Universidade Católica do Salvador
Bacharelado em Engenharia de Software

Gustavo Peixoto de Oliveira Dias
Iann Luca Rocha Lino
Leon Nascimento Moreira
Ruan Luis Matos Brandão Suarez
Ubirajara do Rosário Santana Júnior

Asterix: Proposta para redução da carga cognitiva de desenvolvedores em sistemas legados em Java

Salvador
2026

Gustavo Peixoto de Oliveira Dias
Iann Luca Rocha Lino
Leon Nascimento Moreira
Ruan Luis Matos Brandão Suarez
Ubirajara do Rosário Santana Júnior

Asterix: Proposta para redução da carga cognitiva de desenvolvedores em sistemas legados em Java

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como parte dos requisitos necessários para a obtenção do Título de Engenheiro de Software.
Orientador: Prof. Angela Peixoto

Universidade Católica do Salvador

Salvador
2026

Gustavo Peixoto de Oliveira Dias
Iann Luca Rocha Lino
Leon Nascimento Moreira
Ruan Luis Matos Brandão Suarez
Ubirajara do Rosário Santana Júnior

Asterix: Proposta para redução da carga cognitiva de desenvolvedores em sistemas legados em Java

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como requisito parcial para a obtenção do título de Engenheiro de Software.

Salvador, 24 de junho de 2026
Banca Examinadora:

Prof. Angela Peixoto
Universidade Católica do Salvador
Orientador

Prof. Me. Luan Galvão
Universidade Católica do Salvador

Prof. Me. Cristiana Bispo
Universidade Católica do Salvador

Agradecimentos

Eu, Gustavo Dias, aos meus pais, Paulo André e Gabriela Carina, devo tudo. Foram vocês que sustentaram cada uma das minhas escolhas, que me deram condições de estudar mesmo quando isso exigiu esforço e renúncia, e que me mostraram, no dia a dia, o tipo de pessoa que eu queria me tornar. Tudo o que eu conquistar daqui em diante vai carregar um pouco de vocês. Às minhas avós, que estiveram ao meu lado em cada momento e nunca deixaram de me apoiar, deixo um agradecimento especial — o apoio de vocês foi um alicerce ao longo de todo esse caminho. Estendo essa gratidão a toda a minha família, presente em cada fase dessa jornada. Aos amigos que fiz durante o curso, obrigado pela companhia nas longas noites de estudo e por transformarem uma rotina pesada em algo que valeu a pena. À professora Angela, nossa orientadora, agradeço pela condução precisa, pela paciência e pelo cuidado em cada etapa deste trabalho — essenciais para que ele chegasse até aqui. A todos que fizeram parte desse percurso, o meu mais sincero obrigado.

Eu, Iann Luca Rocha Lino, agradeço aos meus pais, Alessandro e Janisa, a quem dedico a maior parte desta conquista. O apoio incondicional, a paciência e os valores que me transmitiram foram o alicerce para que eu chegasse até aqui. Sem o suporte de vocês em cada etapa dessa jornada, a conclusão não seria possível. Ao meu irmão, Italo, por ser uma presença tão importante na minha vida, pela parceria e por me apoiar ao longo de todo esse caminho. À minha orientadora, Angela, pela paciência, pelas correções precisas e por guiar o desenvolvimento deste trabalho com tanta dedicação. Aos demais professores do curso, que compartilharam seus conhecimentos, exigiram o meu melhor e me prepararam para o mercado. Aos meus amigos de faculdade, obrigado pela companhia indispensável. Sobreviver aos trabalhos, projetos complexos e viradas de noite foi muito mais fácil e divertido com vocês ao meu lado. Por fim, agradeço a todos que, direta ou indiretamente, me apoiaram, torceram por mim e ajudaram a transformar esse objetivo em realidade.

Eu, Leon Nascimento Moreira, quero começar agradecendo a Deus, por sempre cuidar de mim e me guiar pelos melhores caminhos. Em segundo lugar, agradeço à meu pai Reginaldo Brito e minha mãe Maria Nascimento, por terem me dado total suporte durante todos esses anos e por nunca ter me desamparado. Sem esses primeiros citados, eu não estaria aqui como estou hoje. Também quero agradecer aos meus colegas de classe, por terem tornado essa fase mais leve e tranquila. Sem eles, esse caminho teria sido muito mais difícil. Por último, deixo meus agradecimentos aos professores que se dedicaram, prestaram suporte e me motivaram. A esses, agradeço por todo o apoio, dentro e fora da sala de aula, especialmente à professora Angela,

agradeço pela paciência e pela dedicação na condução deste trabalho.

Eu, Ruan Brandão Suarez, agradeço em primeiro lugar a Deus, por permitir chegar até aqui, concedendo-me forças e sabedoria para possibilitar superar as mais diversas adversidades. Aos meus pais, Halime e Luiz por, através de tantas lutas, terem me proporcionado boa educação e por serem minha base - sem vocês, nada do que conquistei hoje seria possível. Aos meus colegas de faculdade, agradeço pelos vários dias de alegria, parceria e trabalhos em conjunto. À nossa orientadora, Angela Peixoto, muito obrigado por toda a dedicação, parceria, orientação e confiança dedicadas a nós para o desenvolvimento deste trabalho. Por fim, a todos que fizeram parte dessa caminhada, obrigado!

Eu, Ubirajara Santana, agradeço primeiramente a Deus, pela força, sabedoria e por guiar meus passos em cada etapa dessa jornada. Aos meus pais, Ubirajara Santana e Barbara Santana, pelo amor incondicional, pelos sacrifícios e pela educação exemplar que sempre dedicaram a mim e à minha irmã. Vocês são a minha base. Aos meus avós, pelo carinho, pelas orações e pelo apoio que sempre me fortaleceu. Aos colegas de faculdade que se tornaram grandes amigos, pelos desafios compartilhados, pelas madrugadas de trabalho e pela parceria que levarei para a vida. À minha orientadora, Angela, pela paciência, condução brilhante e ensinamentos fundamentais para o desenvolvimento deste trabalho. Por fim, agradeço a todos que, direta ou indiretamente, fizeram parte deste ciclo, que representa um marco para a minha vida e para a minha carreira.

”Preocupar-se com um único galho fará com que você perca a árvore de vista. Preocupar-se com a árvore fará com que você perca a floresta inteira. Não se fixe em apenas um ponto.”

(Eiji Yoshikawa)

Resumo

A manutenção de sistemas legados e a inserção de novos desenvolvedores nesses cenários, o chamado processo de *onboarding*, representam desafios críticos na Engenharia de Software, frequentemente agravados em decorrência da ausência de documentação atualizada e pela alta complexidade estrutural. Este cenário eleva, de maneira drástica, a carga cognitiva dos profissionais, que gastam energia mental de maneira excessiva decifrando sintaxes complexas e arquiteturas desorganizadas. A fim de mitigar tal problemática, este trabalho propõe e descreve o desenvolvimento do Asterix, uma plataforma inteligente voltada à documentação automatizada e análise estrutural de projetos legados em Java. A solução apresenta uma abordagem ativa que combina o mapeamento de sistemas legados para Árvores Sintáticas Abstratas (*AST*) com o Processamento de Linguagem Natural (*PLN*) proporcionado por Grandes Modelos de Linguagem (*LLMs*). A metodologia engloba o processo de mapeamento e limpeza de código-fonte legado, o processo de conversão deste em uma estrutura hierárquica e de fácil compreensão e o envio para o agente gerar a documentação. Os resultados demonstram a possibilidade da ferramenta em processar o código e fornecer modelos mentais objetivos e de fácil compreensão. Por fim, conclui-se que o sistema reduz de maneira substancial a fadiga mental associada à exploração do código, permitindo que o desenvolvedor redirecione o seu foco cognitivo para a compreensão do domínio, acelerando e reduzindo o processo de *onboarding*.

Palavras-Chave: Inteligência Artificial, Sistemas Legados, Documentação, Carga Cognitiva

Abstract

The maintenance of legacy systems and the integration of new developers into these environments — the onboarding process — represent critical challenges in Software Engineering, frequently exacerbated by the lack of up-to-date documentation and high structural complexity. This scenario drastically elevates the cognitive load of professionals, who expend excessive mental energy deciphering complex syntaxes and disorganized architectures. To mitigate this issue, this work proposes and describes the development of Asterix, an intelligent platform aimed at the automated documentation and structural analysis of legacy Java projects. The solution presents an active approach that combines the mapping of legacy systems into Abstract Syntax Trees (AST) with the Natural Language Processing (NLP) provided by Large Language Models (LLMs). The methodology encompasses the mapping and cleaning of legacy Java source code, its conversion into a hierarchical and easily comprehensible structure, and its subsequent processing by the AI agent to generate the final documentation. The results demonstrate the tool possibility of processing the code and providing mental models and easy comprehension. Finally, it is concluded that the system substantially reduces the mental fatigue associated with code exploration, allowing the developer to redirect their cognitive focus toward domain understanding, thereby accelerating and reducing the onboarding process.

Keywords: Inteligência Artificial, Legacy Systems, Documentation, Cognitive Load

Lista de figuras

Lista de Siglas e Abreviaturas

ACID	*Atomicity, Consistency, Isolation, Durability*
API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
BES	Bacharelado em Engenharia de Software
CQRS	*Command Query Responsibility Segregation*
CSS	*Cascading Style Sheets*
DDD	*Domain-Driven Design*
DER	*Diagrama Entidade Relacionamento*
HTML	HyperText Markup Language
HTML5	*HyperText Markup Language 5*
HTTP	*Hypertext Transfer Protocol*
HTTPS	*HyperText Transfer Protocol Secure*
IA	Inteligência Artificial
JIT	*Just-in-Time*
JPA	*Java Persistence API*
JVM	*Java Virtual Machine*
JWT	*JSON Web Token*
LLM	*Large Language Models*
LSTM	*Long Short-Term Memory*
MVP	*Minimum Viable Product*
NPM	*Node Package Manager*
RA	*Research Answer*
ReAct	*Reasoning and Acting*
RF	Requisito Funcional
RNF	Requisito Não Funcional
RNN	*Recurrent Neural Networks*
RQ	*Research Question*
SGBDOR	Sistema Gerenciador de Banco de Dados Objeto-Relacional
SQL	*Structured Query Language*
TCC	Teoria da Carga Cognitiva
UI	*User Interface*
UK	*Unique Key*
UML	*Unified Modeling Language*
URL	*Uniform Resource Locator*
UX	*User Experience*

SEO	*Search Engine Optimization*
SRP	*Single Responsibility Principle*
WORA	*Write Once Run Anywhere*

Sumário

1 Introdução

O mundo da Engenharia de *Software* vem, com o passar dos anos, ganhando cada vez mais espaço no mercado e no cotidiano da sociedade de maneira exponencialmente acelerada, estando presente nas tarefas mais simples do dia a dia até as de maior complexidade. Entretanto, com este avanço rápido, muitos sistemas e aplicações acabam não acompanhando a consequente evolução da área, tornando-se defasados e difíceis de manter a longo prazo, mesmo aquelas ainda operantes de maneira crítica, sendo *softwares* considerados legados.

De acordo com Monteiro e Vieira (2022), mesmo em sistemas grandes e duráveis, a mudança é necessária para os manter úteis, sendo seu maior desafio encontrar soluções que aliem qualidade e custo-benefícios para aprimorar as aplicações e atender aos novos requisitos necessários. Dentre as alternativas existentes para possibilitar a atualização e refatoração de sistemas legados, a inserção de desenvolvedores nestes contextos apresenta-se como principal meio de viabilizar a implementação de melhorias, o chamado processo de *onboarding*. Apesar disso, este processo torna-se muitas vezes dificultoso, tendo em vista a natureza dos sistemas legados, principalmente se ausentes de documentação - regras de negócio engessadas, comentários sem necessidade semântica, métodos e atributos sem utilização, hierarquia confusa e muitas vezes defasada entre classes e alto acoplamento a torna uma atividade complexa e intelectualmente exigente ao desenvolvedor, configurando alta carga cognitiva necessária para a executar.

Diante desse cenário, é proposto o desenvolvimento do sistema *Asterix*, voltado para o mapeamento, estruturação e documentação de código legado na linguagem de programação Java, uma das principais do mercado, com o objetivo de reduzir a alta carga cognitiva necessária para desenvolvedores em processo de inserção (*onboarding*) no contexto destes sistemas.

1.1 Objetivo Geral

O objetivo geral do presente trabalho é o desenvolvimento de uma ferramenta com foco na redução da carga cognitiva de desenvolvedores em processo de *onboarding* em sistemas legados na linguagem Java, através do processo de documentação automatizada, aliando etapas de limpeza de código e mapeamento de hierarquias com o uso de modelos de linguagem de grande porte (*LLMs*) para fornecer ao desenvolvedor uma documentação rica em detalhes técnicos do código-fonte a ser analisado.

Diferentemente de outras abordagens já existentes para documentação de códi-

gos, o *Asterix* realiza o mapeamento hierárquico do código para fornecer ao LLM uma estrutura de melhor compreensão, análise e entendimento pelo mesmo, não se restringindo ao levantamento puramente estático e sintático existente em demais soluções no mercado. Para possibilitar atingir tais objetivos, será desenvolvido um MVP (Mínimo Produto Viável) do sistema, utilizando de tecnologias modernas e robustas para possibilitar a criação de uma aplicação escalável e com produto final de qualidade, permitindo o envio de projetos legados compactados para serem documentados e fornecidos aos usuários.

1.2 Objetivos Específicos

- Apresentar pontos-chave que corroboram para o aumento da carga cognitiva necessária para desenvolvedores em processo de *onboarding* em sistemas legados, levando a processos como re-engenharia e manutenção mais demorados e exaustivos.
- Desenvolver uma ferramenta capaz de mapear códigos Java, estruturar e gerar documentação das classes nele implementadas, destacando detalhes técnicos como hierarquia, funcionalidades, regras de negócio e lógica existentes.
- Documentar o desenvolvimento do *Asterix*, apresentando detalhes técnicos da implementação e o resultado obtido - a documentação técnica e descritiva do sistema legado.

1.3 Estrutura do Trabalho

Neste capítulo foram apresentados o contexto, a motivação, objetivos, estratégias de pesquisa, e contribuições associados a este trabalho de conclusão de curso. O capítulo 2 apresenta a fundamentação teórica necessária para embasamento e compreensão do presente artigo. O capítulo 3 apresenta a metodologia utilizada para confeccionar o presente trabalho de conclusão de curso e as questões de pesquisa levantadas mediante o tema. O capítulo 4 apresenta detalhes técnicos do desenvolvimento da aplicação, abordando, em diferentes níveis, as diferentes tecnologias utilizadas e integradas que, juntas, tornaram possível a execução do sistema proposto. Já o capítulo 5 aborda, mediante utilização de imagens de autoria própria e suas respectivas descrições, as telas do sistema *Asterix* de maneira a simular a utilização por um usuário. Por fim, no capítulo 6 é apresentada a conclusão e os possíveis trabalhos futuros envolvendo a temática e sistema desenvolvido no presente trabalho de pesquisa.

2 Fundamentação Teórica

Este capítulo tem como objetivo apresentar e contextualizar os conceitos fundamentais para a compreensão da temática abordada neste trabalho. Inicialmente, são discutidos os principais aspectos relacionados a sistemas legados, Inteligência Artificial e LLMs, bem como suas aplicações no contexto da Engenharia de *Software*.

2.1 Linguagem Java

Desenvolvido no inicio dos anos 90, o Java é uma linguagem de programação orientada a objetos, de tipagem forte e de alto nível. Inicialmente, foi pensada como uma nova alternativa similar ao C++ mas sem toda a ”sujeira”das seções e com uma estrutura mais ”amigável”, tendência das demais linguagens que estavam emergindo na época (HAMAAMIN; ALI; KAREEM, 2024).

Um dos grandes diferenciais da linguagem Java é a sua natureza *Write Once Run Anywhere (WORA)* - através do desenvolvimento de um único código, é possível executá-lo em diferentes sistemas e ambientes, tudo isso graças à *JVM (Java Virtual Machine)*, atuando como uma camada intermediária que interpreta o código compilado (o *bytecode*), bastando apenas ter a JVM instalada na maquina a qual se executa o código. De acordo com Martinez, Remegio e Lincopinis (2023), devido a esta natureza, o desenvolvimento de aplicações e o *deploy* em diversas plataformas se torna mais fácil utilizando a linguagem, sendo, devido a isso, muito utilizada em *Big Techs* como a *Google* e a *Amazon*.

Devido a estas características e a massiva adoção desta tecnologia por grandes empresas, inúmeras aplicações e sistemas foram desenvolvidos a partir da década de 90, período em que foi concebida, para os mais diversos usos. Ademais, o suporte contínuo da comunidade e a evolução da própria linguagem possibilitou a sua consolidação como uma das principais linguagens de programação para uso geral (DIMITRIJEVIC; ZDRAVKOVIC; MILICEVIC, 2023).

Entretanto, dada as suas diversas aplicações e as inúmeras situações as quais os códigos foram concebidos, a utilização de boas práticas de programação e a atualização para versões mais recentes da linguagem tornam-se ações não prioritárias, o qual, a longo prazo, dificultam a manutenibilidade da aplicação, tornando-as legadas.

2.2 Sistemas Legados

Por definição, sistemas legados são plataformas, aplicações e *Softwares* desenvolvidos no passado, utilizando tecnologias compatíveis para a época, mas que não foram devidamente atualizados para abordagens modernas e continuam a ser utilizados por organizações. São aplicações que operam há muitos anos, geralmente críticas ao negócio, construídas utilizando tecnologias ou paradigmas defasados (SOUZA, 2025).

