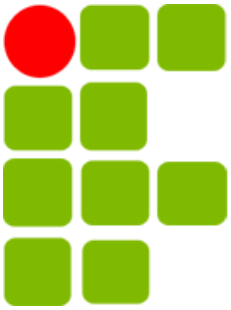


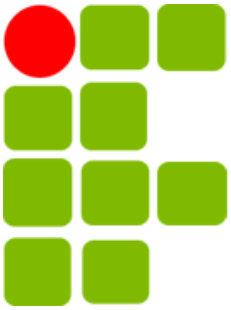
Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Curso Superior Bacharelado em Engenharia de Computação

Clarity: Sistema Inteligente para Geração Automatizada de Documentação Técnica em Projetos de Programação

Fabrício Moreno da Silva
Ynnayron Juan Lopes da Silva

Orientador: Prof. Dr. Elmano Ramalho Cavalcanti, DSc.

Campina Grande – PB
Junho de 2026



Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Curso Superior Bacharelado em Engenharia de Computação

Clarity: Sistema Inteligente para Geração Automatizada de Documentação Técnica em Projetos de Programação

Fabrício Moreno da Silva
Ynnayron Juan Lopes da Silva

Trabalho de conclusão de curso apresentado ao Curso de Bacharelado em Engenharia de Computação, do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba – Campus Campina Grande, em cumprimento às exigências parciais para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Elmano Ramalho Cavalcanti, DSc.

Campina Grande – PB
Junho de 2026

Catalogação na fonte:
Ficha catalográfica elaborada por Thiago Ferreira Cabral de Oliveira – CRB 15/628

S586c SILVA, Fabrício Moreno da.
Clarity: Sistema inteligente para geração automatizada de documentação técnica em projetos de programação / Fabrício Moreno da Silva, Ynnayron Juan Lopes da Silva. - Campina Grande, 2026.
73 f.

Trabalho de Conclusão de Curso (Graduação em Bacharelado em Engenharia de Computação) - Instituto Federal da Paraíba, 2026.
Orientador: Prof. Elmano Ramalho Cavalcanti.

1. Documentação de software. 2. Modelos de linguagem de grande escala. 3. Orquestração de agentes de LLM. 4 I. Silva, Ynnayron Juan Lopes da. II. Cavalcanti, Elmano Ramalho. III. Instituto Federal de Educação, Ciência e Tecnologia da Paraíba. IV. Título.

CDU 004.8

AGRADECIMENTOS

Agradecimentos de Fabrício Moreno da Silva

Queria agradecer primeiramente à Deus, meu maior suporte e a quem sempre se pode recorrer em qualquer momento, independente da dificuldade.

Segundo, à minha família. Meu irmão, Guilherme, e principalmente os meus pais, Fabíola e Marivaldo, que desde sempre, fizeram de tudo para que eu pudesse ser uma boa pessoa, um bom profissional, e estar onde eu estou hoje. Vocês são incríveis!

E também não posso deixar de fora ela, Maria Luíza, que me acompanha desde os primeiros dias de curso, que dividiu toda essa experiência universitária comigo, mesmo fazendo outro curso completamente diferente, sempre me entendeu em todos os momentos e me deu todo tipo de suporte que precisei. Você é minha luz e minha morada.

Ao meu grande amigo Liedson Augusto, por toda a parceria nos dois anos de estágio juntos, nos cinco anos de curso e nas correrias de todos os eventos que participamos juntos. E, de forma ainda mais especial, ao meu colega e coautor Ynnayron Juan, por toda a dedicação, empenho e parceria durante todo o curso, e sobretudo, na construção desse trabalho. Para sempre, levarei vocês mais do que como amigos, mas sim como irmãos.

E por fim, mas não menos importante, ao nosso orientador, Prof. Dr. Elmano Ramalho, e a todos os professores e servidores do IFPB, que contribuíram para a minha formação ao longo do curso.

Obrigado à todos.

Agradecimentos de Ynnayron Juan Lopes da Silva

Agradeço primeiramente a minha família que me formaram como pessoa, principalmente o meu pai , Francisco, por todo os ensinamentos e suporte através desses anos.

Agradeço aos meus amigos Liedson Augusto e Fabrício Moreno por toda a companhia e amizade durante esses 5 anos, todas as adversidades vencidas durante a formação não seriam as mesmas sem o apoio de vocês e por fim, quero agradecer a mim mesmo, por acreditar em mim, por nunca desistir e por ser apenas eu em todos os momentos.

E por ultimo, agradeço ao nosso orientador Prof Dr.Elmano Ramalho junto ao corpo de docentes do IFPB por contribuirem na minha formação profissional ao longo do curso.

“I will break away, I’ll find myself today”

— Linkin Park, “Somewhere I Belong”.

RESUMO

A documentação técnica é um artefato essencial do ciclo de vida do software, mas é frequentemente negligenciada, sobretudo em ambientes ágeis, o que resulta em arquivos README incompletos ou desatualizados e no acúmulo de dívida técnica documental. Este trabalho apresenta o Clarity, um sistema de geração automatizada de documentação técnica implementado como extensão para o Visual Studio Code. A solução emprega uma arquitetura de orquestração de agentes de Modelos de Linguagem de Grande Escala executados localmente por meio do Ollama: um agente analisa o código-fonte por meio de análise estática e outro redige o arquivo README, ambos encadeados pelo *framework* CrewAI. Para adequar repositórios de diferentes tamanhos à janela de contexto dos modelos locais, adota-se uma estratégia de divisão em fragmentos (*chunking*). O protótipo foi publicado na loja oficial do editor sob licença MIT e avaliado por meio de uma rubrica com cinco critérios — completude, atualidade, clareza, utilidade e consistência — aplicada de forma comparativa a três repositórios reais, frente à ferramenta de mercado README-AI. Os resultados indicam alta fidelidade da documentação gerada em relação ao código, principal contribuição da abordagem, enquanto as limitações concentram-se em seções de natureza inferencial deixadas como marcadores, no vazamento de prefixo de *prompt* e na variabilidade entre execuções. Conclui-se que a geração automatizada e local de documentação técnica é viável e promissora como apoio à mitigação da dívida técnica documental, preservando a privacidade do código-fonte.

Palavras-chave: documentação de software; modelos de linguagem de grande escala; orquestração de agentes de LLM; dívida técnica; geração automática de documentação.

ABSTRACT

Technical documentation is an essential artifact of the software life cycle, yet it is often neglected, especially in agile environments, resulting in incomplete or outdated README files and the accumulation of documentation technical debt. This work presents Clarity, an automated technical documentation generation system implemented as a Visual Studio Code extension. The solution employs an architecture based on the orchestration of Large Language Model agents running locally through Ollama: one agent analyzes the source code via static analysis and another writes the README file, both chained by the CrewAI *framework*. To fit repositories of different sizes within the context window of the local models, a *chunking* strategy is adopted. The prototype was published in the editor's official marketplace under the MIT license and evaluated through a rubric with five criteria — completeness, currency, clarity, usefulness, and consistency — applied comparatively to three real repositories, against the market tool README-AI. The results indicate high fidelity of the generated documentation to the actual code, the main contribution of the approach, while the limitations concentrate on inference-based sections left as placeholders, prompt prefix leakage, and variability across executions. It is concluded that automated and local technical documentation generation is viable and promising as a means to support the mitigation of documentation technical debt while preserving source-code privacy.

Keywords: software documentation; large language models; LLM agent orchestration; technical debt; automatic documentation generation.

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

A evolução acelerada da indústria de software nas últimas décadas transformou profundamente a forma como sistemas computacionais são desenvolvidos, implementados e mantidos. Impulsionada por paradigmas como a computação em nuvem, arquiteturas de microsserviços e metodologias ágeis, essa evolução permitiu a construção de sistemas cada vez mais complexos e distribuídos. Segundo Sommerville (2019), essa crescente complexidade exige não apenas excelência na codificação, mas também um rigoroso processo de engenharia que garanta a manutenção, a escalabilidade e a longevidade dos artefatos produzidos.

Nesse contexto, a documentação emerge como um pilar fundamental do ciclo de vida do desenvolvimento de software. Longe de ser um mero registro descritivo, ela atua como um instrumento de comunicação essencial entre as equipes, facilita o processo de integração (*onboarding*) de novos desenvolvedores, serve de base para auditorias e apoia a conformidade com normas técnicas, contratuais ou regulatórias aplicáveis ao projeto, como políticas internas de qualidade e exigências de setores regulados (PRESSMAN; MAXIM, 2020). Uma documentação clara e atualizada reduz a sobrecarga cognitiva e acelera as atividades de manutenção e evolução do sistema, impactando diretamente a produtividade e a qualidade do produto final.

Contudo, apesar de sua importância reconhecida, a documentação é frequentemente tratada como uma atividade secundária e negligenciada, especialmente em ambientes que adotam metodologias ágeis, onde a entrega contínua de funcionalidades tende a ser priorizada em detrimento de artefatos de suporte. Essa lacuna processual acarreta a acumulação do que a literatura especializada denomina “dívida técnica” (FOWLER, 2009). Evidências empíricas dimensionam o problema: mapeamento sistemático da literatura conduzido por Narváez-Narváez, Pardo-Calvache e Orozco-Garcés (2023) confirmou que a dívida de documentação — caracterizada por artefatos ausentes, inconsistentes ou incompletos — é recorrente no desenvolvimento ágil e de difícil remoção; e estudo retrospectivo em projeto industrial com requisitos ágeis constatou que tarefas de manutenção associadas à dívida documental geraram esforço adicional equivalente a aproximadamente 47% do total estimado para o desenvolvimento e custo extra de cerca de 48% do valor da fase inicial do projeto (MENDES et al., 2016). A ausência ou desatualização da documentação representa, portanto, um débito que, embora não impeça a entrega imediata de funcionalidades, cobra juros elevados a longo prazo, materializados em retrabalho, dificuldades de depuração e maior risco de falhas operacionais.

A problemática se agrava sob a ótica da Gestão do Conhecimento. Nonaka e Takeuchi (1995) distinguem o conhecimento em tácito (implícito, pessoal) e explícito (formal, codificado). Grande parte do entendimento sobre a lógica de um sistema reside no conhecimento tácito dos desenvolvedores. Na ausência de documentação adequada — um mecanismo de ex-

ternalização desse conhecimento —, a rotatividade de profissionais resulta na perda irreparável de capital intelectual, comprometendo a capacidade da organização de compreender e evoluir seus próprios sistemas.

Embora a documentação de software abranja múltiplos artefatos — de especificações de requisitos a documentação de API e diagramas de arquitetura —, este trabalho concentra-se especificamente no arquivo README. Essa delimitação justifica-se por seu papel singular: versionado junto ao código e localizado na raiz do repositório, o README é a porta de entrada do projeto e, com frequência, o primeiro (e por vezes único) documento consultado por um novo desenvolvedor. É também um dos artefatos que mais rapidamente se desatualiza, por acompanhar diretamente a evolução do código. Atacar o README, portanto, endereça a parcela da dívida documental de maior alcance e menor custo de manutenção.

Diante da persistência desse problema, cabe questionar por que as soluções já disponíveis não o resolvem de forma satisfatória. Ferramentas tradicionais de documentação, como JSDoc, Doxygen e Sphinx, dependem da disciplina do desenvolvedor em manter comentários formatados no código e operam em nível de API, não do projeto como um todo. Assistentes de código baseados em IA, por sua vez, atuam de forma pontual e interativa, exigindo condução manual e, muitas vezes, o envio do código a servidores de terceiros. Persiste, assim, uma lacuna: a ausência de um fluxo automatizado, integrado ao editor e executável localmente, capaz de produzir um README fiel à estrutura real do repositório. Essas abordagens são retomadas e comparadas em detalhe no Capítulo 3.

Diante desse cenário, os recentes avanços nos Modelos de Linguagem de Grande Escala (LLMs) abrem uma nova fronteira para o problema. Ao demonstrarem capacidade de compreender e gerar texto técnico com alta fluência semântica — incluindo código-fonte —, esses modelos viabilizam a automação inteligente de tarefas que antes dependiam exclusivamente do esforço humano. Dessa forma, a pesquisa busca responder à seguinte questão: como a aplicação de LLMs pode automatizar a geração de documentação de software, na forma de README, de modo a apoiar a mitigação da dívida técnica documental e da perda de conhecimento organizacional? A relevância desta proposta reside em seu potencial para aumentar a eficiência das equipes, melhorar a qualidade e a manutenção do software e preservar o conhecimento crítico dentro das organizações.

1.2 OBJETIVOS

O objetivo geral deste trabalho é desenvolver um sistema inteligente de geração automatizada de documentação técnica que, por meio de uma arquitetura baseada em agentes de Modelos de Linguagem de Grande Escala (LLMs) executados localmente, seja capaz de analisar componentes, dependências e estrutura de projetos de software e produzir documentação estruturada sob demanda, na forma de um arquivo README, integrada ao ambiente de desenvolvimento Visual Studio Code. Para alcançá-lo, definiram-se os seguintes objetivos específicos:

- Investigar o estado da arte em documentação automática de software, dívida técnica documental e aplicações de LLMs — incluindo modelos locais de código aberto — na análise e compreensão de código-fonte, identificando as lacunas deixadas pelas soluções existentes.
- Definir uma arquitetura para apoio à documentação automática de sistemas de software, fundamentada na análise estática de código, na divisão do repositório em fragmentos (*chunking*) para adequação à janela de contexto e na orquestração de agentes de LLM executados localmente via Ollama.
- Implementar um protótipo funcional dessa arquitetura como extensão para o Visual Studio Code, com suporte a projetos nas linguagens JavaScript, TypeScript, Python e Java, acionado por comandos explícitos do desenvolvedor e gerando um arquivo README separado para revisão antes da adoção.
- Avaliar a qualidade da documentação gerada pelo protótipo em repositórios reais, segundo os critérios de completude, atualidade, clareza, utilidade e consistência, em caráter comparativo com a ferramenta de referência de mercado README-AI.

1.3 PROBLEMA/QUESTÃO DE PESQUISA

Devido à rápida evolução de projetos de programação, é comum que os arquivos README se tornem incompletos ou desatualizados logo após novos *commits* ou *pull requests*. Isso aumenta o tempo de *onboarding* e o volume de retrabalho, sobretudo em repositórios multilinguagem que combinam tecnologias como JavaScript, TypeScript, Python e Java. Embora os Modelos de Linguagem de Grande Escala (LLMs) tenham demonstrado capacidade de resumir e estruturar conhecimento a partir de bases de código extensas, ainda não existe um fluxo consolidado que integre ao editor a coleta automática de dados do repositório, a geração de documentação técnica útil e o controle do desenvolvedor sobre a revisão antes de qualquer substituição do README principal.

Um desafio adicional emerge quando se considera o uso de LLMs locais de menor escala em substituição a modelos *online*: a janela de contexto e a capacidade de inferência reduzidas exigem uma arquitetura de *pipeline* mais estruturada, na qual o modelo não precisa "inventar" o documento livremente, mas sim preencher uma estrutura a partir de dados já organizados por uma etapa anterior de análise estática do código.

Questão de pesquisa principal: como empregar LLMs locais, aliados à análise automática de repositórios de código-fonte e acionados por comandos no Visual Studio Code, para gerar documentação técnica na forma de README com qualidade suficiente para servir de ponto de partida confiável ao desenvolvedor? Essa questão desdobra-se nas seguintes subquestões:

- **SQ1 (fidelidade)**: em que medida a documentação gerada é fiel ao código real do repositório, evitando a inclusão de informações inexistentes (alucinações)?

- **SQ2 (utilidade)**: a documentação gerada é útil como apoio ao *onboarding* e à compreensão inicial do projeto?
- **SQ3 (viabilidade local)**: é viável, em termos de custo, privacidade e tempo de execução, produzir essa documentação inteiramente com modelos locais, sem recorrer a APIs externas?

1.4 JUSTIFICATIVA E RELEVÂNCIA

Do ponto de vista científico, há uma lacuna na literatura em relação a abordagens que integrem compreensão semântica via LLM com sinais estruturais extraídos diretamente do projeto — como arquivos, dependências, *scripts* e hierarquia de diretórios — e que avaliem a documentação gerada por critérios objetivos mensuráveis, como completude, atualidade, clareza e utilidade. A maior parte dos trabalhos existentes aborda geração de documentação em nível de função ou método isolado, não na perspectiva do projeto como um todo, que é justamente o que um README precisa capturar.

Do ponto de vista prático, a integração do fluxo de geração diretamente no editor de código reduz o atrito de adoção, pois o desenvolvedor não precisa abandonar seu ambiente de trabalho para documentar. A geração de um arquivo separado (README-CLARITY.md) antes de qualquer substituição do README principal garante que o usuário mantenha controle total sobre o que é incorporado ao repositório, eliminando o risco de sobrescrita acidental — algo especialmente relevante em equipes sob pressão por produtividade e com alta rotatividade de membros.

Adicionalmente, a escolha por LLMs locais de código aberto via Ollama confere ao Clarity uma vantagem concreta frente a soluções baseadas em APIs externas pagas: o uso da ferramenta não gera custos recorrentes por volume de requisições, não expõe o código-fonte a servidores de terceiros e funciona em ambientes sem acesso à internet — atributos relevantes em contextos corporativos com restrições de segurança e conformidade.

1.5 DELIMITAÇÃO E ESCOPO

Este trabalho delimita-se ao desenvolvimento do Clarity como extensão para o Visual Studio Code, com as seguintes restrições de escopo:

- **Linguagens suportadas**: JavaScript, TypeScript, Python e Java; arquivos em outras linguagens são identificados como não reconhecidos e ignorados no processo de análise;
- **Repositórios locais**: a ferramenta opera exclusivamente sobre o diretório aberto no VS Code no momento da execução do comando, sem integração com repositórios remotos (GitHub, GitLab, Bitbucket ou similares);

- **Tamanho do repositório**: não há limite formal imposto pela ferramenta, porém projetos de grande porte estão sujeitos às restrições de janela de contexto dos modelos locais utilizados; a estratégia de divisão em fragmentos (*chunking*) é adotada para mitigar esse efeito;
- **Ambiente de execução**: requer o interpretador Python 3.11 ou superior e o serviço Ollama (versão 0.1.x ou superior) instalado e em execução na máquina do desenvolvedor, com os modelos `deepseek-coder:6.7b` e `llama3:8b` previamente disponibilizados; a ferramenta não funciona sem acesso local ao serviço de modelos;
- **IDE suportada**: Visual Studio Code, por meio de extensão nativa acionada por comando explícito do desenvolvedor;
- **Escopo da avaliação**: os estudos de caso realizados neste trabalho limitam-se a repositórios cedidos pela empresa de um dos autores, não constituindo uma amostra estatisticamente representativa do universo de projetos de software; a avaliação comparativa com a ferramenta de referência, por sua vez, restringe-se a essas amostras e apoia-se em julgamento automatizado, de modo que seus resultados são tratados como indicativos e não conclusivos.

