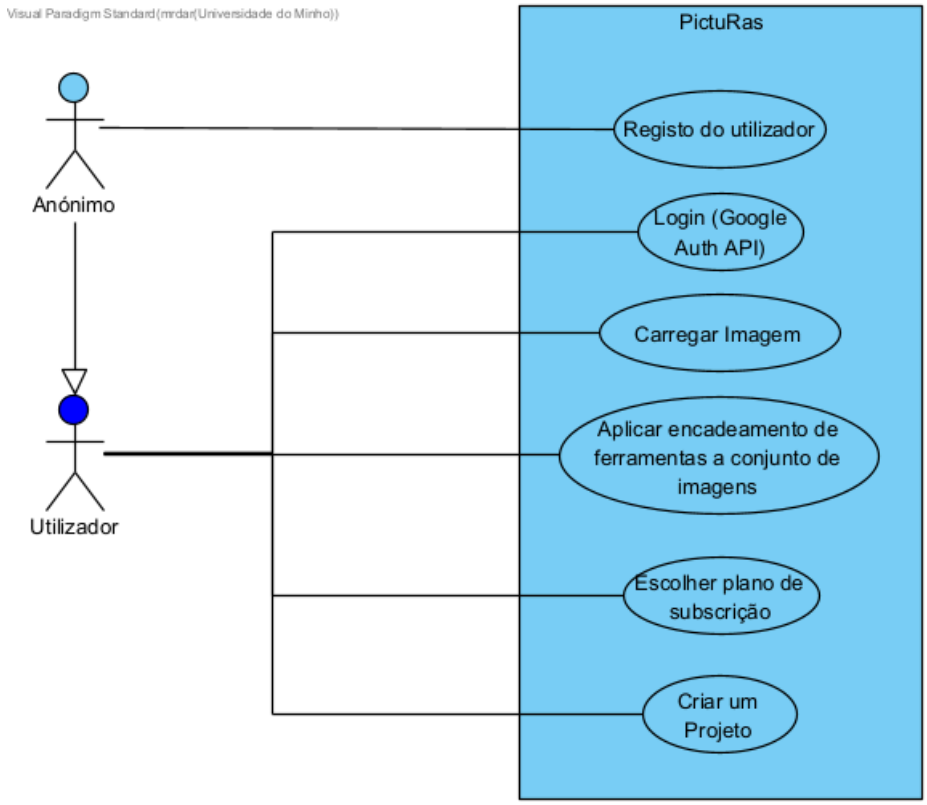


Figura 1: Diagrama dos *Use Cases* mais importantes

No diagrama acima encontra-se os *Use Cases* que o grupo considerou mais fundamentais para que o desenvolvimento da aplicação.

2.2. Principais Requisitos

A equipa de desenvolvimento decidiu focar-se nos requisitos não funcionais que considera ter maior impacto no sistema e, por consequência, influenciar a seleção da solução arquitetural. Esta escolha arquitetural tem como objetivo maximizar as características prioritárias da arquitetura.

Requisito #: RNF21	Tipo: Não funcional	Use cases #: 4

Descrição	**As ferramentas de edição de imagem devem ser processadas rapidamente.**
Rationale	A velocidade é crucial para garantir uma boa experiência ao utilizador, especialmente em tarefas básicas e repetitivas.
Origem	*Engenheiro de Software*
Fit criterion	Pelo menos 95% das ferramentas básicas devem executar em até 1 segundo por imagem e as avançadas em até 3 segundos. O tempo de execução nunca deve ultrapassar os 2 e 5 segundos, respetivamente para cada tipo de ferramenta.
Prioridade	*Should*
Data	2024/10/03

Requisito 1: Performance das ferramentas

Requisito #: RNF22	Tipo: Não funcional	Use cases #: 4

Descrição	**O sistema deve processar até 100 imagens ao mesmo tempo, sem quebras percetíveis no desempenho.**
Rationale	Suportar computação paralela é essencial para manter a eficiência e garantir que múltiplos utilizadores ou processamentos encadeados não causem *starvation*.
Origem	*Engenheiro de Software*
Fit criterion	O sistema deve processar 100 imagens simultâneas com uma degradação de desempenho inferior a 20%.
Prioridade	*Must*
Data	2024/10/03

Requisito 2: Múltiplos processamentos em paralelo

Requisito #: RNF23	Tipo: Não funcional	Use cases #: —
Descrição	**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.**	
Rationale	O crescimento do número de utilizadores pode sobrecarregar o sistema. A escalabilidade garante que o sistema continue eficiente com o aumento da carga.	
Origem	*Engenheiro de Software*	
Fit criterion	O sistema deve suportar crescimentos de 100% dos utilizadores a cada 10 minutos sem quebras de desempenho significativas (menos de 20% de variação no tempo de execução). Recursos são desalocados quando deixam de ser necessários.	
Prioridade	*Must*	
Data	2024/10/03	

Requisito 3: Escalabilidade e elasticidade

Requisito #: RNF24	Tipo: Não funcional	Use cases #: —
Descrição	**A aplicação deve estar disponível 24 horas por dia, todos os dias.**	
Rationale	Garantir alta disponibilidade é crucial para utilizadores globais que dependem da aplicação para trabalho contínuo, independentemente da zona horária.	
Origem	*Engenheiro de Software*	
Fit criterion	A disponibilidade mínima deve ser de 99,9%, permitindo até 40 minutos de downtime por mês.	
Prioridade	*Should*	
Data	2024/10/03	

Requisito 4: Disponibilidade

Requisito #: RNF26	Tipo: Não funcional	Use cases #: 4
Descrição	**A aplicação deve otimizar o uso de recursos computacionais do cliente (e.g., CPU, memória).**	
Rationale	Uma aplicação que consome menos energia em dispositivos móveis prolonga a duração da bateria e melhora a experiência do utilizador.	
Origem	*Engenheiro de Software*	
Fit criterion	A utilização de memória deve ser minimizada.	
Prioridade	*Should*	
Data	2024/10/03	

Requisito 5: Eficiência energética

Requisito #: RNF32	Tipo: Não funcional	Use cases #: —
Descrição	**O sistema deve ser projetado para facilitar a execução de testes.**	
Rationale	A testabilidade assegura que a aplicação seja verificável, mantendo a qualidade e reduzindo o tempo de deteção de erros.	
Origem	*Engenheiro de Software*	
Fit criterion	Testes de regressão devem ser realizados antes de cada atualização para garantir que as funcionalidades anteriores não sejam afetadas.	
Prioridade	*Must*	
Data	2024/10/03	

Requisito 6: Testabilidade

A equipa priorizou os requisitos que oferecem uma experiência de utilizador contínua e eficaz, mesmo sob carga elevada. O impacto é classificado em uso individual e coletivo, sendo que os requisitos de uso coletivo visam garantir uma experiência estável e eficiente para um grande número de utilizadores simultâneos.

2.3. Requisitos Extra

Ao longo do desenvolvimento, surgiram várias sugestões de novos requisitos, que foram posteriormente analisadas e validadas com os stakeholders. Após essa análise, ficou decidido que um desses requisitos será realmente implementado. Assim, apresentamos de seguida o novo requisito:

Requisito #: RNF42	Tipo: Não funcional	Use cases #: —
Descrição	**Oferecer periodo de experimentção "trial" gratuito apenas a utilizadores sem histórico de trial ou plano premium.**	
Rationale	Aumentar conversões permitindo testes gratuitos para novos utilizadores elegíveis.	
Origem	*Engenheiro de Software*	
Fit criterion	Garantir que apenas utilizadores sem histórico de trial ou premium possam aceder ao trial.	
Prioridade	*Must*	
Data	2024/10/03	

Requisito 7: Trial

2.4. Impacto dos Requisitos no Sistema

Para o uso individual, a ênfase está na performance das ferramentas de edição de imagem (requisito 21). A prioridade é que as ferramentas sejam processadas rapidamente, assegurando uma experiência de utilizador eficiente e responsiva. Este requisito é essencial para garantir que o utilizador não experimente atrasos durante operações básicas e avançadas, mantendo uma sensação de fluidez e usabilidade na aplicação. O impacto disso é que a arquitetura precisa ser otimizada para execução rápida de tarefas e minimizar o tempo de processamento por imagem.

No uso coletivo, a capacidade de processamento em massa (requisito 22) tem grande importância. A possibilidade de processar até 100 imagens ao mesmo tempo, sem quebras significativas de desempenho, reflete a necessidade de uma arquitetura capaz de suportar computação paralela eficiente. Esse requisito implica que a aplicação utilize técnicas de distribuição de carga e processamento simultâneo para garantir que múltiplos utilizadores possam executar operações em paralelo sem comprometer a performance.

A escalabilidade e elasticidade (requisito 23) são críticas para suportar o crescimento da base de utilizadores e picos de carga, mantendo o desempenho e controlando os custos operacionais. A arquitetura precisa de estar preparada para escalar horizontalmente e ajustar a alocação de recursos em tempo real, o que garante que a aplicação se mantenha eficiente mesmo com um aumento abrupto de utilizadores ou operações.

A disponibilidade contínua (requisito 24) é um fator que reforça a necessidade de manter a aplicação acessível 24/7. Para os utilizadores globais, isso significa que o sistema deve ser resistente a falhas e ter uma alta taxa de uptime. A infraestrutura deve ser robusta, com redundâncias que assegurem que o serviço continue operacional mesmo em casos de falhas técnicas.

Por último, a eficiência energética (requisito 26) destaca-se, especialmente em dispositivos móveis. Otimizar o uso de CPU e memória reduz o consumo de energia, prolongando a duração da bateria e melhorando a experiência do utilizador. Este impacto implica que a aplicação deve ser projetada para usar os recursos de forma inteligente, evitando sobrecarga de processamento.

3. Restrições

Aqui, apresentamos as restrições definidas no documento de requisitos e analisamos de que forma estas irão impactar a nossa solução.

3.1. Restrições Temporais

Descrição: O projeto deve ser entregue nas datas mencionadas, incluindo as principais funcionalidades implementadas, a arquitetura da solução e um relatório técnico detalhado que descreva o processo de desenvolvimento. Cada entrega será avaliada para garantir que o projeto está no caminho certo e cumpre os requisitos iniciais estabelecidos pelos clientes. As seguintes restrições temporais aplicam-se:

- Numa primeira fase, uma versão preliminar do relatório técnico será entregue até o dia 18 de outubro de 2024. Este será apresentado aos stakeholders entre os dias 21 e 25 de outubro de 2024 para avaliação e feedback.
- As seguintes fases serão submetidas nas seguintes datas:
    - **Entrega da arquitetura da app:** 22 de novembro de 2024, incluindo a definição completa da arquitetura do sistema.
    - **Defesa da arquitetura:** 25 a 29 de novembro de 2024, durante a qual a equipa deverá justificar as escolhas técnicas e receber feedback detalhado.
    - **Entrega da implementação da app:** 17 de janeiro de 2025, incluindo a versão final do software com todas as funcionalidades implementadas.
    - **Defesa da implementação:** 27 a 31 de janeiro de 2024, focando na avaliação da qualidade da implementação e na identificação de melhorias finais.

Implicações: O cumprimento deste prazo é fundamental para garantir tempo suficiente para o feedback dos stakeholders e para os ajustes necessários antes das fases subsequentes. Atrasos na entrega comprometem as atividades futuras do projeto.

3.2. Restrições Técnicas/Solução

Descrição: A aplicação deverá ser projetada para funcionar na cloud, garantindo acessibilidade global a qualquer utilizador com uma conexão à internet. As seguintes restrições técnicas devem ser consideradas:

- **Plataforma Web:** A aplicação deve ser exclusivamente desenvolvida como uma aplicação web.
- **Hospedagem:** A aplicação deverá estar preparada para ser disponibilizada numa infraestrutura Cloud (AWS, Azure, Google Cloud), aspetos como escalabilidade, segurança e otimização de custos são fulcrais.
- **Segurança e Privacidade:** É obrigatório garantir que a plataforma implemente medidas robustas de segurança, incluindo a proteção ataques comuns (e.g.,SQL injection, XSS) e políticas adequadas de gestão de dados dos utilizadores, conforme regulamentos de privacidade como RGPD.
- **Escalabilidade:** A arquitetura da aplicação deve ser elástica para suportar múltiplos utilizadores simultâneos, e o desempenho do sistema deve ser otimizado para garantir uma experiência fluida, independentemente do número de utiliza dores ativos ou da carga de processamento.

Implicações: Estas restrições técnicas são fundamentais para assegurar que o desenvolvimento da aplicação ocorra eficientemente. Falhas no cumprimento destas restrições podem comprometer a usabilidade,

segurança e acessibilidade da aplicação, resultando num produto que não corresponde às expectativas dos stakeholders.

Estas restrições exigem que a equipa de desenvolvimento esteja atenta a alguns detalhes, tais como: a necessidade de projetar uma arquitetura que suporte uma infraestrutura escalável na cloud. Além disso, a aplicação deve incluir camadas robustas de segurança integradas na arquitetura, garantindo proteção contra ameaças. É igualmente essencial que a arquitetura seja focada em tecnologias web responsivas e compatíveis com diversos dispositivos.