Apesar desta natureza, a reescrita manual desses sistemas muitas vezes não é algo viável para o negócio, uma vez que, ao longo do tempo, regras de negócio vitais e refinadas vão sendo encapsuladas, podendo perder-se em meio a tantas lógicas, conforme explicitado por Annela (2025) - processos tradicionais de reescrita de código legado apresentam interrupções substanciais na regra de negócio, estouros de orçamento e longos prazos de implementação.

Diante disto, a manutenção e até mesmo a refatoração progressiva dessas aplicações apresentam-se como opções viáveis em contraste com a re-engenharia completa destas. Nesse sentido, o fator humano, representado pelos desenvolvedores em si, entra em cena, dado o alto grau de complexidade e acoplamento entre módulos, principalmente se ausentes de uma documentação devida.

2.2.1 Carga Cognitiva

A Teoria da Carga Cognitiva (TCC) foi proposta por Sweller (2011), e diz respeito a como o esforço ao realizar uma tarefa acaba impactando o sistema cognitivo humano, especificamente analisando os limites da memória de trabalho durante a aprendizagem. De acordo com a teoria, a carga cognitiva exigida no processamento de informações pode ser dividida em duas categorias básicas:

- Carga Cognitiva Intrínseca: Compreende a complexidade natural do conhecimento a ser adquirido, não levando em consideração como ele é ensinado. Esta carga é determinada pela interatividade dos elementos da informação, e não pode ser alterada - a menos que se mude o que está sendo aprendido ou o nível de especialização do profissional.

- Carga Cognitiva Extrínseca: Representa a maneira como a informação é apresentada ao profissional, ocorrendo quando o contexto exige que o mesmo processe informações demasiadas e, muitas vezes, desnecessárias, em memória de trabalho, interferindo diretamente no aprendizado.

Tal teoria, quando aplicada ao contexto da Engenharia de *Software*, discorre sobre como desenvolvedores tem maior gasto cognitivo para compreender artefatos de soft-

ware, como códigos-fonte, diagramas arquiteturais e modelos UML (*Unified Modeling Language*). (GONÇALES, 2022).

Ainda de acordo com Gonçales (2022), é possível analisar, através de manifestações fisiológicas nos desenvolvedores, sinais de alta carga cognitiva, tais como alterações na atividade cerebral, temperatura da pele e frequência cardíaca. Segundo experimentos realizados por Crk e Kluthe (2016), o aumento da carga cognitiva de desenvolvedores se relacionam diretamente com a redução do nível de expertise dos mesmos. Ademais, em Samuel (2025), é enfatizado que a pesquisas da carga cognitiva nas interações humano-computador destacam a importância de apresentação de informações de maneira alinhada com o limite de memória dos desenvolvedores, reforçando a importância da atenção à esta temática.

Nesse sentido, profissionais recém integrados a uma equipe ou *software* - em processo de *onboarding* - se deparam com aplicações complexas e de difícil leitura, muitas vezes ausentes até mesmo de documentação. A ausência de um conhecimento prévio e de um guia sobre tais sistemas acabam por elevar a sua carga cognitiva, dado o esforço necessário para entendimento e posterior execução das tarefas necessárias.

2.3 Documentação de Código

Historicamente, a documentação de código desempenha um papel mister para transferência de conhecimento entre equipes de *software* e desenvolvedores no geral, ela é um dos principais artefatos existentes no contexto de desenvolvimento. De acordo com Rai, Belwal e Gupta (2022), a documentação de código-fonte é um material essencial para o entendimento do sistema durante a fase de manutenção, principalmente em trechos importantes e críticos do código.

Para Dvivedi et al. (2024) a documentação de código refere-se ao processo de descrever a funcionalidade, propósito e uso do código através de textos, comentários e anotações, desempenhando papel indispensável na Engenharia de *Software* como um todo, ao promover clareza, legibilidade e eficiência a todo o processo de desenvolvimento.

Diante desse cenário, é seguro afirmar que a documentação é uma peça chave para o desenvolvimento no geral. No contexto de sistemas legados, este artefato serve como ponte principal entre o desenvolvedor e o código-fonte, através da explicação de lógicas, relacionamento e regras de negócio dificeis e extremamente específicas. Quando o código-fonte carece de uma clara descrição de funções, classes e dependências, processos como o *onboarding* de desenvolvedores e manutenção tornam-se dificultosos, por configurar-se como uma operação de alto custo e incertezas.

2.4 Inteligência Artificial

A evolução da inteligência artificial generativa, impulsionada pela arquitetura *Transformers*[1], criou um ponto de inflexão no ambiente da Engenharia de *Software*. Diferentemente das ferramentas de análises estáticas tradicionais, as LLMs não processam apenas sintaxe, mas demonstram uma compreensão semântica profunda de linguagens de programação. Essa característica é o que permite a mitigação do débito cognitivo no desenvolvimento de *software*. Segundo Ziegler et al. (2022), a IA generativa atua na redução da carga cognitiva ao automatizar tarefas de baixo nível, o que "permite que o desenvolvedor mantenha o foco na arquitetura e na lógica de alto nível", acelerando a curva de aprendizado de novos integrantes em sistemas complexos.

Essa capacidade de interpretação contextual, que distingue as LLMs das ferramentas convencionais, fundamenta-se na arquitetura de redes neurais denominada *Transformers*. Introduzida por Vaswani et al. (2017) no artigo *"Attention Is All You Need"*, essa arquitetura rompeu com os modelos sequenciais anteriores ao introduzir o mecanismo de atenção (*Self-Attention*). Diferente das arquiteturas recorrentes, que processam informações de forma linear, os *Transformers* permitem que o modelo analise todos os elementos de um bloco de código simultaneamente. Na prática da engenharia de software, isso significa que a IA consegue identificar relações de dependência entre variáveis e funções, mesmo que estejam distantes, capturando a lógica global do sistema. Dessa forma, as LLMs deixam de ser meros geradores de código para se tornarem sistemas capazes de mapear a estrutura semântica do *software*.

2.4.1 Uso de IA na documentação de código

A sumarização de código, ou *Code Summarization*, consiste no processo de geração automática de descrições legíveis em linguagem natural. O mecanismo atua de forma similar a um tradutor, extraindo a lógica técnica e convertendo-a em uma explicação funcional para os desenvolvedores. Este fluxo operacional fundamenta-se em três estágios essenciais:

- *Input*: Compreende a inserção do código-fonte bruto. Nesta etapa, a integridade ou legibilidade original do código não é um impedimento, permitindo que sistemas legados sejam processados.
- *Processing*: Representa a parte central, onde a LLM realiza a análise semântica e a extração da intenção lógica. Utilizando o mecanismo de atenção da arquitetura

[1] Os *Transformers* integram o campo do Aprendizado Profundo (*Deep Learning*), surgindo como uma evolução às arquiteturas recorrentes tradicionais ao permitir o processamento paralelo e não sequencial dos dados.

Transformers, o modelo identifica relações de dependência e o propósito daquele trecho de código.

- *Output*: É a entrega final do processamento, resultando em documentações estruturadas ou resumos explicativos prontos para o consumo humano.

Essa tríade é o que viabiliza a redução da carga cognitiva discutida anteriormente. Ao transformar o conhecimento implícito, muitas vezes ”preso” em algoritmos complexos e obsoletos, em conhecimento explícito e documentado, a sumarização de código atua como o motor principal para um *onboarding* eficiente e seguro em ambientes de software legados.

2.4.2 Uso de IA no desenvolvimento de softwares

A Engenharia de *Software* atravessa uma transformação profunda, onde a exigência para o desenvolvedor desloca-se da proficiência em *syntax* para uma atuação focada em arquitetura e processos de negócio. Nesse novo paradigma, a habilidade de traduzir requisitos complexos em sistemas sustentáveis torna-se mais valiosa do que o domínio mecânico da codificação.

Atualmente, observa-se a ascensão dos agentes de IA, como o *Claude Code*, *Codex* e *Antigravity*. Diferente dos assistentes de *chat* tradicionais, esses agentes operam com autonomia dentro do projeto, interagindo diretamente com o sistema de arquivos e o terminal. O *Claude Code*, da *Anthropic*, é um exemplo notável dessa evolução. De acordo com a Anthropic (2025), ele opera em um ciclo fechado (*agentic loop*), frequentemente isolado em *containers* por segurança, onde a tarefa só é considerada finalizada após a validação do resultado.

O Fluxo de Operação do *Claude Code*

- Planejamento Semântico (*Gather Context*): Ao receber o *prompt*, o agente não gera código imediatamente. Ele realiza uma leitura exaustiva do contexto do projeto e elabora um plano de etapas sequenciais.
- Execução de Ferramentas (*Take Action*): O agente utiliza ferramentas de sistema para ler arquivos, executar comandos de *build* ou rodar testes unitários.
- Monitoramento e Correção (*Verify Results*): Se um comando resulta em erro, o agente analisa o *stack trace*, ajusta sua estratégia e reinicia o ciclo de execução. Esse comportamento persistente é o que garante a conclusão de tarefas complexas que exigiriam horas de depuração manual.

2.4.3 A IA como facilitadora de complexidade

O impacto desses agentes é particularmente visível em tarefas de alta complexidade técnica. Conforme apontam Fan et al. (2023), a utilização de modelos de linguagem permite que desafios de engenharia sejam reformulados como tarefas de tradução e análise, onde a IA atua como um catalisador de produtividade ao lidar com as minúcias técnicas que costumam sobrecarregar o desenvolvedor.

Um caso emblemático dessa facilitação é a linguagem *Rust*. Embora seja reconhecida por sua segurança de memória e performance, o *Rust* possui uma curva de aprendizado íngreme devido ao seu rigoroso sistema de *ownership* e *borrow checker*. A ascensão dos agentes de código tem impulsionado a adoção do *Rust* no mercado, uma vez que a IA consegue mediar essa complexidade, sugerindo implementações que respeitam as regras estritas da linguagem de forma automática. Iniciativas como o projeto *TRACTOR* da *DARPA* exemplificam essa tendência, utilizando IA para automatizar a migração de códigos legados em C para *Rust*, visando eliminar vulnerabilidades de memória de forma eficiente e segura (DARPA, 2024).

2.4.4 Conceituação de Agentes Autônomos de IA

Enquanto as aplicações iniciais de LLMs na engenharia de software operavam em um paradigma *prompt-resposta*, onde o usuário fornece uma entrada e o modelo gera uma saída, o cenário evoluiu para os Agentes de IA. Segundo Wooldridge (2020), um agente de software autônomo é um sistema de computador, capaz de realizar ações variáveis e autônomas nesse ambiente para atingir seus objetivos. Relacionado ao desenvolvimento de software, esse ambiente transita do chat textual para o repositório de código, o sistema de arquivos e o terminal de execução.

A arquitetura base de um agente de IA baseia-se em quatro pilares: perfilamento (*profiling*), memória (de curto e longo prazo), planejamento (*planning*) e o uso de ferramentas (*tool use*). Diferentemente de uma LLM, o agente utiliza o modelo de linguagem como um cérebro, mas é dotado de capacidades computacionais para interagir com o mundo externo, conforme ilustrado no fluxo operacional proposto pelo framework *ReAct* (*Reasoning and Acting*) de Yao et al. (2022).

O funcionamento de um agente autônomo de engenharia de software estrutura-se através de ciclos iterativos que superam as limitações de contexto e execução das LLMs tradicionais:

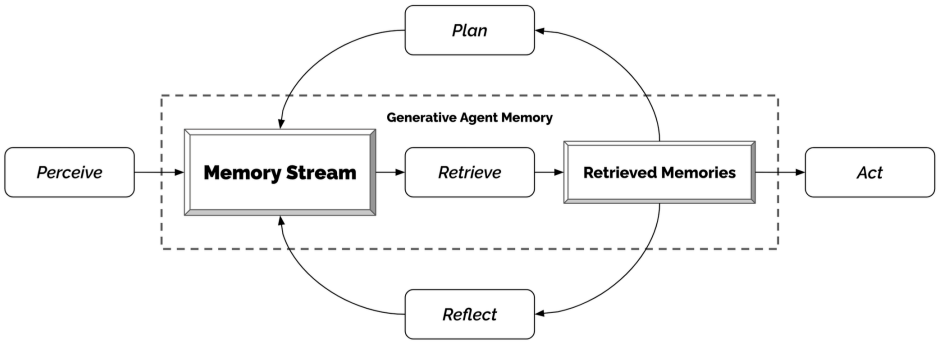


Figura 1 – Visão geral da arquitetura de um sistema de agentes.

Fonte: Park et al. (2023).

Em suma, a integração desses quatro pilares transforma a LLM estática em um componente dinâmico e interativo. Ao unificar a capacidade de planejamento à execução prática via ferramentas, o agente deixa de apenas sugerir trechos de código e passa a atuar de forma ativa no ecossistema de desenvolvimento.

2.5 React

Muito utilizada em inúmeras aplicações, o React é uma biblioteca da linguagem de programação JavaScript, voltada ao desenvolvimento do chamado *front-end*, a camada de interface do usuário de uma aplicação. De acordo com Boduch (2017), React faz parte da chamada camada *view* de um sistema, responsável por exibir dados ao usuário e por lidar com as interações realizadas pelo mesmo.

Para atingir tais objetivos eficientemente, esta biblioteca utiliza orientação a componentes - as interfaces mais complexas são divididas em partes menores, independentes e reutilizáveis. Essa abordagem não apenas facilita a manutenção e a escalabilidade do código ao longo do tempo, mas também otimiza o fluxo de trabalho do desenvolvimento, permitindo a padronização visual e estrutural da aplicação.

2.6 PostgreSQL

Tendo o início do seu desenvolvimento concebido no ano de 1986, fruto da evolução do Ingres (banco de dados utilizado anteriormente, na Universidade da Califórnia), o *PostgreSQL* é um sistema de gerenciamento de banco de dados relacional de objetos de código aberto (*SGBDOR*) conhecido por sua robustez, performance e conformidade com padrões SQL (*Structured Query Language*).

Dentre as principais vantagens e diferenças do PostgreSQL em relação a outras bases de dados como o MySQL e o Oracle, destacam-se a sua natureza em código aberto (*open-source*) e a sua gratuidade. Ademais, a sua capacidade de suporte para

SQL e NoSQL também são importantes diferenciais presentes em sua estrutura. Vale citar a sua conformidade nativa com os princípios *ACID* (Atomicidade, Consistência, Isolamento e Durabilidade), permitindo integridade de dados e melhor confiabilidade (RAVSHANOVICH, 2024).

2.7 Trabalhos Relacionados

Esta seção apresenta os principais estudos focados na utilização de IA em sistemas legados, e na mitigação da carga cognitiva durante o processo de *onboarding* de desenvolvedores. Serão apresentadas pesquisas que abordam desde técnicas tradicionais de análise de código-fonte até soluções que empregam LLMs para a geração de documentação de modernização de software. A análise destes trabalhos permitiu a identificação de lacunas na área, fundamentando as motivações e propostas deste trabalho.

A utilização de Inteligência Artificial em sistemas legados é muito discutida nas mais diversas abordagens. Em Araujo (2025), é realizada uma pesquisa voltada à identificação de qual LLM, dentre as selecionadas pelo autor, apresenta melhor desempenho para gerar documentação relevante através de comentários, voltados a códigos-fonte de sistemas legados em linguagens como Java, C++ e Cobol. O estudo analisou diversos modelos, e elencou através de métricas, como adequação de comentários, os que obtiveram melhor desempenho. Este estudo apresenta a capacidade dos LLMs de documentar códigos-fonte, principalmente em sistemas legados, destacando a possibilidade de otimização das entradas (*prompts*) para melhor interação com as ferramentas.

Ademais, o estudo de BOUKHARI, KHARMOUM e ZITI (2025) explora o uso do modelo *CodeLlama* para auxiliar na migração de sistemas legados para o padrão *CQRS* (*Command Query Responsibility Segregation*), mais moderno. Os autores propõem uma pipeline que converte o código-fonte em estruturas passiveis de classificação via tokenização, automatizando a geração de *handlers* com posterior avaliação humana. Entretanto, a ausência de uma estratégia clara de mapeamento estrutural de maneira prévia ao processamento pela IA evidencia uma lacuna crítica, uma vez que a compreensão da arquitetura e das regras de negócio é justamente um dos maiores gargalos para a redução da carga cognitiva durante o processo de *onboarding* em sistemas legados.

Através do levantamento destes estudos, é evidente que, apesar dos LLMs apresentarem alto potencial analítico, sua aplicação direta sobre o código-fonte bruto frequentemente resulta em artefatos de baixa manutenibilidade e incongruentes, evidenciando uma lacuna no mapeamento estrutural prévio do código. Deste modo, torna-se essencial a adoção de abordagens controladas e voltadas a extração da estrutura e

regras de negócio em formatos passíveis de melhor interpretação pelos LLMs, garantindo resultados mais precisos e coerentes.

Portanto, é proposta uma ferramenta cujo objetivo é automatizar sistematicamente e de maneira controlada, através de processos definidos em código, a documentação de sistemas legados na linguagem Java, com a finalidade de reduzir a carga cognitiva necessária para a inserção de novos desenvolvedores nestes cenários.

3 Metodologia

Este capítulo descreve os métodos utilizados para confecção do presente documento e desenvolvimento da aplicação proposta. O processo de pesquisa foi feito com base no objetivo de analisar e possibilitar o desenvolvimento de um sistema voltado à documentação de código legado, focando na redução da carga cognitiva para novos desenvolvedores - o processo de *onboarding*.

A partir deste ideal, iniciou-se a etapa de seleção de pesquisas relacionadas na plataforma Google Acadêmico. Foram utilizadas como termos de busca palavras-chave ligadas à temática da presente proposta, tais como ”*legacy systems and LLMs*”, ”*legacy code onboarding*”e ”*software cognitive load*”, as quais possibilitaram o levantamento, seleção, leitura e extração de informações mister para o desenvolvimento do trabalho.

