UNIVERSIDADE
LUSÓFONA

QAlytics: Manutenção de testes com IA

Trabalho Final de Curso

Relatório Intercalar 1º Semestre

Diogo Correia
Ricardo Santos
José Brás
Luís Gomes
www.ulusofona.pt

Direitos de cópia

Resumo

O desenvolvimento de software moderno depende fortemente de testes automatizados. Contudo, à medida que os projetos escalam, a introdução de novos casos de teste em *suites* extensas torna-se um desafio complexo, resultando frequentemente em redundâncias lógicas, contradições com requisitos de negócio e regressões não intencionais.

Este projeto propõe o **QAlytics: Manutenção de testes com IA**, uma ferramenta CLI que utiliza Inteligência Artificial (LLMs) aliada a uma arquitetura de *Retrieval-Augmented Generation* (RAG) para analisar semanticamente o impacto da inserção de novos testes. A solução atua como um validador preventivo, cruzando a intenção de um novo teste com a base de conhecimento existente (User Stories e Casos de Teste antigos) para aprovar, bloquear ou sugerir refatorização. A ferramenta foca-se na precisão e explicabilidade, garantindo que cada veredicto gerado pela IA é acompanhado de uma justificação clara em linguagem natural.

O protótipo a ser desenvolvido (MVP) valida a viabilidade técnica desta abordagem preventiva, demonstrando a capacidade de detetar duplicações semânticas e conflitos com regras de negócio de forma automatizada, mitigando a dívida técnica e otimizando os processos de Garantia de Qualidade (QA).

Abstract

Modern software development relies heavily on automated testing. However, as projects scale, introducing new test cases into extensive suites becomes a complex challenge, often resulting in logical redundancies, contradictions with business requirements, and unintentional regressions.

This project proposes **QAlytics: AI-driven test maintenance**, a CLI tool that uses Artificial Intelligence (LLMs) combined with a Retrieval-Augmented Generation (RAG) architecture to semantically analyze the impact of inserting new tests. The solution acts as a preventive validator, cross-referencing the intention of a new test with the existing knowledge base (User Stories and previous Test Cases) to approve, block, or suggest refactoring. The tool focuses on precision and explainability, ensuring that each AI-generated verdict is accompanied by a clear justification in natural language.

The developed prototype (MVP) validates the technical feasibility of this preventive approach, demonstrating the ability to automatically detect semantic duplications and conflicts with business rules, thereby mitigating technical debt and optimizing Quality Assurance (QA) processes.

Índice

Lista de Figuras

Lista de Tabelas

1 - Introdução

Neste capítulo apresenta-se o enquadramento do projeto, a motivação e identificação do problema a resolver, os objetivos gerais e específicos, e a estrutura organizacional do documento.

1.1 Enquadramento Empresarial e Confidencialidade

O presente trabalho é desenvolvido em ambiente empresarial em parceria com a CGI. Devido à natureza sensível dos dados e processos envolvidos, o projeto encontra-se sob um acordo de confidencialidade (*Non-Disclosure Agreement* - NDA).

Em conformidade com as normas para trabalhos protegidos, o repositório de código fonte original não será tornado público na segunda entrega. Para colmatar esta omissão e permitir a validação científica e técnica da solução:

- Foi desenvolvido um **repositório de testes sintéticos** (*dummy dataset*), que replica a complexidade do ambiente real sem expor propriedade intelectual da empresa;
- A demonstração funcional da solução é apresentada através do **vídeo demonstrativo**, que ilustra o funcionamento do motor RAG e da interface CLI sobre o referido *dataset* sintético.

1.2 Enquadramento

No contexto atual do desenvolvimento de software, a automação de testes estabeleceu-se como um pilar fundamental para assegurar a qualidade e a estabilidade das aplicações. Contudo, a longevidade e a escala dos projetos introduzem uma complexidade crescente não apenas na base de código, mas na gestão da própria base de conhecimento e de requisitos, tipicamente composta por especificações de negócio (*User Stories*) e respetivos Casos de Teste (*Test Cases*).

À medida que um produto evolui, a inserção contínua de novos testes numa *suite* já extensa torna-se uma tarefa de alto risco. O principal desafio transita da execução técnica para o alinhamento semântico: garantir que um novo teste não contradiz regras de negócio previamente estabelecidas, não duplica validações já existentes e não introduz regressões lógicas no sistema.

A validação manual deste processo, onde um profissional necessita de cruzar mentalmente a intenção de um novo teste com dezenas de *User Stories* e centenas de testes, é uma tarefa incomportável e tediosa. Este esforço de investigação e correlação consome tempo considerável das equipas de *Quality Assurance* (QA) e desenvolvimento. Em organizações com múltiplas equipas, onde o conhecimento do domínio é fragmentado, esta gestão manual é propensa a falhas, resultando frequentemente na aprovação de testes redundantes ou conflituosos que alimentam a dívida técnica do projeto.

A filosofia subjacente a este projeto assenta no princípio de que tarefas exaustivas de pesquisa e identificação de impactos devem ser delegadas a sistemas automatizados. O objetivo é libertar os profissionais da carga administrativa e de pesquisas morosas, permitindo-lhes focar o seu tempo naquilo que verdadeiramente acrescenta valor: a conceção de novos cenários complexos, a análise crítica e o *design* da arquitetura de testes.

Desenvolvido emparceria com uma empresa do setor tecnológico, este projeto nasce da necessidade real de reduzir o tempo gasto em manutenção manual de *Test Cases* e de devolver aos profissionais a componente humana do seu trabalho.

A Inteligência Artificial, particularmente os *Large Language Models* (LLMs) integrados numa arquitetura de recuperação de informação (*Retrieval-Augmented Generation* - RAG), apresenta-se como uma solução promissora para este desafio.

1.3 Motivação e Identificação do Problema

A motivação para este projeto surge da confluência entre interesse académico em Inteligência Artificial aplicada e a necessidade de resolver um problema real identificado pela empresa parceira.

1.3.1 Identificação do Problema

A origem deste projeto reside numa lacuna tecnológica específica: a ausência de mecanismos automatizados para validar o impacto semântico da introdução de novos casos de teste face a uma base de conhecimento (*User Stories* e *Test Cases* legados) em constante expansão.

Sem ferramentas que detetem as redundâncias lógicas, contradições com regras de negócio já estabelecidas ou cenários fora de âmbito (*out of scope*), as *suites* de testes tornam-se redundantes, inconsistentes e difíceis de manter. A verificação da conformidade de um novo teste exige que um engenheiro de QA procure, leia e interprete manualmente a documentação anterior para garantir que a nova adição não gera regressões ou duplicação de esforço.

Esta gestão manual obriga a um esforço cognitivo exaustivo e consome uma parte significativa do tempo das equipas, manifestando-se em múltiplos impactos negativos:

- **Desperdício de tempo:** Profissionais altamente qualificados dedicam horas a tarefas de pesquisa manual e correlação de ficheiros (cruzar o novo teste com dezenas de *User Stories* e testes antigos) em vez de se dedicarem à criação de valor.
- **Desmotivação:** Atribuir consistentemente a tarefa tediosa de investigar a origem de regressões ou validar manualmente a sobreposição de testes conduz à frustração e reduz o *engagement* da equipa.
- **Degradação da qualidade:** Suítes mal mantidas levam à aprovação de testes contraditórios, gerando *suites* ruidosas e diminuindo a confiança nos resultados da automação.
- **Atrasos nos ciclos de desenvolvimento:** A deteção reativa de conflitos e a necessidade de refatorizar testes legados aumentam o tempo de revisão e atrasam a entrega de novas funcionalidades.

A empresa parceira identificou este problema como uma dor crónica que afeta múltiplas equipas. A necessidade de automatizar esta análise motivou a procura de uma solução baseada em IA que auxilie, mas não substitua, o julgamento humano.

1.4 Objetivos

O objetivo geral deste projeto é desenvolver uma solução de Inteligência Artificial que atue como um validador semântico preventivo durante a introdução de novos testes automatizados. A ferramenta acederá a repositórios de especificações para cruzar o conteúdo de novos *Test Cases* com a base de conhecimento existente (*User Stories* e testes

existentes), identificando automaticamente duplicações, contradições lógicas ou necessidades de refatorização, auxiliando assim as equipas de QA a otimizar as suas *suites* e a evitar regressões.

1.4.1 Objetivos Específicos

Para concretizar o objetivo geral e responder aos desafios de manutenção identificados, foram definidos quatro objetivos específicos (OE) que agrupam as funcionalidades nucleares do sistema:

- **OE1 – Integração e Ingestão de Artefactos de Teste:** Desenvolver mecanismos de integração com repositórios de controlo de versões (como o GitHub) para extrair e processar automaticamente ficheiros estruturados (ex: JSON) contendo as *User Stories* e os *Test Cases* existentes. O objetivo é tratar destes dados e preparar uma base de conhecimento otimizada para consulta.
- **OE2 – Avaliação Semântica e Matriz de Decisão (RAG + LLM):** Implementar uma arquitetura *Retrieval-Augmented Generation* (RAG) acoplada a um *Large Language Model* (LLM) capaz de avaliar o impacto de um novo teste. O sistema deverá classificar a inserção utilizando uma matriz estrita de *Status* (`Ok`, `Refactor`, `Obsolete`) e *Target* (`New Test`, `Old Tests`, `None`), detetando cenários fora de âmbito, duplicados ou conflituosos.
- **OE3 – Explicabilidade e Rastreabilidade (Human-in-the-loop):** Garantir que cada decisão da IA é acompanhada de uma justificação clara em linguagem natural e evidências concretas (por exemplo, listando os IDs exatos dos testes antigos que necessitam de refatorização). A interface de linha de comandos (CLI) deverá apresentar estes dados de forma legível, mantendo o humano como decisor final.
- **OE4 – Validação Técnica e Métricas de Desempenho:** Validar a eficácia da solução através do desenvolvimento de um *Minimum Viable Product* (MVP), aferindo a precisão da IA na classificação dos testes, o tempo de análise do *pipeline* e a consequente redução do esforço manual exigido aos engenheiros de QA na revisão de novos *commits*.

1.5 Estrutura do Documento

O presente relatório está organizado da seguinte forma:

- **Capítulo 1 - Introdução:** Apresenta o enquadramento, motivação, objetivos e estrutura.
- **Capítulo 2 - Pertinência e Viabilidade:** Analisa a pertinência do projeto, viabilidade técnica/económica, e compara com soluções existentes.
- **Capítulo 3 - Especificação e Modelação:** Detalha requisitos, casos de uso e arquitetura do sistema.
- **Capítulo 4 - Solução Proposta:** Descreve a implementação técnica, tecnologias e componentes desenvolvidos.
- **Capítulo 5 - Método e Planeamento:** Descreve a metodologia ágil e o cronograma do projeto.

2 - Pertinência e Viabilidade

Neste capítulo demonstra-se a pertinência do projeto através de análise de necessidades, avalia-se a viabilidade técnica, económica e social, e apresenta-se uma análise comparativa com soluções de mercado.

2.1 Pertinência

A pertinência deste projeto fundamenta-se numa necessidade real e concreta identificada pela empresa parceira, que reconheceu a manutenção de testes automatizados como um desafio crítico na sua operação.

2.1.1 Contexto do Problema

A pertinência deste projeto valida-se pela necessidade urgente de mitigar o desperdício de recursos na triagem de novos testes. Como detalhado no capítulo anterior, a ausência de um processo de validação semântica automatizado conduz a um cenário desfavorável.

A alocação ineficiente de recursos humanos para tarefas determinísticas impede que as equipas se foquem naquilo que verdadeiramente acrescenta valor: a análise crítica e a resolução criativa de problemas.

Conforme destacado pela empresa durante o levantamento de requisitos:

> *"Neste momento particular em que estamos, nós temos de devolver tempo às pessoas, aos trabalhadores. Dar aos robôs e ao software aquilo que é claramente robótico, e devolver aos seres humanos a parte humana do trabalho: a empatia, a comunicação com outras pessoas, o estabelecimento de conexões e a análise de problemas de uma forma mais sensitiva e intuitiva, que não tenha um caráter tão determinístico e racional."*

2.2 Viabilidade

A viabilidade do projeto foi avaliada segundo múltiplas dimensões para assegurar que a solução pode ser efetivamente implementada, testada e validada, mesmo perante constrangimentos técnicos.

2.2.1 Viabilidade Técnica

A viabilidade técnica assenta na maturidade das tecnologias necessárias de recuperação de informação e modelos de linguagem generativa:

- **Infraestrutura e Dados de Teste:** Para efeitos de validação controlada e isolada do *Minimum Viable Product* (MVP), foi criado um repositório *dummy* no GitHub. Este ambiente simula a estrutura real da empresa, contendo artefactos (em formato JSON) de *User Stories* e *Test Cases* antigos e novos, permitindo validar a extração e a lógica da ferramenta com precisão.
- **Pesquisa e Vetorização (RAG):** A implementação de bases de dados vetoriais locais (como o ChromaDB) é tecnicamente viável e assegura a pesquisa semântica rápida entre o novo teste e o histórico do projeto.

- **Inteligência Artificial (Modelo Local):** Devido a constrangimentos operacionais da empresa parceira no fornecimento de chaves de API comerciais (ex: OpenAI GPT-4), adotamos o **Ollama 3.1** (modelo Llama executado localmente). Embora esta solução imponha limitações conhecidas ao nível da janela de contexto e da profundidade de raciocínio lógico face a modelos de fronteira, garante a continuidade do projeto e valida a viabilidade da arquitetura geral.
- **Segurança e Privacidade:** A adoção forçada do modelo local transformou-se numa forte mais-valia arquitetónica. A análise ocorre 100% na máquina do utilizador (ou num servidor interno), assegurando que a propriedade intelectual (regras de negócio e código de testes) nunca é transmitida para serviços de terceiros na *cloud*. Em conjunto com as medidas de processamento em memória RAM (volátil) asseguramos que, mesmo escalando a nossa solução com uma API comercial, respeitamos políticas de segurança, no que toca ao armazenamento local.

