



Universidade Católica do Salvador
Bacharelado em Engenharia de Software

André Luis Trocoli Azevedo Cruz
Gabriel de Souza Oliveira Guimarães Cunha
Jade Sampaio Cerqueira
Maria Eduarda de Paula Amaral

**DocScrum: Geração, validação e autocrítica fundamentada de
artefatos ágeis a partir de código Java por LLM**

Salvador

2026

André Luis Trocoli Azevedo Cruz
Gabriel de Souza Oliveira Guimarães Cunha
Jade Sampaio Cerqueira
Maria Eduarda de Paula Amaral

**DocScrum: Geração, validação e autocrítica
fundamentada de artefatos ágeis a partir de código
Java por LLM**

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

Universidade Católica do Salvador

Salvador

2026

André Luis Trocoli Azevedo Cruz
Gabriel de Souza Oliveira Guimarães Cunha
Jade Sampaio Cerqueira
Maria Eduarda de Paula Amaral

DocScrum: Geração, validação e autocrítica fundamentada de artefatos ágeis a partir de código Java por LLM

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

Salvador, 30 de junho de 2026

Banca Examinadora:

Prof. Elton Figueiredo da Silva
Universidade Católica do Salvador
Orientador

Prof. Flavio Dusse
Universidade Católica do Salvador

Prof. Rafael Bispo de Souza Neto
Universidade Católica do Salvador

Agradecimentos

Eu, Jade Sampaio Cerqueira, agradeço a Deus pela força e por ter me ajudado a chegar até aqui. Agradeço à minha família pelo apoio, incentivo e por sempre estarem ao meu lado durante essa jornada. Aos professores, agradeço pelos ensinamentos e pela contribuição na minha formação acadêmica. Aos colegas e amigos, obrigada pelo companheirismo e pelas trocas ao longo do curso. Por fim, agradeço a todos que contribuíram, de alguma forma, para a realização deste trabalho.

Eu, André Luis Trocoli Azevedo Cruz, dedico este trabalho e a minha trajetória acadêmica à minha mãe, Andrea Rocha Trocoli, que com toda a sua virtude me guiou e guia todos os dias; sem ela, nada disso seria possível. Gostaria de agradecer a Deus por tudo que Ele me proporciona e ao meu pai, que me ofereceu as condições necessárias para a realização dos meus estudos. Ademais, agradeço ao meu coordenador de TCC, professor Elton Figueiredo da Silva, ao coordenador do bacharelado, Osvaldo Requião Melo, e à minha equipe. Todos vocês foram fundamentais nesse processo de construção. Muito obrigado.

Eu, Gabriel de Souza Oliveira Guimarães Cunha, dedico esta conquista e toda a minha trajetória acadêmica, primeiramente, a Deus, por ter me concedido força, sabedoria e perseverança para superar os desafios e alcançar este objetivo. Agradeço à minha mãe, Roselma de Sousa Oliveira Cunha, por todo amor, dedicação e apoio incondicional ao longo da minha vida. Ao meu pai, Julio Dias da Cunha, meu maior ídolo e referência como homem e pessoa, por todos os ensinamentos, valores e exemplos que carrego comigo. À minha madrinha, Elen Fabiane Guimarães Herval, expresso minha profunda gratidão por ter me possibilitado continuar meus estudos acadêmicos em um momento de dificuldade, tornando esta conquista possível. Ao meu padrinho, Marcosuel Souza, por ser uma grande inspiração profissional e de liderança, servindo como exemplo de dedicação, responsabilidade e comprometimento ao longo de sua trajetória. Ao meu primo, Fernando Oliveira Mendes, por ser uma grande inspiração profissional, um exemplo de dedicação e excelência, além de ter sido uma das pessoas que me inspiraram a ingressar no ensino superior. À minha prima, Karina Oliveira Mendes, por sua dedicação aos estudos e por ter sido uma grande inspiração para minha trajetória acadêmica, demonstrando, por meio de seu exemplo, a importância do conhecimento, da disciplina e da perseverança. Aos meus amigos, especialmente Diogo Ribeiro, Kauã Caldeira e Matheus Ramos, que, além de serem amigos que levarei para toda a vida, sempre serviram como fonte de inspiração, incentivo e aprendizado,

especialmente nos momentos em que os via conquistando suas oportunidades de estágio e dando os primeiros passos em suas carreiras profissionais. Agradeço também a todos os professores que fizeram parte da minha formação, compartilhando conhecimentos e contribuindo para meu crescimento acadêmico e pessoal. Por fim, agradeço ao meu orientador, professor Elton Figueiredo da Silva, pelas orientações e pelos necessários "puxões de orelha", que foram fundamentais para a realização deste trabalho e para o meu desenvolvimento acadêmico. A todos que contribuíram, direta ou indiretamente, para a realização deste sonho, o meu mais sincero agradecimento.

Eu, Maria Eduarda de Paula Amaral, gostaria de prestar meus agradecimentos a toda a minha família, que sempre me apoiou e foi suporte durante todo o meu período da faculdade, em especial para minha mãe Sandra Maria Macêdo de Paula, por ter sido a minha base para passar por todas as dificuldades que enfrentei pelo caminho, sem ela não teria sido possível chegar até onde cheguei. Gostaria de prestigiar todos os meus professores, que foram excelentes em me auxiliar e ensinar; o apoio deles foi fundamental para eu me tornar a melhor profissional que posso ser. Vale ressaltar meu orientador Elton Figueiredo da Silva, que foi muito além de um excelente orientador e professor, sendo uma grande referência com quem tive a oportunidade de aprender muito ao longo dos anos. Agradeço a todos os amigos que me ajudaram de forma direta ou indireta durante toda a minha trajetória, à equipe deste projeto por toda a sua dedicação, e principalmente a Deus por me dar forças de nunca desistir.

“Seja a mudança que você quer ver no mundo.”

(Mahatma Gandhi)

Resumo

A crescente complexidade dos sistemas de software, aliada à presença significativa de sistemas legados, tem intensificado os desafios relacionados à documentação, frequentemente inexistente, incompleta ou desatualizada. Esse cenário impacta diretamente a manutenção, evolução e compreensão dos sistemas, tornando o processo mais oneroso e suscetível a erros. Este trabalho propõe a utilização de Modelos de Linguagem de Grande Escala (LLMs) como suporte à reengenharia ágil de software, com foco na geração e atualização automatizada da documentação a partir de código-fonte escrito na linguagem Java. A abordagem consiste na aplicação de técnicas baseadas em LLMs para análise de código Java e extração de informações relevantes, transformando estruturas complexas em documentação compreensível e organizada. O objetivo deste estudo é apresentar um processo que integre práticas ágeis à utilização de Inteligência Artificial, visando aumentar a eficiência, reduzir o esforço manual e melhorar a qualidade da documentação produzida. Como resultados obtidos, destaca-se a capacidade de automatizar a geração de documentação técnica, reduzir o tempo necessário para compreensão de sistemas existentes e apoiar equipes de desenvolvimento na tomada de decisões. Por fim, são discutidas as limitações da abordagem, incluindo a possibilidade de inconsistências geradas pelos modelos, a necessidade de validação humana e questões relacionadas à segurança e privacidade do código analisado.

Palavras-Chave: Reengenharia de Software. Documentação de Software. Metodologias Ágeis. Modelos de Linguagem de Grande Escala. Inteligência Artificial.

Abstract

The increasing complexity of software systems, combined with the significant presence of legacy systems, has intensified challenges related to software documentation, which is often nonexistent, incomplete, or outdated. This scenario directly impacts system maintenance, evolution, and comprehension, making these processes more costly and error-prone. This study proposes the use of Large Language Models (LLMs) as support for agile software reengineering, focusing on the automated generation and updating of documentation from source code written in the Java language. The approach consists of applying LLM-based techniques to analyze Java code and extract relevant information, transforming complex structures into clear and organized documentation. The objective of this work is to present a process that integrates agile practices with Artificial Intelligence, aiming to increase efficiency, reduce manual effort, and improve the quality of the generated documentation. The results include the ability to automate technical documentation generation, reduce the time required to understand existing systems, and support development teams in decision-making. Finally, the study discusses the limitations of the approach, including potential inconsistencies generated by the models, the need for human validation, and concerns related to security and code privacy.

Keywords: Software Reengineering. Software Documentation. Agile Methodologies. Large Language Models. Artificial Intelligence.

Lista de Figuras

Figura 1 – Metodologia de pesquisa adotada no desenvolvimento e avaliação do <i>docScrum</i>	24
Figura 2 – Visão em camadas da arquitetura do <i>docScrum</i>	74
Figura 3 – Fluxo de funcionamento do <i>docScrum</i>	86
Figura 4 – Tela inicial do sistema <i>docScrum</i>	109
Figura 5 – Tela de análise do projeto Java em estado inicial.	110
Figura 6 – Tela de análise com pasta de projeto Java selecionada.	111
Figura 7 – Diálogo de confirmação do navegador para envio em lote.	111
Figura 8 – Tela de processamento inicial: leitura e interpretação do código-fonte pela IA.	112
Figura 9 – Documentação técnica gerada a partir da análise do código Java. .	113
Figura 10 – Requisitos funcionais e não funcionais extraídos do código-fonte. .	114
Figura 11 – <i>Product Backlog</i> gerado a partir da análise do código Java.	115
Figura 12 – Planejamento de <i>Sprints</i> gerado pelo sistema.	116
Figura 13 – Processamento da etapa de avaliação pelo juiz fundamentado no Guia do Scrum.	117
Figura 14 – Avaliação inicial pelo juiz: nota 3.7/10 antes da correção fundamentada.	118
Figura 15 – Processamento da etapa de correção pelo revisor com base no <i>feedback</i> do juiz.	119
Figura 16 – Avaliação pós-correção: nota 9.7/10 após o ciclo de auto-crítica fundamentada.	120
Figura 17 – Capa do documento consolidado em PDF.	121
Figura 18 – Sumário do documento consolidado em PDF.	121

Lista de Tabelas

Tabela 1 – Comparativo entre SaaS e Execução Local de LLMs.	36
Tabela 2 – Comparativo conceitual entre a geração <i>one-shot</i> e o ciclo de auto-crítica iterativa.	38
Tabela 3 – Protocolo de pesquisa baseado em Kitchenham (2004).	45
Tabela 4 – Mapeamento de Lacunas e Diferencial Estratégico.	50
Tabela 5 – Levantamento inicial de Modelos de Inteligência Artificial sob a restrição orçamentária.	53
Tabela 6 – Resultados do Benchmark Artificial Analysis para os Modelos Finalistas.	57
Tabela 7 – Métricas de Desempenho, Custo e Latência.	58
Tabela 8 – Requisitos funcionais do docScrum.	72
Tabela 9 – Requisitos não funcionais do docScrum.	73
Tabela 10 – Tecnologias utilizadas no <i>frontend</i>	76
Tabela 11 – Tecnologias utilizadas no <i>backend</i>	77
Tabela 12 – Parâmetros de integração com o provedor de LLM adotado.	77
Tabela 13 – Características quantitativas do projeto-alvo.	92
Tabela 14 – Notas médias gerais (escala 0–10) obtidas em cada execução.	94
Tabela 15 – Notas médias por dimensão ($N = 5$).	95
Tabela 16 – Trechos do Guia do Scrum mais citados na avaliação do <i>Backlog</i>	96
Tabela 17 – Trechos do Guia do Scrum mais citados na avaliação do <i>Sprints</i>	96
Tabela 18 – Custo computacional do experimento ($N = 5$, modelo DeepSeek R1).	97

Lista de Siglas e Abreviaturas

ACM	<i>Association for Computing Machinery</i>
API	<i>Application Programming Interface</i>
BDD	<i>Behaviour-Driven Development</i>
BES	Bacharelado em Engenharia de Software
BPE	<i>Byte Pair Encoding</i>
CI/CD	<i>Continuous Integration / Continuous Delivery</i>
CORS	<i>Cross-Origin Resource Sharing</i>
CSS	<i>Cascading Style Sheets</i>
DDD	<i>Domain-Driven Design</i>
DI	<i>Dependency Injection</i>
DIP	<i>Dependency Inversion Principle</i>
DoD	<i>Definition of Done</i>
DOM	<i>Document Object Model</i>
DTO	<i>Data Transfer Object</i>
GGUF	<i>GPT-Generated Unified Format</i>
GPT	<i>Generative Pre-trained Transformer</i>
GQM	<i>Goal Question Metric</i>
HTML	<i>HyperText Markup Language</i>
HTTP	<i>Hypertext Transfer Protocol</i>
HTTPS	<i>Hypertext Transfer Protocol Secure</i>
IA	Inteligência Artificial
IEEE	<i>Institute of Electrical and Electronics Engineers</i>
I/O	<i>Input/Output</i>
ISO/IEC	<i>International Organization for Standardization / International Electrotechnical Commission</i>
JSON	<i>JavaScript Object Notation</i>
JVM	<i>Java Virtual Machine</i>
JWT	<i>JSON Web Token</i>
LGPD	Lei Geral de Proteção de Dados
LLM	<i>Large Language Model</i>
LSP	<i>Liskov Substitution Principle</i>
LSTM	<i>Long Short-Term Memory</i>
MDE	<i>Model-Driven Engineering</i>
MMLU-Pro	<i>Massive Multitask Language Understanding – Professional</i>
OCP	<i>Open/Closed Principle</i>

OOV	<i>Out-of-Vocabulary</i>
PDF	<i>Portable Document Format</i>
PLN	Processamento de Linguagem Natural
QA	<i>Quality Assurance</i>
RAG	<i>Retrieval-Augmented Generation</i>
REST	<i>Representational State Transfer</i>
RF	Requisito Funcional
RNF	Requisito Não Funcional
RNN	<i>Recurrent Neural Network</i>
SaaS	<i>Software as a Service</i>
SDK	<i>Software Development Kit</i>
SDLC	<i>Software Development Life Cycle</i>
SOLID	<i>Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion</i>
SPA	<i>Single Page Application</i>
SRP	<i>Single Responsibility Principle</i>
TCC	Trabalho de Conclusão de Curso
TDD	<i>Test-Driven Development</i>
URL	<i>Uniform Resource Locator</i>
VRAM	<i>Video RAM</i>
WIP	<i>Work in Progress</i>
XP	<i>Extreme Programming</i>

Sumário

1	INTRODUÇÃO	20
1.1	Situação/Problema	21
1.2	Questões de Pesquisa	22
1.3	Objetivos	22
1.3.1	Objetivos Específicos	22
1.4	Metodologia a ser seguida	23
1.5	Estrutura do Documento	24
2	FUNDAMENTAÇÃO TEÓRICA	26
2.1	Reengenharia de Software	26
2.2	Metodologias Ágeis	27
2.3	Conceitos e Pilares do Scrum	28
2.3.1	Artefatos e Compromissos	29
2.3.1.1	Product Backlog (Compromisso: Product Goal)	30
2.3.1.2	Sprint Backlog (Compromisso: Sprint Goal)	30
2.3.1.3	Increment (Compromisso: Definition of Done)	30
2.4	Modelos de Linguagem de Grande Escala (LLMs)	31
2.4.1	Arquitetura Transformer e Autoatenção	32
2.4.2	Janela de Contexto	33
2.4.3	Parâmetros de Configuração e Controle de Inferência	34
2.4.3.1	Temperatura e a Modulação da Confiança Estatística	34
2.4.4	Paradigmas de Execução: APIs de Nuvem vs. Modelos Locais	35
2.4.4.1	Segurança e Riscos das API Keys	35
2.4.4.2	Execução Local com Ferramentas como Ollama	36
2.4.5	Alucinações e Autocrítica Fundamentada	37
2.4.6	Estratégias de Injeção de Contexto: <i>Context Stuffing</i> e RAG	38
2.5	Java	39
2.6	NestJS	40
2.7	React	40
2.8	Estudo de Caso na Reengenharia Assistida por Inteligência Artificial	41
2.9	Considerações sobre o Capítulo	42
3	TRABALHOS RELACIONADOS	44
3.1	Procedimento de Pesquisa Bibliográfica	44
3.2	Trabalhos Seleccionados	46

3.2.1	Assistência na Geração de Documentação (Madruga, 2025)	46
3.2.2	Inteligência Artificial em Sistemas Legados (Araújo, 2025)	46
3.2.3	Automação Segura e Arquitetura RAG (Rotilli e Silva, 2025)	47
3.2.4	Refatoração Assistida por LLM (Freitas, 2025)	48
3.2.5	Modelagem Visual e Diagramas de Arquitetura (Hatahet et al., 2025)	48
3.2.6	Engenharia de Modelos e Orquestração Agêntica (Di Rocco et al., 2025)	49
3.3	Quadro Comparativo e Justificação do Diferencial Acadêmico	49
3.4	Considerações sobre o Capítulo	50
4	METODOLOGIA DE SELEÇÃO DE LLM	52
4.1	Levantamento Inicial dos Modelos	52
4.1.1	Filtragem e Seleção dos Modelos Candidatos	54
4.1.2	Estratégia de Avaliação Prática	54
4.1.2.1	Testes e Avaliação: IA de Classe	55
4.1.2.2	Definição das Classes de Teste	56
4.1.2.3	Análise de Desempenho em Codificação: Artificial Analysis	56
4.1.2.4	Análise de Desempenho e Viabilidade: Benched.ai	58
4.1.2.5	Ensaio Prático: Avaliação de Refatoração em Nível de Classe	59
4.1.2.6	Metodologia de Análise dos Resultados	60
4.2	Resultados e Discussão	60
4.2.1	CrITÉRIOS de Avaliação de Resolução	60
4.2.2	Análise dos Ensaio (NÍVEL de Classe)	61
4.2.2.1	Conclusão da Avaliação de NÍVEL de Classe	67
4.2.2.2	Unificação da Estratégia e Redefinição de Escopo	68
4.3	Conclusão da Seleção do Modelo	68
5	O DOCS CRUM	70
5.1	Visão Geral	70
5.2	Objetivos do Sistema	71
5.2.1	Objetivo Geral	71
5.2.2	Objetivos Específicos	71
5.3	Requisitos do Sistema	72
5.3.1	Requisitos Funcionais	72
5.3.2	Requisitos Não Funcionais	73
5.4	Arquitetura do Sistema	73
5.4.1	Padrões Arquiteturais	75
5.5	Tecnologias Utilizadas	75
5.5.1	Frontend	76
5.5.2	Backend	76
5.5.3	Provedor de LLM Adotado	76

5.5.4	Estratégia de <i>Prompting</i>	78
5.6	Componentes Principais	79
5.6.1	Camada de Interface	79
5.6.1.1	Página Inicial	79
5.6.1.2	Página de Análise	79
5.6.1.3	Componente de Upload	80
5.6.1.4	Componente de Resultados	80
5.6.1.5	Painel de Validação Scrum	82
5.6.1.6	Cabeçalho e Sistema de Tema	82
5.6.2	Camada de Aplicação	83
5.6.2.1	Módulo de Análise	83
5.6.2.2	Módulo de Validação	84
5.6.2.3	<i>Parsing</i> da Resposta de Análise	84
5.6.2.4	Inicialização e Configuração do Servidor	85
5.7	Fluxo de Funcionamento	85
5.8	Artefatos Gerados	87
5.9	Considerações de Implementação	89
5.10	Considerações sobre o Capítulo	90
6	PROTOCOLO DO ESTUDO DE CASO	91
6.1	Design do Estudo e Objetivos (GQM)	91
6.2	Unidade de Análise	92
6.3	Procedimento de Coleta e Protocolo de Execução	92
6.4	Análise de Resultados	94
6.4.1	Notas Individuais por Execução	94
6.4.2	Notas Médias por Dimensão	94
6.4.3	Mapeamento das Citações do Guia do Scrum	95
6.4.4	Custo Computacional	97
6.5	Discussão Crítica	97
6.5.1	Resposta às Questões de Pesquisa	97
6.5.2	Contribuição Metodológica	98
6.6	Ameaças à Validade e Limitações	98
6.7	Considerações sobre o Capítulo	100
7	CONSIDERAÇÕES FINAIS	101
7.1	Resposta às Questões de Pesquisa	102
7.1.1	Q1: Eficácia do Ciclo de Geração, Validação e Correção	102
7.1.2	Q2: Viabilidade Orçamentária	102
7.1.3	Q3: Impacto Quantitativo do Ciclo de Correção	102
7.2	Limitações	103

7.3 Trabalhos Futuros	103
REFERÊNCIAS	105
APÊNDICE A – DEMONSTRAÇÃO DA APLICAÇÃO	109

1 Introdução

A documentação técnica de software configura infraestrutura essencial para a sustentabilidade organizacional, funcionando como interface crítica entre código executável, desenvolvedores e tomadores de decisão estratégicos (AGHAJANI et al., 2020). Historicamente, a gestão desse conhecimento enfrentou extremos: se por um lado as metodologias tradicionais geravam uma “sobrecarga de documentação” que se tornava um obstáculo operacional (NAGY et al., 2019), o advento do Manifesto Ágil em 2001 trouxe o problema oposto ao priorizar o software em funcionamento (NAGAI; SBRA-GIA, 2023). Sob a ótica ágil, a documentação frequentemente torna-se incompleta, ambígua ou obsoleta, resultando em negligência documental e no aumento direto do custo de manutenção (CANEDO et al., 2024).

Tendo em vista que o código-fonte constitui o último artefato gerado no ciclo de desenvolvimento, à Engenharia Reversa é empregada como um processo para analisá-lo e extrair o design e as regras de negócio originais. Contudo, embora seja uma abordagem eficaz para a recuperação de conhecimento técnico, trata-se de um procedimento tradicionalmente lento e manual (MOHOTTIGE et al., 2025). Diante dessa limitação operacional, o surgimento da Inteligência Artificial Generativa e dos Modelos de Linguagem de Grande Escala (LLM) alterou esse cenário (FAN et al., 2023).

Naimi, Kacem e Tabbane (2024) demonstra que o uso de LLMs permite automatizar a criação de artefatos técnicos, como casos de uso e resumos de código, com elevada precisão. Complementarmente, estudos recentes reportam acerto de 82–87% na geração automática de documentação técnica a partir de códigos sem comentários (GUPTA et al., 2024). Essa evolução tecnológica permite a adoção de uma Reengenharia Ágil, que promove a modernização incremental e a evolução contínua de sistemas legados em paralelo às demandas de negócio (LANO; ALFRAIHI, 2025).

A integração dessas capacidades computacionais aos fluxos de reengenharia configura uma fronteira estratégica de pesquisa. Ao unir a capacidade analítica da Engenharia Reversa com a agilidade dos LLMs, este trabalho propõe uma abordagem capaz de mitigar a dívida técnica documental, recuperando a integridade do conhecimento institucional sem gerar sobrecarga operacional.

A contribuição metodológica central deste trabalho, no entanto, transcende a simples geração automatizada: consiste no desenho de um *ciclo iterativo de geração, validação e correção* no qual o próprio LLM desempenha, em chamadas separadas e

com *prompts* especializados, três papéis distintos: *gerador* na etapa de análise, *juiz fundamentado em documento normativo* na etapa de validação contra o Guia do Scrum 2020 e *revisor orientado por citação literal* na etapa de correção. A separação explícita desses papéis materializa um padrão de *autocrítica auditável* cuja eficácia é demonstrada empiricamente, com replicação em cinco execuções independentes, no estudo de caso experimental que constitui o núcleo prático deste trabalho.

1.1 Situação/Problema

A raiz do problema que este trabalho busca enfrentar está no abismo entre a agilidade necessária para o negócio e a preservação do conhecimento técnico. No dia a dia do desenvolvimento, essa lacuna se manifesta em três grandes obstáculos:

Primeiramente, observa-se o peso da entrega sobre a memória técnica. Desde que o Manifesto Ágil estabeleceu que software funcional possui prioridade sobre a documentação abrangente, criou-se um efeito colateral não planejado. De acordo com Canedo et al. (2024), a pressão por entregas constantes acaba negligenciando a documentação, gerando uma “Dívida de Documentação”. O resultado é um sistema que evolui tecnicamente, mas cuja inteligência de negócio se torna opaca.

Somado a isso, nota-se o dilema operacional entre documentar ou produzir. Manter registros técnicos atualizados nos moldes tradicionais tornou-se uma tarefa inviável no ritmo atual. Nagy et al. (2019) destacam que tentativas de documentação exaustiva levam a uma sobrecarga, consumindo o tempo que deveria ser gasto em inovação. Isso torna a documentação um obstáculo operacional em vez de um facilitador.

Por fim, há o alto custo da recuperação manual de conhecimento. Quando os registros inexistem, a única saída é a engenharia reversa. No entanto, Mohottige et al. (2025) explicam que o método clássico para analisar código e extrair regras de negócio é um processo predominantemente lento e manual. Isso transforma sistemas legados em ativos de alto risco, nos quais qualquer alteração exige um esforço desproporcional de análise.

Diante desse cenário, emerge a seguinte questão: *Como a integração de Modelos de Linguagem de Grande Escala (LLMs) pode automatizar a engenharia reversa de sistemas legados, eliminando a dívida técnica de documentação sem gerar sobrecarga operacional às equipes?*

1.2 Questões de Pesquisa

A partir dos objetivos delineados, esta pesquisa busca responder às seguintes questões:

1. **Q1:** Qual a eficácia do ciclo de geração, validação e correção fundamentada do *docScrum* na produção de artefatos de *Backlog* e *Sprint Planning* com aderência mensurável ao Guia do Scrum 2020, medida em três dimensões (*Conformidade Estrutural*, *Cobertura Conceitual* e *Aderência Semântica*)?
2. **Q2:** O ensaio comparativo entre modelos LLM candidatos permite identificar pelo menos um modelo que atenda simultaneamente ao teto orçamentário de US\$ 3,00 por milhão de *tokens* e à qualidade técnica exigida para a geração e o refinamento de artefatos ágeis a partir de código Java?
3. **Q3:** Qual o impacto quantitativo do ciclo de correção fundamentada sobre as notas iniciais dos artefatos gerados em *one-shot*, medido em variação média de pontuação e em estabilidade entre execuções independentes?

1.3 Objetivos

Este trabalho propõe-se a avaliar o sistema *docScrum* como ferramenta de reengenharia ágil assistida por LLM, nos seguintes contextos:

- Automatizar a geração de artefatos ágeis (documentação, requisitos, *Backlog* e *Sprints*) a partir de código Java.
- Avaliar a aderência dos artefatos gerados ao Guia do Scrum 2020 oficial.
- Demonstrar empiricamente a eficácia do ciclo de autocrítica fundamentada do modelo, no qual o LLM atua como gerador, juiz e revisor.
- Manter o custo operacional do *pipeline* dentro do teto de US\$ 3,00 por milhão de *tokens* processados.

1.3.1 Objetivos Específicos

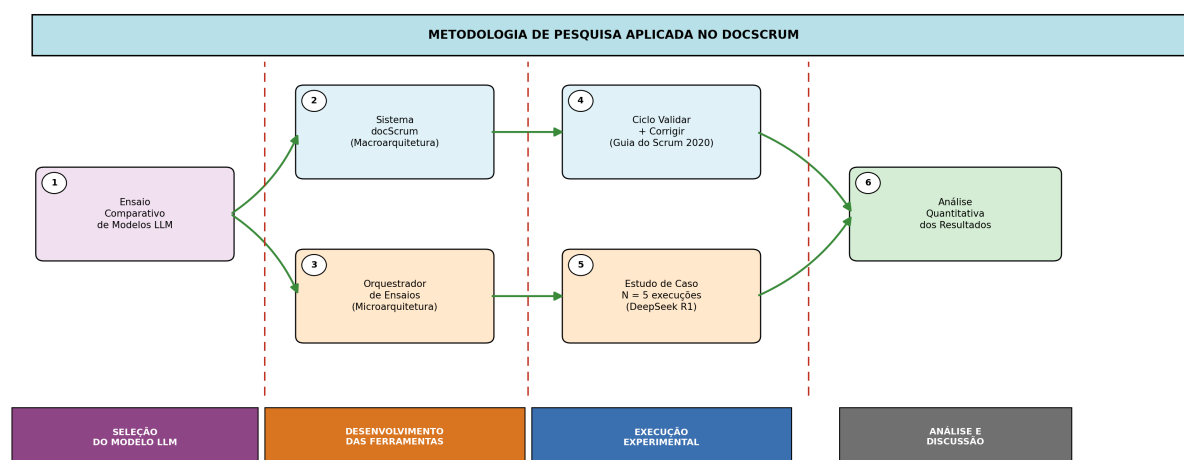
Para apoiar a realização do objetivo proposto, os seguintes objetivos específicos foram definidos:

- Conduzir ensaio comparativo entre modelos candidatos a provedor de LLM sob restrição orçamentária, identificando o modelo mais adequado ao escopo da pesquisa.
- Implementar um *pipeline* de geração automatizada dos quatro artefatos ágeis a partir de código Java.
- Desenvolver um serviço de validação fundamentada contra o Guia do Scrum 2020 em três dimensões, com citação literal do documento normativo como justificativa.
- Implementar um mecanismo de correção automática orientado pelo *feedback* estruturado da validação, vedando a introdução de elementos sem fundamentação.
- Avaliar experimentalmente, em $N = 5$ execuções independentes sobre um projeto Java real, o impacto do ciclo de correção sobre as notas iniciais dos artefatos.

1.4 Metodologia a ser seguida

Para obtenção dos objetivos específicos da pesquisa, foi utilizada uma metodologia experimental e aplicada, organizada em quatro fases sequenciais (Seleção do Modelo, Desenvolvimento das Ferramentas, Execução Experimental e Análise) que se materializam em duas frentes complementares (macro e microarquitetura) e convergem para uma etapa única de análise quantitativa. As atividades desenvolvidas neste trabalho são: (1) Conduzir ensaio comparativo entre modelos LLM candidatos sob restrição orçamentária de US\$ 3,00 por milhão de *tokens*, do qual resultou a seleção do DeepSeek R1; (2) Desenvolver o sistema *docScrum* como aplicação web responsável pela geração automatizada dos quatro artefatos ágeis a partir de código Java; (3) Desenvolver o orquestrador de ensaios em Python para automação da coleta de métricas e registro auditável das respostas integrais do modelo; (4) Implementar o ciclo de validação e correção fundamentada contra o Guia do Scrum 2020, aplicado aos artefatos de *Backlog* e *Sprint Planning*; (5) Executar estudo de caso replicado em $N = 5$ execuções independentes sobre um projeto Java acadêmico real, com coleta automatizada de *tokens*, latência e *checkpoints* JSON de toda resposta gerada; e (6) Realizar a análise quantitativa consolidada dos resultados, reunindo as evidências empíricas sobre a eficácia do ciclo de autocrítica fundamentada e sobre a viabilidade operacional da ferramenta. A Figura 1 sintetiza graficamente o fluxo entre as atividades e o agrupamento por fases.

Figura 1 – Metodologia de pesquisa adotada no desenvolvimento e avaliação do *docScrum*.



Fonte: autoria própria.

1.5 Estrutura do Documento

Este trabalho está organizado em sete capítulos e um apêndice, estruturados da seguinte forma:

O **Capítulo 1** Apresenta a introdução, descrevendo o contexto da pesquisa, o desafio central referente ao abismo entre a agilidade e a preservação do conhecimento técnico, bem como os objetivos gerais e específicos, as questões de pesquisa e a metodologia experimental adotada.

O **Capítulo 2** aborda a fundamentação teórica, detalhando os conceitos de Reengenharia de Software, o funcionamento das Metodologias Ágeis (com ênfase no *framework* Scrum), os fundamentos dos LLMs e as tecnologias que compõem o sistema (Java¹, NestJS² e React³).

O **Capítulo 3** expõe os Trabalhos Relacionados, estabelecendo diálogo crítico com autores que investigam a aplicação de LLMs à engenharia de software e identificando, por meio de matriz comparativa, as lacunas que o presente trabalho preenche.

O **Capítulo 4** apresenta a metodologia de seleção do modelo de LLM, detalhando o levantamento inicial dos candidatos sob restrição orçamentária, a filtragem por critérios de *benchmarking* (*Artificial Analysis* e *Benched.ai*), os ensaios práticos de refatoração estrutural em classes Java e a justificativa formal da escolha do DeepSeek R1 como modelo definitivo.

O **Capítulo 5** descreve o sistema *docScrum* em três planos complementares: o

conceitual (objetivos e requisitos), o arquitetural (camadas, módulos e padrões adotados) e o de implementação (componentes principais, fluxo de execução e decisões técnicas relevantes).

O **Capítulo 6** apresenta o protocolo e os resultados do estudo de caso experimental, com replicação em $N = 5$ execuções independentes sobre um projeto Java real, abrangendo as notas pré e pós-correção em três dimensões, o mapeamento das citações do Guia do Scrum, o custo computacional agregado e a discussão das ameaças à validade.

O **Capítulo 7** sintetiza os resultados obtidos nas Considerações Finais, responde às questões de pesquisa formuladas neste capítulo, discute as limitações observadas durante a execução e propõe caminhos para pesquisas futuras.