1.6 METODOLOGIA

Este trabalho caracteriza-se como uma pesquisa aplicada de natureza quali-quantitativa, orientada ao desenvolvimento de um artefato tecnológico com avaliação empírica. Do ponto de vista dos objetivos, classifica-se como exploratória e descritiva, uma vez que investiga a aplicabilidade de Modelos de Linguagem de Grande Escala na automação da geração de documentação técnica e descreve o comportamento da solução desenvolvida em cenários de uso real. Do ponto de vista dos procedimentos técnicos, adota a estratégia de estudo de caso para a avaliação do protótipo em repositórios reais, mediante avaliação comparativa por rubrica com uma ferramenta de referência de mercado. O trabalho foi conduzido nas seguintes etapas:

(i) Revisão direcionada sobre documentação de software, dívida técnica, LLMs — incluindo modelos locais de código aberto — e ferramentas de geração automática de documentação afins;

(ii) Projeto e implementação da extensão para o Visual Studio Code e do *pipeline* em Python — composto por análise estática multilinguagem, estratégia de *chunking* para adequação à janela de contexto, e orquestração de agentes de LLM via CrewAI com modelos locais servidos pelo Ollama — para varrer o repositório, extrair metadados estruturais e gerar o arquivo README-CLARITY.md;

(iii) Estudos de caso em repositórios reais cedidos pela empresa de um dos autores, avaliando a qualidade da documentação gerada pela ferramenta em cenários de uso concretos;

(iv) Avaliação comparativa por rubrica, com critérios de completude, atualidade, clareza, utilidade e consistência, aplicada por meio de julgamento automatizado e em comparação com a ferramenta de referência de mercado README-AI.

1.7 ESTRUTURA DO TRABALHO

Além desta introdução, este trabalho está organizado em mais seis capítulos. O Capítulo 2 apresenta a fundamentação teórica, abordando documentação de software e README, dívida técnica, análise estática de código, Modelos de Linguagem de Grande Escala e a orquestração de agentes de LLM. O Capítulo 3 revisa as ferramentas e os trabalhos relacionados, posicionando o Clarity frente ao estado da arte e da prática. O Capítulo 4 descreve a modelagem do sistema Clarity, detalhando sua arquitetura em camadas, as tecnologias empregadas, o fluxo de processamento e o esquema JSON que serve de contrato entre os agentes. O Capítulo 5 trata da implementação do protótipo e dos resultados obtidos, cobrindo a extensão para o Visual Studio Code, o *pipeline* em Python, a publicação da ferramenta e os estudos de caso em repositórios reais. O Capítulo 6 apresenta a avaliação formal da documentação gerada, por meio de uma rubrica aplicada de forma comparativa à ferramenta README-AI. Por fim, o Capítulo 7 reúne as considerações finais, as contribuições, as ameaças à validade e as sugestões de trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo reúne os fundamentos teóricos que sustentam o desenvolvimento do Clarity. Parte-se do papel da documentação na Engenharia de Software e do README como seu artefato central, passando pela dívida técnica documental que motiva o trabalho. Em seguida, apresentam-se as bases técnicas da solução: a análise estática de código, os Modelos de Linguagem de Grande Escala e a orquestração de agentes de LLM por meio do *framework* CrewAI. O capítulo encerra posicionando as ferramentas e os trabalhos relacionados frente à abordagem proposta.

2.1 ENGENHARIA DE SOFTWARE E O PAPEL CRÍTICO DA DOCUMENTAÇÃO

A Engenharia de Software é a disciplina que aplica princípios de engenharia para o desenvolvimento sistemático de software, visando garantir sua qualidade, manutenção e confiabilidade (PRESSMAN; MAXIM, 2020). Dentro do ciclo de vida do desenvolvimento, a documentação técnica é um artefato essencial que serve a múltiplos propósitos: facilita a comunicação entre os membros da equipe, orienta as atividades de manutenção e evolução do sistema, acelera a integração de novos desenvolvedores e constitui um registro vital para auditorias e conformidade regulatória.

Apesar de sua importância amplamente reconhecida, a prática de documentar é frequentemente relegada a um segundo plano, especialmente em ambientes que adotam metodologias ágeis, onde a entrega contínua de funcionalidades tende a ser priorizada em detrimento de artefatos de suporte. A criação manual de documentação é percebida como uma tarefa onerosa e de difícil sincronização com um código-fonte em constante mudança, resultando em documentos obsoletos, incompletos ou inexistentes. É nesse contexto que a automação inteligente da documentação, viabilizada pelos recentes avanços em Modelos de Linguagem de Grande Escala, apresenta-se como uma resposta promissora ao problema.

2.2 README

O README é o documento principal de um repositório e a peça central de entrada da documentação de um projeto de software. Versionado junto ao código-fonte e localizado na raiz do repositório, funciona como fonte de verdade para o *onboarding* de novos colaboradores e para o uso básico da solução: descreve propósito e escopo, requisitos, instruções de instalação e execução com exemplos reproduzíveis, comandos de *build* e testes, matriz de compatibilidade e versões, política de suporte e segurança, licença e indicações de materiais aprofundados, como diretrizes de contribuição, registros de decisões arquiteturais (ADRs), diagramas C4 e *runbooks*.

A qualidade do README impacta diretamente a facilidade de adoção e a manutenção do software: melhora a sua descoberta, tende a reduzir o tempo de ambientação (*ramp-up*) de no-

vos membros, previne uso incorreto e estabelece uma governança mínima do projeto. Como princípio, deve ser claro, conciso, orientado a tarefas e continuamente atualizado, delegando os detalhes mais extensos aos documentos complementares. A dificuldade de manter o README sincronizado com a evolução contínua do código-fonte é amplamente reconhecida: Tan, Wagner e Treude (2023) demonstraram empiricamente que referências a elementos de código em documentação de repositórios tornam-se desatualizadas de forma frequente e silenciosa à medida que o código evolui, sendo essa uma das principais razões pelas quais a documentação passa a enganar em vez de orientar desenvolvedores e usuários. Adicionalmente, Luo et al. (2024) destacam que a Primeira Lei de Evolução de Software de Lehman implica que qualquer programa em uso continuará a evoluir, tornando inevitável o desalinhamento entre código e documentação caso não haja mecanismos ativos de atualização.

É justamente essa dificuldade de manter o README consistente ao longo do ciclo de vida do projeto que motiva a proposta deste trabalho: ao automatizar sua geração a partir da análise estrutural do repositório, o Clarity busca reduzir o esforço necessário para que esse artefato cumpra sua função de forma contínua e confiável.

2.3 DÍVIDA TÉCNICA

A dívida técnica é uma metáfora utilizada na Engenharia de Software para descrever o custo adicional que uma equipe enfrenta no futuro devido a escolhas de *design* ou desenvolvimento de baixa qualidade no presente. Assim como uma dívida financeira, acumula-se ao longo do tempo se não for tratada e pode levar a problemas significativos no projeto.

O termo foi cunhado em 1992 por Ward Cunningham — um dos coautores do Manifesto Ágil e criador da *wiki* —, que propôs a analogia com dívidas financeiras para explicar a executivos por que era estratégico reservar tempo e orçamento para reestruturar o código (CUNNINGHAM, 1992). Posteriormente, Fowler (2009) popularizou e expandiu o conceito, propondo um quadrante para classificar os diferentes tipos de dívida técnica baseando-se em dois eixos: a intenção (deliberada ou inadvertida) e a prudência (prudente ou imprudente). Uma dívida pode ser contraída de forma deliberada e prudente, quando uma equipe conscientemente assume um atalho para cumprir um prazo estratégico, com um plano para refatorar o código posteriormente. Por outro lado, pode ser inadvertida e imprudente, quando a equipe simplesmente não possui o conhecimento necessário sobre boas práticas de *design*, gerando problemas sem se dar conta. Estudos mais recentes têm evidenciado que a dívida técnica é um fenômeno multidimensional de ampla prevalência na indústria: mapeamento sistemático da literatura cobrindo 39 artigos publicados entre 2010 e 2023 demonstrou que equipes ágeis enfrentam desafios específicos na gestão de dívidas acumuladas, especialmente no que diz respeito à identificação, monitoramento e priorização de itens de dívida ao longo de *sprints* (FREIRE et al., 2024).

Essas escolhas de baixa qualidade podem incluir código mal escrito, falta de documentação, arquiteturas inadequadas, ausência de testes ou atrasos na implementação de correções necessárias. Embora permitam uma entrega mais rápida no curto prazo, geralmente resultam em

complicações e custos adicionais no futuro, à medida que a equipe precisa lidar com problemas de manutenção, correção de *bugs* e dificuldades para incorporar novos recursos.

No contexto deste trabalho, o foco recai sobre a chamada dívida técnica documental — a parcela da dívida técnica originada especificamente pela ausência ou desatualização da documentação. Mapeamento sistemático da literatura conduzido por Narváez-Narváez, Pardo-Calvache e Orozco-Garcés (2023) identificou que esse tipo de dívida é, comparativamente, menos removido e mais lentamente tratado do que outras formas de dívida técnica, o que torna sua mitigação um problema de relevância prática crescente. Em termos de impacto mensurável, estudo retrospectivo em projeto industrial com requisitos ágeis registrou que tarefas de manutenção diretamente atribuíveis à dívida documental consumiram 455,3 horas adicionais em 18 meses — equivalente a aproximadamente 48% do custo da fase de desenvolvimento (MENDES et al., 2016). Reduzir essa modalidade de dívida exige três elementos complementares: processo (quando atualizar), padrão (o que cobrir) e ferramenta (como facilitar a atualização). O Clarity atua diretamente no terceiro elemento, oferecendo uma ferramenta integrada ao ambiente de desenvolvimento capaz de automatizar a geração do README a partir da análise do repositório. Cabe destacar que a dívida documental permanece pouco atendida mesmo pelas abordagens automatizadas mais recentes: mapeamento sistemático conduzido por Albuquerque et al. (2023), cobrindo 150 estudos primários sobre o uso de técnicas inteligentes na gestão de dívida técnica, constatou que a documentação está entre os tipos de dívida menos contemplados, concentrando-se o esforço em identificação e mensuração de dívida de código e de projeto. Levantamento junto a profissionais de software conduzido pelo mesmo grupo (ALBUQUERQUE et al., 2022) corrobora a percepção de que a gestão da dívida documental é reconhecida como relevante, porém raramente apoiada por ferramentas dedicadas — lacuna que motiva a proposta deste trabalho.

2.4 ANÁLISE ESTÁTICA DE CÓDIGO

A análise estática de código é uma técnica da Engenharia de Software que consiste no exame automatizado do código-fonte sem que este precise ser executado. Em contraste com a análise dinâmica — que observa o comportamento do programa em tempo de execução —, a análise estática opera diretamente sobre a estrutura textual e sintática do código, permitindo a extração de informações como funções, classes, dependências, parâmetros, tipos de retorno e métricas de complexidade. Sua aplicação vai desde a detecção de anomalias e vulnerabilidades de segurança até a geração de metadados estruturais para ferramentas de navegação e documentação de código (CLEM; THOMSON, 2021).

A base técnica da maioria das abordagens de análise estática é a Árvore Sintática Abstrata (AST — *Abstract Syntax Tree*). Uma AST é uma representação hierárquica e estruturada do código-fonte, na qual cada nó da árvore corresponde a uma construção sintática da linguagem — como uma declaração de função, um bloco condicional ou uma expressão aritmética — e as arestas representam as relações de composição entre essas construções (AHO et al., 2006).

Ao transformar o código em uma AST, torna-se possível percorrer e consultar sua estrutura de forma programática, sem depender de expressões regulares ou heurísticas frágeis baseadas em texto.

Nos últimos anos, ferramentas modernas de análise estática baseadas em AST tornaram-se mais acessíveis e consolidadas para uso em múltiplas linguagens. Destaca-se o *Tree-sitter*, uma biblioteca de geração de *parsers* incrementais desenvolvida originalmente pelo GitHub, que produz árvores sintáticas concretas com plena fidelidade ao código-fonte e suporta dezenas de linguagens de programação por meio de gramáticas intercambiáveis. O GitHub adota o *Tree-sitter* como fundação da sua infraestrutura de navegação simbólica de código em escala, servindo mais de 40.000 requisições por minuto sobre seis milhões de repositórios (CLEM; THOMSON, 2021). Essa maturidade das ferramentas de *parsing* multilinguagem representa um avanço considerável em relação às abordagens baseadas em reconhecimento de padrões textuais, ampliando a precisão e a confiabilidade da análise estática aplicada a projetos heterogêneos.

A disponibilidade dessas ferramentas varia entre linguagens de programação. O Python, por exemplo, oferece o módulo `ast` em sua biblioteca padrão, que permite analisar e percorrer a árvore sintática de qualquer código Python de forma nativa, confiável e sem dependências externas. Para linguagens como JavaScript, TypeScript e Java, abordagens baseadas em reconhecimento de padrões sobre o conteúdo textual dos arquivos são mais simples de implementar, porém apresentam limitações em cenários de código mais complexo ou com sintaxe não convencional.

No contexto do Clarity, a análise estática é a primeira etapa do *pipeline* de geração de documentação. O sistema percorre todos os arquivos do repositório e aplica um analisador específico por linguagem: para Python, utiliza a AST nativa para extrair com precisão funções, classes, parâmetros, *docstrings* e dependências; para JavaScript, TypeScript e Java, aplica reconhecimento de padrões sobre o conteúdo textual dos arquivos. Os metadados extraídos são então serializados em JSON e repassados ao agente analisador do CrewAI, que os utiliza como base para a geração do relatório técnico intermediário. Essa separação entre extração estrutural e compreensão semântica é o que permite ao sistema operar com LLMs locais de menor escala, reduzindo a carga de inferência exigida dos modelos.

A adoção de parsers baseados em AST para essas demais linguagens — por exemplo, via Tree-sitter — foi deliberadamente deixada como evolução futura, priorizando-se, nesta etapa, a viabilização do fluxo completo de geração de ponta a ponta.

2.5 A REVOLUÇÃO DOS MODELOS DE LINGUAGEM DE GRANDE ESCALA (LLMS)

O Processamento de Linguagem Natural (PLN), campo da Inteligência Artificial focado na interação entre computadores e a linguagem humana, passou por uma transformação radical nos últimos anos. Antes dessa transformação, arquiteturas como as Redes Neurais Recorrentes (RNNs) e suas variantes — notadamente o LSTM (*Long Short-Term Memory*), proposto no tra-

balho seminal de Hochreiter e Schmidhuber (1997) e que se manteve como referência na área por mais de duas décadas —, embora eficazes para sequências curtas, enfrentavam dificuldades em capturar dependências de longo prazo no texto, um desafio associado ao problema do desvanecimento do gradiente.

O ponto de inflexão ocorreu com a publicação do artigo "*Attention Is All You Need*" por Vaswani et al. (2017), que introduziu a arquitetura *Transformer*. Diferentemente de seus predecessores, o *Transformer* processa os dados de forma paralela e utiliza um mecanismo chamado autoatenção (*self-attention*), que permite ao modelo ponderar a importância de diferentes palavras — ou *tokens* — em uma sequência de entrada ao processar cada *token* individualmente. Em outras palavras, para entender o significado de uma palavra em uma frase, o modelo consegue "prestar atenção" a todas as outras palavras e identificar quais são as mais relevantes para contextualizá-la, superando as limitações das arquiteturas anteriores.

A arquitetura *Transformer* é tipicamente composta por duas partes: um codificador (*encoder*), que lê a sequência de entrada e constrói uma representação numérica rica em contexto, e um decodificador (*decoder*), que utiliza essa representação para gerar uma nova sequência de saída. Modelos como o BERT (DEVLIN et al., 2019) utilizam predominantemente a pilha de codificadores e são especializados em tarefas de compreensão de texto. Já modelos como a série GPT (BROWN et al., 2020) utilizam a pilha de decodificadores e são especializados na geração de texto, o que os torna particularmente adequados para tarefas como a produção de documentação técnica.

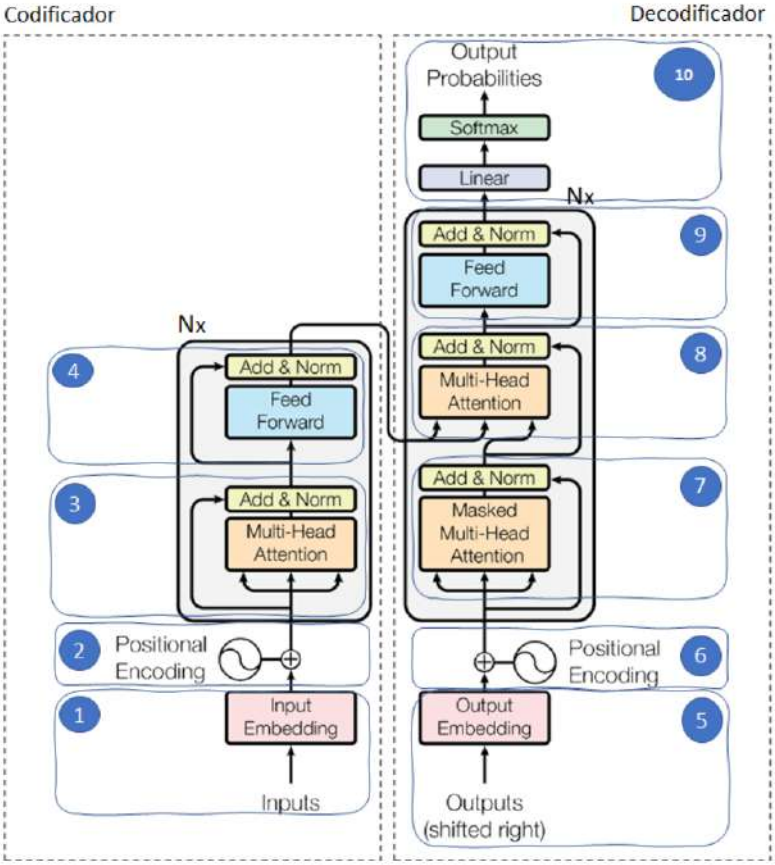
A arquitetura ilustrada na Figura 2.1 combina, portanto, um codificador que constrói uma representação contextualizada da entrada e um decodificador que, de maneira autorregressiva, gera a saída *token* a *token* até um marcador de fim de sequência. Para os fins deste trabalho, o relevante não é o detalhamento interno desse mecanismo, mas a consequência prática dele: modelos baseados em decodificador são especialmente aptos à geração de texto, o que fundamenta seu uso na produção de documentação.

O poder dos LLMs deriva de um processo de treinamento em duas etapas (BROWN et al., 2020):

1. **Pré-treinamento** (*pre-training*): o modelo é treinado de forma autossupervisionada em um *corpus* massivo de dados, contendo trilhões de palavras e trechos de código provenientes de diversas fontes. O objetivo é aprender padrões estatísticos, gramática, sintaxe, fatos sobre o mundo e capacidades de raciocínio, por meio da predição do próximo *token* em uma sequência ou do preenchimento de lacunas em um texto.

2. **Ajuste fino** (*fine-tuning*): após o pré-treinamento, o modelo base — de caráter generalista — pode ser especializado para tarefas específicas por meio de treinamento adicional em um conjunto de dados menor e de alta qualidade, voltado para a tarefa desejada, como pares de código-fonte e sua respectiva documentação.

