Universidade Estadual de Feira de Santana

Programa de Pós-Graduação em Ciência da Computação

Avaliação Empírica da Eficácia de Modelos de Linguagem Grande (LLMs) na Refatoração de Projetos Python

Pedro Carneiro de Souza

Feira de Santana

2025

Universidade Estadual de Feira de Santana

Programa de Pós-Graduação em Ciência da Computação

Pedro Carneiro de Souza

Avaliação Empírica da Eficácia de Modelos de Linguagem Grande (LLMs) na Refatoração de Projetos Python

Dissertação apresentada à Universidade Estadual de Feira de Santana como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Profª Drª. Larissa Rocha Soares Bastos

Coorientador: Profº Drº. Eduardo Figueiredo

Feira de Santana

2025

Ficha Catalográfica – Biblioteca Central Julieta Carteado

S717a Souza, Pedro Carneiro de

Avaliação empírica da eficácia de Modelos de Linguagem Grande (LLMs) na refatoração de Projetos Python./ Pedro Carneiro de Souza. – 2025.

129 f.: il.

Orientadora: Larissa Rocha Soares Bastos

Coorientador: Eduardo Figueiredo

Dissertação (mestrado) – Universidade Estadual de Feira de Santana, Programa de Pós-Graduação em Ciência da Computação, 2025.

1.Refatoração de código. 2.Manutenibilidade. 3.Modelos de Linguagem de Larga Escala (LLMs). 4.Python. I.Bastos, Larissa Rocha Soares, orient. II.Figueiredo, Eduardo, coorient. III.Universidade Estadual de Feira de Santana. IV.Título.

CDU: 004.43

Maria de Fátima de Jesus Moreira – Bibliotecária – CRB5/1120

ATA DA SESSÃO PÚBLICA DE DEFESA DE DISSERTAÇÃO DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO Nº 41

No dia 19 de agosto de 2025, às 09:00, na https://meet.google.com/tsv-pgvs-wmy, realizou-se a Sessão Pública de Defesa de Dissertação de Mestrado em Ciência da Computação número 41, do(a) mestrando(a) **Pedro Carneiro de Souza**, matrícula nº 23215036, intitulada *Avaliação Empírica da Eficácia de Modelos de Linguagem Grande (LLMs) na Refatoração de Projetos Python,* vinculada à Área de Ciência da Computação e Linha de Pesquisa Software e Sistemas Computacionais do Programa de Pós-Graduação em Ciência da Computação (PGCC), tendo como orientador(a) o(a) Professor(a) Larissa Rocha Soares Bastos. Inicialmente, a Banca Examinadora foi instalada, sendo composta pelos seguintes membros: o(a) Orientador(a) do(a) mestrando(a) e Presidente da Banca Examinadora, Dr(a). Larissa Rocha Soares Bastos (UNEB/PGCC(UEFS)), Dr(a). Ivan do Carmo Machado (UFBA) e Dr(a). Anderson Gonçalves Uchôa (UFC-Itapajé). Em seguida, foram esclarecidos os procedimentos e a palavra foi passada ao(à) mestrando(a), que apresentou o seu trabalho. Ao final da apresentação, a Banca Examinadora passou à arguição do(a) candidato(a). Ato contínuo, a Banca Examinadora reuniu-se para elaborar seu parecer final. Concluída a reunião, foi lido o parecer final sobre a dissertação apresentada, tendo a Banca Examinadora atribuído o conceito **APROVADO** à referida dissertação, sendo esta aprovação um requisito parcial para a obtenção do título de Mestre em Ciência da Computação. O(A) mestrando(a) terá 90 (noventa) dias para realizar modificações consideradas essenciais para a aprovação da dissertação, conforme o parecer exarado pela Banca Examinadora, anexo a este documento. Nada mais havendo a tratar, foi encerrada a sessão e lavrada a presente ata, abaixo assinada pelo Presidente e demais membros da Banca Examinadora.

Feira de Santana, 19 de agosto de 2025.

Documento assinado digitalmente
gov.br
LARISSA ROCHA SOARES BASTOS
Data: 19/08/2025 14:24:19-0300
Verifique em https://validar.iti.gov.br

Dr(a). Larissa Rocha Soares Bastos (Presidente)
Universidade do Estado da Bahia (UNEB/PGCC(UEFS))

Documento assinado digitalmente
gov.br
IVAN DO CARMO MACHADO
Data: 19/08/2025 20:09:24-0300
Verifique em https://validar.iti.gov.br

Dr(a). Ivan do Carmo Machado
Universidade Federal da Bahia (UFBA)

Documento assinado digitalmente
gov.br
ANDERSON GONCALVES UCHOA
Data: 19/08/2025 19:03:11-0300
Verifique em https://validar.iti.gov.br

Dr(a). Anderson Gonçalves Uchôa
Universidade Federal do Ceará-Itapajé (UFC-Itapajé)

PARECER DA BANCA EXAMINADORA SOBRE A DISSERTAÇÃO DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO Nº 41

Mestrando(a): **Pedro Carneiro de Souza**
Título da Dissertação: *Avaliação Empírica da Eficácia de Modelos de Linguagem Grande (LLMs) na Refatoração de Projetos Python*
Data: 19 de agosto de 2025
Horário: 09:00
Local: https://meet.google.com/tsv-pgvs-wmy

APROVADO:	X	REPROVADO:		INSUFICIENTE:	

RESULTADO:

Observações e sugestões da Banca Examinadora:

- Reestruturar e condensar os capítulos;
- Inclusão da seção de ameaças a validade;
- Ser mais crítico na discussão;
- Rever conceitos (ex.: prompt, refatoração)

Feira de Santana, 19 de agosto de 2025.

Documento assinado digitalmente
gov.br LARISSA ROCHA SOARES BASTOS
Data: 19/08/2025 14:24:19-0300
Verifique em https://validar.iti.gov.br

Dr(a). Larissa Rocha Soares Bastos (Presidente)

Documento assinado digitalmente
gov.br IVAN DO CARMO MACHADO
Data: 19/08/2025 20:09:24-0300
Verifique em https://validar.iti.gov.br

Dr(a). Ivan do Carmo Machado

Documento assinado digitalmente
gov.br ANDERSON GONCALVES UCHOA
Data: 19/08/2025 19:02:27-0300
Verifique em https://validar.iti.gov.br

Dr(a). Anderson Gonçalves Uchôa

Abstract

Code refactoring is an essential practice to ensure the quality and continuous evolution of software systems, especially in languages like Python, which demand high maintainability. Although static analysis tools such as SonarQube provide support in identifying issues, the refactoring process still presents challenges, such as preserving functionality and improving code readability. In this context, Large Language Models (LLMs), such as GPT-4, DeepSeek, and Claude AI, emerge as promising tools by combining advanced contextual analysis with automated code generation.

This study aims to evaluate the effectiveness of LLMs in refactoring Python code, focusing on correcting maintainability issues, identifying limitations, and proposing improvements. To this end, we conducted an empirical study with four widely used models: Copilot Chat 4o, LLaMA 3.3 70B Instruct, DeepSeek V3, and Gemini 2.5 Pro. Furthermore, this study also investigates whether these models perform better when used with more refined prompting techniques, such as few-shot learning. For this purpose, each LLM was submitted to two distinct prompting styles: zero-shot and few-shot, allowing a comparative analysis of the impact of these approaches on the quality of the generated refactorings.

We evaluated 150 methods with maintainability problems by LLM and by prompt technique, and the results indicate that, although the models achieved considerable effectiveness rates in the few-shot scenario—Gemini (64.67%), DeepSeek (64.00%), Copilot (63.33%), and LLaMA 3.3 70B (55.33%), all faced significant limitations. Among the main challenges observed were the introduction of new maintainability problems, runtime errors, failures in automated tests, and, in some cases, the failure to correct the originally identified issue.

Additionally, we conducted a human participant evaluation to analyze the readability of the code refactored by the models. The results indicate that 81.25% of the solutions were perceived as improvements, especially in structural aspects. However, there were also cases where readability was impaired, either due to the introduction of unnecessary complexity or lack of standardization in code style. These findings reinforce the need for caution when automatically adopting suggestions generated by LLMs, and highlight the importance of validation by developers in the final code review.

This work contributes a comparative analysis of the capabilities of LLMs, pointing

out their limitations and proposing practical methodologies for integrating AI into the code refactoring process. The results of this study aim to pave the way for new research, especially in the development of more efficient prompting techniques and in the evaluation of models yet to come. We hope these contributions help developers and researchers find more practical, reliable, and long-lasting solutions to improve software maintainability in everyday practice.

Keywords: Code Refactoring, Maintainability, Large-Scale Language Models (LLMs), Python.

Resumo

A refatoração de código é uma prática essencial para garantir a qualidade e a evolução contínua dos sistemas de software, especialmente em linguagens como Python, que exigem alta manutenibilidade. Embora ferramentas de análise estática, como o SonarQube, ofereçam suporte na identificação de problemas, o processo de refatoração ainda apresenta desafios, como a preservação da funcionalidade e a melhoria da legibilidade do código. Nesse cenário, os Modelos de Linguagem de Grande Escala (LLMs), como o GPT-4, DeepSeek e Claude AI, surgem como ferramentas promissoras por combinarem análise contextual avançada com geração automatizada de código.

Este estudo tem como objetivo avaliar a eficácia de LLMs na refatoração de código Python, com foco na correção de problemas de manutenibilidade, identificação de limitações e proposição de melhorias. Para isso, conduzimos um estudo empírico com quatro modelos amplamente utilizados: Copilot Chat 4o, LLaMA 3.3 70B Instruct, DeepSeek V3 e Gemine 2.5 Pro. Além disso, este estudo também investiga se esses modelos apresentam melhor desempenho quando utilizados com técnicas de prompting mais refinadas, como o few-shot por exemplo. Para isso, cada LLM foi submetido a dois estilos distintos de prompting: zero-shot e few-shot, permitindo uma análise comparativa do impacto dessas abordagens na qualidade das refatorações geradas.

Avaliamos 150 métodos com problemas de manutenibilidade por LLM e por técnica de prompt, e os resultados indicam que, embora os modelos tenham alcançado taxas consideráveis de eficácia no cenário few-shot, Gemini (64,67%), DeepSeek (64,00%), Copilot (63,33%) e LLaMA 3.3 70B (55,33%), todos enfrentaram limitações importantes. Entre os principais desafios observados estão: a introdução de novos problemas de manutenibilidade, erros de execução, falhas em testes automatizados e, em alguns casos, a não correção do problema original identificado.

Além disso, conduzimos uma avaliação com participantes humanos para analisar a legibilidade do código refatorado pelos modelos. Os resultados indicam que 81,25% das soluções foram percebidas como melhorias, especialmente em aspectos estruturais. No entanto, também foram observados casos em que a legibilidade foi prejudicada, seja pela introdução de complexidade desnecessária ou pela falta de padronização no estilo do código. Esses achados reforçam a necessidade de cautela ao adotar au-

tomaticamente as sugestões geradas por LLMs, além de destacar a importância da validação por desenvolvedores na revisão final do código.

Este trabalho contribui com uma análise comparativa das capacidades dos LLMs, apontando suas limitações e propondo metodologias práticas para a integração de IA no processo de refatoração de código. Os resultados deste estudo buscam contribuir para abrir caminho para novas pesquisas, principalmente no desenvolvimento de técnicas de prompting mais eficientes e na avaliação de modelos que ainda estão por vir. Esperamos que essas contribuições ajudem desenvolvedores e pesquisadores a encontrar soluções mais práticas, confiáveis e duradouras para melhorar a manutenibilidade do software no dia a dia.

Palavras-chave: Refatoração de Código, Manutenibilidade, Modelos de Linguagem de Larga Escala (LLMs), Python.

Prefácio

Esta dissertação de mestrado foi submetida à Universidade Estadual de Feira de Santana (UEFS) como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

A dissertação foi desenvolvida no Programa de Pós-Graduação em Ciência da Computação (PGCC), tendo como orientadora **Profª Drª. Larissa Rocha Soares Bastos**. **Profº Drº. Eduardo Figueiredo** foi coorientador deste trabalho.

Agradecimentos

Gostaria de iniciar meus agradecimentos, primeiramente, a Deus e ao nosso Senhor Jesus Cristo. Sem Eles, eu não teria sequer começado este mestrado. Sou profundamente grato pela saúde, força, sabedoria e paciência concedidas durante toda esta jornada.

Agradeço, de coração, à minha família,especialmente à minha esposa, Mônica Dias, e à minha filha, Maria Vitória. Vocês são a razão da minha vida. Obrigado pelo apoio incondicional, pela paciência e pela compreensão nos dias difíceis. Amo vocês demais.

À minha orientadora, Profª Drª Larissa Rocha, deixo meu sincero agradecimento por sua firmeza, incentivo e orientação, mesmo nos momentos em que pensei em desistir, você sempre me manteve motivado. Sua trajetória e profissionalismo são admiráveis. Ao meu coorientador, professor Drº Eduardo, um ser humano incrível, de humildade ímpar e competência extraordinária, minha gratidão por todo o aprendizado. Ao colega Henrique Nunes, minha gratidão especial por toda a ajuda, especialmente na fase de elaboração desta dissertação. Sua generosidade e apoio foram fundamentais. Agradecer também aos avaliadores desta dissertação, pelo tempo dedicado à leitura, sugestões e contribuições que enriqueceram meu o trabalho.

Aos professores do programa, agradeço por todo o conhecimento compartilhado, pela compreensão nos momentos difíceis e por contribuírem de forma significativa para minha formação acadêmica. Aos colegas do curso, companheiros de jornada, com quem dividi momentos de angústia, dificuldade, mas também de muita alegria e descontração. Um agradecimento especial ao colega Alex, um ser humano de grande coração, sempre disposto a colaborar com todos. Obrigado, meu amigo!

A todos os participantes da avaliação humana, que gentilmente doaram seu tempo e suas percepções para este estudo, meu muito obrigado. Desejo paz e sucesso a cada um de vocês.

Agradeço também aos meus colegas de trabalho, que compreenderam os desafios de conciliar a vida profissional com os compromissos acadêmicos. Em especial, ao amigo Carlos Alberto, à querida, ex secretária de educação de Ipirá, um ser de luz, a profª Vanda Barreto e à dedicada atual secretária Iandra Gusmão.

"Com grandes poderes, vêm grandes responsabilidades."

– Tio Ben

Dedico este trabalho, acima de tudo, a Deus, meu refúgio e força em cada etapa desta jornada, por me sustentar nas dúvidas e me guiar nas decisões.

À minha família, meu maior alicerce. À minha esposa, por estar ao meu lado em cada madrugada silenciosa e em cada momento de desânimo, lembrando-me com amor e firmeza do propósito que me trouxe até aqui. Sua força me impulsionou. À minha filha, luz do meu caminho, cuja alegria e carinho me deram ânimo nos dias mais difíceis.

Sumário

Alinhamento com a Linha de Pesquisa

Linha de Pesquisa: Engenharia de Software e Qualidade de Software

Esta dissertação integra a linha de pesquisa Engenharia de Software e Qualidade de Software, fundamentando suas conclusões em dados concretos e evidências coletadas durante os experimentos. Os resultados deste estudo têm o potencial de contribuir para o avanço de pesquisas nas áreas de inteligência artificial e engenharia de software, ampliando o entendimento sobre a aplicação de modelos de linguagem no desenvolvimento e manutenção de software.

O principal objetivo desta pesquisa é avaliar a eficácia dos Modelos de Linguagem Grande na refatoração e correção de problemas relacionados à manutenibilidade em código Python, buscando identificar suas capacidades e limitações nesse contexto.

Lista de Tabelas

Lista de Figuras

Lista de Abreviações

Abreviação	Descrição
LLM	Modelos de Linguagem de Grande Escala
CCM	Complexidade Cognitiva de funções não deve ser muito alta
FMP	Funções, métodos e lambdas não devem ter muitos parâmetros
FNC	Nomes de funções devem seguir convenção de nomenclatura
BCI	Verificações booleanas não devem ser invertidas
SLD	Literais de string não devem ser duplicados
LVN	Variáveis locais e parâmetros devem seguir convenção de nomenclatura
UFP	Parâmetros não utilizados devem ser removidos
MIS	Estruturas condicionais ”if”mescláveis devem ser combinadas
ULV	Variáveis locais não utilizadas devem ser removidas
SES	Métodos ‘startswith‘/‘endswith‘ devem ser usados ao invés de slicing
BSV	Ramificações condicionais não devem ter implementação idêntica
TUT	Tags TODO devem ser rastreadas

Capítulo 1

Introdução

1.1 Considerações Preliminares

A busca contínua pela excelência na qualidade de software é um dos principais desafios da engenharia de software (Sommerville, 2016). Um código de alta qualidade não apenas atende aos requisitos funcionais e não funcionais, mas também facilita a evolução contínua do software, garantindo sua relevância em um cenário tecnológico em constante mudança (Pressman et al., 2021; Sommerville, 2016).

Garantir essa qualidade exige atenção à manutenibilidade e legibilidade, aspectos frequentemente negligenciados em projetos com prazos apertados ou equipes multidisciplinares (Martin, 2008; Sommerville, 2016). Como destacado por Fowler (2019), a qualidade do código é crucial, pois passamos muito mais tempo lendo do que escrevendo código, dependendo de sistemas claros e bem estruturados para realizar correções ou adicionar funcionalidades sem comprometer a estabilidade.

Nesse contexto, a prática da refatoração de código tem se estabelecido como uma técnica fundamental para assegurar a qualidade e a evolução contínua dos software (Martin, 2008). Refatorar consiste em modificar a estrutura interna do código para torná-lo mais claro e organizado, sem alterar seu comportamento funcional (Fowler, 2019). Esse processo pode envolver ações como eliminar duplicidades, simplificar métodos extensos, reorganizar classes ou adotar nomes de variáveis mais intuitivos (Fowler, 2019). Neste estudo, focamos nossa análise no *pure refactoring* (Fowler, 2019), em que as mudanças têm como único propósito a melhoria da legibilidade e da manutenibilidade do código, sem a introdução de novas funcionalidades.

Ao longo dos anos, ferramentas como o SonarQube[1] têm desempenhado um papel crucial ao apoiar os desenvolvedores nesse processo, identificando padrões conhecidos como *Code Smells*, por exemplo. Esses padrões sinalizam potenciais problemas de manutenibilidade, como métodos excessivamente longos ou complexos, código fora

[1] `https://www.sonarsource.com/`

dos padrões de projetos e que dificultam o entendimento do sistema e aumentam o risco de erros futuros (SonarSource, 2024).

Apesar do suporte fornecido por ferramentas automatizadas, a refatoração ainda demanda um esforço significativo por parte dos desenvolvedores Fowler (2019). É nesse cenário que os Modelos de Linguagem de Larga Escala (*Large Language Models – LLMs*) emergem como uma solução com grande potencial. Esses modelos, como GPT-4[2], Codex[3], Gemini[4], Copilot Chat [5] e Llama[6], foram originalmente concebidos para processar linguagem natural, mas rapidamente demonstraram potencial para lidar com tarefas relacionadas à programação (Al Madi, 2022; Xueying et al., 2024; AlOmar et al., 2024), incluindo geração de código, depuração e, mais recentemente, refatoração (Ziegler et al., 2024; Nunes et al., 2025; Liu et al., 2024; Shirafuji et al., 2023). Ao combinar grandes volumes de dados de treinamento com arquiteturas avançadas de redes neurais, esses modelos são capazes de compreender contextos complexos e fornecer soluções rápidas para problemas que tradicionalmente demandariam horas de trabalho humano Brown et al. (2020), demonstrando seu potencial em tarefas de programação e refatoração.