3.3. Restrições Orçamentais

Descrição: O desenvolvimento da aplicação deve respeitar restrições financeiras estabelecidas pelo cliente. A equipa de desenvolvimento deve planear o projeto tendo em conta os custos de infraestrutura, desenvolvimento e manutenção. As principais restrições orçamentais incluem:

- **Salários dos elementos da equipa de desenvolvimento:** Os salários devem ser claramente definidos e controlados no orçamento.
- **Custos de hospedagem na nuvem:** A aplicação será disponibilizada na nuvem, com custos a serem controlados relativos a:
    - **Armazenamento:** Os custos de armazenamento devem ser monitorizados e estimados com precisão.
    - **Escalabilidade:** Deve-se considerar os custos adicionais para escalar a infraestrutura em resposta a picos de uso.
- **Manutenção e suporte:** A manutenção contínua da aplicação deve ser planeada financeiramente, garantindo a sustentabilidade do projeto a longo prazo.

Implicações: O não cumprimento dessas restrições financeiras pode comprometer a viabilidade do projeto, limitando a escalabilidade e o suporte aos utilizadores.

3.4. Restrições de Âmbito

Descrição: O projeto deve seguir requisitos específicos definidos pelos stakeholders. As seguintes restrições de âmbito são aplicáveis:

- **Perfis de utilizador:** A aplicação terá os seguintes perfis:
    - **Anónimo:** Permite operações em fotografias com dimensões limitadas.
    - **Registado (gratuito):** Permite até cinco operações por dia em fotografias.
    - **Premium:** Sem limites de operações ou dimensões.
- **Ferramentas básicas:** A aplicação deve incluir um conjunto básico de ferramentas de processamento de imagens (e.g.: binarização, alteração de tamanho, remoção de fundo, etc.). As ferramentas devem aceitar parâmetros ajustáveis pelo utilizador.
- **Ferramentas avançadas:** Para além das ferramentas básicas, a aplicação deve oferecer ferramentas avançadas, geralmente apoiadas em IA, tais como:
    - Contagem de pessoas.
    - Extração de texto (OCR).
    - Identificação de objetos.
    - Alterações específicas (e.g.: colocar óculos em pessoas).
- **Encadeamento de ferramentas e processamento em lote:** A aplicação deve permitir encadear ferramentas, onde o output de uma ferramenta serve como input para a próxima. Deve ser possível

aplicar sequências de ferramentas automaticamente a um conjunto de imagens, utilizando arquivos .zip ou diretorias. A funcionalidade deve ser restrita a ferramentas compatíveis, que aceitem o tipo de output produzidos pela ferramenta anterior.

Implicações: O não cumprimento destas restrições de âmbito, pode resultar em funcionalidades inadequadas ou incompletas, comprometendo a experiência do utilizador e dos objetivos do projeto.

Estas restrições exigem que a equipa de desenvolvimento preste atenção a alguns aspetos, tais como: a necessidade de uma arquitetura que inclua um sistema de controlo de acesso com perfis distintos e limitações específicas para cada perfil. Além disso, a necessidade de suportar operações encadeadas e processamento em lote aponta para uma arquitetura modular, que permita a combinação de várias ferramentas e o processamento eficiente de grandes volumes de dados.

4. Contexto e Âmbito do Sistema

O sistema proposto, denominado "*PictuRAS*", é uma solução inovadora para facilitar o processamento e a edição de imagens, voltada para diferentes tipos de utilizadores, desde amadores até profissionais de design gráfico.

O desenvolvimento do *PictuRAS* é motivado pela necessidade de superar as limitações das ferramentas de edição tradicionais, que muitas vezes requerem infraestrutura robusta ou conhecimentos técnicos avançados. A plataforma busca oferecer um ambiente acessível e flexível, permitindo a edição de imagens diretamente na nuvem, o que elimina a necessidade de recursos computacionais avançados por parte dos utilizadores.

O objetivo principal do *PictuRAS* é fornecer uma plataforma de fácil utilização, acessível a qualquer utilizador, independentemente de suas limitações de recursos ou experiência técnica. Este conceito promove a democratização do acesso a ferramentas de edição de imagens, possibilitando que o utilizador realize operações complexas com poucos cliques.

De acordo com os use cases e os requisitos funcionais estabelecidos no documento de requisitos, foi possível elaborar o seguinte diagrama de contexto do sistema:

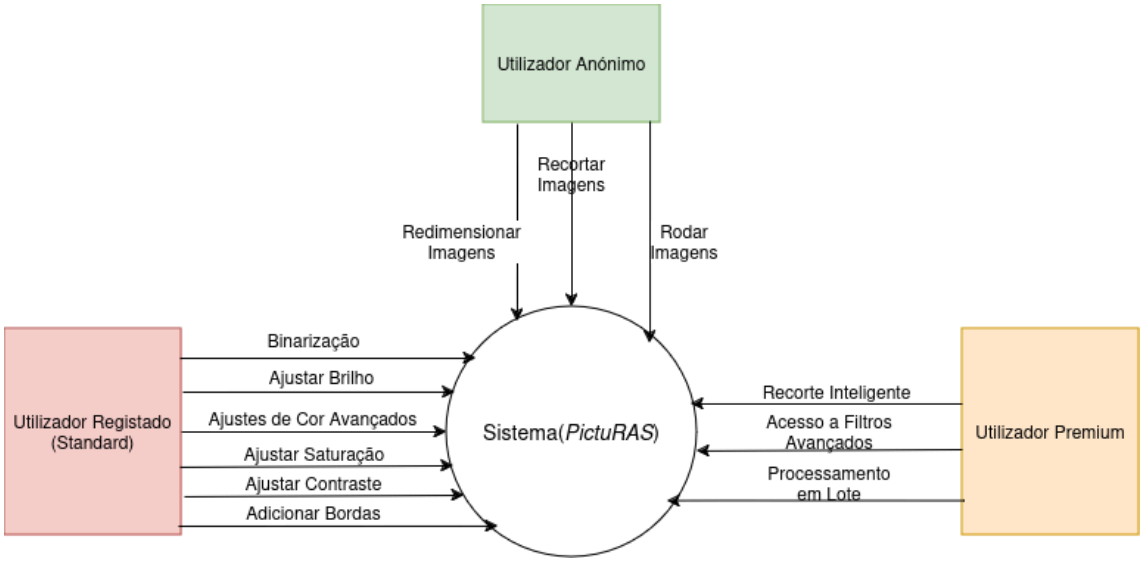


Figura 2: Diagrama de Contexto

O diagrama apresentado visa ilustrar as interações entre o sistema e os atores envolvidos. Com este diagrama, pretende-se definir de forma precisa os limites do sistema, especificando o que está incluído no âmbito do projeto e clarificando as responsabilidades e funcionalidades atribuídas ao sistema. Além disso, permite concluir que cada utilizador dispõe de funcionalidades distintas. Por exemplo, um utilizador anónimo possui acesso limitado às ferramentas básicas, enquanto um utilizador premium dispõe de todas as funcionalidades, incluindo processamento em lote e uso das ferramentas avançadas.

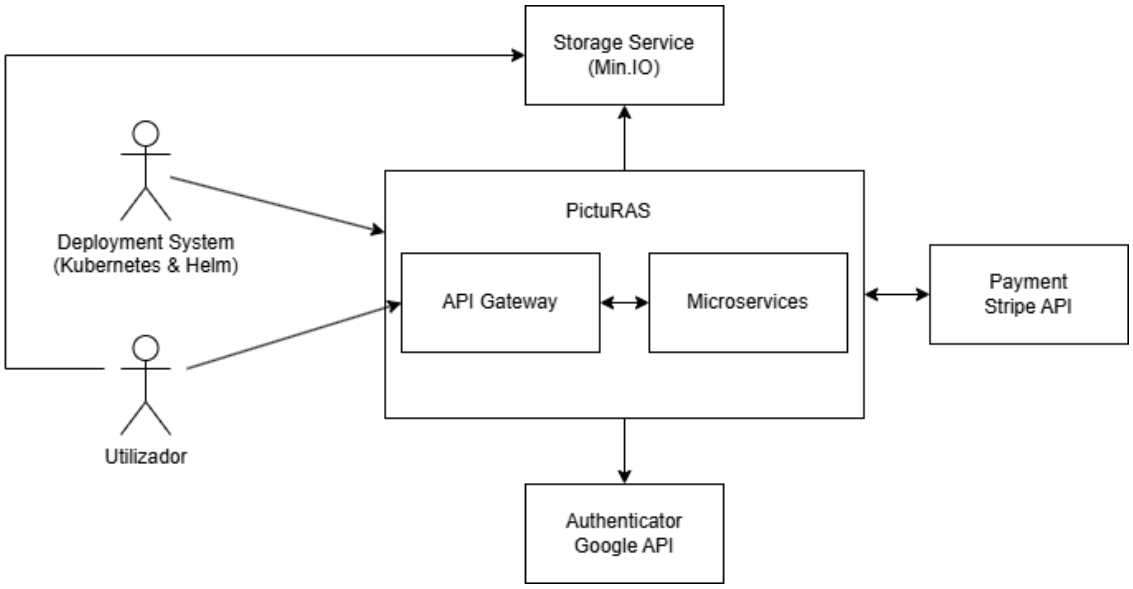


Figura 3: Diagrama de Contexto

O diagrama apresentado mostra como funciona o sistema PictuRAS e as suas conexões principais. Ele é usado por utilizadores que acessam as funcionalidades através de um API Gateway, que organiza as requisições e as direciona para os microserviços internos. Para autenticação, o sistema usa o API do Google e para armazenar os dados utilizamos o Min.IO, um serviço de armazenamento. Além disso, a implantação e o gerenciamento do sistema são feitos com Kubernetes e Helm, que garantem que tudo funcione corretamente e de forma escalável.

5. Estratégia da Solução

Começaremos por escolher a arquitetura com base na análise de arquiteturas já existentes. Em seguida, iremos definir os vários componentes da arquitetura selecionada, adaptando-a para responder às necessidades específicas deste projeto, e detalhando as tecnologias e decisões adotadas.

5.1. Padrão arquitetural

Nesta secção, procederemos à escolha do padrão arquitetural mais adequado, analisando algumas arquiteturas já existentes. Tendo em conta os requisitos definidos na secção anterior, avaliaremos a adequação de cada arquitetura ao caso específico em estudo.

5.1.1. Map Reduce

Requisito Não Funcional	***Rate***	**Justificação**
[24] A aplicação deve estar disponível 24 horas por dia, todos os dias	5/5	A arquitetura MapReduce possui tolerância a falhas robusta, com dados distribuídos e redundância que garantem a continuidade do processamento mesmo em caso de falhas de hardware. Isso contribui para uma alta disponibilidade e robustez, ideal para um sistema que exige operação contínua sem interrupções.
[23] 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.	4/5	O MapReduce é muito adequado para escalabilidade horizontal, uma vez que foi projetado para operar em *clusters* e adicionar nós para lidar com maiores volumes de dados. No entanto, essa escalabilidade não resolve a latência individual de cada tarefa. Portanto, é uma boa escolha para lidar com crescimento de volume e usuários, mas não elimina a limitação de latência.
[22] O sistema deve processar até 100 imagens ao mesmo tempo, sem quedas percetíveis no desempenho	3/5	Embora o MapReduce consiga distribuir o processamento entre vários nós, o sistema não foi otimizado para tarefas de baixa latência ou para um número de pequenas tarefas paralelas, como o processamento de imagens simultâneas em tempo real.
[26] A aplicação deve otimizar o uso de recursos computacionais do cliente (e.g., CPU, memória).	3/5	O MapReduce é relativamente eficiente em termos de uso de recursos em grandes processamentos de dados em lote. No entanto, ele não é tão eficaz para otimização de recursos em tarefas menores e de baixa latência, como o processamento individual de imagens em tempo real, o que pode levar a uso sub-ótimo de CPU e memória.
[32] O sistema deve ser projetado para facilitar a execução de testes	2/5	Testar uma aplicação baseada em MapReduce pode ser complexo devido ao seu processamento distribuído, que exige um ambiente e dados representativos da escala completa do sistema para garantir testes confiáveis.
[21] As ferramentas de edição de imagem devem ser processadas rapidamente.	2/5	MapReduce é ideal para o processamento em lote de grandes volumes de dados, mas apresenta limitações quando é necessário baixa latência e respostas rápidas para tarefas de menor porte, como edição de imagem em tempo real.

5.1.2. Micro-Serviços

Requisito Não Funcional	*Rate*	Justificação
[22] O sistema deve processar até 100 imagens ao mesmo tempo, sem quedas percetíveis no desempenho	5/5	Microsserviços são projetados para escalar horizontalmente e distribuir carga, tornando-os ideais para lidar com grandes volumes de requisições simultâneas. Ao escalar cada serviço de forma independente, o sistema pode facilmente processar várias imagens ao mesmo tempo, mantendo o desempenho.
[23] 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.	5/5	A arquitetura de microsserviços foi desenvolvida exatamente para este tipo de escalabilidade. Serviços independentes permitem aumentar ou diminuir a quantidade de instâncias de cada serviço conforme a demanda. Isso permite que o sistema suporte grandes volumes de tráfego sem afetar o desempenho geral.
[32] O sistema deve ser projetado para facilitar a execução de testes.	5/5	Microsserviços, sendo modulares e independentes, facilitam o desenvolvimento de testes unitários e de integração para cada serviço individualmente. Isso torna o processo de teste muito mais fácil e eficaz, já que cada serviço pode ser testado em isolamento, melhorando a deteção de falhas e facilitando a automação de testes.
[26] A aplicação deve otimizar o uso de recursos computacionais do cliente (e.g., CPU, memória).	4/5	Cada microserviço pode ser desenvolvido e ajustado de forma independente, facilitando a otimização do uso de CPU e memória para cada funcionalidade específica. Essa flexibilidade permite uma otimização eficiente; no entanto, o consumo de recursos pode aumentar se os serviços não forem projetados e monitorizados adequadamente.
[24] A aplicação deve estar disponivel 24 horas por dia, todos os dias	4/5	A arquitetura de microserviços aumenta a disponibilidade ao isolar falhas, garantindo que uma falha em um serviço específico não afete todo o sistema. No entanto, a maior complexidade de monitorização e o aumento no número de pontos de falha requerem maior atenção.
[21] As ferramentas de edição de imagem devem ser processadas rapidamente.	4/5	A arquitetura de microserviços possibilita a execução de operações em tempo real graças à independência e flexibilidade de cada serviço. No entanto, a latência pode aumentar devido à necessidade de comunicação entre os serviços e ao overhead de rede.