Figura 2.1 – Arquitetura *Transformer*



Fonte: Adaptado de Vaswani et al. (2017).

Esses avanços na arquitetura e no método de treinamento permitem que os LLMs processem e compreendam código-fonte como uma forma de linguagem, identificando padrões, estruturas lógicas e o propósito semântico de funções e algoritmos. É essa capacidade que abre a fronteira para a automação inteligente de tarefas complexas da Engenharia de Software, como a geração de documentação técnica.

No entanto, a aplicação de LLMs a tarefas de Engenharia de Software traz um desafio central para este trabalho: a *alucinação*, fenômeno em que o modelo gera conteúdo plausível na forma, porém factualmente incorreto em relação ao contexto fornecido. No contexto da documentação de código, uma alucinação pode se manifestar como a menção a uma dependência, função ou *framework* que não existe no projeto — exatamente o tipo de erro que compromete a fidelidade de um README ao repositório real. Mitigar esse comportamento é um dos requisitos centrais da arquitetura proposta, e a estratégia adotada para tanto — restringir a geração a dados previamente extraídos por análise estática — é detalhada nos Capítulos 4 e 5.

Nos últimos anos, modelos de linguagem de código aberto com capacidade de execução local — como os utilizados neste trabalho, `deepseek-coder:6.7b` e `llama3:8b`, disponibilizados via Ollama — tornaram-se alternativas viáveis aos modelos proprietários de grande escala. Embora apresentem janelas de contexto e capacidade de inferência mais limitadas, esses

modelos eliminam a dependência de APIs externas pagas, preservam a privacidade do código-fonte e permitem uso em ambientes sem acesso à internet. A limitação de janela de contexto, no entanto, impõe restrições ao volume de código que pode ser processado em uma única chamada.

2.6 AGENTES DE LLM E ORQUESTRAÇÃO DE FLUXOS

A noção de *agente* tem origem no estudo dos Sistemas Multiagente (MAS — *Multi-Agent System*), campo da Inteligência Artificial que investiga a criação e a interação de múltiplos agentes autônomos em um ambiente compartilhado. Nessa tradição, um agente é definido como uma entidade computacional que percebe seu ambiente e atua sobre ele de forma autônoma para atingir seus objetivos, exibindo propriedades como autonomia, sociabilidade, reatividade e proatividade; e a inteligência do sistema emerge da interação social, da negociação e da coordenação descentralizada entre esses agentes (RUSSELL; NORVIG, 2021; WOOLDRIDGE, 2009). É importante, contudo, distinguir esse conceito clássico daquele empregado neste trabalho: o Clarity não constitui um MAS no sentido estrito, pois seus agentes não negociam, não competem nem coordenam suas ações de forma emergente em um ambiente compartilhado. O que se adota aqui é a noção, mais recente, de **agentes baseados em LLMs**.

No contexto dos Modelos de Linguagem de Grande Escala, um agente é comumente entendido como uma instância de um LLM à qual se atribui um papel (*role*) e um objetivo (*goal*) específicos, encapsulada de modo a executar uma tarefa bem delimitada dentro de um fluxo maior. Diferentemente dos agentes autônomos do MAS clássico, esses agentes não possuem sensores e atuadores próprios nem tomam iniciativa fora do escopo que lhes é designado: sua "autonomia" restringe-se à geração de texto a partir das instruções recebidas. A construção de soluções com vários desses agentes especializados, encadeados por um fluxo predefinido, é frequentemente designada na literatura e na prática de engenharia como *orquestração de agentes*, e é essa a abordagem que fundamenta a arquitetura do Clarity.

O princípio que justifica essa decomposição é a especialização por tarefa. Em vez de uma única instância de modelo responsável por analisar o código e redigir a documentação simultaneamente, o problema é dividido em dois agentes com responsabilidades distintas — o agente analisador, focado na extração e estruturação de metadados do código-fonte, e o agente redator, focado na produção do texto final do README —, cada um operando com um LLM distinto, otimizado para sua função específica. A interação entre eles não é dialógica nem negociada, mas sequencial e determinística: a saída estruturada do primeiro agente serve de entrada controlada para o segundo, conforme detalhado na Seção 2.7 e no Capítulo 4. Essa separação de responsabilidades é o mecanismo central pelo qual o sistema restringe o espaço de geração livre do redator e, com isso, mitiga alucinações.

2.7 O FRAMEWORK CREWAI PARA ORQUESTRAÇÃO DE AGENTES

Uma vez estabelecida a distinção entre o MAS clássico e a orquestração de agentes de LLM (Seção 2.6), *frameworks* modernos como o CrewAI oferecem as ferramentas práticas para implementar essa segunda abordagem. O CrewAI é um *framework* de código aberto projetado para orquestrar agentes baseados em LLMs, permitindo definir papéis, tarefas e um fluxo de execução que os encadeia para realizar objetivos complexos (MOURA, 2023).

A filosofia do CrewAI é facilitar a criação de equipes (*crews*) de agentes com papéis bem definidos, que trabalham de forma coordenada. Sua arquitetura é baseada em quatro componentes principais:

- **Agentes** (*Agents*): são os executores especializados. Cada agente é definido com um *role* (papel, ex.: "Analista de Código"), um *goal* (objetivo, ex.: "Identificar as dependências do código-fonte") e um *backstory* (contexto narrativo que orienta seu comportamento e tom de resposta).
- **Tarefas** (*Tasks*): são as unidades de trabalho específicas atribuídas a um agente. Cada tarefa descreve o que precisa ser feito e qual o resultado esperado, podendo depender da saída de tarefas anteriores.
- **Ferramentas** (*Tools*): são capacidades adicionais que os agentes podem utilizar para interagir com o ambiente externo e obter informações além de seu conhecimento intrínseco, como funções para leitura de arquivos, análise estática de código ou geração de estruturas de dados.
- **Equipes** (*Crews*): são o conjunto de agentes e tarefas orquestrados por um processo definido — sequencial ou hierárquico — que trabalham em conjunto para alcançar um objetivo final.

No Clarity, o CrewAI é utilizado para orquestrar um *pipeline* sequencial composto por dois agentes: o agente analisador, que utiliza o modelo `deepseek-coder:6.7b` via Ollama e tem como responsabilidade realizar a análise estática do código-fonte, consolidar os resultados dos fragmentos processados e gerar um relatório técnico intermediário; e o agente redator, que utiliza o modelo `llama3:8b` e tem como responsabilidade receber esse relatório estruturado e produzir o arquivo README-CLARITY.md final. Essa divisão de responsabilidades entre agentes com LLMs distintos permite que cada modelo opere dentro de suas competências específicas, otimizando a qualidade do resultado gerado.

3 TRABALHOS RELACIONADOS

A automação da documentação de software tem sido um desafio persistente na Engenharia de Software, evoluindo de extratores estáticos para assistentes baseados em inteligência artificial generativa. Este capítulo revisa as principais abordagens existentes, organizadas em três gerações tecnológicas, e posiciona o Clarity em relação a elas, explicitando a lacuna técnica e empírica que a ferramenta busca preencher.

3.1 FERRAMENTAS DE DOCUMENTAÇÃO ESTÁTICA E BASEADAS EM COMENTÁRIOS

Historicamente, ferramentas como JSDoc (para JavaScript), Doxygen (para C++/Java) e Sphinx (para Python) têm sido o padrão da indústria. Essas ferramentas operam através da extração de comentários formatados diretamente no código-fonte para gerar arquivos HTML ou PDFs. Embora eficazes para documentar APIs técnicas, elas possuem limitações críticas: dependem inteiramente da disciplina do desenvolvedor para escrever e atualizar os comentários manualmente. Caso o código mude e o comentário não seja revisado, a documentação torna-se instantaneamente uma "dívida técnica" (SOMMERVILLE, 2019).

3.2 ASSISTENTES DE CÓDIGO BASEADOS EM IA (PRIMEIRA GERAÇÃO)

A primeira geração de assistentes de código baseados em Inteligência Artificial, amplamente difundida a partir de 2021, introduziu o conceito de "IA como par de programação" (*AI Pair Programmer*). Ferramentas como o GitHub Copilot, alimentado pelo modelo Codex da OpenAI, e o Amazon CodeWhisperer consolidaram-se como extensões fundamentais nos Ambientes de Desenvolvimento Integrados (IDEs), como o Visual Studio Code e a suíte JetBrains (CHEN et al., 2021).

Diferente das ferramentas de autocompletar tradicionais, que se baseavam em análise estática e heurísticas simples de nomes de variáveis, esses assistentes oferecem um conjunto de capacidades fundamentais voltadas à produtividade imediata do desenvolvedor.

3.3 ASSISTENTES DE CÓDIGO BASEADOS EM IA (SEGUNDA GERAÇÃO)

A segunda geração de assistentes de IA marca a transição da "completude de código" para a "agência de codificação". Enquanto a primeira geração atuava como um preenchimento automático sofisticado, as ferramentas de segunda geração, como o modelo Claude (Anthropic) integrado ao ambiente de desenvolvimento, possuem maior autonomia e capacidade de execução.

Diferente de seus antecessores, esses assistentes de segunda geração não se limitam a sugerir trechos de código; eles são capazes de:

- **Manipulação de Arquivos e Diretórios**: ler e escrever em múltiplos arquivos de forma coordenada, permitindo a compreensão de contextos de projeto inteiros em vez de apenas funções isoladas.
- **Execução de Comandos de Terminal**: rodar testes, instalar dependências e executar *scripts* de *build* diretamente no ambiente de desenvolvimento.
- **Documentação Dinâmica**: gerar artefatos complexos, como arquivos README.md e documentações técnicas, analisando a estrutura completa dos diretórios.
- **Integração com Controle de Versão**: realizar operações de Git, como criar mensagens de *commit* contextualizadas e enviar as alterações diretamente para repositórios no GitHub.

3.4 TRABALHOS RELACIONADOS E PESQUISA ACADÊMICA

A literatura acadêmica e o ecossistema de código aberto têm explorado intensamente o potencial da Inteligência Artificial no suporte ao ciclo de vida do software. No âmbito das ferramentas práticas de código aberto, destaca-se o README-AI, um sistema que processa repositórios e utiliza Modelos de Linguagem de Grande Escala (LLMs) para produzir um README estruturado, oferecendo opções de customização e *templates*. De acordo com o autor do projeto, Salamie (2023):

> ReadmeAI é uma ferramenta para desenvolvedores que gera automaticamente arquivos README usando um mecanismo robusto de processamento de repositórios e modelos de linguagem avançados. Basta fornecer uma URL ou caminho para sua base de código e um README bem estruturado e detalhado será gerado. (SALAMIE, 2023)

Além dessas iniciativas, outros trabalhos relevantes contribuem para o estado da arte na área:

- Feng et al. (2020) — **CodeBERT**: este trabalho introduziu um modelo pré-treinado bimodal para linguagens de programação e linguagem natural. O CodeBERT é fundamental para a área, pois demonstrou que modelos treinados especificamente em código podem realizar tarefas de documentação e busca de código com uma precisão muito superior aos modelos de linguagem genéricos.
- Hou et al. (2023) — **Large Language Models for Software Engineering: A Survey**: esta pesquisa fornece um mapeamento abrangente sobre como os LLMs estão sendo usados em todas as fases da Engenharia de Software. Os autores destacam que, embora a geração de documentação seja uma das áreas mais beneficiadas, o desafio atual reside na

"fidelidade" da IA ao código real, reforçando a necessidade de abordagens como a do Clarity, que utilizam analisadores de código para embasar a geração textual.

- Zhang et al. (2023) — **Self-Edit: Fault-Aware Code Editor**: este estudo apresenta a ideia de agentes que revisam o próprio conteúdo gerado. Embora o protótipo atual do Clarity não implemente um ciclo de autorrevisão entre agentes, essa lógica de "autocorreção" inspira a arquitetura de múltiplos agentes adotada e aponta uma direção promissora: a inclusão de um agente revisor que verifique a saída do Agente Redator antes da entrega, elevando a confiabilidade da documentação gerada

A análise dessas três gerações revela um gap específico que o Clarity busca preencher. As ferramentas estáticas (JSDoc, Doxygen, Sphinx) produzem documentação fiel ao código, mas apenas em nível de API e dependentes de comentários mantidos manualmente — não geram uma visão de projeto como a de um README. Os assistentes de primeira geração aumentam a produtividade pontual, mas não produzem documentação estruturada de repositório. Os de segunda geração já geram READMEs completos, porém operam de forma interativa (exigindo condução manual), enviam código a servidores de terceiros e não isolam a etapa de extração estrutural da etapa de redação, o que os deixa suscetíveis a alucinações. O README-AI, por fim, é a ferramenta mais próxima do Clarity em propósito, mas emprega um agente único que aplica um gabarito fixo, com seções frequentemente deixadas como marcadores não resolvidos, e não preserva o README original. Nenhuma das abordagens combina, simultaneamente, os quatro atributos que caracterizam a proposta deste trabalho: geração em nível de projeto, execução integralmente local, separação entre análise estrutural e redação, e preservação do artefato original mediante geração de arquivo separado. É nessa interseção que o Clarity se posiciona, e é ela que a avaliação empírica do Capítulo 6 busca examinar.

O Quadro 3.1 sintetiza esse posicionamento, mas cabe uma leitura crítica de suas limitações. A comparação não deve ser interpretada como uma classificação absoluta de superioridade: cada ferramenta foi projetada para um propósito distinto, e vários dos atributos avaliados refletem decisões de escopo, não deficiências. Ferramentas como JSDoc e Doxygen permanecem superiores para documentação de API detalhada, tarefa fora do escopo do Clarity. Os assistentes de segunda geração oferecem capacidades de automação de tarefas que o Clarity não possui, como execução de comandos e edição direta de múltiplos arquivos. O próprio README-AI apresenta maior maturidade de configuração e suporte a mais linguagens e provedores de LLM do que o protótipo aqui descrito. A vantagem do Clarity concentra-se, portanto, em um nicho específico — a geração local, fiel e não destrutiva de um README em nível de projeto —, e não em uma superioridade generalizada. Ademais, por se tratar de um levantamento qualitativo conduzido pelos autores a partir da documentação pública de cada ferramenta, o quadro está sujeito a imprecisões decorrentes da evolução rápida dessas soluções, cujas capacidades podem ter mudado desde a consulta.

Característica	**JSDoc / Doxygen**	**GitHub Copilot (1ª Gen)**	**Cline / Claude (2ª Gen)**	**README-AI**	**Clarity**
Escopo principal	Documentação de API via comentários	Sugestão de código e comentários pontuais	Automação de tarefas e refatoração	README automatizado	README automatizado
Linguagens suportadas	Específicas (JS/TS, C++, Java)	Multilinguagem	Multilinguagem	C++, Go, Python, Rust, Swift, Java, JS/TS	JavaScript, TypeScript, Python, Java
Provedores de LLM	Não utiliza (baseado em regras)	OpenAI (Codex / GPT-4)	Anthropic (Claude 3.5 Sonnet)	OpenAI, Anthropic, Gemini, Ollama (local), modo *offline*	Ollama local (deepseek-coder:6.7b e llama3:8b)
Arquitetura de geração	Analisador estático (Regex/Parser)	Preditiva (*Single Prompt*)	Agêntica (acesso a arquivos/terminal)	Agente único	Multiagente (analisador + redator via CrewAI)
Integração no VS Code	Via extensões de terceiros	Extensão nativa (Chat/Inline)	Extensão nativa (agente autônomo)	Não — opera via CLI, Streamlit ou Docker	Sim — extensão nativa acionada por comando
Preserva README original	Sobrescreve conforme *tags*	N/A (sugere no editor)	Modifica arquivos diretamente	Não especificado	Sim — gera README-CLARITY.md separado

Fonte: Autores (2026).

Quadro 3.1: Comparativo de ferramentas de geração de documentação

4 MODELAGEM DO SISTEMA

Este capítulo detalha a arquitetura lógica e física do sistema Clarity, descrevendo as tecnologias integradas, o fluxo de dados entre agentes e a estrutura de decisão que permite a automação da documentação técnica.

4.1 ARQUITETURA GERAL

O Clarity é estruturado em uma arquitetura local em camadas, na qual a interface de usuário (extensão do Visual Studio Code) se comunica, por invocação de processo, com um motor de orquestração de inteligência artificial executado localmente em Python. A opção por uma arquitetura em camadas, em vez de um único módulo monolítico, permite isolar responsabilidades e trocar componentes de forma independente — por exemplo, substituir os modelos de linguagem sem alterar a extensão. A arquitetura é dividida em três camadas principais:

1. **Camada de Interação** (VS Code Extension): responsável pela captura de comandos do usuário e pela exibição do progresso e dos arquivos gerados. *Entrada*: comando do desenvolvedor e caminho do diretório aberto. *Saída*: invocação do *backend* e notificações na interface.

2. **Camada de Orquestração** (*backend* Python com CrewAI): onde reside a lógica dos agentes e das etapas de análise estática, *chunking* e consolidação. *Entrada*: caminho do repositório. *Saída*: o arquivo README-CLARITY.md e mensagens de progresso via saída padrão.

3. **Camada de Inferência** (Ollama): responsável pela execução local dos modelos de linguagem. *Entrada*: os *prompts* montados pelos agentes. *Saída*: o texto gerado por cada modelo, garantindo que nenhum dado do código-fonte saia da máquina do desenvolvedor.

A comunicação entre a camada de interação e a de orquestração ocorre por invocação de processo, com troca de dados serializados em JSON; entre a orquestração e a inferência, pela API local do Ollama. O tratamento de exceções segue um princípio de degradação controlada: falhas na análise de um arquivo individual (por exemplo, um arquivo com sintaxe inválida) não interrompem o *pipeline*, sendo o arquivo registrado como não processado e a execução prosseguindo com os demais.

4.2 TECNOLOGIAS UTILIZADAS

A escolha das tecnologias baseou-se em critérios de extensibilidade, desempenho em execução local e maturidade no ecossistema de engenharia de software. O Quadro 4.1 relaciona

as principais tecnologias empregadas e suas versões, e os parágrafos seguintes justificam as decisões mais relevantes.

Tecnologia	**Versão**	**Papel no projeto**
TypeScript	5.x	Linguagem da extensão do VS Code
Node.js	18+	Ambiente de execução da extensão
Python	3.11+	*Backend* de análise e orquestração
CrewAI	0.x	Orquestração dos agentes de LLM
Ollama	0.1.x	Execução local dos modelos
deepseek-coder	6.7b	Modelo do Agente Analisador
llama3	8b	Modelo do Agente Redator
vsce	—	Empacotamento e publicação da extensão

Fonte: Autores (2026).