2.2.2 Viabilidade Económica

A abordagem técnica adotada beneficia de uma estrutura de custos praticamente nula. O desenvolvimento ocorre em contexto académico e, ao contornar a dependência de APIs pagas (substituídas pela execução local do Ollama), a ferramenta não gera custos recorrentes por requisição (*pay-per-token*). O retorno esperado manifesta-se na redução de tempo gasto em manutenção e no aumento da eficiência das equipas.

2.2.3 Viabilidade Social e ODS

O projeto alinha-se com os Objetivos de Desenvolvimento Sustentável (ODS), nomeadamente o **ODS 8 (Trabalho Digno)**, ao eliminar tarefas repetitivas e promover o bem-estar no trabalho, e o **ODS 9 (Inovação)**, aplicando IA a desafios de engenharia. A resistência potencial à automação é mitigada pela obrigatoriedade de validação humana em todas as decisões.

2.3 Análise Comparativa com Soluções Existentes

O mercado oferece dezenas de ferramentas de gestão e automação de testes, mas existe uma lacuna evidente no que toca à validação preventiva e análise semântica de requisitos.

2.3.1 Benchmarking

A Tabela 2.1 apresenta uma comparação entre as principais tipologias de soluções de mercado e a proposta deste projeto.

Table 2.1: Comparação de Soluções

Critério de Avaliação	**Xray / TestRail**	**Testim / Mabl**	**LLMs Genéricos**	**QAlytics (Proposta)**
Gestão e arquivo de artefactos	Sim	Sim	Não	**Lê via Git (Não armazena)**
Validação Preventiva de Requisitos	Não	Não	Parcial	**Sim**
Deteção de Duplicação Semântica	Não	Não	Parcial	**Sim (via RAG)**
Explicabilidade Direcionada (IDs)	N/A	Limitada	Não garante	**Sim (Matriz Target)**
Privacidade Total (*Zero-Data*)	Não (*Cloud*)	Não (*Cloud*)	Não (*Cloud*)	**Sim (Modelo Local)**

2.3.2 Análise Crítica

A análise comparativa revela lacunas significativas na forma como a indústria aborda a manutenção das *suites* de testes:

Gestão Passiva vs. Validação Ativa: Ferramentas tradicionais de *test management* (como o Xray ou Zephyr) funcionam como excelentes arquivos relacionais, permitindo ligar um teste a uma *User Story*. No entanto, são sistemas passivos: não ”compreendem” o conteúdo. O trabalho cognitivo de garantir que o novo teste faz sentido ou não entra em conflito recai 100% sobre o humano.

Limitações na Deteção de Redundância: A pesquisa em repositórios clássicos baseia-se em *tags* ou palavras-chave. Isto falha redondamente na linguagem natural. Um teste sobre ”Abortar reserva” e outro sobre ”Cancelar estadia” não partilham palavras-chave, mas são semanticamente idênticos. Sem um motor vetorial (RAG), a redundância passa despercebida.

Ausência de Análise Semântica Profunda: Ferramentas modernas de automação com IA (como Testim ou Mabl) usam inteligência artificial essencialmente para *Self-Healing* (corrigir seletores de *interface* gráfica que quebram), e não para compreender a intenção da lógica de negócio face às *User Stories*.

Falta de Explicabilidade Pragmática: A utilização crua de *chatbots* (como o ChatGPT) para validar testes introduz problemas de privacidade e gera respostas não estruturadas. A solução proposta resolve isto através da sua matriz de *Status* e *Target*, devolvendo não apenas texto, mas apontando os *IDs* exatos dos testes antigos que requerem refatorização.

A solução **QAlytics** distingue-se por preencher exatamente esta lacuna: atua como um motor semântico que compreende a linguagem do negócio e a intenção do teste, sem comprometer a privacidade dos dados da empresa.

2.4 Proposta de Inovação e Mais-Valias

A ferramenta **QAlytics: Manutenção de testes com IA** demarca-se da concorrência através de três vetores de inovação:

1. **Gatekeeping Preventivo:** Em vez de tentar ”limpar” repositórios já poluídos, a ferramenta atua antes da inserção, bloqueando regressões e duplicações logo na raiz do processo de *Pull Request* ou *Commit*.

2. **Arquitetura RAG Local (Privacidade *By Design*):** A orquestração da ingestão de JSONs e bases de dados vetoriais com LLMs executados localmente (Ollama) garante que a propriedade intelectual da empresa nunca transita para fora do seu perímetro de segurança.

3. **Explicabilidade Direcionada:** Através de uma matriz de decisão estrita (*Status* e *Target*), a IA gera um veredicto acionável, apontando aos *IDs* dos artefactos afetados, poupando horas de investigação humana.

2.5 Identificação de Oportunidade de Negócio

O problema da manutenção de *test suites* legadas é transversal na indústria. A solução possui potencial para comercialização como **SaaS B2B** ou ferramenta *Enterprise*, direcionada a *Software Houses* e Consultoras. A proposta de valor centra-se na redução da dívida técnica de testes e aceleração dos ciclos de entrega.

3 - Especificação e Modelação

Neste capítulo identificam-se detalhadamente as características da solução a produzir sob a forma de requisitos, modelos e outros elementos que permitem perceber a estrutura e comportamento do sistema **QAlytics: Manutenção de testes com IA**.

3.1 Análise de Requisitos

A análise de requisitos visa definir o *scope* funcional e as restrições técnicas do projeto. Os requisitos foram classificados em Funcionais (o que o sistema faz) e Não-Funcionais (como o sistema se comporta).

3.1.1 Requisitos Funcionais

A Tabela 3.1 apresenta os requisitos funcionais identificados para o MVP e fases seguintes, adaptados à nova arquitetura de análise de Test Cases.

Table 3.1: Requisitos Funcionais do Sistema

ID	Requisito	Descrição	Prioridade
RF1	Extração de Artefactos	Aceder a repositórios via API (ex: GitHub) para leitura de ficheiros estruturados JSON (*User Stories* e *Test Cases*).	MUST
RF2	Tratamento de Dados	Limpar a informação extraída, removendo *tags* e campos irrelevantes para otimizar o *prompt* da IA.	MUST
RF3	Vetorização e Pesquisa	Converter artefactos em *embeddings* e guardá-los numa base de dados vetorial (ChromaDB) para pesquisa semântica rápida (RAG).	MUST
RF4	Avaliação Semântica	Analisar o impacto da inserção de um novo teste, identificando duplicações lógicas, contradições ou cenários fora de âmbito.	MUST
RF5	Matriz de Decisão	Classificar o impacto estruturadamente através de um *Status* (`Ok`, `Refactor`, `Obsolete`) e um *Target* (`New Test`, `Old Tests`, `None`).	MUST
RF6	Explicabilidade Direcionada	Gerar justificação em linguagem natural e apontar os *IDs* exatos dos artefactos (US ou TC antigos) que fundamentam a decisão.	MUST
RF7	Interface de Validação (CLI)	Apresentar os veredictos gerados pela IA no terminal de forma legível (com cores e tabelas) e aguardar aprovação/rejeição humana.	MUST
RF8	Sistema de Feedback	Registar as decisões do utilizador (*Human-in-the-loop*) para monitorização e futura afinação do sistema.	SHOULD

3.1.2 Descrição Detalhada dos Requisitos Principais

Esta subsecção detalha os requisitos de maior impacto e complexidade, explicitando dependências, processos envolvidos e cenários de utilização na nova arquitetura *Retrieval-Augmented Generation* (RAG).

RF1 e RF2 - Extração e Tratamento de Artefactos

Objetivo: Estabelecer conectividade com repositórios para extração e preparação da base de conhecimento semântico.

Descrição Detalhada: O sistema deve integrar-se com repositórios Git (simulando plataformas como o Azure DevOps) para ler as especificações ágeis guardadas em formato JSON. Após a extração, o módulo de Tratamento de dados (*Data Sanitizer*) deve isolar campos vitais (como `System.Description` e `Custom.TestSteps`), garantindo que o contexto enviado para a LLM é limpo e não esgota o limite de *tokens*.

RF3 e RF4 - Vetorização (ChromaDB) e Avaliação Semântica (RAG)

Objetivo: Recuperar o contexto relevante e avaliar a intenção do novo teste face ao histórico.

Descrição Detalhada: Sendo impossível enviar centenas de *User Stories* de uma só vez para o modelo local, o sistema deve converter os textos tratados em *embeddings* e armazená-los no ChromaDB. Quando um novo *Test Case* é submetido, o sistema realiza uma pesquisa de similaridade para injetar no *prompt* apenas os artefactos mais relevantes. A LLM avalia então se o novo teste repete comportamentos, se vai contra os requisitos documentados ou se obriga à atualização de testes antigos.

RF5 e RF6 - Matriz de Decisão e Explicabilidade Direcionada

Objetivo: Estruturar o *output* da IA de forma determinística e acionável para o utilizador final.

Descrição Detalhada: Para evitar respostas vagas (*alucinações*), o sistema obriga a LLM a classificar a sugestão em duas dimensões: o ***Status*** da ação a tomar (`Ok`, `Refactor`, `Obsolete`) e o ***Target*** da ação (`New Test`, `Old Tests`, `None`). Adicionalmente, a IA deve fornecer a lista exata de *IDs* dos testes antigos afetados, suportada por uma justificação concisa.

3.1.3 Requisitos Não-Funcionais

As restrições de qualidade e operação, fortemente marcadas pela transição para um modelo local, estão descritas na Tabela 3.2.

Table 3.2: Requisitos Não-Funcionais

ID	Categoria	Métrica de Aceitação
RNF1	Performance de Inferência	A pesquisa vetorial e a geração da justificação pela LLM local devem completar-se num tempo aceitável para fluxo CLI.
RNF2	Precisão Semântica	Alta taxa de acerto na atribuição correta das *flags* de *Status* e *Target* face à *baseline* de testes *dummy*.
RNF3	Autenticação Segura	Autenticação via *Tokens* nos repositórios, sem armazenamento permanente de credenciais em disco.
RNF4	Privacidade (Zero-Data Retention)	**Crítico:** O processamento (RAG e LLM via Ollama) deve ocorrer localmente ou em rede interna, sem enviar artefactos sensíveis para *clouds* públicas ou armazenar os mesmos em memória persistente local.
RNF5	Formato de Suporte	Suporte a leitura de artefactos de QA estruturados no formato JSON (norma de exportação de ferramentas modernas).
RNF6	Explicabilidade	100% das sugestões devem ter justificação em linguagem natural.

3.2 User Stories e Casos de Uso

Para facilitar o desenvolvimento incremental, os requisitos foram decompostos em *User Stories*, organizadas por Épicos.

3.2.1 Épico 1: Integração e Configuração Inicial

US1 - Autenticação e Extração de Artefactos

Como Engenheiro de QA
Quero conectar a ferramenta ao meu repositório (ex: GitHub) de forma segura
Para que o sistema extraia os ficheiros JSON contendo as *User Stories* e *Test Cases*.

Critérios de Aceitação:

- Autenticação OAuth2 ou via Token Pessoal estabelecida com sucesso.
- Sistema navega na estrutura do repositório e identifica os ficheiros JSON relevantes.
- O *Data Sanitizer* limpa as *tags* de metadados inúteis, retendo apenas identificadores, títulos, descrições e passos de teste.

US2 - Configuração do Motor Local (Ollama)

Como Tech Lead preocupado com a privacidade
Quero poder configurar a CLI para apontar para um modelo LLM local (Ollama)
Para garantir que a análise semântica ocorre sem custos e sem envio de dados para o exterior.

Critérios de Aceitação:

- Comando de configuração (`config --llm-provider`) suporta *endpoints* locais.
- O sistema verifica a conectividade e disponibilidade do modelo local (ex: *Llama 3.1*) antes de iniciar a análise.

3.2.2 Épico 2: Análise Semântica e Contextual (RAG)

US3 - Vetorização do Histórico (ChromaDB)

Como utilizador da ferramenta
Quero que os artefactos ingeridos sejam convertidos em vetores (*embeddings*)
Para que a IA consiga encontrar o contexto certo rapidamente.

Critérios de Aceitação:

- Os textos tratados são processados e guardados localmente no ChromaDB.
- É apresentada uma barra de progresso durante a criação dos *embeddings*.
- Dada uma *query* (um novo teste), a BD retorna os K artefactos mais relevantes.

US4 - Deteção de Duplicação Semântica

Como Revisor de Pull Requests
Quero que o sistema identifique se um novo teste é apenas um sinónimo de um antigo
Para evitar inchar a *suite* de testes sem adicionar nova cobertura.

Critérios de Aceitação:

- IA cruza os passos do novo teste com os testes antigos.
- Se a intenção for idêntica (ex: ”Abortar” vs ”Cancelar”), a sugestão gerada deverá ser `Status: Obsolete` e `Target: None`.
- Justificação deve explicar que se trata de uma cópia semântica.

US5 - Deteção de Conflito Lógico (Contradição)

Como Engenheiro de QA
Quero ser alertado caso o meu teste contradiga a regra de negócio base
Para corrigir o meu teste antes de o inserir no repositório.

Critérios de Aceitação:

- IA verifica os critérios de aceitação da *User Story* associada.
- Se o teste validar algo que a US proíbe (ex: reembolso a $< 24h$ quando a US diz $< 48h$), o sistema gera `Status: Refactor` (ou `Obsolete`) e `Target: New Test`.

US6 - Deteção de Comportamento Fora de Âmbito (Out of Scope)

Como Tech Lead
Quero barrar testes que introduzam fluxos não previstos nos requisitos
Para manter o escopo do projeto controlado.

Critérios de Aceitação:

- Se o novo teste usar entidades ou vocabulário não previsto nas US selecionadas (ex: mencionar ”Aluguer de Carro” numa US de ”Quartos de Hotel”), o sistema atua como *Gatekeeper*.
- Saída obrigatória da Matriz: `Status: Obsolete` e `Target: New Test`.

US7 - Sugestão de Refatorização de Testes Legados

Como Engenheiro de Automação
Quero que a IA me diga quando um novo teste, sendo bom, ”parte” os antigos
Para eu poder fundir ou apagar o ”lixo” acumulado.