5.1.3. Publish-Subscribe

Requisito Não Funcional	*Rate*	Justificação
[23] 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.	5/5	A arquitetura publish-subscribe é altamente eficaz para escalar horizontalmente. Ela permite a adição de novos produtores e consumidores sem interromper o sistema, distribuindo a carga de trabalho e mantendo o desempenho à medida que o número de utilizadores e o volume de imagens aumentam.

[22] O sistema deve processar até 100 imagens ao mesmo tempo, sem quedas percetíveis no desempenho	4/5	Com a arquitetura publish-subscribe, é possível processar múltiplas mensagens (imagens) em paralelo, distribuindo-as entre vários consumidores. Isso permite o processamento simultâneo de até 100 imagens sem impactos significativos no desempenho.
[21] As ferramentas de edição de imagem devem ser processadas rapidamente.	4/5	A publish-subscribe facilita a comunicação assíncrona e a notificação de eventos, permitindo que os resultados sejam transmitidos aos clientes assim que estiverem prontos. Isso possibilita a atualização quase em tempo real, melhorando a experiência do usuário ao fornecer feedback imediato.
[24] A aplicação deve estar disponivel 24 horas por dia, todos os dias	3/5	Embora a arquitetura publish-subscribe possa contribuir para a alta disponibilidade através de redundância e distribuição de carga, ela não garante por si só disponibilidade contínua. São necessárias implementações adicionais, como mecanismos de failover e monitoramento, para assegurar operação 24/7.
[32] O sistema deve ser projetado para facilitar a execução de testes.	2/5	Testar sistemas baseados em publish-subscribe pode ser mais complexo devido à natureza assíncrona e distribuída da arquitetura. Embora seja possível, requer frameworks e abordagens de teste específicos para lidar com a comunicação assíncrona entre componentes.
[26] A aplicação deve otimizar o uso de recursos computacionais do cliente (e.g., CPU, memória).	2/5	A publish-subscribe pode ajudar na distribuição eficiente de tarefas, mas não é especificamente orientada para a otimização de recursos internos. Para otimizar CPU e memória, seriam necessárias técnicas adicionais de gestão de recursos e otimização de código.

5.1.4. Peer-to-Peer

Requisito Não Funcional	***Rate***	**Justificação**
[23] 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.	5/5	A escalabilidade horizontal é mais que garantida numa arquitetura P2P, pois novos nós podem ser adicionados conforme o aumento de utilizadores, mantendo o desempenho através da distribuição da carga de trabalho.
[22] O sistema deve processar até 100 imagens ao mesmo tempo, sem quedas percetíveis no desempenho	4/5	A arquitetura P2P permite a distribuição de tarefas entre vários nós da rede, possibilitando o processamento paralelo de múltiplas imagens. Isso torna viável processar até 100 imagens em simultâneo com um impacto reduzido no desempenho.
[21] As ferramentas de edição de imagem devem ser processadas rapidamente.	2/5	A arquitetura P2P facilita a distribuição de processamento entre diferentes nós, o que pode ajudar a acelerar o tempo de resposta. No entanto, a latência de comunicação e a coordenação entre os nós podem impactar a rapidez da resposta ao cliente em comparação com arquiteturas mais centralizadas.

[24] A aplicação deve estar disponível 24 horas por dia, todos os dias	3/5	A arquitetura contribui para a alta disponibilidade ao eliminar pontos únicos de falha, uma vez que cada nó pode operar de forma independente. No entanto, para garantir disponibilidade contínua, é necessário implementar mecanismos de monitoramento e estratégias de recuperação de falhas.
[32] O sistema deve ser projetado para facilitar a execução de testes.	2/5	Testar sistemas baseados em P2P pode ser mais complexo devido à natureza distribuída e à comunicação entre os nós. Requer ferramentas e estratégias específicas para simular e verificar a interação entre os vários componentes da rede.
[26] A aplicação deve otimizar o uso de recursos computacionais do cliente (e.g., CPU, memória).	3/5	A arquitetura P2P pode distribuir a carga de processamento, mas a otimização do uso de recursos locais depende de como cada nó lida com as suas tarefas. Implementações eficientes e a gestão de recursos locais são necessárias para garantir uma utilização otimizada da CPU e da memória dos clientes.

5.1.5. Model-Centered

Requisito Não Funcional	***Rate***	**Justificação**
[23] 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.	5/5	No estilo Model-centered, a centralização do modelo permite a adição de mais visualizações e controladores sem afetar o sistema, facilita a escalabilidade e suporta o crescimento de usuários e carga de imagens.
[22] O sistema deve processar até 100 imagens ao mesmo tempo, sem quedas percetíveis no desempenho	5/5	O Model-centered garante que os componentes funcionem de forma independente e eficiente, permite a execução de muitas operações simultâneas sem comprometer o desempenho.
[21] As ferramentas de edição de imagem devem ser processadas rapidamente.	4/5	O Model-centered facilita atualizações em tempo real, permite que o modelo central notifique as visualizações de forma ágil. No entanto, o desempenho pode depender da complexidade das operações.
[26] A aplicação deve otimizar o uso de recursos computacionais do cliente (e.g., CPU, memória).	4/5	No estilo Model-centered, a centralização e separação dos componentes ajudam a otimizar o uso de recursos, garante um desempenho eficiente. No entanto, operações mais complexas podem sobrecarregar os recursos do cliente.
[24] A aplicação deve estar disponivel 24 horas por dia, todos os dias.	4/5	O Model-centered contribui para alta disponibilidade, pois a centralização do modelo e a separação dos componentes permitem isolar falhas e manter a operação contínua.
[32] O sistema deve ser projetado para facilitar a execução de testes.	3/5	A facilidade de teste é uma qualidade associada à modifiabilidade e extensibilidade. Em uma arquitetura Model-centered, a separação de componentes de visualização e controle ajuda na testabilidade do sistema.

5.2. Decomposição Funcional

O software Picturas é um sistema com elevada complexidade. Assim sendo, o grupo de desenvolvimento decidiu elaborar uma decomposição funcional descrita num diagrama de blocos. Esse diagrama divide as funcionalidades do sistema a vários níveis de profundidade, o que facilita a compreensão do sistema e por sua vez permite um desenvolvimento mais otimizado.

Através de uma análise dos requisitos funcionais e os dos elementos modificados por esses requisitos ao nível de dados, assim como os *use cases* a que estão associados, foi elaborado o diagrama abaixo.

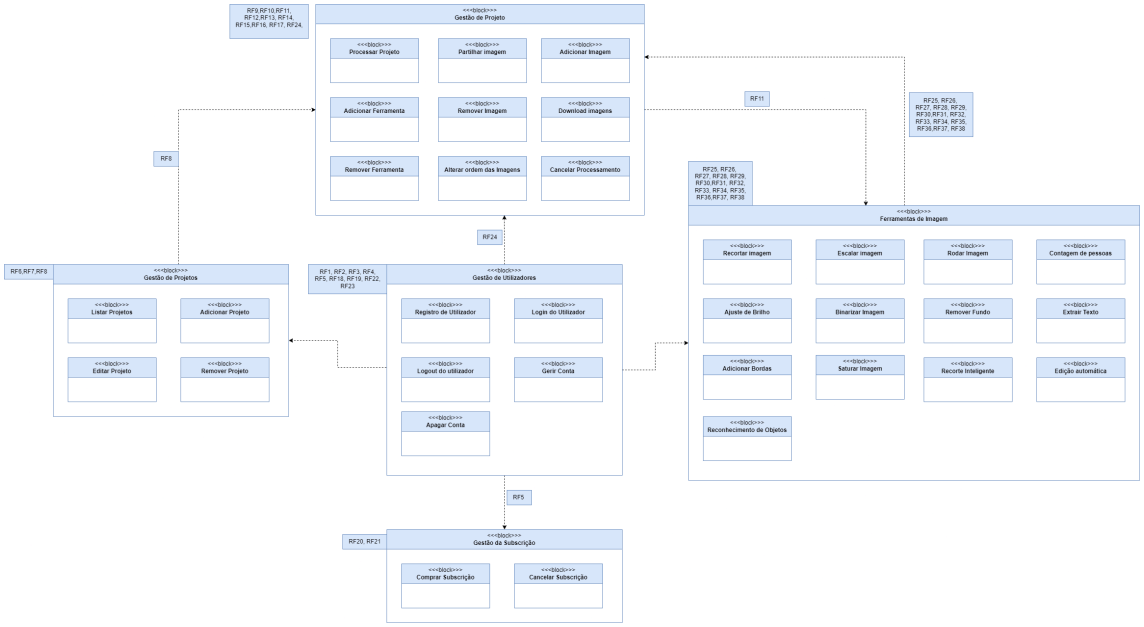


Figura 4: Diagrama de Blocos - Decomposição Funcional

5.3. Definição da Arquitetura

Após a análise de várias arquiteturas realizada anteriormente, a equipa de desenvolvimento avaliou que, para este tipo de sistema, a melhor arquitetura a adotar é o padrão de **microsserviços**. Esta arquitetura oferece grande escalabilidade, desempenho, resiliência (com falhas isoladas), facilidade de manutenção e atualização, bem como maior simplicidade nos testes. Além disso, permite o uso de tecnologias específicas para cada microserviço. A escolha foi feita por consenso geral.

5.3.1. Identificação de microsserviços

Através do estudo do projeto *PictuRAS*, foram definidos os seguintes microsserviços:

1. **Microserviço de Recorte de Imagens**
    - **Responsabilidades**: Realizar o recorte manual de imagens com base nas coordenadas fornecidas pelo utilizador.
    - **Entrada**: Imagem original e parâmetros de recorte
    - **Saída**: Imagem recortada.
    - **Tecnologias**: Sharp.
2. **Microserviço de Redimensionamento de Imagens**
    - **Responsabilidades**: Redimensionar imagens para dimensões específicas.
    - **Entrada**: Imagem original, nova largura e altura.
    - **Saída**: Imagem redimensionada.
    - **Tecnologias**: Sharp.

3. **Microserviço de Ajuste de Saturação**
    - **Responsabilidades**: Ajustar a saturação das cores de uma imagem.
    - **Entrada**: Imagem original e nível de saturação desejado.
    - **Saída**: Imagem com saturação ajustada.
    - **Tecnologias**: Sharp.
4. **Microserviço de Ajuste de Brilho**
    - **Responsabilidades**: Modificar o brilho de uma imagem.
    - **Entrada**: Imagem original e nível de brilho desejado.
    - **Saída**: Imagem com brilho ajustado.
    - **Tecnologias**: Sharp.
5. **Microserviço de Ajuste de Contraste**
    - **Responsabilidades**: Ajustar o contraste de uma imagem.
    - **Entrada**: Imagem original e nível de contraste desejado.
    - **Saída**: Imagem com contraste ajustado.
    - **Tecnologias**: Sharp.
6. **Microserviço de Binarização**
    - **Responsabilidades**: Converter a imagem em preto e branco utilizando um valor de *threshold* específico.
    - **Entrada**: Imagem original e valor de *threshold*.
    - **Saída**: Imagem binarizada.
    - **Tecnologias**: Sharp.
7. **Microserviço de Rotação de Imagens**
    - **Responsabilidades**: Rodar imagem num ângulo especificado.
    - **Entrada**: Imagem original e ângulo de rotação.
    - **Saída**: Imagem rotacionada.
    - **Tecnologias**: Sharp.
8. **Microserviço de Recorte Inteligente**
    - **Responsabilidades**: Realizar um recorte automático com base no conteúdo da imagem.
    - **Entrada**: Imagem original.
    - **Saída**: Imagem recortada automaticamente.
    - **Tecnologias**: TensorFlow.Js + coco-ssd, Sharp
9. **Microserviço de Ajuste Automático**
    - **Responsabilidades**: Ajustar automaticamente brilho, contraste e saturação com base no conteúdo da imagem.
    - **Entrada**: Imagem original.
    - **Saída**: Imagem ajustada automaticamente.
    - **Tecnologias**: Sharp.
10. **Microserviço de Remoção de Fundo**
    - **Responsabilidades**: Remover o fundo de uma imagem, deixando apenas o objeto principal.
    - **Entrada**: Imagem original.
    - **Saída**: Imagem com fundo removido.
    - **Tecnologias**: sharp-remove-bg-ai.
11. **Microserviço de OCR (Reconhecimento Óptico de Caracteres)**
    - **Responsabilidades**: Extrair texto de imagens.
    - **Entrada**: Imagem original.
    - **Saída**: Texto extraído.
    - **Tecnologias**: Tesseract.Js.
12. **Microserviço de Reconhecimento de Objetos**
    - **Responsabilidades**: Identificar e etiquetar objetos numa imagem.
    - **Entrada**: Imagem original.

- **Saída**: Lista de objetos identificados e as suas localizações.
- **Tecnologias**: TensorFlow com o modelo coco-ssd.