Por fim, o **Apêndice A** apresenta a demonstração da aplicação, percorrendo as principais telas do sistema em funcionamento e descrevendo os elementos visuais que compõem o fluxo de uso, do carregamento do código-fonte Java à entrega final dos artefatos ágeis.

2 Fundamentação Teórica

Neste capítulo serão apresentados os conceitos de reengenharia de software (Seção 2.1), metodologias ágeis (Seção 2.2), Scrum (Seção 2.3) e Modelos de Linguagem de Grande Escala (LLMs) (Seção 2.4), de modo a oferecer suporte à compreensão das bases teóricas e tecnológicas que sustentam este trabalho.

2.1 Reengenharia de Software

Manter um sistema legado em operação sem documentação atualizada é uma das situações mais críticas enfrentadas pelas organizações, pois o conhecimento sobre o comportamento do sistema permanece implícito no código-fonte, dificultando manutenções e evoluções (AGHAJANI et al., 2020). A reengenharia de software surge nesse cenário não apenas como uma reforma técnica do código, mas como uma estratégia para recuperar o valor de negócio de sistemas que se tornaram difíceis de manter. Para Mohottige et al. (2025), os métodos tradicionais de reengenharia (agrupados em abordagens estática, dinâmica, híbrida e baseada em artefatos) ainda enfrentam desafios significativos para escalar diante do tamanho e da complexidade dos sistemas legados contemporâneos.

O processo clássico de reengenharia, composto pela engenharia reversa para compreender o legado e pela engenharia avante para modernizá-lo, esbarra em um custo elevado quando conduzido de forma predominantemente manual (MOHOTTIGE et al., 2025; PIETRINI et al., 2025). Analisar grandes volumes de código em linguagens obsoletas exige profissionais cada vez mais escassos, o que prolonga prazos e eleva os riscos do projeto (PIETRINI et al., 2025).

Nesse contexto, os LLMs têm sido investigados como uma alternativa às ferramentas de automação baseadas em regras, por oferecerem capacidade de compreensão semântica do código (FAN et al., 2023; HOU et al., 2024). Estudos recentes indicam que LLMs podem apoiar atividades como geração de documentação, sumarização de código e recuperação de especificações a partir de implementações em Java (LANO; ALFRAIHI, 2025; MACEDO et al., 2024). Aplicados à reengenharia, esses modelos têm potencial para reduzir o esforço manual envolvido na compreensão de sistemas legados, embora a literatura também aponte limitações relevantes, como perda de contexto em bases de código extensas e necessidade de validação humana das saídas geradas (MACEDO et al., 2024).

2.2 Metodologias Ágeis

As metodologias ágeis surgiram como uma resposta aos modelos tradicionais de desenvolvimento, como o Modelo em Cascata, percebidos como excessivamente rígidos e burocráticos diante da volatilidade dos requisitos de software (SASSA et al., 2023). O marco inicial dessa mudança foi o Manifesto para Desenvolvimento Ágil de Software, publicado em 2001, que estabeleceu quatro valores fundamentais: indivíduos e interações mais que processos e ferramentas; software em funcionamento mais que documentação abrangente; colaboração com o cliente mais que negociação de contratos; e responder a mudanças mais que seguir um plano (BECK et al., 2001).

Na prática, a agilidade não se resume a um único método, mas a um conjunto de frameworks que implementam esses princípios com ênfases distintas no ciclo de vida do software:

- **Scrum:** Framework amplamente adotado para a gestão de produtos complexos. Organiza o desenvolvimento em ciclos iterativos chamados *Sprints*, com responsabilidades definidas (*Scrum Master*, *Product Owner* e *Developers*) e eventos formais que sustentam os pilares empíricos de transparência, inspeção e adaptação (SCHWABER; SUTHERLAND, 2020; SASSA et al., 2023).
- **Extreme Programming (XP):** Concentra-se na excelência técnica e na qualidade do código, propondo práticas como Programação em Par, Desenvolvimento Orientado a Testes (TDD), Integração Contínua e Refatoração contínua (SHRI-VASTAVA et al., 2021).
- **Kanban:** Inspirado no Sistema Toyota de Produção, foca na visualização do fluxo de trabalho e na limitação do trabalho em progresso (*Work in Progress – WIP*), com o objetivo de identificar gargalos e promover melhoria evolutiva incremental (DAMIJ; DAMIJ, 2024).
- **Lean Software Development:** Adapta os princípios do Lean Manufacturing ao desenvolvimento de software, organizados em sete princípios, entre eles: eliminar desperdícios, amplificar o aprendizado e entregar o mais rápido possível (ALDAEEJ; NGUYEN-DUC; GUPTA, 2023).

Apesar da diversidade de abordagens, todas as metodologias ágeis compartilham um desafio recorrente: a manutenção de documentação técnica adequada ao longo do ciclo de entrega. Pesquisas empíricas com profissionais de desenvolvimento ágil indicam que, em ambientes de entrega contínua, o esforço de documentação tende a

competir com a produção de novas funcionalidades, resultando em registros incompletos ou desatualizados (CANEDO et al., 2024; AGHAJANI et al., 2020). Esse fenômeno tem sido descrito na literatura como *documentation technical debt*, um tipo específico de dívida técnica cujas principais consequências são retrabalho e déficit de conhecimento na equipe (CANEDO et al., 2024; ALDAEEJ; NGUYEN-DUC; GUPTA, 2023).

É importante destacar que o Manifesto Ágil não advoga a ausência de documentação, mas sim sua subordinação ao software funcionando (BECK et al., 2001). Para que a agilidade não comprometa a engenharia de longo prazo, é necessário que os processos de registro e compreensão do software acompanhem a velocidade das entregas. É nesse cenário, de busca de equilíbrio entre agilidade e rigor técnico, que este trabalho se apoia.

Por ser a base operacional desta pesquisa, o Scrum será explorado em detalhes na próxima seção, na qual analisamos como seus eventos e artefatos permitem a integração da solução de documentação proposta.

2.3 Conceitos e Pilares do Scrum

A fundamentação desta seção baseia-se principalmente no *Scrum Guide* (SCHWABER; SUTHERLAND, 2020), documento mantido pelos criadores do framework, complementado por estudos acadêmicos recentes que investigam empiricamente sua adoção e desafios práticos (HRON; OBWEGESER, 2022; SASSA et al., 2023). O Scrum é definido como uma estrutura leve projetada para auxiliar pessoas, equipes e organizações na geração de valor por meio de soluções adaptativas para problemas complexos (SCHWABER; SUTHERLAND, 2020). Diferente de metodologias prescritivas, o Scrum é propositadamente incompleto, estabelecendo apenas as partes necessárias para implementar a sua teoria. Suas regras não fornecem instruções detalhadas, mas guiam as relações e interações entre os membros da equipe.

Pesquisas empíricas indicam que o Scrum é hoje o framework ágil mais adotado em projetos de software, com aplicação documentada em diferentes domínios e portes de equipe (SASSA et al., 2023; HRON; OBWEGESER, 2022).

A teoria do framework sustenta-se em dois conceitos principais:

- **Empirismo:** Afirma que o conhecimento provém da experiência e que as decisões devem ser tomadas com base no que é observado (SCHWABER; SUTHERLAND, 2020).

- **Pensamento Lean:** Foca na redução de desperdícios e na concentração de esforços apenas no que é essencial (SCHWABER; SUTHERLAND, 2020; AL-DAEEJ; NGUYEN-DUC; GUPTA, 2023).

Para otimizar a previsibilidade e mitigar riscos, o Scrum utiliza uma abordagem iterativa e incremental (SCHWABER; SUTHERLAND, 2020). Essa estrutura operacional é viabilizada pela implementação de três pilares empíricos:

1. **Transparência:** Exige que o processo emergente e o trabalho sejam visíveis tanto para quem executa quanto para quem recebe os resultados. Artefatos com baixa transparência podem levar a decisões que diminuem o valor e aumentam o risco (SCHWABER; SUTHERLAND, 2020).
2. **Inspeção:** Os artefatos do Scrum e o progresso rumo aos objetivos acordados devem ser inspecionados de forma frequente e diligente para detectar variâncias ou problemas indesejáveis. A inspeção fornece a cadência necessária para a mudança, suportada pelos cinco eventos do framework (SCHWABER; SUTHERLAND, 2020).
3. **Adaptação:** Caso algum aspecto de um processo se desvie para fora dos limites aceitáveis ou se o produto resultante for inaceitável, os ajustes devem ser realizados o mais rapidamente possível para minimizar novos desvios (SCHWABER; SUTHERLAND, 2020).

Estudos empíricos recentes mostram, contudo, que a aplicação rigorosa desses pilares na prática é desafiadora. Hron e Obwegeser (2022) identificam que equipes frequentemente adaptam o Scrum, omitindo eventos ou artefatos, o que pode comprometer especialmente a inspeção e a transparência. Sassa et al. (2023) também apontam dificuldades de coordenação, autogestão e compartilhamento de conhecimento como barreiras recorrentes em projetos Scrum.

2.3.1 Artefatos e Compromissos

Os artefatos do Scrum representam trabalho ou valor e são concebidos para maximizar a transparência da informação chave (SCHWABER; SUTHERLAND, 2020). Cada artefato contém um compromisso para assegurar que fornece informação contra a qual o progresso pode ser medido. Esses compromissos existem para reforçar o empirismo e os valores do Scrum para a *Scrum Team* e seus *stakeholders*.

2.3.1.1 Product Backlog (Compromisso: Product Goal)

O *Product Backlog* é uma lista emergente e ordenada do que é necessário para melhorar o produto, sendo a única fonte de trabalho realizado pela *Scrum Team* (SCHWABER; SUTHERLAND, 2020). Os itens do *Product Backlog* adquirem transparência pelas atividades de refinamento, que consistem no ato contínuo de decompor e definir melhor os itens em partes menores e precisas, acrescentando detalhes como descrição, ordenação e tamanho. A formalização desses itens em formato de *user stories* é uma prática consolidada em equipes ágeis brasileiras (CANEDO et al., 2024). Embora a gestão do *Product Backlog* seja uma responsabilidade do *Product Owner*, são os *Developers* que realizam o trabalho e detêm a responsabilidade pelo seu dimensionamento (SCHWABER; SUTHERLAND, 2020). O *Product Owner* pode influenciar os *Developers* ajudando-os a compreender e a selecionar os itens.

O seu compromisso é o *Product Goal*, que descreve um estado futuro do produto e serve de meta a longo prazo para a *Scrum Team* delinear seu planejamento (SCHWABER; SUTHERLAND, 2020). Para manter o foco contínuo, a equipe deve cumprir (ou abandonar) um *Product Goal* antes de assumir o próximo.

2.3.1.2 Sprint Backlog (Compromisso: Sprint Goal)

O *Sprint Backlog* é composto pelo objetivo da *Sprint* (porquê), pelo conjunto de itens selecionados do *Product Backlog* (o quê) e por um plano acionável para a entrega do incremento (como) (SCHWABER; SUTHERLAND, 2020). Esse artefato é um plano criado exclusivamente de e para os *Developers*, funcionando como uma imagem altamente visível e em tempo real do trabalho planejado para a *Sprint*. Ele não é estático: o *Sprint Backlog* é atualizado durante a *Sprint* à medida que os *Developers* aprendem mais, devendo ter detalhe suficiente para que o progresso seja inspecionado diariamente na *Daily Scrum*.

O compromisso associado é o *Sprint Goal*, único objetivo da *Sprint*, que cria coerência e foco e encoraja a *Scrum Team* a trabalhar em conjunto em vez de em iniciativas separadas (SCHWABER; SUTHERLAND, 2020). Embora o *Sprint Goal* seja fixo, o trabalho exato necessário para alcançá-lo (o *Sprint Backlog*) oferece flexibilidade. Se o trabalho se revelar diferente do esperado durante a execução, os *Developers* colaboram com o *Product Owner* para negociar o escopo sem afetar o *Sprint Goal*.

2.3.1.3 Increment (Compromisso: Definition of Done)

O *Increment* é um degrau concreto em direção ao *Product Goal* (SCHWABER; SUTHERLAND, 2020). Para fornecer valor, ele deve ser utilizável e garantir que funciona em conjunto com todos os *Increments* anteriores. Podem ser criados múltiplos

Increments dentro de uma mesma *Sprint*. Além disso, um *Increment* pode ser entregue aos *stakeholders* antes mesmo do fim da *Sprint*, pois a *Sprint Review* nunca deve ser considerada como uma porta ou barreira para a entrega de valor.

O compromisso do *Increment* é a *Definition of Done* (DoD), descrição formal do estado do *Increment* quando este cumpre as medidas de qualidade exigidas para o produto (SCHWABER; SUTHERLAND, 2020). Um item do *Product Backlog* torna-se um *Increment* no exato momento em que satisfaz a DoD. O rigor da DoD exige que, se um item não a cumprir, ele não pode ser lançado nem apresentado na *Sprint Review*, regressando ao *Product Backlog* para consideração futura. Quanto à sua origem, se a DoD fizer parte dos padrões organizacionais, todas as *Scrum Teams* devem segui-la como um mínimo; caso contrário, a *Scrum Team* deve criar uma DoD apropriada para o seu produto.

Os três artefatos do Scrum (*Product Backlog*, *Sprint Backlog* e *Increment*), em conjunto com seus respectivos compromissos (*Product Goal*, *Sprint Goal* e *Definition of Done*), constituem o conjunto canônico de elementos que define a estrutura, os critérios de qualidade e a cadência de entrega do *framework*. Essa base normativa é o referencial contra o qual qualquer artefato Scrum pode ser avaliado quanto à sua aderência aos princípios do empirismo e do pensamento Lean discutidos no início desta seção.

2.4 Modelos de Linguagem de Grande Escala (LLMs)

Os Modelos de Linguagem de Grande Escala (LLMs, do inglês *Large Language Models*) representam uma evolução significativa no Processamento de Linguagem Natural (PLN) e na Inteligência Artificial Generativa (FAN et al., 2023; HOU et al., 2024). Diferentemente das abordagens anteriores baseadas em redes recorrentes ou modelos estatísticos, os LLMs utilizam redes neurais profundas baseadas na arquitetura *Transformer* para processar, compreender e gerar sequências complexas de texto e código-fonte com coerência (VASWANI et al., 2017).

Essa capacidade resulta do treinamento em grandes volumes de dados, permitindo que os modelos aprendam padrões sintáticos e semânticos de forma simultânea (HOU et al., 2024). Na Engenharia de Software, os LLMs oferecem capacidades promissoras para compreensão de código e processamento de estruturas computacionais, com aplicações documentadas em geração de documentação, refatoração e modernização de sistemas legados (FAN et al., 2023; MACEDO et al., 2024).

Para avaliar seu potencial real na reengenharia ágil, é necessário compreender os

fundamentos técnicos: desde a arquitetura *Transformer* que os sustenta até os mecanismos de atenção e os parâmetros de inferência que governam o comportamento desses modelos (VASWANI et al., 2017).

2.4.1 Arquitetura Transformer e Autoatenção

A arquitetura *Transformer*, introduzida por Vaswani et al. (2017) no artigo seminal “*Attention is All You Need*”, substitui o processamento sequencial das redes neurais recorrentes (RNNs) e LSTMs por paralelismo massivo via mecanismo de autoatenção. Diferentemente do processamento linear (*token por token*), a autoatenção calcula pesos de relevância entre todos os pares de *tokens* simultaneamente, independentemente da distância posicional entre eles.

A autoatenção projeta cada *token* em três vetores, *Query* (Q), *Key* (K) e *Value* (V), derivados por transformações lineares da entrada (VASWANI et al., 2017). A fórmula da Atenção por Produto Escalar Escalonado é:

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V \quad (2.1)$$

onde d_k é a dimensão das chaves. A equação se desdobra em quatro etapas:

1. QK^T (Produto Escalar): Calcula similaridade entre cada *Query* e todas as *Keys*, gerando escores de afinidade semântica entre *tokens*. Em código-fonte, identifica dependências como parênteses correspondentes distantes.
2. $\sqrt{d_k}$ (Escalonamento): Estabiliza variância dos escores dividindo pela raiz quadrada da dimensão das chaves, evitando saturação da *softmax* e garantindo gradientes estáveis em sequências longas.
3. *Softmax*: Normaliza escores em distribuição de probabilidade ($\sum = 1.0$), determinando pesos de atenção precisos: *tokens* relevantes recebem valores altos, irrelevantes aproximam-se de zero.
4. $\times V$: Aplica pesos à matriz de valores, produzindo representação contextual em que cada *token* incorpora informações ponderadas de toda a sequência.

Para a Engenharia de Software, essa capacidade permite que LLMs mantenham contexto global ao analisar arquivos extensos, compreendendo relações entre funções

distantes e estruturas declaradas em módulos separados (HOU et al., 2024). Na reengenharia, recupera dependências não-lineares, transformando código em representações semânticas analisáveis (MACEDO et al., 2024).

2.4.2 Janela de Contexto

A janela de contexto define o limite máximo de *tokens* que um modelo consegue processar e manter em sua “memória de trabalho” durante uma única inferência. Tecnicamente, esse limite é determinado pela arquitetura do modelo e pelas restrições de memória de vídeo (VRAM) do *hardware*, uma vez que a complexidade computacional do mecanismo de autoatenção padrão cresce de forma quadrática ($O(n^2)$) em relação ao comprimento da sequência (VASWANI et al., 2017). Em modelos modernos, essa capacidade evoluiu significativamente, variando de janelas de 8.000 *tokens* até extensões que superam os 128.000 *tokens*, permitindo o processamento de volumes massivos de dados de uma só vez (LIU et al., 2024).

Na Engenharia de Software, a dimensão da janela de contexto é o fator que dita a viabilidade de tarefas arquiteturais complexas. Enquanto janelas reduzidas limitam a atuação do modelo a funções isoladas ou pequenos trechos de código, janelas expandidas permitem que o LLM analise classes inteiras, arquivos de configuração e documentações técnicas simultaneamente (HOU et al., 2024). Essa percepção sistêmica da base de código constitui o alicerce para processos de reengenharia, uma vez que viabiliza o rastreamento de acoplamentos e dependências cruzadas entre componentes funcionalmente distantes (MACEDO et al., 2024).

Entretanto, a literatura técnica alerta para o fenômeno conhecido como *Lost in the Middle* (LIU et al., 2024). De acordo com o estudo experimental conduzido por Liu et al. (2024), a capacidade de recuperação de informações em LLMs não é uniforme ao longo da janela de contexto. Os autores demonstraram que o desempenho dos modelos é significativamente superior quando a informação relevante está localizada no início (*primacy effect*) ou no final (*recency effect*) do *prompt*, com degradação notável na precisão quando dados cruciais são posicionados no centro do contexto fornecido.

Para a Engenharia de Software, esse comportamento implica que, ao fornecer um repositório extenso para análise, o modelo pode ignorar regras de negócio ou definições de funções cruciais se estas estiverem “soterradas” no meio do arquivo, exigindo estratégias de priorização de dados na montagem do *prompt*.

Dessa forma, a escolha de um modelo para a documentação automatizada deve equilibrar a necessidade de um contexto amplo com a eficácia da atenção. Em cenários de sistemas legados de grande porte, nos quais o volume de código excede até

as maiores janelas disponíveis, tornam-se necessárias estratégias complementares, como o particionamento inteligente do código ou o uso de arquiteturas de Recuperação Aumentada por Geração (*Retrieval-Augmented Generation*, RAG), que filtram e entregam ao modelo apenas os fragmentos de contexto mais relevantes para a resolução de um problema específico (FAN et al., 2024).

2.4.3 Parâmetros de Configuração e Controle de Inferência

A geração de conteúdo por um LLM é, fundamentalmente, um processo estocástico no qual o modelo calcula a distribuição de probabilidade para o próximo *token* com base no contexto precedente (HOLTZMAN et al., 2020). O controle sobre como o sistema seleciona o próximo elemento dessa distribuição é exercido através de hiperparâmetros de inferência. Esses mecanismos permitem calibrar o equilíbrio entre o determinismo técnico necessário para a sintaxe e a variabilidade linguística exigida para a documentação (HOLTZMAN et al., 2020).

2.4.3.1 Temperatura e a Modulação da Confiança Estatística

O parâmetro de Temperatura (T) atua como um regulador da entropia e do nível de confiança estatística do modelo durante a fase de amostragem. Matematicamente, ele modifica a distribuição dos *logits* antes da normalização pela função *softmax*:

$$P_i = \frac{\exp(q_i/T)}{\sum_j \exp(q_j/T)} \quad (2.2)$$

Conceitualmente, a temperatura não altera o conhecimento contido nos pesos do modelo, mas sim sua estratégia de busca dentro do espaço latente. De acordo com a modulação aplicada, o comportamento do modelo transita entre três estados fundamentais:

- **Baixa Temperatura ($T \rightarrow 0$): Determinismo e Convergência.** A distribuição de probabilidade torna-se concentrada em torno dos *tokens* de maior probabilidade. O modelo seleciona predominantemente sequências de alta confiança estatística, reduzindo a variabilidade entre execuções em favor da precisão técnica (HOLTZMAN et al., 2020).
- **Temperatura Média ($0,5 \leq T \leq 0,7$): Equilíbrio e Fluidez.** O modelo preserva a coerência técnica, mas admite leve dispersão na curva de probabilidade. O texto resultante apresenta variação estilística sem comprometer o rigor conceitual.

- **Alta Temperatura ($T \rightarrow 1,0$): Exploração e Divergência.** A curva de probabilidade se achata, ampliando a chance de seleção de *tokens* de menor probabilidade. O modelo passa a explorar caminhos semânticos alternativos, favorecendo a diversidade lexical em detrimento do determinismo (HOLTZMAN et al., 2020).

Para tarefas de natureza técnica, em que a fidelidade ao objeto analisado é mais valiosa do que a variedade estilística, a literatura recomenda manter a temperatura em valores baixos, próximos da faixa determinística (MACEDO et al., 2024; HOLTZMAN et al., 2020). Esse regime conservador é particularmente importante em cenários de documentação derivada de código-fonte, nos quais temperaturas mais elevadas (próximas a 1,0) podem ampliar o risco de o modelo produzir descrições que extrapolam o que está efetivamente implementado, fenômeno relacionado à alucinação discutido em seções subsequentes. A literatura descreve ainda outros hiperparâmetros de amostragem, como *Top-K*, *Top-P* (*nucleus sampling*, proposto por Holtzman et al. (2020)) e penalidades de presença e frequência, que oferecem controle adicional sobre coerência e diversidade lexical da saída. Tais parâmetros não foram empregados neste trabalho, dado que a calibração da temperatura mostrou-se suficiente para assegurar a geração determinística e reproduzível requerida pelo *pipeline*.

2.4.4 Paradigmas de Execução: APIs de Nuvem vs. Modelos Locais

A viabilidade de integração de LLMs no ciclo de vida de desenvolvimento de software depende da escolha do ambiente de execução. Este paradigma define não apenas o desempenho da solução, mas também a segurança dos dados e a sustentabilidade financeira do projeto. Atualmente, o mercado se divide entre o consumo de modelos via API (*cloud*) e a implantação de modelos de pesos abertos em infraestrutura privada (HOU et al., 2024).

2.4.4.1 Segurança e Riscos das API Keys

O uso de APIs proprietárias implica em confiança no provedor externo. Ao utilizar uma *API Key* para enviar blocos de código para refatoração ou documentação, a organização expõe sua propriedade intelectual ao provedor de nuvem.

- **Vazamento de Informações:** A exposição acidental de chaves de API em repositórios públicos é uma falha recorrente em Engenharia de Software e pode resultar em prejuízos financeiros e acesso não autorizado a recursos.

Quadro 1 – Comparativo entre SaaS e Execução Local de LLMs.

Critério	APIs de Nuvem (SaaS)	Execução Local (Open-Weights)
Capacidade do Modelo	Acesso aos maiores modelos do mercado (1 trilhão ou mais de parâmetros)	Limitada pela memória de vídeo (VRAM) disponível
Privacidade	Código e dados trafegam para servidores externos	Controle total dos dados, que não saem do perímetro local
Custo Operacional	Variável (pagamento por milhão de <i>tokens</i>)	Fixo (investimento inicial em <i>hardware</i>)
Latência	Dependente da conexão de rede e carga do provedor	Baixa e estável (processamento na borda/ <i>edge</i>)
Customização	Limitada a parâmetros de inferência	Controle total sobre <i>fine-tuning</i> e quantização

Fonte: autoria própria.

- **Compliance e LGPD:** Para sistemas que lidam com dados sensíveis de usuários, o envio de informações para servidores estrangeiros pode violar regulamentações de proteção de dados e políticas de sigilo industrial. No Brasil, a Lei Geral de Proteção de Dados Pessoais (LGPD – Lei nº 13.709/2018) estabelece diretrizes rigorosas sobre a privacidade e o tratamento de informações. A legislação impõe restrições severas à transferência internacional de dados, exigindo que o país de destino ofereça um grau de proteção adequado ou que existam garantias contratuais e o consentimento explícito do titular. Dessa forma, a adoção de LLMs de terceiros (SaaS) exige uma avaliação criteriosa para evitar sanções legais e exposição de dados corporativos críticos.
- **Dependência de Terceiros:** Mudanças súbitas em termos de serviço, preços ou depreciação de modelos podem paralisar operações que dependem exclusivamente de uma API específica.

2.4.4.2 Execução Local com Ferramentas como Ollama

Para mitigar os riscos de segurança discutidos acima e garantir autonomia operacional, ferramentas como o *Ollama* vêm sendo adotadas para a execução local de LLMs (Ollama, 2025). Essas plataformas simplificam o gerenciamento de modelos de pesos abertos em máquinas ou servidores internos, oferecendo três benefícios principais: (i) *ambiente isolado*, no qual o código analisado nunca trafega pela internet; (ii) *independência de modelo*, com alternância entre diferentes tamanhos (7B, 14B ou 70B parâmetros) por meio de comando único; e (iii) *eficiência via quantização*, utilizando

formatos como GGUF para rodar modelos robustos em *hardware* de nível consumidor com técnicas de 4 ou 8 bits (DETTMERS et al., 2022).

A escolha entre o paradigma via API e a execução local é guiada por critérios como restrição orçamentária, sensibilidade dos dados manipulados, requisitos de latência e disponibilidade de *hardware* dedicado. A decisão concreta adotada nesta pesquisa, com sua respectiva justificativa, é apresentada no capítulo dedicado ao produto técnico.

2.4.5 Alucinações e Autocrítica Fundamentada

Apesar do ganho de produtividade que os LLMs oferecem em atividades de Engenharia de Software, sua aplicação exige postura crítica. Devido à natureza estocástica do processo de geração, discutida nas seções anteriores, esses modelos estão sujeitos ao fenômeno das alucinações (TONMOY et al., 2024). Conceitualmente, uma alucinação ocorre quando o modelo gera informações factualmente incorretas ou logicamente incoerentes, mas mantém uma estrutura linguística altamente convincente (TONMOY et al., 2024). Em código, isso pode se manifestar como sugestão de bibliotecas ou APIs inexistentes e criação de lógicas que parecem corretas sintaticamente, mas falham em casos de borda (LIU et al., 2024).

A mitigação convencional desse risco apoia-se no paradigma *Human-in-the-loop*, no qual o engenheiro de software atua como revisor e validador técnico das saídas geradas pelo modelo (MACEDO et al., 2024; TONMOY et al., 2024). Em complemento a esse paradigma, a literatura recente investiga estratégias em que o próprio LLM assume papéis distintos dentro de um mesmo *pipeline*, com *prompts* especializados, configurações de inferência diferenciadas e contextos isolados em cada chamada. Essa abordagem, conhecida como *LLM-as-judge*, utiliza o modelo de linguagem como avaliador crítico de saídas geradas por outro modelo (ou pelo próprio), produzindo notas, justificativas e recomendações de melhoria (TONMOY et al., 2024).

Quando combinada com um terceiro papel de *revisor*, que reescreve a saída original com base no *feedback* estruturado do juiz, configura-se um padrão denominado *self-refinement* ou *autocrítica iterativa*. Esse padrão é particularmente útil em cenários nos quais a saída do modelo deve aderir a um documento normativo: o juiz é instruído a fundamentar suas críticas em citações literais do documento, e o revisor é instruído a corrigir apenas as lacunas explicitamente apontadas, vedada a introdução de elementos sem fundamentação. A separação explícita dos três papéis (gerador, juiz e revisor) materializa um ciclo de validação auditável que mitiga o risco de alucinação documental discutido na subseção anterior (LIU et al., 2024; TONMOY et al., 2024).

A Tabela 2 sintetiza, no plano conceitual, as diferenças entre a geração *one-shot* convencional e o padrão de autocrítica iterativa descrito acima. A instanciação concreta desse padrão, com os respectivos *prompts* especializados e parâmetros de inferência, é apresentada no capítulo dedicado ao produto técnico.

Tabela 2 – Comparativo conceitual entre a geração *one-shot* e o ciclo de autocrítica iterativa.

Critério	Geração <i>one-shot</i>	Ciclo de Autocrítica Iterativa
Número de chamadas ao LLM	Uma chamada	Múltiplas chamadas (geração, validação e correção)
Avaliação da saída	Ausente; saída entregue diretamente	Avaliação estruturada do artefato gerado, segundo dimensões pré-definidas
Mitigação de alucinação	Limitada à instrução do <i>prompt</i> de geração	Reforçada pela exigência de fundamentação por citação literal do documento normativo
Auditabilidade	Resposta única, sem rastro de validação	Cada nota acompanhada de citação literal verificável
Estabilidade entre execuções	Sujeita à variância estocástica do modelo	Reduzida pela convergência do ciclo iterativo
Custo computacional (<i>tokens</i>)	Menor	Maior, proporcional ao número de chamadas

Fonte: autoria própria.

2.4.6 Estratégias de Injeção de Contexto: *Context Stuffing* e RAG

Quando o LLM precisa raciocinar sobre um documento normativo extenso (como o Guia do Scrum, normas ISO/IEC ou padrões corporativos), duas estratégias predominam na literatura para disponibilizar esse conteúdo ao modelo:

- **Retrieval-Augmented Generation (RAG):** arquitetura proposta para contornar os limites da janela de contexto. O documento normativo é fragmentado em *chunks*, indexado em um banco vetorial, e apenas os trechos mais semanticamente próximos à consulta são recuperados e injetados no *prompt* (FAN et al., 2024). RAG é a estratégia padrão quando o documento de referência excede a janela disponível ou quando há múltiplos documentos.
- **Context stuffing:** estratégia mais simples na qual o documento normativo é injetado *integralmente* no *system message* de cada chamada, sem fragmentação

ou recuperação seletiva. Funciona apenas quando a dimensão do documento cabe na janela de contexto do modelo, mas oferece a vantagem de garantir que o modelo opere sempre sobre o documento completo, sem omissões decorrentes de *retrieval* imperfeito (LIU et al., 2024).

A escolha entre as duas abordagens é determinada pelo binômio *tamanho do documento normativo* versus *janela de contexto do modelo*. Para documentos da ordem de dezenas de milhares de caracteres, contra modelos com janelas de 128 mil *tokens* ou mais, o *context stuffing* é frequentemente preferível por sua simplicidade arquitetural e por dispensar a complexidade adicional de infraestrutura de indexação semântica.

2.5 Java

Originalmente apresentada ao mercado em 1995 pela Sun Microsystems, Java emergiu como uma linguagem de programação concebida para responder a um problema então recorrente nas plataformas corporativas: a heterogeneidade de *hardware* e sistemas operacionais que dificultava a portabilidade do software (GOSLING et al., 2023).

A resposta arquitetural a esse desafio foi a introdução de uma camada intermediária de execução, a Máquina Virtual Java (JVM), responsável por interpretar um formato binário portátil denominado *bytecode*. Esse modelo dissocia a etapa de compilação da etapa de execução e fundamenta o princípio de portabilidade que se tornou marca registrada da linguagem. Em torno dessa base, Java incorpora um conjunto de decisões de projeto que reforçam sua confiabilidade: tipagem estática, orientação a objetos como paradigma central, gerenciamento automático de memória por meio de coletor de lixo (*garbage collector*) e um modelo rigoroso de tratamento de exceções (GOSLING et al., 2023).

Como consequência dessa combinação, Java consolidou-se ao longo de três décadas como uma das linguagens mais utilizadas mundialmente, particularmente em domínios que exigem escalabilidade horizontal, integração entre sistemas heterogêneos e longevidade operacional. O índice TIOBE (TIOBE Software, 2025) registra a presença sustentada da linguagem entre as cinco mais populares do mundo, fenômeno raramente observado em uma indústria marcada por ciclos rápidos de obsolescência tecnológica. Esse mesmo perfil de adoção explica, em larga medida, o volume expressivo de bases de código Java em uso ativo no setor produtivo, frequentemente acumuladas ao longo de décadas e mantidas sob constante demanda evolutiva.

2.6 NestJS

O NestJS é um *framework* de aplicações servidor para a plataforma Node.js, apresentado em 2017 com a proposta de elevar o nível de abstração arquitetural no desenvolvimento *back-end* sobre JavaScript (NestJS, 2025).

