INSTITUTO UNIVERSITÁRIO DE LISBOA

Mestrado em Engenharia Informática

Arquitetura e Desenho de Software

Autores:

Ana Carolina Vieira, Nº 106568

Ekaterina Popkova, Nº 134454

Jéssica Vieira, Nº 110812

Vasco Conde, Nº 133746

Grupo: Grupo_PosLaboral-A

Data: 20-12-2025

Título do trabalho: GitDash - Uma Aplicação Web para Visualização de Estatísticas de Repositórios GitHub

Docente: José Pereira dos Reis

Resumo

O presente relatório documenta a análise, arquitetura e desenvolvimento do GitDash, uma aplicação web concebida para atuar como ferramenta analítica para repositórios GitHub. O projeto visa colmatar a lacuna entre os registos técnicos de desenvolvimento e a gestão de equipas, transformando dados brutos, como *commits*, *pull requests* e *issues*, em visualizações intuitivas e métricas de produtividade acionáveis.

A solução assenta numa arquitetura distribuída e containerizada, utilizando Docker para garantir o isolamento e a consistência entre ambientes. O sistema adota uma separação clara de responsabilidades, combinando um *Frontend* reativo (SPA) desenvolvido em Vue.js com um *Backend* em .NET Core, estruturado em camadas lógicas segundo o padrão MVC "Headless".

Um dos desafios centrais abordados foi a gestão da latência e dos limites de utilização (*rate limits*) impostos pela API do GitHub. Para mitigar estes riscos, implementou-se uma estratégia de persistência híbrida que combina uma base de dados relacional (PostgreSQL) para a gestão de identidades com o uso de Redis para *caching*, assegurando a rapidez na visualização de dados e a fiabilidade do sistema.

O projeto foi executado seguindo a metodologia ágil (Scrum).

Palavras-chave: Arquitetura e Desenho de Software, Engenharia de Software, Visualização de Dados, GitHub API, .NET Core, C#, Vue.js, TypeScript, PostgreSQL, Redis.

Índice

Índice de Tabelas

Índice de Figuras

Glossário

DOM *Document Object Model*

DTO *Data Transfer Object*

FR *Functional Requirement*

HTML *HyperText Markup Language*

HU História de Utilizador

IA Inteligência Artificial

JSON JavaScript *Object Notation*

MVC *Model-View-Controller*

MVP *Minimum Viable Product*

OAuth *Open Authorization*

PDF *Portable Document Format*

PR *Pull Request*

REST *Representational State Transfer*

RF Requisito Funcional

RTM *Requirements Traceability Matrix*

SPA *Single Page Application*

SQL *Structured Query Language*

TTL *Time-To-Live*

UML *Unified Modeling Language*

URL *Uniform Resource Locator*

US *User Story*

1. Introdução

1.1. Contextualização e Visão do Projeto

Descrição da Aplicação: GitDash

O GitDash é uma aplicação web concebida para atuar como uma ferramenta analítica para repositórios GitHub. A sua principal finalidade é colmatar a lacuna entre os dados brutos de desenvolvimento e a gestão de equipas, transformando registos de atividade (*logs*, *commits* e metadados) em visualizações intuitivas e métricas acionáveis.

A aplicação integra-se diretamente com a API do GitHub, permitindo aos utilizadores autenticarem-se de forma segura via OAuth e acederem instantaneamente aos repositórios associados à sua conta. Ao selecionar um projeto, o GitDash agrega e processa dados históricos e recentes, oferecendo uma visão holística sobre o estado do código e a dinâmica da equipa.

O sistema distingue-se pela capacidade de apresentar não apenas metadados estáticos (como estrelas e *forks*), mas principalmente estatísticas dinâmicas de atividade, incluindo o fluxo de *commits*, *pull requests* e resolução de *issues*. O GitDash foca-se também na granularidade da informação, permitindo a análise de colaboradores individuais e métricas de qualidade de código, como o rácio de adição/eliminação de linhas.

Visão e Objetivos

A visão do projeto GitDash é tornar-se uma ferramenta essencial para *Team Leads* e *Developers*, facilitando a monitorização de projetos de software através de *dashboards* centralizados que eliminam a necessidade de navegação manual e dispersa pelo histórico do GitHub.

Para concretizar esta visão, o projeto estabelece os seguintes objetivos estratégicos, derivados diretamente dos requisitos funcionais:

- Centralização da Análise de Projetos: Permitir que o utilizador carregue e visualize repositórios públicos ou privados através de uma interface unificada, seja por seleção direta ou inserção de URL.
- Monitorização de Atividade em Tempo Real: Fornecer gráficos temporais e estatísticas atualizadas (e.g. *commits* semanais ou mensais) que permitam identificar tendências de produtividade ou estagnação no desenvolvimento.
- Avaliação de Desempenho da Equipa: Disponibilizar métricas claras sobre a contribuição de cada colaborador, detalhando o seu papel, volume de *commits* e impacto no código (linhas adicionadas *vs.* removidas).
- Controlo de Qualidade e Processos: Facilitar o rastreio do ciclo de vida do *software* através da análise de *Issues* (abertas/fechadas) e da eficiência dos *Pull Requests* (taxas de aprovação e fusão).
- Flexibilidade e Usabilidade: Garantir uma experiência de utilização fluida, permitindo a filtragem global de dados por intervalos de datas específicos (e.g. últimos 7 dias) e a navegação dinâmica entre visões gerais e detalhadas sem perda de contexto.

Em suma, o GitDash visa transformar dados técnicos em inteligência de gestão, suportado por uma arquitetura robusta que garante a segurança dos dados e a rapidez na visualização da informação.

1.2. Estrutura do Documento

Este relatório está organizado em cinco capítulos principais.

No Capítulo 1 é apresentada a contextualização do projeto, a visão da aplicação GitDash e a estrutura global do documento.

O Capítulo 2 descreve a análise e especificação de requisitos, incluindo os requisitos funcionais, as *epics* e respetivas *user stories*, bem como a matriz de rastreabilidade que relaciona requisitos e funcionalidades implementadas.

O Capítulo 3 centra-se na análise dos atributos de qualidade, recorrendo à *Utility Tree* e aos cenários arquiteturalmente significativos.

O Capítulo 4 apresenta a visão geral da arquitetura proposta, o estilo adotado e a justificação das principais decisões de desenho. Aborda ainda a *stack* tecnológica selecionada e a metodologia de desenvolvimento aplicada no projeto.

O Capítulo 5 aprofunda o desenho detalhado e a modelagem, através dos diagramas de casos de uso, diagrama Entidade-Relação do modelo de dados, diagramas UML estruturais e comportamentais, diagramas BPMN e descreve a estratégia de persistência e *caching* utilizada.

O Capítulo 6 descreve a estratégia de qualidade do projeto, focando-se nos testes de cobertura realizados tanto no *Frontend* como no *Backend*.

Por fim, o Capítulo 7 reúne as conclusões, sintetiza as decisões de *design*, identifica possíveis evoluções futuras do sistema e apresenta os anexos, incluindo ligações para o protótipo de interface, código-fonte e artefactos de suporte ao desenvolvimento.

2. Análise e Especificação de Requisitos

2.1. Requisitos Funcionais

Esta secção descreve os Requisitos Funcionais (RFs) do GitDash, detalhando as funcionalidades e comportamentos que o sistema deve oferecer para cumprir os seus objetivos.

RF1. O sistema deve permitir que os utilizadores se autentiquem através do GitHub OAuth.

RF2. O sistema deve listar os repositórios acessíveis a partir da conta GitHub do utilizador autenticado.

RF3. O sistema deve permitir que os utilizadores selecionem ou carreguem um repositório inserindo o seu URL ou escolhendo a partir de uma lista pendente (*dropdown*).

RF4. O sistema deve apresentar metadados do repositório, incluindo nome, descrição, visibilidade (público/privado), estrelas, *forks* e linguagens de programação utilizadas.

RF5. O sistema deve apresentar estatísticas de atividade tais como *commits*, *pull requests*, *issues* e revisões, com contagens totais e recentes (por exemplo, últimos 7 dias).

RF6. O sistema deve apresentar informações dos colaboradores (nome, nome de utilizador, função, *commits*, PRs, *issues*).

RF7. O sistema deve apresentar métricas de contribuição de código, incluindo linhas adicionadas, eliminadas, alterações totais e o rácio de adição/eliminação.

RF8. O sistema deve apresentar gráficos para a atividade de *commits* ao longo do tempo (por exemplo, *commits* semanais).

RF9. O sistema deve permitir a definição de um intervalo de datas (por exemplo, últimos 7 dias, mês passado).

RF10. O sistema deve apresentar *issues* com título, estado e etiquetas (*labels*).

RF11. O sistema deve apresentar métricas de *pull requests*, incluindo taxa de aprovação, taxa de fusão (merge) e detalhes dos PRs fundidos.

RF12. O sistema deve mostrar registos de atividade recente (por exemplo, *commits*, alterações de ficheiros e carimbos de data/hora).

RF13. O sistema deve permitir alternar entre as vistas de Visão Geral do Repositório, Colaboradores e Contribuidor Individual.

RF14. O sistema deve fornecer atualizações de dados após um pedido de mudança de repositórios ou intervalos de datas.

RF15. O sistema deve mostrar notificações ou mensagens toast para autenticação e carregamento de repositório bem-sucedidos.

2.2. *Epics* e *User Stories*

Épico 1: Autenticação e Acesso

- **HU1.** Como utilizador, quero iniciar sessão com a minha conta GitHub para poder aceder de forma segura aos meus repositórios e painéis (*dashboards*).
    - Critérios de Aceitação:
        - Quando clico em "Iniciar sessão com o GitHub", o ecrã do GitHub OAuth abre-se.
        - Após a autorização, sou redirecionado para a página de seleção de repositório.
        - Uma mensagem de sucesso confirma a minha autenticação.
        - Um início de sessão de demonstração simula a autenticação e carrega dados de repositório fictícios (*mock*).