13. **Microserviço de Contagem de Pessoas**
    - **Responsabilidades**: Contar o número de pessoas presentes numa imagem.
    - **Entrada**: Imagem original.
    - **Saída**: Número de pessoas identificadas.
    - **Tecnologias**: TensorFlow.Js com o modelo coco-ssd.
14. **Microserviço de Adição de Bordas**
    - **Responsabilidades**: Adicionar bordas coloridas às imagens.
    - **Entrada**: Imagem original e cor da borda desejada.
    - **Saída**: Imagem com borda adicionada.
    - **Tecnologias**: Sharp.

Decidimos definir um microserviço para cada funcionalidade do sistema, em vez de agrupar várias funcionalidades, de forma a evitar a sobrecarga de um microserviço no caso de agrupamento. Se as funcionalidades estivessem agrupadas no mesmo microserviço, uma funcionalidade poderia não conseguir ser utilizada devido à sobrecarga das funcionalidades do mesmo microserviço. Com a abordagem adotada, este problema não ocorre, embora haja um aumento na latência de comunicação entre os microserviços. No entanto, a equipa de desenvolvimento considerou que esta era a melhor solução a seguir.

5.4. Modelos de dados das Bases de Dados

A equipa de desenvolvimento concluiu que a maioria dos microserviços, devido ao seu foco exclusivo no processamento de imagens, não necessita de bases de dados individuais integradas. Assim, foi decidido que toda a arquitetura será suportada por uma única base de dados centralizada.

Para a base de dados, optámos pela utilização do ***MongoDB***, no qual definimos duas *collections* fundamentais: uma dedicada à gestão de projetos e outra à gestão de utilizadores.

Nesta secção, serão apresentados os dicionários de dados correspondentes a cada uma das *collections* definidas, descrevendo em detalhe a sua estrutura, os campos e a finalidade de cada elemento.

5.4.1. Users Collection

Nome do campo	**Tipo**	**Descrição**
_id	MongoID	Identificação única do usuário na base de dados.
name	String	Nome do Usuário
email	String	Endereço de e-mail do usuário (único).
googleId	String	Identificador único associado à conta Google do utilizador
isPremium	Boolean	Se o utilizador comprou o Premium e a sua subscrição está ativa.
usedTrial	Boolean	Se o utilizador utilizaou o periodo de experimentação (trial) de premium.

5.4.2. Collection Gestão de Projetos

Nome do campo	**Tipo**	**Descrição**
_id	MongoID	Identificação única do Projeto na base de dados.

name	String	Nome do Projeto.
userId	MongoID	ID do utilizador responsável pelo projeto.
images	Array(Image)	Lista de imagens associadas ao projeto. Cada imagem é um objeto.
stages	Array(Stage)	Lista de etapas associadas ao projeto.
createdAt	Date	Data e hora em que o projeto foi criado.
updatedAt	Date	Data e hora da última atualização do projeto.

5.4.3. Especificação dos objetos Imagem, Stage e Operation

Para garantir uma estrutura de dados clara e funcional, detalhamos abaixo os elementos que compõem os objetos Imagem, Stage e Operation, usados na coleção de Gestão de Projetos. Estes objetos são fundamentais para a organização e gestão eficiente dos dados relacionados aos projetos e suas operações.

1. Imagem

O objeto Imagem é responsável por armazenar informações detalhadas sobre cada imagem associada ao projeto, incluindo metadados essenciais para o seu processamento e análise. Fundamentalmente, um objeto Imagem incluirá:

- **_id**: um identificador único que distingue a imagem na base de dados;
- **url** (String): o caminho ou URL onde a imagem está armazenada;
- **stages**: Array de URLs (Strings) que contem o caminho para a imagem processada ou dados (CSV ou JSON) com a informação do estágio de mesmo indice.

2. Stage

O objeto Stage representa uma etapa ou operação realizada no âmbito do projeto. Cada etapa é detalhadamente descrita para permitir o rastreamento do histórico de alterações e dos parâmetros utilizados. De forma geral, um objeto Stage incluirá:

- **_id**: um identificador único que distingue a etapa na base de dados;
- **type** (Int): o tipo de operação ou filtro aplicado na etapa;
- **args**: os argumentos específicos associados à operação, que definem os parâmetros aplicados (exemplo: intensidade, valores de redimensionamento, configurações de brilho ou contraste).

5.5. Tecnologias para a implementação

Nesta secção, iremos descrever as tecnologias selecionadas pela equipa de desenvolvimento para a implementação do sistema.

5.5.1. Frontend

Para o desenvolvimento do frontend, utilizaremos a linguagem JavaScript com o apoio da biblioteca ***React*** para a componente interativa da interface do utilizador.

5.5.2. Microsserviços e API Gateway

Nas aplicações dos microsserviços, será utilizada a linguagem de programação ***JavaScript***, juntamente com o ambiente de execução ***Node.js*** e a framework ***Express***, bem como o ***Mongoose*** para ligação à Base de Dados.

5.5.3. Bases de Dados

A tecnologia a utilizar para a persistência de dados será o ***MongoDB***, um sistema de gestão de base de dados não relacional. O ***MongoDB*** foi escolhido pela sua flexibilidade, escalabilidade e por ser adequado para o tipo de dados em causa no sistema.

5.6. Decisões organizacionais

A equipa de desenvolvimento é composta por 8 elementos, e decidiu-se que cada um ficará responsável por uma parte do código. A distribuição dos elementos pelos diferentes componentes do sistema será feita da seguinte forma:

1. **Responsáveis pelos Microsserviços de Filtros:**

A equipa responsável pelo desenvolvimento, manutenção e comunicação dos 14 microsserviços será de dois elementos.

2. **Responsável pela Gateway e Gestão de Utilizadores, de Filtros e de Monetização:**

Três elementos da equipa ficarão responsáveis pelo serviço com a gestão de utilizadores, incluindo funcionalidades como login, registo de utilizadores, gestão de permissões e uso de projetos, pipelining de filtros, monetização, entre outros.

3. **Responsáveis pelo Frontend:**

Três elementos da equipa serão responsáveis pelo desenvolvimento do frontend, criando as interfaces de utilizador e garantindo a integração com a API.

5.7. Arquitetura Final

Considerando o trabalho desenvolvido nas categorias anteriores, apresentamos agora a nossa proposta de arquitetura final:

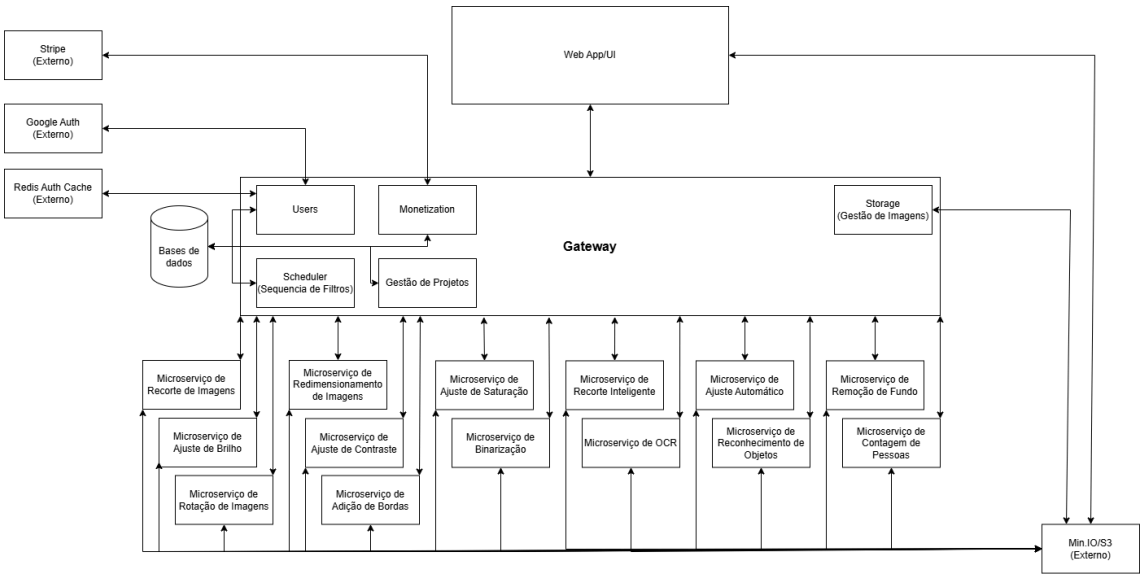


Figura 5: Arquitetura Final do Trabalho

A proposta de arquitetura apresentada na Figura 5 é composta por diversos componentes interligados, organizados de forma a maximizar a eficiência e a escalabilidade do sistema. O núcleo da solução é o *Gateway*, que atua como ponto central de comunicação entre a interface da aplicação *web* e os diversos

microserviços. Estes executam operações específicas, como reconhecimento de objetos, remoção de fundo, e ajustes de imagem, garantindo modularidade e flexibilidade.

Os serviços externos, como *Stripe* para pagamento e *Google Auth* para autenticação, são integrados de forma a oferecer funcionalidades adicionais sem sobrecarregar a infraestrutura interna. A utilização de *Min.IO/S3* para armazenamento externo assegura a persistência e a escalabilidade dos dados.

A arquitetura foi projetada para suportar alta disponibilidade e balanceamento de carga, aproveitando mecanismos de *cache (Redis Auth Cache)* e uma base de dados de utilizadores centralizada para gestão de sessões e autenticações. Este design permite um bom fluxo de trabalho, onde os pedidos dos utilizadores são encaminhados de forma sequencial recorrendo a um *Scheduler* que coordena a execução dos diferentes filtros e processos de imagem.

O grupo decidiu adicionar o Serviço de Storage no Gateway, que seria responsável por realizar o upload para o Bucket no Min.IO e S3. Essa decisão justifica-se, em parte, pela necessidade de atualizar o projeto posteriormente, e também para evitar a troca de grandes ficheiros entre os microserviços, que sobrecarregaria a rede inutilmente.

Outro tema relevante abordado pelo grupo foi o tratamento de utilizadores anónimos. Considerando que as imagens precisam de ser armazenadas no serviço Min.IO/S3 e associadas a um projeto (de forma a unificar a lógica para utilizadores autenticados e anónimos), foi definido que os projetos de utilizadores anónimos seriam mantidos por um período fixo, como 24 ou 48 horas. Esta funcionalidade seria implementada utilizando os mecanismos de Time To Live (TTL) já suportados pelo MongoDB e pelo serviço de Min.Io/S3.

5.8. Reflexões e Futuras Reestruturações

De acordo com o diagrama apresentado, o API Gateway comunica diretamente com os microserviços via HTTP. Para este mecanismo funcionar é necessário que os microserviços escalem horizontalmente e estejam expostos através de um Load Balancer, uma funcionalidade já suportada por ferramentas como o Kubernetes, utilizadas no Deployment. No entanto, se o Gateway falhar durante o processamento de um pedido enviado a um microserviço, a imagem seria armazenada no bucket, mas a base de dados não seria atualizada.

Uma possível solução para este problema envolve a utilização de um Message Broker, como o Apache Kafka ou o RabbitMQ. Este método permitiria subdividir ainda mais o Gateway, aumentando a escalabilidade do sistema. Além disso, melhoraria o desempenho em situações de latência nas respostas dos microserviços e garantiria resiliência em caso de falha de um microserviço, sendo esta uma situação excecional e não prevista pela arquitetura original. Com esta abordagem, os microserviços deixariam de necessitar de um Load Balancer, pois o Gateway não precisaria conhecer diretamente os microserviços.

Contudo, como a API utiliza REST HTTP, o mesmo servidor precisaria responder ao cliente, o que significa que o Gateway continuaria a "bloquear" até obter uma resposta para entregar ao cliente. De forma geral, esta solução aumenta a complexidade da arquitetura, especialmente na gestão de autenticação e serviços de monetização. Por este motivo, o grupo decidiu considerar a implementação desta abordagem como um "extra". Mesmo assim, o grupo optou por conceber a arquitetura com esse mecanismo, apresentado a seguir.