Diferentemente das abordagens minimalistas tradicionais do ecossistema Node.js (caracterizadas por bibliotecas leves e pouco opinativas), o NestJS apoia-se em decoradores, módulos e injeção de dependências nativa para impor uma organização de código próxima à praticada em *frameworks* de longa tradição corporativa, como Angular e Spring (NestJS, 2025). Sua adoção transparente do TypeScript reintroduz a disciplina da tipagem estática em um ambiente historicamente dominado por linguagens dinâmicas, oferecendo verificações em tempo de compilação e ferramental sofisticado de *refactoring*.

Esse desenho favorece a construção de APIs REST e arquiteturas baseadas em microsserviços com separação clara entre camadas de transporte, regra de negócio e infraestrutura. Estudos comparativos sobre o ecossistema Node.js apontam que *frameworks* estruturados nessa linha, ao contrário de soluções ad hoc, tendem a apresentar menor taxa de regressão e maior longevidade em projetos de média e grande escala (CHANLOTIS; KYRIAKOU; TSELIKAS, 2015).

2.7 React

Desenvolvido internamente pelo Facebook a partir de 2011 e aberto ao público em 2013, o React tornou-se uma das principais bibliotecas voltadas à construção de interfaces gráficas em ambiente *web* (React, 2025).

Sua principal contribuição reside na adoção de um paradigma declarativo: o estado da aplicação é declarado de forma explícita e a interface resultante é derivada como função desse estado, eliminando a manipulação imperativa do DOM tradicionalmente associada ao JavaScript do início dos anos 2000 (React, 2025). Para viabilizar essa abordagem com desempenho aceitável, a biblioteca introduz uma representação intermediária do DOM, conhecida como *Virtual DOM*, e um algoritmo de reconciliação capaz de calcular o conjunto mínimo de alterações necessárias para refletir uma mudança de estado.

A unidade fundamental de composição é o componente, entidade encapsulada que combina marcação, estilo e comportamento (AGGARWAL, 2018). Essa granularidade favorece a reutilização entre projetos, a testabilidade isolada de partes da interface e

a divisão de responsabilidades entre equipes que atuam em uma mesma aplicação. Ao longo de uma década, o ecossistema em torno do React amadureceu de forma significativa, com extensa oferta de bibliotecas para roteamento, gerenciamento de estado, internacionalização e renderização do lado do servidor.

2.8 Estudo de Caso na Reengenharia Assistida por Inteligência Artificial

A utilização de Inteligência Artificial na Engenharia de Software tem ganhado destaque nos últimos anos, principalmente em atividades relacionadas à manutenção, compreensão e modernização de sistemas legados. Com o avanço dos Modelos de Linguagem de Grande Escala (LLMs), tornou-se possível automatizar tarefas que tradicionalmente exigiam grande esforço manual, como geração de documentação técnica, análise de código-fonte e recuperação de regras de negócio (FAN et al., 2023; HOU et al., 2024).

Diferente das abordagens tradicionais de Engenharia Reversa, que dependem majoritariamente de análise manual e ferramentas baseadas em regras fixas, os LLMs possuem capacidade de interpretar o contexto semântico do código, identificando relações entre classes, métodos e componentes do sistema. Isso permite uma análise mais ampla da estrutura do software, contribuindo para a compreensão de sistemas desenvolvidos em Java e facilitando processos de manutenção e evolução arquitetural (CHAZETTE; SCHNEIDER, 2023).

Além disso, estudos recentes mostram que a integração entre Inteligência Artificial e Engenharia de Software vem sendo aplicada na geração automatizada de artefatos, transformação entre documentos técnicos e apoio à documentação de sistemas (RAHARJANA; SIAHAAN; FATICHAH, 2023). Esse cenário demonstra que a IA não atua apenas como suporte complementar, mas passa a ocupar um papel importante na automatização de processos relacionados à reengenharia de software.

No contexto das metodologias ágeis, essa aplicação torna-se ainda mais relevante devido às dificuldades encontradas pelas equipes para manter a documentação atualizada durante ciclos contínuos de entrega. Em muitos projetos, a prioridade dada ao desenvolvimento de funcionalidades acaba reduzindo o tempo destinado à produção documental, resultando em registros incompletos ou desatualizados. Dessa maneira, o uso de IA surge como alternativa para reduzir a dívida técnica documental sem comprometer a agilidade do desenvolvimento.

Com base nesse cenário, este trabalho propõe a ferramenta *docScrum*, desenvol-

vida com o objetivo de automatizar a geração de documentação ágil a partir da análise de código-fonte Java. A proposta busca utilizar a Inteligência Artificial como elemento central do processo de reengenharia, permitindo a extração automática de requisitos, documentação técnica e organização inicial de artefatos do *framework* Scrum, como *Product Backlogs* e *Sprints*. Adicionalmente, sobre os artefatos diretamente alinhados ao escopo do Guia do Scrum 2020, o sistema implementa um ciclo de *validação e correção fundamentada* no qual o próprio LLM atua sucessivamente como gerador, juiz crítico e revisor, mecanismo que constitui a contribuição metodológica central deste trabalho e cujo detalhamento é apresentado no capítulo de estudo de caso experimental.

2.9 Considerações sobre o Capítulo

Neste capítulo foram apresentados os conceitos norteadores para o desenvolvimento deste trabalho. A Seção 2.1 discutiu a reengenharia de software como estratégia para recuperar o valor de negócio de sistemas legados sem documentação atualizada, evidenciando o elevado custo do processo clássico conduzido manualmente. A Seção 2.2 apresentou o panorama das metodologias ágeis e o desafio recorrente da *documentation technical debt*, situando a tensão entre velocidade de entrega e rigor técnico. A Seção 2.3 detalhou os pilares, artefatos e compromissos do Scrum, base normativa do *framework* ágil mais adotado em projetos de software. A Seção 2.4 discutiu os fundamentos técnicos dos Modelos de Linguagem de Grande Escala (arquitetura *Transformer*, janela de contexto e calibração da temperatura), os paradigmas de execução (APIs de nuvem versus modelos locais) e as estratégias de *LLM-as-judge*, *self-refinement* e *context stuffing* relevantes para abordagens de autocritica fundamentada. Por fim, as Seções 2.5, 2.6 e 2.7 apresentaram, em caráter conceitual, as tecnologias que compõem o sistema desenvolvido: Java como linguagem-alvo da análise, NestJS como *back-end* e React como *front-end*.

A integração desses corpos teóricos compõe o aparato conceitual sobre o qual o presente trabalho se assenta. Do Scrum, são herdados a estrutura normativa de artefatos e compromissos e os pilares empíricos de transparência, inspeção e adaptação. Da literatura sobre LLMs, são herdadas as noções de calibração de hiperparâmetros para geração técnica determinística, a estratégia de *context stuffing* para injeção integral de documentos normativos no contexto do modelo, e o padrão *self-refinement*, no qual o mesmo modelo desempenha papéis distintos em chamadas separadas.

A separação explícita dos papéis de *gerador*, *juiz* e *revisor*, fundamentada na Seção 2.4.5, constitui o pano de fundo conceitual para a contribuição metodológica deste

trabalho, sintetizada no comparativo da Tabela 2. A instanciação concreta dessa abordagem, com as decisões específicas de provedor, modelo, hiperparâmetros, documento normativo e *prompts* especializados, é apresentada no capítulo dedicado ao produto técnico.

O próximo capítulo descreve a metodologia experimental adotada para a seleção do modelo de LLM a ser integrado ao sistema, detalhando os critérios de *benchmarking* aplicados e os ensaios práticos conduzidos sobre cenários representativos de refatoração estrutural em código Java.

3 Trabalhos Relacionados

Neste capítulo, apresenta-se uma revisão bibliográfica acerca do estado da arte na utilização de Modelos de Linguagem de Grande Escala (*Large Language Models*, doravante designados por LLMs) aplicados à Engenharia de Software. A finalidade deste capítulo consiste em mapear o estado da arte e confrontar a proposta desta monografia com as soluções vigentes, evidenciando as lacunas acadêmicas na integração de LLMs para a automação de reengenharia e governança de software. Busca-se, assim, consolidar o embasamento teórico necessário para a construção de um metaprocesso que harmonize a documentação técnica e a modelagem arquitetural dentro dos ritos da agilidade.

A integração de Inteligência Artificial generativa no ciclo de vida do desenvolvimento de software representa uma mudança de paradigma estrutural. A engenharia de software transita de ferramentas puramente sintáticas para assistentes semânticos, os quais são capazes de compreender a intenção intrínseca do código. Esta revisão concentra a sua análise na forma como essa capacidade cognitiva artificial é explorada para mitigar a dívida técnica e a obsolescência de sistemas clássicos.

3.1 Procedimento de Pesquisa Bibliográfica

Para mapear o cenário tecnológico na direção da integração de Inteligência Artificial Generativa em Métodos Ágeis, foi feita uma pesquisa bibliográfica baseada em técnicas sistemáticas propostas por Kitchenham (2004). A adoção deste rigor metodológico, que é amplamente aceito na comunidade científica de Engenharia de Software, visa garantir a total replicabilidade do estudo e a mitigação de vieses durante a seleção dos trabalhos primários.

O processo de busca não se resumiu a uma coleta aleatória de títulos, constituindo antes uma investigação qualitativa estruturada em três fases críticas:

1. **Planejamento e Definição de Escopo:** Nesta etapa inicial, elaborou-se a Questão de Pesquisa (QP) e estruturou-se o protocolo formal de busca. Definiu-se que o foco estaria na intersecção exata entre Agilidade, Engenharia Reversa e Inteligência Artificial. Selecionaram-se termos técnicos específicos para garantir que a pesquisa abrangesse a teoria de agentes virtuais e a prática de infraestruturas seguras.

2. **Condução e Filtragem Técnica:** Este foi o momento da execução técnica da busca nas bibliotecas digitais conceituadas, a saber, IEEE Xplore, ACM Digital Library e repositórios institucionais de excelência. Após a coleta bruta de centenas de títulos, aplicou-se um filtro rigoroso. Os critérios de exclusão removeram trabalhos que tratavam a Inteligência Artificial apenas para a geração de trechos de código simples, priorizando as obras que discutiam a arquitetura de sistemas complexos.
3. **Sumarização e Análise Crítica:** A etapa final consistiu na extração de dados e síntese qualitativa. Os trabalhos selecionados foram confrontados em uma matriz de características técnicas (detalhada na Tabela 4), evidenciando as falhas literais que justificam o desenvolvimento prático desta dissertação.

A questão de pesquisa (QP) que direcionou este levantamento sistemático foi formulada nos seguintes termos: **“QP: Como se caracteriza o cenário atual sobre a hibridização e a integração entre Modelos de Linguagem de Grande Escala e os métodos ágeis de desenvolvimento de software?”**. O protocolo detalhado encontra-se apresentado na Tabela 3.

Tabela 3 – Protocolo de pesquisa baseado em Kitchenham (2004).

Componente	Descrição Detalhada do Protocolo de Pesquisa
Questão de Pesquisa	Investigar métodos de integração de LLMs em fluxos de Reengenharia Ágil e a sua eficácia na manutenção de software e governança corporativa.
String de Busca	("LLM"OR "GPT"OR "RAG") AND ("Reverse Engineering"OR "Documentation"OR "MDE") AND ("Agile Methods"OR "Scrum")
Bases de Dados	IEEE Xplore, ACM Digital Library, Google Scholar, Repositórios institucionais UFRN, UESPI e URI.
Critérios de Inclusão	Artigos com evidência experimental, estudos de caso reais, foco em automação de artefatos complexos e publicações recentes entre 2024 e 2026.
Critérios de Exclusão	Trabalhos conceituais sem protótipo, artigos focados apenas no ensino de programação e publicações sem revisão por pares.

Fonte: elaborado pelos autores com base em Kitchenham (2004).

3.2 Trabalhos Seleccionados

Nesta seção, os seis trabalhos fundamentais aprovados pelo protocolo são dissecados ao detalhe. Para cada autor, apresenta-se o problema corporativo abordado, a solução técnica proposta por meio de experimentação e a forma como esse conhecimento se conecta diretamente com o metaprocessos de Reengenharia Ágil proposto neste trabalho.

3.2.1 Assistência na Geração de Documentação (Madruga, 2025)

O trabalho de Madruga (2025) ataca um problema endêmico da cultura de desenvolvimento iterativo: a resistência comportamental dos programadores em manter a documentação técnica atualizada. Madruga argumenta que, em ambientes de alta velocidade de entrega, a documentação é frequentemente sacrificada em favor do código funcional, gerando uma dívida técnica invisível que compromete a equipe na fase de manutenção futura.

Metodologia e Resultados: O autor implementou um rigoroso experimento controlado com doze engenheiros de software seniores. Estes profissionais foram divididos em grupos para documentar módulos de microsserviços desatualizados. O grupo assistido por ferramentas inteligentes utilizou modelos treinados especificamente para compreender a semântica de linguagens de programação. Os resultados práticos foram impressionantes, registrando uma redução de cinquenta e oito por cento no tempo total de escrita. Adicionalmente, Madruga utilizou métricas de legibilidade reconhecidas internacionalmente para provar que os textos gerados de forma artificial eram mais claros e padronizados do que os manuais escritos pelos próprios criadores do sistema.

Relacionamento com esta Pesquisa: Madruga (2025) fornece a validação estatística necessária para provar que a nossa etapa de geração de artefatos é exequível na prática corporativa. Ao demonstrar que é possível reduzir o esforço documental em mais de metade, remove-se a principal barreira para que equipes ágeis adotem a documentação completa como um requisito para a conclusão de uma tarefa.

3.2.2 Inteligência Artificial em Sistemas Legados (Araújo, 2025)

Araujo (2025) debruça a sua investigação sobre a técnica de arqueologia de código. O grande desafio delineado pela autora reside na manutenção de sistemas legados massivos, construídos em linguagens antigas como COBOL ou Java clássico, nos quais os programadores originais já não integram a organização e o código opera como um bloco completamente opaco.

Metodologia e Resultados: Araújo testou a capacidade de raciocínio de contexto longo dos modelos modernos, especificamente o GPT-4o e o Claude 3.5. O estudo empírico submeteu dezenas de milhares de linhas de código desestruturado à Inteligência Artificial, solicitando que a máquina extraísse e explicasse as regras de negócio implícitas. O modelo Claude 3.5 atingiu noventa e dois por cento de acurácia, superando largamente o trabalho de analistas humanos. A pesquisa destaca que os algoritmos conseguem identificar dependências lógicas dispersas por múltiplos arquivos, correlações estas que são quase invisíveis à análise visual humana.

Relacionamento: O estudo de Araújo constitui a fundação científica para o nosso módulo de extração e engenharia reversa. A prova documental de que uma inteligência artificial consegue interpretar o histórico do software com altíssima precisão permite que o nosso metaprocessamento transforme sistemas antigos em novos requisitos ágeis de forma automatizada, assegurando que nenhuma regra de negócio crítica se perca durante a transição tecnológica.

3.2.3 Automação Segura e Arquitetura RAG (Rotilli e Silva, 2025)

O trabalho de Rotilli e Silva (2026) traz uma contribuição de importância vital sobre a segurança e a infraestrutura dos dados. Os autores abordam o justificado receio corporativo de compartilhar código-fonte proprietário com plataformas de inteligência artificial alocadas em nuvens públicas e servidores externos.

Metodologia e Resultados: Os pesquisadores desenvolveram um protótipo focado em equipes de software que utiliza a arquitetura de Geração Aumentada por Recuperação, vulgarmente conhecida pela sigla RAG. O sistema foi construído sobre uma infraestrutura local, processando dados internamente sem necessidade de conexão à internet para o motor de inferência. A integração ocorreu de forma cifrada e segura, permitindo que a inteligência artificial consultasse documentos da empresa de forma estritamente privada. O fluxo de processamento dividiu a atuação da máquina em duas camadas distintas, obtendo total precisão na classificação da intenção do usuário e eliminando o fenômeno das alucinações algorítmicas.

Relacionamento com esta Dissertação: Rotilli e Silva resolvem de forma brilhante o obstáculo da confiança corporativa. O uso de modelos operados localmente e com arquiteturas controladas, tal como eles propõem, será formalmente incorporado na arquitetura da nossa solução para garantir que a Reengenharia Ágil cumpra todas as diretrizes da Lei Geral de Proteção de Dados.

3.2.4 Refatoração Assistida por LLM (Freitas, 2025)

Freitas (2025) investiga detalhadamente como os algoritmos generativos podem atuar na otimização profunda da estrutura do software. O foco central da obra é a identificação de anomalias no código e a aplicação automática de padrões de projeto internacionalmente validados.

Metodologia e Resultados: Por meio de um estudo efetuado em larga escala em dezenas de repositórios de código aberto, Freitas demonstrou que as sugestões da inteligência artificial conseguem reduzir a complexidade ciclomática do sistema em trinta por cento. O diferencial metodológico desta investigação assentou no uso contínuo de testes automatizados para validar cada sugestão fornecida pela máquina. Em noventa e cinco por cento dos cenários testados, as alterações estruturais sugeridas foram totalmente seguras e não comprometeram a funcionalidade original da aplicação.

Relacionamento: Esse trabalho justifica rigorosamente o passo inaugural da nossa proposta prática. Antes de efetuar qualquer modelagem documental, é absolutamente imperativo garantir que o código base possui qualidade estrutural. Freitas prova categoricamente que os modelos atuais possuem capacidade para realizar essa limpeza técnica, permitindo que o nosso processo de reengenharia evolua a partir de alicerces sólidos e modernos.

3.2.5 Modelagem Visual e Diagramas de Arquitetura (Hatahet et al., 2025)

O trabalho de Hatahet, Knieke e Rausch (2025) aborda a necessidade premente da representação visual na arquitetura de software. O grande problema crônico exposto pelos autores é que os diagramas sistêmicos tornam-se obsoletos exatamente no momento em que a primeira linha de código é modificada pelo programador.

Metodologia e Resultados: Os autores desenvolveram uma ponte tecnológica entre o código fonte e as ferramentas avançadas de diagramação estrutural. O modelo analisa autonomamente as chamadas de interface e as dependências de bases de dados, gerando o código vetorial para os diagramas arquiteturais sem intervenção humana. A estrutura proposta atingiu noventa por cento de fidelidade visual, possibilitando que os arquitetos de software dediquem o seu tempo apenas à revisão analítica dos modelos, abandonando o desenho manual moroso.

Relacionamento: Esse estudo fundamenta o núcleo do nosso módulo de modelagem visual. A nossa proposta de Reengenharia Ágil incorpora esta técnica de mapeamento para assegurar que o desenvolvedor, ao encerrar o seu ciclo de trabalho, tenha

todos os diagramas de componentes atualizados de forma instantânea e autônoma pela Inteligência Artificial.

3.2.6 Engenharia de Modelos e Orquestração Agêntica (Di Rocco et al., 2025)

Rocco et al. (2025) discute a evolução teórica e prática da Engenharia Dirigida por Modelos no contexto contemporâneo. Os pesquisadores propõem de forma assertiva que a inteligência artificial não deve ser encarada meramente como um assistente de texto, mas sim elevada à condição de um agente orquestrador de processos.

Metodologia e Resultados: Rocco et al. (2025) demonstra conceitualmente como múltiplos agentes artificiais podem colaborar em um ecossistema fechado. Nesse cenário, um agente dedica-se exclusivamente à leitura do código, outro realiza a validação das regras sintáticas e um terceiro é responsável pela geração da documentação final. Essa orquestração distribuída garante um nível de consistência e governança que uma única ferramenta raramente consegue manter em sistemas de larga escala.

Relacionamento com este Trabalho: Esse trabalho fornece a inteligência processual indispensável para o metaprocesso proposto neste trabalho. O nosso metaprocesso não se resume a uma ferramenta singular, mas consiste em uma orquestração complexa inspirada diretamente na lógica de colaboração de Rocco et al. (2025), assegurando que todas as etapas de evolução do software sejam rigorosamente integradas e validadas.

3.3 Quadro Comparativo e Justificação do Diferencial Acadêmico

A análise aprofundada dos seis trabalhos revela um panorama claro. Embora a tecnologia generativa apresente um grau de maturidade excepcional, as soluções disponíveis no mercado e na academia encontram-se altamente fragmentadas. A Tabela 4 cruza as matrizes de funcionalidades para destacar com precisão o espaço científico que esta dissertação preenche.

Interpretação Técnica da Matriz Comparativa: O mapeamento efetuado foi estruturado segundo cinco competências técnicas cruciais. A coluna **C1** refere a capacidade de Engenharia Reversa. A coluna **C2** indica a Geração de Documentação Técnica. A coluna **C3** representa a Modelagem Visual. A coluna **C4** foca a Refatoração e a Dívida Técnica. Finalmente, a coluna **C5** aborda a Integração Ágil, as estruturas de

Tabela 4 – Mapeamento de Lacunas e Diferencial Estratégico.

Trabalho Analisado	C1	C2	C3	C4	C5
Madruga (2025)	Não	Sim	Não	Não	Parcial
Araújo (2025)	Sim	Sim	Não	Não	Não
Rotilli e Silva (2025)	Parcial	Sim	Não	Não	Sim
Freitas (2025)	Não	Não	Não	Sim	Parcial
Hatahet (2025)	Sim	Sim	Sim	Não	Não
Di Rocco (2024)	Não	Não	Parcial	Não	Sim
Esta Pesquisa (Proposta)	Sim	Sim	Sim	Sim	Sim

Fonte: autoria própria.

conhecimento fechado e a Governança.

A observação detalhada da matriz evidencia que nenhum autor consegue cobrir de forma holística todas as frentes necessárias para uma reengenharia completa e segura. Os autores Rotilli e Silva trazem a segurança arquitetural absolutamente necessária, mas não dedicam esforços práticos à refatoração matemática do código. Por sua vez, Hatahet entrega os diagramas visuais perfeitos, mas não consegue integrá-los no ritmo acelerado e diário exigido pelo método Scrum. A nossa proposta científica é o amálgama orquestrado de todas estas capacidades dispersas.

3.4 Considerações sobre o Capítulo

A revisão bibliográfica minuciosa realizada neste capítulo demonstra sem margem para dúvidas que o campo da Inteligência Artificial aplicada à Engenharia de Software se encontra em plena efervescência produtiva. As evidências de Madruga sobre a economia drástica de tempo produtivo, somadas à elevada acurácia nos sistemas clássicos atestada por Araújo, fornecem a segurança absoluta de que os modelos de linguagem não representam apenas uma tendência acadêmica passageira. Trata-se, de fato, de ferramentas basilares de produtividade industrial.

Contudo, a principal conclusão retirada desta extensiva revisão reside no fato de que o maior desafio atual não recai sobre a capacidade analítica da máquina, mas sim sobre a engenharia de processos que a circunda. O programador contemporâneo possui acesso a plataformas incrivelmente potentes, porém, estas operam em silos isolados de conhecimento. Sem a existência de um metaprocessos centralizador que consiga ditar o momento exato em que a máquina deve refatorar, desenhar diagramas ou gerar manuais técnicos, o resultado será uma automação completamente desor-

ganizada, suscetível de gerar ainda mais confusão técnica do que o processo manual tradicional.

Conclui-se de forma inequívoca que o diferencial estratégico desta dissertação é a orquestração sistêmica unificada. Os grandes pilares teóricos aqui exaustivamente levantados perdem a sua condição de ilhas tecnológicas isoladas para se transformarem em etapas colaborativas e fluidas no interior do nosso metaprocesso de Reengenharia Ágil. No próximo capítulo, iremos detalhar com precisão esquemática o modo como esta fusão teórica é formalizada e transposta para o desenvolvimento prático do produto de software.

4 Metodologia de Seleção de LLM

Este capítulo apresenta a metodologia experimental adotada para a seleção do modelo de Inteligência Artificial integrado ao sistema. Esse modelo ocupa posição central no trabalho, sendo responsável pela compreensão do código-fonte Java e pela geração dos artefatos ágeis. Em razão dessa centralidade, a escolha recebeu tratamento metodológico rigoroso, organizado em três etapas: o levantamento inicial dos candidatos sob restrição orçamentária, a análise comparativa em portais de *benchmarking* e os ensaios práticos de refatoração estrutural sobre cinco cenários representativos em Java. As próximas seções detalham cada uma dessas etapas, culminando na justificativa formal da escolha do DeepSeek R1 como modelo definitivo.

4.1 Levantamento Inicial dos Modelos

Para viabilizar o processo de reengenharia orientada por IA para geração de documentação ágil a partir de código Java proposto neste trabalho, fez-se necessária a seleção de um modelo de Inteligência Artificial (IA) generativa capaz de analisar, compreender e refatorar códigos-fonte com alta precisão. O processo de seleção foi pautado por um limite orçamentário e por métricas de desempenho técnico.

Cabe esclarecer a estratégia metodológica adotada para a seleção do modelo. Embora o objetivo final do sistema seja a geração de artefatos ágeis (*Backlog* e *Sprint Planning*), a etapa de avaliação dos LLMs candidatos foi conduzida por meio de ensaios de reengenharia estrutural sobre código Java, e não diretamente pela produção dos artefatos finais. Essa decisão fundamenta-se em três razões metodológicas:

1. A geração de artefatos Scrum a partir de código-fonte é, antes de tudo, um problema de *compreensão semântica do código*: o modelo precisa interpretar regras de negócio, dependências e responsabilidades implícitas nas classes para então traduzi-las em *user stories* consistentes. Essa mesma capacidade de compreensão é o pré-requisito central das tarefas de reengenharia.
2. Ensaios de refatoração oferecem critérios de avaliação objetivos e verificáveis (compilação, ausência de regressões, aplicação correta de padrões de projeto, identificação de antipadrões). Em contraste, a avaliação direta da qualidade de documentação tende a esbarrar em julgamentos subjetivos, uma vez que diferentes saídas podem ser igualmente plausíveis sob um *prompt* bem estruturado.

3. A reengenharia, por exigir alterações estruturais sem comprometer o comportamento do sistema, funciona como um teste de estresse mais severo da capacidade analítica do modelo. Um LLM que falha em reconhecer violações arquiteturais (como a quebra do Princípio da Substituição de Liskov) dificilmente será capaz de produzir um *Product Backlog* fiel às responsabilidades reais do sistema.

Em síntese, esta seção busca identificar o modelo com maior potência de compreensão estrutural de código Java, requisito que sustenta o ciclo de geração e validação de artefatos descrito nos capítulos subsequentes.

A pesquisa inicial estabeleceu um teto de investimento de US\$ 3,00 a cada 1 milhão de tokens processados. Sob esta restrição, realizou-se um levantamento dos modelos mais recentes e atualizados do mercado, utilizando como base os dados fornecidos pela plataforma de benchmarking *Benched.ai* (Benched.ai, 2025). Os modelos pré-selecionados e seus respectivos dados de desempenho estão detalhados na Tabela 5.

Tabela 5 – Levantamento inicial de Modelos de Inteligência Artificial sob a restrição orçamentária.

Modelo	Inteligência	Contexto	Custo Input (1M)	Custo Output (1M)	MMLU-Pro	Tokens/s
Gemini 2.5 Flash	53.43	1.000.000	\$0.15	\$0.60	80.9%	279.32
DeepSeek V3	53.24	128.000	\$0.27	\$1.10	81.9%	28.57
GPT-4.1 mini	52.86	1.000.000	\$0.40	\$1.60	78.1%	70.92
DeepSeek R1	68.29	128.000	\$0.55	\$2.19	85.0%	28.57
Llama 3.3 (70B)	41.11	128.000	\$0.58	\$0.70	71.3%	60.03
Grok 3 mini Reason	66.67	1.000.000	\$0.30	\$0.50	82.8%	209.18
GPT-5 mini	67.00	400.000	\$0.60	\$2.40	87.0%	136.90
MiniMax M1 40k	61.35	1.000.000	\$0.40	\$2.10	80.8%	41.13

Fonte: elaborado pelos autores com base em dados de *Benched.ai*.

Para a correta interpretação dos dados coletados, definiram-se as seguintes métricas de avaliação técnica e financeira:

- **Contexto (Janela de Contexto):** Refere-se à quantidade máxima de tokens (fragmentos de palavras ou código) que a IA consegue processar e “lembrar” em uma única interação. É um fator crucial para a análise de arquivos de código extensos ou múltiplos arquivos simultâneos.
- **Custo Input (1M) e Custo Output (1M):** Representam, respectivamente, o valor financeiro cobrado pela provedora da API para ler 1 milhão de tokens enviados pelo usuário (Input) e para gerar 1 milhão de tokens de resposta (Output).

- **MMLU-Pro (Massive Multitask Language Understanding – Professional):** É um benchmark rigoroso utilizado para medir a capacidade de raciocínio, lógica e conhecimento do modelo em múltiplas disciplinas. Taxas elevadas de MMLU geralmente correlacionam-se com uma maior capacidade de resolver problemas complexos de programação.
- **Tokens/s (Tokens por segundo):** Indica a velocidade de inferência do modelo, ou seja, a rapidez com que a IA consegue gerar a resposta. É um fator determinante para a fluidez e usabilidade da ferramenta de reengenharia.

4.1.1 Filtragem e Seleção dos Modelos Candidatos

O critério de maior peso estabelecido para a filtragem foi o nível de inteligência e precisão, uma vez que o processo de reengenharia exige alterações estruturais seguras sem comprometer o comportamento do sistema. Por esta razão, modelos como o *Gemini 2.5 Flash*, embora apresentassem as melhores taxas de velocidade e o menor custo financeiro, foram descartados por não acompanharem o parâmetro de inteligência das propostas concorrentes.

Como não houve um modelo que apresentasse superioridade unânime em todos os quesitos, selecionaram-se os três que mais se destacaram em aspectos específicos:

1. **DeepSeek R1:** Apresentou o maior índice de inteligência geral e a segunda melhor taxa de MMLU-Pro. Contudo, possui como desvantagens uma velocidade de inferência reduzida e uma janela de contexto menor em relação aos demais.
2. **Grok 3 mini Reason:** Destacou-se por oferecer a maior janela de contexto (1 milhão de tokens), o custo de *output* mais baixo e a segunda maior velocidade, mantendo, simultaneamente, altas taxas de inteligência e MMLU-Pro.
3. **GPT-5 mini:** Apresentou a melhor taxa de MMLU-Pro entre os avaliados, com uma janela de contexto intermediária e excelente inteligência. Configura-se como a opção mais equilibrada, unindo bons resultados em quase todos os parâmetros avaliados.

4.1.2 Estratégia de Avaliação Prática

A estratégia de avaliação prática foi planejada considerando que a refatoração de uma classe isolada exige menos da janela de contexto do modelo do que a análise de um projeto inteiro. Inicialmente, cogitou-se conduzir avaliações distintas para os

dois cenários (denominados IA de Classe e IA de Projeto). Como será discutido posteriormente, os resultados obtidos no nível de classe foram suficientes para unificar a escolha do modelo, dispensando a segunda frente.

4.1.2.1 Testes e Avaliação: IA de Classe

A avaliação para a IA de Classe busca responder à seguinte questão norteadora: *Apesar do modelo DeepSeek R1 apresentar a maior potência intelectual, ele é, de fato, o mais capacitado para a geração e refatoração de códigos? Em caso afirmativo, a sua limitação de contexto e sua baixa velocidade comprometem a eficiência da reengenharia em nível de classe?*

Para responder a esta questão, estruturou-se uma rotina de testes baseada em benchmarking de terceiros e ensaios práticos na própria ferramenta. Os passos definidos foram:

1. **Validação em Referência de Programação:** Verificação do modelo com maior capacidade específica para geração de código (*coding*) segundo os portais *Artificial Analysis* (Artificial Analysis, 2025) e *Benched.ai* (Benched.ai, 2025).
2. **Ensaio Prático:** Os modelos que apresentarem os melhores desempenhos na etapa anterior serão submetidos a testes práticos dentro do ambiente de desenvolvimento do presente projeto.

Os testes práticos foram projetados para avaliar três pilares fundamentais, obedecendo aos seguintes critérios de aceitação:

- **Teste de Velocidade:** Medição do tempo de resposta do modelo. Este critério foi tratado como métrica comparativa entre os candidatos e não como restrição eliminatória, uma vez que modelos com maior capacidade de raciocínio tendem a apresentar latência superior em função de processos internos de *chain-of-thought*.
- **Teste de Eficiência:** Avaliação qualitativa do código retornado. A IA deve efetivamente aumentar a qualidade estrutural da classe, aplicando boas práticas de programação, padrões de projeto adequados e código limpo (*Clean Code*).
- **Teste de Confiabilidade:** Monitoramento da taxa de alucinação e precisão. A IA não deve cometer erros de sintaxe, apagar lógicas de negócio existentes ou gerar código incompilável.

Critério de Desempate: Estabeleceu-se como parâmetro que, caso o portal *Artificial Analysis* aponte uma IA vencedora no quesito codificação e o *Benchded.ai* aponte uma IA distinta, e ambas sejam aprovadas nos testes práticos com níveis de eficiência e confiabilidade semelhantes, a escolha final será definida pelo modelo que apresentar o menor custo de operação.