Desta maneira, foi possível realizar o levantamento de requisitos do sistema, bem como definir questões como arquitetura a ser utilizada, escolher tecnologias para implementação e possibilitar a criação de um MVP. O sistema possibilita o envio de códigos-fonte legados, que passará por um processo de normalização, transformação e posterior análise por um LLM em um formato voltado à otimizar este processo, permitindo melhor entendimento inicial e instruções à novos desenvolvedores.

3.1 Questões de Pesquisa

Com base nas pesquisas selecionadas e analisadas, foi possível mapear os principais pontos relacionados à temática da presente proposta, bem como pontos de interesse e lacunas existentes. Diferentes abordagens são permeadas nos estudos, revelando, principalmente, pontos como a falta de tratativa eficaz de normalização de código legado antes de processamento e o alto grau de carga cognitiva expressada por desenvolvedores em cenários de desenvolvimento.

Portanto, o presente trabalho tem por objetivo, com base nos levantamentos e análises realizadas, propor uma solução com foco em mitigar a carga cognitiva de novos desenvolvedores sendo inseridos em contextos de sistemas legados, aliando o uso de LLMs para possibilitar melhores resultados de entrega e desempenho. Nesse sentido, algumas perguntas de pesquisa foram levantadas para auxiliar na produção desta pesquisa e do sistema.

- *Research Question 1 (RQ1)*: Como a extração e representação estrutural de códigos legados em Java impactam a precisão e a utilidade da documentação gerada por LLMs?

- *Research Question 2 (RQ2)*: De que maneira a documentação criada por IA atua na mitigação da carga cognitiva de novos desenvolvedores durante o processo de *onboarding*?
- *Research Question 3 (RQ3)*: De que maneira o *Asterix* se diferencia de outras ferramentas de documentação de códigos, como o *Doxygen*?

3.2 Processo de Pesquisa

O processo de pesquisa utilizado neste artigo foi estruturado de maneira sistemática, visando a garantia do rigor metodológico e a coerência entre os objetivos, os procedimentos técnicos e a construção dos resultados. A trajetória investigativa foi organizada em etapas sequenciais que envolveram: revisão da literatura sobre LLMs e Débito Técnico; definição das perguntas de pesquisa focadas em produtividade e carga cognitiva, escolha das abordagens metodológicas, desenvolvimento do fluxo de trabalho estruturado assistido por IA e a avaliação de sua eficácia na modernização de código e a integração de novos desenvolvedores.

Do ponto de vista técnico, foram aplicadas metodologias de Engenharia de *Software* para a implementação de um fluxo de trabalho assistido por agentes de IA. Este fluxo contemplou a análise de códigos legados, a geração de documentação semântica através de técnicas de *Code Summarization* e a aplicação de padrões de documentação de código de forma automática.

Para a última etapa do processo, foram conduzidos testes práticos com trechos de código de sistemas reais, possibilitando verificar o desempenho da ferramenta na redução da carga cognitiva e na precisão das explicações geradas. Esse processo permitiu a observação no comportamento das LLMs neste cenário, fornecendo evidências sobre a viabilidade do uso de agentes inteligentes para acelerar o *onboarding* e garantir a manutenibilidade de sistemas legados.

4 Desenvolvimento

Este capítulo descreve o processo de implementação e desenvolvimento do sistema proposto, uma plataforma de fácil acesso voltada à reduzir o esforço cognitivo de desenvolvedores em processo de inserção em novos projetos considerados legados através da disponibilização de documentação. Através desta, apresentam-se descrições do processo de planejamento e desenvolvimento dos principais artefatos, módulos e integrações do sistema.

4.1 Introdução

Em cenários de desenvolvimento, principalmente em empresas com muito tempo de mercado, é comum a existência de sistemas com papéis críticos e legados na linguagem Java, bem como a necessidade da inserção de desenvolvedores sem conhecimento prévio em tais sistemas para solucionar as mais diversas necessidades.

Diante disso, o sistema foi proposto e desenvolvido como uma plataforma que possibilita o envio de arquivos (no formato compactado *.zip*), passando por pré-processamento para, por fim, ser interpretado e devidamente documentado por um LLM, gerando documentação de fácil compreensão e didática, visando a redução da carga cognitiva exigida para desenvolvedores em *onboarding*.

4.2 Fluxo de Trabalho

Visando possibilitar a documentação, auditoria e melhor visibilidade da divisão de tarefas das diferentes frentes e o andamento dos fluxos de trabalho, foi utilizado o quadro *Kanban* para organizar os processos e metrificar o progresso do presente trabalho. O *Kanban* é uma metodologia japonesa que tem por objetivo possibilitar o mapeamento e acompanhamento de processos e tarefas através de um quadro - o chamado Quadro *Kanban*.

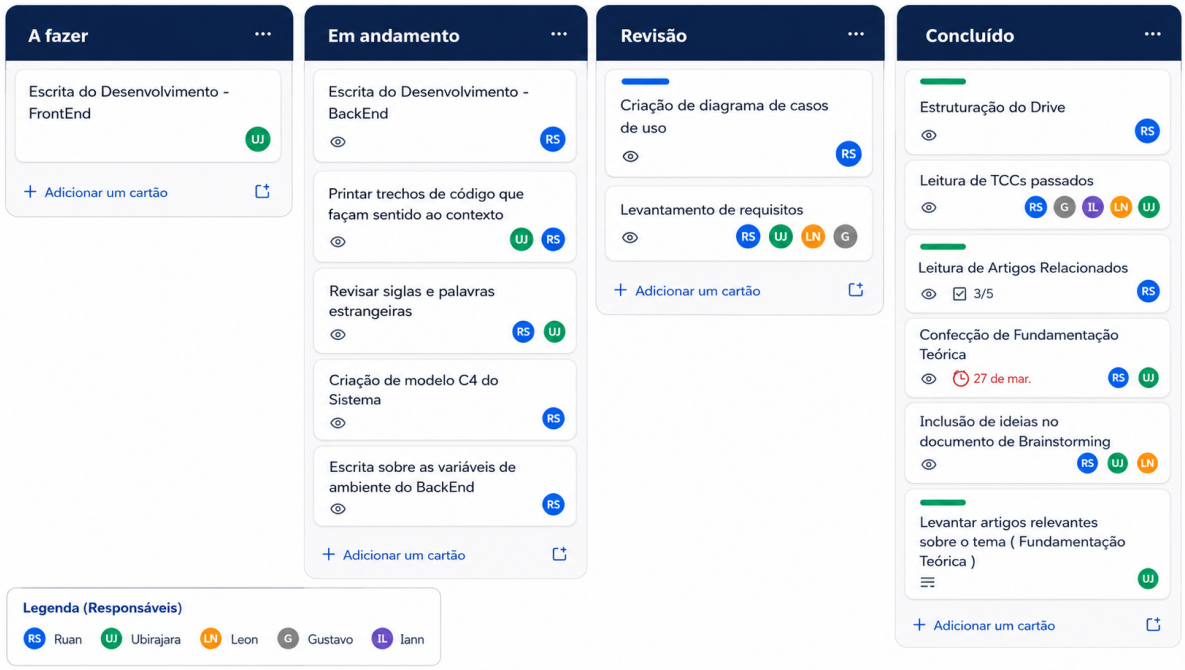
Segundo Hossain (2023), o *Kanban* prioriza flexibilidade, colaboração e aprimoramento continuo, sendo muito utilizado na indústria de desenvolvimento de *software*. Diante disso, foram utilizadas diferentes ferramentas para acompanhar as diferentes frentes de trabalho: o Trello com quadro *Kanban* para tarefas relacionadas ao artigo e o *Github Actions* para as diferentes frentes de desenvolvimento do sistema.

4.2.1 Trello

O Trello[1] é uma ferramenta para gestão de projetos e tarefas baseadas em quadros visuais, utilizando como base para tal o método *Kanban* para organizar os fluxos de forma simples e colaborativa.

Através de sua utilização, foi possível mapear, atualizar e auditar as tarefas de escrita do trabalho. Com a finalidade de facilitar a metrificação de progresso de escrita, foi divido o quadro em: A fazer; Em Andamento; Revisão e, por fim, Concluído, conforme apresentado na Figura 2

Figura 2 – Organização do Trello



Autoria própria

4.2.2 GitHub Actions

Para o controle e direcionamento das atividades, utilizou-se a plataforma de integração contínua *GitHub Actions*, que permite a automação e o gerenciamento do software. A ferramenta foi essencial para a operacionalização do fluxo de trabalho (*workflow*) baseado em um *Git Flow* enxuto, garantindo que as implementações em cada *branch* seguissem critérios de aceitação antes da sua integração final.

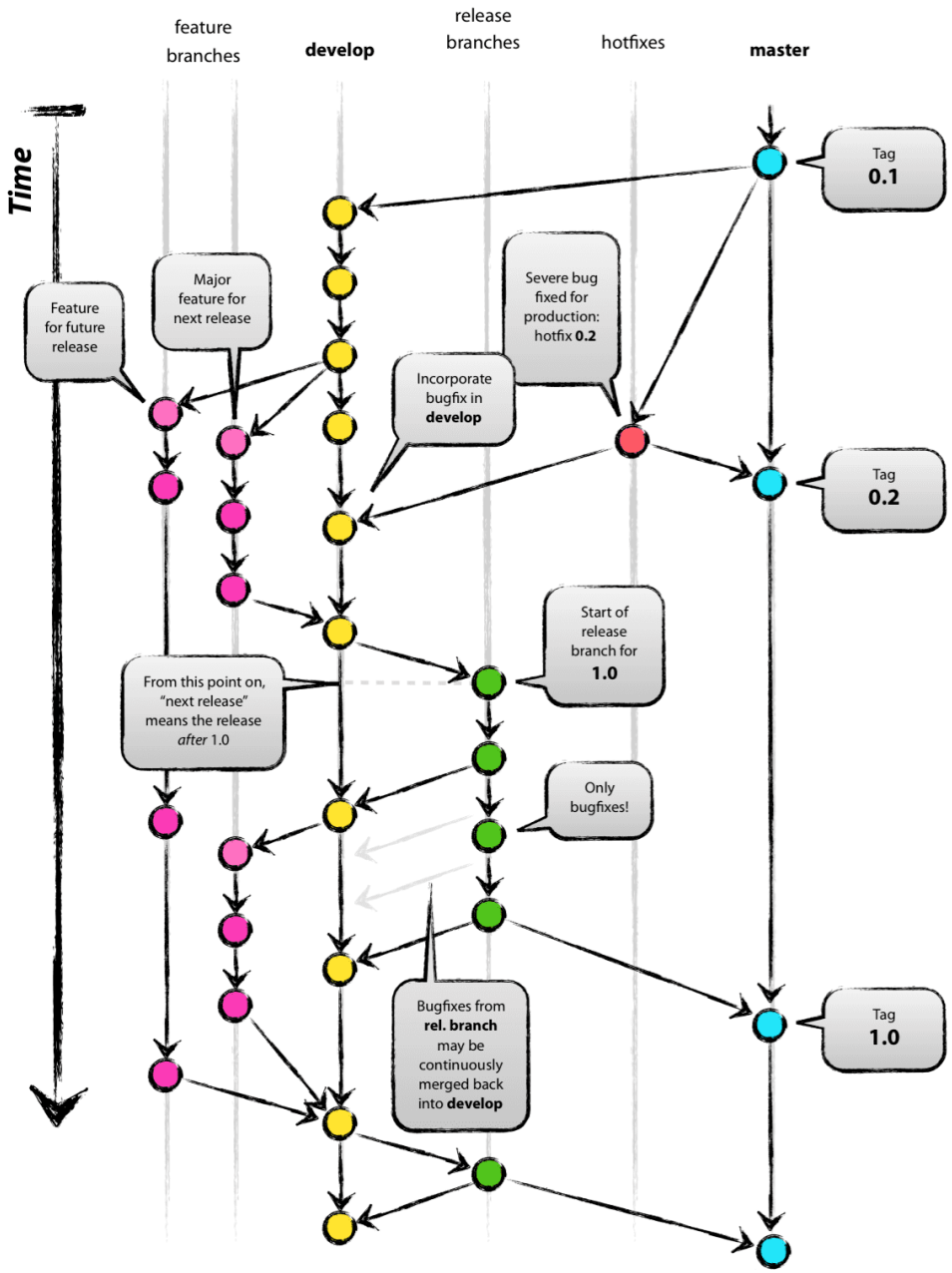
A metodologia adotada baseou-se no modelo de ramificação, o qual possibilitou uma organização ideal para a entrega contínua. Esta estrutura fundamenta-se na segregação de diversas *features* desenvolvidas individualmente pelos membros da equipe. Ao término de cada funcionalidade, era realizado um *Pull Request* para a ra-

[1] https://trello.com/

mificação de desenvolvimento (*development*); após as etapas de validação e revisão de código (*code review*), procedia-se com o *merge*.

Alinhado à metodologia ágil de *sprints*, utilizaram-se também ramificações de *release* versionadas. Nestas, após a execução de testes que validavam as novas implementações, o código era integrado à ramificação principal (*master*), consolidando a versão estável do sistema. O fluxo operacional deste modelo de ramificação pode ser visualizado na Figura 3.

Figura 3 – Modelo de ramificação *Git* (*Git Flow*).



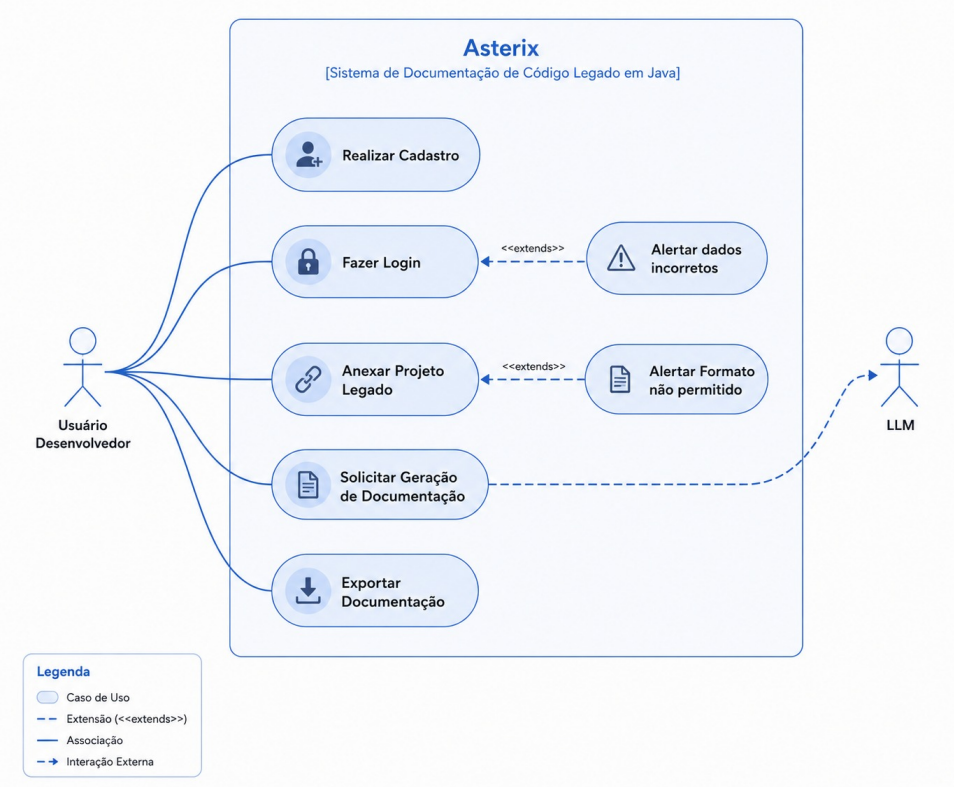
Fonte: Driessen (2010)

4.3 Estrutura

O sistema foi desenvolvido utilizando práticas ágeis (metodologia *Scrum*) e partindo de princípios como o código limpo (*Clean Code*), visando garantir um resultado funcional e de qualidade, atendendo ao objetivo proposto.

A Figura 4 descreve, visualmente, as interações entre o usuário e as funcionalidades presentes no sistema.

Figura 4 – Diagrama de Caso de Uso



Autoria própria

O usuário, representado pelo ator ”Usuário Desenvolvedor”, é quem faz uso do sistema em sua totalidade, desde que autenticado. Ele pode anexar o projeto legado, exclusivamente na linguagem Java e em formato compactado *.zip*, para enfim solicitar a geração da documentação pelo LLM, representado pelo ator homônimo. Após a disponibilização do artefato, o usuário desenvolvedor é capaz de baixá-lo em sua máquina, e terá acesso a este registro mesmo que saia do sistema.

4.4 Requisitos

A etapa de engenharia de requisitos foi fundamental para possibilitar o desenvolvimento do sistema, uma vez que foi possível estabelecer diretrizes, restrições e funcionalidades que a aplicação deve prover para alcançar o objetivo proposto. Neste traba-

lho, o levantamento de requisitos reflete a necessidade de uma orquestração precisa entre a ingestão de código, a extração manual e o processamento analítico através do uso de inteligência artificial.

Para garantir organização e clareza técnica, o escopo do MVP foi dividido em Requisitos Funcionais (RF), que definem ações e comportamentos diretos que o sistema deverá executar, e em Requisitos Não Funcionais (RNF), que definem atributos de qualidade, privacidade, usabilidade e restrições arquiteturais da aplicação.
A seguir, apresentam-se os requisitos mapeados para a construção do sistema.