Épico 2: Gestão de Repositórios

- **HU2.** Como utilizador autenticado, quero ver os meus repositórios acessíveis e selecionar um para analisar.
    - Critérios de Aceitação:
        - A lista de repositórios mostra o nome, descrição e visibilidade (público/privado).
        - Posso carregar um repositório colando o seu URL ou selecionando a partir da lista pendente.
        - Após o carregamento, sou redirecionado para o painel de visão geral do repositório.
- **HU3.** Como utilizador, quero ver metadados (estrelas, *forks*, visibilidade) para o repositório selecionado.
    - Critérios de Aceitação:

        - A parte superior do painel apresenta os detalhes e métricas do repositório.

Épico 3: Visualização e Análise

- **HU4.** Como utilizador, quero ver estatísticas do repositório (*commits*, PRs, *issues*, revisões) durante um período selecionado.
    - Critérios de Aceitação:
        - Cada cartão mostra totais e tendências para o intervalo de datas escolhido.
        - A atividade de *commits* aparece num formato de gráfico (semanal/mensal).
- **HU5.** Como utilizador, quero ver dados de contribuição para cada colaborador.
    - Critérios de Aceitação:
        - Os colaboradores são listados com nome, função e métricas (*commits*, PRs, *issues*).
        - Clicar num colaborador filtra o painel para mostrar apenas os seus dados.
- **HU6.** Como utilizador, quero ver métricas detalhadas do repositório, incluindo adições, eliminações e rácios.
    - Critérios de Aceitação:
        - Uma barra de progresso ou gráfico visualiza as adições versus eliminações.
        - O total de alterações e o rácio adição/eliminação são apresentados numericamente.
- **HU7.** Como utilizador, quero acompanhar a atividade de *issues* e PRs.
    - Critérios de Aceitação:
        - A secção de *Issues* lista as *issues* abertas/fechadas com etiquetas.
        - A secção de *Pull Requests* mostra a taxa de aprovação, taxa de fusão e detalhes dos PRs fundidos.
- **HU8.** Como utilizador, quero ver a atividade recente (últimos *commits* ou alterações de ficheiros).
    - Critérios de Aceitação:
        - Uma secção de "Atividade Recente" lista as atualizações recentes com mensagens de *commit* e carimbos de data/hora.

Épico 4: Filtragem e Interação

- **HU9.** Como utilizador, quero definir um intervalo de datas (por exemplo, últimos 7 dias) para poder analisar períodos específicos.
    - Critérios de Aceitação:

        - O filtro de data aplica-se globalmente em todos os gráficos, cartões e tabelas.

- **HU10.** Como utilizador, quero alternar entre a visão geral do repositório, a vista de colaborador e separadores de contribuidores específicos.
    - Critérios de Aceitação:
        - Cada separador atualiza o conteúdo dinamicamente sem recarregamentos de página.

2.3. Matriz de Rastreabilidade de Requisitos (RTM)

A Tabela 1 mapeia os Requisitos Funcionais às Histórias de Utilizador (HU):

Tabela 1 - Matriz de Rastreabilidade de Requisitos (RTM).

ID do Requisito	ID(s) da História de Utilizador	Descrição
RF1	HU1	Permitir que os utilizadores se autentiquem via GitHub OAuth e simular autenticação de demonstração para testes.
RF2	HU2	Listar repositórios acessíveis a partir da conta GitHub autenticada.
RF3	HU2	Permitir que os utilizadores selecionem ou carreguem um repositório inserindo o seu URL ou escolhendo a partir de uma lista pendente.
RF4	HU3	Apresentar metadados do repositório tais como nome, descrição, visibilidade, estrelas e *forks*.
RF5	HU4	Apresentar estatísticas de atividade (*commits*, *pull requests*, *issues*, revisões) com contagens totais e recentes. [53]
RF6	HU5	Apresentar informações do colaborador (nome, nome de utilizador, função, *commits*, PRs, *issues*).
RF7	HU5	Apresentar métricas de contribuição de código (linhas adicionadas, eliminadas, alterações totais, rácio adição/eliminação).
RF8	HU4	Apresentar gráficos visuais para atividade de *commits* ao longo do tempo (por exemplo, semanalmente ou mensalmente).
RF9	HU9	Fornecer filtragem por intervalo de datas (por exemplo, últimos 7 dias, mês passado).
RF10	HU7	Apresentar *issues* com título, estado e etiquetas.
RF11	HU7	Apresentar métricas de *pull request*, incluindo taxa de aprovação, taxa de fusão e detalhes de PRs fundidos.
RF12	HU8	Mostrar registos de atividade recente tais como *commits*, alterações de ficheiros e carimbos de data/hora.
RF13	HU10	Permitir alternar entre vistas de Visão Geral do Repositório, Colaboradores e Contribuidor Individual.
RF14	HU4, HU9	Fornecer atualizações de dados em tempo real ou a pedido ao mudar de repositórios ou intervalos de datas.

RF15	HU1, HU2	Mostrar notificações ou mensagens *toast* para autenticação e carregamento de repositório bem-sucedidos.

3. Análise dos Atributos de Qualidade

Após a definição dos Requisitos Funcionais e das Histórias de Utilizador, que estabelecem o âmbito funcional do sistema, é imperativo analisar os requisitos não funcionais. Estes requisitos impõem restrições técnicas que influenciam diretamente as decisões de desenho do sistema.

Dada a natureza da aplicação, que depende da integração com uma API externa (GitHub) para apresentar metadados, estatísticas e gráficos em tempo real ou a pedido, identificaram-se desafios críticos relacionados com a latência de rede e limites de utilização (*rate limits*). Adicionalmente, a necessidade de consistência entre ambientes de desenvolvimento e produção impõe restrições de *deployment*.

Para priorizar estes desafios, utilizou-se o método da Árvore de Utilidade (*Utility Tree*). Este método permite derivar Requisitos Arquiteturalmente Significativos (ASRs - *Architecturally Significant Requirements*) e classificá-los com base em duas dimensões:

- Valor para o Negócio (Alto (A)/Médio (M)/Baixo (B)): Criticidade da funcionalidade para o utilizador final.
- Impacto na Arquitetura (A/M/B): Dificuldade técnica ou esforço estrutural necessário para a implementação.

3.1. Árvore de Utilidade (*Utility Tree*) e ASRs

A Tabela 2 formaliza os cenários identificados, mapeando os requisitos funcionais para atributos de qualidade concretos.

Tabela 2 - Requisitos Arquiteturais Significativos (ASRs) e respetiva análise de impacto.

ID	Atributo	Cenário (ASR)	Valor Negócio	Impacto Arq.	Racional (Baseado nos Requisitos)
ASR 1	**Desempenho**	O sistema deve atualizar gráficos e métricas em < 2 segundos ao alterar filtros de data.	**A**	**A**	O FR14 exige atualizações "*on-demand*". A latência da API do GitHub tornaria o sistema lento sem uma estratégia de *caching*.
ASR 2	**Fiabilidade**	O sistema deve gerir quotas de pedidos para evitar bloqueios por *rate limit* da API do GitHub.	**A**	**A**	O acesso contínuo a estatísticas detalhadas por múltiplos utilizadores esgotaria rapidamente a quota de pedidos diretos.
ASR 3	**Usabilidade**	A alternância entre abas (Visão Geral, Colaboradores, *Issues*) deve ser fluida e sem recarregamento da página.	**A**	**M**	A US10 e o FR13 especificam que a troca de vistas deve atualizar o conteúdo dinamicamente.
ASR 4	**Segurança**	A autenticação deve ser delegada via	**A**	**M**	O FR1 exige explicitamente

		OAuth, sem persistência de credenciais sensíveis no sistema.			autenticação via GitHub OAuth, transferindo a responsabilidade da gestão de identidade.
ASR 5	**Mantenabilidade**	As regras de cálculo de métricas (e.g. rácio add/del) devem estar isoladas da lógica de acesso a dados.	**M**	**M**	O cálculo de métricas complexas (FR7) pode sofrer ajustes frequentes, exigindo desacoplamento de código.
ASR 6	**Auditoria**	O sistema deve manter um histórico de atividades e repositórios carregados.	**M**	**B**	Suporta o requisito de mostrar *logs* de atividade recente (FR12).
ASR 7	**Portabilidade**	O sistema deve garantir consistência de execução em diferentes ambientes (dev/prod) e facilitar a instalação de dependências.	**M**	**A**	Elimina erros de configuração ("*It works on my machine*") ao lidar com múltiplos serviços (BD, Cache).
ASR 8	**Escalabilidade**	O sistema deve permitir o aumento da capacidade de processamento horizontalmente para suportar picos de utilizadores.	**M**	**A**	Garante que o sistema suporta o crescimento da base de utilizadores sem reescrita da arquitetura base.

4. Desenho da Arquitetura de Software

4.1. Visão Geral e Diagrama de Arquitetura

Com base na análise da *Utility Tree* (Tabela 2), propõe-se uma arquitetura desenhada para mitigar os riscos de Desempenho e Fiabilidade, garantindo a Escalabilidade necessária.

O Estilo Arquitetural adotado é híbrido e robusto, combinando três abordagens complementares:

- Microserviços (Infraestrutura): A solução é decomposta em contentores independentes (Docker), permitindo a gestão isolada de cada serviço.
- Arquitetura em Camadas (*Backend*): O serviço principal é estruturado internamente em camadas lógicas para organização de código.
- MVC "Headless" (Padrão de Desenho): Aplica-se o padrão *Model-View-Controller*, onde o *Backend* implementa o *Model* e o *Controller*, mas exclui a *View* ("MVC sem o V"), delegando a apresentação inteiramente para o cliente (*Frontend*).