Quadro 4.1: Tecnologias e versões utilizadas no desenvolvimento do Clarity

A escolha de **TypeScript e Node.js** para a extensão decorre de serem o ambiente nativo de desenvolvimento de extensões do VS Code: a API do editor é oferecida oficialmente para esse ecossistema, o que garante acesso estável ao painel de comandos, ao sistema de arquivos e aos componentes visuais, além de simplificar o empacotamento e a publicação na loja oficial. O **Python** foi adotado no *backend* de inteligência artificial por concentrar a maior parte das bibliotecas maduras para análise sintática de código, integração com modelos de linguagem e orquestração de agentes — incluindo o próprio CrewAI, que é distribuído como pacote Python. A separação entre uma extensão em TypeScript e um *backend* em Python, embora introduza a necessidade de comunicação entre processos, permite que cada camada empregue a linguagem mais adequada à sua tarefa.

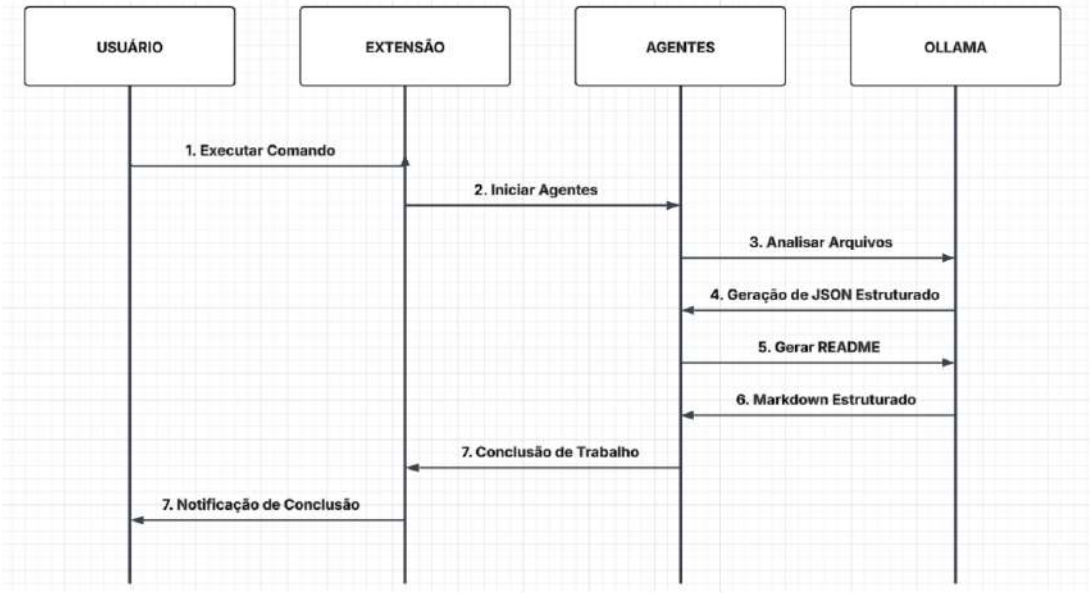
A opção pelo **Ollama** como camada de inferência é central para a proposta do trabalho. Ele permite executar modelos de linguagem inteiramente na máquina do desenvolvedor, sem envio de código a servidores externos, o que atende aos requisitos de privacidade, ausência de custos recorrentes e funcionamento *offline* discutidos na Seção 1.4. Adicionalmente, o Ollama expõe uma API compatível com a da OpenAI, o que facilita a substituição dos modelos e a eventual migração para outros provedores sem reescrever a lógica de integração. Os modelos `deepseek-coder:6.7b` e `llama3:8b` foram escolhidos por combinarem, respectivamente, especialização em código e fluência textual, dentro de um porte compatível com o *hardware* de uma estação de trabalho de desenvolvedor; a discussão detalhada dessa escolha encontra-se na Seção 5.5.1.

4.3 FLUXO DE PROCESSAMENTO (DIAGRAMA DE SEQUÊNCIA)

O funcionamento do sistema segue um fluxo determinístico, garantindo que a documentação seja sempre baseada na realidade atual do código:

1. **Gatilho**: o desenvolvedor executa o comando “Clarity: Generate Documentation”.
2. **Extração**: a extensão identifica os arquivos modificados e envia o contexto para o Agente Analisador.
3. **Estruturação**: o Analisador processa o código e gera um JSON contendo:
    - Lista de módulos e suas responsabilidades.
    - Assinatura de funções, parâmetros e tipos de retorno.
    - Dependências internas e externas identificadas.
4. **Redação**: o Agente Redator consome o JSON e aplica técnicas de *prompt engineering* para transformar dados brutos em um arquivo Markdown estruturado.
5. **Persistência**: o sistema salva o resultado no arquivo README-CLARITY.md, na raiz do projeto, preservando integralmente o arquivo README.md original. Caso o comando seja executado novamente sobre o mesmo repositório, o arquivo README-CLARITY.md existente é sobrescrito pela nova geração — decisão de projeto que evita a proliferação de arquivos e mantém sempre uma única versão gerada disponível para revisão. O README.md principal nunca é modificado, independentemente do número de execuções.

Figura 4.1 – Diagrama de Sequência do Clarity



Fonte: Lucidchart (feito pelos autores).

4.4 JSON INTERMEDIATE

Um ponto central da modelagem é o esquema JSON que serve de ponte entre o Agente Analisador e o Agente Redator. Sua função é garantir que a geração do README seja fundamentada exclusivamente em dados estruturados extraídos do repositório, e não em inferências livres do modelo de linguagem. Ao restringir o Agente Redator a consumir apenas as informações contidas nesse esquema, mitiga-se o risco de alucinação — fenômeno em que o LLM gera conteúdo plausível, porém factualmente incorreto em relação ao código real.

O JSON é produzido pelo Agente Analisador ao final do processamento de cada *chunk* e consolidado em uma estrutura única antes de ser repassado ao Agente Redator. Essa estrutura contempla metadados globais do projeto e uma lista detalhada por arquivo, conforme descrito no Quadro 4.2.

Campo	**Tipo**	**Descrição**
projeto_nome, projeto_descricao, projeto_versao	*string*	Metadados gerais do projeto
projeto_scripts	objeto	*Scripts* disponíveis (dev, build, start, lint)
linguagens	*array*	Linguagens detectadas no projeto
quantidade_arquivos, total_linhas, total_funcoes, total_classes	*integer*	Métricas globais do projeto
detalhes	*array*	Lista com informações por arquivo (caminho, linguagem, linhas, funções, classes e *imports*)

Fonte: Autores (2026).

Quadro 4.2: Estrutura do JSON Intermediate

Essa separação entre extração estrutural e geração textual é o que viabiliza o uso de modelos locais de menor escala: o Agente Analisador não precisa produzir linguagem natural, apenas dados organizados; e o Agente Redator não precisa compreender código, apenas transformar um relatório já estruturado em documentação legível.

Três aspectos do contrato de dados merecem detalhamento, por serem os mecanismos que sustentam a fidelidade da documentação gerada. O primeiro é a distinção entre campos obrigatórios e opcionais. Os campos de métricas globais (`quantidade_arquivos`, `total_linhas`, `total_funcoes`, `total_classes`) e a lista `detalhes` são sempre preenchidos pela análise estática, pois derivam diretamente da varredura do repositório. Já campos como `projeto_descricao` podem ser nulos quando a informação não está disponível nos artefatos de configuração — si-

tuação em que o Agente Redator é instruído a inserir um marcador de revisão (*placeholder*) em vez de inferir livremente um conteúdo.

O segundo aspecto é a consolidação entre *chunks*. Como cada fragmento é analisado de forma independente, um mesmo projeto produz vários JSONs parciais. Esses resultados são fundidos em uma estrutura única antes da redação: as métricas globais são somadas, as listas de linguagens e dependências são unidas com eliminação de duplicatas, e as entradas por arquivo são concatenadas. Como cada arquivo pertence a um único *chunk*, não há conflito de versões para um mesmo arquivo; a deduplicação atua sobre metadados agregados (linguagens e dependências), e não sobre arquivos individuais. Vale registrar uma limitação decorrente dessa estratégia: por serem processados isoladamente, os *chunks* não compartilham contexto entre si, de modo que relações entre arquivos situados em fragmentos distintos podem não ser capturadas — uma troca deliberada entre viabilidade de execução local e completude da análise, retomada na Seção 5.3.2.

O terceiro aspecto é o tratamento de erros de *parsing* e o mecanismo anti-alucinação. Quando um arquivo não pode ser analisado — por conter sintaxe inválida ou construções não suportadas —, ele é registrado como não processado e omitido da estrutura, sem interromper o *pipeline*. Já a contenção de alucinações no Agente Redator não decorre de uma única regra, mas da própria natureza do contrato: o redator recebe instruções explícitas para produzir o README exclusivamente a partir dos dados presentes no JSON, sendo-lhe vedado introduzir bibliotecas, dependências ou funcionalidades que não constem da estrutura; quando um campo necessário está ausente, a instrução é inserir um *placeholder* de revisão. Os detalhes dessa engenharia de *prompt* são apresentados na Seção 5.3.4.

Figura 4.2 – Exemplo de Estrutura do JSON

STDOUT completo:


```
[DEBUG] JSON GERADO PELO ANALYST (Chunk 1/8):
{
  "projeto_nome": "siga-talonario-eletronico",
  "projeto_descricao": null,
  "projeto_versao": "0.1.0",
  "projeto_scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "eslint"
  },
  "linguagens": [
    "ts",
    "tsx"
  ],
  "quantidade_arquivos": 15,
  "total_linhas": 1807,
  "total_funcoes": 21,
  "total_classes": 0,
  "detalhes": [
    {
      "arquivo": "c:\\Users\\yagam\\Documents\\Trabalho\\STTP\\SIGA - Talonario Eletronico\\SIGA---Talonario-Eletronico\\next-env.d.ts",
      "linguagem": "ts",
      "linhas": 6,
      "funcoes": 0,
      "classes": 0,
      "imports_count": 0
    },
    {
      "arquivo": "c:\\Users\\yagam\\Documents\\Trabalho\\STTP\\SIGA - Talonario Eletronico\\SIGA---Talonario-Eletronico\\next.config.t
s",
      "linguagem": "ts",
      "linhas": 7,
      "funcoes": 0,
      "classes": 0,
      "imports_count": 1
    },
    {
      "arquivo": "c:\\Users\\yagam\\Documents\\Trabalho\\STTP\\SIGA - Talonario Eletronico\\SIGA---Talonario-Eletronico\\app\\layout.ts
x",
```


Fonte: Logs do Projeto.

5 IMPLEMENTAÇÃO E RESULTADOS

Este capítulo descreve a implementação do protótipo funcional do Clarity, apresenta as funcionalidades efetivamente desenvolvidas e expõe os resultados obtidos a partir da execução da ferramenta em repositórios reais. A organização segue o fluxo natural do sistema: parte-se de uma visão geral do protótipo, detalha-se a implementação da extensão para o Visual Studio Code e do *pipeline* em Python, discute-se a publicação da ferramenta na loja oficial do editor e, por fim, apresentam-se as observações decorrentes dos estudos de caso conduzidos.

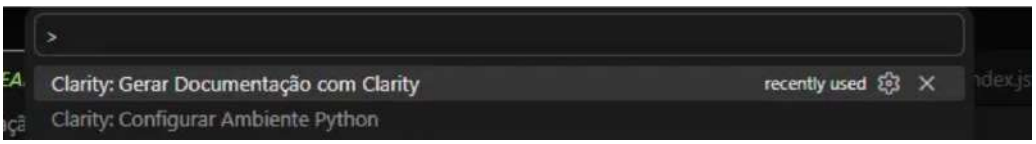
5.1 VISÃO GERAL DO PROTÓTIPO

O protótipo do Clarity foi desenvolvido como uma extensão funcional do Visual Studio Code, denominada `fabsms-clarity`, publicada na loja oficial do editor (Visual Studio Code Marketplace) e disponibilizada como software de código aberto sob licença MIT no repositório público mantido pelos autores. A versão correspondente a este trabalho é a 0.0.5 (Beta), implementada conforme o modelo arquitetural descrito no Capítulo 4.

A solução é composta por dois subsistemas integrados. O primeiro é a extensão propriamente dita, escrita em TypeScript sobre o ambiente Node.js, responsável pela interação com o desenvolvedor por meio do painel de comandos do Visual Studio Code e pela manipulação dos arquivos do repositório aberto no editor. O segundo é o *pipeline* de inteligência artificial, escrito em Python, que executa as etapas de análise estática do código-fonte, divisão em fragmentos (*chunking*), consolidação de metadados, orquestração dos agentes via CrewAI e comunicação com os modelos de linguagem locais por meio do serviço Ollama. A comunicação entre os subsistemas ocorre via invocação de processo Python pela extensão, com troca de dados em formato JSON.

A extensão expõe ao desenvolvedor dois comandos principais, acessíveis a partir do painel de comandos do editor. O primeiro é o *Clarity: Configurar Ambiente Python*, responsável pela preparação inicial do ambiente local, executado uma única vez antes do primeiro uso da ferramenta. O segundo é o *Clarity: Gerar Documentação com Clarity*, responsável pelo acionamento efetivo do *pipeline* de análise e geração da documentação sobre o repositório aberto no editor.

Figura 5.1 – Comandos do Clarity expostos no painel do Visual Studio Code



Fonte: Autores (2026).

Por se tratar de uma ferramenta que depende de componentes externos ao editor — notadamente o interpretador Python e o serviço Ollama com seus respectivos modelos — o protótipo incorpora um mecanismo de verificação e configuração de ambiente, acionado pelo comando *Clarity: Configurar Ambiente Python*. Essa abordagem foi adotada para reduzir o atrito de adoção da ferramenta, eliminando a necessidade de o desenvolvedor consultar documentação externa para preparar manualmente seu ambiente local.

A verificação contempla três dependências críticas:

1. **Interpretador Python (3.11 ou superior)**: caso a extensão não detecte uma instalação válida, é exibida uma notificação contendo um botão que redireciona o usuário diretamente para a página oficial de *download* da linguagem.

2. **Serviço Ollama em execução local**: caso este não esteja instalado, a extensão emite uma notificação dedicada com um botão para a página oficial de instalação.

3. **Disponibilidade local dos modelos**: os dois modelos utilizados pelo *pipeline* — `deepseek-coder:6` empregado pelo Agente Analisador, e `llama3:8b`, empregado pelo Agente Redator. Caso o Ollama esteja instalado, mas algum dos modelos exigidos não esteja disponível, a extensão apresenta ao usuário as instruções específicas para o *download* via Ollama, incluindo os comandos exatos a serem executados.

Uma vez satisfeitos os pré-requisitos, a operação do protótipo segue o fluxo determinístico apresentado na Figura 4.1: o desenvolvedor aciona o comando *Clarity: Gerar Documentação com Clarity* a partir do editor, o *pipeline* percorre o repositório aberto, extrai metadados estruturais, processa o conteúdo em fragmentos, gera um relatório técnico intermediário em formato JSON e, ao final, produz o arquivo README-CLARITY.md na raiz do projeto. Em nenhuma etapa o conteúdo do código-fonte é transmitido a servidores externos, uma vez que toda a inferência ocorre localmente na máquina do desenvolvedor.

O Quadro 5.1 sintetiza o estado atual do protótipo.

5.2 IMPLEMENTAÇÃO DA EXTENSÃO PARA VISUAL STUDIO CODE

A extensão para o Visual Studio Code foi desenvolvida em TypeScript, utilizando a API nativa de extensões do editor e o ambiente de execução Node.js. Sua função é atuar como camada de interação entre o desenvolvedor e o *pipeline* de inteligência artificial, traduzindo comandos emitidos pela interface do editor em invocações ao *backend* escrito em Python e exibindo o progresso da execução ao usuário.

A extensão registra dois comandos no painel de comandos do Visual Studio Code, acessíveis por meio do atalho habitual (Ctrl+Shift+P, no Windows e Linux, ou Cmd+Shift+P, no macOS) e identificados pelo prefixo *Clarity*. O primeiro deles, *Clarity: Configurar Ambiente Python*, é destinado à preparação inicial do ambiente local, conforme descrito na Seção 5.1, e executado

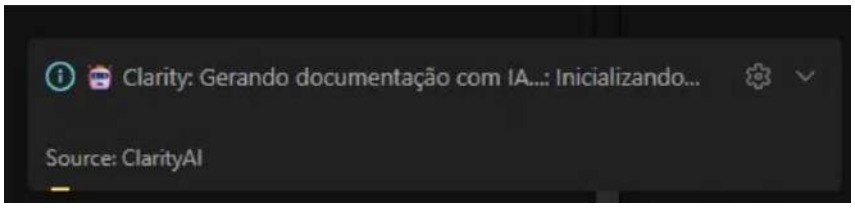
Atributo	**Valor**
Versão publicada	0.0.5 (Beta)
Identificador no Marketplace	FabsMS.fabsms-clarity
Linguagens suportadas	Python, JavaScript, TypeScript, JSX, TSX, Java
Versão mínima do Python	3.11
Modelo Analisador	deepseek-coder:6.7b (temperatura 0.1)
Modelo Redator	llama3:8b (temperatura 0.4)
Tamanho máximo de *chunk*	15 arquivos ou 5.000 linhas
Comandos disponíveis	Clarity: Configurar Ambiente Python; Clarity: Gerar Documentação com Clarity
Saída gerada	README-CLARITY.md
Distribuição da base de código	88,9% Python; 10,0% TypeScript; 1,1% outros
Licença	MIT

Fonte: Autores (2026).

Quadro 5.1: Estado atual do protótipo Clarity

uma única vez antes do primeiro uso da ferramenta, ou após alguma atualização da extensão. O segundo, *Clarity: Gerar Documentação com Clarity*, é o comando operacional principal: ao ser acionado, obtém o caminho absoluto do diretório atualmente aberto no editor, valida a presença do ambiente Python configurado e instancia o processo responsável pela execução do *pipeline*. Durante o processamento, mensagens de progresso são transmitidas do *backend* para a extensão por meio da saída padrão (*stdout*) e exibidas ao desenvolvedor por meio do painel de notificações nativo do editor, permitindo o acompanhamento das fases de coleta, análise, consolidação e redação descritas na Seção 4.3.

Figura 5.2 – Notificação de progresso exibida pela extensão



Fonte: Autores (2026).

A escolha do Visual Studio Code como ambiente de execução fundamenta-se em três fatores. O primeiro é a sua predominância no mercado: segundo a Stack Overflow Developer Survey 2024, realizada com mais de 65.000 desenvolvedores em 185 países, o Visual Studio Code é utilizado por 74% dos respondentes, mais que o dobro do segundo colocado, Visual Studio, com 29% (Stack Overflow, 2024). Trata-se, portanto, do editor de código mais adotado mundialmente entre desenvolvedores profissionais e estudantes. O segundo é a maturidade da sua API de extensões, que permite acesso programático ao sistema de arquivos do projeto aberto, ao painel de comandos, ao terminal integrado e aos componentes visuais nativos do editor. O

terceiro é a existência de um canal oficial de distribuição — o Visual Studio Code Marketplace — que viabiliza a publicação da ferramenta para um amplo público de desenvolvedores sem custos adicionais e com mecanismos integrados de versionamento e atualização automática.