Critérios de Aceitação:

- Se o novo teste cobre a mesma US, mas com mais detalhe ou abrangendo regras atualizadas, o sistema aprova o novo mas ataca os antigos.
- Saída obrigatória: `Status: Refactor` e `Target: Old Tests`.
- A justificação tem de listar, obrigatoriamente, os `Expected IDs` dos testes antigos a rever.

US8 - Aprovação Limpa (Happy Path)

Como Engenheiro de QA
Quero que o sistema aprove testes perfeitos sem criar atrito
Para poder focar-me nos problemas reais.

Critérios de Aceitação:

- Teste não duplica, não contradiz e está no âmbito.
- Matriz gerada: `Status: Ok` e `Target: None`.

3.2.3 Épico 3: Visualização e Tomada de Decisão (CLI)

US9 - Interface de Validação Interativa

Como utilizador da ferramenta
Quero visualizar a decisão da IA de forma clara no meu terminal
Para poder tomar a decisão final (Human-in-the-loop).

Critérios de Aceitação:

- CLI utiliza cores (ex: *Rich*) para diferenciar os Status (Verde para `Ok`, Amarelo para `Refactor`, Vermelho para `Obsolete`).
- Ecrã apresenta o ID do Teste, a Justificação e as Evidências (Target IDs).
- Utilizador tem teclas de atalho claras para `[A]provar` sugestão, `[R]ejeitar` análise da IA ou fazer `[S]kip`.

3.2.4 Épico 4: Privacidade e Segurança (Privacy by Design)

US10 - Zero-Data Retention e Execução Offline

Como administrador de sistemas
Quero garantias de que os artefactos proprietários não saem do ambiente de desenvolvimento local
Para cumprir as diretrizes rigorosas de *Compliance* e RGPD da empresa.

Critérios de Aceitação:

- Toda a vetorização (ChromaDB) ocorre *in-memory* ou numa base de dados SQLite estritamente local.
- Toda a inferência semântica é redirecionada para a instância do Ollama na rede interna.
- Os ficheiros JSON importados não são arquivados permanentemente na ferramenta.

3.3 Modelação

Esta secção apresenta a modelação técnica do sistema, detalhando a estrutura de dados, a organização das classes e os fluxos de atividade.

3.3.1 Modelo de Dados (Entidade-Relação)

A persistência de dados para o MVP é assegurada por uma base de dados SQLite local, garantindo o requisito de privacidade (*Zero-Data Retention*). O modelo foi desenhado na 3ª Forma Normal (3NF) para garantir a integridade referencial.

A Figura 3.1 ilustra o Diagrama Entidade-Relação (DER) do sistema.

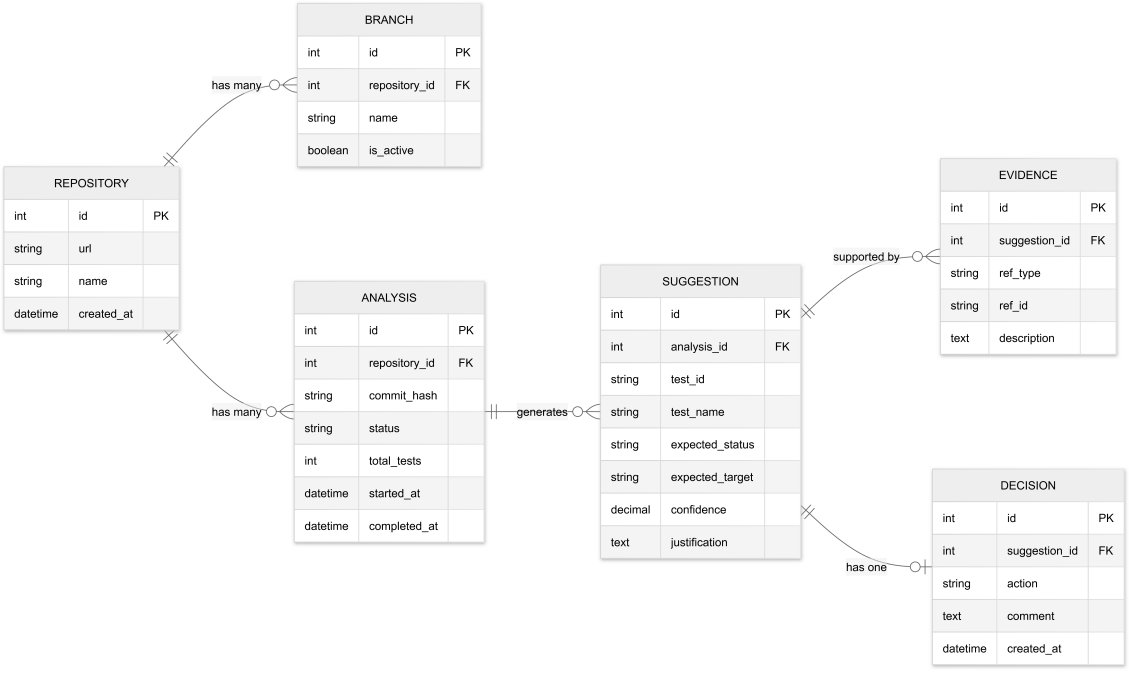


Figure 3.1: Diagrama Entidade-Relação do QAlytics (Arquitetura RAG)

Dicionário de Dados

Abaixo apresenta-se a especificação detalhada de cada tabela, tipos de dados e restrições.

1. **Tabela** `Repository` Armazena os metadados dos projetos monitorizados.

Coluna	Tipo	Restrições	Descrição
id	INTEGER	PK, AUTO_INCREMENT	Identificador único.
url	VARCHAR(500)	NOT NULL, UNIQUE	URL remoto (ex: GitHub).
name	VARCHAR(255)	NOT NULL	Nome do projeto (ex: user/repo).
created_at	DATETIME	DEFAULT NOW()	Data de registo.

2. **Tabela** `Branch` Define os ramos de desenvolvimento ativos para cada repositório.

Coluna	Tipo	Restrições	Descrição
id	INTEGER	PK, AUTO_INCREMENT	Identificador da branch.
repository_id	INTEGER	FK (Repository), NOT NULL	Repositório pai.
name	VARCHAR(100)	NOT NULL	Nome (ex: 'main', 'dev').
is_active	BOOLEAN	DEFAULT TRUE	Se está ativa para análise.

3. **Tabela** `Analysis` Regista cada execução de análise num repositório.

Coluna	Tipo	Restrições	Descrição
id	INTEGER	PK, AUTO_INCREMENT	Identificador único da análise.
repository_id	INTEGER	FK (Repository), NOT NULL	Repositório alvo.
commit_hash	VARCHAR(40)	NOT NULL	Hash do commit analisado.
status	VARCHAR(20)	CHECK IN ('running', 'completed', 'failed')	Estado da execução.
total_tests	INTEGER	DEFAULT 0	Total de novos testes avaliados.
started_at	DATETIME	DEFAULT NOW()	Início da execução.
completed_at	DATETIME	NULL	Fim da execução.

4. **Tabela** `Suggestion` Armazena a matriz de decisão gerada pelo LLM para um novo teste. **(Core da Aplicação)**

Coluna	Tipo	Restrições	Descrição
id	INTEGER	PK, AUTO_INCREMENT	Identificador da sugestão.
analysis_id	INTEGER	FK (Analysis), NOT NULL	Análise que gerou a sugestão.
test_id	VARCHAR(100)	NOT NULL	ID do novo teste (ex: TC-146).
test_name	VARCHAR(500)	NOT NULL	Título/Nome do novo teste.
file_path	VARCHAR(500)	NOT NULL	Localização do ficheiro.
expected_status	VARCHAR(20)	CHECK IN ('Ok', 'Refactor', 'Obsolete')	Veredicto de ação da IA.
expected_target	VARCHAR(20)	CHECK IN ('New Test', 'Old Tests', 'None')	Alvo da ação sugerida.
confidence	DECIMAL(5,2)	CHECK (0-100)	Score de confiança da IA.
justification	TEXT	NOT NULL	Explicação gerada pelo modelo local.

5. **Tabela** `Evidence` Contém as referências (Target IDs) que a IA usou para basear a sua decisão.

Coluna	Tipo	Restrições	Descrição
id	INTEGER	PK, AUTO_INCREMENT	Identificador da evidência.
suggestion_id	INTEGER	FK (Suggestion), NOT NULL	Sugestão associada.
ref_type	VARCHAR(50)	CHECK IN ('User Story', 'Old Test')	Tipo de artefacto referenciado.
ref_id	VARCHAR(50)	NOT NULL	ID do artefacto (ex: US-12, TC-127).
description	TEXT	NULL	Contexto de conflito/duplicação.

6. **Tabela** `Decision` Regista a validação humana sobre as sugestões da IA.

Coluna	Tipo	Restrições	Descrição
id	INTEGER	PK, AUTO_INCREMENT	Identificador da decisão.
suggestion_id	INTEGER	FK (Suggestion), UNIQUE	Sugestão decidida (1:1).
action	VARCHAR(20)	CHECK IN ('APPROVED', 'REJECTED')	Ação tomada pelo QA.
comment	TEXT	NULL	Comentário opcional do utilizador.
created_at	DATETIME	DEFAULT NOW()	Data da decisão.

3.3.2 Diagrama de Classes

O sistema segue uma arquitetura orientada a objetos, organizada em camadas lógicas para garantir a separação de responsabilidades e facilitar a testabilidade de cada componente isolado no *pipeline* RAG. A Figura 3.2 apresenta as principais classes do sistema.

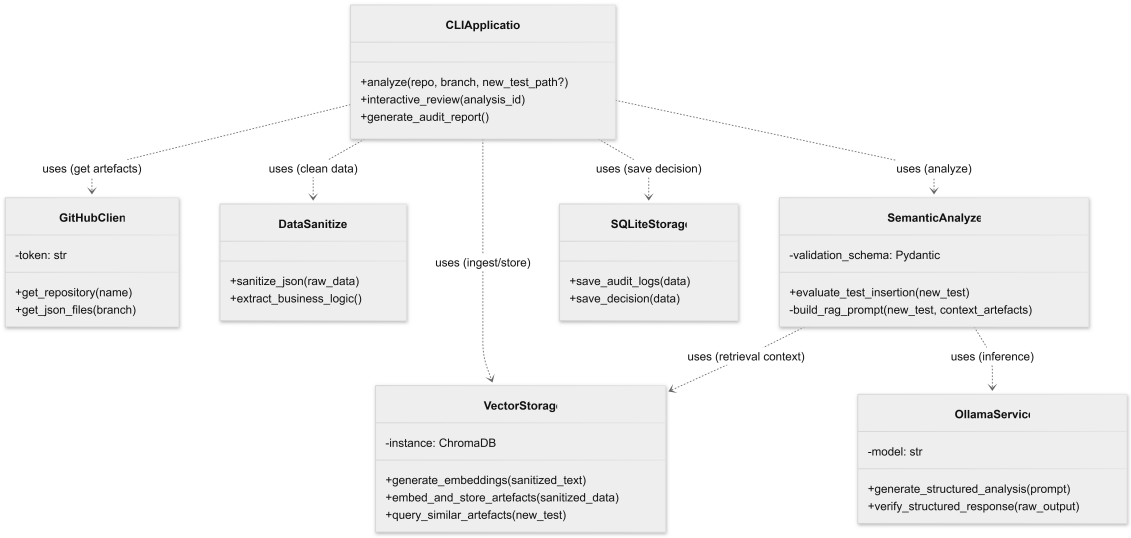


Figure 3.2: Diagrama de Classes (Componentes da Arquitetura RAG)

As principais classes identificadas na nova arquitetura são:

- **CLI (Interface):** Responsável pela interação com o utilizador, orquestração dos comandos no terminal e renderização visual dos veredictos (utilizando a biblioteca `Rich`).
- **GitHubClient:** Encapsula a comunicação com a API REST do repositório para extração dos ficheiros JSON contendo as *User Stories* e *Test Cases*.
- **DataSanitizer:** Componente crucial de pré-processamento. Analisa os JSONs extraídos, remove *tags* de sistema irrelevantes e isola os campos vitais (ex: `System.Description`, `Custom.TestSteps`) para otimizar o consumo de *tokens*.
- **VectorStorage:** Interface direta com a base de dados vetorial local (ChromaDB). É responsável por gerar os *embeddings* dos artefactos higienizados e executar as pesquisas de similaridade (*Semantic Search*).
- **SemanticAnalyzer:** Recebe um novo teste, consulta o `VectorStorage` para obter o contexto histórico mais relevante e constrói o *prompt* estruturado. Parte do VectorStorage.
- **AIService:** Adaptador que gere a comunicação HTTP com a instância local do LLM (Llama 3.1). Garante que a inferência cumpre o requisito de *Zero-Data Retention* e devolve a matriz de decisão (*Status* e *Target*).

- **SQLiteStorage:** Implementa o padrão *Repository* para persistência de dados. Grava as análises, sugestões e as validações humanas de forma segura no disco local.

3.3.3 Diagrama de Atividade

Para ilustrar o fluxo de execução principal da nova arquitetura (”Análise Semântica de Novo Teste”), apresenta-se o diagrama de atividade na Figura 3.3. Este fluxo espelha o *pipeline* completo de *Retrieval-Augmented Generation* (RAG), desde a extração de dados até à validação humana.