Embora esta combinação de estilos possa sugerir uma aproximação a arquiteturas de microserviços, é importante clarificar que o GitDash não implementa microserviços no sentido estrito. Os contentores Docker garantem isolamento ao nível da infraestrutura, mas a aplicação

mantém uma dependência centralizada, sobretudo na base de dados, e o backend funciona como um único serviço lógico. Assim, se um componente crítico falhar, como a base de dados, toda a aplicação fica comprometida, o que não corresponde aos princípios de autonomia e resiliência característicos de uma arquitetura de microserviços. Por este motivo, o sistema deve ser entendido como uma arquitetura distribuída (*service based + space based*) e containerizada, beneficiando de portabilidade e separação física, mas não como uma arquitetura de microserviços pura. Esta distinção é relevante para enquadrar corretamente o estilo arquitetural adotado.

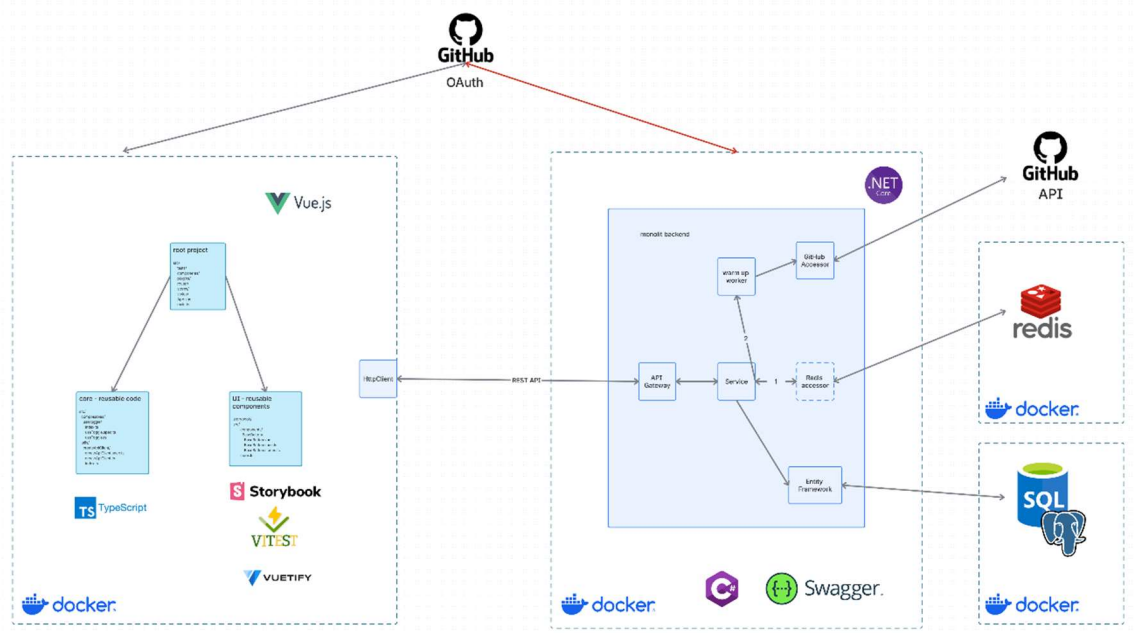

Figura 1 - Arquitetura geral do GitDash.

4.2. Estilo e Justificação das decisões tomadas

Abaixo detalha-se como cada componente e decisão arquitetural responde aos Requisitos (ASRs) e princípios de engenharia de software.

1. Estratégia de *Cache* com Redis (Resposta ao ASR 1 e ASR 2)

Para satisfazer a exigência de atualizações rápidas ao mudar intervalos de datas e evitar a dependência direta da API do GitHub a cada clique, propõe-se a utilização de um sistema de cache baseado em Redis.

- **Papel:** Atuar como *buffer* de alto desempenho em memória.
- **Justificação:** O Redis permite armazenar as respostas JSON do GitHub com um TTL (*Time-to-Live*) configurável, reduzindo a latência e protegendo a quota da API.

2. *Frontend* Reativo: Separar a "View" (Resposta ao ASR 3)

Para cumprir o requisito de interatividade fluida, optou-se por separar fisicamente a camada de apresentação (*View*) do resto do sistema, implementando uma *Single Page Application* (SPA) em Vue.js.

- **Papel:** Gestão de estado no cliente e renderização dinâmica.
- **Justificação:**

- **Desacoplamento da Interface:** Ao separar o *Frontend*, garantimos que a interface pode evoluir ou ser completamente substituída (e.g. mudar para uma *app* móvel) sem qualquer impacto na lógica do servidor.
- **Interatividade:** O Vue.js permite componentes que reagem instantaneamente, proporcionando a experiência exigida na US10.

3. *Backend* em Camadas e o Princípio de Baixo Acoplamento (Resposta ao ASR 5)

O núcleo lógico do sistema é implementado em .NET Core, organizado numa Arquitetura em Camadas (*Controllers, Services, Data Access*), seguindo o padrão MVC mas servindo apenas dados (API REST).

- **Papel:** Orquestração de pedidos, lógica de negócio e segurança.
- **Justificação:**
  - **MVC (sem o V):** O *backend* foca-se puramente no processamento (*Controller*) e nos dados (*Model*), não desperdiçando recursos a renderizar HTML (*View*).
  - **Baixo Acoplamento e Camadas:** A decisão de separar as responsabilidades em camadas visa garantir o baixo acoplamento. Como ditam as boas práticas, "*as camadas podem ser trocadas sem mexer no restante*". Isto significa que podemos, por exemplo, alterar a tecnologia de base de dados na camada de *Data Access* sem ter de reescrever uma única linha de código na camada de *Controllers* ou na interface.

4. Persistência de Auditoria com PostgreSQL (Resposta ao ASR 6)

Para os dados que necessitam de persistência a longo prazo, como os *logs* de atividade e repositórios consultados, utiliza-se o PostgreSQL.

- **Papel:** Armazenamento relacional de *logs* e metadados de sessão.
- **Justificação:** Garante a durabilidade dos registos de acesso e consulta histórica fiável.

5. Orquestração com Docker: Abordagem de "Microserviços" (Resposta ao ASR 7 e ASR 8)

Todo o sistema será executado em contentores isolados, emulando uma arquitetura de "microserviços".

- **Papel:** Unidade padrão de *deployment*, isolamento e escalabilidade horizontal.
- **Justificação:**
  - **Escalabilidade Horizontal (ASR 8):** Permite lançar múltiplas réplicas do contentor de serviço para absorver picos de carga.
  - **Consistência (ASR 7):** Garante ambientes idênticos em desenvolvimento e produção.

4.3. *Stack* Tecnológica

Como as principais escolhas tecnológicas estruturais (Vue.js, .NET Core, Redis, PostgreSQL e Docker) já foram detalhadas e justificadas na secção anterior, apresenta-se aqui a visão consolidada de todas as ferramentas e *frameworks* que compõem o ecossistema do GitDash, organizadas por componente:

***Frontend* (Cliente)**

- **Vue.js:** *Framework* principal (*conforme referido no ponto 2 da secção 4.2*).

- **TypeScript:** Linguagem de programação que adiciona tipagem estática ao JavaScript e serve como linguagem base para o Vue.js.
- **Vuetify:** Biblioteca de componentes de UI (Material Design).
- **Vitest & Storybook:** Ferramentas para testes unitários e desenvolvimento isolado de componentes, respetivamente.

***Backend* (Servidor)**

- **.NET Core:** *Framework* para API REST (*conforme referido no ponto 3 da secção 4.2*).
- **C#:** Linguagem de programação moderna, orientada a objetos e multiplataforma que serve como base para o desenvolvimento de aplicações na plataforma .NET Core.
- **Swagger:** Utilizado para documentação automática dos *endpoints* da API.

Dados e Persistência

- **Redis:** Cache em memória para otimização de pedidos (*conforme referido no ponto 1 da secção 4.2*).
- **PostgreSQL:** Base de dados relacional para auditoria (*conforme referido no ponto 4 da secção 4.2*).

Infraestrutura

- **Docker:** Contentorização de todos os serviços (*conforme referido no ponto 5 da secção 4.2*).

4.4. Metodologia de desenvolvimento

Para a gestão e desenvolvimento do projeto, a equipa adotou a metodologia ágil SCRUM, uma abordagem iterativa e incremental que permite uma maior flexibilidade na resposta a requisitos e um controlo contínuo sobre o progresso das tarefas. O ciclo de vida do projeto foi estruturado em quatro sprint distintos, cujo planeamento temporal foi ajustado estrategicamente para cumprir os marcos de entrega definidos pela unidade curricular.

As duas primeiras iterações tiveram uma duração de duas semanas cada. Esta cadência mais curta foi definida intencionalmente para acelerar a entrega de um Produto Mínimo Viável (MVP) antes da apresentação intercalar, garantindo assim que as funcionalidades base estavam implementadas e testadas a tempo deste marco avaliativo. Posteriormente, as duas últimas iterações foram estendidas para três semanas, permitindo um aprofundamento no desenvolvimento de funcionalidades mais complexas e na refinação da qualidade do código.

Para suportar esta metodologia, foram realizadas reuniões semanais de planeamento e revisão (*Sprint Planning* e *Review*), bem como pontos de situação breves. A gestão do *backlog* e o acompanhamento das tarefas (*User Stories*) foram realizados através do GitHub *Projects*, garantindo a transparência do estado do projeto para todos os membros (ver Figura 2).

Relativamente à organização dos recursos humanos, a equipa foi segmentada em duas áreas funcionais: *Frontend* e *Backend*. Esta divisão não foi arbitrária, mas sim baseada numa análise das competências técnicas e apetências individuais de cada membro. Os elementos com maior experiência ou facilidade em tecnologias de interface e experiência de utilizador assumiram o desenvolvimento do *Frontend*, enquanto os membros com maior robustez em lógica de negócio, arquitetura de dados e APIs focaram-se no *Backend*. Esta estratégia de especialização permitiu otimizar a velocidade de desenvolvimento e garantir a qualidade técnica em ambas as vertentes da aplicação.