4.1.2.2 Definição das Classes de Teste

Para a realização dos ensaios práticos, o escopo original de testes foi refinado para cinco cenários distintos. Em vez de testes genéricos, optou-se por desenvolver cinco classes na linguagem Java, cada uma contendo antipadrões, falhas estruturais ou bugs lógicos específicos. O objetivo é avaliar a capacidade de cada modelo em identificar e corrigir problemas complexos durante a reengenharia. As classes foram definidas da seguinte forma:

1. **Classe 1:** Contém problemas de encapsulamento e inconsistência de estado, violando os princípios de ocultação de informação ao utilizar atributos públicos.
2. **Classe 2:** Apresenta um vazamento de estado mutável (*Mutable State Leak*), um erro sutil em que o retorno de referências diretas burla as regras de negócio da classe.
3. **Classe 3:** Inclui um bug sutil de concorrência atrelado a uma lógica de cache, exigindo da IA uma compreensão profunda sobre *threads* e condições de corrida (*race conditions*).
4. **Classe 4:** Demonstra um problema de herança mal estruturada que resulta na violação do Princípio da Substituição de Liskov (LSP – *Liskov Substitution Principle*).
5. **Classe 5:** Apresenta um alto nível de complexidade ciclomática e lógicas de negócio escondidas ou acopladas, demandando uma refatoração severa para atingir os padrões de *Clean Code* (violação de SRP e OCP).

4.1.2.3 Análise de Desempenho em Codificação: Artificial Analysis

Antes da submissão aos ensaios práticos, realizou-se um estudo aprofundado dos modelos pré-selecionados utilizando a plataforma *Artificial Analysis* (Artificial Analysis, 2025). O foco desta etapa foi avaliar os parâmetros de programação de cada IA utilizando a métrica *Coding Index* e seus respectivos custos operacionais e de raciocínio. Os dados compilados nesta análise estão apresentados na Tabela 6.

Tabela 6 – Resultados do Benchmark Artificial Analysis para os Modelos Finalistas.

Modelo	Coding Index	Answer Tokens	Reasoning Tokens	Custo Input	Custo Raciocínio	Custo Resposta
GPT-5 Mini	35	830k	4.1M	3	8	2
Grok 3 mini Reason	25	830k	4.4M	7	9	0
DeepSeek R1	24	460k	2.6M	7	14	3

Fonte: elaborado pelos autores com base em dados de Artificial Analysis.

Para a compreensão adequada dos resultados apresentados na Tabela 6, os indicadores avaliados dividem-se nas seguintes categorias:

- **Coding Index:** Métrica agregada que quantifica a proficiência do modelo especificamente em engenharia de software (geração, refatoração e correção de código). O GPT-5 Mini demonstrou ampla vantagem competitiva ao atingir 35 pontos, descolando-se significativamente de seus concorrentes diretos.
- **Composição de Tokens de Saída (Answer vs. Reasoning):** Ilustra a proporção do esforço computacional. Os *Reasoning Tokens* representam o planejamento lógico (o “pensamento” da IA para encontrar a falha arquitetural), enquanto os *Answer Tokens* representam o código entregue. Observa-se que todos os modelos utilizam a técnica de *Chain of Thought*, dedicando um volume massivo de tokens ao raciocínio interno antes da resposta final.
- **Cost Breakdown (Detalhamento de Custos):** Destrincha o peso financeiro da operação nas fases de Leitura (*Input*), Processamento Lógico (*Reasoning*) e Geração (*Resposta*).

Complementarmente, dois pontos merecem destaque na consolidação desta análise:

- **Fator de Atualização Tecnológica:** Além das métricas quantitativas, é imperativo considerar que o GPT-5 Mini é a arquitetura mais recente entre os modelos avaliados nesta fase. No contexto da reengenharia de software, modelos mais recentes são treinados com documentações e discussões arquiteturais mais modernas, reduzindo o risco de a IA sugerir bibliotecas depreciadas ou antipadrões já superados pela comunidade.
- **Conclusão da Análise de Benchmarking:** Avaliando os dados consolidados, o GPT-5 Mini consolida-se como a melhor escolha técnica nesta etapa preliminar. Ele não apenas apresenta uma superioridade expressiva no *Coding Index*, como também mantém os custos de *Input* e de *Raciocínio* consideravelmente inferiores

aos do DeepSeek R1 e do Grok 3 Mini Reason. A combinação de alta capacidade de codificação, eficiência de custo e o fator de ser a arquitetura mais recente justificam sua liderança estatística para as tarefas de refatoração das classes estabelecidas.

4.1.2.4 Análise de Desempenho e Viabilidade: Benched.ai

Para contrastar e validar os dados obtidos anteriormente, os modelos também foram submetidos a uma avaliação sob a ótica da plataforma *Benched.ai* (Benched.ai, 2025). Esta plataforma fornece um detalhamento granular sobre a latência computacional, custos combinados e o desempenho em testes rigorosos de programação e matemática. Os dados extraídos estão consolidados na Tabela 7.

Tabela 7 – Métricas de Desempenho, Custo e Latência.

Modelo	Contexto	LiveCodeBench	HumanEval	Velocidade (t/s)	Tempo (500 tokens)	Custo Blended 3:1
DeepSeek R1	128k	77.0%	97.0%	29 t/s	89.94 s	\$0.96
Grok 3 mini Reason	1.000k	69.6%	97.8%	209 t/s	12.27 s	\$0.35
GPT-5 mini	400k	60.0%	98.0%	80 t/s	6.50 s	\$1.05

Fonte: elaborado pelos autores com base em dados de *Benched.ai*.

A partir dos dados dispostos, estabeleceram-se quatro tópicos de pesquisa cruciais para fundamentar a viabilidade das IAs no escopo da reengenharia de software:

- Capacidade de Codificação e Resolução de Problemas:** O *HumanEval* avalia a capacidade da IA de escrever algoritmos básicos baseados em descrições textuais, cenário em que todos os modelos apresentaram quase excelência (acima de 97%). No entanto, a reengenharia exige a resolução de problemas complexos e contemporâneos. Para isso, o *LiveCodeBench* (um benchmark atualizado continuamente com problemas reais) é a métrica mais confiável. Neste quesito, o DeepSeek R1 (77.0%) demonstra uma superioridade esmagadora sobre o Grok 3 mini (69.6%) e o GPT-5 mini (60.0%), provando ter a melhor capacidade de interpretação de código complexo.
- Janela de Contexto e Escalabilidade:** A refatoração de sistemas exige que a IA analise a classe atual e suas dependências arquiteturais. O Grok 3 mini oferece um milhão de tokens de contexto, permitindo injetar bibliotecas inteiras em uma única requisição. O GPT-5 mini oferece confortáveis 400k. Em contrapartida, os 128k do DeepSeek R1 representam um gargalo severo, limitando a quantidade de arquivos que podem ser analisados simultaneamente.
- Latência e Velocidade de Inferência:** Um fator determinante para a usabilidade de uma ferramenta de reengenharia é o tempo de resposta. Para gerar 500 to-

kens, o GPT-5 mini é o mais rápido em tempo absoluto (6.50 segundos). O Grok 3 mini possui a maior taxa de transferência (209 t/s), entregando a resposta em 12.27 segundos. O DeepSeek R1, devido ao seu profundo processo de raciocínio, demora impraticáveis 89.94 segundos para gerar a mesma quantidade de tokens, impactando negativamente a produtividade do desenvolvedor.

4. **Eficiência de Custo Operacional (Blended 3:1):** A métrica *Blended 3:1* calcula um custo médio ponderado assumindo um uso típico de 3 tokens de entrada para 1 token de saída. O Grok 3 mini consolida-se como a opção mais econômica do mercado (\$0.35 por milhão de tokens), custando quase três vezes menos que o GPT-5 mini (\$1.05) e o DeepSeek R1 (\$0.96).

Conclusão da Análise do Benched.ai. A avaliação dos dados do *Benched.ai* revela um dilema técnico. Se o critério for estritamente a precisão e a capacidade de resolver problemas lógicos em códigos complexos, o vencedor é o DeepSeek R1, que obteve a nota mais alta no rigoroso teste *LiveCodeBench* (77.0%). Contudo, ao avaliar o “Projeto de Reengenharia” como um todo (que exige leitura de múltiplos arquivos, velocidade e baixo custo), o Grok 3 mini Reason apresenta os parâmetros mais sustentáveis: a maior janela de contexto (1M), a operação mais barata (\$0.35) e alta velocidade, mantendo uma taxa de código aceitável. Como a metodologia estabeleceu que a precisão intelectual é o fator de maior peso, o DeepSeek R1 é eleito o modelo vencedor segundo os parâmetros do portal *Benched.ai*. Esse resultado valida a hipótese levantada na estratégia prática: ele possui a maior potência intelectual para geração de códigos. Devemos agora, através dos ensaios práticos, verificar se as suas graves desvantagens de latência (89.94 s) e limite de contexto inviabilizarão a sua aplicação na reengenharia de fato, configurando assim o cenário de desempate direto contra o GPT-5 mini (vencedor do *Artificial Analysis*).

4.1.2.5 Ensaios Práticos: Avaliação de Refatoração em Nível de Classe

Após a análise de dados das fontes de benchmarking, definiu-se que apenas o GPT-5 mini e o DeepSeek R1 seriam submetidos aos ensaios práticos. O *Grok 3 mini Reason*, embora tenha apresentado a maior janela de contexto e o menor custo operacional, foi descartado por não vencer em nenhuma das duas plataformas (*Artificial Analysis* e *Benched.ai*) nos critérios prioritários definidos pela metodologia, quais sejam, precisão e capacidade de raciocínio sobre código complexo. Sua eliminação respeita o critério de maior peso estabelecido na Subseção 4.1.1 (*Filtragem e Seleção dos Modelos Candidatos*): a precisão arquitetural prevalece sobre vantagens em janela e custo. Nesta etapa, utilizam-se os testes práticos em Java para responder à questão norteadora levantada na subseção anterior: “Apesar do modelo DeepSeek

R1 apresentar a maior potência intelectual, ele é, de fato, o mais capacitado para a geração e refatoração de códigos? Em caso afirmativo, a sua limitação de contexto e sua baixa velocidade comprometem a eficiência da reengenharia em nível de classe?”

4.1.2.6 Metodologia de Análise dos Resultados

Para garantir a imparcialidade e a profundidade técnica na avaliação dos códigos refatorados, adotou-se uma abordagem híbrida de correção. A análise dos ensaios práticos foi conduzida com o feedback avançado do modelo de Inteligência Artificial Gemini 3.1 Pro e auxílio de inspeção humana qualificada.

A escolha do Gemini 3.1 Pro para o papel de “avaliador auxiliar” fundamenta-se em seu alto desempenho técnico nas métricas globais. Segundo o portal *Artificial Analysis* (Artificial Analysis, 2025), este modelo possui um *Coding Index* de 56 pontos, consolidando-se como a segunda IA mais capacitada do mundo para a análise e geração de código. É imperativo ressaltar que o modelo líder do ranking neste quesito, o ChatGPT-5.4 xxhigh (57 pontos), foi intencionalmente descartado desta etapa. Como esse modelo compartilha a mesma arquitetura e motor (*engine*) que o GPT-5 mini, um dos sujeitos em teste na pesquisa, a sua utilização como juiz caracterizaria um viés técnico de validação, podendo comprometer a isenção da análise. Dessa forma, a combinação do olhar crítico humano com a capacidade analítica neutra do Gemini 3.1 Pro assegura o rigor científico necessário para classificar com precisão o desempenho dos modelos concorrentes.

4.2 Resultados e Discussão

Nesta seção, são apresentados os dados obtidos nos testes de reengenharia realizados com os modelos selecionados. Para garantir a clareza da análise, os resultados são avaliados conforme os critérios definidos abaixo:

4.2.1 Critérios de Avaliação de Resolução

- **Total:** A IA identificou corretamente todos os antipadrões e bugs, propondo uma solução que cumpre perfeitamente os requisitos de *Clean Code* e arquitetura Java.
- **Parcial:** A IA demonstrou pontos positivos e corrigiu parte do problema, porém falhou em algum detalhe técnico, ignorou um *code smell* secundário ou apresentou uma solução menos robusta em comparação ao outro modelo.

- **Falhou:** A IA não foi capaz de identificar o problema central do código ou propôs uma solução que mantém a falha original.

4.2.2 Análise dos Ensaios (Nível de Classe)

Classe 1: Problema de Encapsulamento e Inconsistência de Estado

- **GPT-5 mini:**
 - **Tempo de resposta:** 27.4 s.
 - **Uso de tokens:** 808 tokens.
 - **Resolução:** Parcial.
 - **Abordagem:**
 - * *Acertos:* Cumpriu o prompt com perfeição. Criou atributos privados, gerou *getters/setters* e colocou as validações solicitadas (idade negativa e o '@' no e-mail). O código funciona e compila.
 - * *Erros:* (no contexto de Reengenharia): Manteve o `System.out.println` dentro da classe. Embora o código esteja encapsulado, ele ainda está fortemente acoplado à saída padrão do sistema (I/O). Se a classe fosse usada em uma API Web (onde não há console), esse método seria inútil.
 - * *Justificativa Acadêmica:* A explicação é correta e didática, mas superficial. Descreve exatamente o que fez linha por linha, sem conectar isso a um panorama maior de design de software. Portanto, apesar de cumprir a proposta, pecou em detalhes importantes.
- **DeepSeek R1:**
 - **Tempo de resposta:** 41.41 s.
 - **Uso de tokens:** 1.648 tokens.
 - **Resolução:** Total.
 - **Abordagem:** O grande diferencial: a IA percebeu que a classe `Usuario` é uma entidade de domínio e, portanto, não deveria ter a responsabilidade de imprimir coisas no console. Removeu o `System.out.println`, alterou o método `exibirPerfil()` para retornar uma `String` formatada e sobrescreveu o `toString()`. Isso é a aplicação perfeita do Princípio da Responsabilidade Única (SRP). Robustez: usou o `.trim()` para evitar que *strings* cheias de espaços em branco passassem pela validação e limitou a idade a 150 anos (um toque excelente de domínio).

Justificativa Acadêmica: A análise gerada apresenta densidade técnica notável. O modelo cita *Design by Contract*, *Layer Supertype* e *Clean Architecture*, articulando o “porquê” arquitetural das alterações, e não apenas o “o quê” sintático.

Classe 2: Vazamento de Estado Mutável (Mutable State Leak)

- **GPT-5 mini:**

- **Tempo de resposta:** 10.12 s.
- **Uso de tokens:** 1037 tokens.
- **Resolução:** Parcial.
- **Abordagem:**
 - * *Acertos:* Identificou e corrigiu o vazamento da lista utilizando `Collections.unmodifiableList(itens)`. Também percebeu que `java.util.Date` é mutável e aplicou o padrão de *Defensive Copying* (Cópia Defensiva) no método `getDataCriacao()`, retornando um novo objeto `Date`. Além disso, extraiu o “magic number” para uma constante e separou as validações.
 - * *Erros:* (no contexto de Reengenharia): Cometeu um erro clássico e sutil. Embora tenha protegido o *getter* da data, esqueceu de proteger o construtor. Ao fazer `this.dataCriacao = dataCriacao;`, a classe armazena a referência original. Se o código cliente que instanciou o `Pedido` modificar aquele objeto `Date` logo em seguida, a data dentro do pedido será alterada, burlando o encapsulamento.
 - * *Justificativa Acadêmica:* A IA aplicou boas práticas de *Clean Code* (métodos pequenos, constantes), mas falhou em dominar completamente o conceito de imutabilidade de objetos legados. A explicação fornecida foca muito na sintaxe e pouco na raiz do problema arquitetural. É um código que aparenta estar seguro, mas ainda possui uma brecha crítica de vazamento de estado.

- **DeepSeek R1:**

- **Tempo de resposta:** 105.27 s.
- **Uso de tokens:** 2947 tokens.
- **Resolução:** Total.
- **Abordagem: O grande diferencial:** a IA percebeu que tentar fazer cópias defensivas de `java.util.Date` é uma prática legada e propensa a erros

(como o que o GPT-5 Mini cometeu). Em vez disso, fez uma reengenharia real e substituiu a API de datas por `java.time.LocalDateTime`, que é nativamente imutável. Isso elimina a raiz do problema de vazamento por design, e não por “remendos”.

- * **Robustez:** Além de usar `Collections.unmodifiableList()`, a IA transformou a classe em `final`, prevenindo que subclasses quebrem o encapsulamento sobrescrevendo métodos. Também adicionou validações rigorosas de estado inicial no construtor (evitando *strings* nulas/vazias), aplicando o conceito de *Fail-Fast* (falhar o mais rápido possível).
- * **Justificativa Acadêmica:** A análise gerada pelo DeepSeek R1 é muito madura. Não apenas lista o que fez, mas justifica arquiteturalmente o porquê. Cita a substituição de classes legadas, *thread-safety*, proteção de estado interno e segurança de herança (ao selar a classe). Mostra um entendimento profundo de que, em orientação a objetos moderna, a melhor forma de evitar vazamento de estado é trabalhar com componentes imutáveis desde a sua concepção.

Classe 3: Condição de Corrida em Concorrência (Race Condition)

- **GPT-5 mini:**

- **Tempo de resposta:** 10.16 s.
- **Uso de tokens:** 1041 tokens.
- **Resolução:** Falhou.
- **Abordagem:**
 - * **Acertos:** A nível estrutural de *Design de Software* (em um cenário *single-thread*), a IA demonstrou conhecimento ao aplicar o padrão Strategy (criando a interface `Operacao` e as classes `Saque` e `Deposito`). Isso evitou blocos `if/else` e deixou a classe aberta para extensão (*Open/Closed Principle*).
 - * **Erros:** (no contexto de Reengenharia): Ignorou totalmente o requisito de concorrência. Ao manter o `Thread.sleep()` e separar a validação (`operacao.valida`) da execução (`operacao.aplicar`), preservou o exato antipadrão Check-then-Act. Se duas *threads* tentarem sacar ao mesmo tempo, ambas passarão pela validação antes de o saldo ser atualizado, resultando em saldo negativo. O padrão Strategy aqui configurou um *Overengineering* (superengenharia) que mascarou o problema real.

- * *Justificativa Acadêmica*: A IA focou exclusivamente em padrões de projeto orientados a objetos (Gang of Four), negligenciando requisitos não funcionais de segurança de *threads* (*Thread Safety*). A abstração adicionou complexidade sem resolver a falha de sincronização.

- **DeepSeek R1:**

- **Tempo de resposta**: 78.20 s.

- **Uso de tokens**: 2314 tokens.

- **Resolução**: Falhou.

- **Abordagem**:

- * *Acertos*: Fez um excelente trabalho de *Clean Code* clássico. Transformou o titular em `final`, consolidou validações e corrigiu a semântica de exceções (trocando `IllegalArgumentException` por `IllegalStateException` quando o estado do saldo é insuficiente).

- * *Erros*: (no contexto de Reengenharia): A IA justificou a remoção do `Thread.sleep()` alegando tratar-se de um “detalhe de infraestrutura”. Esta decisão, entretanto, não ataca o problema na raiz: remover o *sleep* apenas reduz a janela de tempo da condição de corrida, sem eliminar a sua causa. Além disso, assumiu erroneamente que os operadores `+=` e `-=` resolvem algo, mas em Java, operações sobre primitivos `double` de 64 bits não são atômicas. O padrão *Check-then-Act* continua existindo entre o método `verificarSaldoSuficiente()` e a linha `saldo -= valor;`.

- * *Justificativa Acadêmica*: A análise gerada é muito teórica sobre *Domain-Driven Design* (DDD) e princípios SOLID, mas peca gravemente na base de programação concorrente. A IA falhou em reconhecer que, em ambientes *multithread*, a ausência de primitivas de sincronismo (`synchronized`, *Locks* ou variáveis atômicas) corrompe o estado da entidade, invalidando qualquer garantia que o DDD tente propor.

Classe 4: Violação do Princípio da Substituição de Liskov (LSP)

- **GPT-5 mini:**

- **Tempo de resposta**: 8.83 s.

- **Uso de tokens**: 907 tokens.

- **Resolução**: Falhou.

- **Abordagem**:

- * *Acertos:* A IA aplicou boas práticas básicas de *Clean Code* no escopo da classe individual. Introduziu construtores obrigatórios para inicializar o estado e adicionou validações defensivas (`largura <= 0`) para impedir a criação de formas com dimensões negativas ou nulas.
- * *Erros:* (no contexto de Reengenharia): A IA falhou miseravelmente em identificar o problema arquitetural. Manteve a herança (`Quadrado extends Retangulo`) e, pior, manteve a sobrescrita dos métodos `setLargura` e `setAltura` que forçam as duas dimensões a serem iguais. Se o cliente usar essa classe polimorficamente como um `Retangulo`, o contrato continuará sendo quebrado.
- * *Justificativa Acadêmica:* Este resultado evidencia uma falha grave na compreensão semântica de Design de Software por parte do modelo. Ele se limitou a fazer “maquiagem de código” (adicionar construtores e exceções), mas ignorou o princípio fundamental do LSP: subtipos não podem alterar as garantias de comportamento esperadas de seus supertipos.

- **DeepSeek R1:**

- **Tempo de resposta:** 32.05 s.
- **Uso de tokens:** 2240 tokens.
- **Resolução:** Total.
- **Abordagem: O grande diferencial:** a IA identificou o problema de imediato e cortou o mal pela raiz: quebrou a herança entre `Quadrado` e `Retangulo`. Em vez disso, extraiu o comportamento comum para uma interface `Forma` com o método `getArea()`, permitindo que ambas as classes sejam usadas polimorficamente sem compartilharem uma árvore de herança inadequada.
- * **Robustez:** Além de resolver o LSP, a IA elevou o nível do design aplicando Imutabilidade. Ao remover completamente os *setters* e tornar as classes *final*, eliminou qualquer possibilidade de o estado ficar inconsistente após a criação. Também adicionou um padrão *Factory* (`FormaFactory`) para centralizar a criação dos objetos.
- * *Justificativa Acadêmica:* A análise fornecida pela IA é impecável. Ela explica exatamente que a matemática não dita as regras da orientação a objetos quando o estado mutável está envolvido. A introdução de uma interface comum (Segregação de Interfaces) e a aplicação da imutabilidade provam uma compreensão madura de que a melhor maneira de evitar violações de comportamento é reduzir a superfície de mutação do objeto.

Classe 5: Alta Complexidade Ciclométrica e Violação de SRP/OCP

- **GPT-5 mini:**

- **Tempo de resposta:** 15.39 s.
- **Uso de tokens:** 1437 tokens.
- **Resolução:** Parcial.
- **Abordagem:**
 - * **Acertos:** A IA conseguiu resolver a alta complexidade ciclométrica aplicando o padrão Strategy (DescontoStrategy e suas implementações). Inovou ao usar um Map no construtor para mapear a *String* do tipo de cliente diretamente para a estratégia correspondente, eliminando por completo a cascata de if/else.
 - * **Erros:** (no contexto de Reengenharia): A IA falhou drasticamente na resolução do Princípio da Responsabilidade Única (SRP). Ela apenas extraiu o `System.out.println` para métodos privados (`exibirRecibo` e `enviarEmailConfirmacao`). A classe `ProcessadorDeDescontos` continua tendo a responsabilidade de imprimir coisas no console e “enviar e-mails”. Além disso, o construtor da classe instancia o mapa com as estratégias concretas (“hardcoded”); logo, se houver um novo cliente “ATACADO”, o programador terá que abrir e modificar esta classe, o que mantém a violação do OCP em nível de instanciação.
 - * **Justificativa Acadêmica:** A IA aplicou *Cosmetic Refactoring* (Refatoração Cosmética). Melhorou a legibilidade extraindo código procedural para métodos e usando um Map, mas a classe continua fortemente acoplada a I/O e a implementações concretas, desrespeitando o Princípio da Inversão de Dependência (DIP).

- **DeepSeek R1:**

- **Tempo de resposta:** 125.55 s.
- **Uso de tokens:** 4027 tokens.
- **Resolução:** Total.
- **Abordagem: O grande diferencial:** esta IA executou uma reengenharia arquitetural completa. Não apenas aplicou o padrão Strategy para os descontos, mas identificou as violações de SRP e extraiu os comportamentos para as classes `ReciboService` e a interface `Notificador` (com a implementação `EmailNotificador`).

- * **Robustez:** Para unir tudo isso sem violar o acoplamento, a IA utilizou Injeção de Dependência (DI). O `ProcessadorDeDescontos` agora recebe pelo seu construtor uma `DescontoStrategyFactory` (para resolver o OCP da criação de instâncias), um serviço de recibo e um notificador. A classe agora faz apenas uma coisa: orquestra o fluxo de alto nível.
- * **Justificativa Acadêmica:** A justificativa entregue pela IA é digna de um livro de arquitetura. Cita a remoção de acoplamento com o console e levanta um ponto vital: testabilidade. Ao injetar dependências (como o `Notificador`), torna-se possível usar objetos *Mock* em testes unitários para testar o processador sem disparar e-mails de verdade.

4.2.2.1 Conclusão da Avaliação de Nível de Classe

A análise dos cinco cenários de reengenharia evidencia diferenças significativas no comportamento, na capacidade analítica e no custo computacional entre os dois modelos avaliados. O DeepSeek R1 apresentou desempenho superior em relação ao GPT-5 mini nos critérios estabelecidos, consolidando-se como o modelo selecionado nesta etapa.

Em termos quantitativos de resolução, o DeepSeek R1 obteve a classificação Total em 4 dos 5 ensaios (80% de sucesso pleno) e falhou em apenas 1 (20%). Por outro lado, o GPT-5 mini não obteve nenhuma resolução Total (0%), alcançando resultados Parciais em 3 cenários (60%) e Falhando em 2 (40%).

Observa-se um claro *trade-off* entre tempo de resposta e a profundidade arquitetural da refatoração. O GPT-5 mini foi consistentemente mais rápido, com tempos de inferência variando de 8.83 s a 27.4 s, e altamente econômico, com uma média em torno de 1.000 tokens por ensaio. Contudo, essa velocidade traduziu-se em refatorações superficiais. O modelo limitou-se a aplicar *Cosmetic Refactoring* (refatoração cosmética), focando em correções de sintaxe e validações primárias, mas ignorando violações arquiteturais graves, como o Princípio da Substituição de Liskov (LSP) e o vazamento de estado mutável.

Em contrapartida, o DeepSeek R1 exigiu consideravelmente mais tempo de processamento, com tempos variando de 32.05 s a 125.55 s na Classe 5, e um consumo de tokens até três vezes maior (pico de 4.027 tokens). No entanto, a análise qualitativa dos códigos demonstra que o custo extra de tempo e processamento foi plenamente justificado pela altíssima qualidade do código gerado. O modelo ultrapassou a simples correção de bugs visíveis e assumiu uma postura de Arquiteto de Software, reestruturando o domínio da aplicação rigorosa de Imutabilidade (substituição de `Date` por

LocalDateTime), Segregação de Interfaces, Injeção de Dependências e aplicação sólida dos princípios S.O.L.I.D.

O único viés negativo compartilhado ocorreu no ensaio da Classe 3 (Condição de Corrida), no qual ambas as IAs falharam. Este dado empírico é um achado crítico para a pesquisa, evidenciando que as IAs generativas atuais são exímias em reconhecer e aplicar padrões estruturais estáticos, mas ainda apresentam severa “cegueira arquitetural” perante problemas dinâmicos de concorrência e primitivas de sincronização.

Apesar desse ponto de atenção na programação *multithread*, a capacidade do DeepSeek R1 de transcender a sintaxe procedimental e reconstruir as classes sob verdadeiros paradigmas de engenharia de software o define, de forma irrefutável, como a IA vencedora e mais recomendada para a etapa de reengenharia em nível de classe.

4.2.2.2 Unificação da Estratégia e Redefinição de Escopo

Diante da disparidade qualitativa observada nos ensaios de nível de classe, compreendeu-se que a bifurcação originalmente planejada entre IA de Classe e IA de Projeto tornou-se prescindível para o prosseguimento deste trabalho. A premissa fundamental desta pesquisa é a reengenharia de software, processo que exige, acima de tudo, integridade estrutural e segurança lógica. Os testes demonstraram que modelos otimizados para velocidade, como o GPT-5 mini, operam em um nível de abstração superficial, o que comprometeria a confiabilidade de um sistema documentado de forma automatizada.

Dessa forma, como o foco central reside na potência intelectual e na capacidade de raciocínio arquitetural, não se justifica submeter o projeto a uma nova rodada de testes de larga escala com modelos que já falharam em princípios básicos de S.O.L.I.D. e encapsulamento. A latência superior do DeepSeek R1 é aceita como um custo operacional necessário, visto que a “economia de tempo” oferecida pelos demais modelos resultaria em um “débito técnico” gerado pela própria IA. Portanto, a estratégia de IA para este TCC fica unificada sob o modelo DeepSeek R1 para todas as etapas de reengenharia.

4.3 Conclusão da Seleção do Modelo

A jornada de seleção do modelo de Inteligência Artificial para este projeto de reengenharia conclui-se com a vitória técnica do DeepSeek R1. Embora os portais de benchmarking tenham apontado o GPT-5 mini como um forte competidor em métricas sintéticas (como o *Coding Index* do *Artificial Analysis*), a realidade dos ensaios práticos

cos em Java revelou que a profundidade analítica do DeepSeek R1 é superior para cenários de reestruturação de código legado.

Em suma, a escolha do DeepSeek R1 fundamenta-se na sua postura de “Arquiteto”, capaz de identificar e corrigir problemas que fogem à simples sintaxe, como o vazamento de estado mutável e violações de princípios de design. Esta decisão prioriza a qualidade final do software e a redução de *bugs* pós-refatoração, estabelecendo uma base sólida para a implementação da ferramenta de reengenharia proposta por este trabalho.

O modelo selecionado nessa etapa é o mesmo que executará, no Capítulo 5, os três papéis previstos no ciclo de autocrítica fundamentada (*gerador*, *juiz* e *revisor*), com calibração de temperatura conservadora discutida na Seção 2.4 do Capítulo 2. A capacidade de compreensão arquitetural demonstrada aqui é o fundamento técnico que sustenta tanto a etapa de geração inicial dos artefatos de *Backlog* e *Sprint Planning* quanto as etapas subsequentes de avaliação contra o Guia do Scrum 2020 e de correção fundamentada por citação literal.

5 O DocScrum

Este capítulo apresenta o sistema desenvolvido como produto técnico deste Trabalho de Conclusão de Curso, denominado *DocScrum*. São descritos seu propósito, requisitos, arquitetura, tecnologias empregadas, componentes, fluxo de funcionamento e artefatos gerados. As decisões de projeto são justificadas com base nos objetivos de pesquisa apresentados nos capítulos anteriores e nas restrições da integração com Modelos de Linguagem de Grande Escala (LLMs, do inglês *Large Language Models*).

5.1 Visão Geral

O *docScrum* é uma aplicação web destinada à geração automática de artefatos de engenharia de software a partir da análise de código-fonte Java por meio de LLMs. O nome do sistema combina o termo *doc*, em referência à documentação automatizada, com a metodologia Scrum, refletindo seu objetivo de transformar código Java em artefatos para apoiar o desenvolvimento iterativo.

O sistema recebe como entrada um projeto Java, preferencialmente na forma de uma pasta contendo sua estrutura completa, embora também aceite arquivos `.java` individuais. A partir dessa entrada, gera quatro artefatos:

1. **Documentação técnica**, descrevendo arquitetura, padrões de projeto e decisões técnicas identificadas;
2. **Especificação de requisitos**, contendo requisitos funcionais e não funcionais extraídos da análise;
3. **Backlog ágil**, organizado em histórias de usuário com critérios de aceitação;
4. **Sprint Planning**, apresentado como um *roadmap* priorizado de implementação.

Além disso, para os artefatos *Backlog* e *Sprint Planning*, alinhados ao Guia do Scrum 2020, o sistema oferece um ciclo opcional de *validação e correção fundamentada*. A aplicação compara o artefato ao documento normativo, produz uma avaliação quantitativa em três dimensões (*Conformidade Estrutural*, *Cobertura Conceitual* e *Aderência Semântica*) com citações do Guia que justificam cada avaliação e permite sua reescrita com base nesse *feedback*. Esse processo materializa um padrão de *auto-*

crítica fundamentada em documento normativo, no qual o LLM atua, em chamadas distintas, como gerador, avaliador e revisor.

A motivação para o desenvolvimento decorre de que a documentação de sistemas legados e a elaboração inicial de *backlogs* para equipes ágeis demandam tempo significativo de profissionais experientes. O avanço dos LLMs permite delegar parte dessas atividades aos modelos generativos, mantendo o profissional humano como revisor e curador. A delimitação ao ecossistema Java justifica-se por sua ampla adoção em sistemas corporativos legados, nos quais o esforço de retrodocumentação é particularmente elevado.