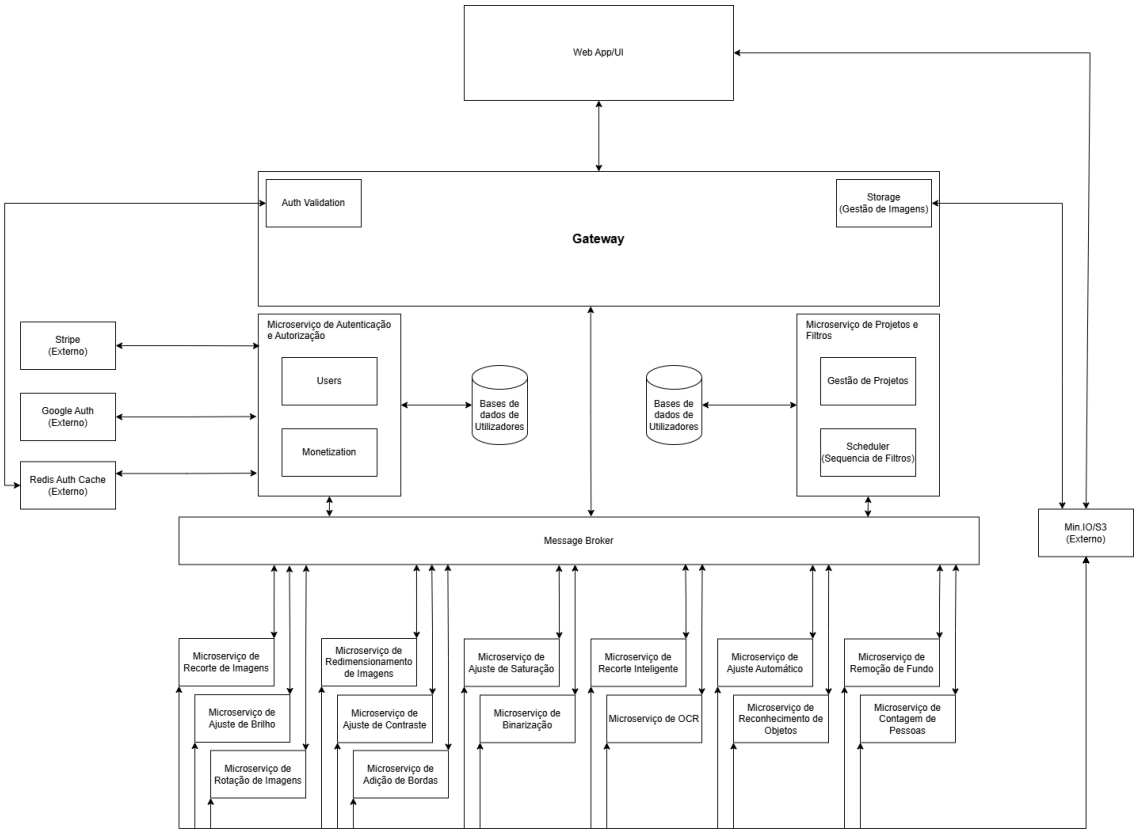


Figura 6: Arquitetura do Trabalho com Message Broker

Um novo componente denominado Auth Validator foi desenvolvido, anteriormente implícito e integrado nas funções de gestão de utilizadores. Este componente é responsável por verificar e validar os pedidos logo na chegada ao Gateway. Assim, os pedidos encaminhados ao Broker já incluem as informações necessárias para validar as disponibilidades relacionadas ao uso dos filtros.

Além disso, a base de dados foi dividida em duas: uma para os microserviços de Autenticação e Autorização e outra para os de Projetos e Filtros. Cada um desses serviços agora abrange as funções que antes estavam centralizadas no Gateway.

6. Building Block View

Nesta secção, iremos explorar os diversos componentes que fazem parte do sistema. O primeiro diagrama apresenta uma visão global, englobando os microserviços, a *API Gateway* e o *frontend* do cliente. O objetivo deste diagrama é identificar os componentes principais do sistema e mostrar as interfaces que cada um fornece e requer, evidenciando a forma como se comunicam entre si. Nos diagramas seguintes, iremos aprofundar os detalhes de cada componente, com especial foco nos microserviços.

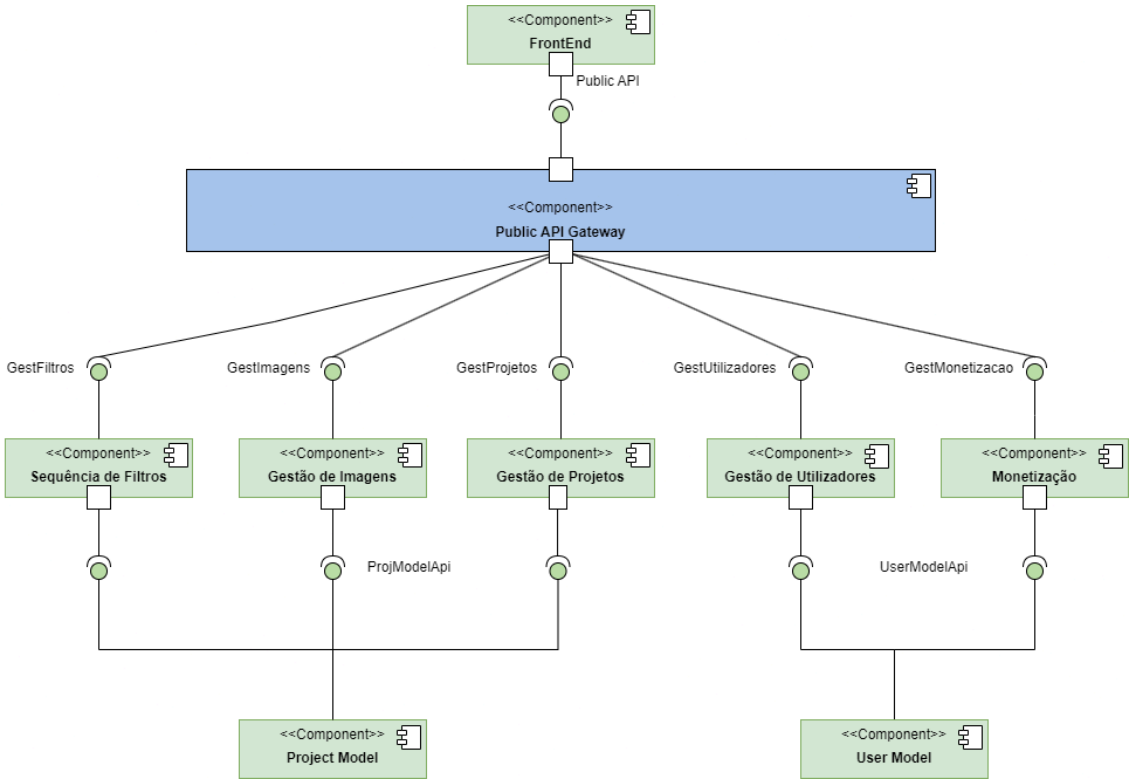


Figura 7: Diagrama de Componentes Geral

6.1. Gestão de Projetos

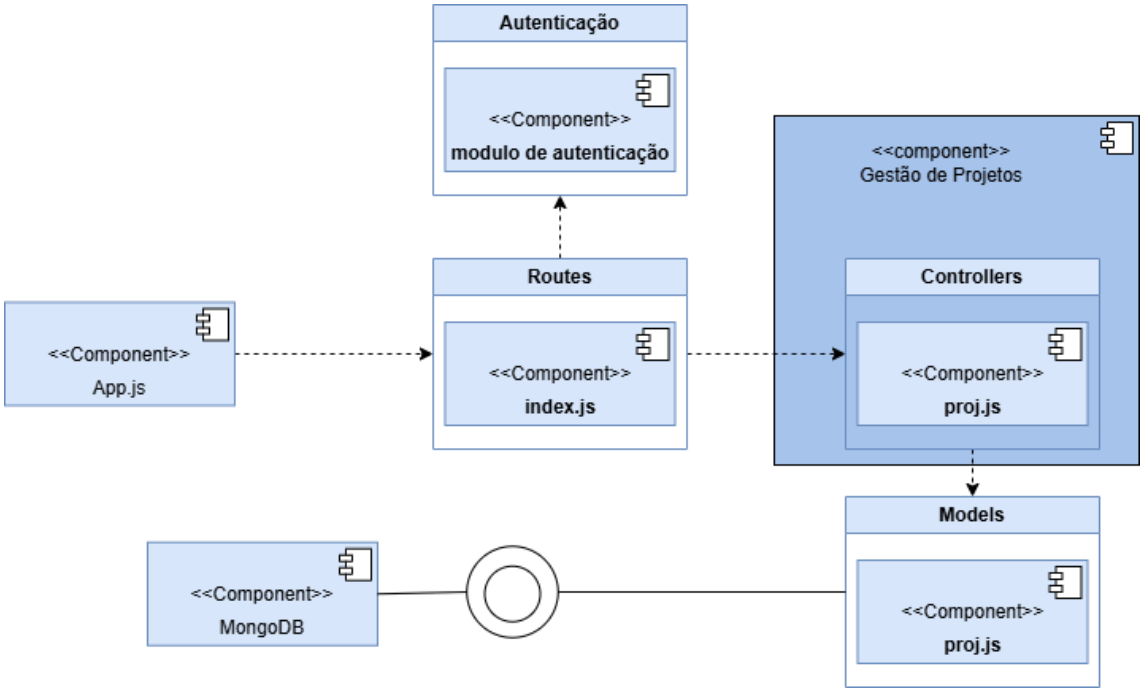


Figura 8: Diagrama de Gestão de Projetos

Neste diagrama, é possível observar a presença de cinco pacotes (packages): Autenticação, Routes, Controllers, Models e a integração com a base de dados (MongoDB).

Os componentes presentes no sistema totalizam seis, sendo eles:

1. O componente MongoDB, correspondente à base de dados.
2. O componente proj.js, incluído no pacote Models.
3. O componente proj.js, incluído no pacote Controllers.
4. O componente index.js, presente no pacote Routes.
5. O componente App.js, que tem como principal função inicializar o serviço.
6. O módulo de autenticação, incluído no pacote Autenticação, responsável pela gestão da autenticação de utilizadores.

No pacote Models, está definida a lógica para estruturar o esquema de dados e realizar todas as operações, como consultas e manipulação de dados. A comunicação entre o MongoDB e os Models é efetuada através de bibliotecas apropriadas para a interação com a base de dados.

No pacote Controllers, mais especificamente no componente proj.js, encontram-se os métodos necessários para gerir as regras de negócio relacionadas com as rotas.

As rotas disponíveis estão implementadas no componente index.js, pertencente ao pacote Routes, e são utilizadas para expor as funcionalidades do sistema.

O componente App.js é responsável por centralizar a inicialização, carregando as rotas definidas em index.js e configurando a aplicação para execução.

O módulo de autenticação, presente no pacote Autenticação, implementa a lógica para gerir o processo de autenticação de utilizadores, integrando-se com o pacote Routes para expor as rotas de autenticação.

A base de dados MongoDB conecta-se diretamente ao pacote Models, representando a camada de persistência de dados.

6.2. Gestão de Utilizadores

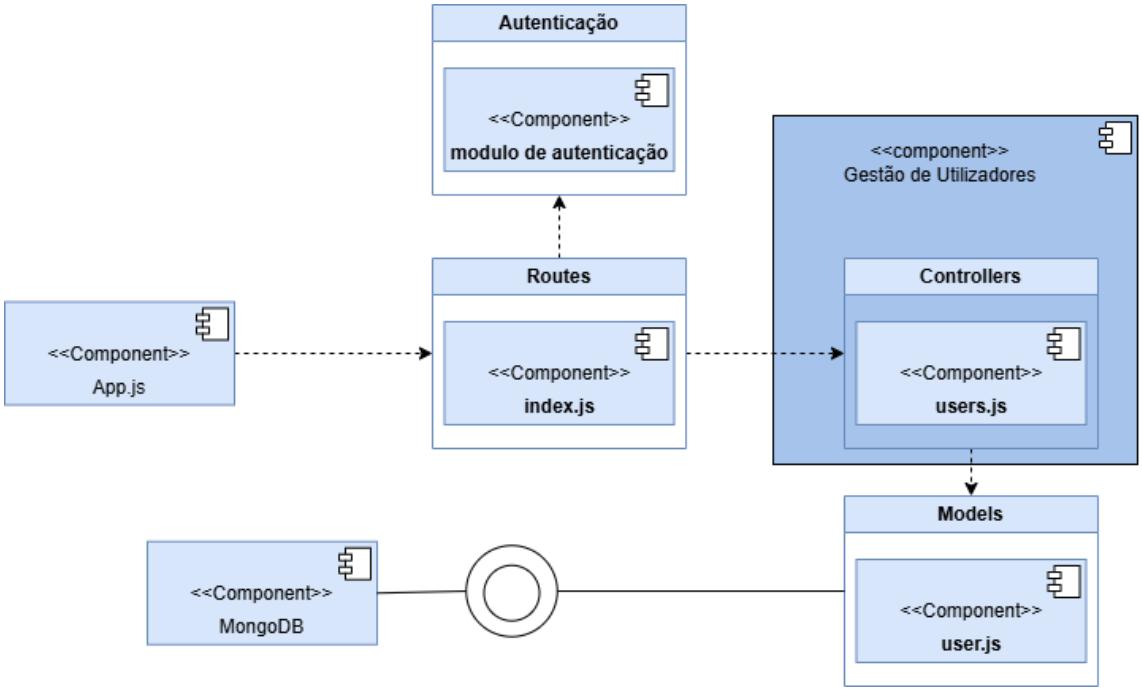


Figura 9: Diagrama de Gestão de Projetos

Neste diagrama, é possível identificar a presença de cinco pacotes (packages): Autenticação, Routes, Controllers, Models e a integração com a base de dados (MongoDB).

Os componentes do sistema totalizam seis, sendo eles:

1. O componente MongoDB, correspondente à base de dados.
2. O componente user.js, incluído no pacote Models.
3. O componente users.js, incluído no pacote Controllers.
4. O componente index.js, presente no pacote Routes.
5. O componente App.js, que desempenha a função principal de inicializar o serviço.
6. O módulo de autenticação, incluído no pacote Autenticação, responsável por gerir a autenticação de utilizadores.

No pacote Models, está definida a lógica para estruturar o esquema de dados e realizar todas as operações, como consultas e manipulação de dados. A comunicação entre o MongoDB e os Models é efetuada através de bibliotecas apropriadas para a interação com a base de dados.

No pacote Controllers, mais especificamente no componente proj.js, encontram-se os métodos necessários para gerir as regras de negócio relacionadas com as rotas.

As rotas disponíveis estão implementadas no componente index.js, pertencente ao pacote Routes, e são utilizadas para expor as funcionalidades do sistema.

O componente App.js é responsável por centralizar a inicialização, carregando as rotas definidas em index.js e configurando a aplicação para execução.