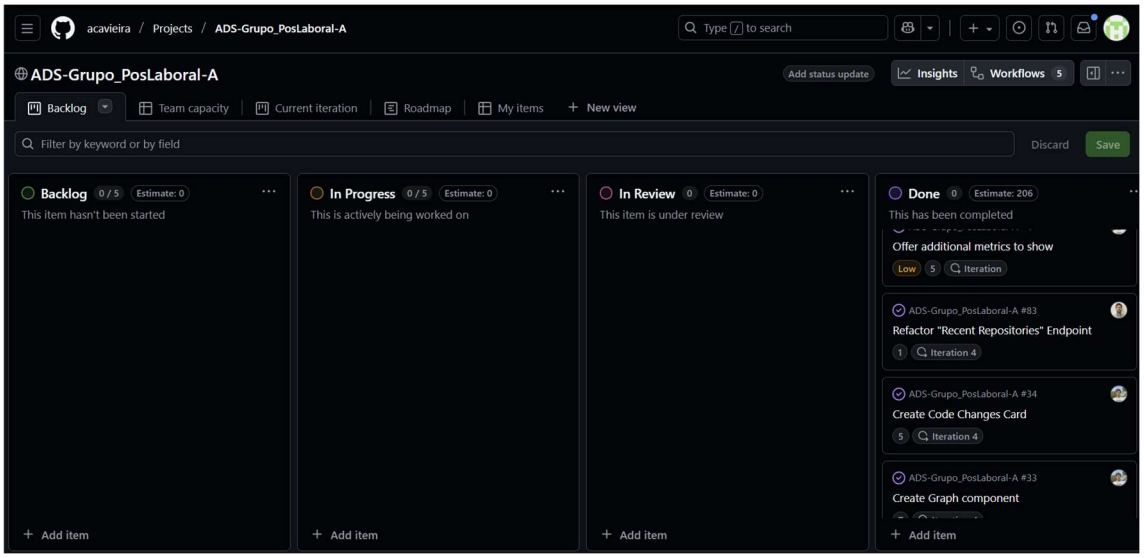

Figura 2 - Painel para gestão e acompanhamento de tarefas no GitHub Projects.

A Figura 3 ilustra um exemplo representativo de um cartão de tarefa (*card*) utilizado na gestão do projeto. A estrutura de cada item foi desenhada para facilitar o planeamento, incluindo metadados essenciais como a iteração (*Sprint*) de alocação, o nível de prioridade e a estimativa de esforço, a qual foi calculada seguindo a escala da sequência de Fibonacci.

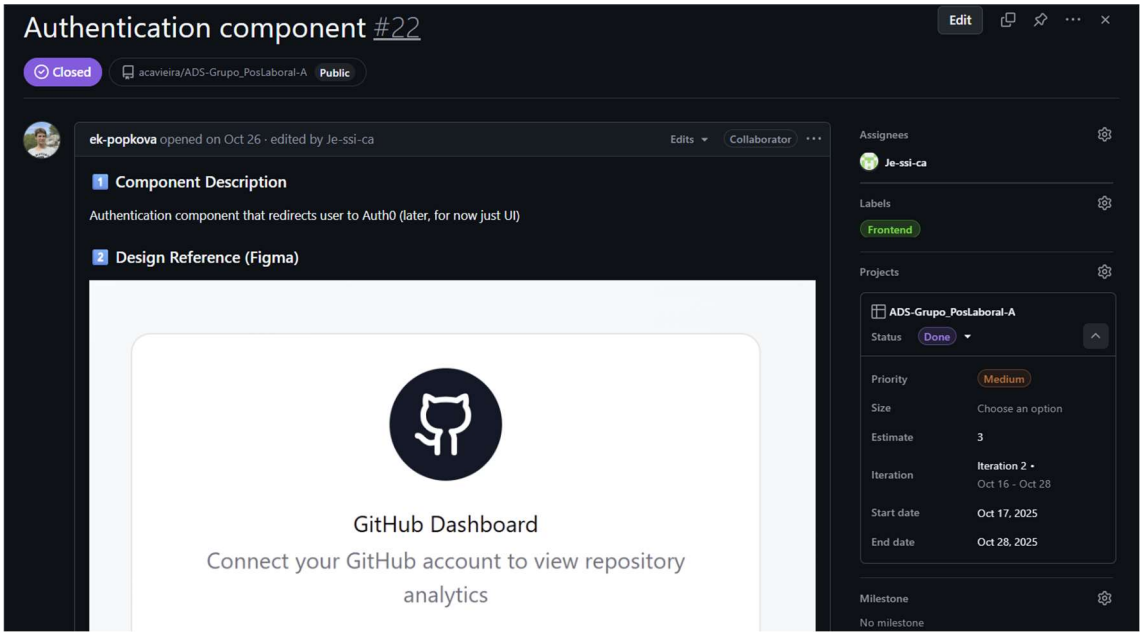

Figura 3 - Exemplo de uma tarefa do projeto.

4.5 Arquitetura do *Backend*: Estrutura e Interação das Camadas

O *backend* do projeto está estruturado segundo uma arquitetura em camadas inspirada no padrão MVC, com o desacoplamento total da camada de apresentação, que é delegada ao *frontend*. Neste modelo, o *backend* assume as responsabilidades de *Model* e *Controller*, concentrando-se na lógica de negócio, integração de dados e exposição de *endpoints* (REST). A comunicação com a API do GitHub, o Postgres e o Redis está integrada de forma independente, garantindo que estas componentes estejam desacopladas da lógica de negócio e dos *endpoints* expostos pela aplicação. Esta combinação garante um desenho modular, facilmente extensível e coerente com as boas práticas de separação de responsabilidades.

Esta organização em camadas é central para garantir o cumprimento dos atributos de qualidade definidos. A separação clara de responsabilidades permite atingir a mantenibilidade (ASR 5)

através do isolamento de código, facilitando simultaneamente a escalabilidade (ASR 8) e a portabilidade (ASR 7) do sistema.

A estrutura do *backend* encontra-se assim distribuída pelas seguintes componentes (conforme Figura 4):

1. *Controllers*

- Localização: GitDashBackend/Controllers/
- Responsabilidade: Funcionam como ponto de entrada dos pedidos HTTP. Recebem pedidos do *frontend*, validam parâmetros e encaminham as operações para os serviços apropriados.
- *Controllers*: GitHubController.cs, AuthController.cs, DbController.cs.

2. *Services*

- Localização: GitDashBackend/Services/
- Responsabilidade: Implementam a lógica de negócio da aplicação. Coordenam o fluxo entre *controllers*, *accessors* e outros serviços, aplicando as regras de processamento e agregação de dados.
- Exemplos: GitHubService.cs, DbService.cs.

3. *Accessors*

- Localização: GitDashBackend/Accessors/
- Responsabilidade: Abstraem o acesso a serviços externos, como a API do GitHub, a cache Redis ou a base de dados. Implementam interfaces que promovem baixo acoplamento e facilitam testes.
- Accesors: GitHubAccessor.cs, RedisAccessor.cs.

4. *Domain / DTOs*

- Localização: GitDashBackend/Domain/DTOs/
- Responsabilidade: Definem os objetos de transferência de dados utilizados entre camadas, assegurando consistência nos formatos e contratos.
- Exemplos: RepositoryDto.cs, CommitDto.cs, CollaboratorDto.cs.

5. *Data*

- Localização: GitDashBackend/Data/
- Responsabilidade: Implementa o contexto da base de dados através do Entity Framework, mapeando entidades persistidas e suportando operações de leitura e escrita.
- AppDbContext.cs.

6. *Models*

- Localização: GitDashBackend/Models/
- Responsabilidade: Representam entidades de domínio que são armazenadas na base de dados.
- Entidades: User.cs, Repository.cs, Log.cs.

7. *Middlewares*

- Localização: GitDashBackend/Middlewares/
- Responsabilidade: Intercetar e processar requisições e respostas, adicionando funcionalidades transversais como logging ou auditoria.
- AccessLogMiddleware.cs.

8. *Configurations*

- Localização: GitDashBackend/Configurations/
- Responsabilidade: Centralizam a configuração de dependências, serviços e definições utilizadas no arranque e funcionamento da aplicação.
- DependencyInjection.cs, GitHubSettings.cs.

4.5.1. Fluxo de Interação Entre Camadas

O funcionamento do *backend* segue um fluxo simples e bem estruturado:

1. O utilizador envia um pedido HTTP para a API.
2. O *controller* correspondente recebe o pedido, valida os dados e encaminha-o para o serviço apropriado.
3. O serviço executa a lógica de negócio, consultando a cache, a API do GitHub ou a base de dados consoante necessário.
4. Os *accessors* tratam da comunicação com sistemas externos, devolvendo os dados ao serviço.
5. O serviço reúne e processa a informação, devolvendo-a ao *controller*.
6. O *controller* constrói a resposta HTTP final e envia-a ao cliente.
7. *Middlewares* podem intervir ao longo do processo, registando os acessos ou aplicando comportamentos transversais.

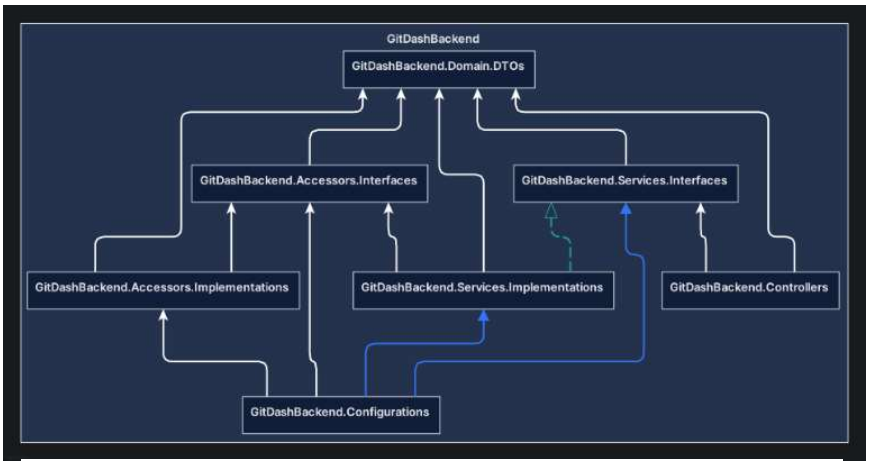

Figura 4 - Diagrama de arquitetura em camadas do Backend.