5.2 Objetivos do Sistema

5.2.1 Objetivo Geral

Automatizar a geração dos quatro artefatos descritos anteriormente a partir da análise de projetos Java, oferecendo uma interface unificada para *upload*, processamento, validação e correção, reduzindo o esforço manual de documentação e apoiando processos de reengenharia ágil. O modelo de LLM adotado é o DeepSeek R1 (*deepseek-reasoner*), selecionado com base nos resultados empíricos apresentados no estudo de caso.

5.2.2 Objetivos Específicos

Os objetivos específicos que orientaram a construção do sistema foram:

- Disponibilizar uma interface de *upload* para projetos Java, aceitando pastas completas (caso prioritário) e arquivos `.java` individuais;
- Aplicar filtros automáticos para descartar arquivos que não sejam `.java` e diretórios irrelevantes ao código-fonte (como `target`, `.gradle` e `.idea`);
- Estruturar a saída do modelo em quatro seções delimitadas por marcadores textuais, garantindo o *parsing* determinístico;
- Exibir os artefatos em *Markdown* renderizado e permitir o *download* individual ou consolidado em PDF (com capa, sumário e seções);
- Validar os artefatos *Backlog* e *Sprint Planning* conforme o Guia do Scrum 2020, gerando uma avaliação quantitativa em três dimensões com citações que justifiquem cada nota;
- Permitir a correção automática de artefatos avaliados, instruindo o modelo a reescrevê-los com base no *feedback* da etapa de validação.

5.3 Requisitos do Sistema

5.3.1 Requisitos Funcionais

Tabela 8 – Requisitos funcionais do docScrum.

Identificador	Descrição
RF01	O sistema deve permitir o envio de uma pasta contendo a estrutura completa de um projeto Java por meio de seleção ou de arrastar-e-soltar.
RF02	O sistema deve permitir, alternativamente, o envio de arquivos <code>.java</code> individuais.
RF03	O sistema deve aceitar exclusivamente arquivos com extensão <code>.java</code> , descartando automaticamente quaisquer outros arquivos presentes no conjunto enviado.
RF04	O sistema deve descartar automaticamente diretórios irrelevantes ao código-fonte de projetos Java (<code>target</code> , <code>.gradle</code> , <code>build</code> , <code>.idea</code> , <code>.settings</code> , entre outros).
RF05	O sistema deve encaminhar o código consolidado ao modelo de LLM adotado para a execução da análise.
RF06	O sistema deve retornar quatro artefatos distintos a cada análise: documentação técnica, requisitos, <i>Backlog</i> ágil e <i>Sprint Planning</i> .
RF07	O sistema deve apresentar os artefatos em interface com abas e em formato <i>Markdown</i> renderizado.
RF08	O sistema deve permitir o <i>download</i> de cada artefato em formato PDF (<code>.pdf</code>), tanto individualmente quanto em arquivo consolidado contendo todos os quatro artefatos.
RF09	O sistema deve permitir o <i>download</i> em lote dos quatro artefatos.
RF10	O sistema deve oferecer alternância entre tema claro e tema escuro, com persistência da preferência do usuário.
RF11	O sistema deve permitir, para os artefatos <i>Backlog</i> e <i>Sprint Planning</i> , a execução de uma validação contra o Guia do Scrum 2020, retornando avaliação JSON com três dimensões (<i>Conformidade Estrutural</i> , <i>Cobertura Conceitual</i> e <i>Aderência Semântica</i>), nota geral, resumo textual e citações literais do Guia.
RF12	O sistema deve permitir o acionamento de uma correção fundamentada sobre um artefato avaliado, submetendo conteúdo original e <i>feedback</i> ao modelo e exibindo o <i>delta</i> quantitativo entre a nota inicial e a nota pós-correção.

Fonte: autoria própria.

5.3.2 Requisitos Não Funcionais

A Tabela 9 apresenta os requisitos não funcionais.

Tabela 9 – Requisitos não funcionais do docScrum.

Identificador	Categoria	Descrição
RNF01	Desempenho	O tamanho total dos arquivos enviados em uma análise não deve exceder 5 MB.
RNF02	Desempenho	O tamanho de cada arquivo individual aceito pelo <i>backend</i> não deve exceder 1 MB.
RNF03	Compatibilidade	O sistema deve aceitar exclusivamente arquivos com extensão <i>.java</i> , rejeitando automaticamente qualquer outro tipo de arquivo.
RNF04	Robustez	O sistema deve rejeitar automaticamente diretórios típicos de <i>build</i> e configuração do ecossistema Java, bem como diretórios de controle de versão.
RNF05	Segurança	As chaves de API do provedor de LLM devem residir exclusivamente no servidor, jamais expostas ao cliente.
RNF06	Usabilidade	A interface deve ser responsiva, adaptando-se a dispositivos móveis, <i>tablets</i> e <i>desktops</i> .
RNF07	Manutenibilidade	O sistema deve adotar separação clara entre <i>frontend</i> e <i>backend</i> , comunicando-se exclusivamente por HTTP REST.
RNF08	Extensibilidade	A arquitetura do <i>backend</i> deve permitir a inclusão de novos provedores de LLM sem alteração da interface pública da API.
RNF09	Auditabilidade	As notas atribuídas pelo serviço de validação devem ser acompanhadas de citações literais do Guia do Scrum como justificativa, garantindo rastreabilidade do julgamento.

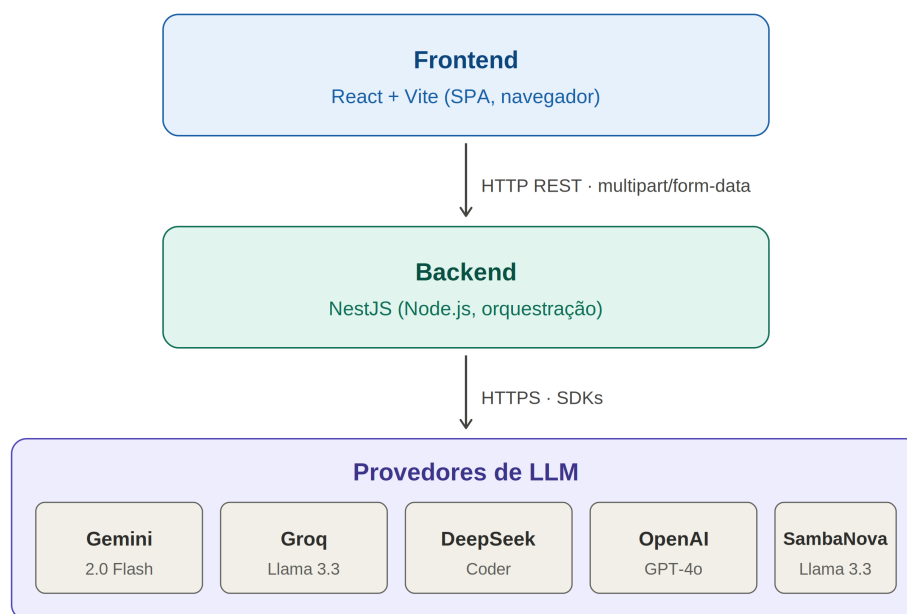
Fonte: autoria própria.

5.4 Arquitetura do Sistema

A arquitetura adotada segue o modelo cliente-servidor desacoplado, organizado em três camadas distintas: a camada de interface do usuário, executada no navegador; a camada de aplicação, que centraliza a orquestração das chamadas ao modelo de linguagem nas etapas de análise, validação e correção; e a camada de provedor

externo, composta pelo serviço de LLM adotado. A Figura 2 ilustra a organização da arquitetura.

Figura 2 – Visão em camadas da arquitetura do *docScrum*.



Fonte: autoria própria.

A camada de interface foi implementada como uma *Single Page Application* (SPA) construída em React com TypeScript, responsável pela coleta dos arquivos enviados pelo usuário, renderização dos resultados, gestão do estado local e exibição do painel de validação Scrum. A comunicação com a camada de aplicação ocorre exclusivamente por meio de requisições HTTP REST, com as cargas referentes ao *upload* trafegando como *multipart/form-data* e as respostas codificadas em JSON.

A camada de aplicação foi construída sobre o *framework* NestJS, executado em ambiente Node.js. Sua responsabilidade abrange quatro funções principais: (i) recepção e validação das requisições oriundas da interface, garantindo que apenas arquivos *.java* sejam considerados; (ii) construção do *prompt* estruturado de análise que será submetido ao modelo de linguagem; (iii) *parsing* da resposta retornada, dividindo-a nos quatro artefatos esperados; e (iv) execução do ciclo de validação e correção sobre os artefatos *Backlog* e *Sprint Planning*, com injeção integral do Guia do Scrum 2020 como *system message*.

A camada de provedor externo abriga o serviço de LLM adotado pelo sistema (o DeepSeek R1), acessado via HTTPS por meio de uma API REST compatível com o protocolo da OpenAI. A integração é centralizada na função utilitária `resolverProvedorLLM`,

que encapsula a configuração do cliente, do modelo e do rótulo exibido na interface. Essa centralização é o ponto de extensão que cumpre o requisito RNF08: a inclusão futura de um provedor alternativo compatível com o mesmo protocolo exige apenas a adição de uma ramificação condicional nessa função, sem qualquer alteração na interface pública da API.

5.4.1 Padrões Arquiteturais

O *backend* adota o padrão modular nativo do NestJS, organizando o código em torno de dois módulos funcionais: *AnalyzeModule*, responsável pela rota `POST /analyze`, e *ValidateModule*, responsável pelas rotas `POST /validate` e `POST /validate/correct`. O módulo raiz *AppModule* importa ambos e inicializa o *ConfigModule* em escopo global, conforme apresentado a seguir:

```
1 @Module({
2   imports: [
3     ConfigModule.forRoot({ isGlobal: true }),
4     AnalyzeModule,
5     ValidateModule,
6   ],
7 })
```

Ambos os módulos seguem a tríade *controller-service-module*, padrão recomendado pelo NestJS e aderente ao princípio de separação de responsabilidades. Os controladores limitam-se ao tratamento de entrada e saída HTTP, enquanto a lógica de negócio reside nas classes de serviço.

A interface, por sua vez, segue o padrão de componentes funcionais com *hooks* introduzido no React a partir da versão 16.8 e consolidado na versão 19.2.0 utilizada no projeto. O estado da aplicação é gerenciado localmente em cada página por meio de *useState*, dispensando bibliotecas externas de gerenciamento global de estado dada a baixa complexidade dos fluxos envolvidos.

5.5 Tecnologias Utilizadas

A escolha das tecnologias buscou equilibrar maturidade, produtividade do desenvolvedor e adequação ao domínio de aplicações orientadas a IA generativa. As Tabelas 10, 11 e 12 apresentam as tecnologias adotadas em cada camada do sistema.

Tabela 10 – Tecnologias utilizadas no *frontend*.

Tecnologia	Versão	Finalidade
React	19.2.0	Biblioteca para construção da interface declarativa
TypeScript	5.9.3	Tipagem estática sobre JavaScript
Vite	7.3.1	Empacotador de módulos e servidor de desenvolvimento
Tailwind CSS	4.2.1	Sistema de estilização baseado em classes utilitárias
React Router DOM	7.13.1	Roteamento entre páginas da SPA
React Markdown	10.1.0	Renderização de <i>Markdown</i> nas abas de visualização
jsPDF	3.0.x	Geração programática dos PDFs de <i>download</i>
Lucide React	0.577.0	Biblioteca de ícones vetoriais
clsx	2.1.1	Composição condicional de classes CSS

Fonte: autoria própria.

5.5.1 Frontend

A combinação React-Vite-TypeScript foi adotada por representar o estado da prática contemporânea no desenvolvimento de SPAs, oferecendo *Hot Module Replacement* eficiente, tipagem forte e bibliotecas maduras para roteamento, renderização de *Markdown* na interface e geração de PDF no lado do cliente. O Tailwind CSS foi escolhido pela velocidade de prototipagem e pelo suporte nativo a temas claro e escuro por meio da classe *dark*.

5.5.2 Backend

A adoção do NestJS deve-se à sua estrutura modular inspirada em *frameworks* como o Angular, ao suporte nativo a injeção de dependências e à compatibilidade direta com TypeScript. Tais características favorecem a manutenibilidade e a extensibilidade do sistema, atendendo aos requisitos não funcionais RNF07 e RNF08.

5.5.3 Provedor de LLM Adotado

O provedor de LLM adotado pelo sistema é o DeepSeek R1, acessado via API REST compatível com o protocolo da OpenAI. A seleção foi consolidada a partir dos resultados quantitativos apresentados no estudo de caso experimental, no qual o modelo demonstrou capacidade consistente de gerar artefatos ágeis a partir de código

Tabela 11 – Tecnologias utilizadas no *backend*.

Tecnologia	Versão	Finalidade
NestJS	11.0.1	<i>Framework</i> Node.js para construção de API REST modular
TypeScript	5.7.3	Tipagem estática
openai	6.32.0	SDK utilizado para acessar a API DeepSeek R1 (compatível com o protocolo OpenAI)
@nestjs/platform-express	11.1.16	Adaptador Express para o NestJS
multer	2.1.1	<i>Middleware</i> para tratamento de <i>uploads</i>
@nestjs/config	4.0.3	Carregamento e exposição de variáveis de ambiente
Jest	30.0.0	<i>Framework</i> de testes unitários

Fonte: autoria própria.

Java e de submeter-se à etapa de auto-crítica fundamentada com elevado *delta* médio entre as notas pré e pós-correção. A Tabela 12 sintetiza os parâmetros de integração.

Tabela 12 – Parâmetros de integração com o provedor de LLM adotado.

Parâmetro	Valor
Provedor	DeepSeek
Modelo	deepseek-reasoner (DeepSeek R1)
Protocolo	API compatível com OpenAI
SDK	openai (com baseUrl customizada)
baseUrl	https://api.deepseek.com
Temperatura (análise/validação)	0,2
Temperatura (correção)	0,3
Janela de contexto	128.000 <i>tokens</i>

Fonte: autoria própria.

A arquitetura do sistema, ainda assim, foi mantida desacoplada do provedor por meio da função utilitária `resolverProvedorLLM`, que centraliza a configuração do cliente, do modelo e do rótulo exibido na interface. Essa centralização atende ao requisito não funcional RNF08: a inclusão futura de um provedor alternativo compatível com o protocolo da OpenAI exige apenas a adição de uma ramificação condicional nessa função, sem qualquer alteração na interface pública da API.

5.5.4 Estratégia de *Prompting*

A qualidade dos artefatos produzidos depende diretamente da formulação dos *prompts* submetidos ao modelo. O sistema emprega três *prompts* distintos, cada um responsável por uma etapa do ciclo e dirigido ao modelo com configurações de temperatura específicas. A separação explícita dessas etapas materializa o papel triplo do LLM: *gerador*, *juiz fundamentado em documento normativo* e *revisor orientado por evidência citada*.

(i) **Prompt de Análise.** Atribui ao modelo o papel de Arquiteto de *Software* Sênior e instrui-o a produzir os quatro artefatos em uma única resposta, delimitados por marcadores textuais escolhidos por apresentarem baixa probabilidade de colisão com o conteúdo Java analisado: DOCUMENTACAO, REQUISITOS, BACKLOG e SPRINTS, no formato ===NOME===. Esse desenho minimiza o número de chamadas ao modelo (quatro artefatos em uma única requisição) e permite *parsing* determinístico da resposta por busca de *substring* entre marcadores consecutivos. A temperatura é fixada em 0,2, priorizando consistência e aderência ao formato solicitado.

(ii) **Prompt de Validação.** Carrega integralmente o Guia do Scrum 2020 em português (aproximadamente 27 mil caracteres) como *system message* (estratégia conhecida como *context stuffing*) e instrui o modelo a produzir uma avaliação em JSON estruturado contendo três dimensões (*Conformidade Estrutural*, *Cobertura Conceitual* e *Aderência Semântica*), cada uma com nota de 0 a 10, justificativa textual e *citação literal* do Guia que fundamenta o julgamento. A exigência de citação literal é o mecanismo que ancora o julgamento no documento normativo e mitiga o risco de alucinação. A temperatura permanece em 0,2.

(iii) **Prompt de Correção.** Recebe como entrada o artefato original e o objeto de *feedback* produzido na etapa anterior. Instrui o modelo a reescrever o artefato corrigindo *apenas* as lacunas apontadas no *feedback*, preservando o conteúdo bem-avaliado e a estrutura Markdown original. A introdução de elementos novos sem fundamentação no *feedback* é explicitamente proibida, restrição que mitiga o risco de *overengineering*. A temperatura é elevada para 0,3, valor ligeiramente mais alto justificado pela necessidade de flexibilidade na reescrita.

A opção por *context stuffing* no lugar de *Retrieval-Augmented Generation* (RAG) é deliberada: o Guia do Scrum tem dimensão moderada e cabe folgadoamente na janela de 128 mil *tokens* do modelo, dispensando a complexidade adicional de infraestrutura de recuperação semântica e garantindo que o validador opere sempre sobre o documento normativo *integral*, sem omissões decorrentes de *retrieval* imperfeito.

5.6 Componentes Principais

Esta seção descreve os componentes internos das camadas de interface e de aplicação, com ênfase nos elementos que materializam as funcionalidades centrais do sistema.

5.6.1 Camada de Interface

A camada de interface é organizada em torno de duas páginas principais e cinco componentes reutilizáveis, conforme detalhado nas subseções a seguir.

5.6.1.1 Página Inicial

A página inicial, mapeada para a rota `/`, possui caráter institucional. Apresenta uma seção *hero* com a chamada principal do sistema, descrição da forma de envio do projeto Java, explicação em três passos do funcionamento, listagem dos quatro artefatos gerados e botão de chamada para ação direcionando o usuário à página de análise. As animações de entrada (`animate-fade-in`, `animate-slide-up`) são definidas em CSS e ativadas no momento da montagem dos componentes.

5.6.1.2 Página de Análise

A página de análise, mapeada para a rota `/analisar`, concentra a interação principal do usuário com o sistema. O roteamento é configurado de forma declarativa por meio do componente `BrowserRouter` da biblioteca `React Router`:

```
1 <BrowserRouter>
2   <Routes>
3     <Route path="/" element={<Home />} />
4     <Route path="/analisar" element={<Analyze />} />
5   </Routes>
6 </BrowserRouter>
```

O estado interno da página é mantido por três variáveis reativas (`loading`, `error` e `result`), atualizadas conforme as etapas do ciclo de requisição. Ao acionar a análise, a página constrói dinamicamente um objeto `FormData` contendo os arquivos selecionados e o submete ao *endpoint* de análise:

```
1 const formData = new FormData()
2 files.forEach(f => formData.append('files', f))
```

Em caso de erro retornado pelo *backend*, a mensagem é extraída do corpo JSON da resposta; caso a desserialização falhe, exibe-se uma mensagem genérica acompa-

nhada do código HTTP. Este tratamento robusto garante que o usuário sempre receba realimentação inteligível.

5.6.1.3 Componente de Upload

O componente `InputModeSelector` é responsável pelo recebimento dos arquivos enviados pelo usuário. O *upload* aceita duas formas equivalentes do ponto de vista de processamento: a seleção de uma pasta inteira contendo a estrutura de um projeto Java (caso prioritário e típico de uso) e o envio de arquivos `.java` avulsos. Em ambos os casos, o componente aplica filtros locais antes de submeter os dados ao *backend*, evitando o tráfego desnecessário de conteúdo irrelevante. Os filtros baseiam-se em três critérios:

- Extensão obrigatória `.java`, sendo descartado qualquer arquivo cuja extensão difira;
- Conjunto de diretórios ignorados (`IGNORED_DIRS`), abrangendo as pastas geradas pelas ferramentas usuais do ecossistema Java, tais como `target` (Maven), `.gradle` e `build` (Gradle), `.idea` (IntelliJ IDEA), `.vscode`, `.settings` e `.classpath` (Eclipse), além de `.git`;
- Limite total (`MAX_TOTAL_SIZE`), fixado em 5 MB.

A função `filterProjectFiles` consolida a aplicação dos filtros:

```
1 function filterProjectFiles(fileList: FileList): File[] {  
2   return Array.from(fileList).filter(f => {  
3     const path = f.webkitRelativePath || f.name  
4     if (isIgnoredPath(path)) return false  
5     if (!isJavaFile(f.name)) return false  
6     return true  
7   })  
8 }
```

Caso o tamanho total dos arquivos selecionados ultrapasse o limite estabelecido, o componente exibe um aviso visual sem bloquear a interação, transferindo a decisão final ao usuário.

5.6.1.4 Componente de Resultados

O componente `ResultsView` é responsável pela exibição dos quatro artefatos retornados pelo *backend*. Implementa um sistema de abas que permite ao usuário alternar

entre os documentos sem recarregar a página e oferece duas modalidades de *download*: individual, em que apenas o artefato da aba ativa é exportado, ou consolidado, em que os quatro artefatos são reunidos em um único documento.

Os documentos circulam internamente, entre *backend* e *frontend*, em formato *Markdown*, escolhido por sua portabilidade e por permitir renderização nativa pelo componente `ReactMarkdown` nas abas de visualização. No momento do *download*, contudo, o *Markdown* é convertido em PDF apresentável por meio da biblioteca `jsPDF`, em uma construção *programática*: cada elemento do PDF (capa, cabeçalhos, parágrafos, listas, citações, blocos de código) é desenhado diretamente em coordenadas A4 a partir de um parser de Markdown próprio, sem qualquer intermediação de *rasterização* HTML por `canvas`. Essa decisão arquitetural elimina dependências pesadas de *headless browsers* ou de captura de DOM e produz PDFs com texto selecionável e copiável.

```
1 class PdfBuilder {
2   doc: jsPDF
3   y: number = MARGIN
4   // ...
5   addCover(title, subtitle, metaLines) { /* desenha capa centralizada */ }
6   addTOC(items) { /* desenha sumário numerado */ }
7   addSectionTitle(title) { /* desenha título de seção */ }
8   renderMarkdown(md) { /* parseia e desenha conteúdo */ }
9 }
10
11 function downloadConsolidatedPdf(result: AnalyzeResult) {
12   const b = new PdfBuilder()
13   b.addCover('Visão Geral do Sistema', 'Artefatos Ágeis Gerados', [
14     'Gerado por docScrum',
15     'Gerado em ${date}',
16   ])
17   b.addTOC(sections.map((s) => s.title))
18   sections.forEach((s) => {
19     b.addSectionTitle(s.title)
20     b.renderMarkdown(s.content)
21     b.doc.addPage()
22   })
23   b.save('visao-geral-do-sistema-artefatos.pdf')
24 }
```

O documento consolidado é estruturado em três blocos: (i) uma capa centralizada com o título fixo “Visão Geral do Sistema”, o subtítulo “Artefatos Ágeis Gerados” e as linhas de identificação (“Gerado por docScrum” e a data de geração); (ii) um sumário numerado com os títulos dos quatro artefatos; e (iii) o conteúdo de cada artefato em

sua própria seção, com título destacado por linha decorativa e quebra de página entre seções.

Além das operações de *download*, o componente exibe um botão “Verificar Eficácia” na barra superior dos resultados, aplicável aos artefatos *Backlog* e *Sprint Planning*. O acionamento desse botão abre o painel de validação Scrum, descrito na subseção seguinte. Coerentemente com a natureza estocástica do modelo, cada acionamento dispara uma nova chamada ao validador, sem qualquer reuso de avaliações anteriores, decisão que assegura que a nota exibida reflita sempre uma avaliação independente do artefato corrente.

5.6.1.5 Painel de Validação Scrum

O componente `ScrumValidationPanel` materializa o ciclo de validação e correção na interface. Acionado a partir do botão “Verificar Eficácia” da barra superior, abre-se como *drawer* lateral exibindo a nota média geral, as três notas dimensionais, o resumo textual produzido pelo validador e a lista de citações literais do Guia do Scrum que fundamentam o julgamento.

Quando o usuário aciona “Corrigir agora”, o componente dispara três operações encadeadas: (i) chamada ao *endpoint* de correção; (ii) atualização do conteúdo do artefato no estado da página; (iii) re-validação do conteúdo corrigido. O resultado é apresentado como *delta*, na forma `nota_antes → nota_depois`, acompanhado de um indicador colorido com o ganho de pontos. Durante toda a operação, o componente exibe estados intermediários (*validando*, *corrigindo*) para manter o usuário informado.

5.6.1.6 Cabeçalho e Sistema de Tema

O componente `Header` concentra a navegação entre páginas e o controle de tema. A preferência do usuário é persistida no `localStorage` sob a chave `docscrum-theme`, e a inicialização ocorre durante o efeito de montagem, considerando também a preferência do sistema operacional informada pela *media query* `prefers-color-scheme`:

```
1 useEffect(() => {
2   const saved = localStorage.getItem('docscrum-theme')
3   const prefersDark = window.matchMedia('(prefers-color-scheme: dark)').
    matches
4   const dark = saved === 'dark' || (!saved && prefersDark)
5   setIsDark(dark)
6   document.documentElement.classList.toggle('dark', dark)
7 }, [])
```

A alternância entre temas é efetivada pela aplicação ou remoção da classe `dark` no elemento raiz `documentElement`, comportamento aderente à convenção adotada pelo Tailwind CSS para temas escuros baseados em classe.

5.6.2 Camada de Aplicação

A camada de aplicação é composta por dois módulos NestJS, descritos nas subseções a seguir, e por um arquivo de *bootstrap* responsável pela inicialização do servidor.

5.6.2.1 Módulo de Análise

O módulo `AnalyzeModule` é o núcleo funcional do sistema. O controlador `AnalyzeController` expõe a rota `POST /analyze`, anotada com o interceptor `FilesInterceptor`, que delega ao Multer o tratamento dos arquivos enviados:

```
1 @UseInterceptors(FilesInterceptor('files', 500, { limits: { fileSize: 1024  
  * 1024 } })))
```

A configuração estabelece que cada requisição pode carregar até 500 arquivos e que o tamanho máximo individual de cada um é de 1 MB, complementando o limite total de 5 MB aplicado no *frontend*. O controlador realiza, em seguida, a validação dos parâmetros recebidos e a montagem da *string* final que será submetida ao serviço de análise.

Os arquivos enviados são concatenados por meio de uma sintaxe que preserva o caminho original como cabeçalho, recurso essencial para que o modelo identifique a estrutura de pacotes e diretórios do projeto Java:

```
1 code = files  
2   .map((f) => {  
3     const path = f.originalname;  
4     const fileContent = f.buffer.toString('utf-8');  
5     return '\n// === ARQUIVO: ${path} ===\n${fileContent}';  
6   })  
7   .join('\n');
```

O serviço `AnalyzeService` encapsula a lógica de comunicação com o provedor de LLM. A obtenção do cliente, do nome do modelo e do rótulo exibido na interface é delegada à função utilitária `resolverProvedorLLM`, exportada pelo próprio serviço e reutilizada pelo módulo de validação. Essa centralização garante que toda chamada a LLM no sistema use a mesma configuração canônica de provedor.

O tratamento de erros é uniforme: qualquer falha originada nos clientes externos é capturada e relançada como `InternalServerErrorException`, com mensagem que

identifica o provedor responsável pela falha:

```
1 } catch (err: unknown) {  
2   const message = err instanceof Error ? err.message : String(err);  
3   throw new InternalServerErrorException('Erro ao chamar IA (${provider}):  
    ${message}');  
4 }
```

5.6.2.2 Módulo de Validação

O módulo `ValidateModule` encapsula a lógica do ciclo de validação e correção. O controlador `ValidateController` expõe duas rotas: `POST /validate`, que avalia um artefato, e `POST /validate/correct`, que reescreve um artefato a partir do *feedback* de uma avaliação anterior.

O serviço `ValidateService` carrega, no momento de inicialização do servidor, o arquivo `scrum_guide_2020_pt-br.md` para a memória do processo. Esse conteúdo (de aproximadamente 27 mil caracteres) é injetado como *system message* em toda chamada de validação, segundo a estratégia de *context stuffing* descrita na Seção 5.5.4. O método `validar(tipo, conteudo)` retorna um objeto JSON com a seguinte estrutura: notas das três dimensões (*estrutural*, *cobertura*, *semantica*), nota média geral, resumo textual e arranjo de citações literais do Guia que fundamentam cada nota. O método `corrigirArtefato(tipo, conteudoOriginal, feedback)` retorna o conteúdo reescrito como *string* Markdown.

A robustez do *parsing* da resposta JSON é tratada por uma cadeia de até três tentativas: caso a resposta do modelo não seja JSON válido na primeira chamada, o sistema repete a requisição com mensagem complementar instruindo o modelo sobre o escape correto de aspas em campos textuais. Essa estratégia mitiga falhas pontuais de aderência ao formato sem onerar a experiência do usuário com erros visíveis.

5.6.2.3 Parsing da Resposta de Análise

A reconstrução das quatro seções a partir da resposta textual do modelo de análise é realizada pela função auxiliar a seguir, que localiza cada marcador por busca de *substring* e extrai o conteúdo entre marcadores consecutivos:

```
1 const extract = (marker: string, nextMarker: string | null): string => {  
2   const start = text.indexOf('==${marker}==');  
3   if (start === -1) return `Secao ${marker} nao encontrada na resposta da  
    IA.`;  
4   const contentStart = start + '==${marker}=='.length;  
5   const end = nextMarker ? text.indexOf('==${nextMarker}==') : text.  
    length;
```

```
6   return text.slice(contentStart, end === -1 ? text.length : end).trim();  
7 };
```

Caso um marcador esteja ausente da resposta (situação atribuível a falhas pontuais de aderência do modelo ao formato solicitado), a seção correspondente recebe uma mensagem de *fallback* que sinaliza explicitamente o ocorrido, evitando o aborto silencioso do processo.

5.6.2.4 Inicialização e Configuração do Servidor

O arquivo `main.ts` consolida a inicialização do servidor. As principais configurações aplicadas envolvem a habilitação seletiva de CORS (restringindo as origens permitidas às portas locais utilizadas em ambiente de desenvolvimento) e a elevação dos limites de corpo da requisição:

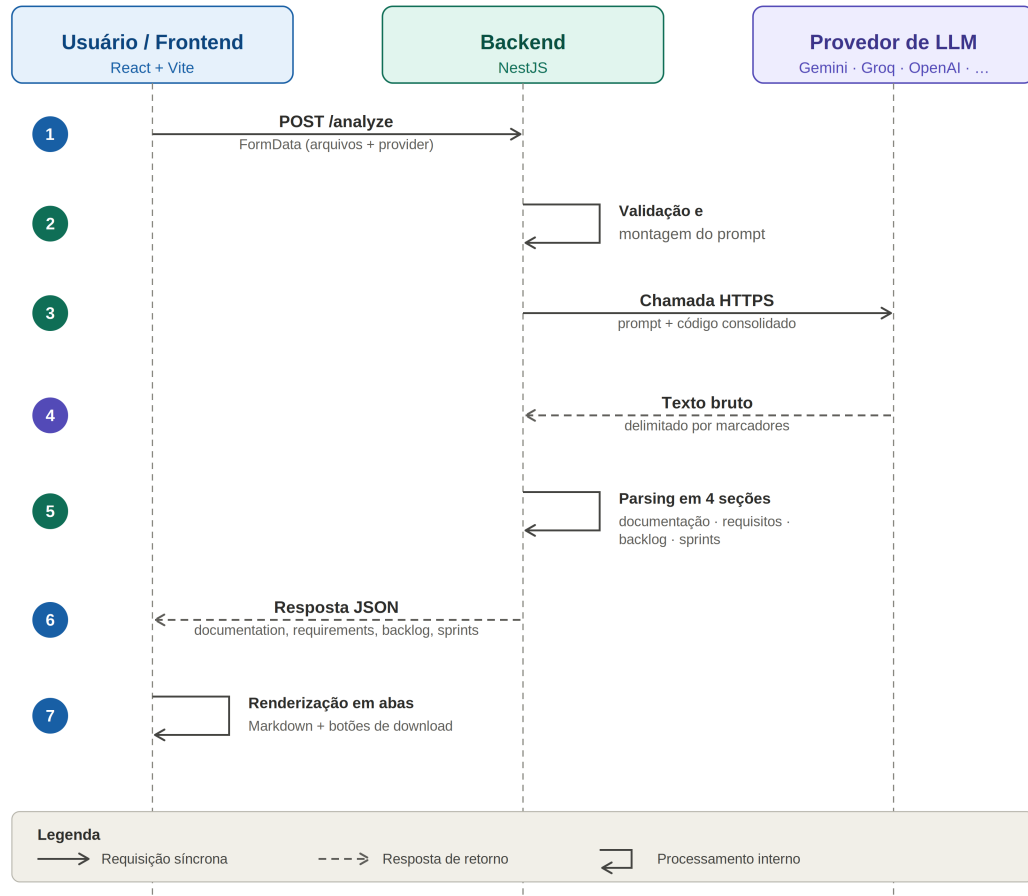
```
1 app.enableCors({  
2   origin: [ 'http://localhost:5173', 'http://localhost:5174', 'http://  
           localhost:4173' ],  
3   methods: [ 'GET', 'POST', 'OPTIONS' ],  
4   allowedHeaders: [ 'Content-Type' ],  
5 });  
6 const expressApp = app.getHttpAdapter().getInstance();  
7 const bodyParser = await import('express');  
8 expressApp.use(bodyParser.json({ limit: '10mb' }));  
9 expressApp.use(bodyParser.urlencoded({ limit: '10mb', extended: true }));
```

A elevação do limite para 10 MB justifica-se pelo tamanho potencial das requisições contendo múltiplos arquivos de código.