A preservação do README.md original do projeto, descrita como requisito na Seção 1.4, é garantida pela própria extensão: ao final do *pipeline*, o arquivo gerado é gravado com o nome README-CLARITY.md na raiz do diretório aberto, sem qualquer interação com o README.md preexistente. Cabe exclusivamente ao desenvolvedor decidir, em uma etapa posterior de revisão, se o conteúdo gerado deve ser incorporado ao README principal do repositório, integrado parcialmente ou descartado. Essa decisão de projeto elimina o risco de sobrescrita acidental — comportamento observado em ferramentas concorrentes, conforme discutido no Quadro 3.1.

5.3 IMPLEMENTAÇÃO DO PIPELINE EM PYTHON

O *pipeline* de inteligência artificial do Clarity foi implementado em Python, escolha justificada pela maturidade do ecossistema da linguagem para tarefas envolvendo modelos de linguagem natural, análise sintática de código-fonte e orquestração de agentes autônomos. O *backend* está organizado em módulos especializados, segregados por responsabilidade: extração e *parsing* dos arquivos do repositório, divisão do conteúdo em fragmentos processáveis, definição e instanciação dos agentes do CrewAI, integração com o serviço Ollama para execução local dos modelos, e composição do arquivo final de saída. Esta seção descreve cada uma dessas etapas, conforme efetivamente implementadas no protótipo.

5.3.1 Análise Estática e Extração de Metadados

A primeira fase do *pipeline* consiste no varrimento completo do repositório aberto no editor, percorrendo recursivamente sua estrutura de diretórios para identificar todos os arquivos com extensões compatíveis com as linguagens suportadas. Arquivos cujas extensões não correspondem a Python (.py), JavaScript (.js), TypeScript (.ts), JSX (.jsx), TSX (.tsx) ou Java (.java) são desconsiderados na etapa de análise, bem como diretórios convencionalmente reservados para artefatos de *build* e dependências, como `node_modules`, `.git`, `__pycache__`, `dist` e `build`.

Para cada arquivo identificado, aplica-se uma estratégia de extração específica conforme a linguagem. Em arquivos Python, utiliza-se o módulo `ast` da biblioteca padrão da linguagem, que produz uma Árvore Sintática Abstrata completa do código a partir da qual são extraídos com precisão os nomes de funções, classes, parâmetros, decoradores, *docstrings* e instruções de importação. Já para JavaScript, TypeScript e Java, dado o caráter heterogêneo desses ecossistemas e a maior complexidade envolvida na adoção de *parsers* nativos por linguagem, adota-se uma abordagem baseada em reconhecimento de padrões textuais sobre o conteúdo dos arquivos, com expressões regulares específicas para identificar declarações de funções, classes, importações e

exportações. Para todos os arquivos, independentemente da linguagem, são também coletadas métricas básicas como quantidade total de linhas e número de instruções de importação.

Adicionalmente à análise por arquivo, o *pipeline* realiza a leitura de artefatos de configuração de projeto presentes na raiz do repositório, dos quais são extraídos os metadados globais. O arquivo `package.json` — característico de projetos JavaScript, TypeScript e seus *frameworks* — fornece o nome do projeto, sua versão, descrição, *scripts* de execução e dependências declaradas. Arquivos como `requirements.txt`, `pyproject.toml`, `pom.xml` e `build.gradle` fornecem informações equivalentes para projetos Python e Java. Esses metadados são posteriormente utilizados pelo Agente Redator para compor seções específicas do README, como instruções de instalação, execução e listagem de dependências.

5.3.2 Estratégia de Chunking

Modelos de linguagem locais, como os adotados neste trabalho, operam sob restrições mais severas de janela de contexto em comparação com modelos de grande escala disponibilizados por meio de APIs comerciais. Para que a ferramenta processe repositórios de qualquer porte sem incorrer em estouro de contexto, implementou-se uma estratégia de divisão do conteúdo extraído em fragmentos menores, denominados *chunks*.

No protótipo atual, cada *chunk* respeita simultaneamente dois limites: no máximo 15 arquivos e no máximo 5.000 linhas de código somadas. Quando qualquer um dos dois limites é atingido, o *chunk* corrente é encerrado e um novo *chunk* é iniciado. Essa heurística foi calibrada empiricamente durante o desenvolvimento, de forma a manter cada fragmento suficientemente abaixo da janela de contexto efetiva do modelo `deepseek-coder:6.7b`, mesmo considerando o *overhead* introduzido pelas instruções dos *prompts* dos agentes.

Cada *chunk* é processado de forma independente pelo Agente Analisador, que produz para ele uma análise intermediária em formato JSON. Ao final, as análises individuais são consolidadas em uma estrutura única, com deduplicação de entradas redundantes, antes de serem repassadas ao Agente Redator. Essa abordagem garante que repositórios extensos sejam decompostos em unidades de trabalho cognitivamente tratáveis pelos modelos locais, preservando a integridade da análise global.

É importante reconhecer, contudo, que o processamento independente dos *chunks* implica um risco de perda de contexto entre fragmentos: como cada *chunk* é analisado isoladamente, relações estruturais entre arquivos alocados em fragmentos distintos — por exemplo, uma função definida em um *chunk* e invocada em outro — não são capturadas pela análise. Essa limitação é mitigada, mas não eliminada, por dois fatores. Primeiro, a análise do Clarity opera predominantemente em nível de metadados estruturais (linguagens, dependências, contagem de funções e classes, organização de diretórios), e não em nível de fluxo de execução entre arquivos, dimensão em que a fragmentação teria maior impacto. Segundo, os metadados globais do projeto — extraídos de artefatos como o `package.json`, que independem da divisão em *chunks* — são coletados de forma unificada, preservando a visão de conjunto do repositório. Ainda assim,

para projetos em que as relações interarquivos sejam essenciais à documentação, a estratégia atual constitui uma limitação reconhecida, cuja superação — por meio de análise com contexto compartilhado entre *chunks* — é apontada como trabalho futuro.

5.3.3 Orquestração de Agentes com CrewAI

A orquestração dos agentes é implementada por meio do *framework* CrewAI, descrito na Seção 2.7. Foram definidos dois agentes com papéis especializados e modelos distintos, ambos servidos localmente pelo Ollama.

O Agente Analisador, denominado *Analyst LLM*, utiliza o modelo `deepseek-coder:6.7b`, especificamente treinado para tarefas envolvendo compreensão e geração de código-fonte. Sua função é processar cada *chunk* produzido na fase anterior e gerar um relatório estruturado em JSON contendo as informações técnicas extraídas: lista de módulos com suas respectivas responsabilidades inferidas, assinatura de funções com parâmetros e tipos de retorno, dependências internas e externas, e identificação de *frameworks* e padrões arquiteturais reconhecíveis a partir da estrutura do projeto. Esse agente opera com temperatura 0.1, valor próximo do mínimo da escala, configuração que reduz a aleatoriedade da geração e privilegia a precisão e o determinismo nas respostas — característica essencial para uma etapa cujo objetivo é a fidelidade aos dados do código real, e não a fluência textual.

O Agente Redator, denominado *Writer LLM*, utiliza o modelo `llama3:8b`, especializado em compreensão e geração de linguagem natural. Sua função é receber o relatório técnico consolidado produzido pelo *Analyst* e transformá-lo no arquivo README-CLARITY.md final, em formato Markdown, organizado segundo as seções convencionais de um README de projeto: descrição, funcionalidades, requisitos, instalação, execução, estrutura do projeto, tecnologias utilizadas e licença. Esse agente opera com temperatura 0.4, valor intermediário que permite ao modelo produzir texto com fluência adequada à documentação técnica sem comprometer a fidelidade aos dados fornecidos pelo agente anterior.

A divisão de responsabilidades entre dois agentes com modelos distintos materializa, no plano da implementação, a abordagem de orquestração de agentes de LLM discutida na Seção 2.6: ao invés de uma única instância de modelo resolver simultaneamente os problemas de compreensão de código e de produção de texto, cada agente opera dentro de suas competências específicas. Cabe reiterar que essa interação não é dialógica nem negociada — como ocorreria em um Sistema Multiagente no sentido clássico —, mas sequencial e determinística: a saída estruturada do *Analyst* é repassada ao *Writer* como entrada controlada, e é justamente esse encadeamento restrito que produz o artefato final e limita o espaço de geração livre do redator.

5.3.4 Engenharia de Prompt para LLMs Locais

A migração do *pipeline* original — concebido para utilizar a API do Google Gemini — para um ambiente baseado exclusivamente em modelos locais via Ollama trouxe consigo desa-

fios significativos no plano da engenharia de *prompt*. Embora os modelos locais selecionados ofereçam capacidade técnica suficiente para as tarefas propostas, sua menor escala de parâmetros, em comparação com modelos comerciais de grande porte, traduz-se em maior propensão a alucinações — fenômeno em que o modelo produz informações plausíveis em forma, porém incorretas em relação ao contexto fornecido — e em menor capacidade de inferir implicitamente o formato esperado da saída a partir de instruções concisas.

A primeira versão do *pipeline*, calibrada para o Gemini, operava com *prompts* relativamente sucintos, uma vez que o modelo comercial demonstrava alta capacidade de inferir formato, tom e estrutura a partir de instruções pouco detalhadas. Ao serem reaproveitados sem adaptação sobre os modelos locais, esses mesmos *prompts* produziram resultados marcadamente inferiores: o Agente Analisador identificava dependências e *frameworks* inexistentes no projeto, e o Agente Redator gerava textos genéricos, com afirmações desconectadas da realidade do código analisado.

A resposta a esse desafio consistiu em quatro frentes complementares de ajuste, conduzidas iterativamente ao longo do desenvolvimento.

A primeira frente foi o **detalhamento e padronização estrutural dos *prompts***. Os *prompts* originais, sucintos, foram substituídos por instruções extensas que especificam de forma explícita o formato esperado da saída, a estrutura completa do README a ser produzido — com a definição prévia e nominal de cada seção que o documento final deve conter — e o estilo de redação esperado. Essa padronização forneceu ao modelo um esqueleto explícito a ser preenchido, em substituição à expectativa, frustrada nos modelos locais, de que o modelo deduzisse a estrutura adequada apenas a partir do objetivo geral.

A segunda frente foi a **introdução de restrições explícitas no *prompt***, especificando ao modelo o que não fazer. Foram incorporadas instruções diretas vedando a invenção de bibliotecas, dependências ou *frameworks* não presentes no JSON intermediário; vedando a inferência de funcionalidades não evidenciadas pelos metadados extraídos; e exigindo que toda afirmação sobre tecnologias utilizadas tivesse correspondência direta com os dados do relatório técnico. Essas restrições reduziram substancialmente a ocorrência de alucinações observadas nas versões iniciais.

A terceira frente foi o **ajuste empírico da temperatura dos modelos**. A temperatura é o hiperparâmetro que controla o grau de aleatoriedade na seleção do próximo *token* gerado: valores baixos privilegiam o *token* mais provável segundo a distribuição aprendida, enquanto valores altos permitem maior variação. Para o Agente Analisador, a temperatura foi progressivamente reduzida de 0.9, valor padrão de muitas implementações, para 0.5 e, finalmente, para 0.1, à medida que se observou que valores elevados levavam o modelo a inferir elementos inexistentes no código. Para o Agente Redator, o trajeto foi não linear: partindo de 0.9, com alucinações severas, passou-se por 0.7, com alucinações ainda significativas, e por 0.1, valor no qual o modelo produzia texto excessivamente seco, mecânico e desprovido da fluência esperada para um README de qualidade. Após sucessivas iterações, o valor 0.4 foi adotado como ponto de

equilíbrio entre fidelidade aos dados e qualidade textual, permitindo ao redator preencher lacunas naturais da redação com inferências moderadas, sem comprometer a aderência ao relatório técnico.

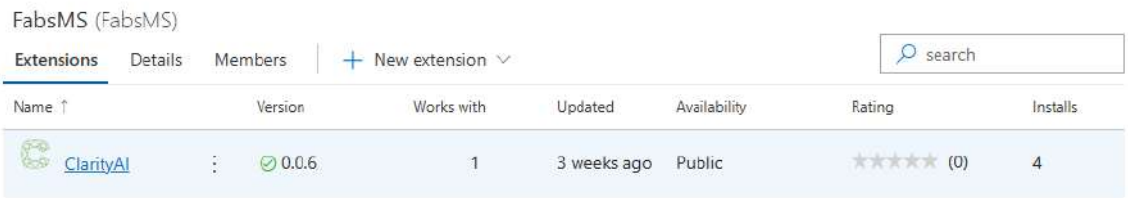
A quarta frente, conduzida em paralelo, foi o **enriquecimento da estrutura do JSON intermediário**. Reconhecendo que parte das alucinações observadas no Agente Redator decorria da ausência de informação suficiente no relatório técnico — situação que efetivamente forçava o modelo a preencher lacunas por inferência —, ampliou-se progressivamente o conjunto de campos extraídos pelo *Analyst*. Quanto maior o detalhamento e a granularidade do JSON repassado ao redator, menor a margem residual para criação de conteúdo não fundamentado, e mais assertiva torna-se a documentação final em relação ao código real.

O conjunto dessas adaptações ilustra um princípio prático relevante para o uso de LLMs locais em tarefas de engenharia de software: a menor capacidade do modelo, em relação a alternativas comerciais de grande escala, deve ser compensada por maior estruturação e explicitação do contexto e das restrições fornecidas. A redução observada na taxa de alucinação ao longo do desenvolvimento, descrita qualitativamente, sugere que essa abordagem é viável para o caso de uso proposto.

5.4 PUBLICAÇÃO NO VISUAL STUDIO CODE MARKETPLACE

Como parte da estratégia de distribuição e de validação externa da ferramenta, o protótipo foi empacotado e publicado na loja oficial de extensões do Visual Studio Code — o Visual Studio Code Marketplace — sob o identificador `FabsMS.fabsms-clarity`. O empacotamento da extensão é realizado por meio da ferramenta oficial `vsce` (Visual Studio Code Extensions), que produz o arquivo no formato VSIX a partir do código-fonte e dos metadados declarados no projeto. O envio do pacote para a loja, na presente etapa, é conduzido manualmente por meio do portal oficial do Marketplace, em substituição a um fluxo automatizado de publicação, opção que se mostrou adequada à cadência atual de lançamentos do projeto. A Figura 5.3 apresenta a extensão publicada no portal de gerenciamento do Marketplace.

Figura 5.3 – Extensão ClarityAI publicada no Visual Studio Code Marketplace



Fonte: Autores (2026).

A disponibilização pública da ferramenta atende a duas finalidades complementares. A primeira, de natureza prática, é eliminar o atrito de instalação para usuários que venham a colaborar com a avaliação do sistema: em vez de exigir a clonagem do repositório, a compilação local

da extensão e sua instalação manual via arquivo VSIX, basta ao desenvolvedor pesquisar pelo nome da extensão no painel de extensões do próprio editor, ClarityAI, e instalá-la com um único clique. A segunda, de natureza metodológica, é viabilizar o acompanhamento da adoção da ferramenta por meio dos canais nativos do Marketplace, que registram automaticamente métricas como número de instalações, avaliações e comentários públicos deixados pelos usuários.

A publicação seguiu as práticas convencionais do ecossistema de extensões do Visual Studio Code. O arquivo `package.json` da extensão declara metadados essenciais como nome, versão, descrição, comandos expostos ao painel do editor e classificações que orientam a descoberta da ferramenta na loja. A extensão foi categorizada sob *Programming Languages* e indexada por meio das seguintes palavras-chave: *documentation*, *readme*, *ai*, *ollama*, *python*, *code analysis*, *generator* e *offline*. Essa combinação reflete os atributos centrais do projeto, abrangendo simultaneamente o propósito da ferramenta (geração de documentação e de README), a abordagem técnica adotada (inteligência artificial, análise de código, modelos via Ollama) e uma característica diferencial em relação a soluções concorrentes baseadas em APIs externas (execução *offline*).

Complementarmente aos metadados, o arquivo README.md da raiz do repositório é automaticamente exibido como página de apresentação da extensão na loja, contendo descrição das funcionalidades, requisitos de ambiente, instruções de uso e informações sobre o vínculo do projeto com este Trabalho de Conclusão de Curso. O arquivo CHANGELOG.md registra o histórico de versões publicadas, permitindo aos usuários acompanhar as evoluções e correções entre lançamentos. O repositório no GitHub é referenciado nos metadados da extensão, oferecendo ao público interessado acesso direto ao código-fonte e ao sistema de *issues* para reporte de problemas.

A versão utilizada nos experimentos deste trabalho é a 0.0.5 (Beta), distribuída sob licença MIT; foi sobre ela que se realizaram todos os estudos de caso e a avaliação relatados nos capítulos seguintes. Uma versão posterior, 0.0.6, foi publicada após a conclusão dos testes, contendo exclusivamente ajustes na página de apresentação da extensão na loja (descrição e textos de *marketing*), sem qualquer alteração no comportamento do *pipeline* de geração; é essa a versão exibida na Figura 5.3. A escolha por uma licença permissiva tem por objetivo maximizar a possibilidade de adoção da ferramenta em contextos diversos — incluindo ambientes corporativos — sem impor restrições que possam desestimular o uso. A continuidade da publicação após a conclusão deste trabalho permitirá que a ferramenta alcance um público para além do escopo institucional inicial, contribuindo para a validação externa da abordagem proposta.

5.5 RESULTADOS

Esta seção apresenta os resultados obtidos a partir da execução do protótipo do Clarity em repositórios reais, conforme previsto na metodologia descrita na Seção 1.6. Os estudos de caso aqui descritos oferecem uma observação qualitativa do comportamento da ferramenta em cenários de uso autênticos, que fundamenta a avaliação formal por rubrica apresentada no Capítulo 6.

As execuções foram realizadas após a estabilização do protótipo na versão 0.0.5, com os ajustes de engenharia de *prompt* e calibração de temperatura descritos na Seção 5.3.4 já consolidados.

5.5.1 Ambiente de Execução

Os testes foram conduzidos em estação de trabalho com a seguinte configuração de *hardware*: processador AMD Ryzen 7 5700, memória RAM de 32 GB DDR4 operando a 3200 MHz, placa de vídeo NVIDIA GeForce RTX 4060 com 8 GB de memória dedicada, e placa-mãe MSI MPG B550 Gaming Plus AM4. A escolha de uma estação com aceleração gráfica dedicada justifica-se pela natureza dos modelos utilizados: tanto o `deepseek-coder:6.7b` quanto o `llama3:8b` beneficiam-se substancialmente da execução em GPU, com redução significativa do tempo de inferência em comparação com execução exclusivamente em CPU. A configuração descrita é compatível com o requisito de *hardware* típico de uma estação de trabalho de desenvolvedor de nível intermediário, o que reforça a viabilidade prática da abordagem proposta.

O serviço Ollama foi executado em sua configuração padrão, com os modelos previamente carregados em memória local. O protótipo foi acionado a partir do próprio Visual Studio Code, por meio do comando *Clarity: Gerar Documentação com Clarity*, e os repositórios analisados foram abertos como diretório de trabalho do editor no momento da execução.