Requisitos Funcionais

- **[RF01] Cadastro de Usuário**: O sistema deve permitir o registro de novos usuários mediante o fornecimento de nome, e-mail e criação de uma senha.
- **[RF02] Autenticação de Usuário**: O sistema deve permitir login de usuários (cadastrados) usando suas credenciais.
- **[RF03] Encerramento de Sessão**: O sistema deve fornecer a funcionalidade de logout para encerrar a sessão do usuário.
- **[RF04] *Upload* de Arquivo Legado**: O sistema deve permitir o *upload* de um arquivo compactado contendo o código-fonte de um sistema legado desenvolvido em Java.
- **[RF05] Extração de Pacote**: O sistema deve realizar a descompactação automática do arquivo enviado pelo usuário.
- **[RF06] Mapeamento de Arquivos**: Após descompactar, o sistema deve varrer o diretório e listar os arquivos com extensão *.java* encontrados no pacote para processamento.
- **[RF07] Extração Estrutural**: O sistema deve processar o código e extrair as principais características de cada arquivo analisado: nome da classe, pacote, *imports*, métodos (incluindo privados), parâmetros, *Javadocs*, hierarquia, tipos de retorno e corpo dos métodos.
- **[RF08] Conversão de Dados**: O sistema deve converter as informações estruturais extraídas na etapa anterior para um formato de dados hierárquico em JSON.
- **[RF09] Geração de Documentação**: O sistema deve orquestrar a geração automática da documentação completa do código analisado, utilizando da representação JSON como insumo para o modelo de IA.

- **[RF10] Formatação de Saída**: O sistema deve renderizar e disponibilizar a documentação gerada no formato HTML para visualização em tela.
- **[RF11] Exportação de Artefato**: O sistema deve permitir que o usuário realize o *download* da documentação gerada em formato HTML.

Requisitos Não Funcionais

- **[RNF01] Segurança**: Os arquivos enviados pelo usuário devem ser temporários, sendo obrigatoriamente removidos do servidor após a conclusão do processamento e geração da documentação, não havendo quaisquer armazenamento persistente do código original.
- **[RNF02] Usabilidade**: A interface de usuário (UI) deve ser intuitiva e autoguiada, garantindo que o usuário consiga realizar o *upload* e obter a documentação sem a necessidade de treinamentos prévios.
- **[RNF03] Acessibilidade e Plataforma**: A aplicação deve estar totalmente disponível via navegador *web* padrão, eliminando a necessidade de instalação de *softwares* ou dependências locais na máquina do usuário.
- **[RNF04] Portabilidade**: A arquitetura do *back-end* deve ser multi-plataforma, exigindo apenas que o ambiente de execução (servidor) seja compátivel com a JVM.
- **[RNF05] Resiliência e Tratamento de Erros**: O sistema deve ser tolerante a falhas de rede ou instabilidades na comunicação com a API da IA, capturando exceções e exibindo mensagens amigáveis e claras ao usuário, evitando o travamento silencioso da aplicação.
- **[RNF06] Desempenho e Estabilidade**: A arquitetura de orquestração deve permitir o processamento de múltiplos arquivos de forma controlada e sequencial, garantindo que o consumo de memória e requisições não comprometa a estabilidade do sistema ou exceda os limites estruturais da aplicação.

4.5 Arquitetura

O sistema foi desenvolvido baseando-se em princípios de *Domain-Driven Design* (DDD) e de código limpo (*Clean Code*), garantindo a separação de responsabilidades do código entre diferentes camadas, sendo estas:

- *Model*: Armazena os diferentes conceitos de negócio da aplicação, permitindo separar e manter as regras necessárias ao contexto do sistema, encapsulando o

estado, enumerações e regras de negócio de forma independente de *frameworks* externos.

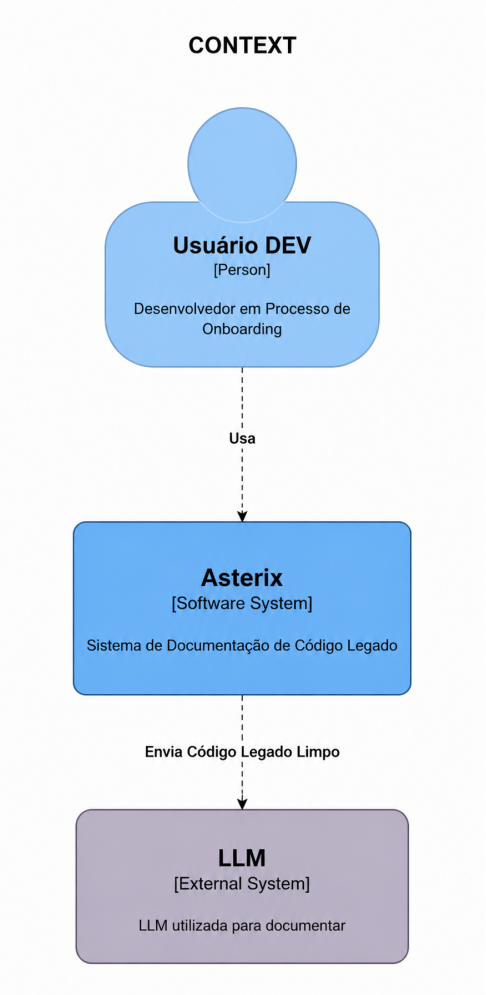
- *Persistence*: Isola a infraestrutura de armazenamento e persistência do sistema, contendo as entidades mapeadas para o banco de dados e as interfaces do repositório, garantindo o desacoplamento de tecnologias externas.
- *Repository*: Encapsula a lógica de acesso ao banco de dados, garantindo uma camada exclusiva para lidar com acessos para leitura e escrita de maneira isolada e independente de outros fatores externos.
- *Controller*: Responsável por centralizar as requisições *HTTP* e atuar como principal interface de comunicação externa da API, recebendo as chamadas iniciais e delegando a execução para o devido serviço.
- *Service*: Orquestra as operações e lógicas de negócio, ditando o próprio fluxo das regras de negócio do sistema. É responsável por interagir com outros serviços entre si para possibilitar o fluxo de execuções e transferências de dados, e pela comunicação com classes de domínio e infraestrutura.

4.5.1 Modelo C4

O Modelo C4, elaborado por Brown (2013), propõe um modo de centralizar representações e abstrações comuns de arquiteturas de *software* sob uma única notação intuitiva e coerente. Esta abordagem possibilita a divisão em quatro níveis hierárquicos - *Context*, *Container*, *Component* e *Code* - cada um focando em tópicos específicos de detalhe técnico referentes à arquitetura utilizada.

O nível mais alto de abstração, o de *Context* (Contexto) permite a visualização dos principais elementos de um sistema e as suas interações com outros sistemas em alto nível, como apresentado na Figura 5.

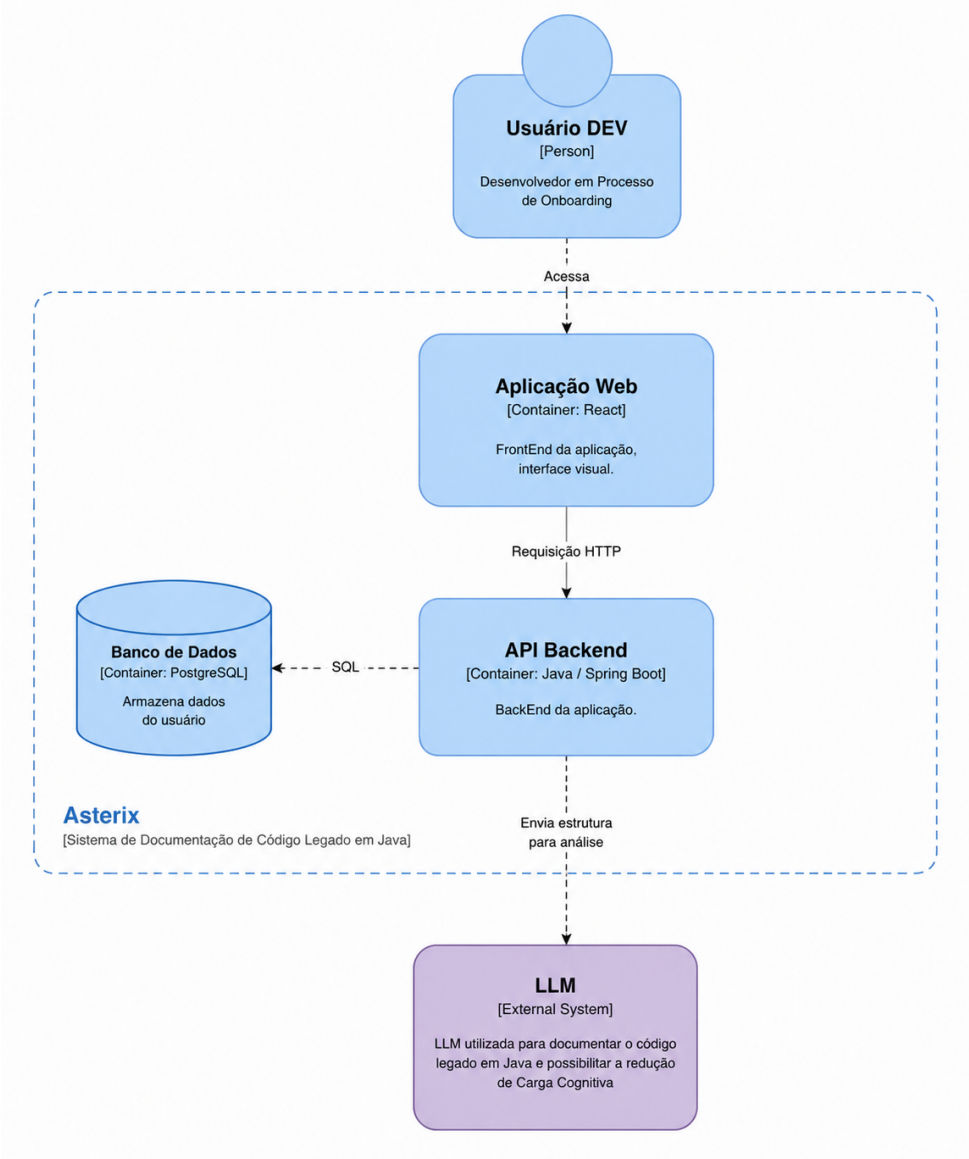
Figura 5 – Diagrama de Contexto



Autoria própria

Já no segundo nível - o de *Containers*, é onde detalhamos os sistemas e subsistemas em termos de contêineres, bem como as interações realizadas entre eles. Na Figura 6, é possível visualizar a representação desta no sistema.

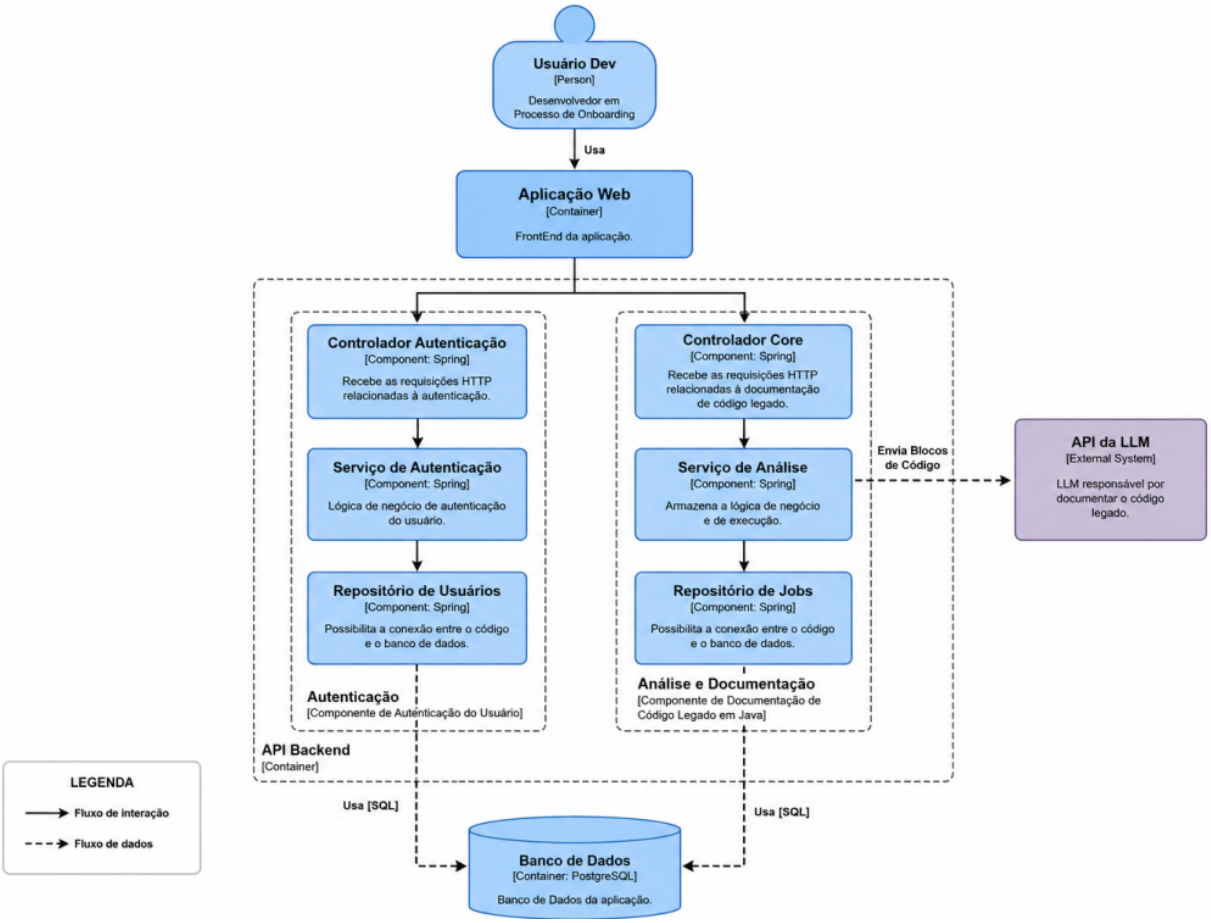
Figura 6 – Diagrama de Contâineres



Autoria própria

O terceiro nível do Modelo C4, o de Componentes, é capaz de representar detalhes internos de um contâiner, entendendo como diferentes módulos ou serviços se comunicam e colaboram para executar suas funções. Neste diagrama, há de se notar a existência do contâiner ”API *Backend*”, cujo papel é indispensável e essencial para a comunicação e funcionamento de todas as funcionalidades do sistema, estando representado na Figura 7.

Figura 7 – Diagrama de Componentes

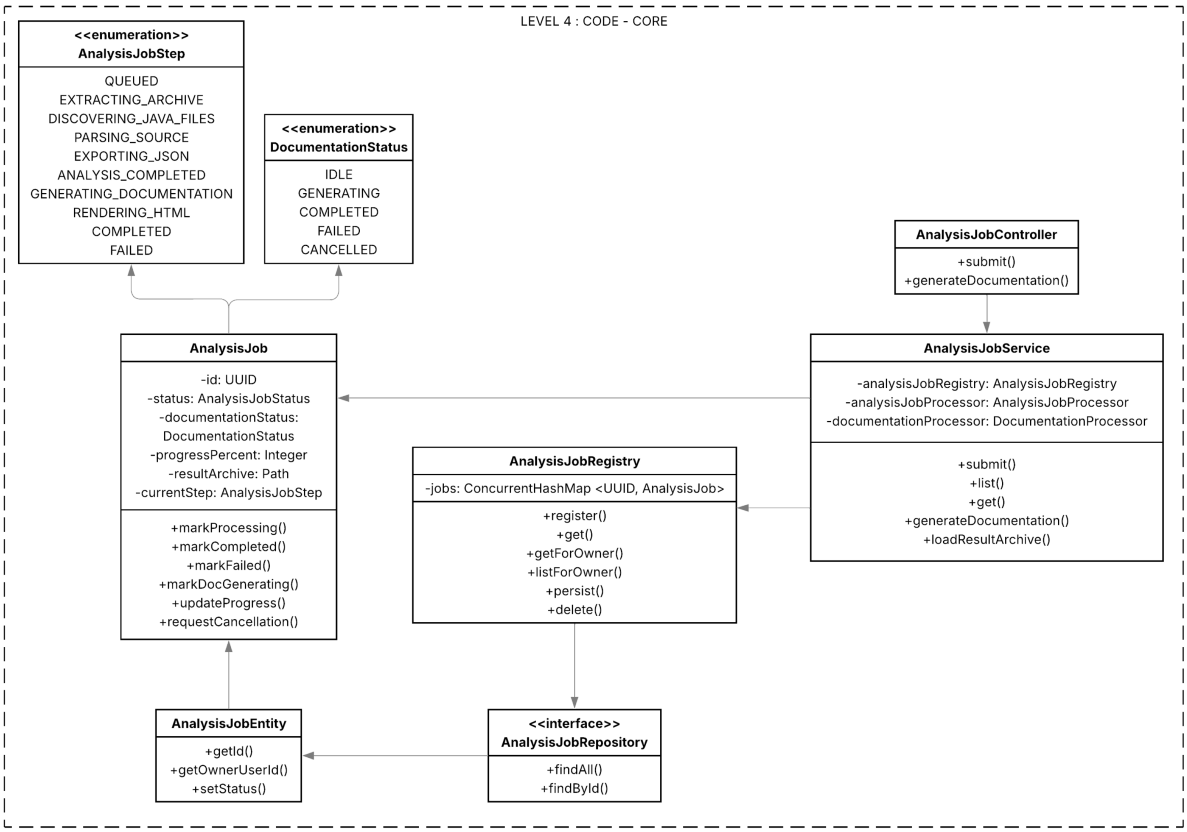


Autoria própria

Por fim, o último nível do Modelo C4 é responsável por apresentar o código do sistema o qual se está desenvolvendo, como se fosse uma generalização do que foi representado nos níveis anteriores por meio da representação visual de como as classes do código são definidas e se relacionam entre si. Para facilitar a visualização e compreensão do fluxo como um todo, foi separada a representação das classes em diagramas menores, mas que representam todo o processo de envio, análise e geração de documentação pelo *LLM*.