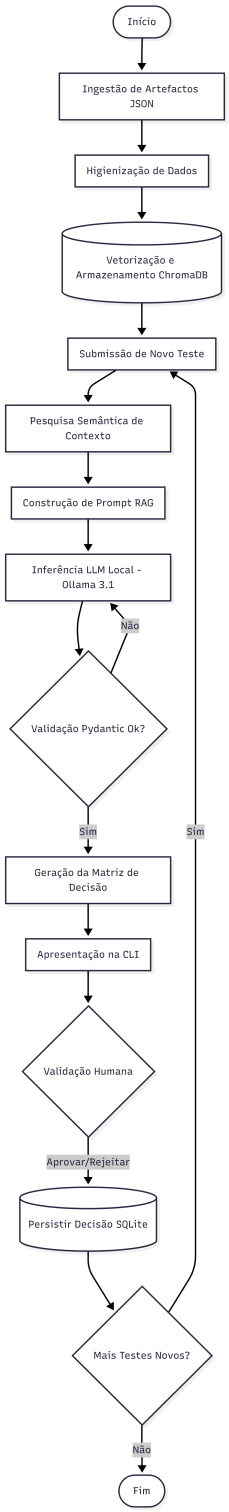


Figure 3.3: Fluxo de Atividade: Análise de Repositório

O fluxo de execução lógico compreende os seguintes passos sequenciais:

1. **Extração de Artefactos:** Autenticação no repositório e extração massiva dos ficheiros JSON contendo a base de conhecimento (*User Stories* e *Test Cases* legados).
2. **Tratamento de Dados (*Data Sanitizer*):** Limpeza das *tags* e metadados irrelevantes dos ficheiros importados, preservando apenas as descrições e os passos lógicos.
3. **Vetorização e Armazenamento:** Conversão dos textos limpos em *embeddings* e armazenamento na base de dados vetorial local (ChromaDB).
4. **Submissão do Novo Teste:** O sistema recebe o ficheiro JSON correspondente ao novo *Test Case* que o utilizador pretende validar.
5. **Pesquisa Semântica (RAG):** O motor consulta o ChromaDB para recuperar os K artefactos mais relevantes e semanticamente próximos do novo teste.
6. **Inferência LLM (Ollama):** Construção de um *prompt* injetando o novo teste e o contexto recuperado, enviando-o para o modelo local avaliar possíveis contradições, duplicações ou falhas de âmbito.
7. **Geração da Matriz de Decisão:** O modelo devolve um veredicto estrito contendo o *Status* (`Ok`, `Refactor`, `Obsolete`), o *Target* (`New Test`, `Old Tests`, `None`) e os respetivos *IDs* dos artefactos referenciados.
8. **Validação Interativa (CLI):** Apresentação visual da sugestão e das evidências ao utilizador, aguardando a decisão final (`[A]provar` ou `[R]ejeitar`) antes da persistência final na base de dados SQLite.

3.4 Protótipos de Interface (CLI)

Dado que o MVP do **QAlytics** é uma ferramenta de linha de comandos (CLI), os protótipos de interface focam-se na experiência de utilização no terminal. Em vez de ecrãs gráficos tradicionais, apresentam-se os fluxos de comandos e os *outputs* estruturados gerados através de bibliotecas de formatação visual (`Rich` / `Typer`).

3.4.1 Mapa de Comandos (Fluxo de Navegação)

A navegação na aplicação é realizada através de argumentos e subcomandos passados ao executável principal. A Figura 3.4 ilustra a hierarquia de comandos atualizada para a arquitetura RAG.

```
QAlytics
|
+-- config                  # Configuração de Infraestrutura
|   |-- --llm-provider      # Define modelo local (ex: ollama/qwen2.5-coder)
|   |-- --vector-db         # Define path para persistência do ChromaDB
|   `-- --reset             # Limpa base de dados vetorial
|
+-- analyze <file.json>     # Ingestão e Análise Semântica
|   |-- --scope <type>      # Define o alvo (ex: all / new_only)
|   `-- --threshold <n>     # Filtra por confiança mínima do LLM
|
+-- interactive             # Modo de Revisão Humana (Decisão)
|   `-- [Loop de Sugestões] -> Aprovar / Rejeitar
|
`-- export                  # Exportação de resultados
    `-- --format <type>     # Formato de saída (csv/json/html)
```

Figure 3.4: Mapa de Comandos e Argumentos da CLI

3.4.2 Mockups de Interação

Abaixo apresentam-se os resultados esperados para as interações principais, demonstrando como o motor RAG expõe as suas decisões ao utilizador.

1. Ingestão e Análise de Dados Sintéticos

Ao executar o comando de análise sobre o *dataset* exportado (ex: Azure DevOps), o sistema fornece *feedback* do progresso da vetorização e inferência (Figura 3.5).

```
$ QAlytics analyze data/hotel_dummy_dataset.json --scope new_only

>> QAlytics - Starting RAG Pipeline Analysis
------------------------------------------------------------
Dataset: hotel_dummy_dataset.json
Target: 6 New Test Cases

[1/4] [+] Parsing JSON (15 User Stories, 40 Existing TCs)... [OK]
[2/4] [~] Generating Embeddings & Updating ChromaDB...       [OK]
[3/4] [#] Retrieving Semantic Context for New TCs...         [OK]
[4/4] [@] LLM Validation (Logical Conflicts/Duplicates)...   [=====--] 66%

------------------------------------------------------------
[OK] ANALYSIS COMPLETE (Duration: 42m 15s)

Total New Tests Evaluated: 6
[!] Anomalies Detected: 5
[V] Valid (Happy Path): 1

-> Next step: Run 'QAlytics interactive' to review decisions.
```

Figure 3.5: Mockup: Execução da análise RAG com feedback de progresso

2. Modo Interativo (Validação de Conflitos Lógicos)

Este é o ecrã principal de tomada de decisão. O sistema apresenta o *Test Case* analisado, o contexto recuperado da *User Story* original e a dedução lógica gerada pelo LLM (Figura 3.6).

```
$ QAlytics interactive

SUGESTÃO 3 de 5  [ALTA CONFIANÇA - 95%]
------------------------------------------------------------
Test Case: TC-143 (Refund 24h)
Categoria de Anomalia: Logical Conflict
Decisão Sugerida: REFACTOR + NEW TEST

(i) DEDUÇÃO DA IA:
O Test Case analisado espera a aprovação de um reembolso total a
24 horas do check-in. No entanto, isto contradiz diretamente as
regras de negócio da User Story base, que estipula uma penalidade
obrigatória de 1 noite para cancelamentos inferiores a 48 horas.

(#) CONTEXTO RECUPERADO (ChromaDB):
-> [Origem: US-12 - Cancellation Policy]
   "...cancellations made < 48h prior to check-in incur a 1-night fee."
-> [Similaridade Vetorial: 0.89]

------------------------------------------------------------
Escolha uma ação:
[A]provar - Assinalar TC-143 para revisão de regras de negócio
[R]ejeitar - Ignorar aviso (Falso Positivo)
[S]kip - Decidir depois
[D]etalhes - Ver JSON estruturado do TC e da US

> _
```

Figure 3.6: Mockup: Modo Interativo ilustrando a deteção de um Conflito Lógico

4 - Solução Proposta

Neste capítulo apresenta-se a solução técnica planeada para o **QAlytics: Manutenção de testes com IA**, detalhando a arquitetura, as tecnologias selecionadas e a especificação dos componentes que serão implementados na fase de desenvolvimento.

4.1 Apresentação

A solução proposta consiste numa ferramenta de linha de comandos (CLI) concebida para automatizar a validação semântica e a triagem de casos de teste automatizados. Em vez de atuar apenas como um mecanismo de limpeza reativa de código legado, a ferramenta posiciona-se estrategicamente como um *Gatekeeper* preventivo no ciclo de engenharia de qualidade.

A solução diferencia-se das abordagens tradicionais de mercado pela implementação de uma arquitetura *Retrieval-Augmented Generation* (RAG) operada localmente. Combina a velocidade e precisão da pesquisa vetorial (através do motor ChromaDB e de modelos de *embeddings* dedicados, como o `all-MiniLM-L6-v2`) com a capacidade de raciocínio lógico de *Large Language Models* (LLMs) executados no perímetro da máquina do utilizador (via Ollama). O objetivo central é mitigar o crescimento da dívida técnica, avaliando se a introdução de um novo teste entra em conflito, duplica intenções ou contradiz a base de conhecimento existente (*User Stories* e testes antigos).

Para garantir o rigor das sugestões e eliminar o risco de formatações imprevisíveis (*alucinações*), o sistema integra mecanismos de validação estrutural (através da biblioteca Pydantic), que obrigam o modelo gerativo a devolver as suas conclusões estritamente formatadas segundo a Matriz de Decisão do projeto (*Status* e *Target*), acompanhadas de justificações em linguagem natural e da identificação dos *IDs* dos test cases afetados.

Para a validação do conceito (PoC/MVP), a solução foca-se na interface CLI, permitindo uma integração simples e rápida em ambientes de desenvolvimento locais, deixando a interface Web para uma fase posterior de maturação do produto.

A solução apresentada constitui um projeto original, desenhado de raiz para este TFC, não existindo reaproveitamento de componentes ou código prévio.

4.2 Arquitetura

A solução segue uma **Arquitetura em Camadas** (*Layered Architecture*), desenhada para garantir a separação de responsabilidades e facilitar a testabilidade individual de cada módulo no contexto de um *pipeline* de *Retrieval-Augmented Generation* (RAG).

A opção por uma arquitetura modular fundamenta-se na necessidade técnica de evoluir os componentes de forma independente. Num ecossistema de Inteligência Artificial em rápida mutação, esta separação garante, por exemplo, que o motor vetorial (ChromaDB) ou o provider do LLM (Ollama) possam ser substituídos no futuro sem impactar a lógica central de avaliação de testes, ou que a interface CLI possa evoluir para uma integração nativa em *pipelines* CI/CD.

A Figura 4.1 ilustra os principais componentes da arquitetura proposta.

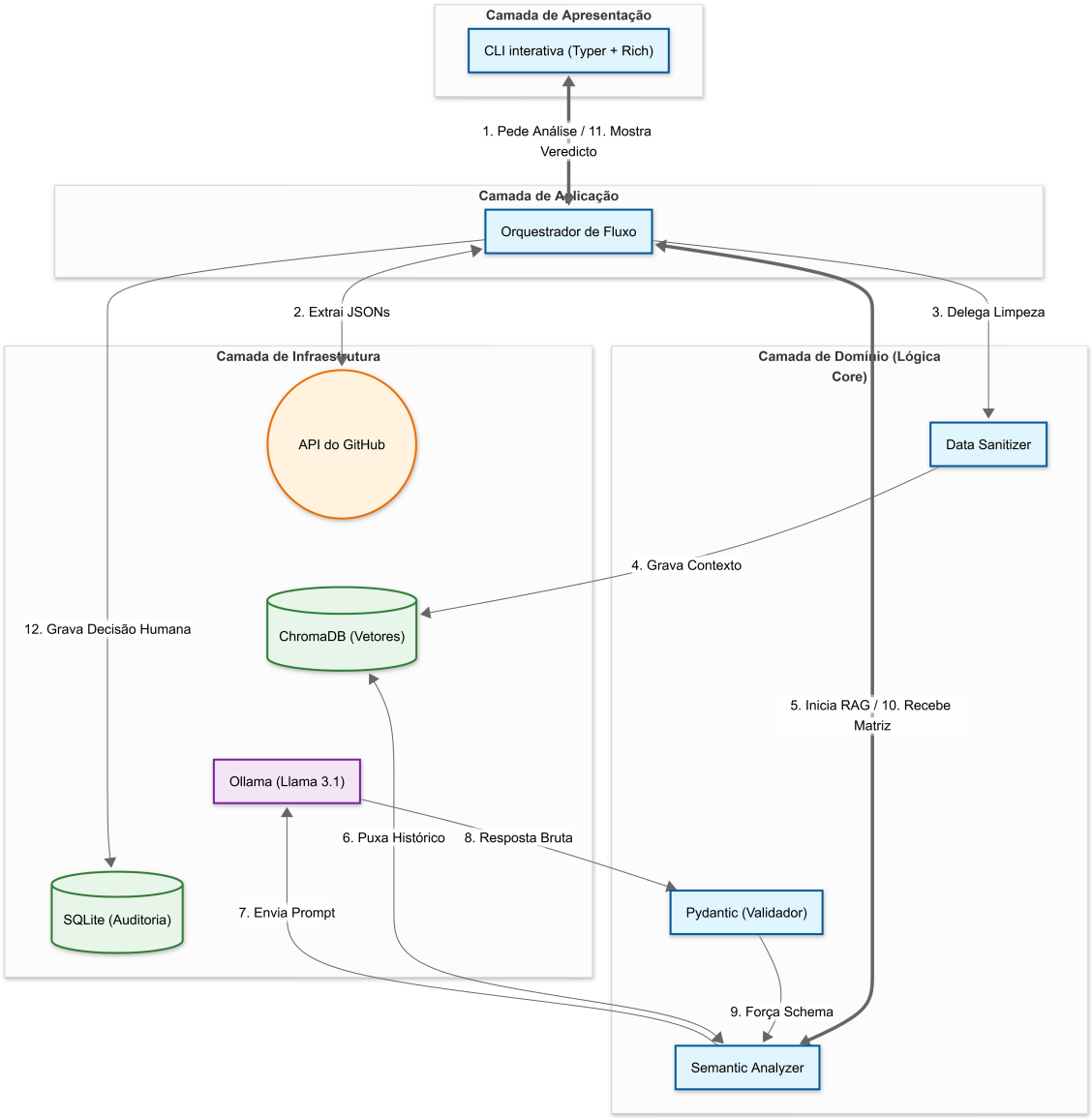


Figure 4.1: Arquitetura da Solução (Pipeline Semântico e RAG)

Os componentes do **QAlytics** estão organizados nas seguintes camadas lógicas:

1. **Camada de Apresentação (Interface):** Gere a interação com o utilizador via CLI. É responsável pela receção dos comandos de análise em lote (*batch*) ou individual, e pela formatação visual da Matriz de Decisão gerada pela IA, utilizando a biblioteca **Rich**.

2. **Camada de Aplicação (Orquestração):** Coordena o fluxo de execução linear. Gere o ciclo de vida dos dados, ordenando a extração dos ficheiros JSON, passando-os para o tratamento e acionando o motor de análise.

3. **Camada de Domínio (Lógica Core):** Contém o `DataSanitizer` (que isola os dados vitais dos artefactos) e o `SemanticAnalyzer`. Este último constrói os *prompts* contextuais rigorosos, validando as respostas estruturadas recebidas (via *Pydantic*) para garantir que a matriz *Status/Target* é estritamente cumprida.