Pesquisas recentes têm destacado o impacto crescente das tecnologias baseadas em modelos de linguagem na engenharia de software. Por exemplo, o estudo de Yetistiren et al. (2022) avaliou o GitHub Copilot e constatou que ele foi capaz de gerar código válido em 91,5% dos casos, com 28,7% de soluções consideradas corretas e 51,2% parcialmente corretas. Os autores consideram o Copilot uma ferramenta promissora no auxílio ao desenvolvimento de software, embora apontem limitações importantes e a necessidade de revisão cuidadosa do código gerado.

De forma semelhante, Coello e Kouatly (2024) demonstraram a eficácia do GPT-4 na realização de tarefas complexas de geração de código, em comparação com outros modelos, como Google Bard, Claude e o modelo integrado ao Microsoft Bing[7], alcançando uma taxa de sucesso de 86,23% em testes específicos. Embora o estudo se concentre na geração de código, vale destacar que a tarefa de refatoração apresenta desafios adicionais, pois exige que as modificações preservem a funcionalidade original enquanto melhoram a estrutura do código(Fowler, 2019).Essa tarefa é particularmente crítica em linguagens como Python, que, apesar da simplicidade sintática, exige maior rigor na manutenção devido à ausência de tipagem explícita e de mecanismos automáticos de verificação (Rossum et al., 2001), tornando a refatoração uma atividade delicada e essencial.

Neste estudo, conduzimos uma avaliação empírica para medir a eficácia dos LLMs no processo de refatoração de código Python com problemas de manutenibilidade,

[2] `https://openai.com/gpt-4`
[3] `https://openai.com/blog/openai-codex`
[4] `https://gemini.google.com/`
[5] `https://github.com/settings/copilot/features`
[6] `https://www.llama.com/`
[7] `https://www.bing.com/?setlang=pt-br`

extraídos de projetos hospedados no GitHub. Escolhemos Python por sua ampla adoção na academia e na indústria, especialmente em ciência de dados e inteligência artificial, além de sua compatibilidade com ferramentas de análise como o SonarQube. Para identificar os códigos com problemas, utilizamos o SonarQube (versão Community Edition v10.5.1)[8] , com base em 12 regras distintas definidas pela ferramenta.

Após essa identificação, os trechos de código com problemas foram submetidos à refatoração por quatro modelos de LLMs: Copilot Chat[9] baseado no GPT-4 da OpenIA; Llama 3.3 70B Instruct[10], criado pela Meta e modelo de código aberto; DeepSeek-V3[11], desenvolvido pela empresa chinesa DeepSeek Inc e também de código aberto; e Gemini 2.5 Pro[12], criado pelo Google, para que sugerissem refatorações visando resolver os problemas detectados. Avaliamos duas abordagens distintas de prompting: inicialmente *zero-shot* e, em seguida, *few-shot*, com o objetivo de avaliar se esses modelos melhoram seu desempenho com técnicas de prompts mais refinadas.

Os resultados evidenciam que, entre os LLMs avaliados, o desempenho foi semelhante nas duas estratégias de *prompting* (zero-shot e few-shot), conforme indicado pela análise estatística. Na técnica zero-shot, o modelo DeepSeek apresentou a maior taxa de sucesso (56,67%), seguido pelo Gemini (54,67%), Copilot (52,5%) e LLaMa (44,00%). Em contrapartida, todos os modelos apresentaram desempenho superior quando aplicados à técnica de *prompt* few-shot, comprovando estatisticamente uma melhora nos valores agregados.

Esses achados sugerem que os LLMs possuem potencial promissor para apoiar o processo de refatoração de código Python, sobretudo na correção de problemas de manutenibilidade. Entretanto, ainda apresentam limitações relevantes, o que reforça a necessidade de revisão e avaliação humana para assegurar sua aplicação em cenários práticos, tanto no contexto acadêmico quanto no industrial.

1.2 Objetivos

O processo de refatoração de código, embora essencial para garantir a qualidade e a manutenibilidade do software, ainda impõe desafios aos desenvolvedores, mesmo com o suporte de ferramentas de análise estática, como o SonarQube. Diante da popularização dos Modelos de Linguagem de Grande Escala e de seu uso crescente na engenharia de software, torna-se relevante investigar sua eficácia no apoio à refatoração de código.

[8] `https://www.sonarsource.com/products/sonarqube/`
[9] `https://github.com/settings/copilot/features`
[10] `https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/`
[11] `https://chat.deepseek.com/`
[12] `https://gemini.google.com/app?hl=pt-BR`

1.2.1 Objetivo Geral

Avaliar a eficácia de LLMs no apoio à refatoração de código Python, com foco na correção de problemas de manutenibilidade identificados por ferramentas de análise estática.

1.2.2 Objetivos Específicos

- Avaliar e comparar o desempenho de quatro LLMs amplamente utilizados (Copilot Chat 4o, LLaMA 3.3 70B Instruct, DeepSeek-V3 e Gemini 2.5 Pro) na refatoração de métodos com problemas de manutenibilidade em Python;
- Comparar os resultados obtidos por meio das técnicas de prompting *zero-shot* e *few-shot*;
- Investigar as principais limitações apresentadas pelos modelos ao realizar refatorações automatizadas;
- Avaliar a legibilidade do código gerado pelos LLMs a partir da perspectiva de programadores humanos;
- Apontar estratégias para mitigar essas limitações e ampliar o uso efetivo de LLMs em cenários reais de desenvolvimento.

1.2.3 Questões de Pesquisa

Com o propósito de atingir os objetivos desta pesquisa, propomos as seguintes questões de investigação:

- **RQ1:** Qual é a eficácia dos LLMs na correção de problemas de manutenibilidade em código Python?
- **RQ2:** Quais tipos de erros os LLMs cometem ao refatorar código Python com problemas de manutenibilidade?
- **RQ3:** Como diferentes técnicas de prompting, como *zero-shot* e *few-shot*, afetam o desempenho dos LLMs na refatoração?
- **RQ4:** Em que medida os desenvolvedores consideram o código gerado pelos LLMs mais legível e compreensível?

1.3 Contribuições

Este trabalho contribui para o avanço da área engenharia de software ao preencher uma lacuna na literatura sobre o uso de modelos de linguagem de grande escala na refatoração automática de código, um tema ainda pouco explorado em comparação à geração de código. Diferentemente de estudos anteriores que se concentram em métricas de desempenho puramente automáticas, nossa pesquisa combina análise

quantitativa, baseada em regras de manutenibilidade do SonarQube, com uma avaliação qualitativa conduzida por desenvolvedores. Além disso, investigamos de forma sistemática o impacto de diferentes técnicas de prompting (zero-shot e few-shot) na eficácia da refatoração, algo pouco discutido em trabalhos relacionados. Assim, o estudo promove a área ao fornecer evidências empíricas de que LLMs já apresentam desempenho promissor para apoiar a refatoração corrigindo problemas de manutenibilidade, e ao mesmo tempo em que revela suas limitações e aponta direções para o uso responsável e supervisionado dessas ferramentas na engenharia de software.

1.4 Organização do Trabalho

Este trabalho está estruturado da seguinte forma:

- **Capítulo 2:** apresenta a fundamentação teórica, abordando os principais conceitos do estudo, como análise estática, refatoração de código e os modelos de LLMs avaliados.
- **Capítulo 3:** reúne uma revisão da literatura relacionada ao tema, destacando estudos e abordagens relevantes.
- **Capítulo 4:** descreve a metodologia adotada, incluindo os critérios de seleção dos projetos, os procedimentos aplicados e o detalhamento das etapas do processo.
- **Capítulo 5:** apresenta os resultados obtidos e analisa as vantagens e limitações do uso de LLMs na refatoração de código.
- **Capítulo 6:** discute os achados e seu impacto no entendimento do papel das LLMs na refatoração, além de propor direções para pesquisas futuras.
- **Capítulo 7:** reúne as conclusões do estudo, sintetizando as principais atividades e resultados alcançados.

1.5 Considerações Finais

Neste capítulo, introduzimos o tema, contextualizando o problema e destacando a relevância da pesquisa. Foram apresentados os objetivos geral e específicos, as questões de pesquisa e as principais contribuições teóricas e práticas, especialmente no uso de modelos de linguagem na refatoração de código. Por fim, descrevemos a organização da dissertação, indicando a estrutura dos capítulos.

No capítulo seguinte, apresentamos a revisão bibliográfica, discutindo os principais estudos relacionados que fundamentam esta pesquisa.

Capítulo 2

Revisão Bibliográfica

Neste capítulo, buscamos construir a base teórica que sustenta este estudo, explorando os principais conceitos ligados ao tema investigado.

- A Seção 2.1 apresenta conceito de refatoração e manutenção de código e seus desafios, além de uma amostra com exemplos;
- A Seção 2.2 discute sobre ferramentas de análise estática de código;
- A Seção 2.3 apresenta os Modelos de Linguagem de Grande Escala, destacando suas características, funcionamento e aplicações no desenvolvimento de software;
- A Seção 2.4 considerações finais do capítulo

2.1 Refatoração e Manutenção de Código

A refatoração é uma prática fundamental para garantir a qualidade e a sustentabilidade do software ao longo do tempo. Consiste na reestruturação do código-fonte sem alterar seu comportamento funcional, com o objetivo de torná-lo mais claro, legível e de fácil manutenção (Fowler, 2019; Martin, 2008). Essa prática facilita a correção de falhas, a implementação de novas funcionalidades e a evolução contínua dos sistemas.

Apesar de seus benefícios, refatorar pode ser desafiador, especialmente em sistemas legados ou com cobertura limitada de testes automatizados (Feathers, 2004). Muitas vezes, o código apresenta sinais de má estrutura, os chamados code smells, que não impedem seu funcionamento, mas dificultam sua compreensão e manutenção (Fowler, 2019; Martin, 2008). Identificar essas áreas problemáticas exige atenção, sobretudo quando o sistema aparenta estar "funcionando bem", mas está mal projetado internamente (Beck, 2004). Ao promover melhorias estruturais, a refatoração contribui diretamente para a manutenibilidade, tornando a base de código mais robusta, menos propensa a falhas e mais fácil de evoluir. Em projetos de longo prazo,

essa prática é essencial para preservar a qualidade do software e reduzir a dívida técnica acumulada (Fowler, 2019; Martin, 2008; Kruchten et al., 2012; Feathers, 2004; Kim et al., 2018).

2.1.1 Desafios Práticos da Refatoração

Refatorar código é uma tarefa complexa e desafiadora, que envolve diversos aspectos, como a identificação de trechos problemáticos, especialmente quando o código está funcionando corretamente (Fowler, 2019). A falta de testes automatizados, assim como a necessidade de lidar com sistemas antigos, frequentemente mal documentados ou sem documentação alguma, aumenta a dificuldade da refatoração (Beck, 2004; Feathers, 2004). Além disso, fatores como prazos apertados, recursos limitados e mudanças frequentes nos requisitos contribuem significativamente para os desafios enfrentados na refatoração do código.

Refatorar, assim como qualquer atividade técnica, exige planejamento e o uso de técnicas adequadas para cada tipo de problema identificado no código (Fowler, 2019; Feathers, 2004). Para que o processo de refatoração seja bem-sucedido, é fundamental aplicar essas técnicas de forma criteriosa. Mesmo códigos que seguem boas práticas ou padrões estabelecidos podem apresentar problemas de manutenibilidade, conhecidos como code smells, como métodos excessivamente longos, variáveis não declaradas, alta complexidade cognitiva, métodos com muitos parâmetros ou o uso inadequado de palavras-chave como nomes de variáveis, entre outros (SonarSource, 2024).

2.1.2 Exemplos de Códigos Refatorados com Problemas de Manutenibilidade

Nesta seção, apresentamos exemplos de código que passaram por refatoração devido a problemas de manutenibilidade. Os exemplos foram retirados do livro Refatoração: *Aperfeiçoando o Design de Códigos Existentes de Martin Fowler* (2019) e da documentação oficial do SonarQube (SonarSource, 2024). Cada exemplo ilustra uma ou mais práticas de refatoração aplicadas para melhorar a legibilidade, modularidade e facilidade de manutenção do código, destacando técnicas como Extração de Métodos, Renomeação de Variável, Internacionalização de Funções , entre outras.

1. **Extração de métodos (Extract Method)**. No Código 2.1, exemplificamos a técnica de refatoração conhecida como *Extract Method*. O código original apresenta alta complexidade cognitiva, dificultando sua leitura e compreensão. A refatoração consistiu na divisão do método principal em métodos auxiliares, linhas 11 e 17, o que contribui para melhorar a legibilidade e a manutenibilidade do código (SonarSource, 2024; Campbell e Papapetrou, 2013). Além disso, este exemplo evidencia como a decomposição de métodos complexos facilita futuras alterações e amplia a compreensão do fluxo do programa para outros desenvolvedores.

Código 2.1: Complexidade Congnitiva Alta

```
1
2 # Original
3 def process_eligible_users(users):
4     for user in users:              # +1 (for)
5         if ((user.is_active and     # +1 (if) +1 (nested) +1 (
              multiple conditions)
6             user.has_profile) or    # +1 (mixed operator)
7             user.age > 18 ):
8             user.process()
9 # ------------------------------------------------------------
10 # Refatorado
11 def process_user(user):
12     if user.is_active():             # +1 (if)
13         process_active_user(user)
14     else:                            # +1 (else)
15         process_inactive_user(user)
16
17 def process_active_user(user):
18     if user.has_profile():           # +1 (if) +1 (nested)
19         ... # process active user with profile
20     else:                            # +1 (else)
21         ... # process active user without profile
22
23 def process_inactive_user(user):
24     if user.has_profile():           # +1 (if) +1 (nested)
25         ... # process inactive user with profile
26     else:                            # +1 (else)
27         ... # process inactive user without profile
```

2. **Consolidação Condicional Ifs (Consolidate Conditional Expression.)** Neste outro exemplo, o código original contém dois blocos `if` aninhados, linhas 2 e 3 , o que aumenta sua complexidade e dificulta a leitura. A refatoração consistiu em unificar essas condições em uma única expressão lógica, tornando o fluxo do programa mais claro e facilitando sua manutenção (SonarSource, 2024; Campbell e Papapetrou, 2013).

Código 2.2: Complexidade Congnitiva Alta

```
1 # Original
2 if condition1:
3     if condition2:             # Noncompliant
4         # ...
5 # ------------------------------------------------------------
6 # Refatorado - Copilot
7 if condition1 and condition2:  # Compliant
8     # ...
```

3. **Extração de String Duplicada para Constante (Extract Duplicate Strings for Constant.)** Neste exemplo, a técnica aplicada foi a Extração de String Duplicada para Constante (Introduce Constant). A string `action` aparecia

repetidamente em três partes do código, linhas 3, 4 e 5, o que pode dificultar a manutenção e aumentar o risco de inconsistências em futuras modificações. Com a refatoração, essa string foi atribuída a uma constante nomeada `ACTION_1`, linha 16, melhorando a legibilidade e contribuindo para a manutenibilidade do código (SonarSource, 2024; Campbell e Papapetrou, 2013).

Código 2.3: Extração de Strings Duplicadas para Constante

```
1  # Original
2  def run():
3      prepare("action1")  # Noncompliant - "action1" is
           duplicated 3 times
4      execute("action1")
5      release("action1")
6
7  @app.route("/api/users/", methods=['GET', 'POST', 'PUT'])
8  def users():
9      pass
10
11 @app.route("/api/projects/", methods=['GET', 'POST', 'PUT'])  #
       Compliant - strings inside decorators are ignored
12 def projects():
13     pass
14 # ------------------------------------------------------------
15 # Refatorado - Copilot
16 ACTION_1 = "action1"
17
18 def run():
19     prepare(ACTION_1)
20     execute(ACTION_1)
21     release(ACTION_1)
```

Os exemplos a seguir estão em linguagem JavaScript e foram retirados do livro de Fowler (2019). Optamos por incluir exemplos em uma linguagem diferente da utilizada no experimento (Python) com o objetivo de diversificar os casos apresentados.

4. **Renomear Variável (Rename Variable.)** Neste exemplo, aplicou-se a renomeação de variável. Segundo Fowler (2019), nomes pouco claros comprometem legibilidade e manutenibilidade. No Código 2.4, a variável `"a"` (linha 2) foi renomeada para `"area"` (linha 5), refletindo melhor o valor calculado.

Código 2.4: Renomear Variável

```
1      # Original
2      let a = height * width;
3      #
           ------------------------------------------------------------
4      # Refatorado - Copilot
5      let area = height * width;
```

5. **Internalizar Função (Inline Function.)** Esta técnica de refatoração consiste em substituir uma chamada a uma função simples pelo próprio corpo dessa

função, eliminando a função original e inserindo seu conteúdo diretamente no local onde ela era chamada (Fowler, 2019). O código refatorado na linha 11 demonstra como o corpo da função `moreThanF` foi incorporado à função **getRating**, simplificando a estrutura e eliminando a necessidade de uma função auxiliar.

Código 2.5: Internalizar Função

```
1 # Original
2 function getRating(driver) {
3   return moreThanFiveLateDeliveries(driver) ? 2 : 1;
4 }
5
6 function moreThanFiveLateDeliveries(driver) {
7   return driver.numberOfLateDeliveries > 5;
8 }
9
10 # Refatorado
11 function getRating(driver) {
12   return driver.numberOfLateDeliveries > 5 ? 2 : 1;
13 }
```

2.2 Ferramenta de Análise Estática de Código

Ferramentas de análise estática são fundamentais no desenvolvimento de software, pois inspecionam o código sem executá-lo, identificando antecipadamente problemas como code smells, bugs, falhas de design e vulnerabilidades de segurança (Campbell e Papapetrou, 2013; Lenarduzzi et al., 2020; Ruohonen et al., 2021). Essa análise precoce garante maior conformidade com padrões de qualidade desde as etapas iniciais do projeto.

Entre as ferramentas mais utilizadas estão PMD, Pylint, ESLint, FindBugs e SonarQube, cada uma com focos e linguagens específicas, mas todas voltadas à melhoria da qualidade do software (Fowler, 2019).

Dentre essas ferramentas, destaca-se o SonarQube[1], uma solução de código aberto que fornece uma análise abrangente da saúde do código, com suporte a múltiplas linguagens e integração com pipelines de integração contínua.

2.2.1 SonarQube

O SonarQube é uma das ferramentas de análise estática mais populares e eficazes na detecção de problemas de qualidade no código-fonte (SonarSource, 2024; Lenarduzzi et al., 2020). Ele oferece um conjunto abrangente de regras que identificam code smells, vulnerabilidades e violações de boas práticas, promovendo um código mais

[1] `https://www.sonarqube.org`

limpo, seguro e manutenível. Sua fácil integração com IDEs e pipelines de CI/CD o torna especialmente útil em ambientes DevOps.

Além de auxiliar na padronização e organização do código desde as fases iniciais do desenvolvimento, o SonarQube contribui para a evolução contínua dos sistemas. Segundo Lenarduzzi et al. (2020), mais de 85.000 organizações utilizam a ferramenta em seus processos de Integração Contínua, reforçando seu papel central na melhoria da qualidade de software ao longo do tempo.

Estudos também investigam a adesão dos desenvolvedores aos alertas gerados pela ferramenta. Marcilio et al. (2019) analisaram projetos de código aberto e instituições governamentais, revelando que apenas 13% das violações reportadas são efetivamente corrigidas. O esforço corretivo também é concentrado: cerca de 20% dos desenvolvedores respondem por 80% das correções. Apesar da baixa taxa de resolução, os profissionais reconhecem a utilidade dos alertas para a melhoria da qualidade e manutenção do software.