4.6 Arquitetura do *Frontend*

O componente de apresentação (*Frontend*) foi desenhado seguindo o paradigma de uma *Single Page Application* (SPA), construída sobre a *framework* Vue.js em conjunto com TypeScript. Esta escolha tecnológica, aliada a uma organização estrutural rigorosa, visa responder aos requisitos de interatividade e manutenibilidade (ASR 3 e ASR 5), garantindo uma separação clara entre a interface visual, a lógica de estado e a comunicação com a API.

A arquitetura segue um modelo Baseado em Componentes, onde a interface é decomposta em pequenas unidades reutilizáveis e testáveis. A estrutura de diretórios do código-fonte (src) reflete esta modularidade, organizando-se pelas seguintes responsabilidades funcionais:

1. Core e Configuração

- Ficheiros: App.vue, main.ts.
- Responsabilidade: O main.ts atua como ponto de entrada da aplicação, inicializando a instância Vue e os *plugins* globais. O App.vue serve como componente raiz (*root*) onde a estrutura base do *layout* é definida. O ficheiro config.ts centraliza variáveis globais de configuração do ambiente.

2. *Pages*

- Localização: GitDash/src/pages
- Responsabilidade: Representam as "vistas" ou "ecrãs" completos da aplicação (e.g. *Dashboard*, *Login*, Detalhe do Repositório). Estes componentes atuam como orquestradores, reunindo vários componentes menores para formar uma página funcional mapeada a uma rota específica.

3. *Components* (Interface Reutilizável)

- Localização: GitDash/src/components/
- Responsabilidade: Contém os componentes visuais específicos do projeto. Os componentes identificados como altamente reutilizáveis foram movidos para uma biblioteca externa independente (*Package UI*). Esta separação foi implementada para promover uma rigorosa Separação de Responsabilidades e maximizar a reutilização de código, mantendo no diretório local apenas os elementos específicos do domínio da aplicação.

4. *Composables* (Lógica Reutilizável)

- Localização: GitDash/src/composables/
- Responsabilidade: Tirar partido da *Composition* API do Vue 3 para encapsular e reutilizar a lógica de estado ou comportamento (e.g. lógica de *fetching* de dados, manipulação de datas, gestão de temas). Estes módulos orquestram a comunicação com o *Backend* delegando as chamadas de rede ao cliente de API centralizado localizado no pacote externo (*Package Core*), garantindo que a lógica de aplicação permanece desacoplada dos detalhes de implementação HTTP. Permite extrair a complexidade dos componentes visuais (.vue) para funções TypeScript puras e testáveis.

5. *Stores* (Gestão de Estado)

- Localização: GitDash/src/stores/
- Responsabilidade: Implementar o padrão de gestão de estado centralizado (tipicamente via Pinia). Armazenar dados globais da aplicação, como a lista de repositórios em *cache*, garantindo que o estado é previsível e acessível por qualquer componente.

6. Router (Navegação)

- Localização: GitDash/src/router/
- Responsabilidade: Gerir a navegação do lado do cliente (*Client-Side Routing*). Define o mapeamento entre os URLs do browser e os componentes da pasta *pages*, gerindo

também "*Guards*" de navegação para proteção de rotas (e.g. impedir acesso a não autenticados).

7. *Models* (Tipagem de Dados)

- Localização: GitDash/src/models/
- Responsabilidade: Centralizar as interfaces e tipos TypeScript. Assegurar a consistência dos contratos de dados em todo o *frontend*, espelhando os DTOs definidos no *Backend* (e.g. *Repository*, *Commit*, *User*).

8. Qualidade e Testes

- Localização: Co-localizados nos diretórios de lógica e componentes (e.g. src/composables).
- Responsabilidade: Garantir a robustez e a fiabilidade da aplicação. Utiliza-se o *framework* Vitest para a execução de testes unitários, focados estritamente na validação isolada da lógica de negócio, manipulação de estado (*Stores*) e casos de uso (*Composables*). A estratégia de co-localização dos ficheiros de teste (.spec.ts) junto ao código-fonte assegura que a validação é parte integrante do ciclo de desenvolvimento de cada funcionalidade.

9. Documentação e Suporte ao Desenvolvimento

- Localização: Ficheiros .stories.ts junto aos componentes visuais; documentação técnica gerada a partir do código.
- Responsabilidade: Manter uma base de conhecimento acessível e atualizada, separada por domínios:
    - Visual (UI): O Storybook atua como documentação viva e ambiente de desenvolvimento isolado ("*sandbox*") para os componentes, permitindo visualizar estados e variações visuais sem depender da lógica da aplicação.
    - Lógica: O DocType é utilizado para documentar a lógica de negócio (*Composables* e *Stores*), gerando referências técnicas detalhadas diretamente a partir das tipagens e comentários do código.

10. Estilos e Recursos Estáticos

- Pastas: GitDash/src/styles, public.
- Responsabilidade: *styles* contém as folhas de estilo globais (CSS/SCSS) e definições de tema. A pasta *public* armazena recursos estáticos (imagens, favicons) servidos diretamente.

4.6.1. Fluxo de Dados no *Frontend*

A interação dentro desta arquitetura segue um fluxo unidirecional para garantir a previsibilidade:

1. Interação: O utilizador interage com um Componente Visual (do Projeto ou do *Package UI*).
2. Orquestração: O componente apenas notifica a Página (*Smart Component*) através de eventos. A Página então invoca uma ação num *Store* ou utiliza um *Composable*.
3. Lógica/API: A lógica de negócio (no *Composable/Store*) utiliza o *Package Core* para comunicar com o *Backend* (via REST/HTTP).
4. Estado: O estado global (Pinia) ou local é atualizado reativamente.
5. Renderização: O Vue deteta a alteração de estado e atualiza automaticamente a *View* (DOM) para refletir os novos dados.

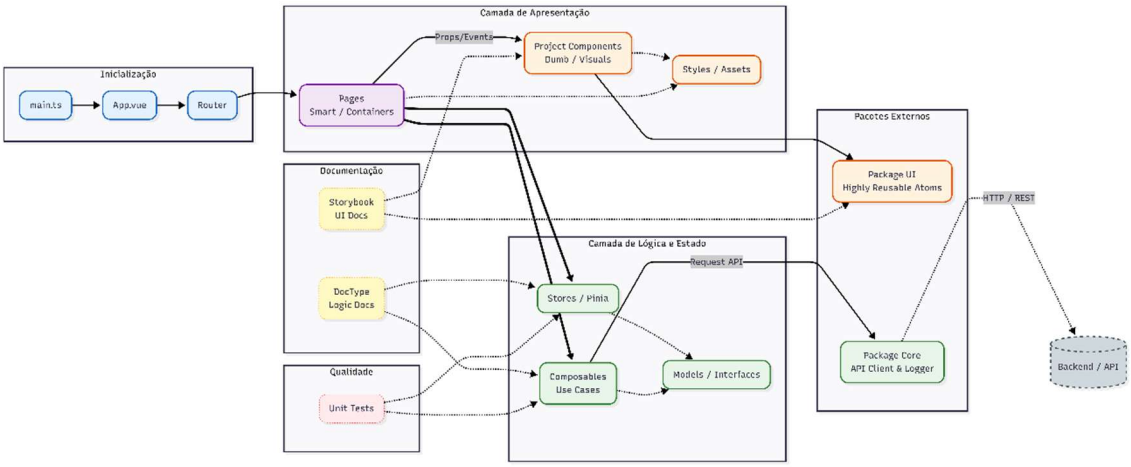

Figura 5 - Arquitetura do Frontend

5. Desenho Detalhado e Modelagem

Neste capítulo, apresenta-se o desenho detalhado da solução GitDash, materializando as decisões arquiteturais previamente definidas em modelos concretos de implementação. O objetivo é fornecer uma visão aprofundada sobre a estrutura e o comportamento do sistema, recorrendo a normas de representação padrão da indústria, como a UML (*Unified Modeling Language*) e o BPMN (*Business Process Model and Notation*).

Ao longo das secções seguintes, são detalhadas as interações do utilizador com o sistema através de Diagramas de Casos de Uso, bem como a lógica interna de processamento de dados ilustrada nos Diagramas de Sequência e Atividade. Adicionalmente, explora-se o fluxo transversal de informação e a estratégia híbrida de persistência, que combina SQL e Redis, essencial para garantir o cumprimento dos requisitos de desempenho e a gestão eficiente dos limites da API do GitHub

5.1. Diagramas de Casos de Uso

A Figura 6 contém uma representação gráfica dos principais casos de uso, sendo estes a autenticação, a visualização de estatísticas relativas a um repositório e as estatísticas relativas a um colaborador específico.