4. **Camada de Infraestrutura:** Gere as integrações e a persistência externa de forma isolada:

- **Vector Storage:** Instância embutida do ChromaDB que aloja o modelo de *embeddings* (`all-MiniLM-L6-v2`) e executa a pesquisa semântica ultrarrápida.
- **AI Adapter:** Cliente HTTP que comunica com a instância local do Ollama (Llama 3.1), assegurando o perímetro de *Zero-Data Retention*.
- **Log Persistence:** Base de dados SQLite tradicional dedicada exclusivamente à gravação do histórico de auditoria (registo de sugestões e validações manuais do utilizador).

4.3 Tecnologias e Ferramentas Utilizadas

A seleção tecnológica privilegiou ferramentas maduras do ecossistema Python, garantindo estabilidade para o MVP e facilidade de integração com serviços de IA.

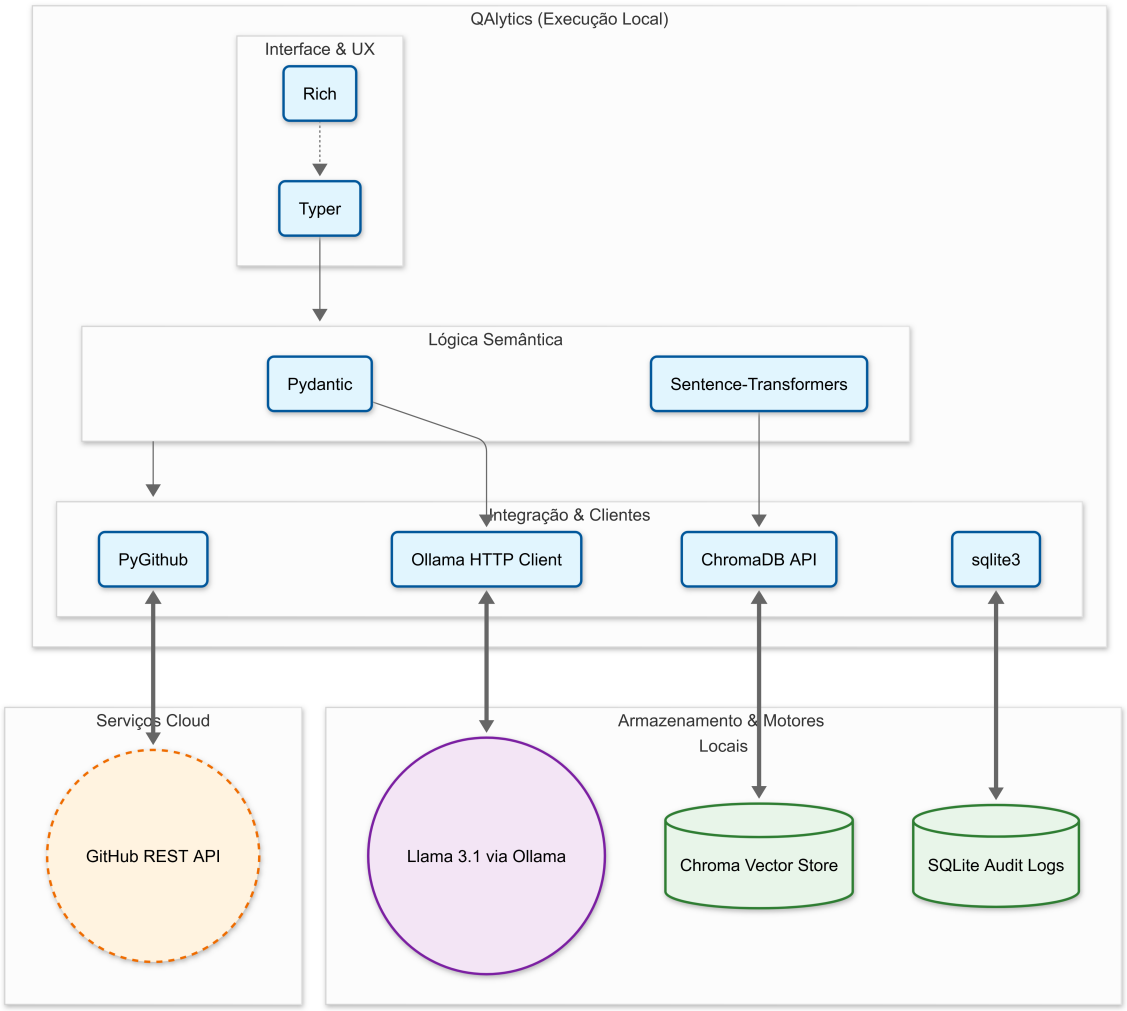


Figure 4.2: Tecnologias e Perímetros Tecnológicos (Stack RAG Local)

Lista de Ferramentas e Justificação:

Python 3.11+: Linguagem principal do projeto. Escolhida pelo seu domínio absoluto no ecossistema de Inteligência Artificial e processamento de dados, além do suporte nativo a *type hints* avançados.

Typer + Rich: Bibliotecas responsáveis pela interface (CLI). O `Typer` foi escolhido em detrimento de soluções mais antigas por permitir a construção de interfaces baseadas em anotações de tipo Python (*type hints*), reduzindo código *boilerplate*. O `Rich` atua como motor de renderização visual no terminal, formatando matrizes, cores e *outputs* para uma excelente experiência de utilização.

PyGithub: Cliente oficial para a API REST do GitHub. Utilizado para a ingestão massiva dos ficheiros JSON (artefactos e especificações) e navegação na árvore de diretórios do repositório.

ChromaDB: Base de dados vetorial (*Vector Database*) embutida e autónoma. Atua como o motor de pesquisa semântica do sistema. O ChromaDB gere nativamente tanto a indexação matemática dos *embeddings* (via `hnswlib`) como o armazenamento estruturado dos metadados (*tags* de *User Stories*) num ficheiro SQLite interno, orquestrando ambos de forma invisível.

Sentence-Transformers (`all-MiniLM-L6-v2`): Modelo de *embeddings* integrado para a vetorização dos textos. Este modelo foi escolhido por ser o padrão de alta eficiência para RAG: é extremamente leve, rápido a correr em CPU e altamente preciso a mapear intenções semânticas em inglês.

Ollama + Llama 3.1 (8B): Motor de inferência *Large Language Model* (LLM) operado localmente. Substitui dependências comerciais (como a OpenAI) para garantir privacidade absoluta (*Zero-Data Retention*). O modelo *Llama 3.1* oferece a capacidade de raciocínio lógico necessária para cruzar o contexto do RAG com o novo teste. *Sujeito a mudanças de escolha do modelo.*

Pydantic: Biblioteca de validação de dados. Essencial para domar o *output* gerativo da IA. O Pydantic força o modelo a devolver respostas estritamente compatíveis com um *Schema* JSON predefinido, garantindo que as chaves `status` e `target` nunca falham a formatação esperada pela aplicação.

SQLite: Base de dados relacional clássica. Enquanto o ChromaDB gere os vetores e o contexto semântico, o SQLite é utilizado exclusivamente como base de dados aplicacional para persistir o histórico de auditoria, ou seja, guardar o registo das sugestões geradas ao longo do tempo e as decisões manuais de aprovação/rejeição dos utilizadores.

4.4 Ambientes de Teste e de Produção

Considerando a natureza da aplicação e o foco estrito na privacidade de dados (*Zero-Data Retention*), o ambiente produtivo corresponde à estação de trabalho local do utilizador final (Engenheiro de QA ou *Developer*) ou a um servidor interno da empresa.

Dada a adoção de uma arquitetura baseada em LLMs executados localmente, os requisitos de *hardware* assumem uma importância crítica para garantir uma experiência de utilização fluida.

Requisitos de Produção (Máquina do Utilizador):

- **Software:** Python 3.11+ e cliente Git instalados. Motor **Ollama** instalado e a correr em *background* com o modelo *Llama 3.1* descarregado.
- **Hardware (Mínimo recomendado):** 16 GB de memória RAM (necessários para alocar os pesos do modelo de 8B parâmetros em memória) e armazenamento SSD (para garantir a velocidade de leitura/escrita do ChromaDB). A presença de uma GPU dedicada ou arquiteturas otimizadas (como Apple Silicon) reduz drasticamente o tempo de inferência.
- **Rede:** Acesso à Internet necessário apenas para a comunicação inicial com a API do repositório (GitHub). A fase de inferência e consulta à base de dados ocorre de forma 100% *offline*.

4.5 Abrangência

O desenvolvimento desta solução reflete a maturidade académica adquirida, integrando conhecimentos transversais de múltiplas áreas científicas e unidades curriculares, adaptadas às exigências da inteligência artificial moderna, conforme descrito na Tabela 4.1.

Table 4.1: Mapeamento de Áreas Científicas e Unidades Curriculares

Área Científica	**Aplicação na Solução**	**Componentes**
Programação	Desenvolvimento da CLI (Typer), estruturação de código Python com tipagem estática e programação orientada a objetos.	Todo o sistema
Algoritmia e Estruturas de Dados	Processamento e tratamento de estruturas JSON complexas; modelação de matrizes multidimensionais (*embeddings*).	`DataSanitizer`, `VectorStorage`
Engenharia de Software	Arquitetura em camadas, *Type-Driven Development* (Pydantic), separação de responsabilidades e foco na escalabilidade.	Capítulos 3 e 4
Bases de Dados	Modelação E-R relacional (SQLite) combinada com bases de dados vetoriais não-relacionais (ChromaDB).	`VectorStorage`, `SQLiteStorage`
Inteligência Artificial	Implementação de arquitetura RAG, engenharia de *prompts* restritivos e orquestração de LLMs locais.	`SemanticAnalyzer`
Computação Distribuída	Consumo de APIs REST (GitHub) e comunicação HTTP assíncrona com instâncias de inferência local (Ollama).	`GitHubClient`, `OllamaService`
Segurança Informática	Gestão segura de credenciais de acesso (*Tokens*) e arquitetura *Privacy by Design* (execução *offline*).	Configuração

4.6 Componentes

Esta secção detalha a implementação técnica dos módulos core da aplicação, refletindo a transição para a arquitetura de análise semântica (*Retrieval-Augmented Generation*).

4.6.1 Interface CLI (Typer e Rich)

A `CLI Application` atua como o ponto de entrada do sistema. Desenvolvida sobre a biblioteca `Typer`, a interface mapeia comandos de terminal para funções Python utilizando anotações de tipo (*type hints*), o que garante validação automática de argumentos (ex:

caminhos de pastas, níveis de confiança). A renderização do *output* é delegada à biblioteca `Rich`, que formata a matriz de decisão da IA (o veredicto de *Status* e *Target*) em tabelas coloridas e legíveis, gerindo também o fluxo interativo que permite ao utilizador aprovar ou rejeitar uma sugestão em tempo real. Estado atual: Lógica implementada no CLI.py, falta o fluxo iterativo.

4.6.2 Data Sanitizer (Ingestão e Pré-Processamento)

O `DataSanitizer` é um componente vital para a viabilidade do LLM local. Ficheiros de especificações ágeis (JSONs exportados de plataformas como Jira ou Azure DevOps) contêm centenas de linhas de metadados inúteis para a análise de qualidade. Este módulo atua como um filtro: faz o *parsing* dos ficheiros JSON e extrai exclusivamente as chaves que contêm intenção de negócio (como `System.Description` e `Custom.TestSteps`). Assim poupamos *tokens* na janela de contexto da IA e melhora a qualidade da vetorização. Estado atual: Implementado no ficheiro PythonParser.py

4.6.3 Vector Storage (Motor RAG via ChromaDB)

O `VectorStorage` encapsula a base de conhecimento do projeto. Utilizando o motor autónomo **ChromaDB**, este componente realiza duas tarefas em simultâneo de forma orquestrada:

- **Vetorização:** Converte os textos limpos pelo `DataSanitizer` em vetores matemáticos (*embeddings*) utilizando o modelo `all-MiniLM-L6-v2` (otimizado via Sentence-Transformers).
- **Gestão de Metadados:** Associa dicionários estruturados a cada vetor (contendo identificadores únicos como o *ID da User Story* ou do *Old Test*), gravando-os num ficheiro SQLite embutido, gerido de forma invisível pela própria API do ChromaDB.

Quando um novo teste é submetido, este componente calcula a similaridade cosseno (*cosine similarity*) e devolve os artefactos históricos mais próximos para formar o contexto. Estado atual: Lógica implementada no ficheiro ChromaScript.py

4.6.4 Semantic Analyzer e AI Adapter

O `SemanticAnalyzer` é o ”cérebro” do *QAlytics*. A sua responsabilidade é cruzar o conteúdo do novo teste com o contexto devolvido pelo `VectorStorage`. O fluxo interno deste componente é altamente estruturado:

1. **Prompt Engineering:** Constrói um *System Prompt* com diretrizes estritas de validação de requisitos, injetando os artefactos extraidos recuperados da base vetorial.
2. **Inferência (Ollama):** Envia o *prompt* via HTTP para a instância local do Llama 3.1.
3. **Validação Estrutural (Pydantic):** O *output* gerativo da IA é forçado a respeitar um *Schema* pré-definido. O componente valida em tempo real se a resposta contém chaves JSON válidas para o `expected_status` (`Ok`, `Refactor`, `Obsolete`) e para o `expected_target` (`New Test`, `Old Tests`, `None`), bem como os IDs dos artefactos afetados.

Estado atual: Lógica implementada no fichiero AIService.py.

4.6.5 Storage Layer (Histórico e Auditoria)

Enquanto o contexto semântico vive no ChromaDB, o `SQLiteStorage` é responsável pelo registo persistente de operações. Implementa o padrão *Repository* interagindo com uma base de dados relacional clássica (SQLite). A sua única função é garantir um *log* de auditoria à prova de falhas: armazena a matriz de sugestões gerada para cada análise e guarda o registo cronológico das decisões de aprovação/rejeição tomadas pelo utilizador, essencial para futuras auditorias e extração de métricas de qualidade. Estado atual: Por implementar.

5 - Testes e Validação

Este capítulo detalha a estratégia de validação da solução *QAlytics*, focando-se na precisão do motor de análise semântica e na viabilidade operacional de correr modelos de linguagem de grande escala (LLM) em hardware local. O guião detalhado de testes, com os passos de execução e resultados esperados, encontra-se documentado no Anexo B.