2.3 Modelos de Linguagem de Grande Escala (LLMs)

Modelos de Linguagem em Grande Escala (LLMs) são redes neurais profundas capazes de prever a próxima palavra em uma sequência textual, a partir do aprendizado obtido com extensos conjuntos de dados linguísticos (Brown et al., 2020; Jurafsky e Martin, 2025). Essa característica possibilita a geração de textos coerentes e a execução de tarefas avançadas de compreensão da linguagem natural (Vaswani et al., 2017). Esses modelos vêm sendo aplicados com sucesso em diversas áreas, como tradução automática, respostas a perguntas, criação de textos e até mesmo na produção de código em diferentes linguagens, como demonstrado pelo GPT-3, desenvolvido pela OpenAI (Brown et al., 2020). No campo do desenvolvimento de software, os LLMs têm se destacado ao oferecer suporte aos programadores em atividades como a geração e refatoração de código, o que pode melhorar a qualidade do código e aumentar a eficiência da equipe de desenvolvimento (Liu et al., 2024; Chen et al., 2021; Nunes et al., 2025). Essa ajuda é especialmente valiosa em linguagens como Python, que valorizam a clareza e a organização do código (SonarSource, 2024).

1. **Copilot Chat:** O GitHub Copilot Chat é um assistente de programação baseado em modelos de linguagem de grande escala, desenvolvido em colaboração entre GitHub e OpenAI. Derivado do Codex (Chen et al., 2021), suas versões recentes utilizam arquiteturas avançadas como o GPT-4-turbo, permitindo interações em linguagem natural diretamente no ambiente de desenvolvimento (OpenAI, 2024). Integrado ao GitHub e Visual Studio Code, o Copilot Chat oferece sugestões de código, refatoração de trechos existentes e correção de erros, contribuindo para aumentar a produtividade dos programadores (Nguyen e Nadi, 2022; GitHub Copilot Documentation, 2024).

2. **Llama:** O LLaMA (Large Language Model Meta AI) é um modelo de linguagem grande de código aberto desenvolvido pela Meta (Meta AI, 2023). Projetado para ser eficiente e utilizar menos recursos computacionais, o LLaMA pode ser treinado e aplicado em diferentes contextos (Touvron et al., 2023), realizando tarefas de Processamento de Linguagem Natural (NLP), como geração de texto, tradução automática e análise de sentimentos (Meta AI, 2023).

   A versão mais recente, LLaMA-3 (Touvron et al., 2024), mantém o foco em eficiência e acessibilidade, oferecendo desempenho elevado em diversas tarefas de NLP e permitindo que desenvolvedores e pesquisadores utilizem e adaptem o modelo a diferentes necessidades. Disponibilizado como código aberto, o LLaMA-3 proporciona uma ferramenta robusta e escalável para o desenvolvimento de aplicativos baseados em IA (Touvron et al., 2024; Meta AI, 2023).

3. **Gemini:** O Gemini é um modelo de linguagem grande (LLM) desenvolvido pela DeepMind, empresa do Google (Google DeepMind, 2024). Projetado para lidar com tarefas como criação, refatoração e análise de código em diversas linguagens (Google Gemini, 2024), o Gemini busca superar limitações de versões anteriores e oferecer interpretação precisa de contextos de longo alcance, proporcionando respostas mais acuradas e interações mais fluidas, especialmente em diálogos prolongados (Pichai, 2024). Além disso, o modelo contribui para a integração da IA em diferentes produtos e aplicações do Google (Google DeepMind, 2024).

4. **DeepSeek** O DeepSeek é um modelo de linguagem de grande escala desenvolvido pela startup chinesa DeepSeek AI. Gratuito e de código aberto, o modelo foi lançado recentemente e tem ganhado destaque por sua capacidade de lidar com contextos longos e analisar arquivos extensos de forma eficiente[2].

   Alguns estudos têm comparado o DeepSeek com outros modelos de LLMs. O trabalho de Guo et al. (2025), por exemplo, destaca a adoção de um processo de treinamento baseado em reforço por regras (rule-based RL), sem ajuste fino supervisionado (SFT). O estudo propõe uma estratégia de treinamento em múltiplas etapas, que inclui o uso de dados de início frio (cold-start) antes do RL. Essa abordagem permite ao modelo alcançar um desempenho significativo em tarefas de raciocínio e na modelagem de recompensas, otimizando o processo de aprendizado.

2.4 Considerações Finais

Este capítulo revisou os principais conceitos que fundamentam este trabalho, com ênfase na importância da melhoria contínua do código. Abordamos a refatoração como prática essencial para garantir legibilidade e manutenibilidade, e destacamos o papel das ferramentas de análise estática, como o SonarQube, na identificação

[2] `https://github.com/deepseek-ai/DeepSeek-V3`

precoce de problemas. Também discutimos os Modelos de Linguagem de Grande Escala, exemplificados por ferramentas como GPT, GitHub Copilot, Mistral, Claude e Gemini, evidenciando seu potencial na refatoração e na melhoria da qualidade do código.

No capítulo seguinte, apresentamos os trabalhos relacionados ao tema deste estudo. Discutimos pesquisas que abordam o impacto da geração e refatoração de código produzidos por Modelos de Linguagem de Grande Escala (LLMs).

Capítulo 3

Trabalhos Relacionados

Neste capítulo, apresentamos os estudos existentes na literatura que tratam do uso de Modelos de Linguagem de Grande Escala no contexto do desenvolvimento de software, com foco em tarefas como geração e refatoração de código. Discutimos os principais trabalhos recentes sobre o tema, destacando as abordagens adotadas e os resultados obtidos, a fim de compreender como essas tecnologias têm contribuído para a melhoria da produtividade e da qualidade no processo de desenvolvimento.

- A Seção 3.1 apresenta e discute o uso de LLMs em apoio ao desenvolvimento de Software;
- A Seção 3.2 discute estudos que avaliaram sobre LLMs aplicados a refatoração de código.
- A Seção 3.3 aborda estudos sobre os LLMs aplicados a refatoração de código.
- A Seção 3.4 discute lacunas identificadas.
- A Seção 3.5 apresenta considerações finais.

O estudo e desenvolvimento de Inteligência Artificial (IA) existe há décadas, como amplamente documentado na literatura acadêmica (McCorduck et al., 1977; Brunnet et al., 2009). Nos últimos anos, com a introdução de tecnologias como o ChatGPT, que exibe uma notável capacidade de processamento e geração de linguagem natural, uma nova onda de pesquisas emergiu. Pesquisadores de diversas áreas começaram a explorar a eficácia dessas novas ferramentas, especialmente no campo da Engenharia de Software (Ságodi et al., 2024; Al Madi, 2022; Xueying et al., 2024; Nunes et al., 2025). Estudos têm investigado como ferramentas como LLMs podem melhorar a produtividade no desenvolvimento de software, além de analisar o impacto dessas sugestões automatizadas na qualidade do código em diferentes linguagens de programação.

3.1 Uso de LLMs no Apoio ao Desenvolvimento de Software

O primeiro estudo analisado apresenta uma revisão sistemática da literatura sobre o uso de LLMs na engenharia de software. Hou et al. (2024) examinaram 395 trabalhos publicados entre 2017 e 2024, abrangendo tarefas como geração e análise de código, documentação automática, busca de informações e detecção de bugs. Foram considerados diferentes tipos de dados do desenvolvimento de software, como postagens em fóruns, registros de bugs e patches. O artigo também discute técnicas de processamento, otimização e avaliação dos modelos, além de desafios e perspectivas futuras, ressaltando o potencial transformador dos LLMs na área.

Já a pesquisa quantitativa de Coello e Kouatly (2024) avaliou a eficácia dos modelos GPT da OpenAI (GPT-3.5 e GPT-4) em tarefas de programação, comparando-os com ferramentas como Bard e Claude. O GPT-4 obteve a melhor taxa de sucesso (87,51%), seguido do GPT-3.5 (83,18%) e do Bing (81,96%), enquanto Bard e Claude registraram 76,16% e 71,43%, respectivamente. Esses resultados indicam que os LLMs apresentam desempenho elevado na geração de código e na resolução de problemas de programação, oferecendo suporte relevante ao desenvolvimento de software.

3.2 LLMs Aplicados à Refatoração de Código

Liu et al. (2024) investigaram o uso de GPT-4 e Gemini na refatoração de código Java com 180 exemplos reais. O GPT-4 identificou oportunidades de refatoração em 15,6% dos casos, contra 3,9% do Gemini; ao explicitar o tipo de refatoração no prompt, os sucessos aumentaram para 52,2% e 21,1%, respectivamente. Das sugestões geradas, 63,6% do GPT-4 e 56,2% do Gemini foram avaliadas como comparáveis ou superiores às humanas, embora cerca de 7% apresentassem problemas como erros de sintaxe ou alteração de comportamento. Os autores concluem que LLMs podem ser eficazes na refatoração quando bem orientados, mas recomendam supervisão humana e técnicas de segurança.

3.3 Avaliação da Qualidade do Código Refatorado

Siddiq et al. (2024) investigaram a qualidade do código gerado pelo ChatGPT a partir do conjunto de dados DevGPT, analisando aspectos como qualidade, segurança e aplicabilidade em ambientes de desenvolvimento. Os resultados evidenciam problemas recorrentes, como nomes de variáveis pouco claros, documentação insuficiente, não conformidade com convenções e vulnerabilidades de segurança. Entre os riscos identificados estão chamadas de API sem tratamento adequado, uso de credenciais no código e implementações frágeis de proteção, totalizando 84 ocorrências. Apesar dessas limitações, o estudo mostra que desenvolvedores utilizam o ChatGPT em ta-

refas como formatação, refatoração, criação de testes, depuração e aprendizado de frameworks e bibliotecas, ressaltando seu potencial como ferramenta de apoio, desde que acompanhado de avaliação crítica.

De forma complementar, Yetistiren et al. (2022) avaliaram empiricamente a geração de código pelo GitHub Copilot, considerando correção, validade, eficiência e segurança. O modelo alcançou 91,5% de sucesso em código válido, com 28,7% das soluções corretas e 51,2% parcialmente corretas, apresentando desempenho próximo ao de programadores humanos. Contudo, os autores identificaram limitações na resolução de problemas complexos e na dependência da interação humana, destacando a necessidade de investigações adicionais que incluam métricas de manutenibilidade e confiabilidade. O estudo reforça o potencial do Copilot, mas também a importância de reconhecer suas restrições.

3.4 Lacunas Identificadas

Após uma revisão abrangente sobre LLMs e suas capacidades, identificamos diversos estudos que avaliam sua eficácia na geração de código em várias linguagens, tanto em nível de método quanto de classe (Xueying et al., 2024; Shirafuji et al., 2023; Chen et al., 2021). Embora os resultados variem, há consenso sobre a necessidade de pesquisas contínuas para acompanhar a evolução dessas IAs. No entanto, poucos trabalhos focam especificamente na eficácia das LLMs na refatoração de código, especialmente em Python (Liu et al., 2024; AlOmar et al., 2024; Shirafuji et al., 2023). A maior parte das pesquisas concentra-se na geração de código funcional a partir de descrições simples, enquanto investigações sobre refatoração em projetos reais, que envolvem melhorias de legibilidade e manutenibilidade, ainda são escassas (Liu et al., 2024; Nunes et al., 2025).

Os estudos mostram que modelos como GPT-4, Codex e Gemini têm bom desempenho com instruções claras, mas sua aplicação na refatoração enfrenta limitações. Além disso, a maioria das pesquisas utiliza exemplos sintéticos, pouco representativos dos desafios reais relacionados a más práticas e baixa legibilidade em códigos de produção.

3.5 Considerações Finais

Este capítulo revisou trabalhos sobre o uso de LLMs no desenvolvimento e refatoração de código. Observou-se que, embora eficazes na geração de código, essas ferramentas ainda enfrentam desafios em projetos reais e complexos, dependendo de prompts bem formulados e de supervisão humana para garantir segurança e correção. Grande parte da literatura foca em exemplos sintéticos ou linguagens específicas, deixando lacunas quanto à refatoração em cenários de produção, especialmente em Python e na manutenção da manutenibilidade do código.

Futuras pesquisas devem avaliar a eficácia das LLMs em refatorações reais, desenvolver métricas de qualidade e explorar abordagens híbridas que combinem aprendizado estatístico e raciocínio simbólico, ampliando a confiabilidade e o potencial dessas ferramentas na engenharia de software.

No próximo capítulo, apresentamos a metodologia adotada neste estudo, detalhando os procedimentos e etapas do experimento.

Capítulo 4

Metodologia

Neste capítulo, apresentamos a metodologia para avaliar a eficácia de LLMs na refatoração de código Python, focando em problemas de manutenibilidade. Utilizamos o SonarQube[1], uma ferramenta de análise estática amplamente empregada, que identifica defeitos impactando a qualidade e evolução do código (Campbell e Papapetrou, 2013; Lenarduzzi et al., 2020).

- A Seção 4.1 descreve como as questões de pesquisa foram abordadas;
- A Seção 4.2 detalha cada etapa da metodologia adotada;
- A Seção 4.3 apresenta as considerações finais do capítulo.

Selecionamos, de forma exploratória e aleatória, 12 regras específicas do SonarQube focadas em violações que impactam diretamente a manutenibilidade do código, pois esses problemas comprometem sua clareza e facilidade de manutenção. Os LLMs avaliados foram: *Copilot Chat (baseado no GPT-4o), LLaMA 3.3 70B Instruct, DeepSeek V3 e Gemini 2.5 Pro.* Cada modelo foi solicitado a sugerir correções para os problemas identificados, inicialmente utilizando a abordagem *zero-shot* e, em seguida, a abordagem *few-shot*.

Para medir o desempenho, foram considerados três aspectos principais: a taxa de sucesso na correção e refatoração de métodos com problemas de manutenibilidade, as limitações observadas nos LLMs e o impacto das refatorações na legibilidade.

4.1 Objetivos e Questões de Pequisa

A refatoração automatizada por LLMs é uma área emergente, mas ainda pouco explorada, especialmente no que diz respeito à manutenibilidade de código Python. Esta seção define quatro questões de pesquisa **(RQs)** que buscam avaliar a eficácia, limitações e percepções desses modelos em cenários práticos em projetos hospedado do Github.

[1] `https://www.sonarsource.com/`

1. **RQ1. Qual é a eficácia dos LLMs na correção de problemas de manutenibilidade em código Python?** Para responder a essa questão, avaliamos a eficácia dos LLMs na correção de problemas de manutenibilidade em métodos Python identificados pelo SonarQube. Após as refatorações sugeridas, cada método foi analisado quanto à resolução do problema original, preservação da funcionalidade e surgimento de novos problemas. Essa abordagem permitiu comparar os modelos e avaliar seu impacto na manutenibilidade do código.
2. **RQ2. Quais tipos de erros os LLMs cometem ao refatorar códigos Python com problemas de manutenibilidade?**

   Analisamos os erros mais comuns cometidos pelos LLMs durante a refatoração, verificando se introduziram novos problemas de manutenibilidade, como variáveis mal nomeadas, código morto, métodos complexos ou violações de estilo. Também avaliamos se causaram erros de sintaxe, falhas de execução ou problemas nos testes, além de identificar casos em que o problema original permaneceu sem solução. Essa análise permitiu que identificássemos as limitações dos LLMs na refatoração de código Python com problemas de manutenibilidade.
3. **RQ3. Como diferentes técnicas de prompting podem melhorar o desempenho dos LLMs na refatoração de código?** Para responder a esta questão de pesquisa, testamos duas estratégias de prompts nos LLMs. Primeiro, com abordagem *zero-shot*, solicitando correções sem exemplos prévios, e depois a *few-shot*, fornecendo exemplos de código com problemas e soluções (Brown et al., 2020). Aplicamos ambas os mesmos métodos e comparamos os resultados quanto à resolução do problema, preservação da funcionalidade, ausência de novos erros e desempenho nos testes, avaliando o impacto das técnicas de prompting na refatoração. Essa análise permitiu avaliar de forma prática e comparativa o impacto das técnicas de prompting nas refatorações realizadas pelos LLMs.
4. **RQ4. Em que medida os desenvolvedores consideram as soluções geradas pelas LLMs mais legíveis e compreensíveis?** Para responder a essa questão, realizamos uma avaliação com participação humana, com o objetivo de analisar a percepção dos desenvolvedores sobre a legibilidade dos códigos gerados pelas LLMs após o processo de refatoração. Para isso, apresentamos aos participantes pares de código: um gerado pela IA e outro escrito por um desenvolvedor humano. Com essa abordagem, buscamos investigar se o código gerado pelas IAs é percebido como mais legível, menos legível ou igualmente legível em comparação ao código escrito por desenvolvedores humanos.

4.2 Etapas da Metologia de Pesquisa

Este experimento foi conduzido em oito etapas bem definidas, que apresentamos a seguir e explicamos em detalhes na seção de Metodologia.

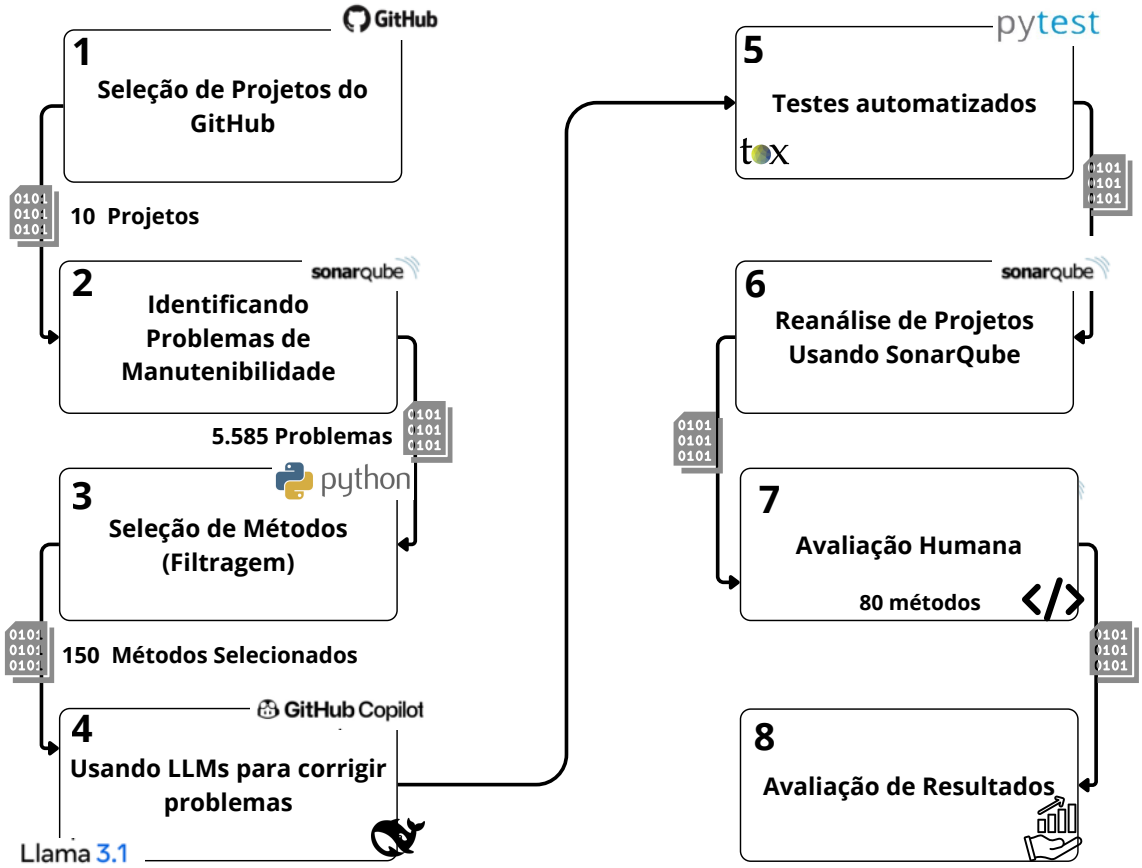

Figura 4.1: Etapas da Metologia

Na Figura 4.1, apresentamos as etapas adotadas na metodologia deste estudo. Cada uma dessas etapas será detalhada ao longo desta seção.

4.2.1 Etapa 1: Seleção de Projetos do Github

Nesta etapa, coletamos e selecionamos projetos seguindo critérios específicos, visando uma amostra diversificada em complexidade (baixa, média e alta) e tipo, abrangendo áreas como comunicação, APIs, frameworks web, ORMs, scraping, machine learning, IA e DevOps. Essa seleção garante uma análise representativa de aplicações em Python.