O módulo de autenticação, presente no pacote Autenticação, implementa a lógica para gerir o processo de autenticação de utilizadores, integrando-se com o pacote Routes para expor as rotas de autenticação.

A base de dados MongoDB conecta-se diretamente ao pacote Models, representando a camada de persistência de dados.

6.3. Sequência de Filtros

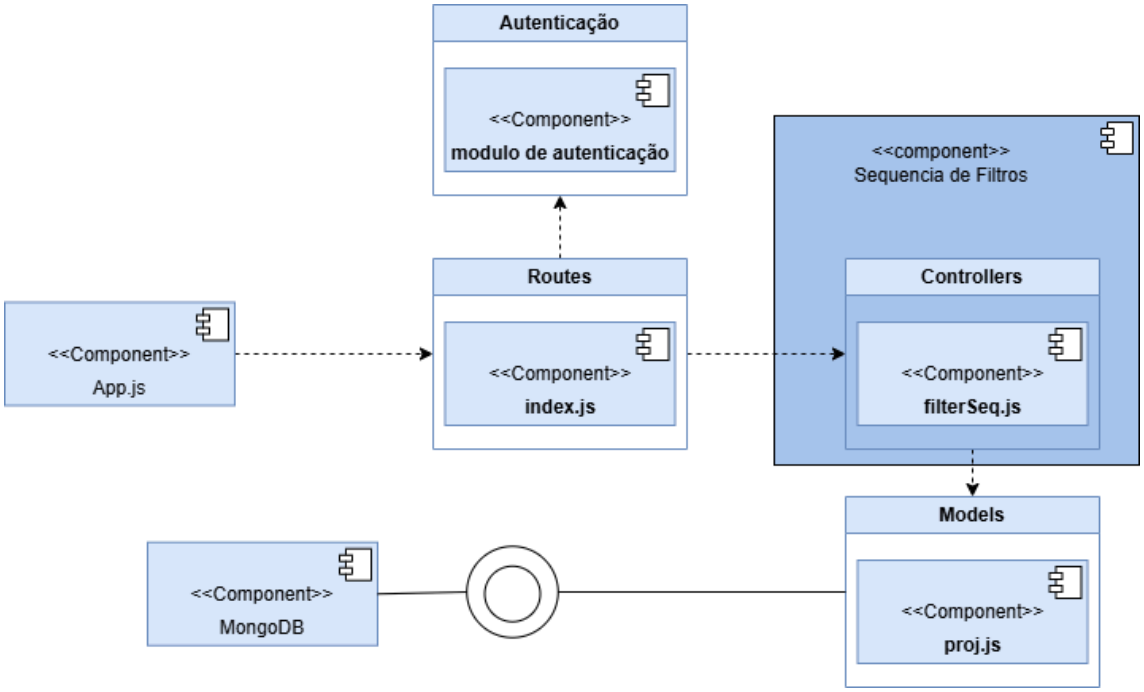


Figura 10: Diagrama de Gestão de Filtros

Neste diagrama, é possível observar a presença de cinco pacotes (packages): Autenticação, Routes, Controllers, Models e a integração com a base de dados (MongoDB).

Os componentes presentes no sistema totalizam seis, sendo eles:

1. O componente MongoDB, correspondente à base de dados.
2. O componente proj.js, incluído no pacote Models.
3. O componente filterSeq.js, incluído no pacote Controllers.
4. O componente index.js, presente no pacote Routes.
5. O componente App.js, que tem como principal função inicializar o serviço
6. O módulo de autenticação, incluído no pacote Autenticação, responsável por gerir o processo de autenticação de utilizadores.

No pacote Models, está definida a lógica para estruturar o esquema de dados e realizar todas as operações, como consultas e manipulação de dados. A comunicação entre o MongoDB e os Models é efetuada através de bibliotecas apropriadas para a interação com a base de dados.

No pacote Controllers, mais especificamente no componente filterSeq.js, encontram-se os métodos necessários para gerir as regras de negócio relacionadas com a sequência de filtros a aplicar às imagens.

As rotas disponíveis estão implementadas no componente index.js, pertencente ao pacote Routes, e são utilizadas para expor as funcionalidades do sistema.

O componente App.js é responsável por centralizar a inicialização, carregando as rotas definidas em index.js e configurando a aplicação para execução.

O módulo de autenticação, presente no pacote Autenticação, implementa a lógica para gerir o processo de autenticação de utilizadores, integrando-se com o pacote Routes para expor as rotas de autenticação.

A base de dados MongoDB conecta-se diretamente ao pacote Models, representando a camada de persistência de dados.

6.4. Gestão de Imagens

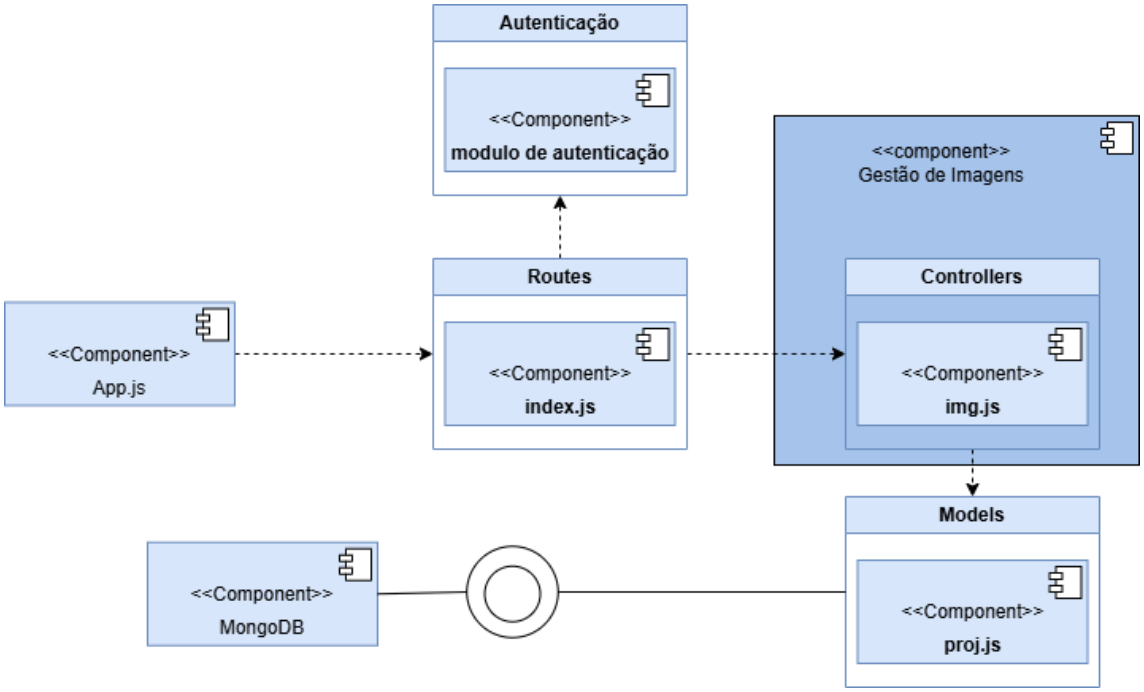


Figura 11: Diagrama de Gestão de Filtros

Neste diagrama, é possível observar a presença de cinco pacotes (packages): Autenticação, Routes, Controllers, Models e a integração com a base de dados (MongoDB).

Os componentes presentes no sistema totalizam seis, sendo eles:

1. O componente MongoDB, correspondente à base de dados.
2. O componente proj.js, incluído no pacote Models.
3. O componente img.js, incluído no pacote Controllers.
4. O componente index.js, presente no pacote Routes.
5. O componente App.js, que tem como principal função inicializar o serviço
6. O módulo de autenticação, incluído no pacote Autenticação, responsável por gerir o processo de autenticação de utilizadores.

No pacote Models, está definida a lógica para estruturar o esquema de dados e realizar todas as operações, como consultas e manipulação de dados. A comunicação entre o MongoDB e os Models é efetuada através de bibliotecas apropriadas para a interação com a base de dados.

No pacote Controllers, mais especificamente no componente proj.js, encontram-se os métodos necessários para gerir as regras de negócio relacionadas com as rotas.

As rotas disponíveis estão implementadas no componente index.js, pertencente ao pacote Routes, e são utilizadas para expor as funcionalidades do sistema.

O componente App.js é responsável por centralizar a inicialização, carregando as rotas definidas em index.js e configurando a aplicação para execução.

O módulo de autenticação, presente no pacote Autenticação, implementa a lógica para gerir o processo de autenticação de utilizadores, integrando-se com o pacote Routes para expor as rotas de autenticação.

A base de dados MongoDB conecta-se diretamente ao pacote Models, representando a camada de persistência de dados.

6.5. Monetização

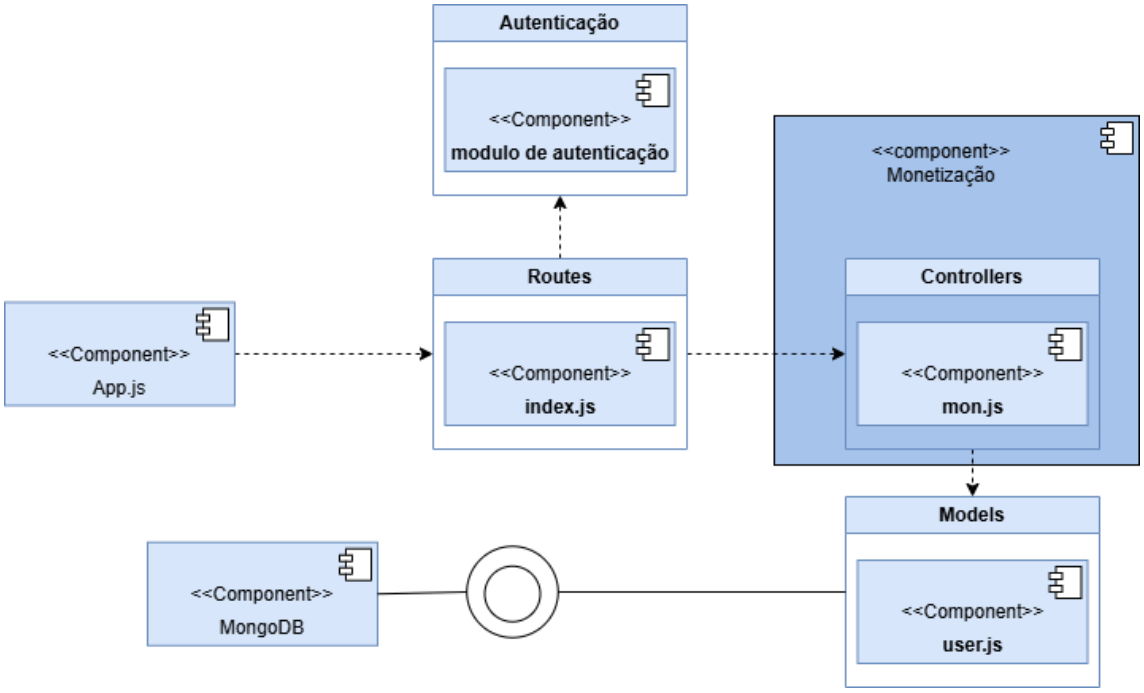


Figura 12: Diagrama de Monetização

Neste diagrama, é possível observar a presença de cinco pacotes (packages): Autenticação, Routes, Controllers, Models e a integração com a base de dados (MongoDB).

Os componentes presentes no sistema totalizam seis, sendo eles:

1. O componente MongoDB, correspondente à base de dados.
2. O componente user.js, incluído no pacote Models.
3. O componente mon.js, incluído no pacote Controllers.
4. O componente index.js, presente no pacote Routes.
5. O componente App.js, que tem como principal função inicializar o serviço
6. O módulo de autenticação, incluído no pacote Autenticação, responsável por gerir o processo de autenticação de utilizadores.

No pacote Models, está definida a lógica para estruturar o esquema de dados e realizar todas as operações, como consultas e manipulação de dados. A comunicação entre o MongoDB e os Models é efetuada através de bibliotecas apropriadas para a interação com a base de dados.

No pacote Controllers, mais especificamente no componente mon.js, encontram-se os métodos necessários para gerir as regras de negócio relacionadas com as rotas.

O comportamento dos restantes componentes já foi descrito anteriormente.

7. Runtime View

Nesta secção, apresentamos o funcionamento da solução proposta através de diagramas de sequência dos use cases do sistema. Estes diagramas cobrem uma variedade de cenários, ilustrando as interações entre os atores e os diversos componentes do sistema, incluindo situações de exceção ou erro e a forma como o sistema as gere.

7.1. Registo do utilizador

Começando pelo **registo do utilizador**, o diagrama de sequência descreve detalhadamente o fluxo necessário para criar uma nova conta. O processo inicia-se com o utilizador a inserir os dados na interface da aplicação, que os encaminha para o servidor para validação. Se os dados forem inválidos, o sistema informa o utilizador, permitindo correções e o reenvio das informações. Caso o utilizador opte por um plano pago, o sistema comunica com o módulo de pagamentos para processar a transação. Após a validação e, se necessário, a confirmação do pagamento, a conta é criada com sucesso, e uma notificação é enviada ao utilizador.