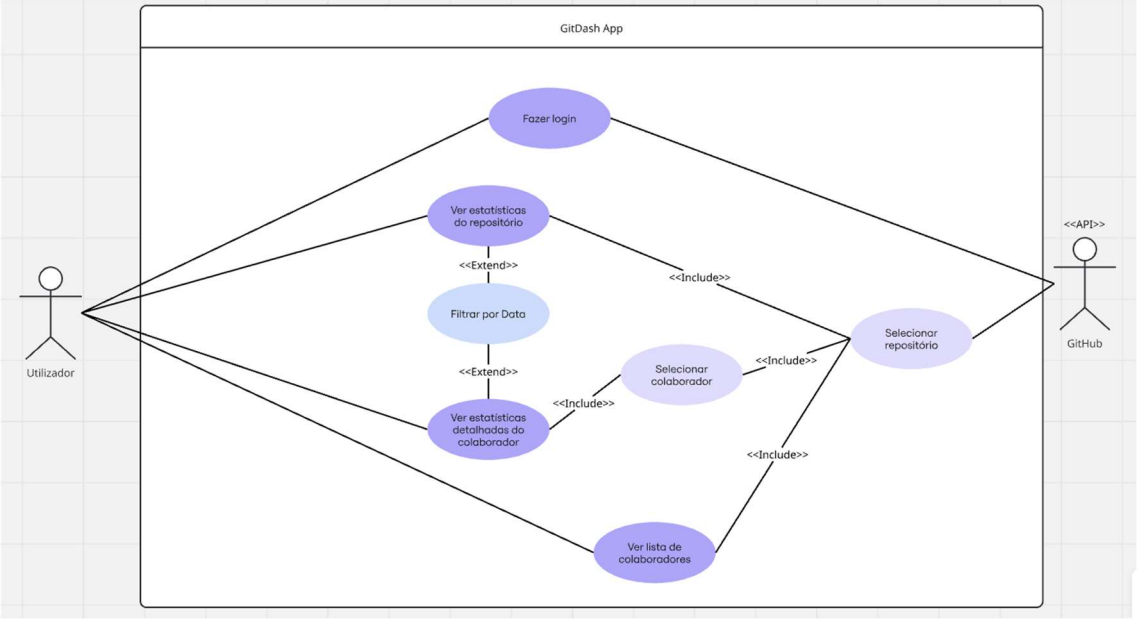

Figura 6 - Diagrama UML de Casos de Uso da aplicação GitDash.

5.2. Diagrama Entidade-Relação – Modelo de Dados

A Figura 7 ilustra o esquema da base de dados relacional (SQL) implementada no projeto através de um diagrama de Entidade-Relação. Este modelo foi desenhado com um âmbito específico: assegurar a persistência dos perfis de utilizador e o respetivo histórico de sessões (*endpoints* acedidos e repositórios consultados). Dado que a carga de dados operacionais complexos é delegada no Redis, a estrutura relacional mantém-se intencionalmente leve e concisa, contendo apenas as entidades essenciais para a gestão de identidades.

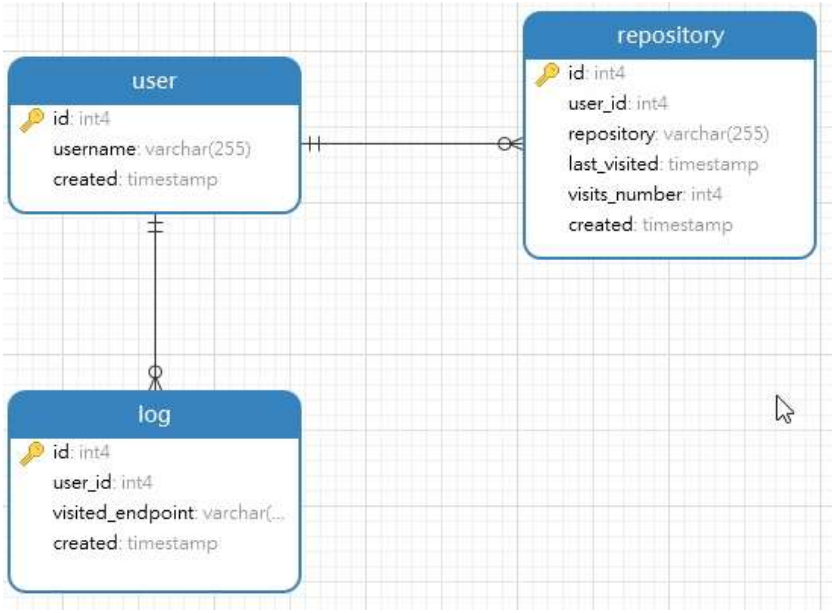

Figura 7 - Diagrama Entidade-Relação do Modelo de Dados Relacional.

Abaixo descrevem-se as entidades representadas no Diagrama Entidade-Relação:

1. Entidade *user* (Utilizador) Representa a entidade central do modelo. Embora a autenticação seja delegada no GitHub, a aplicação necessita de manter um registo local mínimo dos utilizadores que acederam à plataforma para garantir a integridade referencial dos *logs* e históricos de repositórios consultados.

- Atributos: Identificador único (*id*), nome de utilizador (*username*) e data de criação do registo (*created*), correspondente à primeira visita do utilizador à aplicação.

2. Entidade *log* (Auditoria) Responsável por satisfazer os requisitos de auditoria e segurança. Esta tabela contém todas as interações relevantes do utilizador com os *endpoints* da API do backend.

- Relacionamento: Possui uma relação de 1 para N com a tabela *user* (um utilizador gera múltiplos *logs*).
- Atributos: Identificador único (*id*), referência ao utilizador através de chave estrangeira (*user_id*), o *endpoint* acedido (*visited_endpoint*) e o data e hora da ação (*created*).

3. Entidade *repository* (Histórico de Repositórios Visitados) Armazena o histórico de repositórios consultados por cada utilizador, permitindo funcionalidades como "vistos recentemente" ou estatísticas de uso pessoal.

- Relacionamento: Possui uma relação de 1 para N com a tabela *user* (um utilizador visita múltiplos repositórios).
- Atributos: Identificador único (*id*), referência ao utilizador através de chave estrangeira (*user_id*), nome do repositório (*repository*), data e hora da última visita (*last_visited*), contador de acessos (*visits_number*), permitindo ordenar os repositórios por relevância ou frequência de uso, e data e hora da criação do registo (*created*), que corresponde à primeira visita do utilizador ao repositório.

Considerações de Design: Este modelo foi desenhado para ser eficiente e evitar erros de organização. Ele segue regras de otimização (conhecidas como 3ª Forma Normal) que garantem que nenhuma informação é repetida desnecessariamente, por exemplo, os dados de um utilizador são guardados apenas num local e não em todas as tabelas.

Para manter a segurança e a coerência, o sistema utiliza ligações obrigatórias entre as tabelas, através de chaves estrangeiras. Isto significa que é impossível criar um registo de *log* ou de repositório sem que este esteja associado a um utilizador já registado. Assim, evitamos dados "orfãos" ou informações perdidas na base de dados.

5.3. Diagramas UML Comportamentais

A Figura 8 apresenta o diagrama de sequência do sistema, detalhando o fluxo de autenticação do utilizador e o processo de obtenção de dados via API do GitHub. É importante destacar que o padrão de interação aqui ilustrado, que prioriza a verificação de dados em *cache* antes de efetuar pedidos externos, é transversal a todas as funcionalidades da aplicação. Desta forma, assegura-se a eficiência do sistema em todas as consultas, desde a visão geral do repositório até às estatísticas detalhadas de colaboradores individuais.

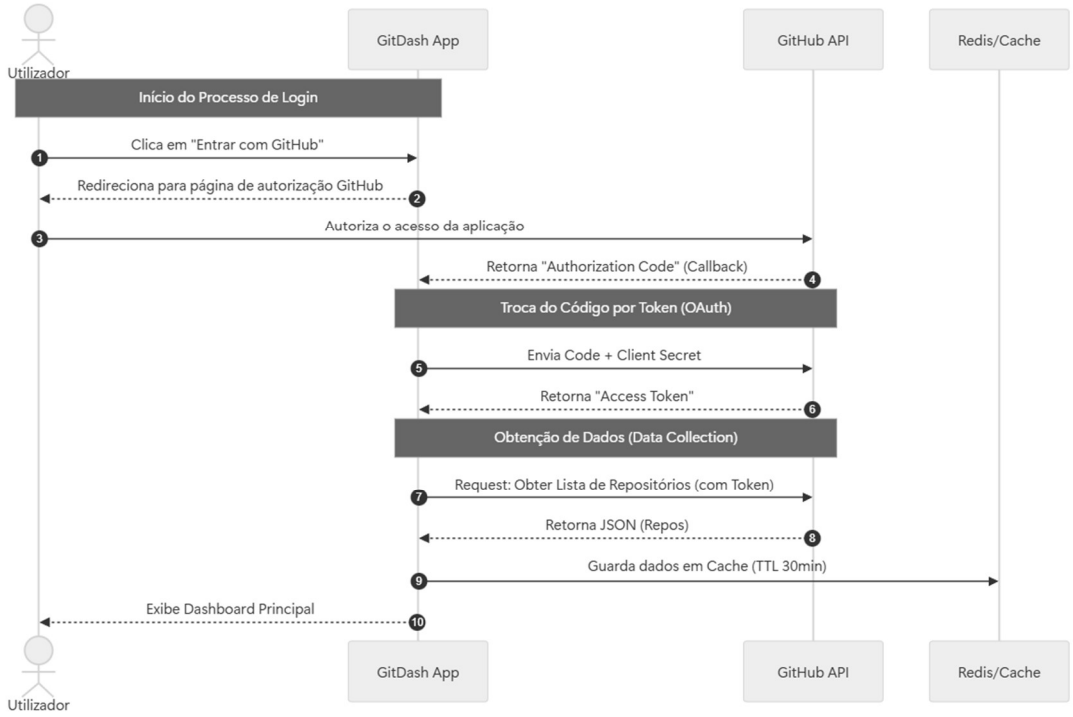

Figura 8 - Diagrama de Sequência

A Figura 9 apresenta o Diagrama de Atividades para atualização dos dados, aplicável tanto à seleção de repositórios quanto de colaboradores. O processo verifica se os dados estão em *cache* e atualizados (dentro do TTL). Caso contrário, realiza-se uma chamada à API do GitHub, atualiza-se a *cache* e os dados são exibidos ao utilizador.