Escolhemos Python por sua ampla adoção (TIOBE Software, 2024), variedade de bibliotecas (Beazley e Jone, 2013) e relevância em IA e aprendizado de máquina (Chollet, 2021). Além disso, poucos estudos investigam a refatoração de projetos reais em Python com LLMs, especialmente em manutenibilidade, representando uma oportunidade para avaliar sugestões e implementações automatizadas na melhoria da qualidade do código.

4.2.1.1 Critérios de Seleção de Projetos

Os projetos selecionados para o estudo seguiram critérios específicos para garantir relevância, consistência e qualidade. **Recentes:** foram escolhidos apenas projetos com atualizações até, no mínimo, 01/01/2024, garantindo que estivessem ativos e em desenvolvimento contínuo. **Popularidade:** adotamos um mínimo de 25.000 estrelas no GitHub, definido empiricamente, para assegurar a escolha de projetos amplamente utilizados e reconhecidos pela comunidade. **Comunidade ativa:** exigimos pelo menos 200 colaboradores ativos, evitando projetos com equipes pequenas ou pouco engajadas, uma vez que comunidades maiores tendem a manter projetos mais robustos e frequentemente atualizados. **Projeto com mínimo de 75% do código em Python:** pelo menos 75% do código precisava ser escrito em Python, evitando repositórios híbridos e garantindo consistência com o foco do estudo (Nunes et al., 2025). **Projetos amplamente testados:** selecionamos apenas projetos com cobertura de testes igual ou superior a 75%, assegurando uma base de código estável e adequada para avaliar o impacto das refatorações.

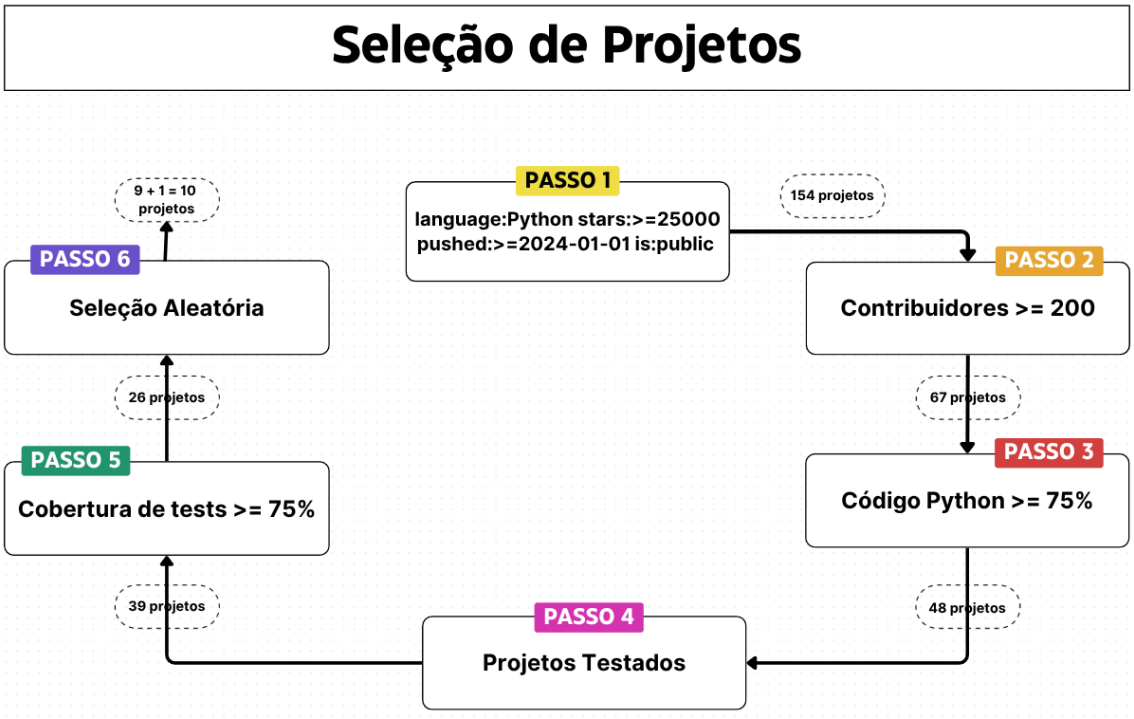

Figura 4.2: Etapas para seleção dos repositórios

A Figura 4.2 ilustra o processo de seleção dos projetos, com base nos critérios de inclusão e exclusão. Inicialmente, criamos uma string de busca, que continha os seguintes parâmetros:

1. **Critérios de Inclusão**

Primeiro passo: O primeiro passo consistiu na realização de uma busca no GitHub, utilizando a seguinte query baseada nos critérios de seleção 1 e 2 (Recente e Popularidade) considerando repositórios públucos: **language:Python stars:>=25000 pushed:>=2024-01-01 is:public**, a qual retornou um total de 154 projetos.

2. **Critérios de Exclusão**

   Segundo passo: Excluímos projetos com menos de 200 contribuidores, reduzindo o conjunto para 66 projetos. **Terceiro passo:** Aplicamos um filtro com base na linguagem predominante, mantendo apenas os projetos com pelo menos 75% do código-fonte escrito em Python, resultando em 48 projetos. **Quarto passo:** Verificamos a presença de diretórios denominados *tests*, *test* ou *testing*, como indício de testes automatizados, e após essa verificação restaram 39 projetos. **Quinto passo:** Aplicamos o filtro de cobertura de testes, mantendo apenas os projetos com cobertura igual ou superior a 75%; projetos que apresentaram falhas de execução ou problemas de dependência não resolvidos foram descartados, restando 26 projetos. **Sexto passo:** Selecionamos aleatoriamente 9 desses projetos e conseguimos executar com sucesso seus testes automatizados, compondo a amostra final utilizada no estudo. **Projeto adicional:** Incluímos também o projeto SQLAlchemy, previamente utilizado na fase piloto, completando um total de 10 projetos na amostra final.

   Execução dos testes: Para garantir a qualidade dos projetos, seguimos as instruções do arquivo `README.md` de cada repositório, instalando as dependências necessárias. A maioria dos testes foi executada com `Pytest`[2], e em projetos como Django utilizamos `Tox`[3] para testar múltiplas versões da linguagem.

   Justificativa Adicional: Embora um dos projetos (Sqlalchemy) selecionados não atenda a um dos critérios estabelecidos, especificamente o número mínimo de estrelas, ele foi incluído devido ao seu uso anterior no projeto piloto do estudo. Essa decisão foi tomada devido à relevância do projeto para a pesquisa e à sua aplicabilidade no contexto do experimento. A escolha foi documentada e justificada com base no impacto positivo que o projeto trouxe na fase piloto.

A Tabela 4.1 apresenta os projetos selecionados com suas principais características e o resultado dos testes de cobertura. A primeira coluna exibe o nome e a versão de cada projeto, a segunda mostra a quantidade de estrelas, seguida pela terceira coluna que indica o número de colaboradores. A última coluna apresenta a cobertura de testes de cada projeto.

[2] `https://docs.pytest.org/en/stable/`
[3] `https://tox.wiki/en/4.27.0/`

Tabela 4.1: Projetos Selecionados

Projeto	Estrelas	Colaboradores	Cob. de Testes	LOC	Categoria
Django v5.1.2	79100	1700	79%	119485	Framework
Django-Rest v3.15.1	25400	1261	96%	22533	Framework
FastAPI v0.113.0	75360	752	98%	29401	Framework
Mitmproxy v10.4.2	35860	486	94%	76607	Com. Rede
Pandas v2.2.3	43200	3152	82%	124359	Machine L. I.A
Poetry v1.8.3	31000	582	95%	15092	Ferr. DevOps
Requests v2.32.0	52000	646	91%	3478	Com. Redes
Rich v0.13.8.1	48800	257	94%	74358	Ferr. Desenvolvimento
Scrapy v2.11.2	52300	560	81%	191201	Ferr. Scraping
SQLAlchemy v2.0.35	9400	631	75%	271684	Framework

Após a seleção dos projetos, realizamos uma análise quantitativa segmentando-os em quartis com base em quatro métricas: número de estrelas no GitHub, quantidade de colaboradores, cobertura de testes e tamanho do projeto. A divisão em quartis é uma técnica estatística descritiva útil para compreender a variabilidade dos dados e construir amostras mais representativas (Larson e Farber, 2015; Morettin e de Oliveira Bussab, 2010; Bracarense, 2010). Essa abordagem permitiu analisar a diversidade e a distribuição das características dos projetos, auxiliando na composição de uma amostra equilibrada e representativa.

A Tabela 4.2 apresenta a classificação dos 10 projetos analisados com base nos quartis de quatro métricas: número de estrelas no GitHub, número de colaboradores, cobertura de testes (%) e linhas de código (LOC). Essa categorização permite uma visão mais precisa da distribuição das características dos projetos, possibilitando comparações mais equilibradas entre eles.

Tabela 4.2: Classificação dos Projetos por Quartis das Métricas

Projeto	Estrelas	Q Estrela	Colaboradores	Q Colab.	Cob. Testes (%)	Q Cob.	LOC	Q LOC
Django v5.1.2	79100	**Q4**	1700	**Q4**	75	**Q1**	119485	**Q3**
Django-Rest v3.15.1	25400	**Q1**	1261	**Q3**	96	**Q4**	22533	**Q2**
FastAPI v0.113.0	75360	**Q4**	752	**Q3**	98	**Q4**	29401	**Q2**
Mitmproxy v10.4.2	35860	**Q2**	486	**Q1**	94	**Q3**	76607	**Q3**
Pandas v2.2.3	43200	**Q2**	3152	**Q4**	82	**Q2**	124359	**Q3**
Poetry v1.8.3	31000	**Q2**	582	**Q2**	95	**Q3**	15092	**Q1**
Requests v2.32.0	52000	**Q3**	646	**Q3**	91	**Q2**	3478	**Q1**
Rich v0.13.8.1	48800	**Q3**	257	**Q1**	94	**Q3**	74358	**Q2**
Scrapy v2.11.2	52300	**Q3**	560	**Q2**	81	**Q2**	191201	**Q4**
SQLAlchemy v2.0.35	9400	**Q1**	631	**Q2**	79	**Q1**	271684	**Q4**

Legenda:

- **Estrelas:** Q1 $\leq$ 29.600, Q2 29.601–46.000, Q3 46.001–58.065, Q4 $>$ 58.065
- **Colaboradores:** Q1 $\leq$ 541, Q2 541.51–638.5, Q3 638.51–1.370,75, Q4 $>$ 1.370,75
- **Cobertura:** Q1 $\leq$ 80,5%, Q2 80,51–92,5%, Q3 92,51–95,25%, Q4 $>$ 95,25%
- **LOC:** Q1 $\leq$ 20.672,75, Q2 20.672,76–75.482,5, Q3 75.482,51–141.069,5, Q4 $>$ 141.069,5

No que diz respeito à **quantidade de estrelas no GitHub**, os intervalos foram: até 29.600 (**Q1**), de 29.601 a 46.000 (**Q2**), de 46.001 a 58.065 (**Q3**) e acima de 58.065 (**Q4**). Para o **número de colaboradores**, os intervalos foram: até 541 (**Q1**), de 541,51 a 638,5 (**Q2**), de 638,51 a 1.370,75 (**Q3**) e acima de 1.370,75 (**Q4**). Para a **cobertura de testes (%)** os limites foram: até 80,5% (**Q1**), de 80,51% a 92,5% (**Q2**), de 92,51%

a 95,25% (**Q3**) e acima de 95,25% (**Q4**). Quanto às **linhas de código (LOC)**, os projetos foram classificados da seguinte forma: até 20.672,75 (**Q1**), de 20.672,76 a 75.482,5 (**Q2**), de 75.482,51 a 141.069,5 (**Q3**) e acima de 141.069,5 (**Q4**).

Cada projeto foi classificado em quartis para cada métrica, permitindo analisar melhor a distribuição das características da amostra. A avaliação mostrou diversidade em popularidade, participação comunitária, cobertura de testes e tamanho, garantindo representatividade e evitando concentração em projetos extremos. Priorizou-se projetos com maior cobertura de testes em cada faixa para aumentar a robustez dos experimentos.

Os limites de cada quartil foram definidos com base nos valores obtidos a partir da ordenação crescente das métricas e cálculo por interpolação, veja o cálculo no Apêndice A. Os intervalos considerados foram os seguintes:

4.2.2 Etapa 2: Identificação dos Problemas de Manutenibilidade

Nesta etapa, utilizamos o SonarQube para analisar os códigos dos projetos e identificar problemas de manutenibilidade. Para isso, instalamos a ferramenta na versão Community Edition 10.5.1, utilizando o banco de dados PostgreSQL.

Ao final da análise, o SonarQube apontou 47 regras específicas relacionadas à manutenibilidade. Muitas dessas regras são comuns a diferentes linguagens de programação, pois seguem boas práticas gerais de desenvolvimento. Essa etapa foi essencial para localizar os métodos que apresentaram problemas, servindo como base confiável para o processo de refatoração nas próximas fases do estudo.

A tabela no Apêndice B apresenta as 47 regras de manutenibilidade do SonarQube. Na primeira coluna, são exibidos os nomes das regras, enquanto na segunda, são mostrados os valores das chaves correspondentes.

4.2.2.1 Análise dos códigos do projetos pelo SonarQube

Após instalar a ferramenta, iniciamos a avaliação dos projetos para identificar problemas de manutenibilidade. A Tabela 4.3 apresenta a quantidade de problemas de manutenibilidade encontrados em cada projeto, considerando apenas o código em Python. Para focar na análise do código principal, aplicamos o SonarQube[4] exclusivamente neste segmento dos projetos, excluindo diretórios e arquivos de teste.

[4] `https://www.sonarqube.org`

Tabela 4.3: Quantidade de Problema por Projetos

Projeto	Quant. de Problemas
Django	1.048
Django-Rest	149
Fastapi	155
Mitmproxy	616
Requests	36
Rich	140
Scrapy	240
Sqlalchemy	1.631
Pandas	1.462
Poetry	108
Total	5.585

A Figura 4.3 refere-se à tela de criação de um projeto no SonarQube. O código exibido na parte inferior da imagem deve ser copiado e executado no diretório raiz do projeto. Dessa forma, o SonarQube realiza a análise estática do código, identificando os problemas existentes.

Figura 4.3: Tela de Criação de Projeto no SonarQube

Desenvolvemos um script, disponível no GitHub[5], para automatizar a execução de testes e a análise com o SonarQube. Ele roda o `pytest` ou `tox` e copia o projeto para uma máquina virtual com o SonarQube. Primeiro, são executados os testes automatizados; se falhas interrompem o processo, os resultados são registradas em log. Se os testes passam, o projeto é analisado pelo SonarQube, que gera um relatório com as issues classificadas por severidade (alta, média e baixa). Esse fluxo agiliza e padroniza o processo, mantendo a validação manual e garantindo registros consistentes.

A Figura 4.4 apresenta a tela com os resultados da análise realizada pelo SonarQube do projeto Request. Na primeira coluna, intitulada *Security*, são exibidos os resultados relacionados a problemas de segurança. A segunda coluna, *Reliability*, apresenta os problemas de confiabilidade do código, enquanto a terceira coluna, *Maintainability*, exibe os resultados relativos a problemas de manutenibilidade, que envolvem

[5] `https://github.com/pedrocarneiroipira/experiment-artifacts-msc`

aspectos como legibilidade, modularidade e facilidade de manutenção do código, sendo este o foco principal da nossa pesquisa.

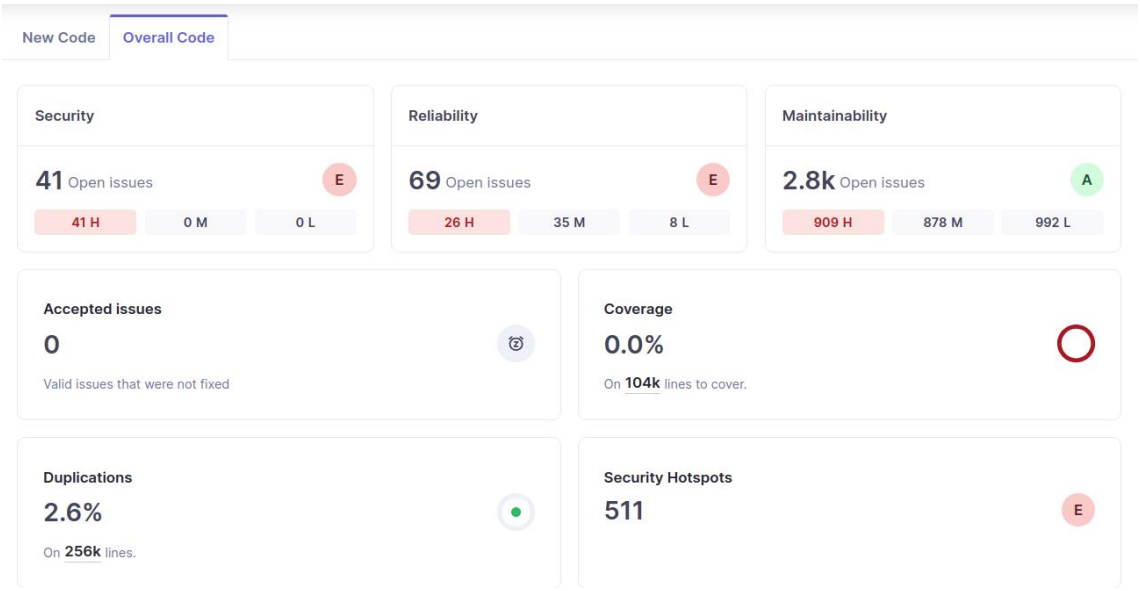

Figura 4.4: Tela dos Resultados da Análise Estática

A Figura 4.5 apresenta um exemplo de problema de manutenibilidade, com dados como chave da regra, arquivo e linha. Quando o método não é exibido diretamente, ele é identificado pelo caminho do arquivo e pela linha correspondente. No caso ilustrado, a regra `S117` foi violada por uso incorreto de nomenclatura de variáveis. A recomendação é adotar o padrão *snake_case*, garantindo clareza e consistência.

Figura 4.5: Tela de Problema de Manutenibilidade

As Tabelas 4.4 e 4.5 ilustram a planilha quem usamos para registrar as informações utilizadas no experimento. A Tabela 4.4 contém dados básicos do problema, como ID, chave e regra associada, localização no código, assinatura do método e o prompt enviado ao LLM. Já a Tabela 4.5 complementa com a análise da refatoração, incluindo severidade (quando bem-sucedida), técnica aplicada, observações, resultado (testes e reanálise) e desfecho final (resolvido, novos problemas ou falhas).

Informações adicionais, como código original e refatorado, feedbacks dos LLMs, registros de logs e erros durante a execução, estão disponíveis em *experiment-artifacts-msc* .

Tabela 4.4: Planilha de Registros de Dados - Parte 1

ID	Regra	Caminho	Linha	Método	Prompt (Zero-Shot)
1	S1481 (Unused local variables should be removed)	`src/poetry/repositories/link_sources base.py`	80	`def link_package_data (cls, link: Link) -> Package | None:`	In this class, the method '`def linkz_packagez_data(cls, link: Link) -> Package | None:`' has the following issue 'Unused local variables should be removed'. Can you identify and fix it?

Tabela 4.5: Planilha de Registros de Dados - Parte 2

Impacto	Técnica de Refatoração	Observação	Categorização	Issue Resolvida?
Low	Renomeação de variável não utilizada por _	Substituiu a variável `ext` não utilizada por _, seguindo boas práticas de programação em Python e PEP8	Resolveu	SIM

4.2.3 Etapa 3: Filtragem dos Métodos com Problemas de Manutenibilidade

Na Etapa 3 do experimento, filtramos os métodos com problemas de manutenibilidade que seriam analisados e aplicados aos LLMs para obter sugestões de refatoração e correção. Estabelecemos critérios específicos para garantir a relevância e consistência dos dados.