Na Figura 8, é representado o fluxo *core* do sistema, onde todos os processos, desde o mapeamento à documentação do código legado enviado, são orquestrados.

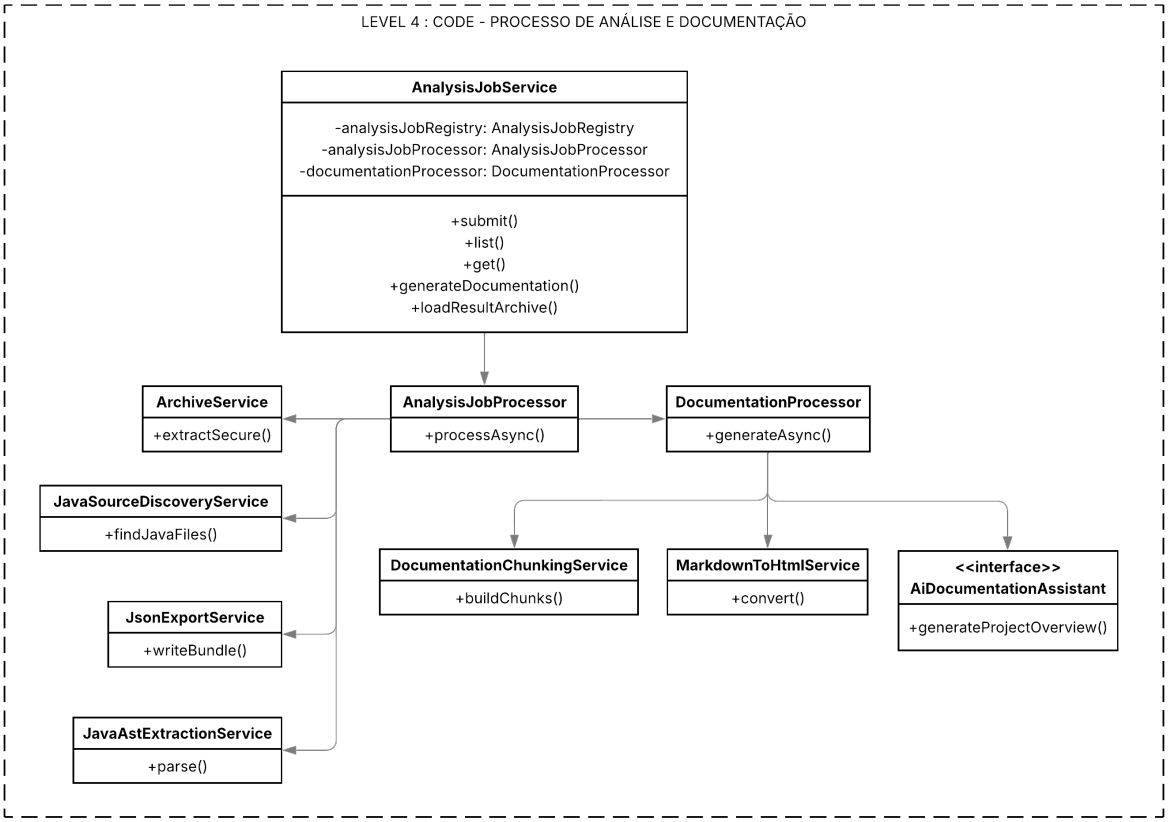
Figura 8 – Diagrama de Código - Core



Autoria própria

O fluxo de mapeamento, extração, conversão, análise e documentação de código legado é orquestrado através da classe `AnalysisJobService`, conforme explicitado na Figura 9.

Figura 9 – Diagrama de Código - Análise e Documentação de Código Legado



Autoria própria

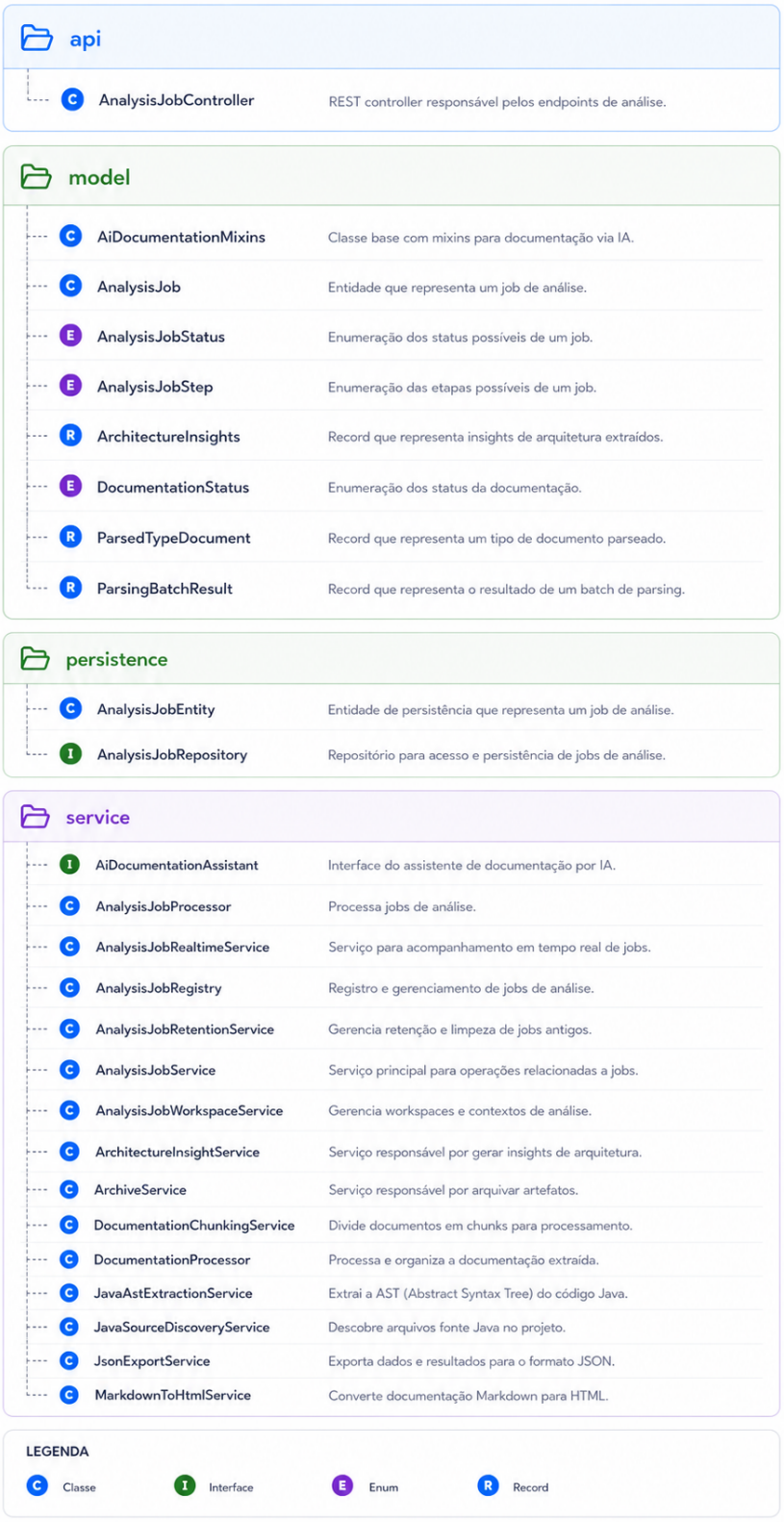
Através dos diagramas apresentados, é possível obter uma visão clara da comunicação entre as classes do sistema, evidenciando a aplicação do primeiro princípio *SOLID* - o Princípio de Responsabilidade Única (SRP), bem como a utilização de princípios de DDD, enfatizando a clara separação entre as regras de negócio, de infraestrutura e de persistência de dados.

4.6 Backend

O *back-end* do sistema foi desenvolvido utilizando Spring Boot, um *framework* Java que facilita a criação, configuração e execução de aplicações baseadas no chamado Spring Framework, com foco em produtividade e simplicidade. Segundo estudo realizado por Choma, Chwaleba e Dzieńkowski (2023), quando comparado a outros *frameworks* como o Django (*Python*) e Express (Node.JS), o Spring Boot destaca-se por prover mais mecanismos que melhoram a performance de aplicações, além de demonstrar agilidade de resposta para requisições *HTTP*.

A Figura 10 apresenta a organização dos arquivos, separada devidamente como representantes da camada às quais estão inseridas.

Figura 10 – Organização dos Arquivos do *Backend*



Autoria própria

4.6.1 Variáveis de Ambiente

A configuração de credenciais e variáveis de ambiente foi projetada para possibilitar que o comportamento da aplicação seja ajustado sem a necessidade de alteração direta no código-fonte ou recompilação do artefato, além de fornecer maior segurança relacionada à dados sensíveis. Para isso, foi utilizado o suporte nativo do *framework* Spring Boot nos arquivos *.properties*, integrados à variáveis de ambiente do sistema operacional e de fácil utilização.

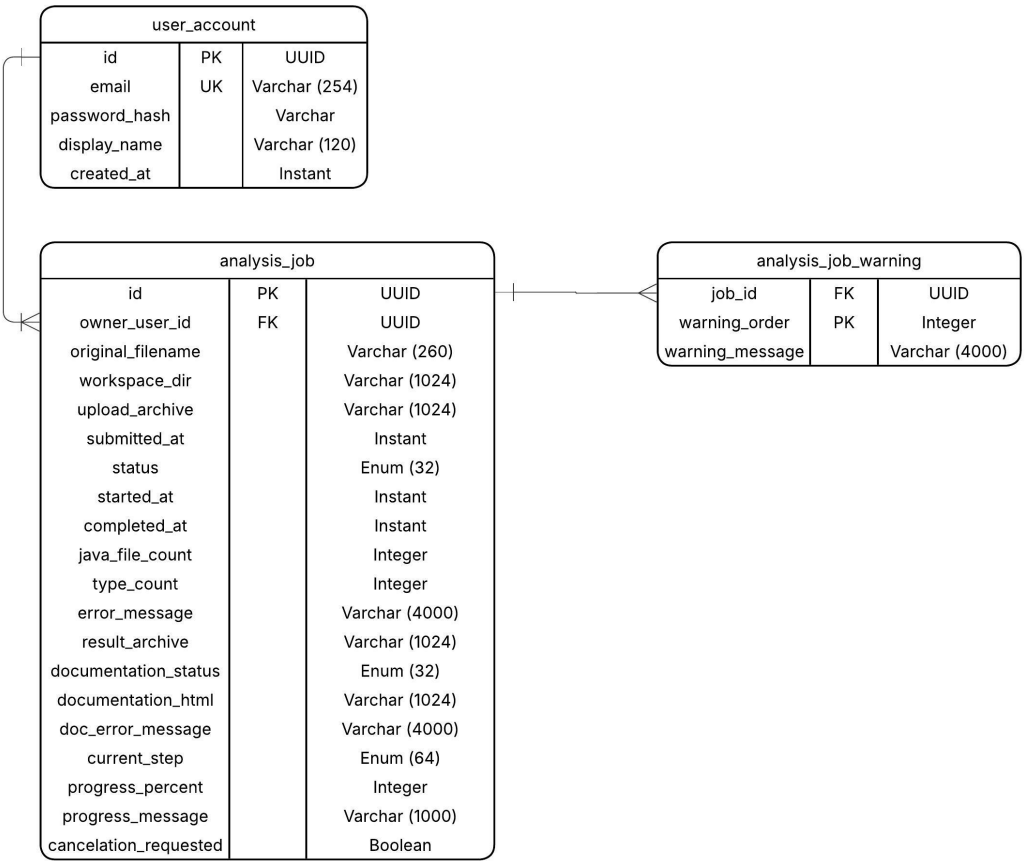
As variáveis definidas foram divididas nos seguintes grupos:

- Configurações de Infraestrutura: Definem o nome da aplicação e os limites de *upload* dos arquivos legados, definidos para 100MB para comportar projetos de médio porte.
- Configurações de Banco de Dados: Parametrizam a URL de conexão, usuário e senha do PostgreSQL, permitindo configurar os dados necessários para conexão de maneira segura.
- Segurança e Autenticação: Controlam o comportamento do protocolo JWT (*JSON Web Token*), essencial para a autenticação do usuário e segurança de suas informações.
- Gestão de Artefatos: Definem políticas de retenção para os arquivos gerados, possibilitando a configuração da quantidade de dias que a documentação resultante deve ser armazenada no servidor antes da limpeza automática.
- Orquestração da Inteligência Artificial: É o ponto crítico de flexibilidade do sistema - permite configurar e alternar entre provedores de LLMs através da definição do modelo específico e de parâmetros de geração, como limite de *tokens* e a URL base da API.

4.6.2 Banco de Dados

Foi utilizado o *SGBD PostgreSQL*, tendo em vista sua natureza robusta e de fácil integração com o ecossistema *Spring* para possibilitar o armazenamento, persistência e manuseio de dados nos processos de autenticação e análise de código legado. A Figura 11 apresenta o Diagrama Entidade-Relacionamento (DER) da base de dados configurada no sistema.

Figura 11 – Diagrama Entidade-Relacionamento da Base de Dados



Autoria própria

A tabela `user account` é responsável por armazenar as informações da conta do usuário, como o seu nome, o e-mail cadastrado - definido como chave única (*UK*) - e a senha armazenada como *hash*, garantindo conformidade no tratamento de dados de acordo com a LGPD (Lei Geral de Proteção de Dados).

A tabela `analysis job` armazena todas as informações do processo de análise do código legado enviado pelo usuário, e sofre alterações ao longo de todo o processamento do código até a sua documentação, servindo posteriormente para recuperar a documentação e estrutura geradas. Esta tabela tem o auxílio da tabela `analysis job warning`, que tem dados inseridos apenas em ocasiões de falhas ou erros ocorridos no processo de análise, e auxilia na auditoria e tratamento de erros do processo de documentação.

A utilização do Spring Data JPA, *framework* distribuído através do Spring Data, se tornou essencial para possibilitar a conexão e comunicação entre o código e o banco de dados, facilitando consultas e inserções de dados de maneira ágil e segura.

4.6.3 Dependências

Visando facilitar o desenvolvimento e proporcionar uma solução final qualitativa e alinhada com o que foi proposto no trabalho, o projeto fez uso de bibliotecas e *frameworks* consolidados no ecossistema Java. A seleção destas dependências foi pautada na necessidade de garantir robustez na manipulação de dados estruturais, eficiência na orquestração de requisições de IA e facilidade na construção de uma API *RESTful*.

Desta maneira, as principais ferramentas incorporadas à arquitetura do sistema, bem como suas respectivas finalidades no escopo da aplicação, são descritas a seguir:

- Spring Boot: *Framework* utilizado como base para a construção do *back-end*, facilitando a configuração da aplicação, injeção de dependência e exposição dos *endpoints* da API.
- LangChain4j: Biblioteca utilizada para orquestração e integração com os *LLMs*, sendo o principal responsável por gerenciar a comunicação com a IA na etapa de geração de documentação.
- JavaParser: Biblioteca que permite analisar, transformar e gerar código-fonte Java através de outros programas em Java, possibilitando gerar estruturas como Árvores Sintáticas Abstratas (AST).
- JWT: Biblioteca que permite fácil gerenciamento de *tokens* para autenticação do usuário.

Todas as dependências utilizadas no sistema são declaradas através do arquivo *pom.xml*, essencial em projetos que utilizam Maven, definindo informações mister sobre o projeto, como pacotes, nome do projeto e suas dependências.

4.6.4 Mapeamento de Arquivos

Etapa essencial para possibilitar a conversão limpa para o formato JSON, tem como objetivo isolar estritamente o código-fonte de interesse estrutural - é responsável por analisar o ecossistema Java do arquivo compactado enviado pelo usuário. Este estágio é iniciado através do serviço de descompactação de disco `ArchiveService`, responsável por extrair do arquivo compactado enviado todos os arquivos presentes e armazena-los em uma constante do tipo *Path*, nativo do Java e ideal para tipificar arquivos. Em seguida, é realizada a busca por arquivos relevantes ao ecossistema Java através de sua extensão, orquestrado através da classe `JavaSourceDiscoveryService`, resultando em uma lista de arquivos sintáticamente importantes ao código.

Através do uso do método `Files.walk()`, proveniente da API NIO.2 (API de manipulação de arquivos nativo do Java), é possível percorrer recursivamente toda a hierarquia de pastas extraídas de forma performática. Em sistemas legados em Java, é comum a ocorrência de árvores de pacotes aninhadas, e a utilização de *streams* da API NIO.2 garante que a leitura dos subdiretórios não estoure a memória do sistema através de um *Out Of Memory Error*, conforme apresentado na Figura 12.

Figura 12 – Metódo findJavaFiles

```
public List<Path> findJavaFiles(Path rootDirectory) throws IOException {
    try (var stream = Files.walk(rootDirectory)) {
        return stream
            .filter(Files::isRegularFile)
            .filter(path -> path.toString().endsWith(".java"))
            .filter(path -> !containsIgnoredDirectory(rootDirectory, relativize(path)))
            .sorted()
            .toList();
    }
}
```

Autoria própria

Durante o mapeamento, o serviço avalia cada nó da arvore de diretórios, garantindo que seja um arquivo comum e que possua a extensão alvo (arquivos terminados em *.java*). Arquivos irrelevantes para a análise arquitetural, como arquivos binários compilados, são ignorados de antemão, garantindo como resultado uma lista em memória contendo exclusivamente os caminhos absolutos das classes Java a serem analisadas.