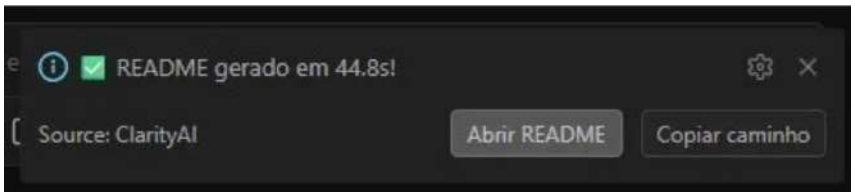
5.5.2 Repositórios Avaliados

Os três repositórios selecionados para esta avaliação pertencem ao acervo de sistemas internos da Superintendência de Trânsito e Transportes Públicos de Campina Grande (STTP), empresa onde um dos autores atuava como estagiário. O uso desses repositórios para os fins acadêmicos deste trabalho foi autorizado informalmente pela equipe responsável da STTP. Como medida de resguardo, a análise restringiu-se à estrutura e aos metadados do código — nomes de arquivos, dependências e organização de diretórios —, sem exposição de dados sensíveis, credenciais ou informações de cidadãos eventualmente processadas por esses sistemas; os arquivos README gerados, reproduzidos nos apêndices, contêm apenas informações de natureza técnica. A escolha desses projetos atende ao requisito metodológico de utilizar repositórios reais — em desenvolvimento ou produção — em substituição a códigos sintéticos preparados especificamente para o teste da ferramenta. Os três projetos pertencem ao ecossistema *front-end* da STTP e abrangem variações relevantes para a avaliação da ferramenta: dois deles utilizam JavaScript com React e Vite, enquanto o terceiro emprega TypeScript com Next.js e React, permitindo observar o comportamento do Clarity sobre *stacks* distintas dentro do mesmo domínio.

O Quadro 5.2 resume as características dos três repositórios analisados e os resultados quantitativos coletados em cada execução.

Os valores aproximados decorrem da ausência, no protótipo atual, de um canal dedicado de *output* persistente no Visual Studio Code que registre as métricas finais de execução. A inclusão desse canal está prevista para as próximas iterações da ferramenta, com a finalidade de viabilizar a coleta sistemática dessas métricas. Para o repositório SIGA Talonário Eletrônico, os valores

Figura 5.4 – Notificação de conclusão exibida pela extensão após a geração do README



Fonte: Autores (2026).

Atributo	**Nutri-STTP**	**Transitolândia Admin**	**SIGA Talonário Eletrônico (em desenvolvimento)**
Domínio	Marcação de consultas de nutrição	Painel administrativo de jogos educativos	Gestão de talonário eletrônico de infrações
Linguagem principal	JavaScript	JavaScript	TypeScript
Framework principal	React + Vite	React + Vite	Next.js + React
Arquivos analisados (aprox.)	50	22	15
Total de linhas	—	—	1.807
Chunks gerados (aprox.)	4	2	8
Tempo total de execução	66,9 s	50,0 s	44,8 s

Fonte: Autores (2026).

Quadro 5.2: Repositórios avaliados e resultados de execução

apresentados são exatos, uma vez que correspondem aos dados registrados na Figura 4.2 do Capítulo 4, extraídos diretamente do *stdout* do *pipeline* durante uma execução de depuração.

Uma observação decorre da comparação entre os projetos, mas deve ser lida com cautela em razão do caráter aproximado das métricas de contagem de *chunks* e de tempo. Nota-se que o repositório com o maior tempo de execução (Nutri-STTP, 66,9 s) não foi o que gerou mais *chunks*, enquanto o SIGA, com o menor tempo (44,8 s), registrou o maior número de fragmentos. Esse descompasso indica que o número de *chunks*, isoladamente, não é um bom preditor do tempo de execução: o custo de inferência depende mais do volume total de conteúdo efetivamente processado — em linhas de código e em *tokens* — do que da quantidade de fragmentos em que ele é dividido. A confirmação rigorosa dessa hipótese exigiria a medição sistemática de linhas de código (LoC) por repositório, informação disponível de forma exata apenas para o SIGA (1.807 linhas) na presente versão do protótipo. A ausência de um canal dedicado de *output* persistente no Visual Studio Code, que registre LoC, número de *chunks* e tempo de forma confiável a cada execução, é assumida como uma limitação deste trabalho, e sua inclusão está

prevista como trabalho futuro (Seção 7.3). Ainda assim, a decisão de projeto de combinar dois limites na estratégia de *chunking* — por arquivos e por linhas (Seção 5.3.2) — permanece justificada, por acomodar tanto repositórios com muitos arquivos pequenos quanto poucos arquivos densos, característicos de projetos TypeScript com TSX.

5.5.3 Análise Qualitativa

A análise dos arquivos README-CLARITY.md gerados nos três repositórios revela um padrão consistente de qualidade que confirma as hipóteses orientadoras do projeto e expõe, também, as limitações esperadas por se tratar de um protótipo.

Pontos fortes observados. Em todos os três projetos, o Clarity identificou corretamente a linguagem principal, os *frameworks* utilizados (React e Vite para os projetos Nutri-STTP e Transitolândia, Next.js para o SIGA), e o conjunto completo de dependências declaradas nos respectivos arquivos `package.json`. A listagem das dependências inclui pacotes como Material-UI, Emotion, Axios, Tailwind, Bootstrap, React-Bootstrap, JWT-Decode e ESLint, todos efetivamente presentes nos projetos analisados, sem invenção de bibliotecas inexistentes — um avanço direto em relação ao comportamento observado nas versões iniciais do *pipeline* com *prompts* não adaptados, conforme discutido na Seção 5.3.4. Os *scripts* de execução (`npm run dev`, `npm run build`, `npm run preview`) foram extraídos com precisão a partir dos metadados do projeto, assim como os requisitos de ambiente (Node.js 18+, npm 9+). A estrutura de diretórios foi reconstruída de forma legível e fiel à organização real dos repositórios, com agrupamento adequado de arquivos por componente.

Pontos que confirmam as limitações esperadas. As seções Sobre e Funcionalidades, que dependem essencialmente de inferência semântica a partir dos nomes de pastas, arquivos e símbolos extraídos, mostraram-se as mais sensíveis à qualidade da geração. Em todos os três projetos, o conteúdo gerado para essas seções é coerente com a realidade do código — o Clarity acerta ao descrever, por exemplo, o Nutri-STTP como sistema de gestão de consultas e cadastro de pacientes, o Transitolândia Admin como plataforma de gerenciamento de perguntas e jogos de memória, e o SIGA como sistema de gestão de talonário eletrônico de infrações. No entanto, trata-se de inferência: o modelo deduz a finalidade do sistema a partir de evidências indiretas, como os nomes dos diretórios `Anamnese`, `ConsultasMarcadas` e `PerfilNutricionista` no Nutri-STTP, ou `AddQuestionQuiz`, `JogoMemoriaService` e `AddQuestionSimulado` no Transitolândia. Quando essas evidências são suficientemente fortes e convergentes, como nos três casos analisados, a inferência produz texto adequado, porém não muito criativo nem muito descritivo em relação a tudo que os sistemas realmente podem oferecer; em projetos com convenções de nomenclatura menos descritivas ou com domínio menos óbvio, espera-se maior variabilidade.

Variabilidade entre execuções. A comparação dos três arquivos gerados também evidenciou um aspecto interessante: a estabilidade da geração varia entre execuções, mesmo dentro da mesma *stack*. No Nutri-STTP, a *tagline* introdutória foi mantida como marcador [TAGLINE],

sinalizando ao desenvolvedor que aquele campo precisa de revisão manual; já no Transitolândia, o modelo produziu uma *tagline* efetiva ("Desenvolvendo memória e conhecimento para a Transitolândia"), demonstrando capacidade de gerar texto criativo quando há informação suficiente. Comportamento análogo ocorreu na seção "Descrição" das pastas: no Nutri-STTP e no SIGA permaneceu parcialmente como *placeholder*, enquanto no Transitolândia foi completamente preenchida com descrições coerentes para cada diretório (Notification, PrivateRoute, Sidebar, Topbar). Essa variabilidade é consistente com a temperatura 0.4 adotada para o Agente Redator, valor que privilegia o equilíbrio entre fidelidade e fluência em detrimento do determinismo absoluto.

Mecanismos de controle e revisão. Em todos os três casos, o arquivo gerado preserva integralmente os mecanismos de controle previstos na Seção 1.4. O cabeçalho de cada README-CLARITY.md contém aviso explícito de que se trata de conteúdo gerado automaticamente, com instrução para revisão antes da publicação. Campos cujo preenchimento depende exclusivamente do conhecimento do desenvolvedor — como variáveis de ambiente e URL do repositório — são deixados como marcadores TODO. O arquivo README.md original de cada projeto foi preservado em todos os casos, em conformidade com o requisito de gerar o conteúdo em arquivo separado.

Tempo de execução. Os tempos observados — entre 44,8 e 66,9 segundos para projetos de pequeno a médio porte — situam-se em patamar compatível com o uso integrado ao fluxo de desenvolvimento. A título de comparação, esse intervalo é da ordem de grandeza do tempo gasto por um desenvolvedor para iniciar manualmente a redação do primeiro parágrafo de um README; sob essa perspectiva, o ganho de produtividade é significativo, mesmo considerando o tempo adicional necessário para revisão e ajuste do conteúdo gerado.

A análise apresentada nesta seção é qualitativa e descritiva. Ela é complementada pela avaliação formal por rubrica (completude, atualidade, clareza, utilidade e consistência), conduzida de forma comparativa no Capítulo 6.

6 AVALIAÇÃO

Este capítulo apresenta a avaliação formal da documentação gerada pelo Clarity, complementando a análise qualitativa preliminar (Seção 5.5.3) com um instrumento de pontuação estruturado. A avaliação por rubrica operacionaliza os cinco critérios definidos na metodologia (Seção 1.6) — completude, atualidade, clareza, utilidade e consistência —, transformando o julgamento qualitativo em uma medida comparável e estabelecendo um paralelo direto com a ferramenta de mercado de referência *README-AI*.

6.1 INSTRUMENTO DE AVALIAÇÃO

A rubrica adota uma escala de cinco níveis (1 a 5), em que 1 corresponde a *Insuficiente*, 2 a *Limitado*, 3 a *Regular*, 4 a *Bom* e 5 a *Excelente*. Cada critério recebe um descritor objetivo por nível, de modo a reduzir a subjetividade da pontuação. Buscando reduzir o viés de uma avaliação conduzida pelos próprios autores — que são parte interessada no resultado —, a aplicação da rubrica foi delegada a um agente de inteligência artificial: os critérios e as valências de cada nível foram fornecidos a uma única instância do Claude Code, da Anthropic, em sua versão Max, que analisou os arquivos gerados por ambas as ferramentas e atribuiu, de forma autônoma, a nota de cada critério para cada documento.

Cabe, no entanto, uma ressalva metodológica explícita: delegar a avaliação a um sistema de IA não elimina o viés, apenas substitui o avaliador humano por outro avaliador automatizado, que possui seus próprios vieses e limitações. Um LLM pode, por exemplo, favorecer sistematicamente determinado estilo de redação, ser sensível à ordem de apresentação dos documentos ou apresentar variabilidade entre execuções. Além disso, por ter sido utilizada uma única instância avaliadora, não há medida de concordância entre avaliadores independentes. Por essas razões, as notas reportadas na Seção 6.2 devem ser interpretadas como indicativas, e não como um veredito definitivo de qualidade; essa questão é retomada como ameaça à validade no Capítulo 7, e a avaliação com múltiplos avaliadores humanos é apontada como trabalho futuro. Para permitir a verificação independente das notas, as saídas completas de ambas as ferramentas são disponibilizadas publicamente, conforme indicado na Seção 6.2. O Quadro 6.1 apresenta o instrumento completo.

6.2 RESULTADOS DA AVALIAÇÃO COMPARATIVA

A rubrica do Quadro 6.1 foi aplicada de forma comparativa aos três arquivos `README-CLARITY.md` gerados pelo Clarity (Apêndices A a C) e às saídas produzidas pela ferramenta *README-AI* sobre os mesmos repositórios experimentais: Nutri-STTP, Transitolândia Admin e SIGA Talonário Eletrônico. O Quadro 6.2 consolida as notas atribuídas pelo agente a ambas as ferramentas para cada projeto avaliado. As saídas completas geradas por ambas as ferramentas para os três

Critério	1 – Insuficiente	2 – Limitado	3 – Regular	4 – Bom	5 – Excelente
Completude	Faltam seções essenciais; documento esquelético	Maioria das seções presente, mas várias incompletas	Todas as seções presentes, com campos não preenchidos (*placeholders*)	Seções completas; poucos campos pendentes	Seções presentes e plenamente preenchidas; sem *placeholders*
Atualidade	Conteúdo obsoleto ou alucinado (itens inexistentes)	Várias informações desatualizadas ou inventadas	Maioria correta, com imprecisões pontuais	Fiel ao código, com pequenos deslizes	Integralmente fiel ao repositório; sem alucinação
Clareza	Desorganizado; ruídos significativos (ex.: prefixo vazado)	Organização fraca; ruídos frequentes	Legível, com ruídos pontuais que atrapalham	Bem organizado; ruídos mínimos	Estrutura clara e fluida; sem ruídos
Utilidade	Não permite instalar nem usar	Instruções incompletas; muito conhecimento externo	Instruções presentes, mas dependem de conhecimento externo	Instala e usa com esforço mínimo adicional	Instalação e uso reproduzíveis só com o documento
Consistência	Contradições internas; formato instável	Incoerências pontuais; formato irregular	Coerente, mas com variabilidade entre execuções/ruídos	Coerente e estável; variabilidade pequena	Plenamente coerente, estável e reprodutível

Fonte: Autores (2026).

Quadro 6.1: Rubrica de avaliação da documentação gerada

repositórios encontram-se disponíveis, na íntegra, no diretório `examples/` do repositório público do Clarity[1], permitindo a verificação independente das notas aqui atribuídas.

Critério	**Nutri-STTP**		**Transitolândia**		**SIGA**	
	Clarity	README-AI	Clarity	README-AI	Clarity	README-AI
Completude	3	2	4	2	3	2
Atualidade	5	4	5	4	5	4
Clareza	3	2	4	2	3	2
Utilidade	4	4	4	4	4	4
Consistência	3	3	4	3	3	3
Total (máx. 25)	**18**	**15**	**21**	**15**	**18**	**15**

Fonte: Autores (2026).

Quadro 6.2: Resultado comparativo por rubrica (Clarity vs. README-AI)

A análise dos resultados, organizada por critério, evidencia um padrão coerente com as escolhas arquiteturais de cada ferramenta e com as observações qualitativas registradas na Seção 5.5.3.

A **atualidade** foi o critério de melhor desempenho para o Clarity, obtendo nota máxima (5) nos três repositórios. Em todos os casos, a linguagem principal, os *frameworks* (React e Vite nos projetos Nutri-STTP e Transitolândia; Next.js no SIGA), as dependências declaradas no

[1] Disponível em: <https://github.com/FabsMS/Clarity-Extension>. Acesso em: mai. 2026.

`package.json` e os *scripts* de execução foram extraídos com precisão integral. Esse resultado decorre da arquitetura de dois agentes, na qual o Agente Analisador isola a extração estrutural, blindando o Agente Redator contra alucinações. O *README-AI* atingiu nota 4 por mapear adequadamente as tecnologias gerais e os pré-requisitos lógicos, porém apresentou imprecisões marginais na vinculação de módulos secundários.

No critério de **completude**, o *README-AI* demonstrou uma limitação severa, obtendo nota 2 (Limitado) em todos os testes. Conforme exposto nas saídas da ferramenta, seções fundamentais como *Overview*, *Features* e o detalhamento técnico do *Project Index* foram preenchidas com o marcador bruto `> REPLACE-ME`, entregando um documento de estrutura esquelética. O Clarity obteve melhor avaliação (notas 3 e 4) pois, embora mantenha marcadores como `[TAGLINE]` e `[DESCRICAO]` sob baixa densidade de metadados, preencheu as lacunas de forma autônoma e completa no repositório Transitolândia assim que encontrou um cenário estrutural mais rico.

A **clareza** do *README-AI* também foi impactada negativamente (nota 2) pela poluição visual gerada pela persistência recorrente do termo `> REPLACE-ME` nas tabelas explicativas de arquivos. O Clarity (notas 3 e 4) demonstrou organização significativamente superior, apresentando como ruído pontual apenas o vazamento do prefixo `README:` oriundo do *prompt*, falha que compromete ligeiramente a fluidez, mas preserva a legibilidade textual.

Em termos de **consistência**, o *README-AI* obteve nota 3 estável por reproduzir rigidamente o mesmo gabarito estrutural em todos os cenários, ignorando as nuances arquiteturais que diferenciam os projetos. O Clarity obteve nota 3 nos projetos Nutri-STTP e SIGA, em razão da variabilidade inerente às seções inferenciais, mas alcançou nota 4 na Transitolândia ao demonstrar capacidade adaptativa de contextualização sob sinais estruturais ricos.

Na dimensão de **utilidade**, observou-se um empate técnico (nota 4). Ambas as ferramentas mostraram-se eficazes e prontas para o *onboarding* básico, estruturando blocos de código funcionais e reprodutíveis para instalação e execução locais a partir dos comandos reais do ecossistema do software, tais como `npm install`, `npm start` e automações em `Dockerfile`.

Cabe destacar que o *README-AI* obteve pontuação idêntica nos três repositórios (15 pontos em cada), enquanto o Clarity apresentou variação (18, 21 e 18 pontos). Essa uniformidade não é um artefato da avaliação, mas um reflexo direto do funcionamento da ferramenta: por aplicar um gabarito fixo e independente do projeto — com as mesmas seções deixadas como `REPLACE-ME` —, o *README-AI* produz saídas equivalentes entre repositórios distintos. O Clarity, por inferir conteúdo a partir dos sinais estruturais de cada projeto, mostrou-se sensível à riqueza desses sinais, o que explica sua pontuação mais alta justamente no repositório de estrutura mais rica (Transitolândia).

Em síntese, a avaliação comparativa por rubrica oferece indícios — ainda que obtidos por um único avaliador automatizado, e portanto não conclusivos — a favor da especialização agêntica implementada no Clarity. Ao segmentar o fluxo de trabalho e utilizar um esquema estruturado em formato JSON como contrato de dados formal, o sistema delimita com precisão a atuação do estágio de redação. Essa engenharia mitiga falhas críticas e o preenchimento arti-

ficial por marcadores comuns em ferramentas de agente único, garantindo uma documentação final mais fluida e fiel ao código — aspectos retomados no Capítulo 7.

7 CONSIDERAÇÕES FINAIS

Este trabalho partiu de um problema concreto e recorrente na engenharia de software: a tendência de os arquivos README se tornarem incompletos ou desatualizados à medida que o código evolui, alimentando a chamada dívida técnica documental e a perda de conhecimento organizacional (Seção 1.3). Diante desse cenário, o objetivo geral foi desenvolver um sistema inteligente de geração automatizada de documentação técnica que, por meio de uma arquitetura baseada em agentes de Modelos de Linguagem de Grande Escala executados localmente, produzisse documentação estruturada sob demanda, integrada ao ambiente de desenvolvimento. O protótipo Clarity, descrito no Capítulo 5, materializa esse objetivo.