Selecionamos, 12 regras do conjunto de 47 definidas na linguagem Python no SonarQube, sendo 8 compartilhadas com outras linguagens, como Java e JavaScript, o que torna o estudo potencialmente replicável em diferentes contextos. Ver regras na tabela no Apêndice C.1.

A escolha das regras seguiu uma abordagem exploratória, focada em investigar como os LLMs sugerem refatorações para um subconjunto representativo de problemas de manutenibilidade. Isso permitiu avaliar o potencial das inteligências artificiais em um cenário prático, com ênfase em regras comuns a várias linguagens, aumentando a aplicabilidade dos resultados e servindo como base para estudos futuros.

Para evitar vieses na seleção das regras, projetos e problemas, adotamos critérios rigorosos e previamente definidos, garantindo imparcialidade e confiabilidade na coleta e análise dos dados.

4.2.3.1 Critérios

Coletamos um total de 5.585 problemas de manutenibilidade nos códigos-fonte principais dos projetos, excluindo trechos de código de teste. Para garantir a viabilidade do estudo e a representatividade da amostra, estabelecemos os seguintes critérios:

Os critérios adotados para a seleção dos problemas de manutenibilidade buscaram garantir **imparcialidade**, **testabilidade** e **representatividade** da amostra. Escolhemos os problemas de forma **aleatória** a partir dos dados exportados do SonarQube, assegurando uma amostra livre de vieses. Para isso, os dados foram importados em formato `.csv`, e utilizamos a função `RAND()` do Google Sheets na seleção aleatória de problemas, projetos e das doze regras analisadas. Buscamos assegurar a **testabilidade automática** mediante a verificação do funcionamento dos testes associados aos métodos, com a inserção de falhas controladas no código; apenas os métodos que se mostraram testáveis e consistentes foram mantidos, reforçando a confiabilidade dos resultados.

Definimos o **tamanho da amostra** buscando um nível de confiança de 90% e margem de erro de aproximadamente 7%, parâmetros adequados a pesquisas exploratórias que visam identificar tendências gerais no uso de LLMs para refatoração de código Bracarense (2010); Gil (2008). O cálculo amostral, ajustado para população finita, indicou 135 métodos, número arredondado para 150 a fim de facilitar a distribuição entre os projetos (detalhes no Apêndice D).

Para evitar distorções, estabeleceu-se um **limite por projeto**, definindo que cada regra poderia ocorrer, no máximo, cinco vezes em um mesmo projeto, promovendo diversidade e equilíbrio entre os tipos de problemas analisados. Além disso, buscou-se garantir **variabilidade entre projetos**, de modo que cada regra estivesse presente em pelo menos quatro projetos distintos, o que ampliou o alcance da análise e permitiu avaliar o desempenho dos modelos em contextos variados de desenvolvimento. Por fim, determinou-se a **diversidade de regras por projeto**, exigindo que cada projeto incluísse, no mínimo, cinco regras diferentes associadas a problemas de manutenibilidade. Esse critério assegurou maior abrangência e reduziu o risco de concentração em tipos específicos de violações, contribuindo para uma avaliação mais ampla e equilibrada dos resultados.

4.2.3.2 Passos para Seleção de Problemas de Manutenibilidade

O processo de seleção e preparação das violações de manutenibilidade seguiu seis passos principais. No **Passo 1 – Análise Inicial no SonarQube**, iniciamos a análise dos 10 projetos selecionados com o SonarQube, visando identificar violações de manutenibilidade nos códigos principais (excluindo arquivos de teste), totalizando

5.585 problemas (Tabela 4.3). No **Passo 2 – Exportação dos Dados**, as violações detectadas foram exportadas para arquivos `.csv`, facilitando a manipulação e aplicação dos critérios de filtragem. Em seguida, no **Passo 3 – Seleção Aleatória**, os problemas foram embaralhados com a função `RAND()` do Google Sheets, garantindo uma escolha aleatória e reduzindo o viés humano na amostragem. No **Passo 4 – Verificação de Testabilidade**, para cada violação selecionada, testamos se o método correspondente era coberto por testes automatizados, inserindo falhas controladas no código e executando os testes; apenas métodos efetivamente testáveis foram mantidos na amostra. O **Passo 5 – Controle de Diversidade** consistiu na aplicação de três critérios para assegurar distribuição equilibrada das violações: (i) cada regra deveria ocorrer em pelo menos 4 projetos diferentes; (ii) cada regra poderia ter, no máximo, 5 ocorrências por projeto; e (iii) cada projeto deveria conter pelo menos 5 regras distintas. Esses critérios promoveram variedade de contextos e tipos de problemas, evitando concentração excessiva em projetos ou regras específicas. Por fim, no **Passo 6 – Consolidação Final**, as violações que atenderam a todos os requisitos foram selecionadas para a etapa experimental com os modelos de linguagem, enquanto as demais foram descartadas.

Com a aplicação dos critérios descritos acima e seguindo as etapas detalhadas anteriormente, selecionamos 150 problemas de manutenibilidade considerados adequados para análise pelas LLMs. A Tabela 4.6 apresenta a distribuição das regras e suas recorrências nos projetos. A primeira coluna exibe o acrônimo das regras, a segunda apresenta o nome da regra e a última indica em quantos projetos cada regra ocorre.

Tabela 4.6: Distribuição de Problemas Identificados por Regra e Projetos

Sigla	Regras	Problemas	Recorrência
CCM	Cognitive Complexity of functions should not be too high	33	10
FMP	Functions, methods and lambdas should not have too many parameters	11	5
FNC	Function names should comply with a naming convention	8	4
BCI	Boolean checks should not be inverted	7	4
SLD	String literals should not be duplicated	13	7
LVN	Local variable and function parameter names should comply with a naming convention	7	6
UFP	Unused function parameters should be removed	12	8
MIS	Mergeable ”if”statements should be combined	15	8
ULV	Unused local variables should be removed	14	8
SES	’startswith’ or ’endswith’ methods should be used instead of string slicing in condition expressions	5	5
BSV	Builtins should not be shadowed by local variables	11	5
TUT	Track uses of ”TODO”tags	14	7
	Total	150	

4.2.3.3 Descrição e Relevância das Regras de Manutenibilidade Selecionadas

A ferramenta de análise estática utilizada foi o SonarQube, de onde foram extraídas 12 regras selecionadas aleatoriamente conforme os critérios descritos. Essas regras, resultantes do processo de seleção dos métodos, mostraram-se representativas do

subconjunto relevante ao estudo e são frequentemente violadas em outras linguagens, possibilitando replicações. Entre elas, destacam-se nomenclatura inadequada (S117), alta complexidade cognitiva (S3776), variável não utilizada (S1481) e strings duplicadas (S1192) — problemas reconhecidos na literatura por afetarem a compreensibilidade e a manutenção do software (Fowler, 2019; Martin, 2008). Assim, as regras escolhidas refletem tanto práticas das ferramentas de análise quanto princípios consolidados de qualidade e manutenibilidade.

A Tabela 4.7 apresenta o número de métodos com problemas de manutenibilidade por projeto e por regra violada. A partir dessa tabela, é possível verificar que os critérios metodológicos definidos nesta subseção foram respeitados, especialmente no que se refere à seleção aleatória das violações e à limitação da quantidade de ocorrências por regra em cada projeto, garantindo a variabilidade e o equilíbrio da amostra.

Tabela 4.7: Distribuição dos Problemas de Manutenibilidade por Projeto e Regra

PROJETO	CCM	FMP	FNC	BCI	SLD	LVN	UFP	MIS	ULV	SES	BSV	TUT	**Total**
Django	3	1	0	0	2	0	1	2	1	1	3	1	**15**
Django-Rest	3	0	0	0	1	1	2	1	4	1	2	0	**15**
FastApi	3	3	4	2	0	1	0	0	0	0	0	2	**15**
Mitmproxy	3	0	1	2	3	1	1	1	1	1	1	0	**15**
Requests	4	1	0	0	0	1	1	2	3	1	0	2	**15**
Rich	3	4	0	0	2	0	0	2	1	0	2	1	**15**
Scrapy	3	0	1	1	0	2	4	2	0	1	0	1	**15**
Sqlalchemy	4	0	0	0	2	0	1	3	2	0	0	3	**15**
Pandas	3	2	2	2	2	1	0	2	1	0	0	0	**15**
Poetry	4	0	0	0	1	0	2	0	1	0	3	4	**15**
Total	33	11	8	7	13	7	12	15	14	5	11	14	**150**

4.2.4 Etapa 4: Uso de LLMs na Refatoração de Código com Foco em Manutenibilidade

Nesta etapa da metodologia, focamos em sugerir correções para os problemas de manutenibilidade usando duas abordagens de prompting: *zero-shot* e *few-shot*. No *zero-shot*, o modelo recebe apenas instruções, sem exemplos, avaliando sua capacidade de refatorar código a partir das diretrizes fornecidas. Foram testados quatro modelos de linguagem, entre soluções abertas e proprietárias, selecionados por popularidade, facilidade de acesso e desempenho em tarefas de geração e melhoria de código (Al Madi, 2022; Coello e Kouatly, 2024; Yetistiren et al., 2022; Guo et al., 2024).

Essa abordagem permitiu avaliar empiricamente o potencial dos modelos para sugerir correções automáticas de problemas detectados pelo SonarQube. Nos tópicos seguintes, detalhamos a estrutura dos prompts, a escolha dos modelos e o processo de geração das refatorações.

1. **Copilot Chat 4o**[6] **:** Modelo de código fechado baseado no GPT-4o, escolhido por ser amplamente utilizado no mercado, principalmente por desenvolvedores,

[6] `github_copilot_individual`

além de ter sido estudada vários pesquisas acadêmicas voltados para geração e melhoria de código (Yetistiren et al., 2022; Ziegler et al., 2024).

2. **LLaMA 3.3 70B Instruct[7]:** Um modelo de código aberto da Meta. Optamos por incluir esse LLM justamente por representar a comunidade open-source, além de apresentar resultados relevantes em tarefas de linguagem e geração de código (Lu et al., 2023).

3. **DeepSeek V3[8]:** Um modelo lançado recentemente, que chamou atenção durante o desenvolvimento deste estudo pelo seu potencial e por competir diretamente com outros grandes modelos no mercado. Por isso, decidimos testá-lo no contexto de refatoração automática de código em Python (Zhu et al., 2024; Liang et al., 2025).

4. **Gemini 2.5 Pro[9]:** Modelo da Google, que vem evoluindo rapidamente. Escolhemos incluí-lo devido aos resultados promissores em tarefas de linguagem e programação, além da sua crescente popularidade, o que nos permite analisar sua capacidade nesse cenário específico Siam et al. (2024).

4.2.4.1 Comparação entre LLMS:

Realizamos o experimento utilizando quatro diferentes LLMs. Essa abordagem teve como objetivo ampliar a confiabilidade dos resultados, permitindo uma comparação mais abrangente entre os modelos e evidenciando as diferenças de desempenho e eficácia de cada um no processo de refatoração de código.

Ao testar um conjunto maior e mais diversificado de modelos, buscamos trazer diferentes perspectivas e capacidades técnicas para o estudo, o que contribui para aumentar a robustez e a relevância dos resultados obtidos. Além disso, ao incluir múltiplas LLMs, foi possível obter insights valiosos sobre o desempenho específico de cada uma na tarefa de refatoração de código em Python, oferecendo uma visão mais prática e comparativa do potencial dessas ferramentas no contexto da engenharia de software.

4.2.4.2 Diferenças da Forma de Interagir Entre LLMs:

É importante destacar as diferenças entre as LLMs utilizadas, já que os ambientes de execução variaram. No Copilot Chat, integrado ao VS Code, o modelo sugeriu e aplicou a refatoração diretamente, sem copiar e colar, e usamos o SonarLint para validar os problemas antes e após a correção. No LLaMA, acessamos a versão 3.3-70B pela Hugging Face, pois a execução local exigiria recursos computacionais muito altos. Para DeepSeek e Gemini, usamos os chats oficiais das plataformas, copiando o código problemático, fornecendo as instruções de refatoração e registrando as respostas geradas.

[7] `https://www.llama.com/`
[8] `https://www.deepseek.com/`
[9] `https://gemini.google.com/app?hl=pt-BR`

4.2.4.3 Escolha dos prompts:

Definimos dois tipos de prompting[10]: *zero-shot* e *few-shot*. No primeiro, o modelo recebeu apenas instruções sobre a tarefa; no segundo, quatro exemplos de cada regra, permitindo observar problemas e correções antes de sugerir refatorações. O objetivo foi comparar o desempenho dos LLMs e o impacto do uso de exemplos nos resultados.

A escolha de quatro exemplos no *few-shot* baseou-se em referências da literatura. Nunes et al. (2025) utilizaram três exemplos por tipo de regra, mostrando que poucos exemplos já são suficientes para guiar o modelo.

Cada refatoração foi realizada em uma única interação com o LLM; quando o modelo apresentou várias sugestões, apenas a primeira foi utilizada nos testes. Essa estratégia visou padronizar o processo e evitar interferências de múltiplas iterações ou ajustes manuais.

1. **Estrutura do Prompt Zero-Shot:** Criamos uma função no Google Sheets que concatena a primeira string "*In this class, the method*" com a assinatura do método, e a segunda string "*has the following issue*", seguida do nome da regra violada.

   As Tabelas 4.9 e 4.8 ilustram o processo e mostra parte da planilha que usamos para registro: a primeira coluna mostra os dados extraídos da planilha (assinatura do método e regra violada) e a segunda apresenta o prompt final gerado. Esses exemplos evidenciam a sistematização e consistência do processo.

Tabela 4.8: Exemplo do Prompt - Zero-Shot

Função	**Prompt**
In this class, the method + assinatura do método + has the following issue + problema + Can you identify and fix it?	`In this class, the method 'def inserted_primary_key_rows(self):' has the following issue 'String literals should not be duplicated'. Can you identify and fix it?`

Tabela 4.9: Exemplos de Células com Rega e Assinatura do Método

Id	Regra	Método
S1192	String literals should not be duplicated	def inserted_primary_key_rows(self):
S3776	String literals should not be duplicated	def to_internal_value (self, value):

[10]Embora o estudo seja em português, os prompts foram construídos em inglês, idioma padrão das LLMs, o que também resultou em respostas nesse idioma

2. **Estrutura do Prompt Few-Shot:** No few-shot, elaboramos um prompting mais refinado e robusto, incluindo contexto voltado à engenharia de software. Fornecemos exemplos de trechos com problemas de manutenibilidade e suas soluções refatoradas. A Tabela 4.10 apresenta o prompt que foi utilizado na abordagem *few-shot*.

Prompt

You are a software engineer specialized in code refactoring to improve maintainability, readability, and quality. Your goal is to identify and fix maintainability issues in Python code, following best software engineering practices. The main rule to consider is *S1192*, as defined by SonarQube: *"String literals should not be duplicated."* To help you understand the desired refactoring pattern, I will provide examples of code that violate this rule and their respective refactorings. Then, I will provide a piece of code with issues, and you should refactor it following the same principles.

```
Ex1:
# Examples:

# Ex1 - Original Code
def run():
    prepare("action1")
    execute("action1")
    release("action1")

# Ex1 - Refactored Code
ACTION_1 = "action1"

def run():
    prepare(ACTION_1)
    execute(ACTION_1)
    release(ACTION_1)

# Ex2 - Original Code
[ ... ]

# Ex2 - Refactored Code
[ ... ]

# Ex3 - Original Code
[ ... ]

# Ex3 - Refactored Code
[ ... ]

# Ex4 - Original Code
[ ... ]

# Ex4 - Refactored Code
[ ... ]
```

Now that you have seen examples of refactoring, identify and fix the code below to improve its maintainability and readability, keeping the original behavior, without adding new features. Analyze the ***def method(self):*** method, which presents the issue ***'String literals should not be duplicated'***. Your task is to identify and fix this problem by following good programming practices.

Use only the provided code as a reference, without making assumptions or adding nonexistent details.

Tabela 4.10: Example of few-shot prompt used

3. **Impactos da Evolução dos LLMs no Contexto do Experimento:**

   Os LLMs evoluem continuamente, o que pode afetar os resultados ao longo do tempo. A pesquisa durou quase 12 meses e, embora o experimento tenha

sido realizado na reta final, os projetos testados podem ter se beneficiado de versões mais otimizadas dos modelos, apesar serem as mesma no estudo todo. Apesar de não termos observado diferenças significativas, é fundamental registrar as condições do experimento para que futuras replicações considerem essa dinâmica.

A seguir, detalhamos o processo de aplicação dos LLMs na sugestão de refatoração de código, destacando as particularidades de cada ambiente e metodologia utilizada.

4. **Aplicação do Prompt Zero-Shot:**

   (a) **Copilot Chat**
   Copiamos o prompt da planilha de problemas de manutenibilidade, localizamos o arquivo, identificamos o método e a linha do problema, e utilizamos a opção *Fix using Copilot*. Em seguida, aplicamos o prompt para que o Copilot Chat analisasse o código e sugerisse uma solução. Usamos a extensão SonarLint[11] no VS Code para validar a existência do problema e confirmar sua correção.

   (b) **LLaMA 3.3 70B Instruct** Primeiro, identificamos e copiamos o método com problemas de manutenibilidade apontados pelo SonarQube. Em seguida, colamos o código no campo de entrada do LLaMA, na plataforma Hugging Face, e inserimos o prompt solicitando que o modelo identificasse e corrigisse o problema por meio de refatoração. Após receber a resposta da LLM, copiamos o código refatorado sugerido, substituímos o trecho original e realizamos os ajustes finais necessários na formatação, especialmente na identação, para garantir a correta execução do código em Python.

   (c) **DeepSeek V3 e Gemini 2.5 Pro**
   Aplicamos os mesmos passos utilizados com o LLaMA. A única diferença é que, no caso desses dois modelos, utilizamos os ambientes de interação online próprios de cada um: chat.deepseek.com e gemini.google.com, respectivamente.

5. **Aplicação do Prompt Few-Shot:**

   Na abordagem few-shot, o processo foi similar para todos os modelos, exceto pelo Copilot Chat, que interagiu diretamente pelo VS Code. Selecionamos exemplos reais de problemas e soluções para contextualizar os modelos. Em seguida, aplicamos o prompt com contexto e os exemplos reais de métodos com problemas e respectivas soluções (Tabela 4.10) solicitando a correção. A partir daí, seguimos o mesmo fluxo do zero-shot: copiamos o código refatorado, aplicamos a solução e ajustamos o código quando necessário. No caso da abordagem few-shot, o processo foi semelhante para todos os modelos, com

[11] `https://www.sonarsource.com/products/sonarlint/`

uma diferença importante no Copilot Chat, cuja interação ocorreu diretamente por meio do chat integrado no VS Code.

As telas de interação com os LLMs estão apresentadas no Apêndice E.

4.2.5 Etapa 5: Execução de Testes Automatizados

Nesta etapa, realizamos os testes automatizados e a execução das aplicações. O Python[12], é uma linguagem interpretada, e erros de execução foram identificados durante os testes. Nosso objetivo foi garantir que as refatorações sugeridas pelos LLMs não comprometessem a aplicação. Inicialmente, executamos testes unitários para os métodos alterados e, em seguida, testes gerais do projeto. Para criar ambientes isolados, utilizamos Pyenv[13], evitando conflitos entre bibliotecas e versões do Python. Um script automatizado executou os testes e gerou logs que permitiram comparar resultados antes e depois das refatorações.

Também medimos a cobertura de testes (*coverage*), conforme Tabela 4.1, garantindo uma análise completa do impacto das alterações. Eventuais falhas eram registradas para análise posterior[14]. Essa etapa foi muito importante, pois começamos a identificar algumas limitações dos LLMs, como erros de execução, falhas em testes automatizados e até erros de sintaxe no código.