Do ponto de vista da arquitetura de *Software*, o isolamento desta rotina em um serviço próprio evidencia a utilização do Princípio de Responsabilidade Única (SRP), de extrema importância para se obter uma solução final de qualidade, escalável e de fácil manutenção. Ademais, esta etapa atua como uma camada de defesa, lançando exceções em casos de listas vazias ou de retornos inesperados, interrompendo a *pipeline* sem impactar diretamente na execução do sistema.

4.6.5 Limpeza de Código e Conversão em AST

Para possibilitar a transformação de um código polido e limpo, após o mapeamento dos arquivos necessários, é realizado o processo de limpeza do código-fonte, onde pontos obsoletos como espaços em branco e identação irregular são removidos. Este processo se torna dinâmico devido a utilização da biblioteca `JavaParser`, tratando tudo aquilo que fora extraído do arquivo compactado como um código de fato, e não somente como um elemento textual qualquer. Através desse processo, elementos não

estruturais são descartados, preservando estritamente os blocos de *Javadoc* que agregam valor descritivo ao código, permitindo que regras definidas através destas não se percam em meio às informações.

Após a etapa inicial de abstração de elementos sem importância semântica, a classe JavaAstExtractionService utiliza da `Streams API` para manipular os dados de maneira funcional e varrer a Unidade de Compilação do código, extraindo apenas propriedades topológicas essenciais - declarações de classes, anotações, assinaturas de métodos, parâmetros, tipos de retorno e interfaces implementadas, como é possível observar na Figura 13. Esses dados são então mapeados de maneira hierárquica para um objeto de domínio que atua como estrutura intermediária livre de ruído sintático - a AST, pronta para ser convertida em uma estrutura de fácil compreensão e análise pelo LLM.

Figura 13 – Metódo de Extração de Tipos

```
private ParsedTypeDocument extractType(
    String sourcePath,
    String packageName,
    List<String> imports,
    TypeDeclaration<?> type
) {
    List<BodyDeclaration<?>> members = getMembers(type);

    List<ParsedTypeDocument.FieldInfo> fields = members.stream()
        .filter(FieldDeclaration.class::isInstance)
        .map(FieldDeclaration.class::cast)
        .flatMap( fieldDeclaration -> fieldDeclaration.getVariables())
        .stream()
        .map( variable -> new ParsedTypeDocument.FieldInfo(
            variable.getNameAsString(),
            variable.getType().asString(),
            toModifierNames(fieldDeclaration),
            toAnnotationNames(fieldDeclaration),
            variable.getInitializer().map(Object::toString).orElse(null),
            toJavadocText(fieldDeclaration),
            toLineSpan(fieldDeclaration)
        ))
    .toList();

    return new ParsedTypeDocument(sourcePath, packageName, imports, type, fields);
}
```

Autoria própria

Com a árvore sintática em mãos, é possível realizar a conversão para o formato JSON, de melhor entendimento por parte dos grande modelos de linguagem. Esta conversão fica a cargo do serviço `JsonExportService`, onde é organizado sintáticamente toda a estrutura da árvore em objetos com chaves identificadoras e valores, garantindo coerência e explicabilidade, bem como a inferência de categorias de diretórios baseados em camadas como *controllers* e *services*. A Figura 14 apresenta o mapeamento das categorias no serviço em questão.

Figura 14 – Mapeamento de Categorias de Diretórios

```
private static final Set<String> CATEGORY_NAMES = Set.of(
    "dto",
    "controller",
    "service",
    "repository",
    "config",
    "model",
    "models",
    "entity",
    "entities",
    "exception",
    "exceptions",
    "mapper",
    "util",
    "utils",
    "security",
    "client",
    "clients",
    "adapter",
    "adapters"
);
```

Autoria própria

4.6.6 Insights Arquiteturais

O serviço `ArchitectureInsightService` tem como objetivo analisar estaticamente o código legado mapeado e convertido, analisando a saúde estrutural do sistema legado por meio de análise estática e sintática. São mapeados *imports* externos à aplicação e os da própria aplicação, bem como a hierarquia da herança entre estes e as diferentes classes existentes.

Através da detecção de ciclos, por meio da aplicação do algoritmo *Tarjan* no próprio código, conforme apresentado na Figura 15, é possível mapear aninhamentos e acoplamentos existentes no código legado, possibilitando identificar empecilhos na possível modularização do sistema.

Figura 15 – Implementação do Metódo Tarjan

```
private List<Set<String>> tarjan(Map<String, Set<String>> graph) {
    Map<String, Integer> indexByNode = new HashMap<>();
    Map<String, Integer> lowLinkByNode = new HashMap<>();
    Deque<String> stack = new ArrayDeque<>();
    Set<String> onStack = new HashSet<>();
    List<Set<String>> components = new ArrayList<>();
    int[] index = {0};

    for (String node : graph.keySet()) {
        if (!indexByNode.containsKey(node)) {
            strongConnect(
                node,
                graph,
                indexByNode,
                lowLinkByNode,
                stack,
                onStack,
                components,
                index
            );
        }
    }
    return components;
}
```

Autoria própria

Em seguida, é realizada uma avaliação sobre tópicos como o tamanho da classe, a quantidade de membros existentes e o nível de acoplamento da classe, atribuindo pontuações incrementais baseadas, por exemplo, no numero de linhas de uma classe, as quais a depender podem ser classificadas como *Hotspots* - pontos críticos que necessitam de atenção imediata.

Por fim, todo o conjunto obtido através dos *insights* arquiteturais é utilizado para possibilitar a criação de sugestões e dicas textuais práticas.

4.6.7 Processamento por LLM

O processamento realizado através do LLM é a etapa cujo valor agregado melhor reflete à proposta estabelecida do presente trabalho, e garante que o código-fonte enviado pelo usuário tenha sua tratativa otimizada e assíncrona. O serviço DocumentationProcessor é responsável por estabelecer um fluxo de processamento iterativo baseado em fragmentos lógicos do código (as *chunks*), através da classe DocumentationChunkingService, separando blocos do código-fonte por categorias lógicas (como *controllers*, *services*) atendendo ao máximo de caracteres disponíveis para o modelo em uso.

Após a separação, é realizada uma iteração sobre as *chunks* obtidas, integrando políticas de tolerância a falhas para garantir que as limitações impostas pelas APIs dos modelos não interrompam a geração da documentação. Dessa maneira, com base em um *prompt* do sistema bem definido visando proporcionar uma descrição clara, objetiva e intuitiva do código a ser analisado, todo aquele fragmento lógico é enviado para ser analisado pelo LLM, cujo resultado é concatenado com os anteriores a cada ciclo.

O fluxo finaliza com a consolidação dos artefatos em *Markdown* e a síntese de uma visão arquitetural global, que são posteriormente renderizados em um formato HTML de fácil acesso para o usuário final.

4.7 Frontend

A camada de *front-end* foi desenvolvida sob o paradigma de componentização utilizando a biblioteca React, operando sobre o ferramental de construção Vite para garantir ciclos de renderização eficientes e alta escalabilidade. Para assegurar a consistência dos dados, implementou-se uma camada de validação com *Zod* e *React Hook Form*, mitigando erros de entrada e garantindo que os dados trafegados sigam os esquemas *(schemas)* estabelecidos. A gestão do estado assíncrono e o controle de *cache* das análises de código foram centralizados no *TanStack Query*, o que permite uma experiência de uso fluida ao reduzir requisições redundantes e otimizar o tempo de resposta do sistema. A gestão de todas as bibliotecas e pacotes do ecossistema foi realizada via NPM *(Node Package Manager)*.

4.7.1 Estilização e *Design System* (*Tailwind CSS*)

No que se refere à apresentação visual e a UX *(User experience)*, a interface foi desenvolvida utilizando a metodologia *utility-first* por meio do *framework Tailwind CSS*. A escolha desta tecnologia baseou-se na necessidade da criação de um *Design System* coeso, onde escalas de espaçamento, paletas de cores e tipografia foram seguidas, garantindo uma consistência estética em toda a aplicação. Diferente das abordagens convencionais de CSS (*Cascading Style Sheets*), onde o crescimento do projeto costuma resultar em arquivos de estilo volumosos e de difícil manutenção, o *Tailwind* CSS opera através de um motor de compilação *Just-in-Time* (JIT) similar ao *JavaScript*. Além disso, a implementação de uma interface responsiva foi simplificada através de modificadores nativos do *framework*. Essa flexibilidade foi essencial para assegurar a usabilidade do sistema em diferentes dispositivos.

4.7.2 Elementos Interativos e Imersão (*Spline*)

Visando romper com a estética puramente textual das ferramentas de desenvolvimento, a aplicação insere elementos tridimensionais interativos desenvolvidos na plataforma *Spline*. A utilização dessas ilustrações 3D permite uma representação visual mais dinâmica e maior interatividade com o usuário final. A integração técnica ocorre via componente nativo para *React*, permitindo que os elementos gráficos respondam a estados da aplicação em tempo real. Conforme ilustrado na Figura 16, o menu principal utiliza esse recursos para fornecer um feedback visual imediato. Essa abordagem de

UI Design foca na iteratividade, transformando a análise técnica em uma experiência visualmente intuitiva e moderna.

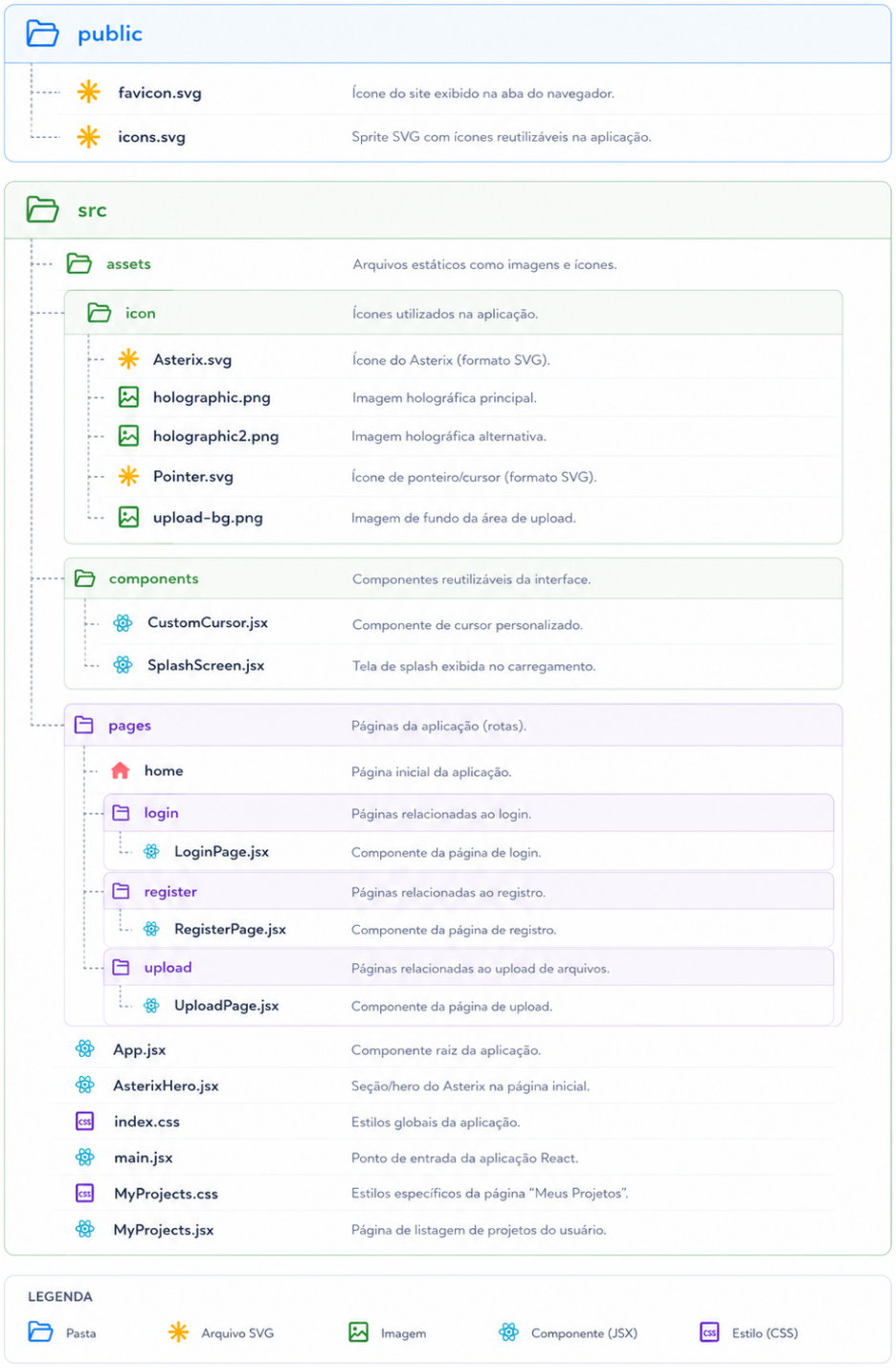
Figura 16 – Tela com elemento 3D do *spline* aplicado

Autoria própria

4.7.3 Organização de código e padrões

A organização estrutural do projeto foi seguindo os padrões de escalabilidade e manutenibilidade do ecossistema *React*. Como pode ser observado na Figura 17, adotou-se uma arquitetura de diretórios que isola responsabilidades: a pasta *components* armazena elementos de interface reutilizáveis, enquanto o diretório *pages* organiza a lógica de roteamento e visualização por domínios específicos (Ex: *Login*, *Upload*, *Home*). Essa segregação permite um desenvolvimento mais organizado, uma vez que a lógica de cada página está encapsulada em seu respectivo módulo.

Figura 17 – Estrutura de diretórios do FrontEnd



Autoria própria

4.7.4 Responsividade e Acessibilidade

Além da estética, a interface foi projetada sob os pilares da responsividade e acessibilidade digital. Utilizando-se *Flexbox* integrado com o *Tailwind css*, implementou-se um layout fluido que se adapta automaticamente a diferentes proporções de tela. No

que diz respeito à acessibilidade, a escolha das cores focaram no alto contraste, visando facilitar a leitura do conteúdo e a redução da fadiga no momento de leitura. Além disso, a estruturação dos componentes seguiu padrões semânticos de HTML5, assegurando que a navegação seja compreensível para tecnologias assistivas. Essas escolhas foram feitas com o intuito de oferecer uma ferramenta inclusiva, alinhada com as melhores práticas de experiência do usuário (UX).

4.7.5 Validação da interface de experiência do usuário (*UX*)

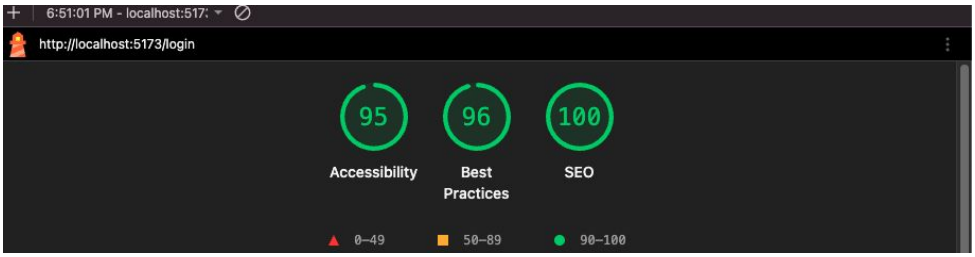
4.7.5.1 Metodologia e avaliação

Com o objetivo de validar tecnicamente a qualidade da interface desenvolvida e validar as premissas de responsividade, acessibilidade e conformidade com padrões modernos de desenvolvimento, a aplicação foi submetida a testes automatizados. Utilizou-se a ferramenta *Google Lighthouse*[2], integrada nativamente ao motor do navegador Google Chrome. A ferramenta avalia páginas web em categorias, atribuindo pontuações de 0 a 100 baseada em métricas auditáveis. Os testes foram feitos em ambiente de desenvolvimento local (*localhost*), cobrindo as principais páginas da aplicação como Login, Cadastro (*Register*) e página inicial (*Home*).

4.7.5.2 Análise dos resultados de acessibilidade e padrões

Os relatórios gerados pelo *Lighthouse* para as três páginas avaliadas demonstram índices elevados. Na página de *Login*(Figura 18), observou-se uma pontuação de 95 em acessibilidade e 96 em melhores práticas, com 100 em SEO (*Search Engine Optimization*).

Figura 18 – Resultado da auditoria *Lighthouse* da página de *login*.

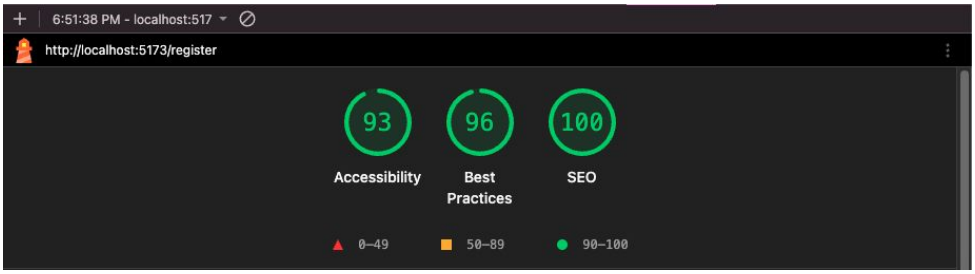


Autoria própria

Resultados semelhantes foram obtidos na página de cadastro (Figura 19), em que a aplicação atingiu 93 em acessibilidade, mantendo as pontuações elevadas de 96 melhores práticas e a pontuação máxima em SEO.