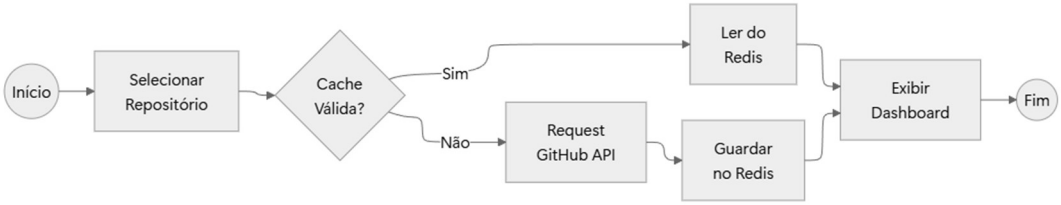

Figura 9 - Diagrama de atividade

5.4. Diagramas BPMN

A Figura 10 ilustra, através de notação BPMN, o fluxo lógico transversal de obtenção e atualização de dados no sistema. Este modelo operacional é aplicável tanto ao contexto global dos repositórios como à análise granular de estatísticas de utilizador.

O processo inicia-se com a autenticação via OAuth, resultando na obtenção de um *token* de acesso junto da API externa. Subsequentemente, o sistema verifica a existência do utilizador na base de dados local. Num cenário de primeiro acesso (inexistência de registo), é criada automaticamente uma entrada de utilizador associada ao *token* obtido. Após esta etapa, o utilizador é redirecionado para a interface principal. Nesta página, a seleção de um repositório pode ser efetuada através de múltiplos mecanismos: seleção via *dropdown*, inserção direta de identificador ou consulta do histórico recente.

Quanto à obtenção de métricas (seja ao carregar um repositório, selecionar um colaborador ou alterar o intervalo temporal), o sistema privilegia a verificação da existência e validade dos dados em *cache* (Redis). O recurso à API do GitHub ocorre estritamente em situações de *cache miss* ou quando os dados se encontram expirados, otimizando assim o desempenho da aplicação.

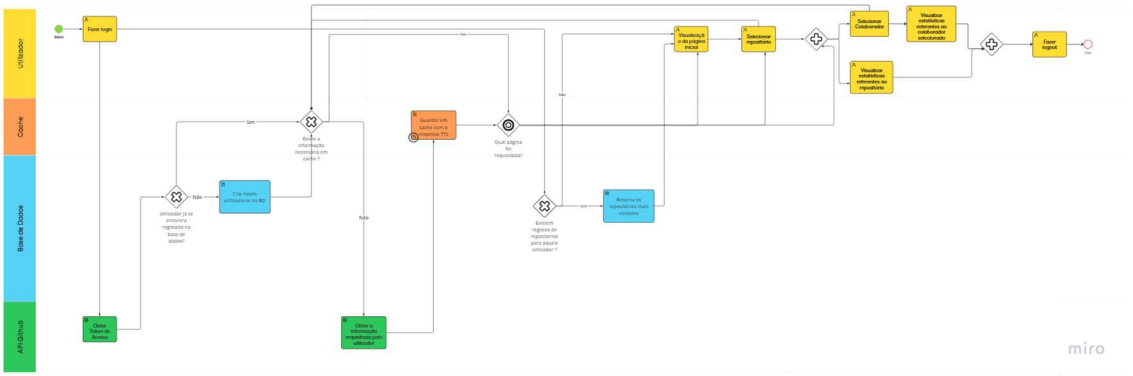

Figura 10 - Diagrama BPMN do fluxo de obtenção e atualização de dados

5.5. Estratégia de Persistência e *Caching*

A estratégia híbrida adotada na aplicação combina o uso de SQL para armazenar dados transacionais, como autenticação e perfis de utilizador, com o uso de Redis para gerir dados operacionais obtidos da API do GitHub. Esta separação permite garantir organização, persistência e suporte às funcionalidades internas, ao mesmo tempo que melhora o desempenho na apresentação de dados externos.

Os dados obtidos através da API do GitHub, como estatísticas de *commits*, atividade de *issues*, *pull requests* e métricas de contribuição, apresentam elevada volatilidade e são consultados com frequência durante a utilização da aplicação. Consultá-los diretamente em cada interação aumentaria a latência e poderia levar ao esgotamento dos limites de utilização da API. Para mitigar estes impactos, o GitDash utiliza o Redis como mecanismo de *cache*, aplicando um período de validade de cerca de 30 minutos (TTL) para manter o equilíbrio entre atualidade e eficiência.

O sistema segue uma abordagem *cache aside*: sempre que é necessária informação, o *backend* verifica primeiro se esta já se encontra disponível no Redis e, se estiver, devolve-a de imediato. Quando não está disponível, os dados são recolhidos da API do GitHub, processados e guardados no Redis para futuras consultas. Esta estratégia assegura tempos de resposta mais rápidos, uma navegação fluida entre *dashboards* e uma redução significativa do número de pedidos feitos à API externa.

6. Qualidade

Esta secção tem como objetivo mostrar as métricas utilizadas para avaliar a qualidade do código.

6.1. Testes de cobertura do *Frontend*

All files

95.8% Statements 1992/1985 **84.1%** Branches 307/365 **84.61%** Functions 99/164 **95.8%** Lines 1992/1891

Press *n* or *j* to go to the next uncovered block, *b*, *p* or *k* for the previous block.

Filter:

File	Statements		Branches		Functions		Lines	
packages/core/src/utils/api	88.81%	389/438	61.11%	77/126	79.48%	31/39	88.81%	389/438
packages/core/src/utils/logger	100%	31/31	92.3%	12/13	100%	8/8	100%	31/31
packages/ui/src/components/BaseButton	100%	13/13	66.66%	2/3	100%	1/1	100%	13/13
packages/ui/src/components/BaseCard	100%	9/9	100%	1/1	100%	0/0	100%	9/9
packages/ui/src/components/DataErrorAlert	100%	21/21	100%	4/4	100%	1/1	100%	21/21
packages/ui/src/components/StatKpiCard	100%	19/19	100%	0/0	100%	0/0	100%	19/19
packages/ui/src/components/Tabs	100%	38/38	100%	8/8	100%	3/3	100%	38/38
packages/ui/src/components/UrlInputCard	100%	32/32	100%	6/6	100%	2/2	100%	32/32
src/components/BaseLayout	100%	75/75	100%	9/9	100%	6/6	100%	75/75
src/components/CodeChangesCard	100%	93/93	100%	16/16	100%	1/1	100%	93/93
src/components/Collaborators	97.89%	93/95	86.66%	13/15	100%	2/2	97.89%	93/95
src/components/HeaderMenu	100%	60/60	88.88%	8/9	100%	3/3	100%	60/60
src/components/LoginCard	100%	44/44	100%	9/9	100%	1/1	100%	44/44
src/components/OpenWorkCard	100%	24/24	100%	4/4	100%	0/0	100%	24/24
src/components/PeakActivityCard	100%	43/43	100%	5/5	100%	0/0	100%	43/43
src/components/RepositoriesList	100%	15/15	100%	3/3	100%	1/1	100%	15/15
src/components/RepositoryCard	100%	62/62	95%	19/20	100%	1/1	100%	62/62
src/components/RepositoryOverviewCard	100%	20/20	100%	4/4	100%	0/0	100%	20/20
src/components/StatisticGraph	100%	62/62	100%	3/3	0%	0/1	100%	62/62
src/components/UserRepositoriesCard	100%	11/11	100%	0/0	100%	0/0	100%	11/11
src/composables	100%	317/317	97.4%	75/77	100%	20/20	100%	317/317
src/pages	97.09%	301/310	96.15%	25/26	100%	4/4	97.09%	301/310
src/stores	61.22%	30/49	100%	4/4	30%	3/10	61.22%	30/49

Figura 11 - Relatório de cobertura de testes do frontend.

No âmbito da validação do *frontend*, foi desenvolvida uma extensa bateria de testes unitários utilizando as ferramentas Vitest e Vue Test Utils. Esta suite é composta por 30 ficheiros de teste, totalizando 165 testes executados. A estratégia de validação foi abrangente, cobrindo não apenas o código da aplicação principal (“src”), mas também os packages de suporte partilhados (“packages/core” e “packages/ui”), garantindo a integridade transversal de toda a solução.

A arquitetura de testes focou-se em três pilares fundamentais: a interface do utilizador (*Components*), a orquestração de vistas (*Pages*) e a lógica de negócio encapsulada (*Composables*). Para os componentes visuais, foram criados testes que validam a renderização correta do DOM, a reatividade às props e a emissão de eventos. Nas páginas, a abordagem privilegiou o uso de *stubs* para isolar a estrutura dos componentes filhos, permitindo testar eficazmente os estados de carregamento e erro. Relativamente aos *Composables*, que contêm a lógica crítica de interação com a API, os testes garantiram a validação de fluxos assíncronos e tratamento de exceções.

Com a execução dos testes, foi gerado um relatório de cobertura detalhado, permitindo uma análise granular por diretoria. Esta análise revela a eficácia das estratégias de *mocking* implementadas, nomeadamente a simulação das *stores* do Pinia e do cliente de API, o que permitiu testar a lógica em total isolamento.

O relatório completo, visível na figura apresentada, demonstra resultados de excelência. A cobertura global de instruções (*Statements*) atingiu os 95.8%, um valor que reflete a robustez da implementação. Destaca-se, em particular, a diretoria “src/composables” e a grande maioria dos componentes UI, que atingiram 100% de cobertura, garantindo que toda a lógica de manipulação de dados e regras de negócio foi devidamente exercitada. As páginas (“src/pages”) apresentam também uma cobertura muito elevada (~97%), validando a correta integração entre o router e os componentes visuais.

A secção referente às *stores* (“src/stores”) apresenta uma cobertura ligeiramente inferior (~61%), o que se justifica pela estratégia adotada: como a lógica pesada foi abstraída para os *composables* e as *stores* foram frequentemente simuladas (*mocked*) nos testes de integração, o

código direto das *stores* serviu essencialmente como contentor de estado, sendo menos exercitado isoladamente.

Em síntese, a elevada taxa de cobertura obtida no *frontend*, tanto no núcleo da aplicação como nas bibliotecas auxiliares, assegura um grau de confiança muito alto na estabilidade do projeto. Ao contrário do *backend*, onde a complexidade limitou a cobertura, no *frontend* foi possível validar a quase totalidade dos cenários de utilização, incluindo casos de erro e limites (*edge cases*), estabelecendo uma base de qualidade sólida e fiável para a evolução do produto.