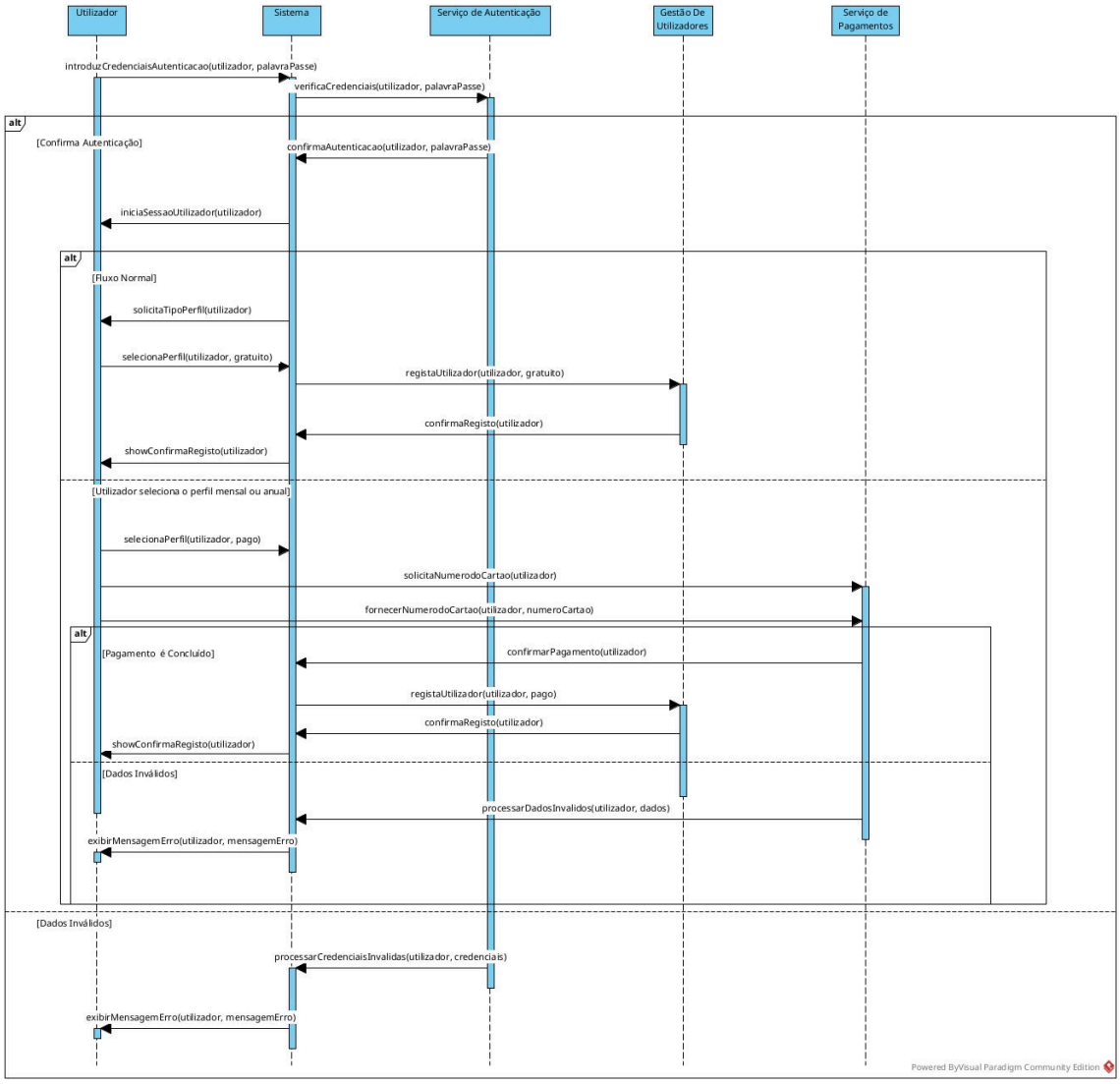


Figura 13: Diagrama de Sequência - Registar Utilizador

7.2. Login do Utilizador

Seguindo para o **login do utilizador**, o diagrama de sequência detalha o fluxo de interação necessário para autenticar um utilizador já registado no sistema. O processo inicia-se com o utilizador a fornecer as suas credenciais de acesso (email ou nome de utilizador e palavra-passe) através da interface da aplicação.

Essas informações são enviadas para o servidor, onde o sistema valida os dados fornecidos. Caso sejam inválidos, o sistema retorna uma mensagem de erro informando o utilizador do problema, permitindo uma nova tentativa de autenticação. Se as credenciais forem válidas, o sistema autentica o utilizador e inicia a sessão, enviando uma confirmação à interface, que atualiza o estado do utilizador para autenticado.

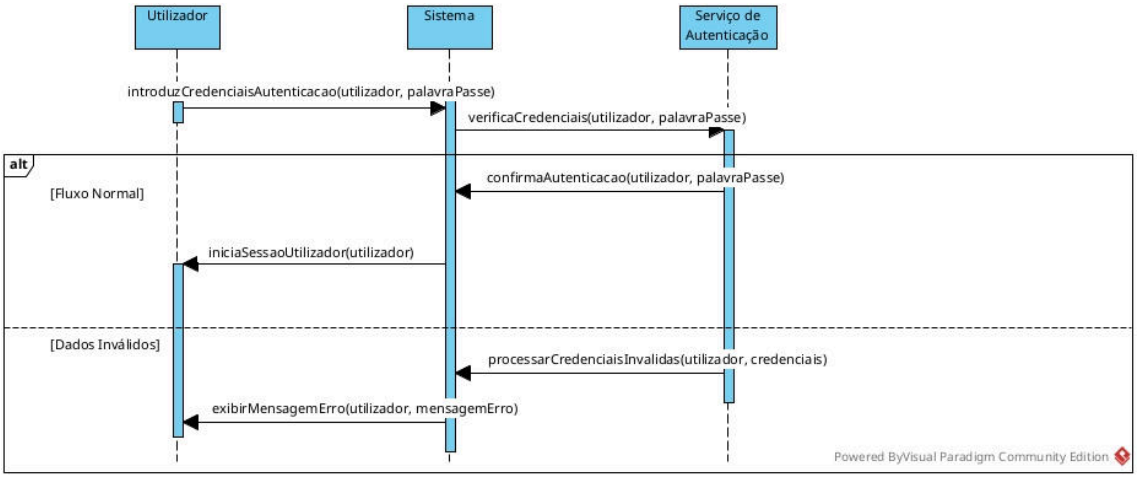


Figura 14: Diagrama de Sequência - Login Utilizador

7.3. Carregar Imagens

Avançando para o **carregamento de imagens**, o diagrama de sequência descreve o processo que permite ao utilizador adicionar uma ou mais imagens ao sistema. O fluxo inicia-se quando o utilizador seleciona as imagens através da interface da aplicação, que as encaminha para o servidor.

No servidor, as imagens são verificadas quanto ao seu formato e tamanho para garantir que cumprem os requisitos definidos. Caso alguma imagem não seja válida, o sistema notifica o utilizador com uma mensagem de erro detalhando o problema. Imagens válidas são então processadas e armazenadas no sistema e uma confirmação é enviada para a interface da aplicação, informando o utilizador do sucesso do carregamento.

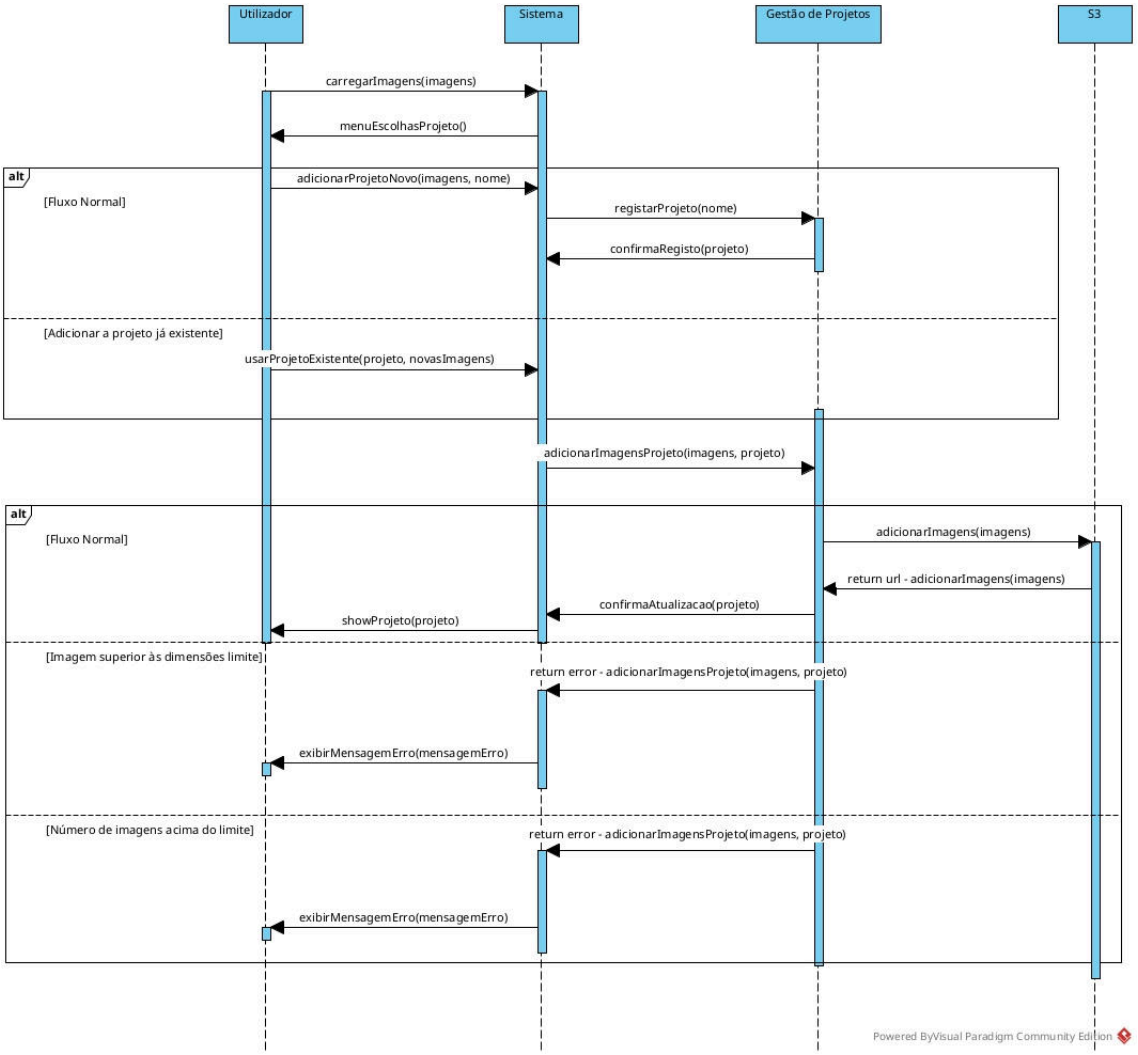


Figura 15: Diagrama de Sequência - Carregar Imagens

7.4. Encadeamento de Ferramentas

O diagrama de sequência descreve o processo de **encadeamento de ferramentas** no sistema. Quando o utilizador seleciona imagens para processamento, o sistema:

1. Aplica as sequências de ferramentas definidas pelo utilizador, como filtros, ajustes de cor, redimensionamento, etc.
2. Após o processamento, devolve as versões finais das imagens ao utilizador.
3. Caso ocorram erros durante o processamento, o sistema notifica o utilizador e devolve as imagens originais sem modificações.

Portanto, este diagrama detalha a interação entre o utilizador e o sistema durante o encadeamento de imagens, aplicando as ferramentas de processamento definidas.