[2] https://developer.chrome.com/docs/lighthouse

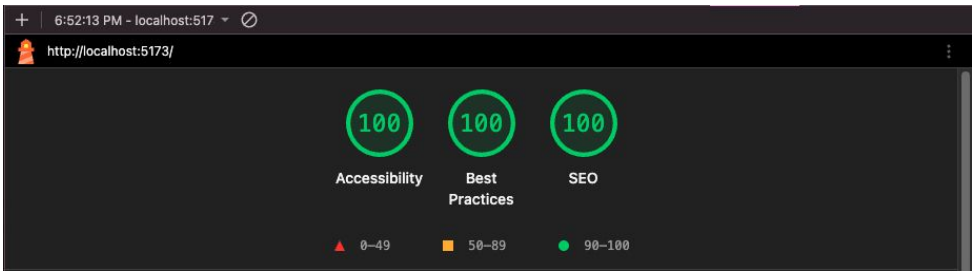
Figura 19 – Resultado da auditoria *Lighthouse* da página de registro de usuários.



Autoria própria

A validação finalizou na página inicial (*Home*) (Figura 20), no qual o sistema atingiu a pontuação máxima em todas as categorias avaliadas (Acessibilidade, Melhores práticas e SEO). Este resultado gabaritado nos mostra a eficácia da *stack* escolhida, provando a componentização com React e a estilização com Tailwind CSS foram implementadas seguindo o os padrões de semântica HTML e legibilidade.

Figura 20 – Resultado da auditoria *Lighthouse* da página inicial (*Home*).



Autoria própria

4.7.5.3 Validação técnica

Os dados apresentados nas Figuras 18, 19 e 20 validam as escolhas técnicas discutidas anteriormente. A elevada pontuação em acessibilidade comprova que a escolha das cores de alto contraste, e o uso de *tags* semânticas e a estrutura da navegação foram bem implementadas.

Em relação a responsividade, o *lighthouse* audita indiretamente este item através da categoria *Mobile-Friendly* na versão *Mobile* do teste. Como todos os outros scores em *desktop* não obtiveram diferença significativa, fora mantido o padrão com alta pontuação, indicando uma aplicação *front-end* robusta.

Adicionalmente, as pontuações em melhores práticas (96 e 100) comprovam que a aplicação está livre de vulnerabilidade de segurança conhecidas, utiliza HTTPS (*HyperText Transfer Protocol Secure*) e adota padrões modernos que facilitam a manutenção e escalabilidade do código, integrando-se perfeitamente aos principios *clean code*.

5 Demonstração da Aplicação

Este capítulo tem por objetivo apresentar, de maneira visual, o funcionamento do sistema integrado (frontend, backend e banco de dados), simulando a sua utilização por um usuário desenvolvedor em processo de *onboarding* em um projeto legado em Java, possibilitando uma visualização real das telas e dos mecanismos disponíveis para utilização.

5.1 Página Inicial e Login

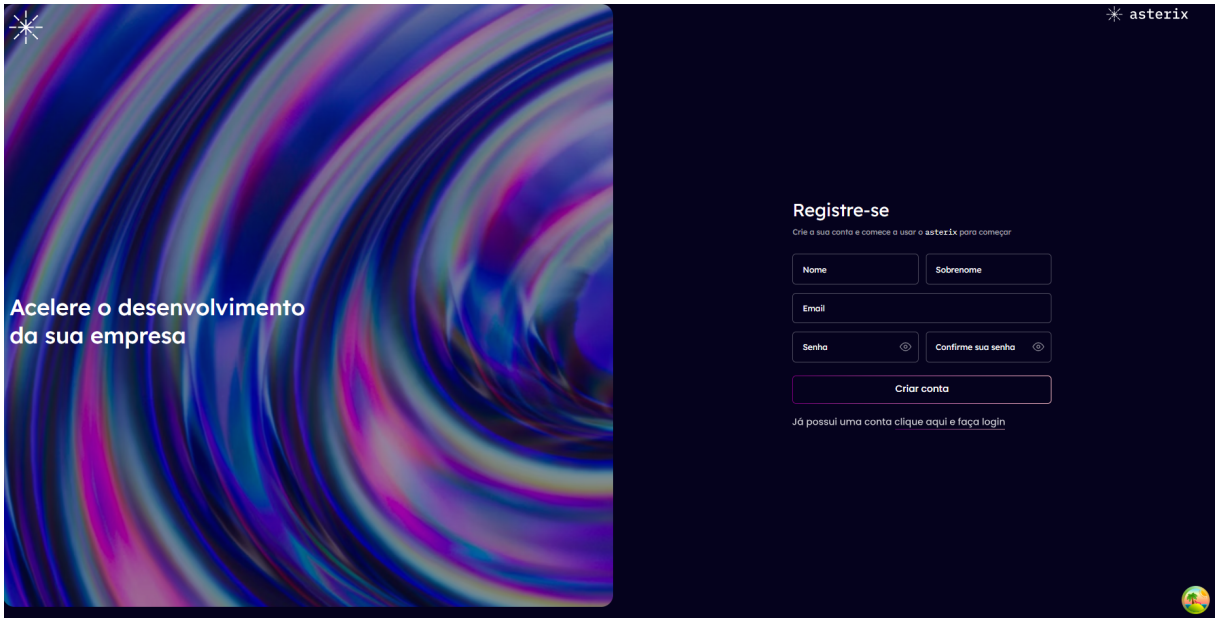
A página inicial do sistema possui um elemento visual convidativo e interativo - um robô reativo, possibilitado através da utilização da biblioteca *Spline*. Nesta tela, apresentada na Figura 21, o usuário pode optar por acessar a página de login para se autenticar, caso já haja cadastro no sistema, ou cadastrar-se, para criar um registro seguro no banco de dados e possibilitar o acesso ao sistema.

Figura 21 – Página Inicial do Sistema

Autoria própria

A página de cadastro, apresentada na Figura 22, permite ao usuário realizar o registro no sistema, possibilitando a criação de sua conta e o conseguinte acesso seguro ao sistema. Nela, é onde a rota `register` do módulo de autenticação é requisitado, possibilitando o registro dos dados fornecidos no formulário no banco de dados.

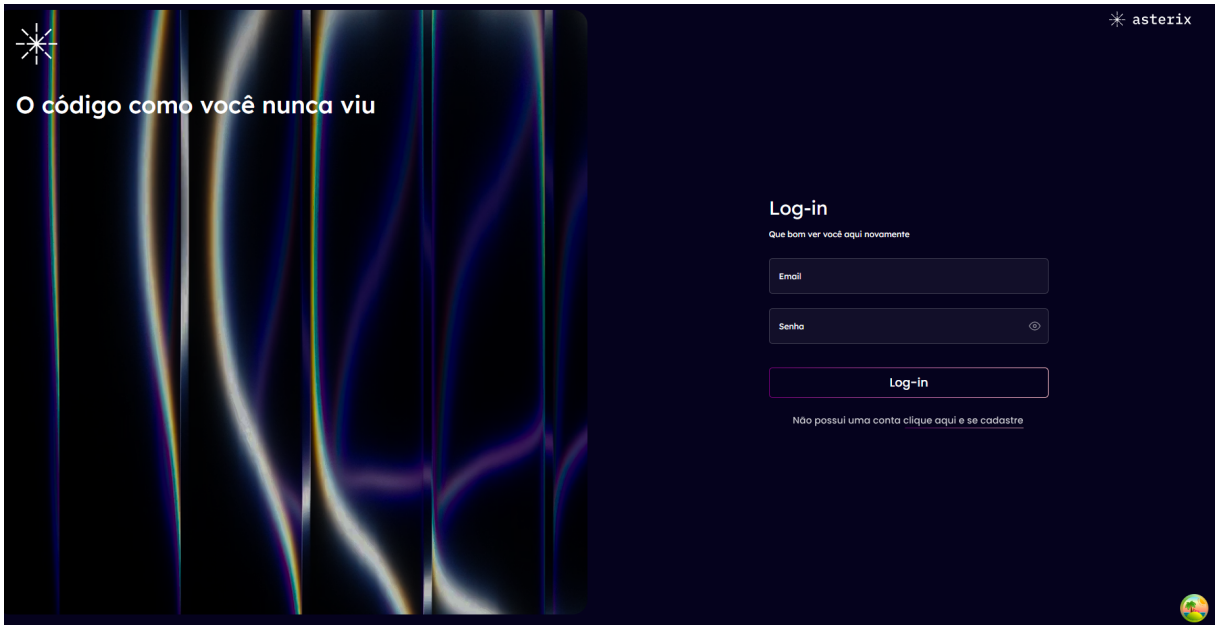
Figura 22 – Página de Cadastro do Usuário



Autoria própria

Caso o usuário já possua registro na plataforma, o mesmo pode acessar diretamente a página de *login*, fornecendo os dados enviados no momento de registro. É realizada uma requisição para a rota de `login`, e em caso de retorno com sucesso, o usuário é autenticado e redirecionado para a *homepage*. A Figura 23 apresenta a tela de login e seus campo para preenchimento.

Figura 23 – Página de *Login*

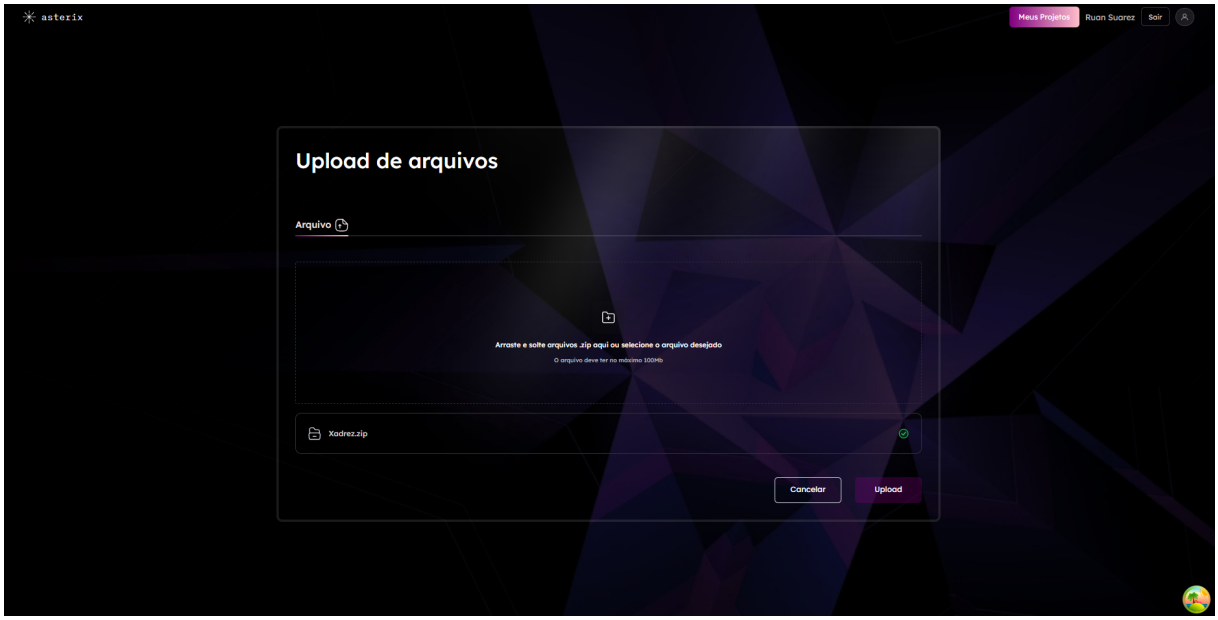


Autoria própria

5.2 *Home*

A *homepage* do sistema, apresentado na Figura 24 apresenta ao usuário um campo sinalizado para anexar o arquivo compactado do projeto legado na linguagem Java a ser documentado, bem como a aba ”Meus Projetos”, responsável por listar os documentos enviados pelo usuário e possibilitar o *download* do JSON e documentação gerados.

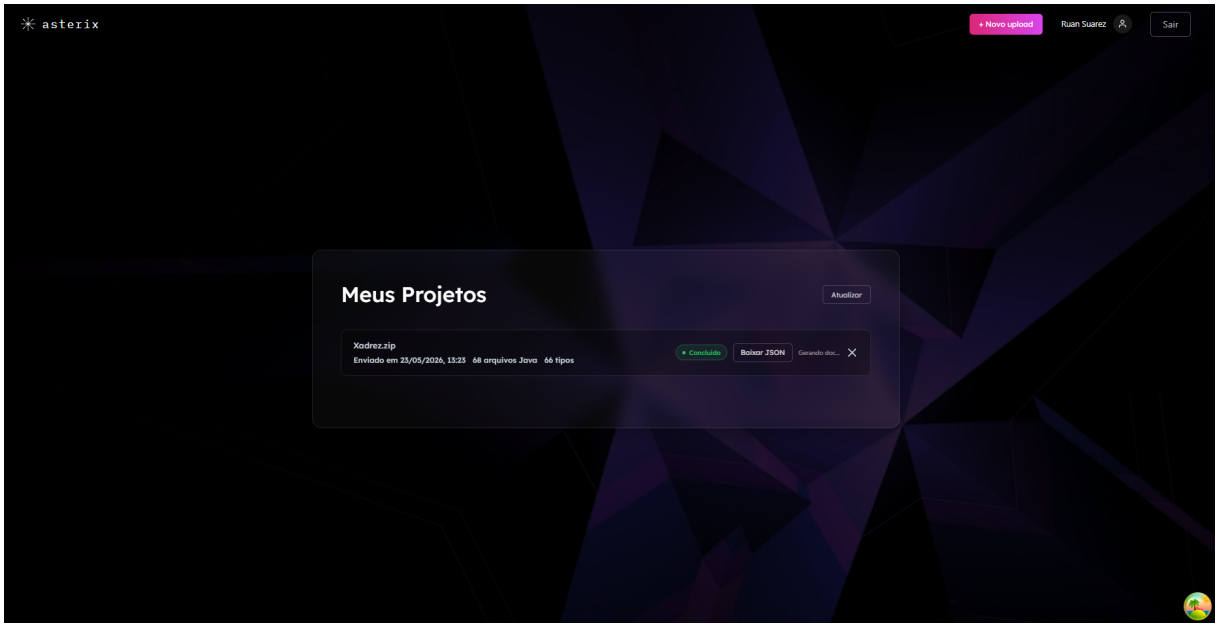
Figura 24 – *Homepage* do Sistema



Autoria própria

Na página ”Meus Projetos”é possível obter-se uma listagem dos projetos enviados para análise e documentação, seu devido status de progresso e a opção de descarregar o JSON gerado e a documentação na máquina do usuário, conforme apresentado na Figura 25.

Figura 25 – Página de Meus Projetos



Autoria própria

5.3 Documentação Gerada

Após o processo de mapeamento, limpeza, estruturação e análise por LLM, a documentação por fim gerada é condensada em *Markdown* e convertida para o formato HTML, possibilitando ao usuário uma interface gráfica intuitiva e de fácil navegação e edição. A documentação gerada contém dados importantes como relações entre classes, hierarquias, *javadocs* existentes previamente no código e categorias selecionáveis. A Figura 26 apresenta um exemplo de documentação gerada para um código legado de Xadrez em Java.

Figura 26 – Documentação Gerada

Documentação Técnica do Projeto

Visão Geral da Arquitetura

O projeto é implementado em Java e utiliza uma arquitetura orientada a objetos. A estrutura do projeto é dividida em categorias que representam as diferentes partes do jogo de xadrez.

Sumário das Categorias

- **Command**: Contém classes responsáveis por executar comandos específicos no jogo, como iniciar um novo jogo, salvar o jogo, exibir coordenadas, etc.
- **Iterator**: Implementa estratégias de iteração para lidar com listas de movimentos e jogadas.
- **Jogo**: Representa as regras e lógica do jogo de xadrez, incluindo estados, jogadores, jogadas e tipos de estratégias.
- **Mediator**: Define a comunicação entre as diferentes partes do jogo.
- **Peca**: Contém classes que representam as peças do xadrez, incluindo fábricas de peças e estratégias de movimento.
- **Strategy**: Implementa estratégias de movimento para cada tipo de peça.
- **Tabuleiro**: Representa o tabuleiro do jogo de xadrez e lida com operações relacionadas a ele.
- **View**: Contém classes responsáveis por renderizar a interface gráfica do usuário.