Os objetivos específicos definidos na Seção 1.2 foram contemplados ao longo do trabalho. A investigação do estado da arte sobre documentação de software, dívida técnica, Modelos de Linguagem de Grande Escala e ferramentas correlatas foi conduzida nos Capítulos 2 e 3. A arquitetura da solução — com extração e análise estática de metadados, estratégia de divisão em fragmentos (*chunking*) e orquestração de agentes de LLM via CrewAI com modelos locais servidos pelo Ollama — foi definida no Capítulo 4. O protótipo funcional foi implementado como extensão para o Visual Studio Code, com suporte às linguagens JavaScript, TypeScript, Python e Java, e publicado na loja oficial do editor sob licença MIT (Capítulo 5). Por fim, a avaliação da solução foi realizada no Capítulo 6, por meio da aplicação de uma rubrica com cinco critérios objetivos — completude, atualidade, clareza, utilidade e consistência —, em caráter comparativo com a ferramenta de mercado *README-AI* e mediante julgamento automatizado, recurso que buscou reduzir — ainda que não eliminar — o viés dos autores na atribuição das notas.

Quanto às contribuições, este trabalho oferece, de forma explícita:

1. **Uma arquitetura de geração de documentação com separação entre análise e redação**: a decisão de projeto central — isolar a extração estrutural (Agente Analisador) da redação textual (Agente Redator), mediada por um contrato de dados em JSON — restringe o espaço para alucinação e diferencia o Clarity de soluções de agente único, mais propensas ao preenchimento por marcadores não resolvidos. Na avaliação realizada, essa arquitetura associou-se ao melhor desempenho no critério de atualidade, no qual o Clarity obteve nota máxima nos três repositórios.

2. **Um protótipo funcional, público e de código aberto**: a extensão foi efetivamente implementada, publicada no Visual Studio Code Marketplace e licenciada sob MIT, constituindo um artefato utilizável e inspecionável pela comunidade, e não apenas uma proposta conceitual.

3. **Uma abordagem de execução integralmente local**: ao operar via Ollama, a ferramenta

dispensa o envio de código a servidores externos e elimina custos recorrentes de API, atributo relevante para contextos corporativos com restrições de privacidade.

4. **Um mecanismo não destrutivo de adoção**: a geração de um arquivo separado (README-CLARITY.md), preservando o README.md original, elimina o risco de sobrescrita acidental e mantém o desenvolvedor no controle da incorporação do conteúdo.

5. **Um instrumento de avaliação comparativa**: a rubrica de cinco critérios, aplicada de forma comparativa a uma ferramenta de mercado, oferece um ponto de partida replicável para a avaliação de documentação gerada automaticamente.

Cabe ressalvar que o desempenho superior observado restringe-se ao conjunto avaliado e ao julgamento de um único avaliador automatizado, não devendo ser generalizado como superioridade absoluta sobre a ferramenta de referência.

As limitações observadas foram igualmente expostas pela avaliação. No plano da qualidade da geração, concentram-se em três frentes: as seções de natureza inferencial — como a *tagline* e as descrições de pastas — são deixadas como marcadores (*placeholders*) quando não há sinais estruturais suficientes, reduzindo a completude e a consistência; o prefixo de *prompt* (`README:`) vaza no início dos arquivos gerados, ruído que afeta a clareza; e há variabilidade entre execuções nas seções inferenciais. A essas somam-se limitações de escopo já anunciadas na delimitação (Seção 1.5): a análise estática de JavaScript, TypeScript e Java baseia-se em reconhecimento de padrões textuais, e não em análise sintática completa como a empregada para Python; o suporte restringe-se a quatro linguagens e a repositórios locais; e a obtenção de tempos de execução adequados beneficia-se de *hardware* com aceleração gráfica. Por fim, a própria avaliação possui limitações de validade: foi conduzida sobre um conjunto reduzido de três repositórios e por meio de um julgador automatizado, o que, embora reduza o viés dos autores, não substitui a apreciação de usuários reais.

Em balanço, conclui-se que a proposta é viável e promissora para o problema a que se destina. Mesmo na condição de protótipo em estágio inicial (versão *Beta*), o Clarity demonstrou ser capaz de produzir, de forma automatizada e local, documentação fiel ao código e útil como ponto de partida para o desenvolvedor, com desempenho favorável frente a uma ferramenta de mercado nos casos avaliados. Cabe distinguir, contudo, o que foi efetivamente demonstrado do que permanece em aberto: o trabalho evidencia a *viabilidade técnica* de gerar localmente um README de qualidade e apresenta *indícios* de que esse artefato pode servir de apoio à mitigação da dívida técnica documental. A comprovação de que a ferramenta *reduz* efetivamente a dívida documental ao longo do tempo — o que exigiria um estudo longitudinal, com acompanhamento de projetos reais em uso contínuo — não está no escopo deste trabalho e permanece como questão em aberto.

7.1 IMPLICAÇÕES

Os resultados deste trabalho têm implicações tanto para a pesquisa quanto para a prática. Do ponto de vista da **pesquisa**, o trabalho reforça a hipótese de que a separação entre extração estrutural e geração textual — mediada por um contrato de dados formal — é um mecanismo eficaz para conter alucinações em tarefas de documentação conduzidas por LLMs de menor escala. Esse achado sugere uma direção promissora para investigações sobre geração aumentada por dados estruturados, alternativa ao uso de modelos cada vez maiores. Além disso, a rubrica proposta e a estratégia de avaliação comparativa oferecem um instrumento reutilizável para estudos futuros sobre a qualidade de documentação gerada automaticamente, ainda que a substituição do avaliador humano por um agente de IA imponha as ressalvas discutidas adiante.

Do ponto de vista da **prática**, o trabalho demonstra que a geração local de documentação é factível com *hardware* de uma estação de desenvolvimento de nível intermediário, o que torna a abordagem acessível a equipes que não podem ou não desejam enviar seu código a serviços externos — em especial em setores com exigências de privacidade e conformidade. A disponibilização da ferramenta como extensão pública e de código aberto reduz a barreira de adoção e permite que profissionais a incorporem imediatamente a seu fluxo de trabalho, ainda que como apoio sujeito a revisão humana.

7.2 AMEAÇAS À VALIDADE

Como todo estudo empírico, este trabalho está sujeito a ameaças à validade que devem ser consideradas na interpretação de seus resultados. Quanto à **validade de construto**, a rubrica de cinco critérios é uma operacionalização possível — entre outras — do conceito de "qualidade de README", e a tradução de critérios qualitativos em uma escala numérica envolve simplificações. Quanto à **validade interna**, a atribuição das notas foi realizada por um único avaliador automatizado, o que impede o cálculo de concordância entre avaliadores e deixa a pontuação sujeita a eventuais vieses e à variabilidade do próprio modelo julgador. Quanto à **validade externa**, a avaliação restringiu-se a três repositórios, todos do ecossistema *front-end* de uma mesma organização e cedidos pela empresa de um dos autores, o que limita a generalização dos resultados para outros domínios, linguagens e portes de projeto. Por fim, quanto à **validade de conclusão**, o caráter aproximado de algumas métricas de execução (número de *chunks* e tempos) e a variabilidade inerente à geração por LLMs recomendam que as diferenças observadas entre as ferramentas sejam lidas como indicativas, e não como definitivas. Essas ameaças não invalidam os achados, mas delimitam seu alcance e orientam os trabalhos futuros descritos a seguir.

7.3 SUGESTÕES PARA TRABALHOS FUTUROS

As limitações e ameaças identificadas apontam diretamente para as oportunidades de evolução da ferramenta, apresentadas a seguir em ordem de prioridade sugerida.

A **primeira prioridade** é aprimorar a análise multilinguagem. A substituição do reconhecimento por expressões regulares por uma análise sintática baseada em *parsers* de árvore — como o *Tree-sitter*, discutido na Seção 2.4 — ampliaria a precisão da extração em JavaScript, TypeScript e Java, equiparando-a à obtida em Python e abrindo caminho para a inclusão de novas linguagens. Por atacar a raiz de parte das imprecisões, essa é a evolução de maior impacto.

A **segunda prioridade** é a introdução de um agente revisor. A arquitetura atual encadeia dois agentes; a inclusão de um terceiro, incumbido de verificar a saída do Agente Redator contra o JSON intermediário antes da entrega — inspirada na lógica de autocorreção discutida no Capítulo 3 —, tenderia a reduzir tanto o vazamento do prefixo de *prompt* quanto a persistência de *placeholders*, elevando a confiabilidade da documentação gerada.

A **terceira prioridade** é o fortalecimento da avaliação. Recomenda-se ampliar o estudo a um conjunto maior e mais diversificado de repositórios e, sobretudo, complementar o julgamento automatizado com uma avaliação empírica conduzida por avaliadores humanos independentes — preferencialmente desenvolvedores — aplicando a rubrica de forma independente, bem como um estudo de utilidade percebida com usuários reais, apoiado em modelos consolidados de aceitação de tecnologia, como o TAM (*Technology Acceptance Model*) ou o UTAUT (*Unified Theory of Acceptance and Use of Technology*). Esse passo é o que permitiria migrar de indícios para evidências mais robustas de qualidade e utilidade.

A **quarta prioridade** reúne evoluções de integração e operação: a inclusão de um canal dedicado de saída persistente (*Output*) no Visual Studio Code, que viabilize a coleta sistemática de métricas de execução (incluindo LoC, número de *chunks* e tempo); a integração com repositórios remotos e com o sistema de controle de versão Git, permitindo atualizar a documentação de forma atrelada a *commits* e *pull requests*; e a adoção de um fluxo automatizado de integração e entrega contínuas (CI/CD) para a publicação da extensão. Complementarmente, no plano da qualidade da geração, permanecem como melhorias incrementais o tratamento do vazamento do prefixo de *prompt* no pós-processamento e a redução da variabilidade entre execuções por refinamento da engenharia de *prompt* e da estrutura do JSON (Seção 5.3.4).

REFERÊNCIAS

AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. 2. ed. Boston: Pearson/Addison-Wesley, 2006.

ALBUQUERQUE, D. et al. Managing technical debt using intelligent techniques — a systematic mapping study. *IEEE Transactions on Software Engineering*, v. 49, n. 4, p. 2202–2220, 2023.

ALBUQUERQUE, D. et al. Perceptions of technical debt and its management activities — a survey of software practitioners. In: *Anais do XXXVI Simpósio Brasileiro de Engenharia de Software (SBES)*. Uberlândia: Sociedade Brasileira de Computação, 2022. p. 220–229. ISSN 2833-0633.

BROWN, T. et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, v. 33, p. 1877–1901, 2020.

CHAKRABARTY, S. et al. *ReadmeReady: free and customizable code documentation with LLMs — a fine-tuning approach*. 2024. ArXiv preprint arXiv:2412.00726. Disponível em: <https://arxiv.org/abs/2412.00726>. Acesso em: mar. 2026.

CHEN, M. et al. *Evaluating Large Language Models Trained on Code*. 2021. ArXiv preprint arXiv:2107.03374. Disponível em: <https://arxiv.org/abs/2107.03374>. Acesso em: abr. 2026.

CLEM, T.; THOMSON, P. Static analysis at github: an experience report. *Queue*, v. 19, n. 4, p. 42–67, set. 2021. DOI: 10.1145/3487019.3487022.

CUNNINGHAM, W. *The WyCash Portfolio Management System*. 1992. Experience Report, OOPSLA '92 — Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications. Disponível em: <https://c2.com/doc/oopsla92.html>. Acesso em: jun. 2026.

DEVLIN, J. et al. Bert: Pre-training of deep bidirectional transformers for language understanding. In: *Conference of the North American Chapter of the Association for Computational Linguistics*. Minneapolis: ACL, 2019. p. 4171–4186.

ECMA International. *ECMA-404: The JSON Data Interchange Format*. Geneva: Ecma International, 2013. Disponível em: <https://www.ecma-international.org/publications-and-standards/standards/ecma-404/>. Acesso em: mar. 2026.

FENG, Z. et al. Codebert: A pre-trained model for programming and natural languages. In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. [S.l.: s.n.], 2020. p. 1536–1547. Disponível em: <https://aclanthology.org/2020.findings-emnlp.139>. Acesso em: abr. 2026.

FOWLER, M. *Technical debt quadrant*. 2009. Martin Fowler's Bliki. Disponível em: <https://martinfowler.com/bliki/TechnicalDebtQuadrant.html>. Acesso em: mar. 2026.

FREIRE, E. S. et al. Technical debt management in agile software development: a systematic mapping study. In: *Brazilian Symposium on Software Quality (SBQS)*. New York: ACM, 2024. DOI: 10.1145/3701625.3701669.

HOCHREITER, S.; SCHMIDHUBER, J. Long short-term memory. *Neural Computation*, v. 9, n. 8, p. 1735–1780, 1997.

HOU, X. et al. *Large Language Models for Software Engineering: A Survey*. 2023. ArXiv preprint arXiv:2305.12138. Disponível em: <https://arxiv.org/abs/2305.12138>. Acesso em: abr. 2026.

LUO, Q. et al. Repoagent: an llm-powered open-source framework for repository-level code documentation generation. In: *Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Miami: ACL, 2024. p. 436–464. DOI: 10.18653/v1/2024.emnlp-demo.46.

MENDES, T. S. et al. Impacts of agile requirements documentation debt on software projects: a retrospective study. In: *ACM Symposium on Applied Computing*. Pisa: ACM, 2016. p. 1290–1295. DOI: 10.1145/2851613.2851761.

MOURA, J. *CrewAI: Framework for orchestrating role-playing, autonomous AI agents*. 2023. Disponível em: <https://github.com/crewAIInc/crewAI>. Acesso em: mar. 2026.

NARVáEZ-NARVáEZ, J. C.; PARDO-CALVACHE, C. J.; OROZCO-GARCéS, C. E. Deuda de la documentación en el desarrollo ágil de software: mapeo sistemático de la literatura. *Revista Científica*, v. 46, n. 1, p. 107–121, jan. 2023. DOI: 10.14483/23448350.19670.

NONAKA, I.; TAKEUCHI, H. *The Knowledge-Creating Company: How Japanese Companies Create the Dynamics of Innovation*. New York: Oxford University Press, 1995.

PRESSMAN, R. S.; MAXIM, B. R. *Engenharia de Software: Uma Abordagem Profissional*. 9. ed. Porto Alegre: AMGH, 2020.

RUSSELL, S.; NORVIG, P. *Artificial Intelligence: A Modern Approach*. 4. ed. Hoboken: Pearson, 2021.

SALAMIE, E. *README-AI: Automated README file generator*. 2023. Disponível em: <https://github.com/eli64s/readme-ai>. Acesso em: mar. 2026.

SOMMERVILLE, I. *Engenharia de Software*. 10. ed. São Paulo: Pearson, 2019.

Stack Overflow. *2024 Developer Survey: Technology*. 2024. Disponível em: <https://survey.stackoverflow.co/2024/technology>. Acesso em: mai. 2026.

TAN, W. S.; WAGNER, M.; TREUDE, C. Detecting outdated code element references in software repository documentation. *Empirical Software Engineering*, v. 28, nov. 2023. DOI: 10.1007/s10664-023-10401-z.

VASWANI, A. et al. Attention is all you need. In: *Advances in Neural Information Processing Systems*. Long Beach: Curran Associates, 2017. p. 5998–6008.

WOOLDRIDGE, M. *An Introduction to MultiAgent Systems*. 2. ed. Chichester: Wiley, 2009.

ZHANG, K. et al. Self-edit: Fault-aware code editor for code generation. In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. Toronto: ACL, 2023. p. 769–787. Disponível em: <https://aclanthology.org/2023.acl-main.43>. Acesso em: abr. 2026.

Apêndices

APÊNDICE A – README-CLARITY.MD GERADO PARA O REPOSITÓRIO NUTRI-STTP

Este apêndice reproduz, na íntegra, o conteúdo do arquivo README-CLARITY.md gerado automaticamente pelo protótipo do Clarity sobre o repositório Nutri-STTP — sistema de marcação de consultas de nutrição da Superintendência de Trânsito e Transportes Públicos de Campina Grande (STTP). O documento é apresentado exatamente como produzido pela ferramenta, sem qualquer edição manual posterior, com a finalidade de viabilizar a análise direta do artefato pela banca examinadora e servir como base para a futura avaliação formal por rubrica prevista na Seção 1.6.