4.2.6 Etapa 6: Reanálise do Código pelo SonarQube

Esta etapa foi fundamental para avaliar a eficácia das sugestões dos LLMs. Após a aprovação dos testes automatizados, reexecutamos o código no SonarQube para verificar se os problemas de manutenibilidade foram resolvidos ou se surgiram novos.

Para cada instância, criamos uma cópia do projeto, garantindo que falhas anteriores não afetassem os resultados. O script desenvolvido na Etapa 5 foi aprimorado para automatizar a reanálise: ele cria uma VM, acessa via SSH, copia o código-fonte, cria um novo projeto no SonarQube e executa a análise automaticamente. Diretórios de teste (`test`, `tests`) foram ignoradas para focar apenas no código principal.

O script gera um log com a quantidade de problemas detectados após a refatoração, que é comparado ao registro inicial para avaliar se o problema original foi **resolvido**, se **novos problemas foram introduzidos** ou se o problema **permaneceu sem solução**. Para maior confiabilidade, os resultados também foram validados com o SonarLint no Visual Studio Code.

Os scripts e arquivos utilizados nesta etapa estão disponíveis no repositório *experiment-artifacts-msc*, que contém todos os artefatos relacionados ao experimento.

[12] `https://docs.python.org/3/tutorial/interpreter.html`
[13] `https://pypi.org/project/pyenv-win/`
[14] experiment-artifacts-msc

4.2.7 Etapa 7: Avaliação Humana

Nesta etapa, realizamos uma análise qualitativa, ou seja, uma avaliação humana dos códigos refatorados pelos LLMs. Esse olhar permitiu identificar aspectos, como clareza, legibilidade, aderência às boas práticas e facilidade de compreensão. Mesmo que o código passe nos testes e não apresente violações no SonarQube, ainda pode conter trechos confusos ou soluções que fogem das práticas recomendadas.

Essa avaliação aproxima o experimento da realidade do mercado, onde desenvolvedores experientes revisam o código considerando manutenibilidade, estilo e legibilidade, trazendo mais robustez e confiabilidade aos resultados.

Optou-se por avaliar apenas os exemplos de código que foram refatorados utilizando a técnica de few-shot prompting, por ter sido a abordagem que apresentou os melhores resultados preliminares.

4.2.7.1 Experimento Piloto

Aplicamos uma pesquisa piloto, na qual dois programadores avaliaram a viabilidade do processo, recebendo dois formulários com cinco pares de código cada. Eles indicaram qual código, A ou B, apresentava melhor legibilidade, se ambos eram legíveis ou se não tinham opinião. Em caso de empate, a decisão ficou a cargo do autor da pesquisa, validando assim a metodologia de avaliação humana. Esse passo foi importante para avaliarmos a viabilidade da percepção humana sobre a legibilidade dos códigos refatorados.

4.2.7.2 Critérios e Metologia Para Avaliação Humana

A seleção da amostra considerou uma população de 371 pares de código gerados e refatorados com *few-shot prompting*. Com nível de confiança de 95% e margem de erro de 10%, a fórmula de amostragem para populações finitas (Morettin e de Oliveira Bussab, 2010; Bracarense, 2010) indicou 77 pares, sendo adotados 80 pares para maior robustez. Ver cálculo em Apêndice F.

Cada par de código foi avaliado por pelo menos dois avaliadores independentes. Foram elaborados 16 formulários, cada um contendo 5 pares de código, apresentados de forma neutra, sem indicar qual versão havia sido refatorada por um LLM. Empates entre versões (original vs. refatorada) foram resolvidos pelos autores do estudo.

Os avaliadores possuíam experiência prévia em leitura e manutenção de código Python, participaram de forma voluntária e ética, e receberam instruções detalhadas sobre os critérios de legibilidade e compreensibilidade. Para garantir imparcialidade e diversidade, selecionaram-se aleatoriamente 20 pares de código de cada projeto, incluindo 5 pares de cada LLM, sendo que a maioria dos métodos possuía até 30 linhas, garantindo diversidade e realismo.

Todos os pares de código estão disponíveis no GitHub, permitindo replicação do estudo.

4.2.7.3 Dados Coletados

Os dados foram coletados por meio de formulários online, elaborados para este estudo. Inicialmente, foram solicitadas informações pessoais e profissionais dos participantes, como nome, e-mail, escolaridade, tempo de experiência em programação e tempo específico de experiência com a linguagem Python. Em seguida, foram apresentados os pares de códigos para avaliação.

A coleta dos dados seguiu as regras determinadas pela Lei Geral de Proteção de Dados Pessoais (Lei 13.709/2018). Todas as informações pessoais dos participantes foram utilizadas exclusivamente para fins acadêmicos e de controle do experimento, sendo garantido o sigilo e a confidencialidade dos dados e o não compartilhamento. Os participantes foram informados sobre os objetivos do estudo e consentiram com a participação de forma livre e esclarecida.

Os resultados da avaliação foram organizados em termos de escolha para cada versão (original/refatorada), e foram analisados por frequência e proporção, conforme descrito no próximo capítulo.

Na avaliação humana, adotamos uma abordagem **mista**:

- **Quantitativa**, por meio da coleta de votos estruturados (código A, código B, ambos ou nenhum);
- **Qualitativa**, por meio da coleta de justificativas abertas fornecidas pelos avaliadores, permitindo a análise de percepções subjetivas sobre clareza, estilo, organização e legibilidade.

Os formulários e respostas dos participantes os quais utilizamos neste estudo encontram-se em experiments-artifacts-msc.

4.2.8 Etapa 8: Análise dos Resultados

Nesta etapa, avaliamos os resultados por meio de um levantamento estatístico simples, com o objetivo de comparar a eficácia dos diferentes LLMs. Para isso, foram calculadas porcentagens em relação ao total de problemas identificados, considerando as seguintes métricas: *taxa de acertos*, proporção de problemas resolvidos corretamente; *taxa de erros de execução*, casos em que as sugestões geraram código inválido; *falhas em testes*, cenários em que as sugestões causaram erros em testes de unidade ou integração; *problemas não resolvidos*, situações em que as falhas permaneceram ou foram tratadas de forma inadequada; *introdução de novos problemas de manutenibilidade*, quando a refatoração degradou o código; *sugestões ausentes*, ausência de resposta mesmo diante de problemas identificados; e *taxas de tipos de erros de execução*, classificação dos diferentes erros encontrados. Além disso, foi realizado um *comparativo entre LLMs*, avaliando o desempenho relativo de cada modelo, bem como um *comparativo de técnicas de prompting*, investigando o impacto das variações no desempenho final.

Aplicamos testes estatísticos, como o qui-quadrado, para comparar a eficácia dos LLMs na correção de problemas de manutenibilidade (Larson e Farber, 2015; Bracarense, 2010). O mesmo teste foi usado para avaliar a influência da técnica de prompting, verificando se o uso de exemplos (few-shot) gera melhora significativa em relação ao zero-shot.

O teste de independência do qui-quadrado foi implementado por meio de um script disponível em experiment-artifacts-msc, que automatizou os cálculos a partir dos dados coletados. Os resultados foram organizados em tabelas e gráficos, evidenciando padrões de desempenho, pontos fortes e limitações dos modelos. Esses achados são discutidos no capítulo seguinte.

4.3 Considerações Finais

A metodologia deste estudo foi planejada para avaliar a capacidade de LLMs em refatorar e corrigir problemas de manutenibilidade em códigos Python. O uso do SonarQube, aliado à seleção criteriosa dos projetos e à definição rigorosa das regras de análise, buscou assegurar resultados seguros e consistentes. Apesar da eficácia das ferramentas, enfrentamos desafios como a complexidade das dependências e o tempo elevado para execução dos testes. Também foi necessário cuidado na aplicação das refatorações, para evitar a introdução de novos problemas.

Além das análises quantitativas, a avaliação humana trouxe uma perspectiva qualitativa importante, permitindo examinar aspectos como legibilidade e aderência a boas práticas, dimensões que muitas vezes escapam às métricas automatizadas. A integração dessas abordagens possibilitou uma avaliação mais completa e realista da eficácia dos modelos.

O próximo capítulo apresenta os resultados comparativos entre os quatro LLMs, considerando tanto a quantidade de problemas resolvidos quanto a qualidade das sugestões geradas.

Capítulo 5

Resultados

Nesta capítulo, apresentamos os resultados de uma análise realizada com 150 métodos Python extraídos de diversos projetos, todos com questões de manutenibilidade. O estudo teve como foco avaliar o desempenho de quatro modelos de linguagem na refatoração automática desses métodos e na correção dessas questões. A seguir, discutimos os principais achados do experimento.

- A Seção 5.1 traz os resultados gerais, incluindo testes estatísticos, comparações entre LLMs e detalha resultados por categorias;
- A Seção 5.1 apresenta as limitações dos LLMs de acordo com os tipos de erros encontrados no experimento ;
- A Seção 5.6 mostra um comparativo das técnicas de prompts bem como ;
- A Seção 5.6 mostra os resultados da avaliação humana, com dados quantitativos e percepções qualitativas;
- A Seção 5.7 apresenta as considerações finais do capítulo.

5.1 RQ1. Eficácia dos LLMs na Correção de Problemas de Manutenibilidade em Código Python

5.1.1 Resultados Gerais

A Tabela 5.1 apresenta as seis categorias que agrupam os resultados obtidos para cada problema analisado neste estudo: *Resolvido, Não Resolvido, Erros de Execução, Erros de Testes, Degradado e Não Sugeriu.* Nas categorias descritas nos itens *Não Resolvido* e *Degradado*, o código refatorado passou com sucesso pelos testes automatizados e pela execução da aplicação. Da mesma forma, isso ocorre na categoria *Resolvido*, onde as refatorações também foram aprovadas nos testes e na execução. Outro ponto a destacar é que, como o Python é uma linguagem interpretada, os

Tabela 5.1: Categorias de Classificação das Refatorações

Categoria	**Descrição**
Resolvido	Problemas totalmente resolvidos pelas LLMs.
Não Resolvido	Refatorações que passaram nos testes, mas não solucionaram o problema de manutenibilidade.
Erros de Execução	Refatorações que geraram falhas de execução, como exceções ou comportamentos inesperados.
Erros de Testes	Refatorações que resultaram em falhas durante a execução dos testes.
Degradado	Refatorações que introduziram novos problemas de manutenibilidade no código.
Não Sugeriu	Casos em que as LLMs não propuseram nenhuma sugestão para o problema identificado.

erros de execução foram detectados durante a aplicação dos testes automatizados. Caso o log apresentasse um erro de execução (e não um erro específico de teste), essas ocorrências foram classificadas como *Erros de Execução*.

A Figura 5.1 apresenta os resultados gerais do estudo, incluindo os dados obtidos com as técnicas *zero-shot* e *few-shot*. Observamos que entre todas as técnicas, as abordagens que utilizam *few-shot* obtiveram um desempenho melhor.

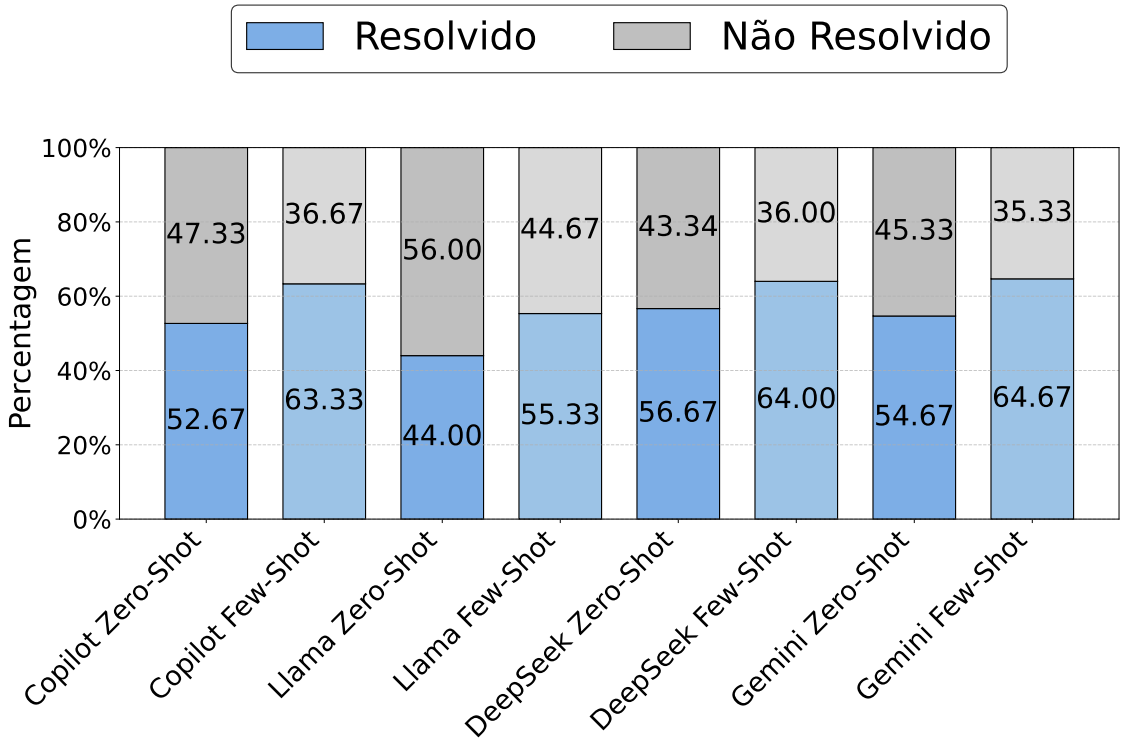

Figura 5.1: Resultados gerais do experimento - Comparativo dos LLMs

Por exemplo, o Copilot Chat, utilizando a técnica *zero-shot*, obteve uma taxa de sucesso de 52,67%, enquanto 47,33% das situações permaneceram sem solução. Com a abordagem *few-shot*, a taxa de acertos aumentou para 63,33%, restando 36,67% de casos não resolvidos. Para o modelo LLaMA, a taxa de acerto foi de 44,00%, enquanto 56,00% dos problemas permaneceram sem solução utilizando a técnica *zero-shot*. Com a técnica *few-shot*, os resultados melhoraram, alcançando 55,33% de acertos e 44,67% de casos não solucionados. Em relação ao modelo DeepSeek, na configuração *zero-shot*, a taxa de sucesso foi de 56,67%, com 43,33% de casos não resolvidos. Já com a técnica *few-shot*, o desempenho aumentou para 64,00% de resoluções bem-sucedidas, enquanto 36,00% dos problemas permaneceram sem solução. Por fim, o modelo Gemini alcançou uma taxa de acerto de 54,67%, com 45,33% de casos não solucionados, considerando a técnica *zero-shot*. Com a abordagem *few-shot*, também houve melhoria, chegando a 64,67% de sucesso e 35,33% de situações não resolvidas.

5.1.2 Teste Estatístico

Realizamos um teste estatístico Qui-Quadrado (Morettin e de Oliveira Bussab, 2010; Larson e Farber, 2015; Bracarense, 2010) de independência para avaliar diferenças no desempenho dos modelos Copilot Chat, LLaMA, DeepSeek e Gemini, considerando o total de casos resolvidos e não resolvidos no cenário *zero* e *few-shot*. Utilizamos o teste para verificar se havia associação estatisticamente significativa entre o modelo utilizado e o sucesso da refatoração.

As hipóteses testadas foram:

Hipótese nula (H_0): não há associação entre o modelo e o resultado, ou seja, os modelos apresentam desempenhos equivalentes.

Hipótese alternativa (H_1): há associação entre o modelo e o resultado, ou seja, pelo menos um modelo apresenta desempenho significativamente diferente.

5.1.2.1 Resultados Qui-Quadrado Comparação Zero-Shot

A Tabela 5.2 apresenta a distribuição dos resultados no cenário *zero-shot*, mostrando a quantidade de métodos *resolvidos* e *não resolvidos* por cada LLM. Por exemplo, o Copilot Chat obteve 79 resolvidos e 71 não resolvidos, enquanto o LLaMA registrou 66 resolvidos e 84 não resolvidos. Já O Gemine e DeepSeek, resolveram 82 e 85 casos e não resolveram 68 e 65, respetivamente.

Tabela 5.2: Tabela de Contingência entre os Modelos de LLMs no Cenário Zero-shot

Modelo	**Resolvidos**	**Não Resolvidos**	**Total**
Copilot Chat	79	71	150
LLaMA	66	84	150
DeepSeek	85	65	150
Gemini	82	68	150
Total	**312**	**288**	**600**

A Tabela 5.3 apresenta os resultados do teste Qui-Quadrado, juntamente com as frequências observadas e esperadas, para a comparação dos modelos de LLMs nas categorias *resolvidos* e *não resolvidos* no cenário *zero-shot*. As frequências esperadas indicam os valores que seriam esperados caso não houvesse diferenças entre os modelos.

,

Tabela 5.3: Resultados do Teste Qui-Quadrado e Frequências Observadas e Esperadas para Comparação dos Modelos no Cenário Zero-shot

Estatística		**Valor**
Qui-Quadrado (χ^2)		5.609
Graus de liberdade (df)		3
p-valor		0.1323
Modelo	**Resolvidos Obs. /Esp.**	**Não Resolvidos Obs. /Esp.**
Copilot Chat	79 / 78.0	71 / 72.0
LLaMA	66 / 78.0	84 / 72.0
DeepSeek	85 / 78.0	65 / 72.0
Gemini	82 / 78.0	68 / 72.0

5.1.2.2 Resultados Qui-Quadrado Comparação Few-Shot

A Tabela 5.4 apresenta a distribuição dos resultados observados para cada modelo de linguagem no cenário *few-shot*. A tabela detalha a quantidade de métodos classificados como *resolvidos* e *não resolvidos* para cada LLM. Por exemplo, o modelo Copilot Chat registrou 95 resoluções bem-sucedidas e 55 malsucedidas e o LLaMA obteve 83 casos resolvidos e 67 não resolvidos.

Tabela 5.4: Tabela de contingência entre os modelos de LLMs no cenário few-shot

Modelo	**Resolvidos**	**Não Resolvidos**	**Total**
Copilot Chat	95	55	150
LLaMA	83	67	150
DeepSeek	96	54	150
Gemini	97	53	150
Total	**371**	**229**	**600**

A Tabela 5.5 apresenta os resultados do teste Qui-Quadrado, juntamente com as frequências observadas e esperadas, para a comparação dos modelos de LLMs nas categorias *resolvidos* e *não resolvidos* no cenário *few-shot*.

O teste Qui-Quadrado de independência não apontou diferenças estatisticamente significativas no desempenho dos modelos, tanto no cenário *zero-shot* ($\chi^2(3) = 5{,}609$, $p = 0.132$) quanto *few-shot* ($\chi^2(3) = 3.637$, $p = 0.303$). As frequências esperadas sob a hipótese nula de igualdade entre modelos variaram de 57.25 a 92.75 (*few-shot*), atendendo aos pressupostos do teste. Embora diferenças numéricas sejam observadas (ex.: LLaMA com menor taxa de resolução), estas não foram estatisticamente significativas nos níveis convencionais ($\alpha = 0.05$).

Tabela 5.5: Resultados do teste Qui-Quadrado e frequências observadas e esperadas para comparação dos modelos no cenário few-shot

Estatística		**Valor**
Qui-Quadrado (χ^2)		3.637
Graus de liberdade (df)		3
p-valor		0.3034
Modelo	**Resolvidos Obs./Esp.**	**Não Resolvidos Obs./Esp.**
Copilot Chat	95 / 92.75	55 / 57.25
LLaMA	83 / 92.75	67 / 57.25
DeepSeek	96 / 92.75	54 / 57.25
Gemini	97 / 92.75	53 / 57.25

Os resultados do teste indicam que, ao nível de significância de 5%, não se pode rejeitar a hipótese nula (H_0). Isso significa que as diferenças observadas entre os modelos podem ser atribuídas ao acaso, sem evidência estatística de que algum modelo tenha se destacado significativamente.