5.7 Fluxo de Funcionamento

O fluxo completo de uma análise compreende as etapas descritas a seguir, ilustradas na Figura 3. As etapas 1 a 9 correspondem ao fluxo principal de geração dos quatro artefatos; as etapas 10 a 12 correspondem ao ciclo opcional de validação e correção sobre os artefatos *Backlog* e *Sprint Planning*.

Figura 3 – Fluxo de funcionamento do *docScrum*, do acionamento da análise pelo usuário até a apresentação dos artefatos gerados e a execução opcional do ciclo de validação e correção.



Fonte: autoria própria.

A sequência de eventos é a seguinte:

1. O usuário acessa a página de análise e seleciona, no componente de *upload*, uma pasta contendo o projeto Java a ser analisado ou, alternativamente, arquivos *.java* individuais;
2. O *frontend* aplica filtros locais sobre o conjunto selecionado, descartando arquivos cuja extensão não seja *.java* e arquivos pertencentes a diretórios ignorados;
3. Ao acionar o botão “Analisar com IA”, o *frontend* constrói o objeto *FormData* com os arquivos remanescentes e dispara uma requisição *POST /analyze* ao *backend*;

4. O `AnalyzeController` valida os parâmetros recebidos e concatena o conteúdo dos arquivos enviados em uma única *string*, preservando o caminho original de cada arquivo como cabeçalho;
5. O `AnalyzeService` injeta o conteúdo consolidado no *prompt* estruturado de análise e dispara a chamada ao DeepSeek R1;
6. O provedor de LLM processa a requisição e retorna a resposta como texto delimitado pelos quatro marcadores;
7. O `AnalyzeService` realiza o *parsing* da resposta, extraindo os quatro artefatos individualmente;
8. O *backend* retorna ao *frontend* um objeto JSON com os campos `documentation`, `requirements`, `backlog` e `sprints`;
9. O componente `ResultsView` renderiza cada artefato em uma aba específica, disponibilizando os botões de *download* e, para *Backlog* e *Sprints*, o botão “Verificar Eficácia”;
10. Opcionalmente, o usuário aciona “Verificar Eficácia” sobre o *Backlog* ou o *Sprint Planning*; o *frontend* dispara `POST /validate` ao *backend*;
11. O `ValidateService` injeta o Guia do Scrum 2020 como *system message* e submete o conteúdo do artefato ao modelo, retornando ao *frontend* a avaliação estruturada em três dimensões com citações literais do Guia, exibida pelo componente `ScrumValidationPanel`;
12. Caso o usuário acione “Corrigir agora”, o *frontend* dispara `POST /validate/correct`, atualiza o artefato exibido com a versão corrigida e re-executa a validação para apresentar o *delta* de notas (antes → depois) ao usuário.

Durante o intervalo entre as etapas 3 e 8, o componente `LoadingState` exibe um indicador de carregamento animado, mantendo o usuário informado quanto ao andamento da operação. Em caso de erro em qualquer etapa do *backend*, a mensagem é repassada ao *frontend* e renderizada em uma faixa visual destacada na página de análise.

5.8 Artefatos Gerados

Cada execução do sistema produz quatro artefatos independentes, descritos a seguir.

Documentação Técnica. Apresenta uma síntese da arquitetura identificada no código analisado, contemplando padrões arquiteturais, padrões de projeto, dependências externas e decisões técnicas relevantes. Destina-se a equipes que precisam *on-boardar* novos desenvolvedores em sistemas legados ou que necessitam de documentação retrospectiva para fins de auditoria ou manutenção.

Especificação de Requisitos. Reúne requisitos funcionais e não funcionais inferidos a partir do comportamento do código. A inferência baseia-se em padrões observáveis como rotas expostas, validações aplicadas, tratamentos de erro e dependências externas. O artefato é especialmente útil em situações em que o código antecede a documentação formal (cenário comum em projetos com forte cultura de prototipagem).

Backlog Ágil. Apresenta histórias de usuário no formato consagrado pela literatura de metodologias ágeis (“Como [perfil], desejo [funcionalidade], para que [benefício]”), acompanhadas de critérios de aceitação. Este artefato é elegível ao ciclo de validação contra o Guia do Scrum 2020, podendo ser avaliado nas dimensões de *Conformidade Estrutural*, *Cobertura Conceitual* e *Aderência Semântica* e, se desejado, submetido a correção fundamentada que preserve o conteúdo bem-avaliado e enderece exclusivamente as lacunas apontadas pelo validador. O *backlog* gerado pode servir tanto como ponto de partida para refinamento por uma equipe Scrum quanto como base para o levantamento inicial de funcionalidades em projetos de reengenharia.

Sprint Planning. Organiza as histórias de usuário em *sprints* sequenciais, propondo uma ordem de implementação fundamentada em dependências técnicas e em prioridades funcionais inferidas. Como o *Backlog*, também é elegível ao ciclo de validação e correção contra o Guia do Scrum. Funciona como *roadmap* inicial passível de revisão pelo *Product Owner* e pelo *time* de desenvolvimento.

Internamente, os artefatos circulam em formato *Markdown*, escolhido por sua portabilidade, por ser interpretado nativamente por plataformas como GitHub, GitLab, Notion e Confluence e por permanecer legível mesmo na ausência de renderizador. Para a entrega final ao usuário, contudo, o sistema converte o *Markdown* em PDF apresentável: o *download* individual exporta o artefato da aba ativa, enquanto o *download* consolidado reúne os quatro artefatos em um único documento com capa (contendo o título fixo “Visão Geral do Sistema”, o subtítulo “Artefatos Ágeis Gerados” e a identificação “Gerado por docScrum”), sumário numerado e seções delimitadas por quebra de página, formato apropriado tanto para arquivamento corporativo quanto para apresentações a *stakeholders*.

5.9 Considerações de Implementação

Esta seção sintetiza decisões de implementação relevantes que não se enquadram nas seções anteriores e cuja documentação contribui para a reprodutibilidade do trabalho.

Gerenciamento de chaves de API. A chave de API do provedor de LLM é lida de variável de ambiente carregada a partir de um arquivo `.env` no diretório do *backend*. O `ConfigModule` do NestJS realiza o carregamento global. Em nenhum ponto do sistema a chave trafega pela rede ou é exposta ao cliente, atendendo ao requisito não funcional RNF05. A ausência de chave válida resulta em exceção lançada pela função `resolverProvedorLLM`, com mensagem explicativa.

Política de CORS. A política de CORS aplicada é restritiva por padrão, autorizando apenas as origens locais utilizadas em ambiente de desenvolvimento:

- `http://localhost:5173`
- `http://localhost:5174`
- `http://localhost:4173`

Os métodos são limitados a GET, POST e OPTIONS, com o cabeçalho `Content-Type` como o único explicitamente permitido. Para implantação em ambiente de produção, esta política precisa ser ajustada para incluir o domínio público do *frontend*.

Limites de tamanho. O sistema aplica limites de tamanho em três níveis distintos: o *frontend* rejeita conjuntos de arquivos cujo total exceda 5 MB; o Multer, no *backend*, rejeita arquivos individuais maiores que 1 MB; e o *body parser* do Express limita o corpo total da requisição em 10 MB. Tal estratificação reflete a separação de responsabilidades entre as camadas, permitindo que cada uma falhe próximo à origem do problema.

Validação contra documento normativo. O Guia do Scrum 2020 (versão oficial em português) é carregado uma única vez na inicialização do `ValidateService` e residente em memória durante todo o tempo de execução do processo. Não há *retrieval* dinâmico nem fragmentação por *chunks*: a estratégia adotada é a de *context stuffing* integral, conforme justificado na Seção 5.5.4. Para alterar o documento normativo de referência (por exemplo, para validar artefatos contra a ISO/IEC ou contra padrões internos de uma organização), basta substituir o arquivo *Markdown* carregado pelo serviço, sem qualquer modificação de código.

Avaliação sem reuso. Toda solicitação de validação (incluindo a re-validação automática que sucede uma correção) dispara uma nova chamada ao modelo, sem reuso de resultados anteriores. Essa decisão é deliberada e coerente com a postura *stateless* do sistema: como o modelo é estocástico, exibir sempre uma avaliação independente do artefato corrente é mais fiel à sua natureza do que estabilizar artificialmente a nota. A mesma variância é mensurada de forma sistemática no estudo de caso experimental (Capítulo 6), por meio de um orquestrador que executa cada validação de forma isolada, com $N = 5$ replicações independentes.

Estado e persistência. No lado do servidor, o sistema opera de modo *stateless*: nenhum dado de análise ou validação é persistido em banco de dados, e cada requisição é independente. No lado do cliente, o último resultado de análise é mantido no `sessionStorage` do navegador durante a sessão, de modo que um recarregamento acidental da página não descarte os artefatos já gerados (evitando o custo de uma nova análise). Trata-se de estado volátil, restrito ao navegador e eliminado ao fechar a aba; em nenhum momento há persistência *server-side* de código de terceiros, o que simplifica a infraestrutura e elimina questões de privacidade associadas ao armazenamento.

Implementação de referência em Python. Adicionalmente à aplicação web, o repositório do projeto contém uma implementação de referência em Python no diretório `scripts/`, organizada como interface de linha de comando. Esta implementação representa a etapa inicial de prototipagem que precedeu a aplicação web e mantém-se disponível como ferramenta auxiliar para experimentação rápida e reprodução do experimento descrito no capítulo de estudo de caso.

5.10 Considerações sobre o Capítulo

Este capítulo descreveu o sistema docScrum em três planos complementares: o conceitual, com a apresentação dos objetivos e dos requisitos que orientaram seu desenvolvimento; o arquitetural, com a explicitação das camadas, módulos e padrões adotados; e o de implementação, com a apresentação dos componentes principais, do fluxo de execução e das decisões técnicas relevantes. O estudo de caso experimental (Capítulo 6) fornece a evidência quantitativa que justifica as decisões aqui registradas, em particular a adoção do DeepSeek R1 como provedor de LLM e a inclusão do ciclo de validação e correção fundamentada como contribuição metodológica central da ferramenta.

6 Protocolo do Estudo de Caso

Este capítulo detalha o estudo de caso experimental realizado como núcleo prático desta pesquisa. A estrutura segue o protocolo formal de engenharia de software, abordando desde a fundamentação do problema até a análise crítica dos resultados quantitativos obtidos. A contribuição metodológica central é a demonstração empírica, com replicação ($N = 5$), de um *ciclo iterativo de geração, validação e refinamento* de artefatos ágeis assistido por Modelos de Linguagem de Grande Escala (LLMs), tomando o *Guia do Scrum 2020* como referência normativa para a etapa de validação.

6.1 Design do Estudo e Objetivos (GQM)

O design desta pesquisa foi elaborado utilizando a metodologia *Goal Question Metric* (GQM). O objetivo principal é avaliar empiricamente, sobre um projeto acadêmico real e *em múltiplas execuções independentes*, a capacidade do sistema *docScrum* de gerar artefatos ágeis e de aprimorá-los por meio de auto-crítica fundamentada em documento normativo. Para direcionar a análise, foram estabelecidas as seguintes questões de pesquisa:

- **QP1:** Qual a qualidade média (e a variabilidade entre execuções) do artefato *Backlog Ágil* gerado em *one-shot* pelo *docScrum* a partir de código Java, medida nas dimensões de *Conformidade Estrutural*, *Cobertura Conceitual* e *Aderência Semântica*?
- **QP2:** O artefato *Sprint Planning* apresenta perfil de qualidade similar ao do *Backlog* ou demanda esforço de correção distinto?
- **QP3:** O ciclo automatizado de *correção fundamentada em feedback*, aplicado sobre os artefatos avaliados, produz melhoria quantitativamente mensurável e estatisticamente consistente nas notas das três dimensões e na nota média geral?
- **QP4:** Quais elementos canônicos do Guia do Scrum são mais frequentemente citados pelo validador como ausentes nos artefatos gerados em *one-shot*, indicando lacunas sistemáticas da geração inicial?
- **QP5:** Qual é o custo computacional médio (medido em *tokens* consumidos e tempo de resposta) do ciclo completo de geração, validação, correção e revalidação?

6.2 Unidade de Análise

A seleção da unidade de análise priorizou um projeto acadêmico real, de pequeno-médio porte, com cobertura representativa dos padrões usuais em aplicações Spring Boot corporativas. O sistema selecionado é um projeto acadêmico desenvolvido como trabalho prático de uma disciplina da grade curricular do curso, consistindo em um *back-end* Spring Boot acompanhado de um *front-end* Angular que implementa um catálogo de veículos com autenticação *JSON Web Token* (JWT) e fluxo de compra. A escolha de um projeto acadêmico foi deliberada: por um lado, garante representatividade de boas práticas de arquitetura corporativa em Java (camadas *controller*, *service*, *repository*, segurança via JWT, conversores customizados); por outro, evita problemas éticos e jurídicos associados ao uso de código proprietário de terceiros como insumo da análise. A Tabela 13 sintetiza suas características quantitativas.

Tabela 13 – Características quantitativas do projeto-alvo.

Atributo	Valor
Linguagem analisada	Java (Spring Boot)
Arquivos .java analisados	38
Camadas arquiteturais	controller, service, repository, entity, dto, security, exception, converter, config
Tamanho consolidado do código	46.119 caracteres
Recursos do domínio	Cadastro e autenticação de usuários (JWT), gerenciamento de carros, fluxo de compras

Fonte: autoria própria.

A escolha desse projeto justifica-se por três fatores: (i) é um sistema acadêmico real, não didático, com camadas e dependências típicas de aplicações corporativas; (ii) sua dimensão é suficiente para representatividade (38 arquivos cobrindo nove camadas arquiteturais) sem extrapolar a janela de contexto do modelo selecionado; e (iii) sua estrutura inclui elementos comumente avaliados em revisões arquiteturais, como serviços com regra de negócio, repositórios JPA, conversores customizados e *filters* de segurança.

6.3 Procedimento de Coleta e Protocolo de Execução

A coleta de dados foi automatizada por meio de um *script* de orquestração que reproduz fielmente o pipeline do sistema *docScrum* em produção, eliminando intervenção manual e garantindo a reprodutibilidade do experimento. Diante da variância estocástica inerente aos LLMs, o protocolo foi executado em $N = 5$ execuções inde-

pendentes do pipeline completo, permitindo análise estatística dos resultados. Cada execução compreende as seguintes etapas:

Etapla 1: Coleta de arquivos. Percurso recursivo do projeto-alvo, com aplicação dos filtros utilizados pela camada de *frontend*: o conjunto `IGNORED_DIRS` (`target`, `.gradle`, `build`, `.idea`, `.settings`, `.classpath`, `.mvn`, `.git`, entre outros) é excluído, e apenas arquivos com extensão `.java` são considerados.

Etapla 2: Consolidação. Concatenação dos arquivos em uma única *string*, preservando o caminho relativo de cada arquivo como cabeçalho delimitador no formato `///
=== ARQUIVO: <caminho> ===`. Este recurso é essencial para que o modelo identifique a estrutura de pacotes e a hierarquia de diretórios do projeto acadêmico.

Etapla 3: Geração dos quatro artefatos. Submissão do código consolidado ao DeepSeek R1 com *prompt* estruturado em quatro marcadores delimitadores, um para cada artefato gerado (`DOCUMENTACAO`, `REQUISITOS`, `BACKLOG` e `SPRINTS`), empregados no formato `===NOME===`. A resposta retornada pelo modelo é em seguida *parseada* em quatro artefatos individuais, cada um gravado em arquivo *Markdown* separado.

Etapla 4: Validação contra o Guia do Scrum. Para os dois artefatos diretamente alinhados ao escopo do Guia (*Backlog* e *Sprints*), é executada uma chamada de validação. O serviço de validação injeta o Guia do Scrum 2020 integral (aproximadamente 27 mil caracteres) como *system message* (estratégia de *context stuffing*) e instrui o modelo a produzir uma avaliação em formato JSON estruturado, composta por três dimensões: *Conformidade Estrutural*, *Cobertura Conceitual* e *Aderência Semântica*. Cada dimensão recebe nota de 0 a 10, justificativa textual e citações literais do Guia que fundamentam o julgamento.

Etapla 5: Correção fundamentada. Sobre cada artefato avaliado, é executada uma chamada de correção que recebe como entrada o conteúdo original e o objeto de *feedback* retornado pela validação. O *prompt* de correção instrui o modelo a reescrever o artefato corrigindo apenas as lacunas apontadas no *feedback*, preservando o conteúdo já bem-avaliado e mantendo a estrutura *Markdown* original. Não é permitida a introdução de elementos novos sem fundamentação no *feedback*, restrição que mitiga o risco de *overengineering*.

Etapla 6: Re-validação automática. O conteúdo corrigido é resubmetido ao validador, permitindo o cálculo do *delta* quantitativo entre a nota inicial e a nota obtida após a correção. Esse passo materializa o ciclo completo do experimento.

A temperatura do modelo foi fixada em 0,2 nas etapas de análise e validação, e em

0,3 na etapa de correção (valor ligeiramente mais alto, justificado pela necessidade de flexibilidade na reescrita do artefato). Toda a execução é registrada em *checkpoints* JSON locais, contendo a resposta integral do modelo (*raw response*) em cada chamada, viabilizando auditoria posterior e replicação dos resultados.

6.4 Análise de Resultados

6.4.1 Notas Individuais por Execução

A Tabela 14 apresenta as notas médias gerais obtidas em cada uma das cinco execuções independentes, antes e após o ciclo de correção.

Tabela 14 – Notas médias gerais (escala 0–10) obtidas em cada execução, antes e após a correção fundamentada.

Execução	Backlog Ágil			Sprint Planning		
	Antes	Após	Δ	Antes	Após	Δ
1	3,3	8,7	+5,4	2,7	10,0	+7,3
2	2,0	9,3	+7,3	2,3	8,7	+6,3
3	3,3	8,3	+5,0	3,0	10,0	+7,0
4	4,7	9,7	+5,0	1,3	4,0	+2,7
5	4,3	9,7	+5,4	3,3	9,0	+5,7
Média	3,53	9,14	+5,61	2,53	8,34	+5,81
Desvio padrão	1,05	0,62	0,96	0,78	2,50	1,85

Fonte: autoria própria.

A análise das notas individuais revela três observações relevantes. Primeiro, a nota inicial dos artefatos é *sistematicamente baixa* em todas as execuções (*Backlog* $\mu = 3,53$; *Sprints* $\mu = 2,53$), confirmando que a geração *one-shot* produz artefatos com lacunas estruturais relevantes em relação ao Guia oficial. Segundo, a melhoria proporcionada pelo ciclo de correção é *consistente em todas as cinco execuções*, com *delta* médio de +5,61 pontos para o *Backlog* e +5,81 pontos para o *Sprints*. Terceiro, observa-se uma execução *outlier* (Execução 4) no caso do *Sprints*, em que a correção produziu nota final de 4,0/10, substancialmente inferior às demais execuções (8,7 a 10,0). Essa variabilidade será discutida na Seção 6.6.

6.4.2 Notas Médias por Dimensão

A Tabela 15 desagrega as notas médias por dimensão de avaliação, evidenciando quando o ciclo de correção produz maior impacto.

Tabela 15 – Notas médias por dimensão ($N = 5$), na forma $\mu \pm \sigma$.

Artefato	Dimensão	Antes	Após	Δ médio
Backlog Ágil	Conformidade Estrutural	$5,6 \pm 2,1$	$9,4 \pm 0,5$	+3,8
	Cobertura Conceitual	$2,4 \pm 0,9$	$8,8 \pm 0,8$	+6,4
	Aderência Semântica	$2,6 \pm 1,5$	$9,2 \pm 0,8$	+6,6
	Média Geral	$3,53 \pm 1,05$	$9,14 \pm 0,62$	+5,61
Sprint Planning	Conformidade Estrutural	$2,8 \pm 0,8$	$8,2 \pm 3,0$	+5,4
	Cobertura Conceitual	$2,2 \pm 0,8$	$8,8 \pm 1,3$	+6,6
	Aderência Semântica	$2,6 \pm 1,3$	$8,0 \pm 3,4$	+5,4
	Média Geral	$2,53 \pm 0,78$	$8,34 \pm 2,50$	+5,81

Fonte: autoria própria.

Em ambos os artefatos, a dimensão de *Cobertura Conceitual* apresenta o maior ganho médio (+6,4 no *Backlog*, +6,6 no *Sprints*), o que indica que o ciclo de correção é particularmente eficaz quando a lacuna identificada pelo validador é de natureza terminológica e estrutural, isto é, quando a citação literal do Guia fornece ao corretor um direcionamento explícito sobre o elemento canônico a ser incorporado (*Meta do Produto*, *Definição de Pronto*, *Sprint Backlog*, etc.). A dimensão de *Conformidade Estrutural* do *Sprints* apresentou o desvio padrão pós-correção mais elevado ($\sigma = 3,0$), consistente com a presença do *outlier* apontado na subseção anterior.

6.4.3 Mapeamento das Citações do Guia do Scrum

Uma das contribuições qualitativas deste estudo é a identificação de quais trechos do Guia do Scrum 2020 são mais frequentemente citados pelo validador como fundamentação para apontar lacunas nos artefatos gerados em *one-shot*. Estas citações, agregadas ao longo das cinco execuções, revelam padrões sistemáticos de omissão da geração inicial. A Tabela 16 apresenta as cinco citações mais frequentes utilizadas pelo validador na avaliação do *Backlog* antes da correção.

A Tabela 17 apresenta o mesmo mapeamento para a avaliação inicial do *Sprints*.

A análise das citações revela três lacunas sistemáticas da geração *one-shot*: (i) a *ordenação por valor* dos itens do *Product Backlog* e o conceito de itens *prontos para seleção* no *Sprint Planning*; (ii) o pilar empírico da *transparência* (citado em ambos os artefatos), indicando que a geração inicial tende a produzir descrições funcionais sem evidenciar como o trabalho será inspecionado; e (iii) a ausência da *Meta do Produto* e da *Meta da Sprint* como compromissos formais dos respectivos artefatos. Estes três

Tabela 16 – Trechos do Guia do Scrum mais citados pelo validador na avaliação do *Backlog* ($N = 5$).

Freq.	Trecho do Guia do Scrum 2020
5	<i>“Os itens do Product Backlog que podem ser realizados pelo Scrum Team em uma Sprint são considerados preparados para seleção no evento Sprint Planning.”</i>
4	<i>“O Product Backlog é uma lista ordenada e emergente do que é necessário para melhorar o produto.”</i>
4	<i>“A transparência permite a inspeção. A inspeção sem transparência é enganosa e gera desperdício.”</i>
2	<i>“A Meta do Produto está no Product Backlog. O restante do Product Backlog emerge para definir ‘o que’ cumprirá a Meta do Produto.”</i>
2	<i>“O Product Backlog refinement é o ato de quebrar e incluir definição adicional aos itens do Product Backlog para ter itens menores e mais precisos.”</i>

Fonte: autoria própria.

Tabela 17 – Trechos do Guia do Scrum mais citados pelo validador na avaliação do *Sprints* ($N = 5$).

Freq.	Trecho do Guia do Scrum 2020
3	<i>“A Meta da Sprint, os itens do Product Backlog selecionados para a Sprint, mais o plano para entregá-los são chamados juntos de Sprint Backlog.”</i>
3	<i>“A transparência permite a inspeção. A inspeção sem transparência é enganosa e gera desperdício.”</i>
2	<i>“Cada evento no Scrum é uma oportunidade formal para inspecionar e adaptar os artefatos do Scrum.”</i>
2	<i>“Para o Product Backlog, é a Meta do Produto.”</i>
2	<i>“A Sprint Planning aborda os seguintes tópicos: Tópico um: Por que esta Sprint é valiosa?”</i>

Fonte: autoria própria.

achados constituem a base empírica para a resposta à QP4 e oferecem direcionamento concreto para futuros refinamentos do *prompt* de geração, que poderia instruir explicitamente o modelo a incorporar tais elementos a partir do próprio *prompt* inicial.

6.4.4 Custo Computacional

A Tabela 18 apresenta o custo computacional agregado das cinco execuções e os valores médios por execução.

Tabela 18 – Custo computacional do experimento ($N = 5$, modelo DeepSeek R1).

Métrica	Total	Médio por execução
Tokens consumidos	389.339	77.868
Tempo acumulado	813,3 s (~13 min 30 s)	162,7 s (~2 min 42 s)

Fonte: autoria própria.

Em relação à QP5, cada execução do pipeline completo consome aproximadamente 78 mil *tokens* e leva pouco menos de três minutos. A predominância de *tokens* de entrada nas etapas de validação e correção é explicada pela injeção integral do Guia do Scrum (aproximadamente 27 mil caracteres) como *system message* em cada chamada. Esta escolha arquitetural, em detrimento de *Retrieval-Augmented Generation* (RAG), justifica-se pela dimensão moderada do documento normativo, que cabe folgadoamente na janela de contexto de 128 mil *tokens* do modelo, dispensando a complexidade adicional de infraestrutura de recuperação semântica.

6.5 Discussão Crítica

6.5.1 Resposta às Questões de Pesquisa

QP1: Qualidade do Backlog em geração *one-shot*. A nota média de $3,53 \pm 1,05$ obtida antes da correção indica que o *Backlog* gerado em *one-shot* apresenta lacunas conceituais sistemáticas. O coeficiente de variação relativamente alto ($\approx 30\%$) revela que a qualidade da geração *one-shot* é *instável* entre execuções, com notas individuais variando de 2,0 a 4,7.

QP2: Qualidade do *Sprint* em geração *one-shot*. A nota média de $2,53 \pm 0,78$ é *ainda inferior* à do *Backlog*, sugerindo que a inferência do planejamento iterativo a partir do código-fonte é uma tarefa estruturalmente mais difícil para o modelo. As principais lacunas apontadas pelo validador foram a ausência de *timebox* explícito por *Sprint*, a falta de descrição dos *eventos Scrum* (*Daily*, *Review*, *Retrospective*) e a definição inconsistente de *Pronto*.

QP3: Eficácia do ciclo de correção. Ambos os artefatos apresentaram *deltas* positivos em *todas as cinco execuções*, com magnitude média substancial (*Backlog*: $\Delta = +5,61 \pm 0,96$; *Sprints*: $\Delta = +5,81 \pm 1,85$). Para o *Backlog*, o desvio padrão pós-correção ($\sigma = 0,62$) é *menor* que o pré-correção ($\sigma = 1,05$), evidenciando que o ciclo de correção atua como *regularizador da qualidade*: independentemente da qualidade inicial sorteada pela geração estocástica, o conteúdo final converge para uma faixa estreita de alta qualidade ($\mu = 9,14$). Para o *Sprints*, embora o ganho médio seja similar, o desvio padrão pós-correção ($\sigma = 2,50$) é amplificado pela presença de uma execução *outlier* (Execução 4, nota final 4,0), sugerindo que a correção do *Sprints* é mais sensível à qualidade da versão inicial.

QP4: Lacunas sistemáticas identificadas. O mapeamento de citações apresentado nas Tabelas 16 e 17 identifica três elementos canônicos do Guia sistematicamente ausentes na geração *one-shot*: ordenação por valor, pilar da transparência e Metas (do Produto e da *Sprint*). Estes achados oferecem base empírica para refinamentos futuros do *prompt* de geração.

QP5: Custo computacional. O custo agregado (389 mil *tokens*, 13,5 minutos para cinco execuções) demonstra a viabilidade prática do pipeline em uso individual de pesquisa acadêmica.

6.5.2 Contribuição Metodológica

A contribuição distintiva deste estudo, em relação aos trabalhos correlatos que se limitam à geração *one-shot* de artefatos por LLM, é a demonstração de um *ciclo completo de geração, validação e refinamento* no qual o próprio modelo participa das três etapas, mas assumindo papéis distintos: *gerador*, na etapa de análise; *juiz crítico fundamentado em documento normativo*, na etapa de validação; e *revisor orientado por evidência citada*, na etapa de correção. A separação explícita destes papéis, materializada em três *prompts* distintos com contextos diferentes, permite a observação empírica do impacto da autocritica fundamentada e produz *evidência quantitativa replicada* ($N = 5$) da eficácia da intervenção.

6.6 Ameaças à Validade e Limitações

Identificaram-se seis ameaças à validade dos resultados obtidos, enumeradas a seguir em ordem de relevância.

1. **Variabilidade entre execuções.** Mesmo com baixa temperatura (0,2), os modelos de linguagem apresentam variabilidade entre execuções sobre o mesmo

conteúdo, fenômeno inerente à arquitetura estocástica desses modelos. O presente estudo *quantificou* essa variância nas cinco execuções: para o *Backlog*, as notas iniciais variaram entre 2,0 e 4,7 e as notas pós-correção entre 8,3 e 9,7; para o *Sprints*, as notas iniciais variaram entre 1,3 e 3,3 e as notas pós-correção entre 4,0 e 10,0. A interpretação correta dos resultados deve ser feita com base nas médias e desvios padrão reportados, não em valores pontuais. Coerentemente com essa natureza, tanto o produto web quanto o orquestrador experimental executam cada validação de forma independente, sem reuso de avaliações anteriores; a replicação em $N = 5$ execuções isoladas é precisamente a estratégia adotada para capturar e quantificar essa variância, em vez de neutralizá-la.

2. **Execução discrepante no Sprints.** A Execução 4 produziu uma correção do *Sprints* com nota final de apenas 4,0/10, em contraste com as demais execuções (8,7 a 10,0). A análise da resposta *raw* do modelo nesta execução, preservada em `run_04/relatorio.json`, indica que o conteúdo corrigido omitiu elementos canônicos que haviam sido apontados pelo validador (comportamento atípico e isolado). A presença desta execução amplifica o desvio padrão da nota pós-correção do *Sprints* ($\sigma = 2,50$), motivo pelo qual recomenda-se interpretar a mediana (9,0) como medida de tendência central mais robusta neste caso específico.
3. **Natureza automatizada do validador.** A avaliação produzida pelo serviço de validação não constitui prova absoluta de qualidade, mas sim um *indicador fundamentado* no Guia do Scrum oficial. A validade desta avaliação repousa sobre dois pilares: a estratégia de *context stuffing*, que torna o Guia integralmente disponível ao modelo no momento do julgamento, e a exigência de *citação literal* do Guia como fundamentação de cada nota atribuída. A supervisão humana permanece essencial para a aceitação final dos artefatos.
4. **Limites de generalização.** O estudo foi conduzido sobre um único projeto acadêmico de pequeno-médio porte. A generalização para projetos de diferentes tamanhos, linguagens ou domínios demanda replicação. Não obstante, a estrutura do pipeline e a estratégia de validação são, em princípio, transferíveis para outros documentos normativos (*ISO/IEC*, *IEEE 830*, *SAFe*) com modificações pontuais no *system message* do serviço de validação.
5. **Risco de *overengineering* em código complexo.** O fenômeno da alucinação é mitigado pela instrução explícita, no *prompt* de correção, de não introduzir elementos novos sem fundamentação no *feedback*. Ainda assim, em testes preli-

minares com classes individuais de alta complexidade ciclomática, observou-se tendência ocasional do modelo a propor padrões de projeto excessivos.

6. **Evolução dos modelos ao longo do tempo.** A natureza efêmera dos modelos de IA, cujas versões são atualizadas constantemente pelos provedores, pode alterar a reprodutibilidade exata deste estudo em datas futuras. Para mitigar essa ameaça, todas as respostas integrais do modelo nas cinco execuções foram preservadas, em formato JSON, no diretório `estudo_caso_v2/run_01/` a `run_05/` do repositório, permitindo auditoria posterior.

6.7 Considerações sobre o Capítulo

Este capítulo apresentou o protocolo e os resultados do estudo de caso experimental que constitui o núcleo prático da pesquisa. A replicação em cinco execuções independentes forneceu evidência quantitativa de que o ciclo iterativo de geração, validação e correção fundamentada produz melhoria substancial e consistente na qualidade dos artefatos (o *Backlog* passou de uma média de 3,53 para 9,14 pontos e o *Sprint Planning* de 2,53 para 8,34 pontos, ambos ancorados no Guia do Scrum 2020) e revelou três lacunas sistemáticas da geração *one-shot* (ordenação por valor, transparência e Metas) que abrem caminho para refinamentos futuros do *prompt*. As decisões arquiteturais e de engenharia que viabilizaram a execução do pipeline aqui avaliado foram apresentadas no capítulo dedicado ao sistema *docScrum*.

7 Considerações Finais

Este capítulo retoma a proposta deste trabalho, sintetiza os resultados obtidos, responde às questões de pesquisa formuladas no Capítulo 1, discute as limitações observadas durante a execução e aponta caminhos para pesquisas futuras.