Detalhes das Categorias

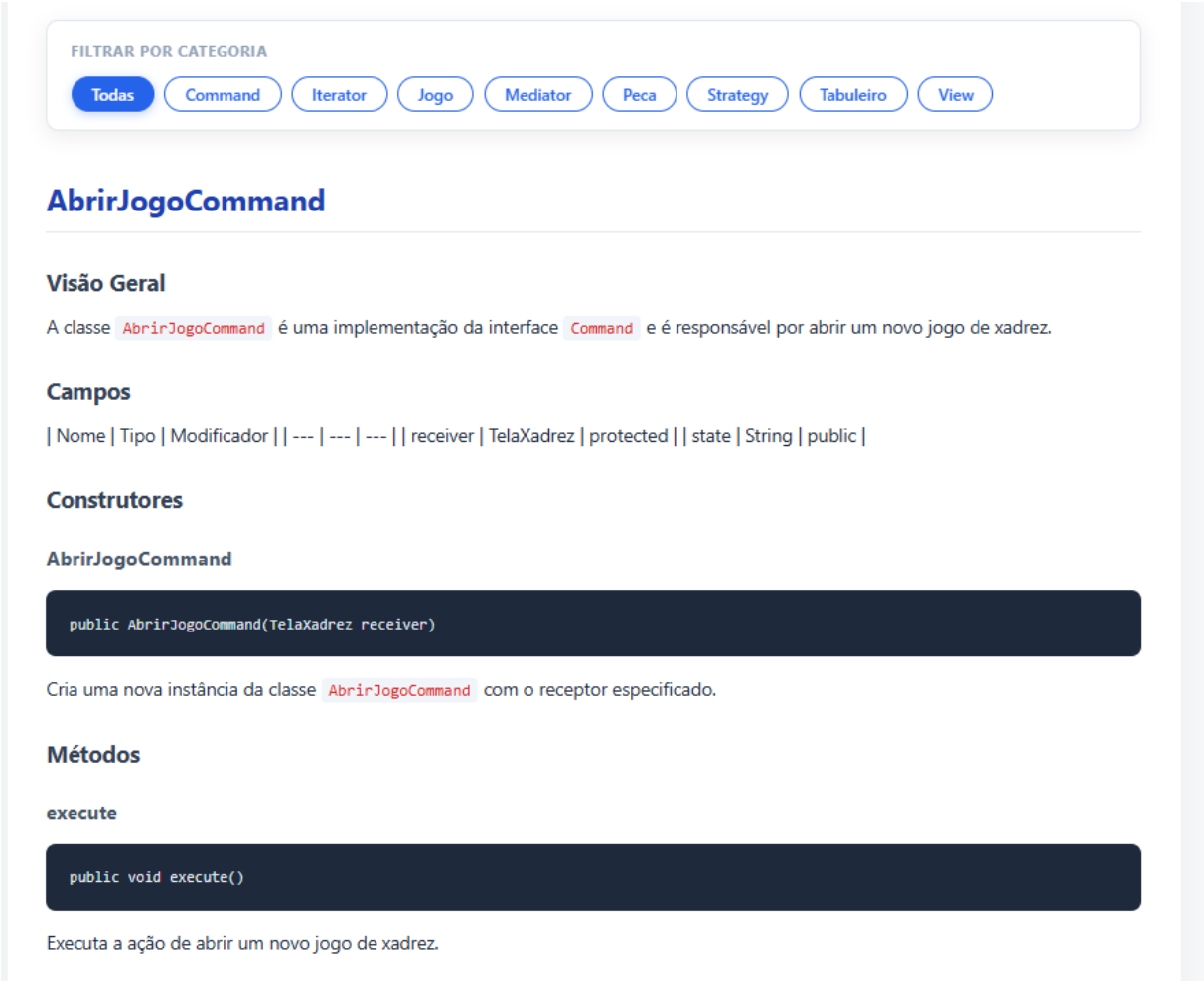
Command

- `AbrirJogoCommand` : Executa o comando de abrir um novo jogo.
- `Command` : Interface que define o método de execução de comandos.
- `ExibirCoordenadasCommand` : Executa o comando de exibir coordenadas no tabuleiro.
- `ExibirListaMovimentosCommand` : Executa o comando de exibir a lista de movimentos possíveis.
- `JogadaPossivelCommand` : Executa o comando de verificar se uma jogada é possível.
- `NovoJogoCommand` : Executa o comando de criar um novo jogo.
- `NovoJogoCommandAvancado` : Executa o comando de criar um novo jogo com configurações avançadas.
- `SairCommand` : Executa o comando de sair do jogo.

Autoria própria

Já na Figura 27, é possível observar a separação da documentação em diferentes categorias, possibilitando fácil navegação e entendimento do código enviado.

Figura 27 – Categorias da Documentação



Autoria própria

Dessa maneira, é possível obter um sistema funcional e com grande valor agregado para possibilitar melhor inserção de novos desenvolvedores em contextos de sistemas legados na linguagem Java, reduzindo a carga cognitiva inerente a este processo. Através da documentação gerada e do código convertido em JSON, a possibilidade de acesso à informações claras e o leque de opções que se abrem para serem utilizadas com esses artefatos posteriormente se expande, justificando o desenvolvimento do presente trabalho.

6 Conclusões

Neste trabalho, foi proposto e descrito o desenvolvimento da aplicação Asterix, com o objetivo central de fornecer uma plataforma de fácil acesso e intuitiva para desenvolvedores em processo de *onboarding* em projetos legado em Java através da redução da carga cognitiva relacionada a sistemas neste contexto por meio da geração de documentação técnica e descritiva.

A utilização de tecnologias e *frameworks* modernos e robustos, como o Spring Boot, React.JS e PostgreSQL integrados possibilitou o desenvolvimento de um projeto escalável e em acordo com padrões arquiteturais e tópicos mister na Engenharia de *Software*, como o DDD e princípios *SOLID*. Através da integração das funcionalidades internas presentes no sistema com a vasta gama de possibilidades fornecidas pelo LLM externo, aliados com a capacidade do Processamento de Linguagem Natural (PLN) fornecidos por estes, foi possível realizar a criação de uma ferramenta com valor agregado para possibilitar a criação de um mapeamento e documentação detalhados e visualmente amigáveis ao desenvolvedor em *onboarding*, sem infrigir questões de privacidade previstas em normas como a LGPD.

Através dos resultados obtidos por meio do desenvolvimento do trabalho, torna-se possível responder as questões de pesquisa anteriormente levantadas:

RQ1 Como a extração e representação estrutural de códigos legados em Java impactam a precisão e a utilidade da documentação gerada por LLMs?

RA1 A extração, mapeamento e representação estrutural de códigos legados em Java para formatos considerados de melhor compreensão á LLMs permite o aprofundamento da análise realizada por estes, através da estruturação hierárquica e livre de ruído presente em muitos contextos de sistemas legados na linguagem.

A limpeza realizada em etapa anterior à análise e á geração de documentação permite entregar ao modelo uma estrutura limpa e objetiva, garantindo que pontos de atenção no funcionamento de grandes modelos de linguagem como a janela de contexto e limite de requisições sejam otimizados. Ademais, o código extraído e estruturado permite a redução drástica de alucinações por parte dos LLMs, tendo em vista a restrição objetiva imposta ao formato enviado para análise, o JSON.

Por fim, a documentação gerada apresenta-se de fácil manuseio por parte do usuário e semanticamente coerente, garantindo usabilidade, tendo em vista a otimização do uso da janela de contexto do modelo, possibilitando que todas as informações presentes no documento sejam de utilidade real para o usuário.

RQ2 De que maneira a documentação criada por IA atua na mitigação da carga cognitiva de desenvolvedores durante o processo de *onboarding*?

RA2 A documentação gerada por IA fornece a descrição e o uso de classes, métodos e interfaces do projeto legado enviado, sem necessariamente depender de artefatos existentes no código como *Javadocs* e comentários, bem como simplifica a descrição hierárquica entre as diferentes camadas.

A abordagem adotada no sistema para documentar o código reduz drasticamente a carga extrínseca do desenvolvedor - o esforço mental exercido de maneira exaustiva para decifrar sintaxes complexas e a desorganização inerente de códigos legados, que foram implementados gradualmente ao longo do tempo por antigos desenvolvedores.

A conversão em árvore sintática abstrata do código legado para posterior análise e documentação pelo LLM, através da limpeza para redução de ruído, permite a simplificação de regras de negócio que, em contextos de *onboarding*, exigem ao desenvolvedor a revisitação de diferentes trechos de código e a consequente necessidade de retenção de muitas informações em memória de trabalho. De maneira inerente, o processo de simplificação e descrição de regras e lógicas permite a redução do esforço exercido e do raciocínio exacerbado gasto pelo desenvolvedor, possibilitando o seu redirecionamento para a carga intrínseca necessária - a compreensão de fato das regras de negócio e domínio do código-fonte.

Ademais, o mapeamento e direcionamento de *insights* arquiteturais permite o levantamento prévio de possibilidades a serem utilizadas para executar a tarefa designada no contexto da inserção do desenvolvedor ao código legado, ou informações arquiteturais a serem consideradas do sistema, garantindo redução do esforço para a compreensão, validação e alteração do mesmo.

RQ3 De que maneira o Asterix se diferencia de outras ferramentas de documentação de códigos, como o Doxygen?

RA3 O Asterix propõe um sistema de extração, mapeamento e análise sintático e semântico de códigos legados em Java, enfatizando a busca em reduzir a carga cognitiva de desenvolvedores em processo de *onboarding* neste sistemas. Diferentemente de ferramentas como o Doxygen, que realizam o mapeamento puramente sintático do código, a presente aplicação realiza, além da limpeza e análise sintática, a geração de *insights* e a análise da estrutura mapeada por meio da utilização de LLMs com apoio de PLN, garantindo uma documentação com detalhes técnicos importantes e inferências ativas de regras de negócio implícitas. Enquanto geradores estáticos são reflexos apenas da topologia do código, o Asterix aponta dependências cíclicas e *hotspots* no sistema, funcionando como um consultor analítico, entregando ao desenvolvedor um modelo mental pronto e contextualizado, preenchendo lacunas que a análise somente sintática é incapaz de suprir.

O desenvolvimento do sistema Asterix reflete, ainda, algumas das maiores dores existentes na Engenharia de *Software* - a ausência de documentação técnica de sistemas, a criação e manutenção de sistemas críticos e com regras de negócio enges-

sadas e o esforço necessário para desenvolvedores em serem inseridos nestes para as mais diversas finalidades - desde a re-engenharia até a manutenção do código. Os resultados obtidos justificam a necessidade da adoção prévia de padrões de documentação e criação de código no desenvolvimento de sistemas, garantindo que sejam escaláveis e de fácil compreensão.

A documentação gerada pelo sistema possui grande valor agregado para o entendimento do funcionamento geral das classes, métodos, atributos e interfaces do código e das regras de negócio nele existentes, permitindo uma visualização clara e intuitiva de como o sistema opera. Entretanto, é reconhecível a dependência de sistemas externos, como o próprio LLM utilizado para análise, podendo gerar riscos relacionados à privacidade de dados e aos custos associados a utilização destas tecnologias, este último dependendo de modelo a modelo.

Ademais, a utilização em larga escala do sistema pode tornar excessivo o uso dos modelos de linguagem, trazendo a tona pontos importantes como a própria escalabilidade do sistema. Estes pontos são abordados no seguinte capítulo, tendo em vista a possibilidade de aprofundamento em trabalhos futuros.

6.1 Trabalhos Futuros

A partir dos resultados obtidos através do desenvolvimento do presente trabalho, é possível delimitar pontos mister a serem abordados em trabalhos futuros, corroborando para o aprimoramento dos conhecimentos aplicados e possíveis melhorias na abordagem utilizada.

A utilização de LLMs existentes no mercado restringe o conhecimento especializado de suma importância para os modelos, bem como torna dependente do funcionamento de tecnologias externas e custosas a longo prazo e grande demanda. O desenvolvimento e treinamento de modelos locais e próprios com informações relevantes ao contexto pode tornar o processo de análise e documentação mais preciso e, por consequência, obter-se uma documentação com detalhes técnicos mais ricos e polidos. Além disso, este processo pode garantir melhor adequação no que diz respeito à privacidade dos dados enviados para análise, adequando o funcionamento às diretrizes impostas através da LGPD.

Outrossim, o processo de envio da estrutura hierárquica gerada para o LLM torna necessário a divisão em partes do código (os *chunks*) em decorrência dos limites de entrada dos modelos utilizados. A utilização de modelos mais avançados, com maior capacidade de entrada ou até modelos locais que comportem maior quantidade de dados pode garantir uma análise mais fluida e precisa do código como um todo, gerando resultados com informações mais assertivas com o que foi enviado.

O refinamento do *prompt* enviado para instruir o agente configurado através do LLM, por meio da descrição de padrões e diretrizes específicas ao contexto de desenvolvimento, pode resultar em melhoria na documentação gerada, análises mais robustas e em acordo com as realidades de desenvolvimento de sistemas, através da instrução, por meio do *system prompt*, de padrões a serem adotados e das possibilidades de abordagem a serem utilizadas para documentação.

Ademais, a existência de autenticação do usuário sem a possibilidade de recuperação de senha pode, em cenários reais com o sistema no ar, causar transtornos para clientes com registro na plataforma mas que porventura possam ter esquecido da senha registrada. Dessa maneira, a criação de uma lógica de recuperação de senha, desde que implementada de maneira a respeitar os dados confidenciais do usuário, pode possibilitar melhor usabilidade e maior aderência à aplicação, aliando tecnologias como mensageria e banco de dados para tornar possível a inserção de uma nova senha para o cliente.

O presente trabalho, apesar de gerar a documentação com detalhes técnicos de regras de negócio, métodos e atributos, não passou por um processo de estudo de caso utilizando desenvolvedores fora do meio dos autores e relacionados. A realização de um experimento elaborado, através de aplicações do cenário estudado aliado ao uso do Asterix, pode trazer à tona resultados numéricos, precisos e justificáveis, contribuindo para o aprofundamento do tema em futuros trabalhos relacionados ou não.

Por fim, destaca-se a importância da maior abrangência das linguagens de programação em alto nível suportadas para documentação - apesar do presente trabalho enfatizar e possibilitar a documentação da linguagem Java, não necessariamente deve-se ater somente a esta. O mapeamento e estruturação para possibilitar a análise e documentação de sistemas legados em outras linguagens de programação existentes, como Python e JavaScript, permite a ampliação da gama de possibilidades apresentados neste trabalho.

Referências

ANNELA, L. Ai-augmented legacy modernization: Transforming enterprise systems with smart automation. *Journal of Computer Science and Technology Studies*, v. 7, n. 5, p. 119–128, 2025. 21

Anthropic. *How Claude Code works*. [S.l.], 2025. Documentação oficial do Claude Code. Disponível em: <https://code.claude.com/docs/en/how-claude-code-works>. 24

ARAUJO, A. B. d. O. Inteligência artificial na documentação de sistemas legados: uma abordagem comparativa de ferramentas. Universidade Estadual do Piauí, 2025. 27

BODUCH, A. *React and react native*. [S.l.]: Packt Publishing Ltd, 2017. 26

BOUKHARI, M. E.; KHARMOUM, N.; ZITI, S. Ai-powered architecture refactoring: From legacy systems to modern patterns. *International Journal of Advanced Computer Science & Applications*, v. 16, n. 12, 2025. 27

BROWN, S. Software architecture for developers. *Coding the Architecture*, 2013. 37

CHOMA, D.; CHWALEBA, K.; DZIEŃKOWSKI, M. The efficiency and reliability of backend technologies: express, django, and spring boot. *Informatyka, Automatyka, Pomiary w Gospodarce i Ochronie Środowiska*, v. 13, 2023. 42

CRK, I.; KLUTHE, T. Assessing the contribution of the individual alpha frequency (iaf) in an eeg-based study of program comprehension. In: IEEE. *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. [S.l.], 2016. p. 4601–4604. 22

DARPA. *Translating All C to Rust (TRACTOR)*. 2024. Acessado em: 23 mar. 2026. Disponível em: <https://www.darpa.mil/research/programs/translating-all-c-to-rust>. 25

DIMITRIJEVIC, N.; ZDRAVKOVIC, N.; MILICEVIC, V. An automated grading framework for the mobile development programming language kotlin. 2023. 20

DRIESSEN, V. *A successful Git branching model*. 2010. Disponível em: <https://nvie.com/posts/a-successful-git-branching-model/>. 33

DVIVEDI, S. S. et al. A comparative analysis of large language models for code documentation generation. In: *Proceedings of the 1st ACM international conference on AI-powered software*. [S.l.: s.n.], 2024. p. 65–73. 22

FAN, A. et al. Large language models for software engineering: A survey. *arXiv preprint arXiv:2308.10620*, 2023. 25

GONÇALES, L. J. Cogeff: uma abordagem para mensurar a carga cognitiva de desenvolvedores em tarefas de compreensão de código. Universidade do Vale do Rio dos Sinos, 2022. 22

HAMAAMIN, R. A.; ALI, O. M. A.; KAREEM, S. W. Java programming language: time permanence comparison with other languages: a review. In: EDP SCIENCES. *ITM Web of Conferences*. [S.l.], 2024. v. 64, p. 01012. 20

HOSSAIN, A. A systematic mapping study on scrum and kanban in software development. 2023. 31

MARTINEZ, D. D.; REMEGIO, A.; LINCOPINIS, D. R. A review on java programming language. *ResearchGate. URL: https://www.researchgate.net/publication/371166744_A_Review_on_Java_Programming_Language (accessed 04.02. 2024)*, 2023. 20

MONTEIRO, A.; VIEIRA, G. Guiding legacy systems for evolution. 2022. 18

PARK, J. S. et al. *Generative Agents: Interactive Simulacra of Human Behavior*. 2023. Disponível em: <https://arxiv.org/abs/2304.03442>. 26

RAI, S.; BELWAL, R. C.; GUPTA, A. A review on source code documentation. *ACM Transactions on Intelligent Systems and Technology (TIST)*, ACM New York, NY, v. 13, n. 5, p. 1–44, 2022. 22

RAVSHANOVICH, A. R. Database structure: Postgresql database. *Psixologiya va sotsiologiya ilmiy jurnali*, v. 2, n. 7, p. 50–55, 2024. 27

SAMUEL, K. Retrieval-augmented llm copilots for erp tasks and cognitive load reduction. 2025. 22

SOUZA, H. V. de. Investigação do uso de técnicas de rag na refatoração de sistemas legados com base na análise de “code smells”. 2025. 21

SWELLER, J. Cognitive load theory. In: *Psychology of learning and motivation*. [S.l.]: Elsevier, 2011. v. 55, p. 37–76. 21

VASWANI, A. et al. Attention is all you need. In: *Advances in neural information processing systems*. [S.l.: s.n.], 2017. p. 5998–6008. 23

WOOLDRIDGE, M. *The road to conscious machines: The story of AI*. [S.l.]: Penguin UK, 2020. 25

YAO, S. et al. Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. 25

ZIEGLER, A. et al. Productivity assessment of neural code completion. *arXiv preprint arXiv:2205.09338*, 2022. 23