6.2. Testes de cobertura do *Backend*

No âmbito da validação do *backend* foram desenvolvidas várias classes de testes unitários dedicadas aos componentes mais críticos da aplicação. A prioridade foi dada aos serviços core, nomeadamente o DbService, responsável pela lógica de persistência, e partes essenciais do GitHubService, cuja função integra dados externos com a camada interna da aplicação. Adicionalmente, foram criados testes específicos para os controladores mais relevantes, como o GitHubController, garantindo a verificação de cenários base, validações iniciais e respostas esperadas perante entradas inválidas. Embora não tenham sido abrangidos todos os *endpoints*, os testes implementados asseguram a verificação dos fluxos principais, constituindo uma fundação sólida para futuras extensões de cobertura.

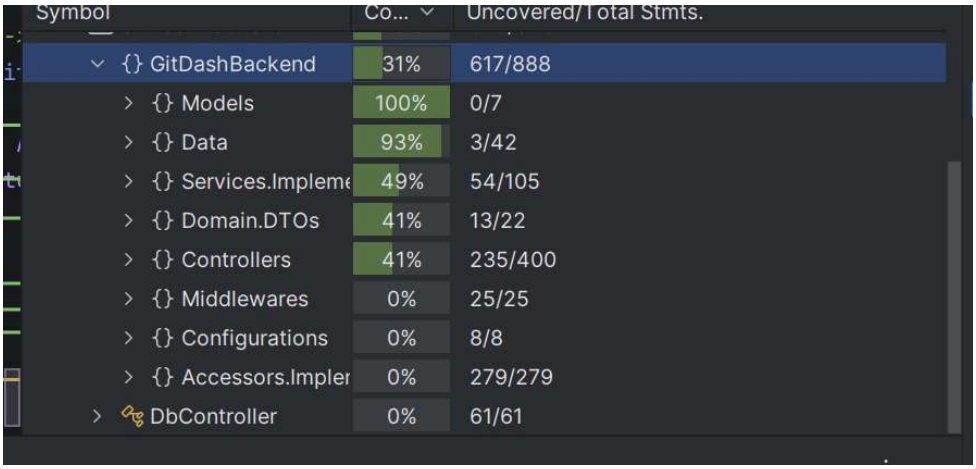

Figura 12 - Cobertura por namespace/classe (Rider).

Com as classes de teste criadas, foi gerado um relatório de cobertura utilizando o ReportGenerator, permitindo uma análise detalhada das secções do código efetivamente exercitadas pelos testes. O relatório apresenta dados por *assembly*, *namespace*, classe e método, facilitando a identificação de áreas com boa cobertura e de zonas ainda não testadas. Esta visão estruturada revelou não apenas a distribuição da cobertura, mas também a complexidade relativa de cada componente, nomeadamente os controladores com múltiplos caminhos lógicos.

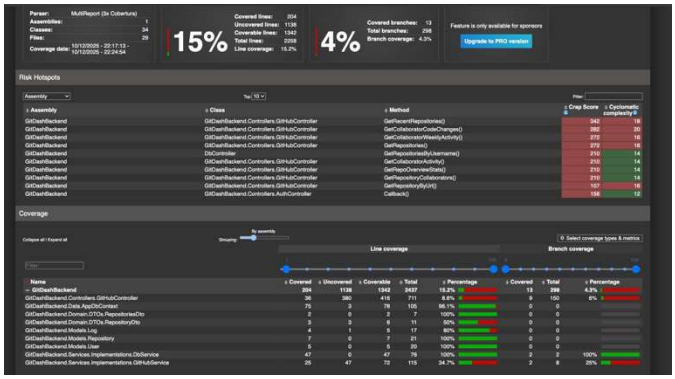

Figura 13 - Relatório HTML do ReportGenerator.

O relatório completo de cobertura de testes, incluindo o índice, estatísticas detalhadas e visualização das linhas exercitadas, encontra-se disponível na pasta *coveragereport/index.html*.

A cobertura global obtida foi de aproximadamente 31%, valor que reflete tanto a dimensão dos controladores como a sua complexidade intrínseca, que inclui dependências de autenticação, interações assíncronas com a API do GitHub e diferentes percursos de execução condicionados por variáveis externas. A limitação de tempo disponível não permitiu expandir os testes unitários a todos os *endpoints* nem replicar, para cada caso, os distintos comportamentos possíveis. Ainda assim, a camada de serviços e os modelos atingiram níveis muito elevados de cobertura, validando a base estrutural da aplicação.

Em síntese, embora a cobertura total ainda não represente o universo completo de funcionalidades do *backend*, o trabalho realizado garante que os componentes mais relevantes e suscetíveis de impacto no funcionamento geral estão testados. O próximo passo natural consistiria em ampliar a cobertura dos controladores, simulando diferentes contextos de autenticação, múltiplos tipos de erros externos e ramificações condicionais, permitindo aproximar a cobertura de valores superiores e reforçar a robustez global da solução.

7. Conclusões e Anexos

O presente relatório documentou o ciclo completo de conceção, análise e desenvolvimento da aplicação web GitDash, uma solução de *software* orientada para a análise visual e monitorização de repositórios GitHub.

7.1. Síntese do Trabalho Desenvolvido

A análise realizada permitiu concluir que os objetivos estratégicos propostos no início do projeto foram atingidos com sucesso. A implementação da autenticação via GitHub OAuth e a centralização da análise de projetos numa interface unificada permitiram simplificar o acesso à informação, eliminando a dispersão característica da navegação tradicional no GitHub.

Do ponto de vista funcional, o sistema cumpriu os requisitos de granularidade exigidos, oferecendo tanto uma visão holística do estado do código (*issues*, *pull requests*, *commits*) como uma análise detalhada da performance individual dos colaboradores. A introdução de métricas de qualidade, como a fração de adição/eliminação de linhas, fornece aos utilizadores ferramentas analíticas que ultrapassam as funcionalidades nativas da plataforma de origem.

7.2. Validação das Decisões Arquiteturais

A arquitetura de *software* adotada revelou-se fundamental para o sucesso do projeto, respondendo eficazmente aos Requisitos Não Funcionais e aos desafios identificados na *Utility Tree*:

1. Desempenho e Fiabilidade: A estratégia de persistência híbrida foi a decisão técnica mais impactante. A utilização do Redis como mecanismo de *caching* (com TTL de 30 minutos) permitiu mitigar a latência de rede e gerir de forma eficiente os *rate limits* da API do GitHub. Esta abordagem garantiu uma navegação fluida e protegeu o sistema contra bloqueios externos, validando a resposta aos atributos de qualidade de Desempenho (ASR 1) e Fiabilidade (ASR 2).

2. Modularidade e Manutenibilidade: A separação física entre o *Frontend* (Vue.js) e o *Backend* (MVC "Headless" em .NET Core) assegurou o desacoplamento necessário para que ambas as camadas pudessem evoluir independentemente. A organização do *Backend* em camadas (*Controllers, Services, Accessors*) facilitou a implementação da lógica de negócio e a testabilidade do código.

3. Metodologia Ágil: A adoção da metodologia SCRUM, com a divisão do projeto em iterações incrementais e a segmentação da equipa por competências técnicas, permitiu uma adaptação contínua aos requisitos e uma entrega de valor constante, culminando num produto robusto dentro dos prazos académicos estabelecidos.

7.3. Limitações e trabalhos futuros

Apesar de o GitDash cumprir os objetivos a que se propôs, o desenvolvimento de *software* é um processo contínuo. Identificaram-se áreas onde a aplicação pode evoluir para oferecer ainda mais valor aos *Team Leads* e *Developers*:

- Análise Preditiva com IA: Integração de algoritmos de *Machine Learning* para prever tendências de atraso na entrega de tarefas ou identificar riscos de qualidade no código com base no histórico de *commits* e *issues*.

- Monitorização de CI/CD: Expansão do dashboard para incluir estatísticas sobre pipelines de integração contínua (e.g. GitHub Actions), permitindo visualizar taxas de sucesso/falha de *builds* e tempos de *deployment*.

- Implementação de Redundância e Alta Disponibilidade: Atualmente, apesar da utilização de contentores, a arquitetura possui pontos únicos de falha onde a interrupção de um serviço pode comprometer todo o sistema. Futuramente, propõe-se a evolução para uma configuração distribuída com replicação de serviços e bases de dados (modelo *Primary-Replica*), garantindo que o sistema permanece operacional mesmo em caso de falha de uma das instâncias.
- Sistema de Notificações e Alertas: Implementação de alertas configuráveis (via email ou Slack) para eventos críticos, como a estagnação de *Pull Requests* por mais de 'X' dias ou picos anormais de eliminação de código.
- Exportação de Relatórios: Desenvolvimento de uma funcionalidade para exportar as visualizações e métricas em formatos portáteis (PDF ou CSV), facilitando a partilha de resultados em reuniões de gestão externas à plataforma.

Em suma, o GitDash posiciona-se como uma ferramenta sólida e escalável, capaz de transformar dados técnicos em inteligência de gestão, apoiada por uma arquitetura moderna que privilegia a performance e a experiência do utilizador.

7.4. Anexos

Abaixo apresentam-se os *links* úteis para este projeto. É também importante referir que no próprio projeto do GitHub, na secção dos *meetings*, existe um *card* com a nomenclatura de "*Important links"* onde se encontram mais links úteis para a organização e desenvolvimento do projeto.

Protótipo da Interface (Link para o Figma: https://melon-touch-19395380.figma.site/)

Código-Fonte do Projeto (Link para o repositório do GitHub: https://github.com/acavieira/ADS-Grupo_PosLaboral-A/tree/main)

SCRUM (Link para o projeto do GitHub: https://github.com/users/acavieira/projects/3)