Hipótese nula (H_0): Estatisticamente, não há associação entre o modelo de linguagem e o resultado (resolvido ou não resolvido), ou seja, todos os modelos apresentam desempenho semelhante.

5.1.3 Resultados Por Regras do SonarQube

Nesta seção, apresentamos os resultados detalhados por regra do SonarQube selecionada na metodologia. A Tabela 5.6 e a Figura 5.2 mostram os resultados dos casos bem-sucedidos para todos os modelos, considerando as duas técnicas de prompting avaliadas. De maneira geral, observa-se que os resultados foram bastante semelhantes entre os modelos para a maioria das regras.

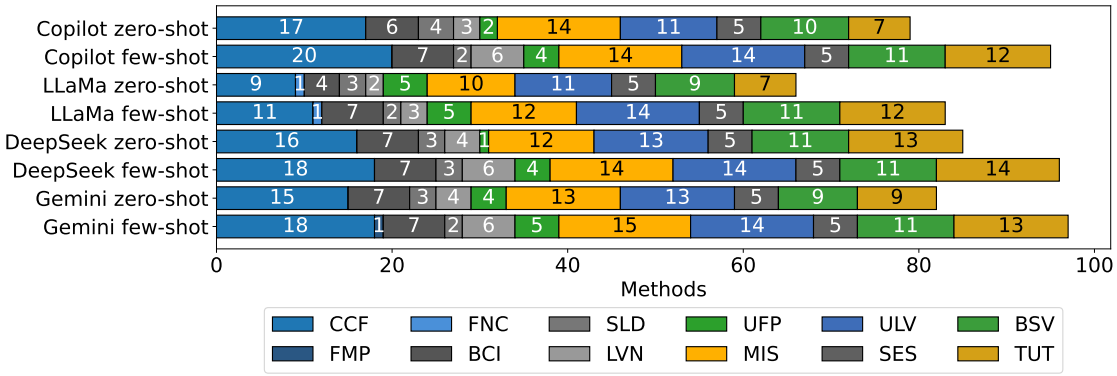

Figura 5.2: Comparativo de Acertos por LLM/Prompt/Regra

Tabela 5.6: Resultados por LLM/Prompt/Regra - Resolvidos

Resolvido	LLM	CCF	FMP	FNC	BCI	SLD	LVN	UFP	MIS	ULV	SES	BSV	TUT	TOT
Zero-Shot	Copilot	**17**			6	4	3	2	**14**	11	5	10	7	**79**
	LLaMA	9		**1**	4	3	2	5	10	11	5	9	7	**66**
	DeepSeek	**16**			7	3	4	1	12	**13**	5	11	**13**	**85**
	Gemini	**15**			7	3	4	4	**13**	**13**	5	9	9	**82**
Few-Shot	Copilot	**20**			7	2	6	4	**14**	**14**	5	11	12	**95**
	LLaMA	11		**1**	7	2	3	5	12	**14**	5	11	12	**83**
	DeepSeek	**18**			7	3	6	4	**14**	**14**	5	11	**14**	**96**
	Gemini	**18**	1		7	2	6	5	**15**	**14**	5	11	13	**97**

Na regra **CCF**, referente à alta complexidade cognitiva, registraram-se 33 ocorrências em 10 projetos. No cenário *zero-shot*, *Copilot Chat* e *DeepSeek* resolveram 17 casos cada (51,5%). No *few-shot*, *DeepSeek* e *Gemini* alcançaram 18 soluções (54,5%), enquanto o *Copilot Chat* obteve 20 (60,6%). O *LLaMA* apresentou o menor desempenho, com 9 (27,3%) e 11 (33,3%) casos resolvidos nos respectivos cenários.

Na regra **MIS**, sobre excesso de condicionais aninhadas, os modelos tiveram bom desempenho. O *Copilot Chat* resolveu 14 casos (93,3%) em ambos os cenários; o *LLaMA* obteve 10 (66,7%) no *zero-shot* e 12 (80%) no *few-shot*; o *DeepSeek*, 11 (73,3%) e 14 (93,3%); e o *Gemini*, 13 (86,7%) e 15 (100%).

Regras de menor complexidade, como **SES**, **TUT** e **BCI**, foram mais simples de serem resolvidas. Os LLMs demonstraram maior facilidade na correção desses problemas. A regra **SES**, por exemplo, foi solucionada em 100% dos casos e em todos os cenários por todos os modelos, seguida da **BCI**, que apresentou taxa de sucesso de 100% na maioria dos cenários, exceto para o *LLaMA* e *Copilot*, ambos em *zero-shot*, que atingiu *acima de 50% dos casos*.

Já nas regras **FMP** (remoção de parâmetros não utilizados) e **FNC** (padronização de nomes de funções), todos os modelos apresentaram baixo desempenho. Apenas o *Gemini*, no *few-shot*, resolveu 1 caso da **FMP**, e o *LLaMA* corrigiu 1 ocorrência da **FNC** em ambos os cenários.

5.1.4 Resultado de Técnica de Refatoração

Registramos a técnica de refatoração aplicada pelos LLMs em cada correção bem-sucedida, permitindo analisar não apenas se o problema foi resolvido, mas também como a refatoração foi realizada. Esses dados revelam o perfil das refatorações, mostrando se os modelos têm mais facilidade com ajustes simples, como renomeação de variáveis, ou com operações mais complexas, como extração de métodos e consolidação de condicionais. Dessa forma, foi possível mapear as capacidades e limitações práticas dos LLMs na correção de problemas de manutenibilidade. Registramos apenas as técnicas nos casos em que a refatoração foi bem-sucedida e o problema identificado foi de fato corrigido.

Tabela 5.7: Legenda das Abreviações das Técnicas de Refatoração

Abreviação	**Descrição Completa**
Extração	Extração de Métodos
Ifs	Consolidação de Condicionais `if`
Renom. Var	Renomeação de Variáveis
Var. Morta	Remoção de Variáveis Mortas
Rem. ToDo	Remoção de Comentários `TODO` obsoletos ou resolvidos
ToDo/Ticket	Conversão de Comentário `TODO` para Referência Formal de Tarefa/Ticket
Bool Inv.	Consolidação de Lógica Booleana Invertida
Renom. Param	Renomeação de Parâmetro
Fat. Método	Substituição de Fatiamento de Strings por Método Específico
Param Morto	Remoção de Parâmetros Não Utilizados
Constante	Extração de Strings Duplicadas para Constantes
Outros	Outras Refatorações Não Classificadas

As técnicas de refatoração listadas na Tabela 5.7 são baseadas nas práticas clássicas descritas por Fowler (2019), que detalha métodos como extração de métodos, renomeação de variáveis e remoção de código morto, além de outras fontes comple-

mentares que abordam boas práticas de código limpo e manutenção, como Martin (2008) e Feathers (2004).

Na Tabela 5.8, apresentamos a quantidade de refatorações realizadas por cada LLM, organizadas por técnica. A primeira coluna exibe o nome do modelo, enquanto as demais colunas indicam as diferentes técnicas de refatoração aplicadas. Em cada célula, consta o número de vezes que determinada técnica foi utilizada pelo respectivo modelo.

Tabela 5.8: Distribuição das Refatorações por Modelo e Técnica

LLM (Técnica)	Extração	Ifs	Renom. Var	Var. Morta	Rem. ToDo	ToDo/Ticket	Bool Inv.	Renom. Param	Fat. Método	Param Morto	Constante	Outros
Copilot (Zero-shot)	**19**	14	20	3	7	0	6	0	5	2	3	0
Copilot (Few-shot)	**20**	12	23	4	**13**	2	9	3	5	2	2	0
LLaMA (Zero-shot)	11	11	19	3	6	2	3	1	5	3	1	1
LLaMA (Few-shot)	13	12	23	4	9	2	8	2	5	3	2	0
DeepSeek (Zero-shot)	**24**	10	20	6	6	3	7	1	5	1	2	0
DeepSeek (Few-shot)	20	15	**27**	2	7	5	7	3	5	2	3	0
Gemini (Zero-shot)	16	14	22	3	6	2	6	1	5	2	4	1
Gemini (Few-shot)	**20**	**16**	**30**	1	7	5	5	3	5	3	2	0
Total	**143**	**104**	**184**	**26**	**61**	**21**	**51**	**14**	**40**	**18**	**19**	**2**

As técnicas de refatoração mais recorrentes neste experimento foram a *extração de métodos* e a *renomeação de variáveis*, aplicadas 143 e 184 vezes, respectivamente, seguidas da *consolidação de condicionais if*, com 104 aplicações. Outras técnicas, como a *remoção de variáveis mortas*, a *consolidação de lógica booleana invertida* e a *extração de strings duplicadas para constantes*, também foram utilizadas, além da *renomeação e remoção de parâmetros ou variáveis não utilizados*.

Resumo Seção 5.1

Na análise geral, os LLMs apresentaram desempenhos semelhantes na refatoração de código Python voltada à manutenibilidade, com leve superioridade da técnica *few-shot* em relação ao *zero-shot*, tanto nos resultados numéricos quanto nos testes estatísticos. O teste Qui-Quadrado indicou que as diferenças entre os modelos não foram estatisticamente significativas nas duas téncias de prompts.
Na análise por regras do SonarQube, regras de alta complexidade cognitiva, como **CCF** e **MIS**, tiveram bom desempenho, especialmente nos modelos Copilot, DeepSeek e Gemini. Regras de baixa complexidade, como **SES**, **TUT** e **BCI**, foram resolvidas com alta eficácia por todos os modelos, enquanto regras **FMP** e **FNC** apresentaram baixo desempenho.
Em relação às técnicas de refatoração, os modelos aplicaram principalmente extração de métodos, renomeação de variáveis e consolidação de condicionais, demonstrando maior facilidade em ajustes simples, mas também capacidade de realizar operações mais complexas, como extração de métodos e manipulação de condicionais.

5.2 RQ2. Tipos de Erros Cometidos Por LLMs ao Refatorar Código Pyhton com Problemas de Mantuenabilidade

Nesta seção, apresentamos os tipos de erros cometidos pelos LLMs, evidenciando suas limitações em diversas situações que comprometem o desempenho desses modelos neste experimento.

5.2.1 Limitações dos LLMs

As Figuras 5.3 apresentam as limitações dos LLMs ao tentar refatorar códigos com problemas de manutenibilidade. O objetivo é comparar o desempenho dos modelos nas diferentes categorias de análise, evidenciando as limitações observadas em cada modelo e em cada técnica de prompting.

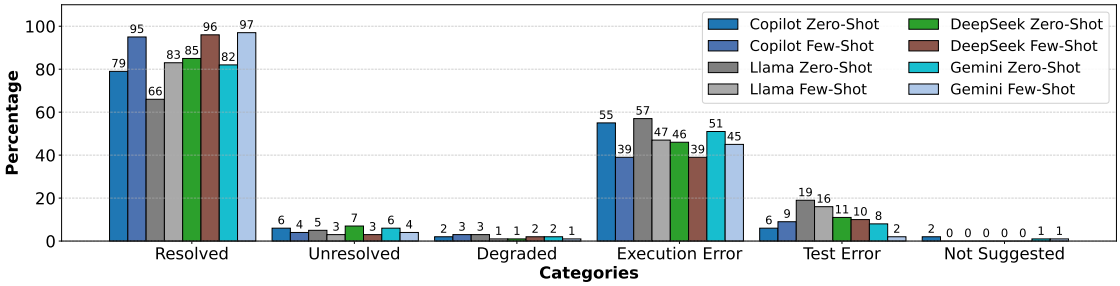

Figura 5.3: Resultado Por Categoria - Comparação por LLM

.

Mesmo com taxas de sucesso acima de 50% em quase todos os cenários, os LLMs cometeram erros relevantes. Em alguns casos, os modelos passaram nos testes, mas não corrigiram o problema de manutenibilidade, ocorrendo entre 2,00% e 4,66% dos exemplos, com pouca variação entre LLMs. Também foram registrados casos em que a refatoração *degradou* o código, introduzindo novos problemas, embora em frequência baixa (entre 0,67% e 2,00%).

Os *erros de execução* foram os mais recorrentes, especialmente no *zero-shot*, chegando a 38,00% no LLaMA e entre 26,00% a 36,65% nos demais modelos. No *few-shot*, essas taxas diminuíram, variando entre 25,66% e 34,00%. Já os problemas relacionados a *falhas em testes* também se destacaram, com maior incidência no LLaMA com 12,66% em *zero-shot* e 10,66% em *few-shot*, seguido por Copilot Chat, DeepSeek e Gemini, cujas taxas oscilaram entre 1,33% e 7,33%.

Tabela 5.9: Resultados por LLM/Prompt/Regra - Não Resolvidos

Não Resolvido	LLM	CCF	FMP	FNC	BCI	SLD	LVN	UFP	MIS	ULV	SES	BSV	TUT	TOT
Zero-Shot	Copilot	2				**4**								**6**
	LLaMA		1			**3**						1		**5**
	DeepSeek					**4**			3					**7**
	Gemini	1				**4**			1					**6**
Few-Shot	Copilot					**4**								**4**
	LLaMA	1				**2**								**3**
	DeepSeek					**3**								**3**
	Gemini	2				**2**								**4**

A Tabela 5.9 apresenta os resultados das situações em que os modelos realizaram a refatoração sem gerar erros nos testes ou durante a execução, porém o problema de manutenibilidade originalmente detectado permaneceu sem solução. Esses casos foram relativamente poucos para todos os LLMs avaliados. Um ponto que chama atenção é a regra **SLD**, onde todos os modelos apresentaram ocorrências semelhantes desse tipo de falha (2 a 4 casos). Já para as regras **FNC**, **BCI**, **LVN**, **UFP**, **ULV**, **SES** e **TUT**, não houve registro de nenhum caso nesse cenário.

Na categoria *Degradados* (Tabela 5.10), o padrão foi semelhante: as refatorações não apenas falharam em corrigir os problemas, como introduziram novas violações. A regra mais afetada foi a **CCF**, em que todos os modelos, exceto o *Gemini* no cenário *zero-shot*, apresentaram esse tipo de falha, variou entre 1 a 3 casos.

Tabela 5.10: Resultados por LLM/Prompt/Regra- Degradados

Degradados	LLM	CCF	FMP	FNC	BCI	SLD	LVN	UFP	MIS	ULV	SES	BSV	TUT	TOT
Zero-Shot	Copilot	**1**											**1**	**2**
	LLaMA	**3**												**3**
	DeepSeek	**1**												**1**
	Gemini					1						1		**2**
Few-Shot	Copilot	**1**				1							1	**3**
	LLaMA	**1**												**1**
	DeepSeek	**2**												**2**
	Gemini	**1**												**1**

Os erros de execução foram os mais frequentes nas refatorações, ocorrendo quando o código falhou ao ser executado após as modificações. A Tabela 5.11 mostra as regras mais afetadas. Na regra **CCF** (33 casos), o *Gemini* apresentou 13 (39,39%) e 11 (33,22%) erros nos cenários *zero-shot* e *few-shot*, respectivamente. O *LLaMA* registrou 13 (39,39%) e 10 (30,30%) também nos dois cenários de prompts, seguindo do *Copilot Chat* 7 (21,21%) e 6 (18,18%), e o *DeepSeek* 9 (27,27%) e 5 (15,15%) no mesmo contexto dos anteriores.

Proporcionalmente, as regras que apresentaram mais falhas foram **FMP** e **FNC**. A regra FMP variou entre 9 e 11 ocorrências, chegando a 100% em quase todos os

modelos e técnicas, com exceção do LLaMA no *zero-shot* e do DeepSeek no *few-shot*, ambos com 90% de erros. Já a regra FNC variou entre 7 e 8 ocorrências, com taxas de falha entre 90% e 100% nos modelos testados.

Por outro lado, algumas regras não geraram erros de execução. Na **SES**, por exemplo, nenhuma refatoração resultou em falhas em qualquer modelo ou técnica.

Tabela 5.11: Resultados por LLM/Prompt/Regra - Erros de Execução

Erros de Execução	**LLM**	**CCF**	**FMP**	**FNC**	**BCI**	**SLD**	**LVN**	**UFP**	**MIS**	**ULV**	**SES**	**BSV**	**TUT**	**TOT**
Zero-Shot	Copilot	**7**	11	8	1	5	4	10		2		1	6	**55**
	LLaMA	**13**	10	7	3	4	3	5	3	3			6	**57**
	DeepSeek	**9**	11	8		4	2	10		1			1	**46**
	Gemini	**13**	11	8		4	1	8		1			4	**50**
Few-Shot	Copilot	**6**	11	8		4	1	8					1	**39**
	LLaMA	**10**	11	7		8	3	5	1				2	**47**
	DeepSeek	**5**	10	8		7	1	8						**39**
	Gemini	**11**	9	8		9	1	7					1	**46**

A Figura 5.12 evidencia as limitações dos LLMs em relação aos *erros de testes*, o segundo tipo mais frequente neste experimento. Nesses casos, os modelos refatoraram o código e, por vezes, corrigiram o problema de manutenibilidade, mas os testes automatizados revelaram divergências na saída, indicando falhas no comportamento esperado. O erro mais recorrente foi o *AssertionError*, caracterizado pela discrepância entre o resultado produzido e o valor esperado.

Tabela 5.12: Resultados por Projeto, LLM e Técnica - Erros de Testes

Erros de Testes	**LLM**	**CCF**	**FMP**	**FNC**	**BCI**	**SLD**	**LVN**	**UFP**	**MIS**	**ULV**	**SES**	**BSV**	**TUT**	**TOT**
Zero-Shot	Copilot	5							1					**6**
	LLaMA	**8**				**3**	2	2	2			1	1	**19**
	DeepSeek	7				2	1	1						**11**
	Gemini	4					2					2		**8**
Few-Shot	Copilot	6				2			1					**9**
	LLaMA	**10**				1	1	2	2					**16**
	DeepSeek	8	1						1					**10**
	Gemini	1				1								**2**

Entre as regras, a **CCF** concentrou a maior parte das falhas dessa categoria. No cenário *zero-shot*, o *Copilot Chat* registrou 5 erros (15,15%), o *LLaMA* 8 (24,24%), o *DeepSeek* 6 (21,21%) e o *Gemini* 4 (12,12%). Já no *few-shot*, foram 6 erros no *Copilot Chat*, 10 (30,3%) no *LLaMA* e 8 (24,24%) no *DeepSeek* e somente 2 para o *Gemini*. Outras regras também apresentaram erros de teste, mas em proporções menores. Em contrapartida, algumas regras, como **FNC**, **BCI**, **ULV** e **SES**, não geraram esse tipo de falha em nenhum dos modelos avaliados.

Por fim, observamos situações em que os LLMs não sugeriram nenhum tipo de correção para códigos com problemas de manutenibilidade. Esses casos foram bastante

raros. O *Copilot Chat* apresentou duas ocorrências: uma na regra **CCF** e outra na **ULV**. O Gemini também registrou dois casos, sendo um na regra **MIS**, no cenário *zero-shot*, e outro na **FMP**, com *few-shot*.

Tabela 5.13: Resultados por LLM/Prompt/Regra - Não Sugeridos

Não Sugeridos	LLM	CCF	FMP	FNC	BCI	UFP	LVN	UFP	MIS	ULV	SES	BSV	TUT	TOT
Zero-Shot	Copilot	1								1				**2**
	LLaMA													
	DeepSeek													
	Gemini								1					**1**
Few-Shot	Copilot													
	LLaMA													
	DeepSeek													
	Gemini		1											**1**

5.2.2 Tipos de Erros (Testes e Execução)

Nesta seção, detalhamos os tipos de falhas geradas pelos modelos, considerando tanto a execução dos programas quanto os resultados dos testes automatizados.