A execução foi realizada na versão 0.0.5 (Beta) do protótipo, sobre a estação de trabalho descrita na Seção 5.5.1, utilizando os modelos `deepseek-coder:6.7b` (Agente Analisador, temperatura 0.1) e `llama3:8b` (Agente Redator, temperatura 0.4), ambos servidos localmente via Ollama. O tempo total de execução foi de 66,9 segundos. A discussão qualitativa sobre os pontos fortes e as limitações observadas neste arquivo encontra-se na Seção 5.5.3.

```
README:

<!--
      [ATENCAO] - README gerado automaticamente pelo CLARITY
  Antes de publicar, revise e preencha todos os campos marcados com TODO.
  Remova este bloco de comentário quando o README estiver pronto.
-->

# **PROJETO-NUTRICAO**

> **[ATENCAO] Aviso:** Este README foi gerado automaticamente. Revise as
> informações e preencha os campos marcados com `<!-- TODO -->` antes de publicar.

> [TAGLINE] - Sistema de Nutrição para Gestão de Consultas e Pacientes

![Licença](https://img.shields.io/badge/licen%C3%A7a-MIT-blue)
![Linguagem](https://img.shields.io/badge/linguagem-JavaScript-informational)
![Status](https://img.shields.io/badge/status-em%20desenvolvimento-yellow)

## Índice

- [Sobre](#sobre)
- [Funcionalidades](#funcionalidades)
- [Tecnologias](#tecnologias)
- [Estrutura do Projeto](#estrutura-do-projeto)
- [Pré-requisitos](#pre-requisitos)
- [Instalação](#instalacao)
- [Como Usar](#como-usar)
- [Variáveis de Ambiente](#variaveis-de-ambiente)
- [Contribuição](#contribuicao)
- [Licença](#licenca)
```

```
---

## Sobre

O Projeto-Nutricao é um sistema de gestão de consultas e pacientes,
desenvolvido em JavaScript utilizando React e Vite. O objetivo do projeto
é fornecer uma plataforma para nutricionistas e pacientes gerenciar
consultas, anamneses e dados de pacientes.

[DESCRICAO]

## Funcionalidades

- Cadastro e edição de pacientes
- Gerenciamento de consultas e agendamentos
- Anamnese e histórico médico dos pacientes
- Gestão de dados de pacientes, incluindo informações nutricionais e de saúde
- Autenticação e autorização para acesso às funcionalidades do sistema

## Tecnologias

**Linguagem principal:** JavaScript

**Frameworks e Ferramentas:**

- **React** - Biblioteca de componentes para construir interfaces de usuário
- **Vite** - Framework de desenvolvimento rápido e escalável para aplicativos web

**Dependências principais:**

- `@date-fns/utc`
- `@emotion/react`
- `@emotion/styled`
- `@lottiefiles/react-lottie-player`
- `@mui/icons-material`
- `@mui/material`
- `@mui/x-date-pickers`
- `@types/js-cookie`
- `@types/react`
- `@types/react-dom`
- `@vitejs/plugin-react`
- `axios`
- `date-fns`

## Estrutura do Projeto

```
projeto-nutricao/
|-- Anamnese/
| `-- (1 arquivo)
|-- Animations/
| `-- (1 arquivo)
|-- BlocoDeConsulta/
| `-- (1 arquivo)
|-- ... e mais 20 pastas e arquivos
```
```

```
[DESCRICAO_PASTAS]

## Pré-requisitos

Antes de começar, instale as seguintes ferramentas:

- [Node.js](https://nodejs.org/) v18 ou superior
- npm v9+

## Instalação

‘‘‘bash
# 1. Clone o repositório
git clone https://github.com/<SEU-USUARIO>/projeto-nutricao.git
cd projeto-nutricao

# 2. Instale as dependências
npm install
# ou: yarn install
‘‘‘

> Ajuste a URL do seu projeto no github.

## Como Usar

**Desenvolvimento:**

‘‘‘bash
npm run dev
‘‘‘

**Produção:**

‘‘‘bash
npm run build
npm run preview
‘‘‘

## Variáveis de Ambiente

Crie um arquivo ‘.env‘ na raiz do projeto com as seguintes variáveis:

‘‘‘env
# <!-- TODO: liste aqui as variáveis de ambiente necessárias -->
# Exemplo:
# API_URL=http://localhost:3000
# SECRET_KEY=sua_chave_secreta
‘‘‘

> Configure as variáveis de ambiente com as variáveis necessárias.

## Contribuição

Contribuições são bem-vindas! Siga os passos abaixo:

1. Faça um **fork** do projeto
```

````
2. Crie uma branch para sua feature:
```bash
git checkout -b feature/minha-feature
```
3. Commit suas alterações:
```bash
git commit -m 'feat: adiciona minha feature'
```
4. Push para a branch:
```bash
git push origin feature/minha-feature
```
5. Abra um **Pull Request**

## Licença

Este projeto está sob a licença **MIT**. Veja o arquivo [LICENSE](LICENSE)
para mais detalhes.

---

<p align="center">
    Gerado por <a href="https://github.com/FabsMS/Clarity-Extension">CLARITY</a> </p>
````

APÊNDICE B – README-CLARITY.MD GERADO PARA O REPOSITÓRIO TRANSITOLÂNDIA ADMIN

Este apêndice reproduz, na íntegra, o conteúdo do arquivo README-CLARITY.md gerado automaticamente pelo protótipo do Clarity sobre o repositório `admin-transitolandia-front` — painel administrativo de gerenciamento de perguntas, jogos de memória e simulados do aplicativo Transitolândia, mantido pela Superintendência de Trânsito e Transportes Públicos de Campina Grande (STTP). O documento é apresentado exatamente como produzido pela ferramenta, sem qualquer edição manual posterior, com a finalidade de viabilizar a análise direta do artefato pela banca examinadora e servir como base para a futura avaliação formal por rubrica prevista na Seção 1.6.

A execução foi realizada na versão 0.0.5 (Beta) do protótipo, sobre a estação de trabalho descrita na Seção 5.5.1, utilizando os modelos `deepseek-coder:6.7b` (Agente Analisador, temperatura 0.1) e `llama3:8b` (Agente Redator, temperatura 0.4), ambos servidos localmente via Ollama. O tempo total de execução foi de 50,0 segundos. A discussão qualitativa sobre os pontos fortes e as limitações observadas neste arquivo encontra-se na Seção 5.5.3.

```
README:

<!--
    ATENÇÃO -- README gerado automaticamente pelo CLARITY
  Antes de publicar, revise e preencha todos os campos marcados com TODO.
  Remova este bloco de comentário quando o README estiver pronto.
-->

# **ADMIN-TRANSITOLANDIA-FRONT**

> ** Aviso:** Este README foi gerado automaticamente. Revise as informações e
> preencha os campos marcados com ‘<!-- TODO -->‘ antes de publicar.

> **Desenvolvendo memória e conhecimento para a Transitolandia!**

![Licença](https://img.shields.io/badge/licen%C3%A7a-MIT-blue)
![Linguagem](https://img.shields.io/badge/linguagem-JavaScript-informational)
![Status](https://img.shields.io/badge/status-em%20desenvolvimento-yellow)

## Índice

- [Sobre](#sobre)
- [Funcionalidades](#funcionalidades)
- [Tecnologias](#tecnologias)
- [Estrutura do Projeto](#estrutura-do-projeto)
- [Pré-requisitos](#pre-requisitos)
- [Instalação](#instalacao)
- [Como Usar](#como-usar)
- [Variáveis de Ambiente](#variaveis-de-ambiente)
- [Contribuição](#contribuicao)
```

```
- [Licença](#licenca)

---

## <a id="sobre"></a> Sobre

O projeto admin-transitolandia-front é uma aplicação web desenvolvida em JavaScript utilizando o
     framework React e a ferramenta Vite. O objetivo do sistema é fornecer uma plataforma para
    que os usuários possam gerenciar perguntas, jogos de memória e simulados, além de realizar
    login e cadastro.

O sistema resolve o problema de falta de uma plataforma unificada para a Transitolandia,
    permitindo que os usuários tenham acesso a conteúdo educacional e divertido de forma fácil e
     escalável. Além disso, o sistema é projetado para ser escalável e flexível, permitindo que
    novas funcionalidades sejam adicionadas facilmente.

## <a id="funcionalidades"></a> Funcionalidades

- Cadastro e edição de perguntas
- Gerenciamento de jogos de memória
- Simulados para avaliação de conhecimento
- Login e cadastro de usuários
- Visualização de conteúdo educacional

## <a id="tecnologias"></a> Tecnologias

**Linguagem principal:** JavaScript

**Frameworks e Ferramentas:**

- **React** -- um framework para construir interfaces de usuário
- **Vite** -- uma ferramenta para desenvolvimento rápido e escalável

**Dependências principais:**

- `@emotion/styled`
- `@eslint/js`
- `@mui/icons-material`
- `@mui/material`
- `@types/react`
- `@types/react-dom`
- `@vitejs/plugin-react`
- `axios`
- `eslint`
- `eslint-plugin-react-hooks`
- `eslint-plugin-react-refresh`
- `globals`
- `js-cookie`
- `jwt-decode`
- `react`

## <a id="estrutura-do-projeto"></a> Estrutura do Projeto

```
admin-transitolandia-front/
|-- Notification/
| `-- (1 arquivo)
```
```

```
|-- PrivateRoute/
|   `-- (1 arquivo)
|-- Sidebar/
|   `-- (1 arquivo)
|-- Topbar/
|   `-- (1 arquivo)
|-- Transitolandia/
|   |-- eslint.config.js
|   `-- vite.config.js
|-- contexts/
|   `-- (1 arquivo)
|-- hooks/
|   `-- (1 arquivo)
|-- pages/
|   |-- AddQuestionQuiz.jsx
|   |-- AddQuestionSimulado.jsx
|   |-- Initial.jsx
|   |-- SignIn.jsx
|   |-- UpdateQuestionQuiz.jsx
|   `-- ... e mais 5 arquivo(s)
|-- routes/
|   `-- (1 arquivo)
|-- services/
|   |-- HttpService.js
|   |-- JogoMemoriaService.js
|   |-- QuizService.js
|   `-- ... e mais 2 arquivo(s)
|-- src/
|   `-- (1 arquivo)
`-- utils/
    `-- (1 arquivo)
```

Descrição das pastas:

- `Notification`: responsável por notificar os usuários sobre novos conteúdos ou atualizações
- `PrivateRoute`: define rotas privadas para usuários autenticados
- `Sidebar`: gerencia a navegação lateral do sistema
- `Topbar`: gerencia a navegação superior do sistema
- `Transitolandia`: contém arquivos de configuração para o sistema

 Pré-requisitos

Antes de começar, instale as seguintes ferramentas:

- [Node.js](https://nodejs.org/) v18 ou superior
- npm v9+

 Instalação

```bash
# 1. Clone o repositório
git clone https://github.com/<SEU-USUARIO>/admin-transitolandia-front.git
cd admin-transitolandia-front

# 2. Instale as dependências
npm install
```

````
# ou: yarn install
```

> Ajuste a URL do seu projeto no github.

## <a id="como-usar"></a> Como Usar

**Desenvolvimento:**

```bash
npm run dev
```

**Produção:**

```bash
npm run build
npm run preview
```

## <a id="variaveis-de-ambiente"></a> Variáveis de Ambiente

Crie um arquivo `.env` na raiz do projeto com as seguintes variáveis:

```env
# <!-- TODO: liste aqui as variáveis de ambiente necessárias -->
# Exemplo:
# API_URL=http://localhost:3000
# SECRET_KEY=sua_chave_secreta
```

> Configure as variáveis de ambiente com as variáveis necessárias.

## <a id="contribuicao"></a> Contribuição

Contribuições são bem-vindas! Siga os passos abaixo:

1. Faça um **fork** do projeto
2. Crie uma branch para sua feature:
```bash
git checkout -b feature/minha-feature
```
3. Commit suas alterações:
```bash
git commit -m 'feat: adiciona minha feature'
```
4. Push para a branch:
```bash
git push origin feature/minha-feature
```
5. Abra um **Pull Request**

## <a id="licenca"></a> Licença

Este projeto está sob a licença **MIT**. Veja o arquivo [LICENSE](LICENSE) para mais detalhes.

---
````
````

```
<p align="center">
  Gerado por <a href="https://github.com/FabsMS/Clarity-Extension">CLARITY</a> </p>
```

APÊNDICE C – README-CLARITY.MD GERADO PARA O REPOSITÓRIO SIGA TALONÁRIO ELETRÔNICO

Este apêndice reproduz, na íntegra, o conteúdo do arquivo README-CLARITY.md gerado automaticamente pelo protótipo do Clarity sobre o repositório SIGA Talonário Eletrônico — sistema de gestão de talonário eletrônico de infrações da Superintendência de Trânsito e Transportes Públicos de Campina Grande (STTP). O documento é apresentado exatamente como produzido pela ferramenta, sem qualquer edição manual posterior, com a finalidade de viabilizar a análise direta do artefato pela banca examinadora e servir como base para a futura avaliação formal por rubrica prevista na Seção 1.6.

A execução foi realizada na versão 0.0.5 (Beta) do protótipo, sobre a estação de trabalho descrita na Seção 5.5.1, utilizando os modelos `deepseek-coder:6.7b` (Agente Analisador, temperatura 0.1) e `llama3:8b` (Agente Redator, temperatura 0.4), ambos servidos localmente via Ollama. O tempo total de execução foi de 44,8 segundos. Este repositório também foi a fonte da Figura 4.2 do Capítulo 4, que ilustra a estrutura intermediária em formato JSON produzida pelo Agente Analisador. A discussão qualitativa sobre os pontos fortes e as limitações observadas neste arquivo encontra-se na Seção 5.5.3.

```
README:

<!--
    ATENÇÃO -- README gerado automaticamente pelo CLARITY
  Antes de publicar, revise e preencha todos os campos marcados com TODO.
  Remova este bloco de comentário quando o README estiver pronto.
-->

# **SIGA-TALONARIO-ELETRONICO**

> ** Aviso:** Este README foi gerado automaticamente. Revise as informações e
> preencha os campos marcados com ‘<!-- TODO -->‘ antes de publicar.

> [TAGLINE] - Sistema Eletrônico de Gestão de Talonário

![Licença](https://img.shields.io/badge/licen%C3%A7a-MIT-blue)
![Linguagem](https://img.shields.io/badge/linguagem-TypeScript-informational)
![Status](https://img.shields.io/badge/status-em%20desenvolvimento-yellow)

## Índice

- [Sobre](#sobre)
- [Funcionalidades](#funcionalidades)
- [Tecnologias](#tecnologias)
- [Estrutura do Projeto](#estrutura-do-projeto)
- [Pré-requisitos](#pre-requisitos)
```

```
- [Instalação](#instalacao)
- [Como Usar](#como-usar)
- [Variáveis de Ambiente](#variaveis-de-ambiente)
- [Contribuição](#contribuicao)
- [Licença](#licenca)

---

## Sobre

O SIGA-Talonario-Eletronico é um sistema eletrônico de gestão de talonário desenvolvido para
    automatizar e otimizar o processo de gerenciamento de documentos. O sistema foi construído
    utilizando TypeScript, Next.js e React, e integra tecnologias como @emotion/react, @emotion/
    styled, @mui/icons-material, @mui/material e @tailwindcss/postcss.

O objetivo do SIGA-Talonario-Eletronico é fornecer uma plataforma segura e escalável para a gest
    ão de documentos, permitindo que os usuários realizem operações como cadastro, edição e
    impressão de documentos com facilidade e eficiência.

## Funcionalidades

- Cadastro e edição de perguntas
- Gestão de catálogo de produtos
- Consultas por condutor e veículo
- Impressão de documentos
- Autenticação e autorização de usuários
- Gerenciamento de lotes e processos

## Tecnologias

**Linguagem principal:** TypeScript

**Frameworks e Ferramentas:**

- **Next.js** -- Framework de construção de aplicativos web com servidor de lado do cliente
- **React** -- Biblioteca JavaScript para construir interfaces de usuário interativas

**Dependências principais:**

- `@emotion/react`
- `@emotion/styled`
- `@mui/icons-material`
- `@mui/material`
- `@tailwindcss/postcss`
- `@types/node`
- `@types/react`
- `@types/react-dom`
- `bootstrap`
- `eslint`
- `eslint-config-next`
- `next`
- `react`
- `react-bootstrap`
- `react-dom`

## Estrutura do Projeto
```

```
```
siga-talonario-eletronico/
|-- (components)/
| |-- DetalhesTipoInfracao.tsx
| |-- DetalhesVeiculo.tsx
| |-- InfracoesFiltro.tsx
| |-- InfracoesOpcaoCard.tsx
| |-- InfracoesTableOperacoes.tsx
| `-- ... e mais 34 arquivo(s)
|-- (private)/
| `-- layout.tsx
|-- (public)/
| `-- (1 arquivo)
|-- DataTable/
| `-- (2 arquivos)
|-- ProgressBar/
| |-- ProgressBar.tsx
| `-- ... e mais 1 arquivo(s)
|-- SIGA---Talonario-Eletronico/
| |-- next-env.d.ts
| `-- next.config.ts
|-- Sidebar/
| |-- NavLink.tsx
| |-- NavLinkClient.tsx
| |-- Sidebar.tsx
| |-- index.ts
| `-- ... e mais 1 arquivo(s)
|-- [id]/
| |-- page.tsx
| |-- page.tsx
| `-- ... e mais 1 arquivo(s)
|-- ait/
| `-- (1 arquivo)
|-- app/
| |-- layout.tsx
| `-- page.tsx
|-- auth/
| `-- (1 arquivo)
|-- condutores/
| `-- (1 arquivo)
|-- condutores-detran/
| `-- (1 arquivo)
|-- consultas/
| `-- (1 arquivo)
|-- impressao-nas/
| `-- (2 arquivos)
|-- impressao-nps/
| `-- (2 arquivos)
|-- infracoes/
| |-- index.ts
| `-- ... e mais 2 arquivo(s)
|-- infracoes-transporte/
| `-- (1 arquivo)
|-- lib/
| `-- (1 arquivo)
|-- login/
| `-- (1 arquivo)
```
```

```
|-- lotes/
|   |-- page.tsx
|   `-- ... e mais 1 arquivo(s)
|-- mock/
|   `-- (1 arquivo)
|-- novo/
|   |-- page.tsx
|   `-- page.tsx
|-- processos/
|   `-- (1 arquivo)
|-- transportes/
|   |-- index.ts
|   `-- ... e mais 1 arquivo(s)
|-- types/
|   |-- usuarios.ts
|   |-- veiculos.ts
|   `-- ... e mais 10 arquivo(s)
|-- usuarios/
|   |-- index.ts
|   `-- ... e mais 1 arquivo(s)
`-- validacao/
    |-- ValidacaoContent.tsx
    `-- index.ts

[DESCRICAO_PASTAS]

## Pré-requisitos

Antes de começar, instale as seguintes ferramentas:

- [Node.js](https://nodejs.org/) v18 ou superior
- npm v9+

## Instalação

```bash
# 1. Clone o repositório
git clone https://github.com/<SEU-USUARIO>/siga-talonario-eletronico.git
cd siga-talonario-eletronico

# 2. Instale as dependências
npm install
# ou: yarn install
```

> Ajuste a URL do seu projeto no github.

Como Usar

Desenvolvimento:

```bash
npm run dev
```

Produção:
```

````
```bash
npm run build
npm start
```

## Variáveis de Ambiente

Crie um arquivo `.env` na raiz do projeto com as seguintes variáveis:

```env
# <!-- TODO: liste aqui as variáveis de ambiente necessárias -->
# Exemplo:
# API_URL=http://localhost:3000
# SECRET_KEY=sua_chave_secreta
```

> Configure as variáveis de ambiente com as variáveis necessárias.

## Contribuição

Contribuições são bem-vindas! Siga os passos abaixo:

1. Faça um **fork** do projeto
2. Crie uma branch para sua feature:
```bash
git checkout -b feature/minha-feature
```
3. Commit suas alterações:
```bash
git commit -m 'feat: adiciona minha feature'
```
4. Push para a branch:
```bash
git push origin feature/minha-feature
```
5. Abra um **Pull Request**

## Licença

Este projeto está sob a licença **MIT**. Veja o arquivo [LICENSE](LICENSE) para mais detalhes.

---

<p align="center">
  Gerado por <a href="https://github.com/FabsMS/Clarity-Extension">CLARITY</a> </p>
````

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Campus Campina Grande - Código INEP: 25137409
R. Tranqüilino Coelho Lemos, 671, Dinamérica, CEP 58432-300, Campina Grande (PB)
CNPJ: 10.783.898/0003-37 - Telefone: (83) 2102.6200

Documento Digitalizado Restrito

Trabalho de Conclusão de Curso - Versão com Ficha Catalográfica

Assunto:	Trabalho de Conclusão de Curso - Versão com Ficha Catalográfica
Assinado por:	Ynnayron Juan
Tipo do Documento:	Relatório
Situação:	Finalizado
Nível de Acesso:	Restrito
Hipótese Legal:	Direito Autoral (Art. 24, III, da Lei no 9.610/1998)
Tipo do Conferência:	Cópia Simples

Documento assinado eletronicamente por:

- **Ynnayron Juan Lopes da Silva**, **DISCENTE (202111250027) DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO - CAMPINA GRANDE**, em 30/07/2026 14:43:59.

Este documento foi armazenado no SUAP em 30/07/2026. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 1931348
Código de Autenticação: efa42bcd11