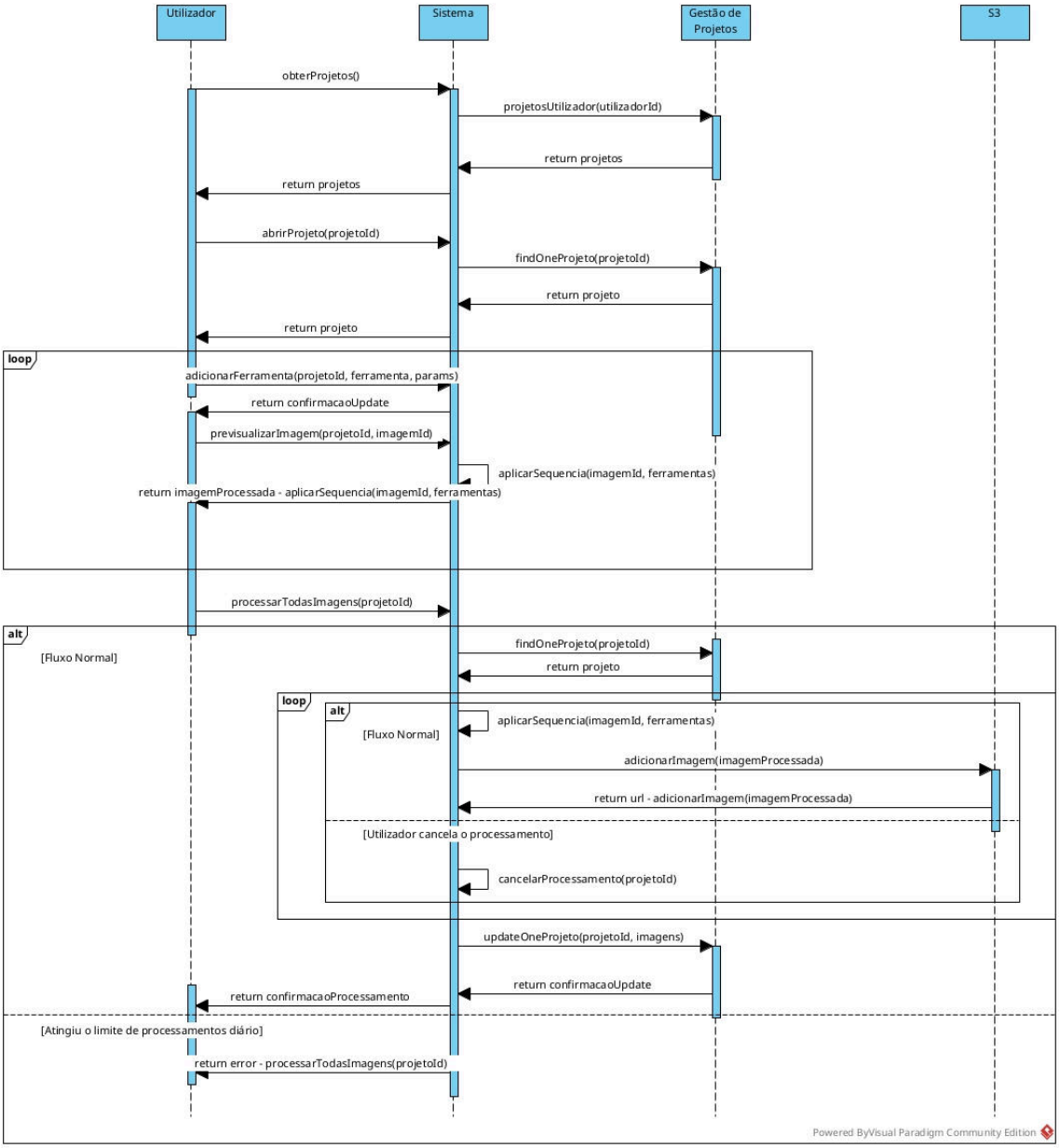


Figura 16: Diagrama de Sequência - Encadeamento de Ferramentas

8. Deployment View

Nesta seção, são apresentados os diagramas de deployment do sistema PictuRAS, destacando tanto o ambiente de desenvolvimento quanto o ambiente de produção. O diagrama de desenvolvimento representa uma configuração simplificada, onde toda a aplicação opera numa única máquina, ideal para testes e validações iniciais. Já o diagrama de produção mostra a arquitetura recomendada para o sistema em funcionamento real, com cada serviço distribuído e isolado para otimizar desempenho e segurança. Comparar essas duas configurações ajuda a esclarecer as adaptações necessárias para garantir a robustez e escalabilidade do sistema em um cenário de produção.

8.1. Contexto de produção

No ambiente de produção, a arquitetura precisa ser mais robusta, escalável e resiliente. Aqui, adotamos uma arquitetura distribuída, com cada serviço isolado em pods do Kubernetes, permitindo maior flexibilidade e controle. O uso do Kubernetes será fundamental para orquestrar esses pods, garantindo alta disponibilidade, gestão automática de recursos e escalabilidade. O Kubernetes permitirá que a infraestrutura se ajuste dinamicamente à carga de trabalho, com pods podendo ser replicados ou removidos quando necessário, dependendo da carga que o servidor tiver. Assim, é garantido o processo de auto scaling, permitindo que a aplicação seja capaz de lidar com variações de tráfego sem comprometer o desempenho. Para começar cada serviço consta sempre com pelo menos dois pods, para garantir alta disponibilidade e tolerância a falhas, de modo que, mesmo que um dos pods falhe ou esteja indisponível, o outro possa assumir a carga, evitando interrupções no serviço. Isso também facilita atualizações sem downtime, utilizando estratégias como rolling updates para substituir pods gradualmente.

Além disso, todos os serviços do PictuRAS são configurados como clusters de pods, o que permite que múltiplas instâncias de cada serviço sejam executadas de forma distribuída e escalável, estas são representadas pelas reticências no diagrama. No entanto, serviços como bases de dados, storage (MinIO/S3) e cache em Redis exigem configurações adicionais para funcionarem adequadamente como um único cluster. Existem duas opções para isso:

1. SaaS. Utilização do Redis e do MongoDB frequentemente fornecidos diretamente pelos principais provedores de serviços. Quanto ao serviço de storage, pode-se adotar o S3 da AWS ou as soluções da MinIO Inc., como o AIStor ou as extensões para os principais provedores de serviços.
2. (Solução retratada no diagrama) Serviços geridos pela equipa de desenvolvimento. Por se tratar de um processo complexo, e este não ser o objetivo deste documento, não foram detalhados a organização e os processos que seriam utilizados para esses componentes de forma muito concreta. Contudo, é de notar que esses serviços já fornecem ferramentas para esse fim através do Helm Charts (Package Manager para o Kubernetes), que devem ser utilizadas para configurar os mesmos no caso de deployment manual desses serviços.

O Helm, deve ser utilizado para facilitar o desenvolvimento para o cluster, fazendo uso do gestor de pacotes.

É de notar que os serviços de Redis e MongoDB no diagrama foram colocados em conjunto com o gateway e os demais microsserviços por não conterem dados confidenciais, enquanto o serviço de storage ficaria em outro cluster.

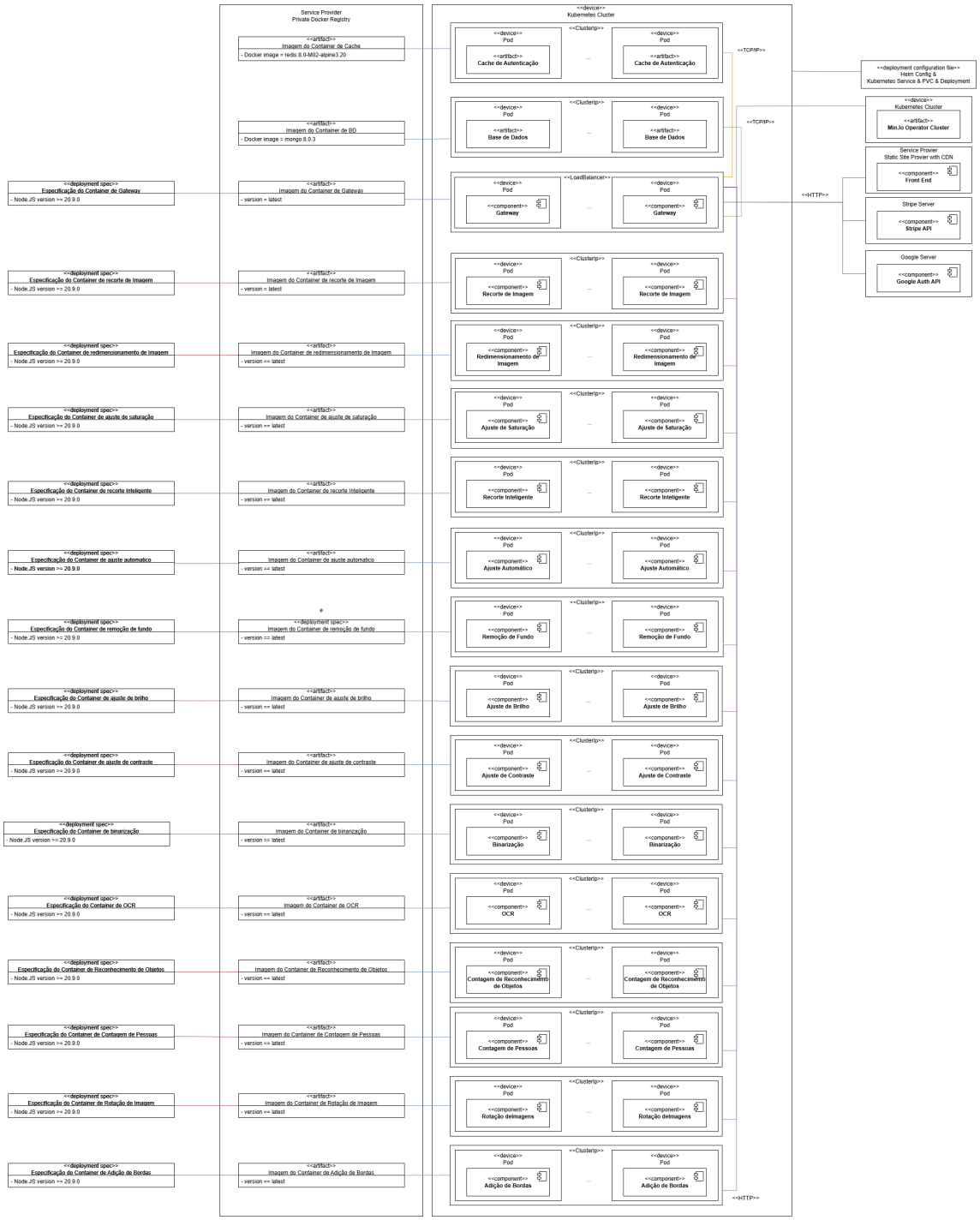


Figura 17: Diagrama de *deployment* (Produção)

8.2. Contexto de desenvolvimento e teste

No contexto de desenvolvimento, a configuração do sistema é simplificada para facilitar a prototipagem e testes rápidos, esta é baseada na arquitetura do produção sendo que apenas um pod corre e no caso do Min.Io é apenas utilizado um container docker. Para alem disso, um novo container é utilizado para permitir o Stripe enviar webhooks para o gateway e assim lidar com os eventos emitidos por este serviço. Desta forma, todos os componentes do PictuRAS, incluindo a aplicação, base de dados e serviços auxiliares, são executados numa única máquina. Essa abordagem permite um ciclo de desenvolvimento ágil e

eficiente, onde os desenvolvedores podem testar novas funcionalidades sem a complexidade de múltiplos ambientes. O diagrama de desenvolvimento reflete esse modelo monolítico, sendo ideal para simulações em pequena escala e para validar a lógica do sistema antes da migração para um ambiente de produção.

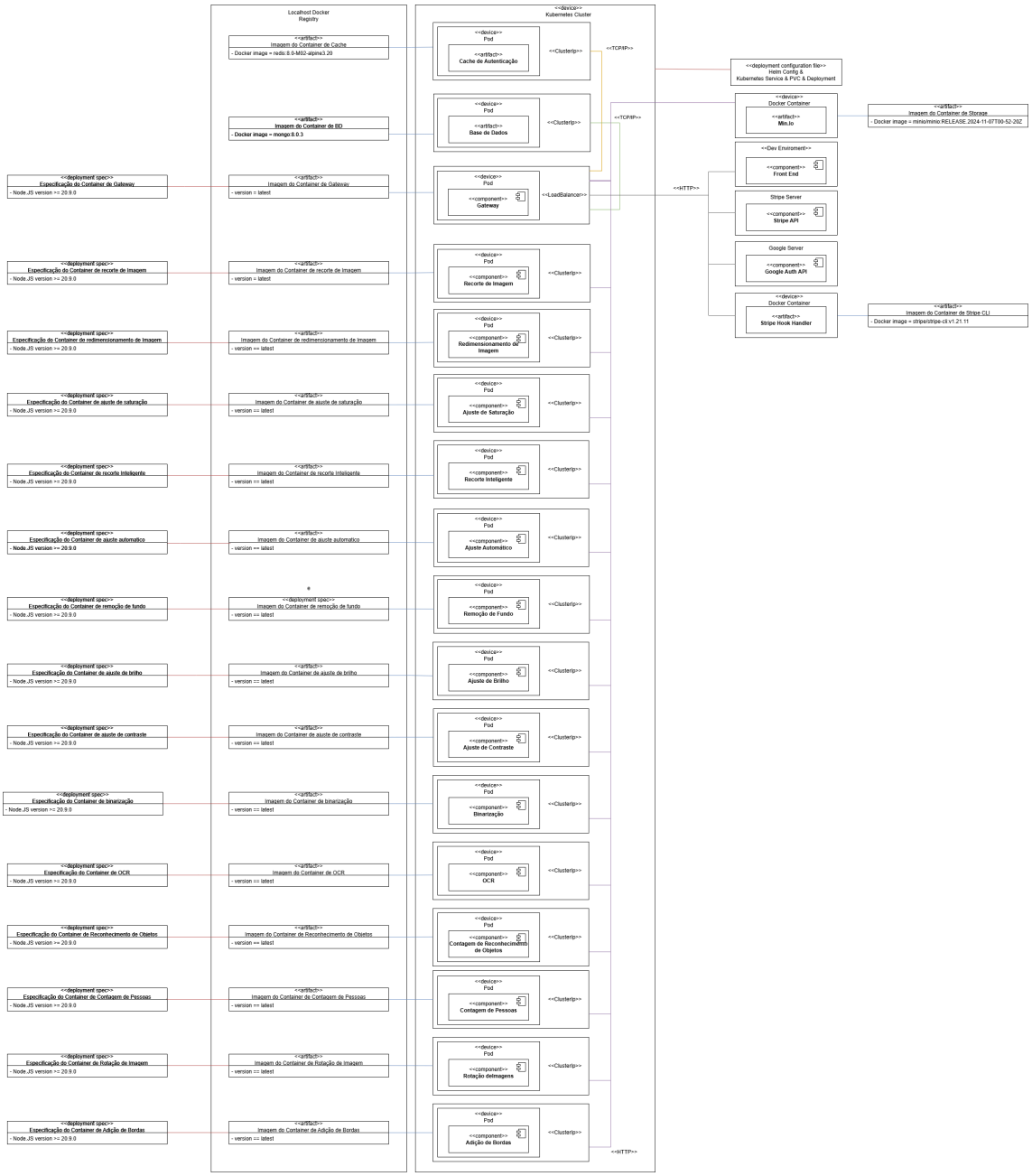


Figura 18: Diagrama de *deployment* (desenvolvedor)