5.1 Abordagem e Metodologia de Testes

Dada a natureza do projeto, optou-se por uma metodologia de testes baseada em **Ambientes Controlados (Sandboxing)**. Devido às restrições de privacidade de dados da empresa parceira, a validação não é efetuada em repositórios de produção, mas sim num repositório *dummy* que replica a complexidade de um sistema de gestão de reservas hoteleiras.

A estratégia foca-se em três eixos:

- **Validação Funcional:** Teste da capacidade do motor RAG em identificar as 6 categorias de anomalias desenhadas no *dataset* sintético.
- **Validação Operacional:** Monitorização do consumo de recursos (RAM e CPU/VRAM) e tempos de resposta do modelo local.
- **Validação por *Stakeholders*:** Recolha de *feedback* qualitativo junto das equipas de QA da empresa parceira.

5.2 Análise de Riscos

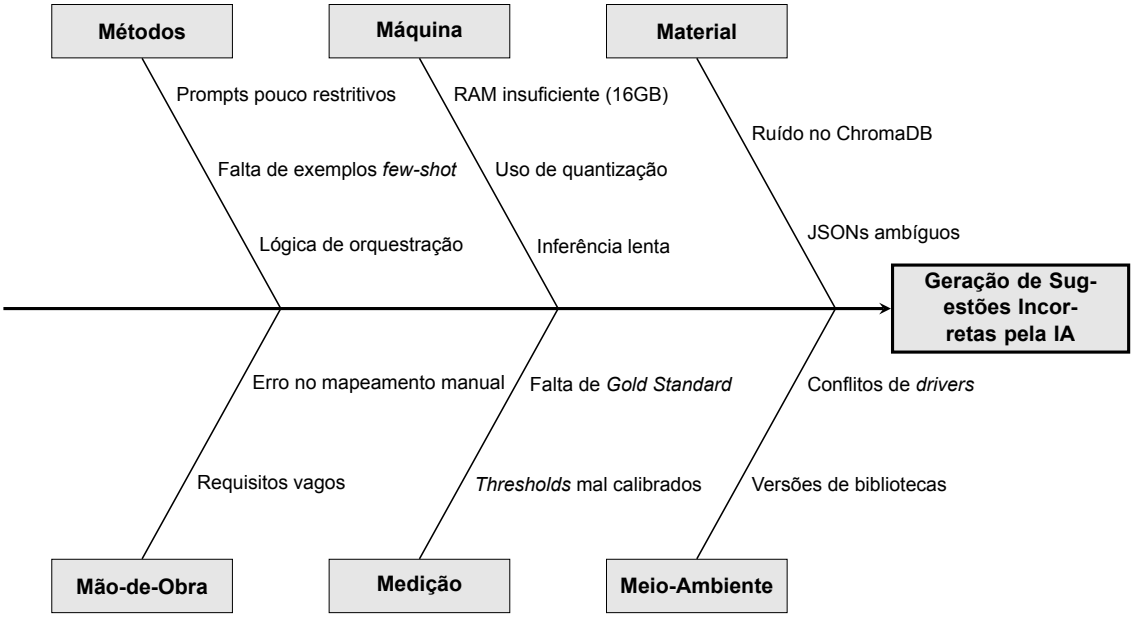


Figure 5.1: Diagrama de Ishikawa: Causas para a geração de resultados incorretos.

Para identificar as causas raízes que podem levar ao maior risco crítico do sistema — a **Geração de Sugestões Incorretas (Alucinação ou Falso Positivo)** — foi elaborado um Diagrama de Causa-Efeito (Ishikawa). Esta análise técnica permitiu categorizar as fontes de incerteza em seis eixos fundamentais:

- **Método:** Refere-se às falhas na engenharia de *prompts*. A ausência de exemplos *few-shot* ou *System Prompts* insuficientemente restritivos podem levar a que o modelo não respeite o esquema JSON de saída ou ignore restrições lógicas de negócio.
- **Máquina (Hardware):** As limitações de hardware local (16 GB de RAM) constituem um risco operacional, pois obrigam à utilização de modelos com maior taxa de quantização. Isto pode reduzir a capacidade de raciocínio lógico e a precisão da inferência comparativamente a modelos de maior escala.
- **Material (Dados):** Focado no ruído proveniente da recuperação de documentos (*Retrieval*). Se o *ChromaDB* injetar contextos irrelevantes ou *User Stories* semanticamente distantes no *prompt*, a IA poderá basear a sua decisão em dados incorretos.
- **Mão-de-Obra (Fator Humano):** A ambiguidade na redação original das *User Stories* ou dos casos de teste no Azure DevOps. Linguagem vaga por parte dos analistas pode dificultar a interpretação semântica tanto pelo motor vetorial como pelo LLM.
- **Medição:** Refere-se à calibração dos *thresholds* (limiares) de similaridade na busca vetorial. Um limiar mal ajustado pode permitir a entrada de ”ruído” no contexto ou, inversamente, bloquear informações cruciais para a análise.
- **Meio-Ambiente:** Inconsistências de ambiente técnico, tais como variações entre versões do motor *Ollama* ou conflitos entre as bibliotecas de processamento JSON, que podem causar comportamentos não-determinísticos na solução.

5.3 Validação de Recursos e Performance

A transição para o modelo `qwen2.5-coder:14b-instruct` permitiu uma melhoria significativa na compreensão de código e lógica, mas impôs desafios operacionais severos.

Table 5.1: Métricas de Performance Operacional (Hardware Local)

Métrica	**Valor Observado**
Consumo de Memória RAM (Pico)	$\approx$ 15 GB / 16 GB Total
Tempo Médio de Execução (6 Cenários)	30 a 40 minutos
Tempo Alvo (Hardware de Produção)	$<$ 1 minuto
Precisão Manual (Sem Ruído RAG)	100%

O elevado tempo de execução é justificado pela execução em CPU/GPU de consumo, mas valida a possibilidade de operação 100% *offline*, cumprindo o requisito de segurança de dados.

5.4 Resultados Experimentais e Validação por Terceiros

Através da apresentação do repositório *dummy* e dos cenários de teste à equipa de QA da empresa, obteve-se validação sobre a pertinência do problema. Os cliente validaram os cenários usados para testar o projeto.

6 - Método e Planeamento

Este capítulo descreve a metodologia de trabalho adotada pela equipa ao longo do projeto, detalha a divisão de responsabilidades técnicas e apresenta uma análise crítica da execução face ao planeamento original. O cronograma inicial (Fase 1), submetido na primeira entrega intercalar, encontra-se preservado no Anexo C para memória futura e avaliação de progresso do júri.

6.1 Planeamento Inicial

6.1.1 Metodologia de Trabalho

O desenvolvimento do projeto foi baseado numa abordagem ágil e iterativa, garantindo a flexibilidade necessária para adaptar requisitos face aos obstáculos técnicos encontrados e ao *feedback* dos clientes.

Para a gestão diária do projeto, a equipa recorreu a ferramentas digitais de gestão de tarefas (organizadas visualmente em quadros *Kanban*) e plataformas de documentação partilhada, permitindo um acompanhamento centralizado do estado de desenvolvimento de cada componente.

O alinhamento e o fluxo de comunicação foram assegurados através de três dinâmicas principais:

- **Reuniões de *Follow-up*:** Duas sessões rápidas semanais de sincronização interna da equipa, focadas no alinhamento de tarefas, partilha de impedimentos técnicos e planeamento a curto prazo.
- **Reuniões com o Departamento de Testes:** Reuniões com os profissionais de QA (os utilizadores finais da ferramenta) para compreender os seus processos diários, recolher *feedback* técnico e garantir que a solução mitigava problemas reais.
- **Sessões de DEMO:** Reuniões de demonstração com a empresa, com o objetivo de apresentar o estado atual do desenvolvimento, validar hipóteses e garantir que o *Minimum Viable Product* (MVP) correspondia à visão e expectativas do cliente.

A nível de engenharia, adotamos uma divisão clara de responsabilidades (*Separation of Concerns*), baseada nos diferentes perímetros tecnológicos da arquitetura:

- **Camada de Infraestrutura e Dados:** Um elemento focou-se em exclusivo na infraestrutura de dados, sendo responsável pela implementação, gestão e otimização da base de dados relacional de auditoria (SQLite) e do motor vetorial RAG (ChromaDB).
- **Camada de Lógica Semântica e Interface:** O outro elemento focou-se na orquestração central, assumindo a responsabilidade pela integração da API do GitHub, construção da interface interativa (`CLI`) e engenharia de *prompts* na comunicação com o LLM local.

6.1.2 Cronograma e Distribuição de Esforço Original (Fase 1)

O planeamento do primeiro semestre, submetido na altura da 1ª Entrega Intercalar (antes do início prático do estágio e do contacto aprofundado com o cliente), focava-se na construção de um MVP estritamente baseado em *parsing* de código (AST).

O esforço total foi inicialmente estimado e distribuído por 5 *Sprints* principais, cobrindo a extração de código, a construção do grafo de dependências e a integração com um LLM para casos ambíguos. Dado que o âmbito do projeto sofreu uma alteração profunda após as primeiras demonstrações de *software*, a listagem exaustiva desse planeamento original (com as respetivas *User Stories* e *Story Points*) foi transposta na íntegra para o Anexo C, preservando o histórico para efeitos de auditoria.

6.2 Análise Crítica ao Planeamento

Esta secção documenta a evolução real do desenvolvimento, as dificuldades encontradas na validação de requisitos e o estado atual da solução à data da 2ª Entrega Intercalar.

6.2.1 O Desafio do Âmbito e o *Pivot* Tecnológico

O desenvolvimento prático iniciou-se em meados de janeiro, seguindo o plano original focado na Análise Estática de código. A equipa implementou com sucesso o *parser* completo e a sua integração inicial com um LLM local.

Contudo, no final de março, durante uma sessão de DEMO fundamental com o cliente, ocorreu um desalinhamento de expectativas clássico em engenharia de *software*. Ao visualizar a solução aplicada a *gaps* de código, o cliente clarificou que a sua visão para os ”testes” residia, na verdade, na validação semântica de artefactos de gestão ágil (*User Stories* e *Test Cases* em formato JSON, exportados de plataformas como o Azure DevOps), e não na leitura de *scripts* de automação em Python ou Java.

Embora o plano inicial estivesse cumprido até àquela data, a equipa tomou a decisão ágil de realizar um *pivot* total no âmbito da aplicação para garantir a satisfação do cliente e o valor real do MVP. O motor AST foi descartado, obrigando à reconstrução da lógica central de IA e à adoção de uma arquitetura baseada em RAG e ChromaDB. Parte do trabalho de infraestrutura foi reaproveitado, adaptando o `GitHubClient` para a extração e *parsing* de ficheiros JSON.

6.2.2 Estado Atual do Desenvolvimento (2ª Entrega Intercalar)

Apesar da alteração no planeamento, o *pivot* tecnológico foi executado com sucesso. À data da submissão desta entrega intercalar, o repositório do projeto apresenta o seguinte estado de maturidade:

- **Implementação Concluída:** O `GitHub Client` e a extração estruturada de dados JSON estão funcionais. O `Core AI Service` (motor de *prompts* e validação Pydantic), a interface `CLI` e a formatação visual rica de *User Stories* e *Test Cases* encontram-se operacionais.
- **Implementação Parcial:** A integração final do motor vetorial (`ChromaDB`) e o armazenamento do histórico de auditoria (`SQLite`) estão em fase de consolidação.
- **Segurança e *Mocking*:** Por rigorosas razões de segurança e proteção de dados, a solução não está ainda a ser executada em repositórios reais da empresa. A equipa construiu um repositório *dummy* isolado, com dados sintéticos representativos, para validar os cenários de teste da IA.

6.3 Planeamento Futuro (Próximos Passos)

Até à entrega final do projeto (junho/julho), o cronograma da equipa foca-se na estabilização dos componentes parcialmente implementados e na preparação do *deployment*.

O trabalho divide-se nas seguintes frentes:

1. **Consolidação RAG e Base de Dados:** Fechar o fluxo de vetorização no ChromaDB e a persistência final das decisões do utilizador no SQLite.
2. **Testes Finais:** Concluir a validação das matrizes de decisão geradas pela IA face aos dados do repositório *dummy*.
3. **Dockerização:** Preparar a imagem Docker com o projeto pronto a executar.

6.3.1 *Parking Lot* (Funcionalidades Extra e Visão Futura)

Durante as recentes reuniões de alinhamento, o cliente propôs uma série de funcionalidades bastante interessantes para adicionar pós MVP. Estas foram catalogadas no *Parking Lot* do projeto para desenvolvimento futuro:

- **Modo *Chat* Interativo:** Possibilidade de o QA ”conversar” com a IA sobre os *Test Cases* em tempo real.
- **Automação Contínua (*Scripting*):** Conversão da CLI para atuar como um *daemon* ou *script* CI/CD, capaz de processar ininterruptamente novos testes à medida que são adicionados ao repositório.

Caso o tempo disponível até à entrega final o permita, a equipa iniciará a investigação em torno do Modo *Chat*.

A - Anexo: Especificação Detalhada dos Requisitos

Este anexo apresenta a descrição técnica detalhada, os fluxos de processo, as dependências e os critérios de validação para os requisitos funcionais de maior complexidade do **QAlytics**, adaptados à arquitetura *Retrieval-Augmented Generation* (RAG) delineada no Capítulo 3.

A.1 RF1 e RF2 - Ingestão e Higienização de Artefactos

Capacidades do Sistema (*DataSanitizer*):

- Autenticar via *Token* em repositórios (ex: GitHub, Azure DevOps).
- Navegar na estrutura de diretórios e extrair ficheiros estruturados em formato JSON (*User Stories* e *Test Cases* legados).
- Filtrar chaves e metadados irrelevantes do JSON original (ex: datas de criação, *tags* de sistema).
- Isolar e formatar apenas os campos com peso semântico, nomeadamente `System.Description` e `Custom.TestSteps`.