Este trabalho propôs e avaliou o sistema *docScrum* como ferramenta de reengenharia ágil assistida por LLM, voltada à geração automatizada de artefatos do *framework* Scrum, em especial o *Product Backlog* e o *Sprint Planning*, a partir de código-fonte Java. Sua contribuição metodológica central foi o desenho de um ciclo iterativo no qual o próprio LLM desempenha, em chamadas separadas e com *prompts* especializados, três papéis distintos: *gerador*, na etapa de análise; *juiz* fundamentado no Guia do Scrum 2020, na etapa de validação; e *revisor* orientado por citação literal, na etapa de correção. A investigação foi conduzida em quatro fases sequenciais: seleção do modelo de LLM sob restrição orçamentária, que culminou na escolha do DeepSeek R1; desenvolvimento do sistema *docScrum*; execução de um estudo de caso experimental com cinco replicações independentes sobre um projeto Java acadêmico real; e análise quantitativa consolidada dos resultados.

A integração de Modelos de Linguagem de Grande Escala ao ciclo de reengenharia ágil mostrou-se viável e produtiva, reduzindo de forma substancial o esforço humano na produção inicial dos artefatos Scrum sem comprometer a aderência ao *framework* normativo. Demonstrou-se empiricamente que o próprio LLM pode atuar como instrumento de validação e correção das suas saídas, desde que os papéis sejam explicitamente separados e o juiz seja instruído a fundamentar cada avaliação em citações literais do documento normativo, condição que ancora o julgamento e mitiga o risco de alucinação.

Esse ciclo de auto-crítica fundamentada configura-se como alternativa concreta ao paradigma puramente *Human-in-the-loop*, transferindo parte da carga de revisão para o próprio sistema sem abrir mão da auditabilidade dos resultados. Tal modelo de operação posiciona-se como caminho promissor para a documentação automatizada de sistemas legados, área em que a dívida técnica documental persiste como obstáculo crítico à evolução do software.

7.1 Resposta às Questões de Pesquisa

7.1.1 Q1: Eficácia do Ciclo de Geração, Validação e Correção

Os resultados das cinco execuções demonstraram que o ciclo de auto-crítica fundamentada produz artefatos com aderência mensurável ao Guia do Scrum 2020 nas três dimensões avaliadas. No *Product Backlog*, a *Conformidade Estrutural* elevou-se de 5,6 para 9,4 (ganho de +3,8 pontos), a *Cobertura Conceitual* de 2,4 para 8,8 (+6,4 pontos) e a *Aderência Semântica* de 2,6 para 9,2 (+6,6 pontos); no *Sprint Planning*, os ganhos nas mesmas dimensões foram de +5,4, +6,6 e +5,4 pontos, respectivamente (escala de 0 a 10), comparando a saída *one-shot* inicial com a versão revisada após o ciclo. Esses ganhos confirmam que a separação explícita dos papéis de gerador, juiz e revisor, combinada com a exigência de fundamentação por citação literal, configura mecanismo eficaz de elevação da qualidade dos artefatos sem intervenção humana adicional.

7.1.2 Q2: Viabilidade Orçamentária

O modelo DeepSeek R1, selecionado no Capítulo 4, opera com custo abaixo do teto orçamentário estabelecido (US\$ 3,00 por milhão de *tokens*): seu custo combinado situa-se na ordem de US\$ 0,96 por milhão de *tokens*, menos de um terço do teto. Durante o estudo de caso, as cinco execuções consumiram 389.339 *tokens* no total (aproximadamente 78 mil por execução, em 13,5 minutos acumulados), o que mantém o custo do experimento completo na casa de centavos de dólar, confirmando a viabilidade financeira da abordagem mesmo sob o regime de múltiplas chamadas que o ciclo de auto-crítica exige.

7.1.3 Q3: Impacto Quantitativo do Ciclo de Correção

A análise comparativa entre a geração *one-shot* e a versão pós-correção evidenciou variação média positiva de +5,61 pontos para o *Product Backlog* e +5,81 pontos para o *Sprint Planning* nas notas globais, com desvio-padrão dos ganhos entre as cinco execuções independentes de 0,96 e 1,85 pontos, respectivamente. A baixa variabilidade observada sugere que o ciclo de autocrítica contribui não apenas para a elevação da qualidade média, mas também para a estabilidade do processo entre execuções, mitigando o impacto da natureza estocástica do modelo.

7.2 Limitações

A condução deste trabalho permitiu identificar limitações relevantes que devem ser consideradas na interpretação dos resultados:

- **Amostra de código:** a avaliação experimental foi realizada sobre um único projeto Java acadêmico. Embora o projeto contenha elementos representativos de sistemas reais (entidades de domínio, regras de negócio, herança), os resultados podem variar para bases de código de natureza distinta, como aplicações corporativas de grande porte ou códigos legados em paradigma procedural.
- **Documento normativo único:** o ciclo de validação foi instanciado exclusivamente sobre o Guia do Scrum 2020. A generalização para outros documentos normativos (normas ISO/IEC, padrões corporativos) requer adaptação dos *prompts* e revalidação experimental.
- **Dependência de provedor:** a escolha do DeepSeek R1 como modelo único introduz dependência de um provedor específico. Variações de desempenho decorrentes de atualizações do modelo ou eventual descontinuidade do serviço poderiam comprometer a reprodutibilidade dos resultados.
- **Ausência de painel humano sistemático:** a avaliação dos artefatos foi conduzida por meio do próprio LLM atuando como juiz fundamentado, complementada por inspeção humana pontual. Uma validação sistemática por painel de especialistas em Scrum representaria contraprova de maior robustez metodológica.
- **Estratégia de injeção de contexto:** o uso do *context stuffing* pressupõe que o documento normativo cabe na janela de contexto do modelo. Para documentos significativamente maiores, a abordagem requereria migração para estratégias de *Retrieval-Augmented Generation* (RAG), o que adicionaria complexidade arquitetural.

7.3 Trabalhos Futuros

A continuidade desta pesquisa pode percorrer múltiplas direções, das quais destacam-se:

- **Expansão do escopo de artefatos:** estender o sistema para a geração automatizada de outros artefatos do ecossistema ágil, como *Definition of Done* customizada por contexto, *user stories* em formato BDD (*Behaviour-Driven Development*) e diagramas arquiteturais derivados do código.

- **Avaliação cruzada de modelos:** replicar o estudo de caso com modelos alternativos (Grok 3 mini Reason, GPT-5 mini, modelos de pesos abertos via Ollama) para mensurar o quanto a contribuição metodológica do ciclo de autocrítica é dependente do LLM subjacente.
- **Validação humana em larga escala:** conduzir avaliação por painel de especialistas em Scrum sobre os artefatos gerados, calculando concordância entre avaliadores (*inter-rater agreement*) e comparando com as notas do juiz automatizado.
- **Generalização para outras linguagens-alvo:** expandir o sistema para análise de código em outras linguagens (Python, C#, Kotlin), avaliando o quanto a qualidade dos artefatos gerados é afetada por diferenças sintáticas e idiomáticas.
- **Migração para RAG:** explorar a substituição do *context stuffing* pelo padrão RAG para suportar documentos normativos extensos (normas ISO/IEC, padrões corporativos), mensurando empiricamente os impactos sobre auditabilidade, custo e fidelidade às citações literais exigidas pelo juiz.
- **Integração com pipelines de CI/CD:** investigar a inserção do *docScrum* em fluxos de integração contínua, executando a geração e validação dos artefatos a cada *merge* significativo, mantendo a documentação ágil sincronizada com a evolução do código-fonte.

Referências

AGHAJANI, E. et al. Software documentation: The practitioners' perspective. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE)*. ACM, 2020. p. 590–601. Disponível em: <<https://dl.acm.org/doi/10.1145/3377811.3380405>>. 20, 26, 28

NAGY, C. et al. Software documentation issues unveiled. In: *Proceedings of the 2019 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019. p. 356–366. Disponível em: <<https://ieeexplore.ieee.org/document/8668010>>. 20, 21

NAGAI, R. A.; SBRAGIA, R. As origens da metodologia ágil: de onde saímos e onde estamos? uma revisão sistemática da literatura. *Revista de Gestão e Projetos (GeP)*, v. 14, n. 1, p. 11–41, 2023. Disponível em: <<https://periodicos.uninove.br/gep/article/view/23723>>. 20

CANEDO, E. D. et al. On the challenges to documenting requirements in agile software development: A practitioners' perspective. In: *Anais do XXVII Congresso Ibero-Americano em Engenharia de Software (CibSE 2024)*. Porto Alegre: SBC, 2024. p. 286–300. Disponível em: <<https://sol.sbc.org.br/index.php/cibse/article/view/28454>>. 20, 21, 28, 30

MOHOTTIGE, T. I. et al. Reengineering software systems into microservices: State-of-the-art and future directions. *Information and Software Technology*, Elsevier, v. 183, p. 107732, 2025. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S0950584925000710>>. 20, 21, 26

FAN, A. et al. Large language models for software engineering: Survey and open problems. In: *Proceedings of the 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 2023. p. 31–53. Disponível em: <<https://ieeexplore.ieee.org/document/10173938>>. 20, 26, 31, 41

NAIMI, A.; KACEM, H. H.; TABBANE, S. Employing llms for precise use case description. *Procedia Computer Science*, Elsevier, v. 235, p. 123–132, 2024. Disponível em: <<https://www.sciencedirect.com/science/article/pii/S1877050924026176>>. 20

GUPTA, A. et al. Comparative analysis of LLMs for code documentation generation. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 2024. Disponível em: <<https://dl.acm.org/doi/10.1145/3664646.3664765>>. 20

LANO, K.; ALFRAIHI, H. S. Using LLMs and model-driven engineering for software reverse engineering. *Frontiers in Computer Science*, Frontiers, v. 7, 2025. Disponível em: <<https://www.frontiersin.org/articles/10.3389/fcomp.2025.1516410>>. 20, 26

- PIETRINI, R. et al. Leveraging LLMs for automated translation of legacy code: A case study on pl/sql to java transformation. *arXiv preprint arXiv:2508.19663*, 2025. Disponível em: <<https://arxiv.org/abs/2508.19663>>. 26
- HOU, X. et al. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 2024. Disponível em: <<https://dl.acm.org/doi/10.1145/3695988>>. 26, 31, 33, 35, 41
- MACEDO, M. et al. Leveraging LLMs for legacy code modernization: Challenges and opportunities for LLM-generated documentation. *arXiv preprint arXiv:2411.14971*, 2024. Disponível em: <<https://arxiv.org/abs/2411.14971>>. 26, 31, 33, 35, 37
- SASSA, A. C. et al. Scrum: A systematic literature review. *International Journal of Advanced Computer Science and Applications (IJACSA)*, v. 14, n. 4, p. 173–181, 2023. Disponível em: <<https://thesai.org/Publications/ViewPaper?Volume=14&Issue=4&Code=IJACSA&SerialNo=20>>. 27, 28, 29
- BECK, K. et al. *Manifesto for Agile Software Development*. 2001. <<https://agilemanifesto.org>>. Acesso em: 30 abr. 2026. 27, 28
- SCHWABER, K.; SUTHERLAND, J. *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*. 2020. <<https://scrumguides.org/scrum-guide.html>>. Acesso em: 30 abr. 2026. 27, 28, 29, 30, 31
- SHRIVASTAVA, A. et al. A systematic review on extreme programming. *Journal of Physics: Conference Series*, v. 1969, p. 012046, 2021. Disponível em: <<https://iopscience.iop.org/article/10.1088/1742-6596/1969/1/012046>>. 27
- DAMIJ, N.; DAMIJ, T. An approach to optimizing kanban board workflow and shortening the project management plan. *IEEE Transactions on Engineering Management*, v. 71, p. 13266–13273, 2024. Disponível em: <<https://ieeexplore.ieee.org/document/9509530>>. 27
- ALDAEEJ, A.; NGUYEN-DUC, A.; GUPTA, V. A lean approach of managing technical debt in agile software projects – a proposal and empirical evaluation. In: *Agile Processes in Software Engineering and Extreme Programming*. [S.l.]: Springer, 2023. 27, 28, 29
- HRON, M.; OBWEGESER, N. Why and how is scrum being adapted in practice: A systematic review. *Journal of Systems and Software*, Elsevier, v. 183, p. 111110, 2022. Disponível em: <<https://www.sciencedirect.com/science/article/abs/pii/S0164121221002077>>. 28, 29
- VASWANI, A. et al. Attention is all you need. In: *Advances in Neural Information Processing Systems (NeurIPS)*. [s.n.], 2017. v. 30, p. 5998–6008. Disponível em: <<https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html>>. 31, 32, 33
- LIU, N. F. et al. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, v. 12, p. 157–173, 2024. Disponível em: <<https://aclanthology.org/2024.tacl-1.9/>>. 33, 39

- FAN, W. et al. A survey on rag meeting llms: Towards retrieval-augmented large language models. *arXiv preprint arXiv:2405.06211*, 2024. Disponível em: <<https://arxiv.org/abs/2405.06211>>. 34, 38
- HOLTZMAN, A. et al. The curious case of neural text degeneration. In: *International Conference on Learning Representations (ICLR)*. [s.n.], 2020. Disponível em: <<https://openreview.net/forum?id=rygGQyrFvH>>. 34, 35
- Ollama. *Ollama: Get up and running with large language models locally*. 2025. <<https://ollama.com>>. Acesso em: 16 jun. 2026. 36
- DETTMERS, T. et al. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In: *Advances in Neural Information Processing Systems (NeurIPS)*. [s.n.], 2022. v. 35. Disponível em: <<https://arxiv.org/abs/2208.07339>>. 37
- TONMOY, S. M. T. I. et al. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313*, 2024. Disponível em: <<https://arxiv.org/abs/2401.01313>>. 37
- LIU, F. et al. Exploring and evaluating hallucinations in LLM-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024. Disponível em: <<https://arxiv.org/abs/2404.00971>>. 37
- GOSLING, J. et al. *The Java Language Specification, Java SE 21 Edition*. Redwood Shores, CA: Oracle America, Inc., 2023. Disponível em: <<https://docs.oracle.com/javase/specs/jls/se21/html/index.html>>. 39
- TIOBE Software. *TIOBE Index for Programming Language Popularity*. 2025. <<https://www.tiobe.com/tiobe-index/>>. Acesso em: 09 jun. 2026. 39
- NestJS. *NestJS – A progressive Node.js framework: Official documentation*. 2025. <<https://docs.nestjs.com/>>. Acesso em: 09 jun. 2026. 40
- CHANIOTIS, I. K.; KYRIAKOU, K.-I. D.; TSELIKAS, N. D. Is Node.js a viable option for building modern web applications? A performance evaluation study. *Computing*, Springer, v. 97, n. 10, p. 1023–1044, 2015. Disponível em: <<https://link.springer.com/article/10.1007/s00607-014-0394-9>>. 40
- React. *React: The library for web and native user interfaces — Official documentation*. 2025. <<https://react.dev>>. Acesso em: 16 jun. 2026. 40
- AGGARWAL, S. Modern web-development using ReactJS. *International Journal of Recent Research Aspects*, v. 5, n. 1, p. 133–137, 2018. ISSN 2349-7688. Disponível em: <<https://www.ijrra.net/Vol5issue1/IJRRRA-05-01-27.pdf>>. 40
- CHAZETTE, L.; SCHNEIDER, K. Behaviour driven development: A systematic mapping study. *Journal of Systems and Software*, Elsevier, v. 203, p. 111749, 2023. Disponível em: <<https://www.sciencedirect.com/science/article/abs/pii/S0164121223001425>>. 41
- RAHARJANA, I. K.; SIAHAAN, D.; FATICHAH, C. User stories and natural language processing: A systematic literature review. *Journal of Software: Evolution and Process*, Wiley, v. 35, n. 12, p. e2554, 2023. Disponível em: <<https://onlinelibrary.wiley.com/doi/10.1002/smr.2554>>. 41

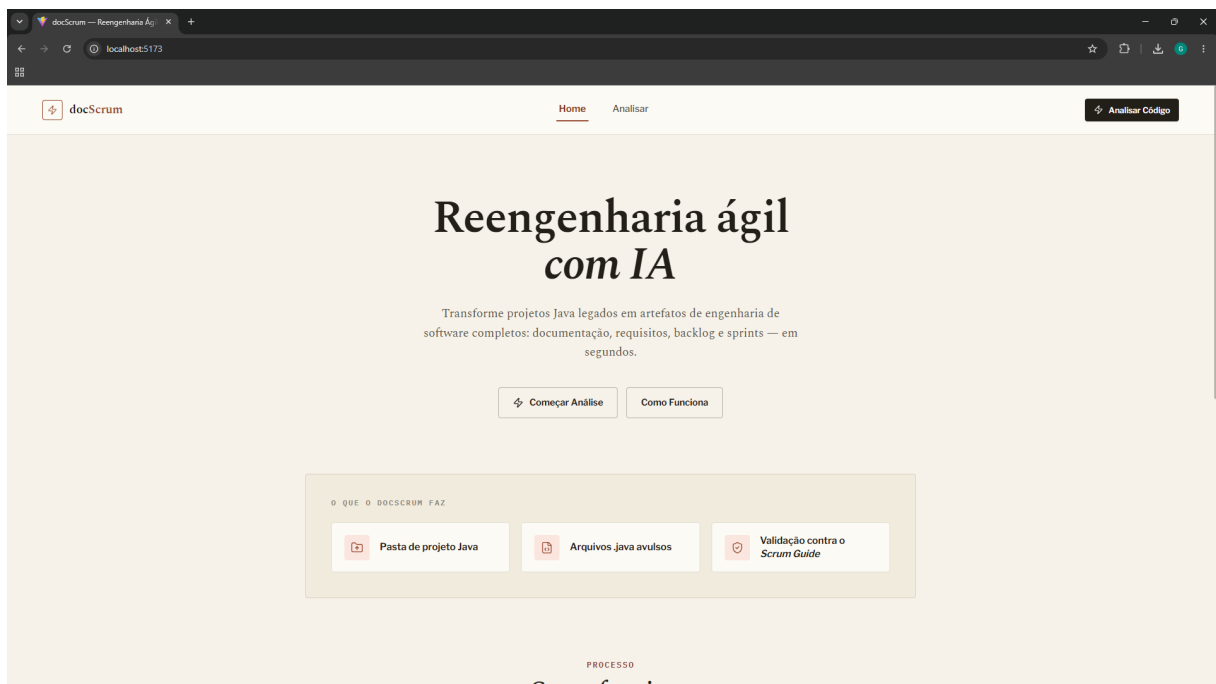
- KITCHENHAM, B. *Procedures for Performing Systematic Reviews*. Staffordshire, UK, 2004. Disponível em: <https://www.researchgate.net/publication/228756057_Procedures_for_Performing_Systematic_Reviews>. 44
- MADRUGA, J. V. M. M. Trabalho de Conclusão de Curso, *Assistência na Geração de Documentação com Base no Processamento de Linguagem Natural e Inteligência Artificial*. Natal, RN, Brasil: [s.n.], 2025. 46
- ARAUJO, A. B. d. O. *Inteligência Artificial na Documentação de Sistemas Legados: Uma Abordagem Comparativa de Ferramentas*. Monografia (Bacharelado em Ciência da Computação) — Universidade Estadual do Piauí, Teresina, Brasil, 2025. Orientador: Prof. Dr. Thiago Carvalho de Sousa. 46
- ROTILLI, F. R.; SILVA, D. R. da. Automatização inteligente de documentação de software com IImS. *Revista GESTO: Revista de Gestão Estratégica de Organizações*, v. 14, n. 1, p. 65–76, 2026. 47
- FREITAS, J. Q. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação), *Avaliação da eficácia de modelos de linguagem de grande escala na refatoração de código Java*. Fortaleza: [s.n.], 2025. 64 p. 48
- HATAHET, A.; KNIEKE, C.; RAUSCH, A. *Generating Software Architecture Description from Source Code using Reverse Engineering and Large Language Model*. 2025. ArXiv preprint arXiv:2511.05165. Acesso em: 07 jun. 2026. Disponível em: <<https://arxiv.org/abs/2511.05165>>. 48
- ROCCO, J. D. et al. On the use of large language models in model-driven engineering. *Software and Systems Modeling*, Springer, 2025. Disponível em: <<https://arxiv.org/abs/2410.17370>>. 49
- Benched.ai. *LLM Performance Benchmarks and Cost Analysis*. 2025. <<https://benched.ai>>. Acesso em: 07 jun. 2026. 53, 55, 58
- Artificial Analysis. *AI Models Benchmarks and Performance Comparison*. 2025. <<https://artificialanalysis.ai>>. Acesso em: 07 jun. 2026. 55, 56, 60

APÊNDICE A – Demonstração da Aplicação

Este apêndice apresenta o sistema *docScrum* em funcionamento, percorrendo as principais telas e fluxos disponíveis ao usuário. Para cada tela, são descritos o propósito, os elementos visuais relevantes e o papel daquela etapa no fluxo geral de reengenharia ágil assistida por LLM, desde o carregamento do código-fonte Java até a obtenção dos artefatos validados contra o Guia do Scrum 2020.

A.1 Tela Inicial

Figura 4 – Tela inicial do sistema *docScrum*.



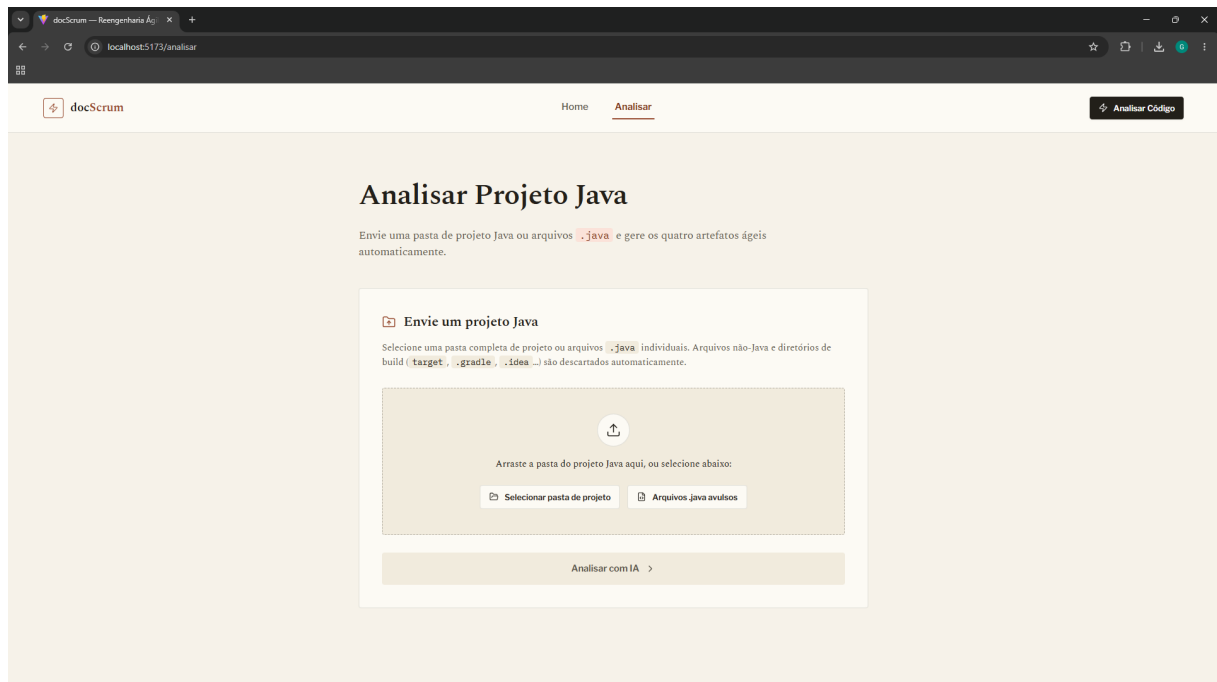
Fonte: autoria própria.

A tela inicial apresenta a proposta do sistema (“Reengenharia ágil com IA”) e estabelece o ponto de partida da interação com o usuário. A região central oferece os botões *Começar Análise* e *Como Funciona*, enquanto a seção inferior (“O que o docScrum faz”) sintetiza, em três cartões, as capacidades do sistema: aceitar pastas completas de projeto Java, processar arquivos `.java` avulsos e validar os artefatos gerados contra o Guia do Scrum 2020. O menu de navegação no topo dá acesso direto à área de análise.

A.2 Tela de Análise e Upload do Código-Fonte

A tela de análise é o ponto de entrada do fluxo de geração. Inicialmente, o usuário encontra a área de *upload* vazia, conforme apresentado na Figura 5.

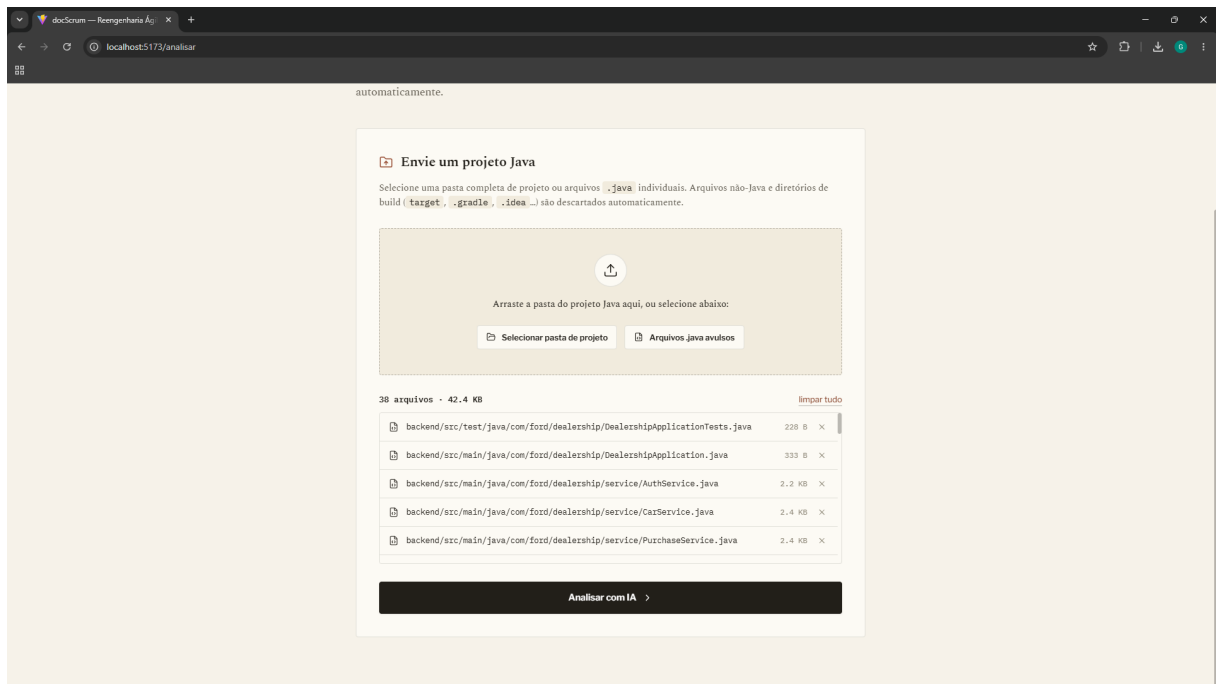
Figura 5 – Tela de análise do projeto Java em estado inicial.



Fonte: autoria própria.

O sistema oferece duas modalidades de entrada: o envio de uma *pasta completa de projeto* (varrendo recursivamente os arquivos .java e descartando automaticamente diretórios de *build* como target, .gradle e .idea) ou o envio de *arquivos .java avulsos*. A Figura 6 apresenta a tela após a seleção de uma pasta de projeto, exibindo a lista de arquivos identificados, o total agregado em tamanho e o botão de envio para análise.

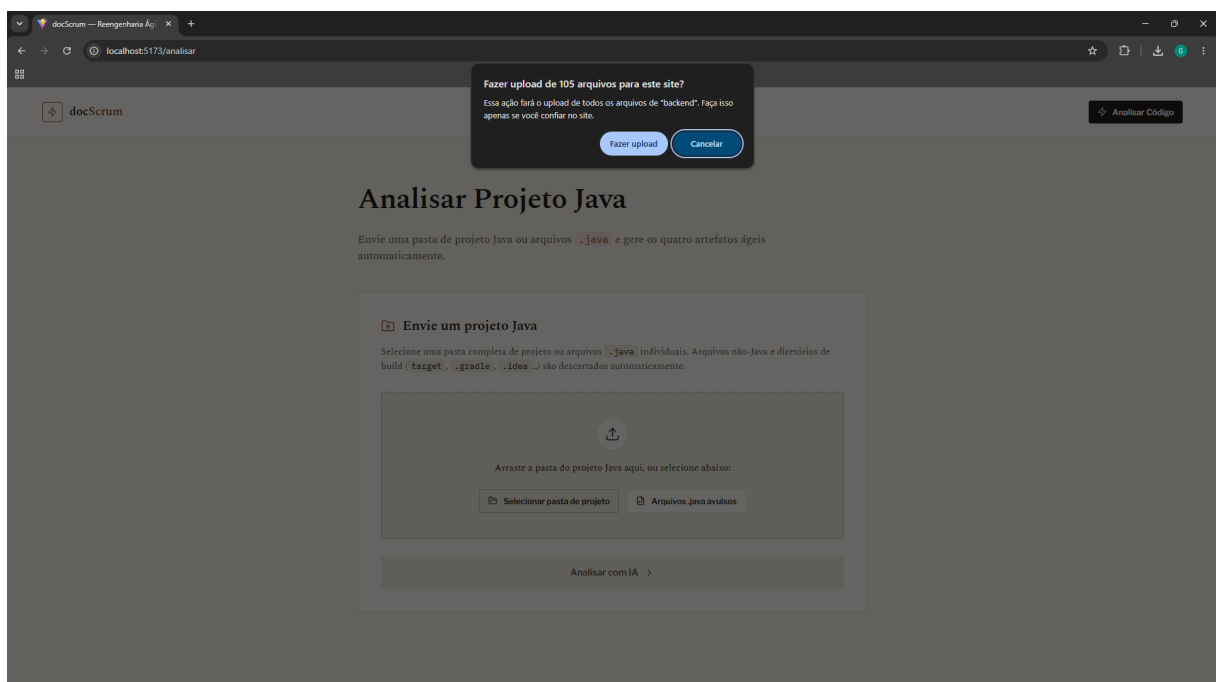
Figura 6 – Tela de análise com pasta de projeto Java selecionada.



Fonte: autoria própria.

Por questões de segurança, o navegador exibe um diálogo de confirmação antes de submeter múltiplos arquivos ao servidor, conforme ilustrado na Figura 7. Esse comportamento é nativo do navegador e funciona como salvaguarda contra envios não intencionais.

Figura 7 – Diálogo de confirmação do navegador para envio em lote.

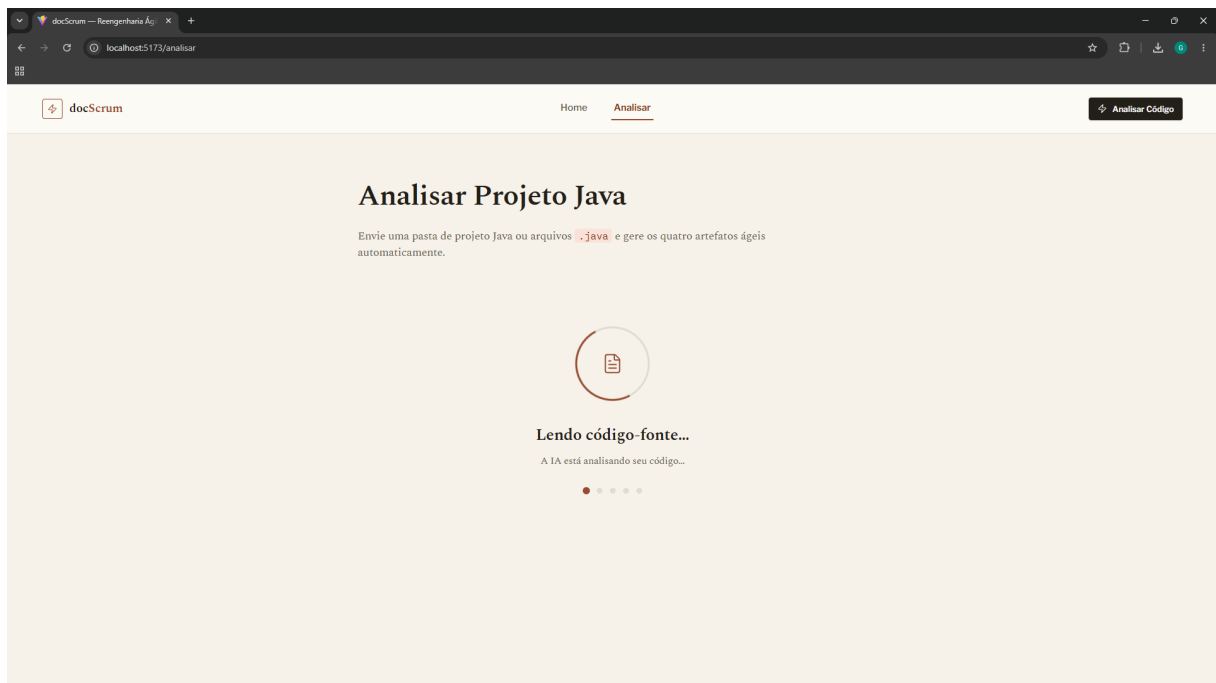


Fonte: autoria própria.