Processo de Negócio (Fluxo de Ingestão):

1. Utilizador invoca o comando de configuração e fornece o *token* de acesso.
2. Sistema acede ao repositório alvo e descarrega os ficheiros JSON para memória.
3. O *DataSanitizer* processa cada ficheiro, removendo ”ruído” estrutural.
4. Os artefactos limpos são preparados e formatados para a fase de vetorização.

Critérios de Validação:

- Capacidade de ler e processar centenas de JSONs sem esgotar a memória local.
- O texto higienizado resultante retém 100% da lógica de negócio e 0% de metadados inúteis.

A.2 RF3 e RF4 - Vetorização e Avaliação Semântica (RAG)

Técnicas de Processamento (*VectorStorage* e *SemanticAnalyzer*):

- **Vetorização Local:** Uso do modelo `all-MiniLM-L6-v2` para converter os artefactos higienizados em *embeddings* matemáticos, armazenando-os no ChromaDB.
- **Pesquisa de Contexto (*Retrieval*):** Cálculo de similaridade (*cosine similarity*) para encontrar o top K de *User Stories* e testes legados semanticamente mais próximos do novo teste submetido.

- **Inferência LLM (Ollama):** Submissão do novo teste acoplado ao contexto recuperado para a instância local do Llama 3.1, solicitando a identificação de conflitos ou redundâncias.

Cenários de Uso Típicos (Avaliação Semântica):

- *Cenário 1 (Duplicação):* Novo teste valida ”Cancelamento de Reserva”, mas a BD vetorial encontra o `TC-142` que valida ”Abortar Reserva”. A intenção é idêntica.
- *Cenário 2 (Contradição Lógica):* Novo teste afirma que o utilizador ganha pontos extra (VIP) numa compra básica. A *User Story* recuperada indica que isso só acontece em compras Premium. O teste viola os requisitos.
- *Cenário 3 (Atualização Justificada):* Novo teste cobre a mesma *User Story* que testes legados, mas introduz passos mais completos e atualizados. Os testes antigos tornam-se candidatos a refatorização.

A.3 RF5 e RF6 - Matriz de Decisão e Explicabilidade Direcionada

Componentes Obrigatórios da Resposta (Validado via Pydantic): Para garantir utilidade prática e evitar alucinações, cada sugestão gerada pela IA deve incluir obrigatoriamente a seguinte estrutura JSON:

1. **Expected Status:** Ação primária categorizada de forma estrita (`Ok`, `Refactor` ou `Obsolete`).
2. **Expected Target:** O alvo da ação sugerida (`New Test`, `Old Tests` ou `None`).
3. **Expected IDs (Evidências):** Lista exata das referências cruzadas (ex: `US-12`, `TC-127`) que a IA utilizou para fundamentar a sua conclusão.
4. **Justificação em Linguagem Natural:** Um parágrafo conciso gerado pelo modelo local (Llama 3.1) explicando o raciocínio semântico (ex: ”Este novo teste duplica a lógica de validação de NIF já coberta de forma idêntica pelo TC-45”).

Critérios de Validação:

- O sistema rejeita e processa falhas caso o LLM tente devolver uma resposta fora da estrutura exigida pela validação do Pydantic.
- A justificação é gerada em tempo útil (compatível com o *hardware* local) e os IDs apontados existem efetivamente na base de dados vetorial.

B - Guião Detalhado de Testes e Resultados

Este anexo detalha os 6 cenários críticos validados no ambiente *dummy*, desenhados para cobrir os principais padrões de análise semântica e ”armadilhas” lógicas na manutenção contínua de *Test Cases* (TC).

Table B.1: Matriz de Execução de Cenários de Teste

ID	Descrição do Cenário	Resultado Esperado	Estado
TC-141	**Happy Path (Kiosk):** Adiciona etapas de verificação de identidade e concordância de termos ao processo de *check-in* num quiosque, expandindo o teste original (TC-101) com asserções mais profundas.	**Refactor** (Melhora o teste antigo)	Passou
TC-142	**Duplicate (Abort Booking):** Descreve exatamente o mesmo comportamento do TC-109 (cancelamento $>$48h para reembolso total), utilizando apenas terminologia distinta (*sinónimos*: ”abort” vs ”cancel”).	**Obsolete** (Deteta cópia exata)	Passou
TC-143	**Logical Conflict (Refund 24h):** Contradição direta de regras de negócio. A *User Story* dita uma penalidade de 1 noite para cancelamentos $<$48h, mas o novo teste espera um reembolso total às 24h.	**Refactor + New Test** (Contradição assinalada)	Passou
TC-144	**Out of Scope (Avis Rental):** O teste valida a adição de um aluguer de carro (produto distinto), enquanto a *User Story* associada se refere exclusivamente a pedidos especiais como camas extra ou berços.	**Obsolete** (Out of Scope)	Passou
TC-145	**Internal Flaw (Checkout Invoice):** Valida a correção interna (cálculo de fatura) de uma funcionalidade já testada (TC-119: *split payment*), não introduzindo novas capacidades para o utilizador.	**Ok + None** (Validação interna, por enquanto definido pelo cliente como fora do âmbito do projeto)	Passou
TC-146	**Refactor (VIP Platinum):** Cobre o mesmo cenário do TC-127 (*upgrade* automático), mas adiciona complexidade e profundidade (verificação de atribuição de *suite* específica e *check-out* às 16h).	**Refactor** (Melhora o teste TC-127)	Passou

Síntese de Validação

A execução deste plano de testes atestou o funcionamento da arquitetura com **100% de acerto** nos cenários isolados avaliados manualmente. O sucesso alcançado prova o elevado nível de maturidade e de *Prompt Engineering* aplicado, sendo que o motor gerativo desenvolvido se revelou *domain-agnostic* — ou seja, é aplicável a qualquer indústria de *software* sem necessidade de reajuste das regras ou padrões.

A solução validou de forma robusta e inequívoca a sua capacidade de:

- Identificar *enhancements* profundos a testes previamente existentes.
- Detetar duplicações semânticas, demonstrando imunidade a ofuscações por sinónimos.
- Identificar rigorosamente contradições face às regras de negócio das *User Stories*.
- Filtrar eficazmente testes irrelevantes ou *Out of Scope*.
- Distinguir validações mecânicas/internas de novas capacidades do utilizador.
- Processar múltiplas *User Stories* e cruzar contexto em simultâneo de forma coerente.

C - Anexo: Planeamento e Especificação da Primeira Entrega

Nota de enquadramento: O conteúdo deste anexo é um artefacto histórico submetido na 1ª Entrega Intercalar. Ele reflete a visão, os requisitos e as User Stories do projeto antes do "pivot" arquitetural detalhado no Capítulo 6. O planeamento aqui exposto focava-se na análise estática de código (AST) em Python/Java, tendo o âmbito atual evoluído para a análise semântica de artefactos ágeis (JSON) através de Inteligência Artificial generativa (RAG).

C.1 Análise de Requisitos da Primeira Entrega

A análise de requisitos da primeira entrega visava definir o escopo funcional e as restrições técnicas do projeto focado na leitura de código fonte de testes automatizados.

C.1.1 Requisitos Funcionais

A Tabela C.1 apresenta os requisitos funcionais inicialmente identificados.

Table C.1: Requisitos Funcionais do Sistema (Planeamento da Primeira Entrega)

ID	Requisito	Descrição	Prioridade
RF1	Integração Git	Aceder a repositórios GitHub/GitLab via API para leitura de código e histórico.	MUST
RF2	Análise Código (Prod)	Extrair estrutura (classes, métodos) através de parsing AST.	MUST
RF3	Análise Código (Teste)	Identificar testes (pytest/JUnit) e suas dependências.	MUST
RF4	Identif. Testes Obsoletos	Detetar testes que validam código removido ou inexistente.	MUST
RF5	Identif. Testes Redundantes	Detetar testes que cobrem cenários duplicados.	SHOULD
RF6	Análise do Impacto	Identificar testes afetados por *commits* recentes.	SHOULD
RF7	Justificações	Gerar explicação clara e evidências (links/diffs) para cada sugestão.	MUST
RF8	Score de Confiança	Atribuir nível de confiança (0-100%) a cada sugestão.	MUST
RF9	Interface de Validação	Dashboard (CLI) para aprovar/rejeitar sugestões.	MUST
RF10	Sistema de Feedback	Registar decisões do utilizador para melhoria contínua.	SHOULD
RF11	Histórico de análises	Manter registo de análises e métricas de evolução.	COULD

C.1.2 Requisitos Não-Funcionais

As restrições de qualidade e operação originais encontram-se na Tabela C.2.

Table C.2: Requisitos Não-Funcionais (Planeamento da Primeira Entrega)

ID	Categoria	Métrica de Aceitação
RNF1	Performance de Análise	Análise de 500 testes deve completar em $<$ 10 minutos.
RNF2	Taxa de Precisão	Precisão $\geq$ 85% na identificação de obsoletos (minimizar falsos positivos).
RNF3	Taxa de Cobertura	Sistema deve analisar 100% dos testes detetados.
RNF4	Autenticação Segura	Autenticação OAuth2; Tokens encriptados; Sem armazenamento de credenciais.
RNF5	Privacidade de Dados	Código fonte processado em memória, sem persistência em disco.
RNF6	Explicabilidade	100% das sugestões devem ter justificação em linguagem natural.
RNF7	Suporte a frameworks	Suporte inicial a Python (pytest), extensível a Java.

C.2 User Stories da Primeira Entrega (Por Épicos)

Para facilitar o desenvolvimento incremental inicial, os requisitos foram decompostos nas seguintes *User Stories*.

C.2.1 Épico 1: Integração e Configuração

US1 - Autenticação com Repositório

Como Developer/QA Engineer
Quero conectar a ferramenta ao meu repositório GitHub através de OAuth
Para que o sistema possa analisar o código e testes sem comprometer a segurança.
Critérios de Aceitação:

- Sistema redireciona para GitHub OAuth quando clico em ”Conectar Repositório”.
- Após autorizar, vejo lista dos meus repositórios acessíveis.
- Token é armazenado de forma segura (encriptado).
- Se o token expirar, o sistema renova automaticamente ou pede re-autenticação.

US2 - Seleção de Repositório e Configuração

Como utilizador autenticado
Quero selecionar qual repositório analisar e configurar opções básicas
Para focar a análise no projeto relevante.
Critérios de Aceitação:

- Consigo procurar/filtrar na lista de repositórios.
- Posso selecionar *branch* específica (ou usar *main* por defeito).
- Posso escolher *frameworks* de teste (pytest, JUnit, ou ambos).
- Configurações são guardadas para a próxima sessão.

C.2.2 Épico 2: Análise de Código e Testes

US3 - Análise Inicial do Repositório

Como utilizador
Quero que o sistema analise automaticamente o código e testes do repositório
Para obter uma visão estrutural do projeto.
Critérios de Aceitação:

- Sistema identifica todos os ficheiros de código (.py, .java).
- Sistema identifica todos os ficheiros de teste.
- Vejo progresso da análise (barra de progresso ou *spinner*).
- Recebo notificação quando a análise completa.
- Análise de repositório com 500 testes completa em < 10 minutos.

US4 - Identificação de Testes Obsoletos

Como QA Engineer
Quero que o sistema identifique automaticamente testes que testam código que já não existe
Para saber quais testes posso remover com segurança.
Critérios de Aceitação:

- Sistema lista todos os testes identificados como obsoletos.
- Cada teste obsoleto tem justificação clara (ex: método X foi removido).
- Vejo *score* de confiança para cada identificação.
- Precisão $\geq 85\%$ validada em testes conhecidos.
- Falsos positivos $< 15\%$.

US5 - Identificação de Testes Redundantes

Como Developer
Quero que o sistema identifique testes que cobrem os mesmos cenários
Para eliminar duplicações e otimizar tempo de execução.
Critérios de Aceitação:

- Sistema agrupa testes similares/redundantes.
- Vejo comparação lado-a-lado de testes redundantes.
- Sistema sugere qual teste manter (mais completo/recente).
- Análise considera tanto estrutura quanto semântica.

US6 - Análise de Impacto de Mudanças

Como Developer
Quero saber que testes foram impactados por mudanças recentes no código
Para saber quais testes preciso de rever/atualizar.
Critérios de Aceitação:

- Posso especificar um *commit* ou intervalo de *commits*.
- Sistema lista testes afetados pelas mudanças.
- Vejo explicação de porque cada teste foi impactado.
- Análise considera dependências diretas e indiretas.

C.2.3 Épico 3: Visualização e Tomada de Decisão

US7 - Dashboard de Sugestões

Como utilizador
Quero ver todas as sugestões organizadas num *dashboard* claro
Para ter visão geral dos problemas identificados.
Critérios de Aceitação:

- Vejo estatísticas: total de sugestões, por tipo, por confiança.
- Consigo filtrar por tipo (obsoleto/redundante/impactado).
- Consigo filtrar por nível de confiança (alta/média/baixa).
- Consigo ordenar por diferentes critérios.
- Interface carrega em < 2 segundos.

US8 - Visualização de Detalhes da Sugestão

Como QA Engineer
Quero ver justificação detalhada e evidências para cada sugestão
Para tomar decisão informada sobre aprovar ou rejeitar.
Critérios de Aceitação:

- Vejo explicação completa em linguagem natural.
- Vejo código do teste com *syntax highlighting*.
- Vejo *links* clicáveis para *commits/diffs* relevantes.
- Vejo histórico de modificações do teste.
- Consigo navegar entre sugestões sem voltar à lista.

US9 - Aprovação e Rejeição de Sugestões

Como utilizador
Quero aprovar ou rejeitar cada sugestão individualmente
Para manter controlo sobre que ações são tomadas.
Critérios de Aceitação:

- Botões claros para ”Aprovar” e ”Rejeitar”.

- Posso adicionar comentário opcional à decisão.
- Sistema pede confirmação antes de aprovar (especialmente remoções).
- Vejo *feedback* visual de que a ação foi registada.
- Decisão é guardada imediatamente.