A.3 Processamento Inicial pela IA

Após a confirmação do envio, o sistema entra na fase de *análise inicial*, na qual o LLM (DeepSeek R1) lê o código-fonte e o interpreta semanticamente para produzir os artefatos ágeis. A Figura 8 apresenta o *feedback* visual disponibilizado ao usuário durante essa etapa.

Figura 8 – Tela de processamento inicial: leitura e interpretação do código-fonte pela IA.



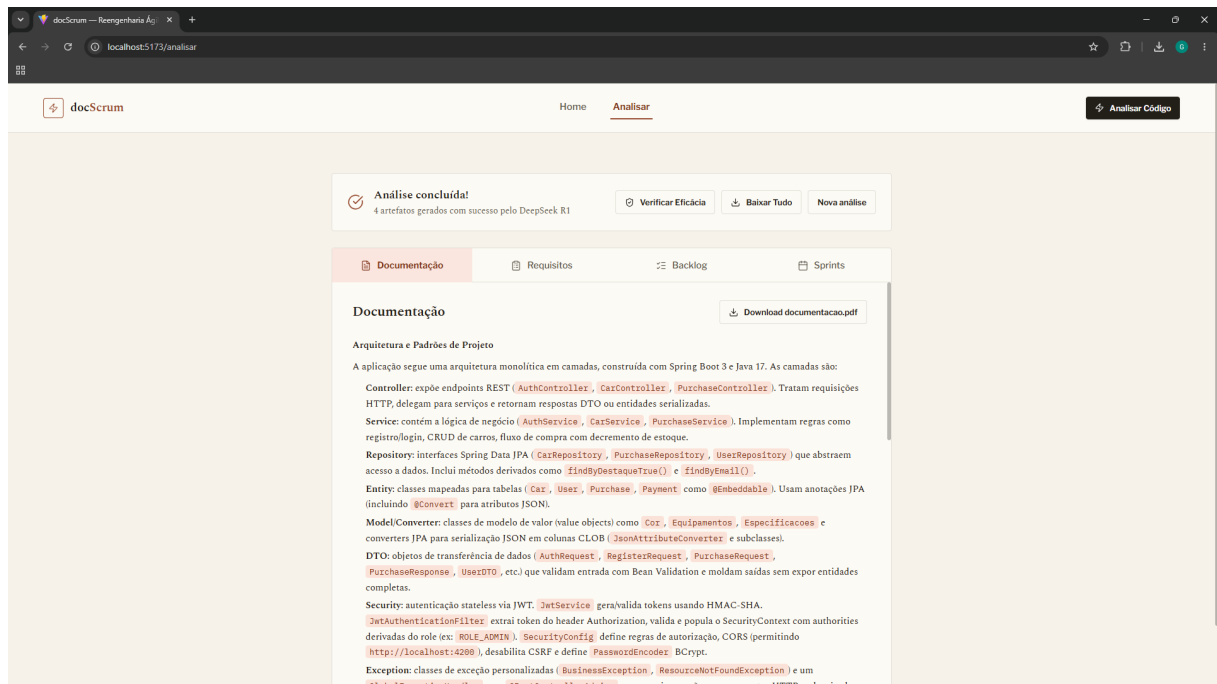
Fonte: autoria própria.

A interface comunica de forma transparente as etapas do *pipeline* (representadas pelos cinco pontos sequenciais), evitando que o usuário fique sem retorno durante o tempo necessário para que o modelo conclua o raciocínio.

A.4 Documentação Técnica Gerada

Concluída a análise, o sistema apresenta os quatro artefatos gerados em abas distintas. A primeira aba exibe a documentação técnica do projeto, conforme ilustrado na Figura 9.

Figura 9 – Documentação técnica gerada a partir da análise do código Java.



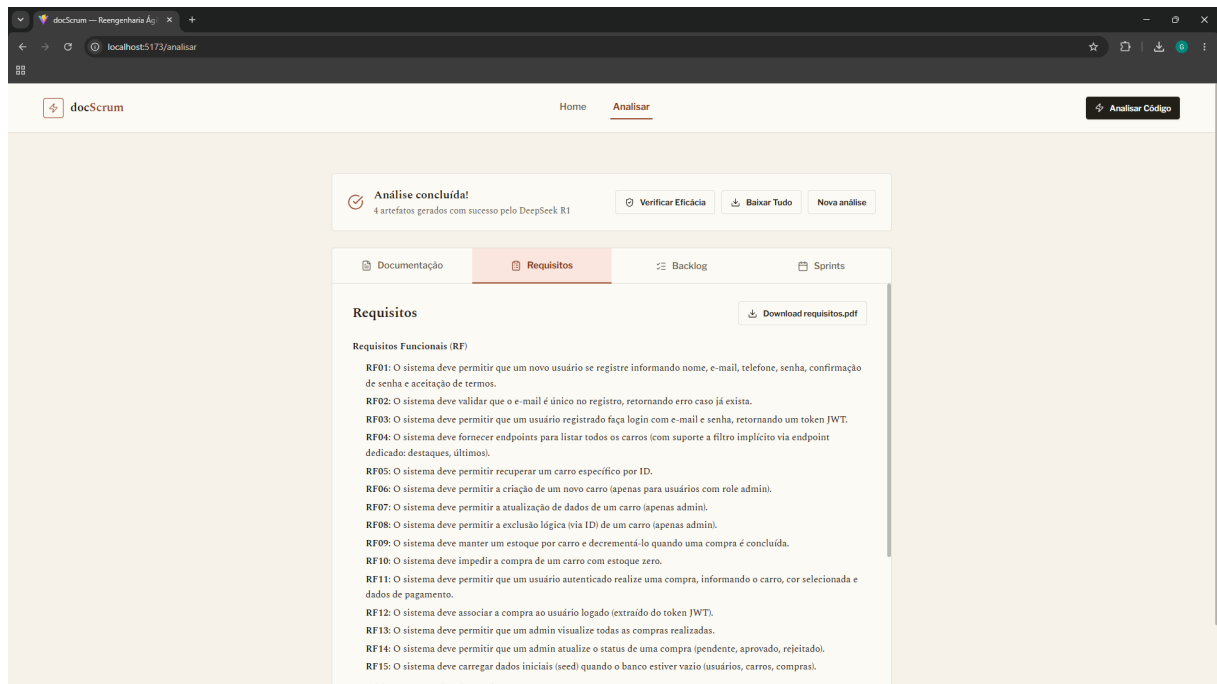
Fonte: autoria própria.

A documentação é organizada por camadas arquiteturais (Apresentação, Serviço, Persistência, entre outras), identificando classes, padrões de projeto aplicados (DTO, Service, Repository, @RequiredArgsConstructor, @Transactional) e responsabilidades de cada componente. Cada aba oferece um botão de *download* que exporta o respectivo artefato em formato PDF apresentável (Download documentacao.pdf, requisitos.pdf, e assim por diante), adequado para arquivamento e compartilhamento com *stakeholders*.

A.5 Requisitos Funcionais e Não Funcionais

A segunda aba apresenta os requisitos extraídos automaticamente do código-fonte, separados em Requisitos Funcionais (RF) e Requisitos Não Funcionais (RNF), conforme a Figura 10.

Figura 10 – Requisitos funcionais e não funcionais extraídos do código-fonte.

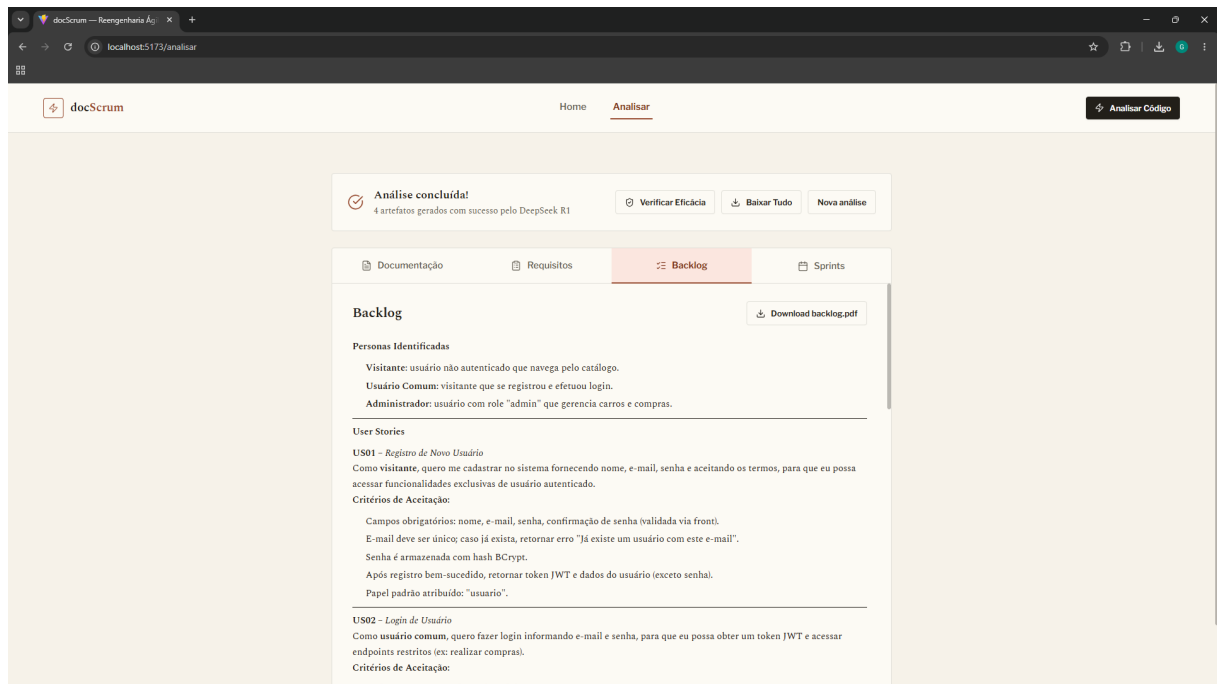


Fonte: autoria própria.

Cada requisito recebe código único (RF01, RF02, ..., RNF01, RNF02, ...) e descrição em linguagem natural. A organização espelha o padrão tipicamente adotado em documentos de Engenharia de Requisitos, com clara separação entre o que o sistema *deve fazer* (funcionais) e *como deve se comportar* (não funcionais: segurança, desempenho, integração).

A.6 Product Backlog Gerado

A terceira aba materializa o *Product Backlog* no formato canônico esperado pelo Scrum, conforme apresentado na Figura 11.

Figura 11 – *Product Backlog* gerado a partir da análise do código Java.

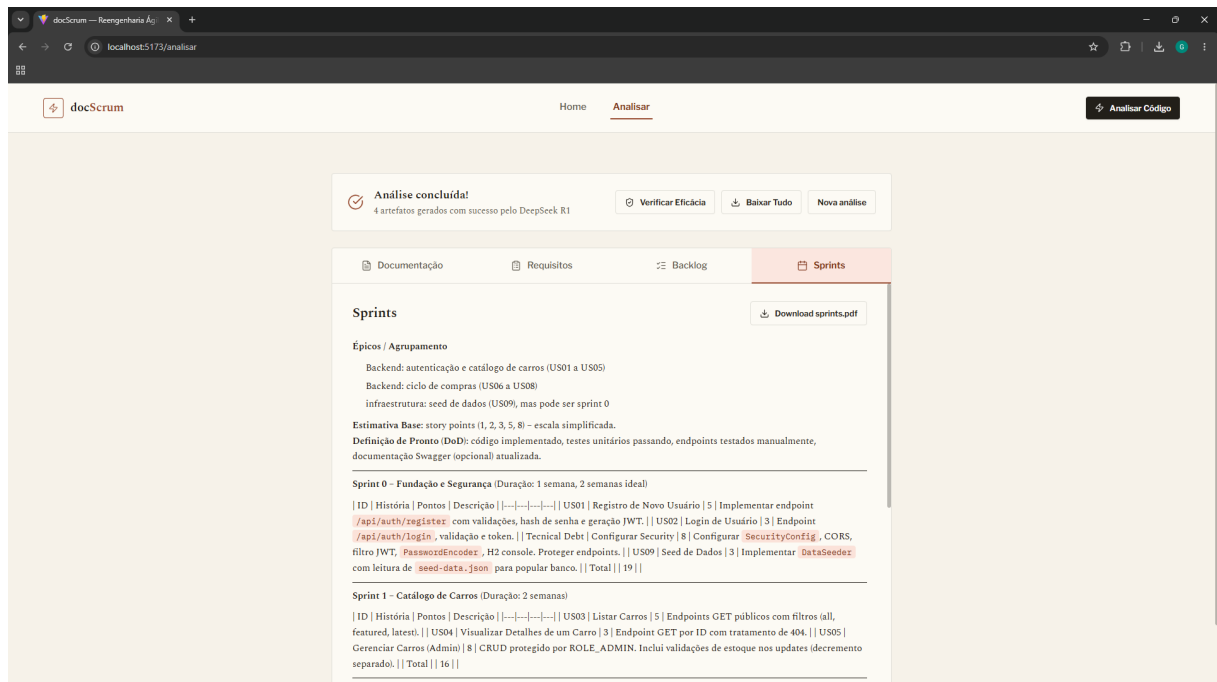
Fonte: autoria própria.

Cada item do *backlog* é estruturado como *user story* no padrão “Como [persona], quero [ação], para [objetivo]” (US01, US02, US03, ...), acompanhada dos respectivos *Critérios de Aceitação* (CA), também numerados sequencialmente (CA01.1, CA01.2, ...). Esse formato permite a importação direta em ferramentas como Jira, Azure DevOps ou Trello sem reescrita manual.

A.7 Sprint Planning

A quarta aba apresenta a sugestão de *Sprint Planning*, distribuindo os itens do *Product Backlog* ao longo de *Sprints* sequenciais, conforme a Figura 12.

Figura 12 – Sprint Planning gerado pelo sistema.



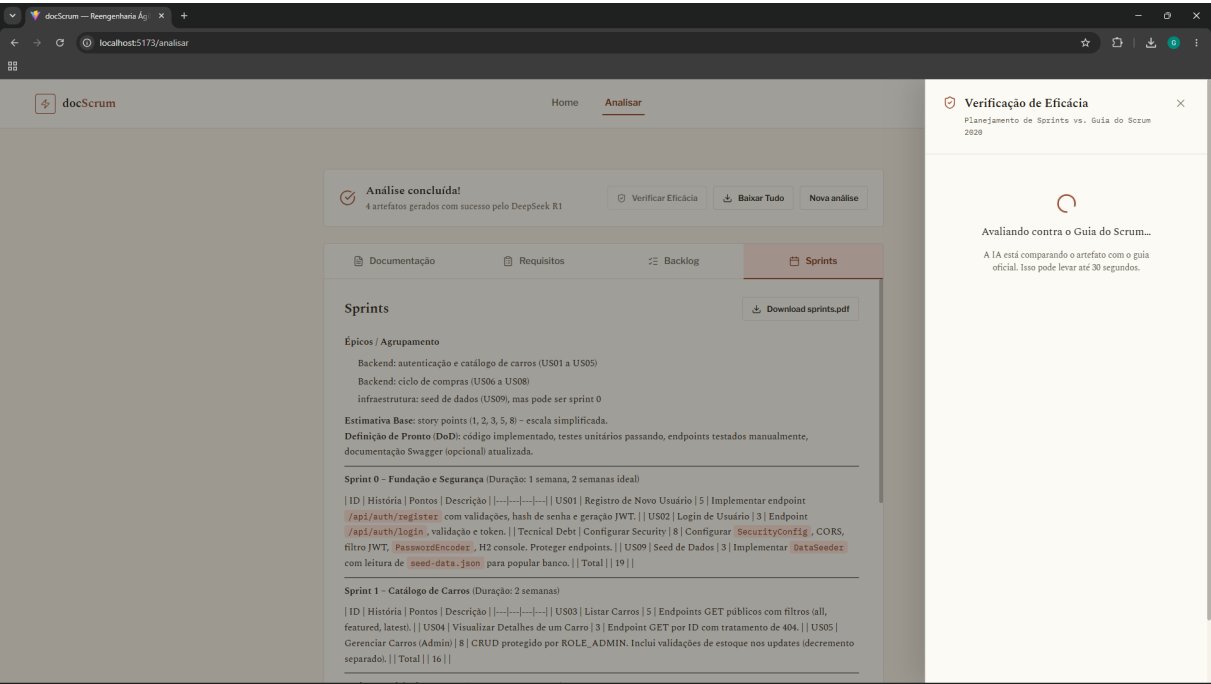
Fonte: autoria própria.

Cada *Sprint* contém o objetivo (*Sprint Goal*), os itens selecionados do *Product Backlog* com estimativas em *story points* (escala Fibonacci), o plano de ação detalhado (Backend, Endpoints, Testes) e a *Definition of Done* associada. A organização respeita o padrão prescrito pelo Guia do Scrum 2020 para o artefato *Sprint Backlog* e seu compromisso correspondente.

A.8 Verificação de Eficácia: Avaliação Inicial

Concluída a geração inicial dos quatro artefatos, o usuário pode acionar o ciclo de validação fundamentada por meio do botão *Verificar Eficácia*. O sistema então submete os artefatos ao papel de *juiz*, que os avalia contra o Guia do Scrum 2020 carregado integralmente em seu contexto via *context stuffing*. A Figura 13 apresenta o *feedback* visual dessa etapa.

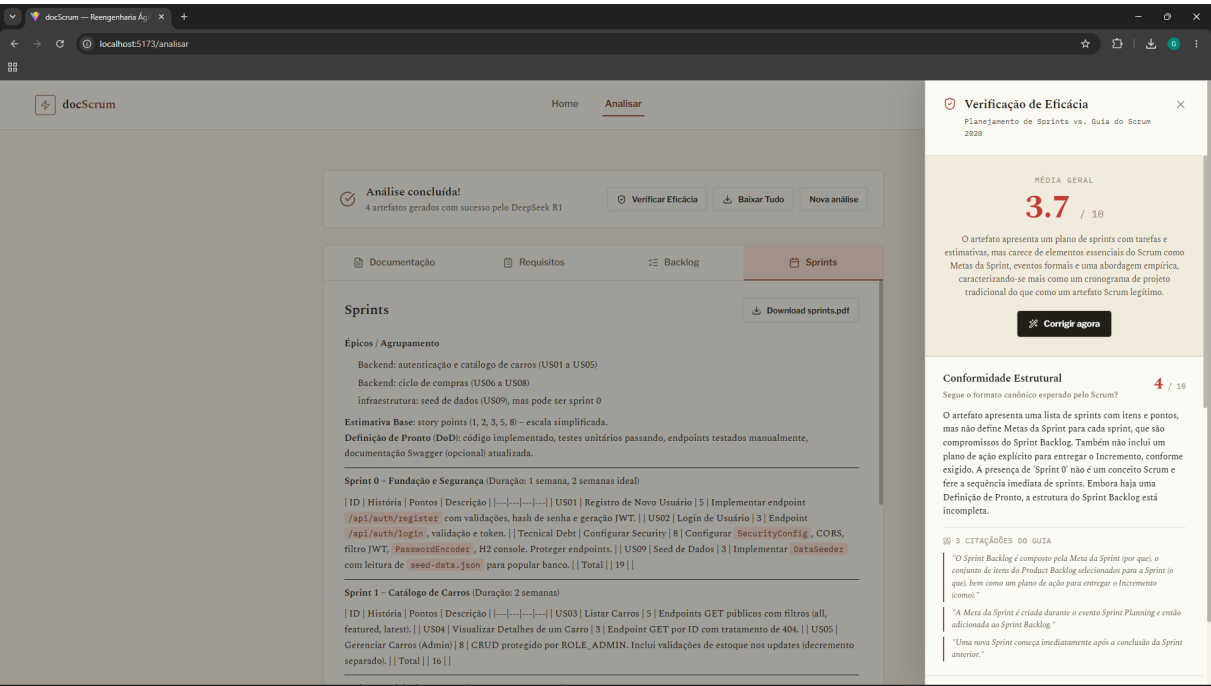
Figura 13 – Processamento da etapa de avaliação pelo juiz fundamentado no Guia do Scrum.



Fonte: autoria própria.

Ao final da avaliação, o sistema apresenta as notas atribuídas pelo juiz em três dimensões (Conformidade Estrutural, Cobertura Conceitual e Aderência Semântica), juntamente com as citações literais do Guia do Scrum que fundamentam cada crítica. A Figura 14 ilustra a tela de resultado para um *Sprint Backlog* avaliado em sua versão inicial (geração *one-shot*).

Figura 14 – Avaliação inicial pelo juiz: nota 3.7/10 antes da correção fundamentada.



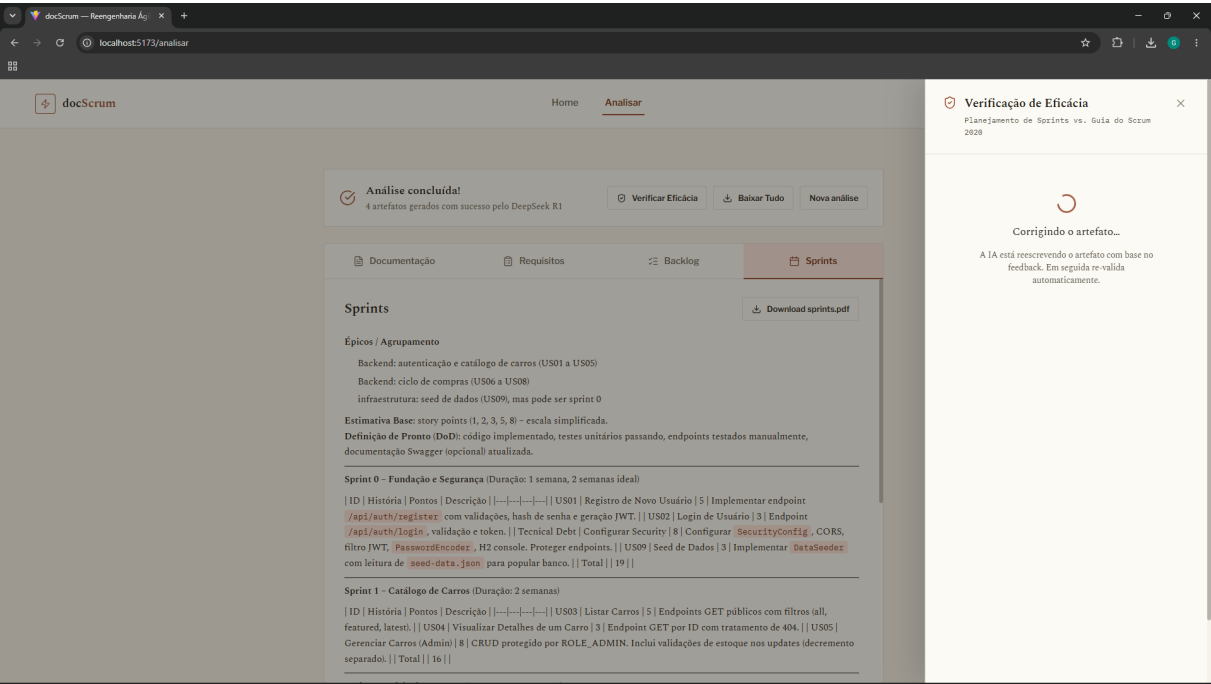
Fonte: autoria própria.

No exemplo apresentado, o artefato recebeu nota global 3,7/10, decomposta em três dimensões: *Conformidade Estrutural* (4/10), *Cobertura Conceitual* (3/10) e *Adequência Semântica* (4/10). O juiz fundamentou cada nota em citações literais do Guia do Scrum, apontando a omissão de *Metas de Sprint*, a ausência de referência aos eventos formais e a falta da abordagem empírica e adaptativa do *framework* (lacunas que aproximam o artefato de um cronograma de projeto tradicional). O botão *Corrigir agora* permite acionar a próxima etapa do ciclo.

A.9 Ciclo de Correção Fundamentada

Acionada a correção, o sistema submete o artefato e o *feedback* estruturado do juiz ao papel de *revisor*, que reescreve o artefato corrigindo exclusivamente as lacunas explicitamente apontadas pelo juiz. A Figura 15 apresenta o estado de processamento dessa etapa.

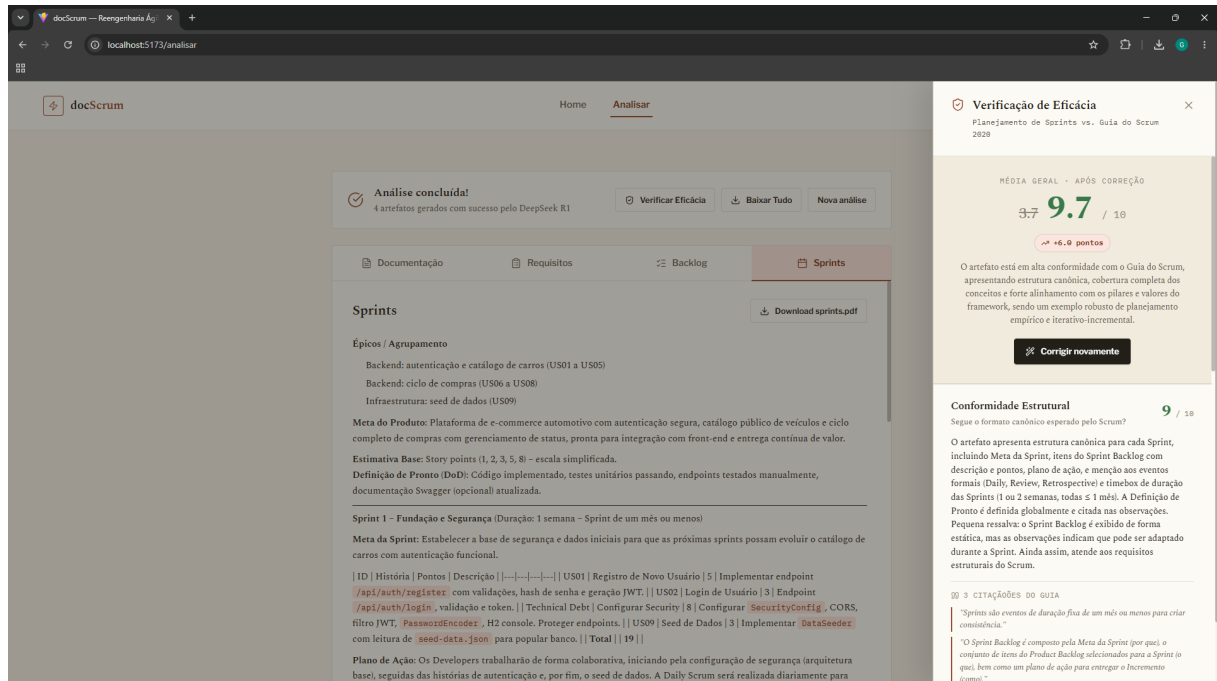
Figura 15 – Processamento da etapa de correção pelo revisor com base no *feedback* do juiz.



Fonte: autoria própria.

Ao concluir a reescrita, o revisor reenvia o artefato corrigido ao juiz, que o reavalia automaticamente. A Figura 16 apresenta o resultado após o ciclo completo, evidenciando a elevação substancial da nota global.

Figura 16 – Avaliação pós-correção: nota 9.7/10 após o ciclo de auto-crítica fundamentada.



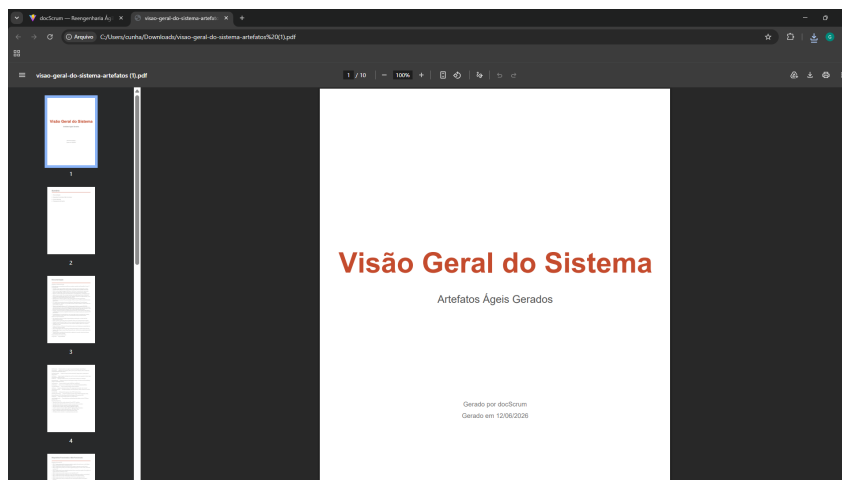
Fonte: autoria própria.

No exemplo, a nota global saltou de 3,7/10 (geração inicial) para 9,7/10 após o ciclo, um ganho de +6,0 pontos. As três dimensões elevaram-se de forma expressiva: *Conformidade Estrutural* de 4 para 9, *Cobertura Conceitual* de 3 para 10 e *Aderência Semântica* de 4 para 10. O juiz registrou que o artefato passou a apresentar estrutura canônica, cobertura completa dos conceitos esperados e forte alinhamento com os pilares e valores do *framework* (incluindo *Meta da Sprint*, *Sprint Backlog* com itens e plano de entrega, *Definition of Done*, *timebox* das *Sprints* e menção aos eventos formais), mantendo apenas pequena ressalva quanto ao detalhamento do *Sprint Backlog* como artefato vivo.

A.10 Documento Consolidado em PDF

Além da visualização em abas e do *download* individual de cada artefato, o sistema oferece a exportação de um documento único em formato PDF, acionada pelo botão *Baixar Tudo*. Esse documento reúne os quatro artefatos em um único arquivo apresentável, gerado de forma programática no próprio navegador. A Figura 17 apresenta a capa do documento.

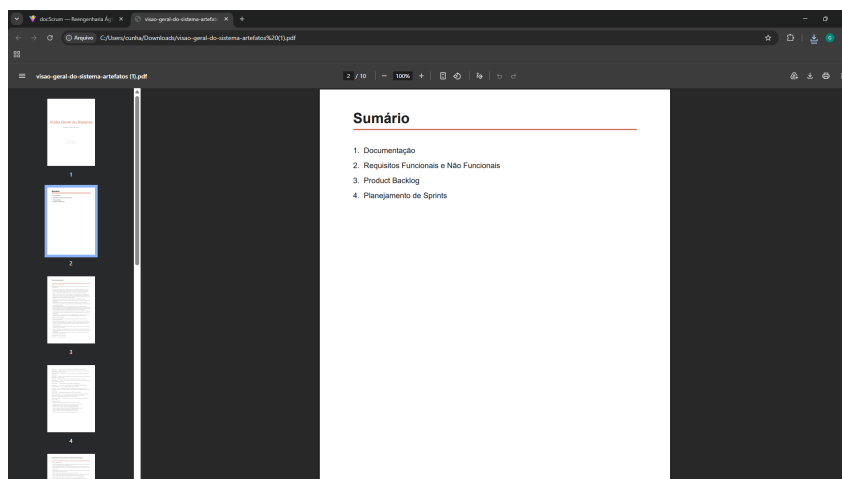
Figura 17 – Capa do documento consolidado em PDF.



Fonte: autoria própria.

A capa exibe o título *Visão Geral do Sistema*, o subtítulo *Artefatos Ágeis Gerados* e as linhas de identificação (*Gerado por docScrum* e a data de geração). Na sequência, o documento apresenta um sumário numerado com os quatro artefatos, conforme a Figura 18.

Figura 18 – Sumário do documento consolidado em PDF.



Fonte: autoria própria.

Após o sumário, cada artefato é apresentado em sua própria seção, com título destacado por linha decorativa e quebra de página entre seções. O formato resultante é adequado tanto para arquivamento corporativo quanto para compartilhamento com *stakeholders*, e preserva texto selecionável (não se trata de imagem rasterizada), por ser construído diretamente a partir do conteúdo dos artefatos.

A.11 Considerações sobre o Apêndice

Este apêndice demonstrou, por meio de capturas de tela do sistema em execução, o fluxo operacional completo do *docScrum*, desde o carregamento do código-fonte Java até a obtenção dos artefatos validados contra o Guia do Scrum 2020. As telas apresentadas evidenciam que a complexidade do *pipeline* de geração, validação e correção fundamentada permanece transparente ao usuário, que interage com o sistema por meio de uma interface enxuta e voltada às etapas centrais do processo de reengenharia ágil. O salto de qualidade observado no exemplo ilustrativo (de 3,7 para 9,7 em uma escala de 0 a 10) prefigura, em caráter visual, o impacto quantitativo do ciclo de autocrítica fundamentada mensurado de forma sistemática no Capítulo 6, com replicação em $N = 5$ execuções independentes.