C.2.4 Épico 4: Feedback e Melhoria Contínua

US11 - Sistema de Feedback

Como utilizador
Quero dar *feedback* sobre qualidade das sugestões
Para ajudar a melhorar o sistema ao longo do tempo.
Critérios de Aceitação:

- Posso classificar sugestão como ”Útil” / ”Não útil”.
- Posso reportar falso positivo com justificação.
- *Feedback* é registado e associado à sugestão.
- Vejo mensagem de agradecimento após submeter *feedback*.

US12 - Histórico de Análises

Como Tech Lead
Quero ver histórico de análises anteriores e evolução ao longo do tempo
Para monitorizar melhoria da qualidade da *suite* de testes.
Critérios de Aceitação:

- Vejo lista de análises anteriores com data/hora.
- Posso comparar resultados entre duas análises.
- Vejo métricas: quantos testes foram removidos, taxa de aprovação.
- Posso exportar dados para relatórios.

C.2.5 Épico 5: Segurança e Privacidade

US13 - Processamento Seguro

Como utilizador preocupado com segurança
Quero garantia de que o código não é armazenado permanentemente
Para cumprir políticas de segurança da empresa.
Critérios de Aceitação:

- Sistema processa código apenas em memória.
- Após análise, nenhum código permanece em disco.
- *Logs* não contêm *snippets* de código sensível.
- Apenas metadados são armazenados (nomes, localizações).
- Documentação de segurança disponível.

C.3 Cronograma e Distribuição de Esforço da Primeira Entrega

O planeamento do primeiro semestre (Setembro a Novembro 2025) focou-se exclusivamente na definição do MVP (*Minimum Viable Product*).

Abaixo apresenta-se a distribuição das *User Stories* e componentes pelos *Sprints* a realizar:

Sprint 1: Infraestrutura e Integração

Foco: Configuração do ambiente e conexão com GitHub.

- **US1** - Autenticação com Repositório (5 SP)
- **US2** - Seleção de Repositório e Configuração (3 SP)
- **US13** - Processamento Seguro em Memória (5 SP)
- *Componentes:* `GitHubClient`, Estrutura do Projeto.

Esforço Total: 13 SP

Sprint 2: Análise Core (AST)

Foco: Parsing de código e identificação de estruturas.

- **US3** - Análise Inicial do Repositório (8 SP)
- **US6** - Análise de Impacto de Mudanças (13 SP)
- **Tecnologia:** Implementação do *parser* AST para Python e mapeamento de classes/métodos.
- *Componentes:* `PythonParser`, `DependencyGraph`.

Esforço Total: 21 SP

Sprint 3: Lógica de Obsolescência e IA

Foco: Algoritmo de deteção e integração com LLM.

- **US4** - Identificação de Testes Obsoletos (13 SP)
- **US5** - Identificação de Testes Redundantes (13 SP)
- **Tecnologia:** Integração com OpenAI API para casos ambíguos.
- *Componentes:* `ObsoleteFinder`, `AIService`.

Esforço Total: 26 SP

Sprint 4: Interface e Persistência

Foco: Interação com o utilizador e armazenamento de resultados.

- **US7** - Dashboard de Sugestões (CLI) (8 SP)
- **US8** - Visualização de Detalhes e Evidências (8 SP)
- **US9** - Aprovação/Rejeição de Sugestões (5 SP)
- *Componentes:* `SQLiteStorage`, Interface CLI (`Click+Rich`).

Esforço Total: 21 SP

Sprint 5: Consolidação e Entrega

Foco: Testes finais, documentação e vídeo demonstrativo.

- Refinamento de *prompts* do LLM.
- Produção do Relatório Intercalar.
- Gravação do vídeo de demonstração do MVP.

D - Declaração de Uso de Ferramentas de IA

De acordo com o regulamento do Trabalho Final de Curso, apresenta-se abaixo o preenchimento da declaração obrigatória de uso de ferramentas de Inteligência Artificial.

1. Utilização de IA

☐ Não foram utilizadas ferramentas de IA na realização deste trabalho.

☒ Foram utilizadas ferramentas de IA na realização deste trabalho.

2. Ferramentas utilizadas

Assistência geral à escrita, análise ou ideação

☐ ChatGPT

☐ Microsoft Copilot

☒ Gemini

☐ Claude

☐ Perplexity

☐ Outras. Quais? _____

Assistência à programação / desenvolvimento

☒ GitHub Copilot

☐ Claude

☐ OpenAI Codex

☐ Cursor

☐ Tabnine

☐ Amazon CodeWhisperer / Amazon Q

☐ Outras. Quais? _____

Geração de imagem / design / multimédia

☐ DALL·E

☐ Midjourney

☐ Stable Diffusion

☐ Canva AI / Magic Design

☐ Outras. Quais? _____

3. Fases do trabalho em que foi utilizada IA

☒ Planeamento do trabalho

☒ Pesquisa exploratória / levantamento inicial de informação

☒ Documentação técnica

☒ Redação do relatório

☒ Desenho / modelação / arquitetura

☒ Design / prototipagem / interface

☒ Geração de código

☒ Revisão / refatoração / debugging de código

☒ Criação de testes / casos de teste

☒ Análise de resultados

☒ Preparação de apresentação ou materiais auxiliares

4. Tipo de utilização

A Inteligência Artificial (nomeadamente Gemini e GitHub Copilot) foi utilizada como uma ferramenta de apoio à engenharia e redação ao longo do ciclo de vida do projeto:

- **Planeamento e Pesquisa:** *Brainstorming* inicial para o cronograma da 1ª entrega, identificação de tecnologias viáveis (ex: ChromaDB) e sumarização de documentação oficial das bibliotecas usadas.
- **Implementação e Debugging:** Geração de esqueletos de funções isoladas em Python, resolução de erros no terminal (*troubleshooting*) e apoio na construção da interface visual da CLI (consulta de documentação das bibliotecas `Typer` e `Rich`).
- **Testes e Validação:** Geração estruturada de dados sintéticos (*datasets dummy* em formato JSON) para testar o motor RAG sem expor dados reais da empresa parceira, e apoio na interpretação dos *outputs* brutos do LLM (*Ollama*).
- **Documentação e Relatório:** Criação das estruturas iniciais dos ficheiros `README.md`, geração de *docstrings* (posteriormente validadas), e assistência extensiva na redação, reestruturação narrativa e formatação do presente relatório em LaTeX, incluindo a geração de código *Mermaid* para os diagramas de arquitetura.

5. Partes do trabalho afetadas

A intervenção das ferramentas de IA afetou tanto a componente prática como a documental do projeto:

- **Componente Prática e Tecnológica:** O repositório de testes *dummy* (geração do *dataset* sintético em JSON), a estruturação visual e tabelas da interface de linha de comandos (`CLI`), esqueletos de funções do motor RAG (integração de `Pydantic` e APIs locais) e a documentação técnica do código (*docstrings* e `README.md`).

- **Componente Documental (Relatório):** Geração do código *Mermaid* para os diagramas arquiteturais, revisão narrativa e formatação nativa em LaTeX. O impacto foi mais expressivo na reestruturação dos seguintes capítulos: Capítulo 3 (Especificação e Modelação), Capítulo 4 (Solução Desenvolvida) e Capítulo 6 (Método e Planeamento) devido à mudança no âmbito.

6. Exemplos de prompt

Abaixo apresentam-se exemplos representativos dos comandos utilizados em diferentes fases. *(Nota: Alguns prompts longos foram truncados por questões de formatação).*

- **Fase de Criação de Testes (Gemini/ChatGPT):** *"Act as a Senior Software Test Architect. I need a synthetic dataset in pure JSON format to test a RAG pipeline. Generate a single, large JSON with three main arrays: user_stories, existing_test_cases, and new_test_cases. CRITICAL: You MUST strictly use the nested Azure DevOps JSON schema [...] [Instruções detalhadas de formatação omitidas]. Create 6 new trap test cases with intentional flaws (Happy Path, Duplicate, Logical Conflict, Out of Scope, Mathematical Flaw, Refactor). Return ONLY the valid JSON."*
- **Fase de Arquitetura (Gemini):** *"Cria o código Mermaid (orientação Left-to-Right) para um diagrama de blocos que mostre o fluxo de dados desde a CLI, passando por um Orquestrador, até ao ChromaDB e ao Ollama."*
- **Fase de Implementação (Copilot):** *"Cria uma docstring para todas as funções do projeto que não a tenham."*

7. Validação, revisão e intervenção dos autores

Toda a informação e texto gerados pela IA no âmbito documental foram lidos de forma crítica, validados e editados pelos autores para garantir a sua total adequação à realidade dos factos e ao histórico do projeto. Nenhum texto foi transposto sem revisão e reescrita humana.

No que diz respeito à componente técnica (geração de código auxiliar, *datasets* JSON e *prompts* aplicados ao Ollama/Llama 3.1),o funcionamento foi extensivamente validado através da execução prática do código e da análise manual dos *outputs* na CLI, utilizando exclusivamente o *dataset* sintético no repositório *dummy*. De forma a garantir a segurança da informação da empresa parceira, nenhum código gerado por IA foi testado contra repositórios ou dados reais do cliente.

8. Grau de utilização

☐ Residual

☒ Moderado

☐ Extensivo

☐ Utilização homogénea

☒ Grau de uso diferenciado por fase ou componente de trabalho

Descrição: O uso foi intensivo na reestruturação e formatação do relatório em LaTeX e na geração do *dataset* sintético de testes (JSON). Na vertente de desenvolvimento, o uso foi pontual (focado em tarefas auxiliares como *docstrings* e formatação visual da CLI), sendo reduzido ou nulo na implementação da lógica *core* e infraestrutura de dados (orquestração do RAG, integração com API do GitHub, ChromaDB e SQLite).

9. Trabalhos em parceria

☒ O trabalho foi realizado em parceria com entidade externa ao DEISI.

☒ O parceiro tem regras para restringir submissão de dados.

☒ As submissões validam aplicação de regras de tratamento de dados.

☒ Foram implementados mecanismos para restringir a partilha de recursos proprietários. *(Uso de repositórios dummy fechados e IA Local Zero-Data Retention).*

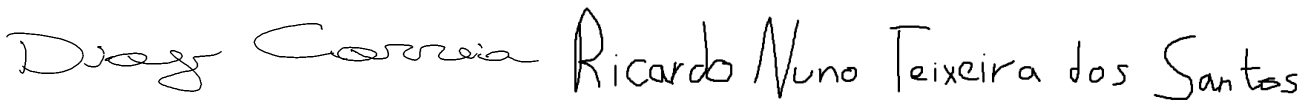
10. Declaração de responsabilidade e Assinaturas

Ao assinarem a presente declaração, os autores declaram que a informação acima é verdadeira e reflete o uso efetivo de ferramentas de IA; compreendem que a IA não substitui autoria; validaram as referências e assumem integralmente a responsabilidade técnica, científica, ética e académica por todo o conteúdo submetido.

Nome(s): Diogo Correia, Ricardo Santos
Número(s): a22207097, a22207422

Data: 12 / 04 / 2026

Assinaturas:



Diogo Correia

Ricardo Santos

Bibliografia

[Mat24] João P. Matos-Carvalho. *The Lusófona LaTeX Template User's Manual*. Lusófona University. 2024. URL: https://github.com/jpmcarvalho/UL-Thesis.

[RH01] Gregg Rothermel and Mary Jean Harrold. "Regression test selection for C++ software". In: *Software Testing, Verification and Reliability* 11.4 (2001), pp. 203–217.

[EMR00] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. "Test case prioritization: A family of empirical studies". In: *Proceedings of the 22nd international conference on Software engineering*. 2000, pp. 464–473.

[Mes07] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Referência clássica sobre "Test Smells" e manutenção de testes. Pearson Education, 2007.

[YH12] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: a survey". In: *Software Testing, Verification and Reliability*. Vol. 22. 2. Wiley. 2012, pp. 67–120.

[Pyt24a] Python Software Foundation. *Python Language Reference, version 3.11*. Acedido em: Abril de 2026. 2024. URL: https://docs.python.org/3.11/.

[Ram24] Sebastián Ramírez. *Typer: Build great CLIs. Easy to code. Based on Python type hints.* Acedido em: Abril de 2026. 2024. URL: https://typer.tiangolo.com/.

[McG24] Will McGugan. *Rich: Python library for rich text and beautiful formatting in the terminal*. Acedido em: Abril de 2026. 2024. URL: https://rich.readthedocs.io/.

[PyG24] PyGithub Team. *PyGithub: Typed interactions with the GitHub API v3*. Acedido em: Abril de 2026. 2024. URL: https://pygithub.readthedocs.io/.

[Chr24] Chroma. *ChromaDB: The AI-native open-source embedding database*. Acedido em: Abril de 2026. 2024. URL: https://docs.trychroma.com/.

[Rei24] Nils Reimers. *Sentence-Transformers: Multilingual Sentence, Image, and Context Embeddings*. Acedido em: Abril de 2026. 2024. URL: https://www.sbert.net/.

[Oll24] Ollama. *Ollama: Get up and running with large language models locally*. Acedido em: Abril de 2026. 2024. URL: https://ollama.com/.

[Met24] Meta AI. *Introducing Llama 3.1: Our most capable models to date*. Acedido em: Abril de 2026. 2024. URL: https://ai.meta.com/blog/meta-llama-3-1/.

[Col24] Samuel Colvin. *Pydantic: Data validation using Python type hints*. Acedido em: Abril de 2026. 2024. URL: https://docs.pydantic.dev/.

[SQL24] SQLite Development Team. *SQLite: Small, Fast, Reliable, Choose any three*. Acedido em: Abril de 2026. 2024. URL: https://www.sqlite.org/docs.html.

[Pyt24b] Python Software Foundation. *Python Documentation: ast — Abstract Syntax Trees*. Acedido em: Abril de 2026. 2024. URL: https://docs.python.org/3/library/ast.html.

[Ope24] OpenAI. *OpenAI API Documentation*. Acedido em: Abril de 2026. 2024. URL: https://platform.openai.com/docs/introduction.

[Gur24] Gurock Software. *TestRail: Comprehensive Test Case Management*. Acedido em: Abril de 2026. 2024. URL: https://www.testrail.com/.

[Xra24] Xray. *Xray: Native Test Management for Jira*. Acedido em: Abril de 2026. 2024. URL: https://www.getxray.app/.