Segundo a documentação oficial do Python (Python Software Foundation, 2024a), erros como *TypeError*, *NameError* e *ValueError* são exceções de tempo de execução, pois ocorrem durante a execução do programa. Já o *SyntaxError* é identificado na fase de análise sintática, antes mesmo da execução (Lutz, 2013). Neste estudo, contudo, os erros de sintaxe foram agrupados aos de execução, por também impedirem o funcionamento do programa. Essa decisão visou simplificar e padronizar a análise.

Na Tabela 5.14, apresentamos os tipos de erros, tanto de teste com de execução, detectados nos logs gerados durante a execução/testes dos programas, após a aplicação da refatoração em códigos com problemas de manutenibilidade. Na primeira coluna da tabela, estão listados os tipos de erros encontrados; na segunda coluna, a descrição de cada erro, ou seja, o que pode ter provocado a falha. Já na terceira coluna, os erros são classificados de acordo com sua natureza, sendo agrupados em erros de execução ou erros de teste.

O erro do tipo *AssertionError* foi o único classificado como erro de teste, identificado durante a execução dos testes automatizados. Entre os erros de execução, destacam-se *NameError*, *TypeError*, *ImportError*, *SyntaxError*, *AttributeError*, *Exception* e *KeyError*. Outros tipos de erro, como *FileNotFoundError*, também foram observados, mas ocorreram com frequência muito baixa.

Já o erro de teste ocorre quando há alguma falha na asserção ou na lógica durante a execução dos testes automatizados (Pytest Development Team, 2024; Python Software Foundation, 2024b). Isso indica que o código foi executado, mas o resultado obtido não corresponde ao que era esperado nos testes, comprometendo o comportamento correto do sistema.

Tabela 5.14: Descrição e Classificação dos Tipos de Erros

Erro	**Descrição**	**Classificação**
`AssertionError`	Ocorre quando uma asserção falha durante os testes; o código executa, mas o resultado não corresponde ao esperado.	Teste
`NameError`	Tentativa de uso de variável ou função que não foi definida.	Execução
`TypeError`	Operação realizada entre tipos incompatíveis.	Execução
`ImportError`	Falha ao tentar importar um módulo ou componente.	Execução
`ValueError`	Valor passado é válido em tipo, mas inadequado para a operação esperada.	Execução
`SyntaxError`	Estrutura sintática incorreta que impede a interpretação do código.	Execução
`AttributeError`	Tentativa de acessar atributo que não existe em um objeto.	Execução
`Exception`	Exceção genérica, utilizada para capturar falhas não específicas.	Execução
`ModuleNotFoundError`	Módulo especificado não foi encontrado.	Execução
`KeyError`	Acesso a chave inexistente em dicionário.	Execução
Outros	Categoria usada para erros não previstos ou específicos demais.	Execução

Na Tabela 5.15, apresentamos a quantidade de erros de execução e de testes cometidos pelos LLMs nas duas técnicas de *prompting* avaliadas. Na primeira coluna é mostrado o tipo de erro, enquanto nas demais colunas são apresentadas as quantidades de erros cometidos por cada LLM e técnica de *prompting*. Na última coluna, exibe-se o total de erros de cada tipo de erro.

Tabela 5.15: Tipos de Erros (Testes e Execução)

Tipo de Erro	**C-ZS**	**C-FS**	**L-ZS**	**L-FS**	**D-ZS**	**D-FS**	**G-ZS**	**G-FS**	**TT**
AssertionError	6	9	**19**	16	11	10	8	2	**81**
NameError	11	8	11	**15**	11	11	**14**	9	**90**
TypeError	**17**	**13**	12	12	**12**	7	12	7	**92**
ImportError	7	7	7	9	10	**13**	9	**14**	**76**
ValueError	1	4	4	2	1	0	1	3	16
SyntaxError	**6**	3	**6**	2	2	1	2	1	23
AttributeError	10	3	**15**	5	10	6	11	7	67
Exception	1	0	0	0	0	0	0	0	2
ModuleNotFoundError	1	0	0	0	0	0	0	4	5
KeyError	0	0	2	2	0	0	0	0	4
Outros	1	1	0	0	0	1	2	0	5
Total	61	48	76	63	56	49	58	46	458

C-ZS = Copilot Zero-Shot / C-FS = Copilot Few-Shot / L-ZS = LLaMA Zero-Shot
L-FS = LLaMA Few-Shot / D-ZS = DeepSeek Zero-Shot / D-FS = DeepSeek Few-Shot
G-ZS = Gemini Zero-Shot / G-FS = Gemini Few-Shot / TT = Total

Observa-se que os erros de execução mais frequentes gerados pelos modelos foram o *TypeError*, com 92 ocorrências no agregado, geralmente relacionado a operações

entre tipos incompatíveis, seguido pelo *NameError*, com 90 registros, que indica o uso de variáveis ou funções não definidas, e pelo *ImportError*, com 76 casos, associado a falhas na importação de módulos. Entre os erros de teste, o *AssertionError* foi o mais comum, com um total de 81 ocorrências.

Outro erro com número expressivo de ocorrências foi o *AttributeError* , que indica tentativas de acessar atributos inexistentes em objetos, com um total de 67 casos. Já o erro que apresentou menos frequência foi o *Exception* com somente uma ocorrência.

No caso do *TypeError*, o maior número de ocorrências foi observado no modelo *Copilot Chat* com a técnica *zero-shot*, totalizando 17 casos (11,33%), no total de 150 problemas avaliados. Em seguida, o demais cenários apresentaram entre 7 (4,67%) a 13 (8.67%) ocorrências. Para o *AttributeError*, o maior número foi identificado no *LLaMA* com *zero-shot*, com 15 registros (10,00%), o demais cenários apresentaram valores entre 3 (2,00%) a 11 (7,33%) casos.

Quanto ao *NameError*, os modelos que mais ocorreram esse tipo de falha foram o *LLaMA* na técnica *few-shot*, com 15 ocorrências (10%), e o *Gemini zero-shot* , com 14 (9,33%). Além desses, também foram registrados outros erros com menor frequência, como *ValueError* (16), *SyntaxError* (23), *ModuleNotFoundError* (5), *KeyError* (4), entre outras exceções pontuais e menos recorrentes.

5.2.3 Classificação dos Tipos de Erros (Testes e Execução)

A fim de agrupar os erros encontrados, optamos por classificá-los em três categorias: *erros semânticos*, *erros de sintaxe* e *erros de teste*. Essa separação foi importante para entendermos quais tipos de falhas os LLMs cometem com maior frequência, permitindo uma análise mais organizada e direcionada dos resultados. A Tabela 5.16 apresenta, na primeira coluna, o tipo de erro; na segunda, os erros identificados; e, na última, sua respectiva descrição.

Tabela 5.16: Classificação dos Tipos de Erros

Tipo de Erro	**Erros**	**Raciocínio**
Erro Semântico	`NameError`, `TypeError`, `ImportError`, `ValueError`, `AttributeError`, `KeyError`, `ModuleError`, `Exception`	Código executa, mas ocorre falha lógica ou em tempo de execução (runtime).
Erro Sintático	`SyntaxError`	Código inválido, não executa devido a erros de estrutura ou sintaxe.
Erro de Teste	`AssertionError`	Código executa, mas não atende aos requisitos esperados, falhando nos testes.

Embora o *AssertionError* possa ser interpretado como um erro semântico do código gerado, neste trabalho ele foi classificado como *Erro de Teste*, pois indica falha no atendimento aos requisitos verificados pelos testes automatizados. Essa distinção

permite avaliar de forma mais precisa a capacidade dos LLMs em produzir soluções funcionais. Testes automatizados, conforme apontam Meszaros (2007), são fundamentais para assegurar o comportamento esperado do sistema e constituem uma etapa crítica e independente no ciclo de desenvolvimento (Sommerville, 2016; Pressman et al., 2021).

Essa abordagem está alinhada com as práticas da Engenharia de Software, em que os testes automatizados são tratados como uma etapa distinta e fundamental dentro do ciclo de desenvolvimento Sommerville (2016); Pressman et al. (2021). Separar esse tipo de erro é essencial para avaliar se os LLMs são capazes de gerar códigos não apenas sintaticamente válidos, mas também semanticamente corretos e funcionais em relação aos requisitos esperados.

A Figura 5.4 apresenta um comparativo das taxas de erros cometidos pelos LLMs durante a refatoração de códigos Python com problemas de manutenibilidade. As taxas foram calculadas com base no total de 150 métodos avaliados por modelo. Nesta análise, foram considerados apenas os erros de execução, erros de teste e erros de sintaxe, ou seja, casos em que o código refatorado não foi executado corretamente ou não passou nos testes automatizados.

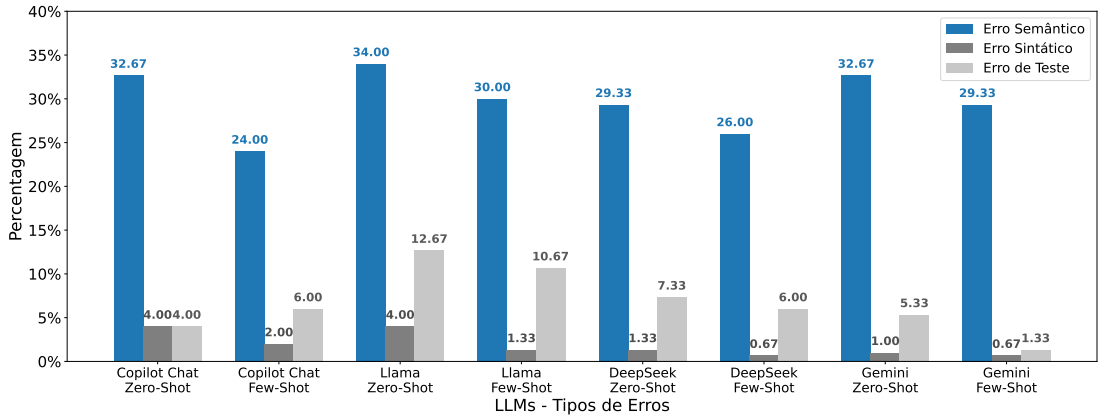

Figura 5.4: Comparativo de Erros de LLM/Execução/Teste

O *LLaMA* apresentou a maior taxa de erro de teste, com 12,67% em zero-shot, enquanto em few-shot registrou 10,67%. Já os modelos *Gemini* e *DeepSeek* (few-shot) tiveram as menores taxas de erro de sintaxe, de apenas 0,67%.

Em relação aos erros semânticos, os maiores índices foram observados nos modelos *Copilot* e *DeepSeek* (32,67% em zero-shot), seguidos pelo *LLaMA* (34,00% em zero-shot e 30,00% em few-shot). Já os menores valores ocorreram no *Copilot Chat* (24,00%) e no *DeepSeek* (26,00%), ambos em few-shot.

De modo geral, todos os LLMs apresentaram mais erros semânticos do que sintáticos, incluindo *TypeError*, *NameError*, erros de importação ou valores inválidos.

Independentemente da técnica (zero-shot ou few-shot), houve falhas de execução e teste, mostrando limitações na refatoração automática. O experimento considerou apenas uma interação por prompt; abordagens mais interativas poderiam reduzir esses casos. Esses resultados indicam que, mesmo com código sintaticamente correto, os LLMs ainda não são suficientemente robustos para uso prático sem supervisão humana, especialmente em códigos com problemas estruturais.

Resumo Seção 5.2

De modo geral, os resultados mostram que os LLMs apresentam limitações importantes na refatoração automática de código. Embora consigam propor soluções em boa parte dos casos, ainda ocorrem falhas relevantes, como erros de execução, de teste e de sintaxe. Observa-se que os erros semânticos são os mais recorrentes, refletindo a dificuldade dos modelos em compreender plenamente a lógica do programa. Além disso, há situações em que as refatorações pioram a qualidade do código ou não trazem nenhuma sugestão útil. Esses achados reforçam que, mesmo com avanços recentes, os LLMs ainda não são robustos o suficiente para aplicações práticas sem supervisão e validação criteriosa.

5.3 RQ3. Impacto Entre as Técnicas de Prompts

Para investigar se havia diferença estatisticamente significativa no desempenho das técnicas de prompting *few-shot* e *zero-shot*, foram realizados testes estatísticos para cada modelo individualmente, além de uma análise geral considerando os resultados agregados de todos os modelos.

Hipóteses para cada modelo individualmente (como na Tabela 5.17):

Hipótese nula (H_0): Não há associação entre a técnica de prompting (few-shot vs. zero-shot) e o desempenho do modelo, ou seja, para esse modelo específico, a taxa de resolução é equivalente nas duas técnicas.

Hipótese alternativa (H_1): Existe associação entre a técnica de prompting e o desempenho do modelo, ou seja, para esse modelo, a taxa de resolução difere entre as técnicas.

Na Tabela 5.17 são apresentados os testes de Qui-quadrado e Fisher (Morettin e de Oliveira Bussab, 2010; Larson e Farber, 2015; Bracarense, 2010) aplicados aos dados de cada modelo. O teste exato de Fisher foi utilizado como complemento ao Qui-Quadrado para garantir maior robustez, considerando a natureza categórica dos dados e, especialmente, casos com frequências esperadas baixas (menores que 0.05) (Larson e Farber, 2015; Morettin e de Oliveira Bussab, 2010).

Observa-se que, individualmente, nenhum dos modelos apresentou diferença estatisticamente significativa entre as técnicas, pois todos os valores de p são maiores que 0,05.

Tabela 5.17: Análise estatística por modelo – testes Qui-quadrado e Fisher

Modelo	**Tabela de Contingência**	**Qui-quadrado (p)**	**Fisher (p)**
Copilot	[[79, 71], [95, 55]]	0,0793	0,0791
LLaMA	[[66, 84], [83, 67]]	0,0647	0,0645
DeepSeek	[[85, 65], [96, 54]]	0,2379	0,2379
Gemini	[[82, 68], [97, 53]]	0,0994	0,0992

Os resultados apontam que, para todos os modelos, dado que o valor de $p > 0.05$, não se rejeita H_0. Isso sugere que, individualmente, a técnica utilizada não afeta significativamente o desempenho.

No entanto, ao considerar o conjunto completo de métodos *resolvidos* e *não resolvidos* agregados entre todos os modelos, conforme apresentado na Tabela 5.18, observa-se que a técnica *few-shot* resultou em maior número absoluto de métodos resolvidos (371 contra 312 no *zero-shot*).

Hipóteses para a análise geral (como na Tabela 5.18):

Hipótese nula (H_0): A técnica de prompting (few-shot ou zero-shot) não influencia o desempenho geral, ou seja, a proporção de métodos resolvidos é igual em ambas as técnicas.

Hipótese alternativa (H_1): A técnica de prompting influencia o desempenho geral, ou seja, existe diferença nas proporções de acertos entre few-shot e zero-shot.

Tabela 5.18: Distribuição dos Métodos Resolvidos e não Resolvidos nas Técnicas Few-Shot e Zero-Shot

Técnica	**Resolvidos**	**Não Resolvidos**	**Total**
Few-shot	371	229	600
Zero-shot	312	288	600
Total	**683**	**517**	**1200**

A análise estatística da Tabela 5.19 mostra que essa diferença é estatisticamente significativa, com um valor de $p = 0,0007$ no teste de Qui-quadrado de Pearson e no teste exato de Fisher, ambos abaixo do nível de significância de 0,05.

Tabela 5.19: Resultado da análise estatística entre as técnicas few-shot e zero-shot

Teste	**Estatística**	**Valor-p**
Qui-quadrado de Pearson	$\chi^2 = 11.53$	0,0007
Teste exato de Fisher	–	0,0007

Esses resultados evidenciam que a técnica *few-shot* apresenta uma taxa de sucesso superior (61,8%) na resolução dos problemas de manutenibilidade em comparação com a técnica *zero-shot* (52,0%).

O tamanho do efeito foi calculado com base na estatística do teste qui-quadrado (χ^2), segundo a fórmula

$$\phi = \sqrt{\frac{\chi^2}{n}},$$

onde n representa o número total de observações. Neste estudo, com $\chi^2 = 11,53$ e $n = 1200$, obteve-se

$$\phi = \sqrt{\frac{11,53}{1200}} \approx 0,098.$$

Os resultados indicam que, como o valor de $p = 0{,}0007$ é inferior a 5%, rejeita-se a hipótese nula (H_0), evidenciando uma associação estatisticamente significativa e apontando que a técnica *few-shot* supera a *zero-shot*. Contudo, o tamanho do efeito ($\phi = 0{,}098$) foi classificado como pequeno segundo Cohen (1988), sugerindo que a diferença prática entre as técnicas é modesta.

Assim, a técnica *few-shot* mostra desempenho ligeiramente superior, possivelmente por fornecer exemplos prévios que oferecem contexto adicional e orientam melhor a geração das respostas na refatoração de código.

Resumo Seção 5.3

Nesta seção, analisamos o impacto das técnicas de prompting *zero-shot* e *few-shot* no desempenho dos modelos. Quando considerados individualmente, não foram observadas diferenças estatisticamente significativas entre as abordagens, sugerindo que o tipo de prompting não influencia de forma relevante o desempenho isolado de cada modelo. Contudo, na análise agregada, a técnica *few-shot* apresentou desempenho superior ao *zero-shot*, diferença essa estatisticamente significativa, embora com efeito prático pequeno. Esses resultados indicam que o fornecimento de exemplos prévios pode favorecer a performance geral dos modelos na refatoração de código, oferecendo um contexto adicional que orienta melhor a geração das respostas.

5.4 RQ4. Opiniões dos Desenvolvedores Sobre as Soluções Geradas pelos LLMs

Realizamos uma avaliação com 32 participantes para verificar se as refatorações feitas pelos LLMs melhoraram a legibilidade do código, além de corrigirem problemas de manutenibilidade. Cada avaliador analisou 5 pares de código (original e refatorado), respondendo à pergunta: **"Qual código você considera mais legível?"**, com as opções: *Código A*, *Código B*, *Ambos* ou *Não sei*. Foram selecionados 20 pares por

modelo, usando a técnica *few-shot* apenas nos casos de refatoração bem-sucedida, devido ao seu desempenho superior ao *zero-shot*. Os formulários estão disponíveis em experiments-artifacts-msc.

Foram coletadas 160 avaliações[1], correspondentes a 80 pares de código, com cada par avaliado por dois participantes. Quando houve concordância, essa escolha foi considerada final; em caso de discordância, um terceiro avaliador independente realizou o desempate, analisando as justificativas anteriores e os códigos de forma imparcial. Dos 80 pares, 42 tiveram concordância direta e 38 exigiram voto de desempate.

5.4.1 Perfil dos Participantes

Tabela 5.20: Perfil dos Participantes da Avaliação

Ocupação		**Escolaridade**		**Experiência em Desenvolvimento**		
Categoria	Qtde	Categoria	Qtde	Tempo	Geral	Python
Profissional e Estudante	21	Graduação	15	0 a 3 anos	10	21
Profissional	6	Pós-graduação	17	3 a 6 anos	9	10
Estudante	4			Acima de 6 anos	13	1
Nenhuma das opções acima	1					

A Tabela 5.20 apresenta o perfil dos participantes do experimento. Quanto à situação de estudo e trabalho, 21 declararam estudar e trabalhar simultaneamente, 6 apenas trabalham, 4 apenas estudam e 1 não se enquadra em nenhuma dessas categorias. Em relação à escolaridade, 15 possuem exclusivamente graduação e 17 já concluíram uma pós-graduação.

No que diz respeito à experiência profissional em desenvolvimento, 10 participantes têm entre 0 e 3 anos de atuação, 9 entre 3 e 6 anos e 13 mais de 6 anos de experiência. Sobre a experiência específica com a linguagem Python, 21 relataram entre 0 e 3 anos de prática, 10 possuem de 3 a 6 anos e apenas 1 participante tem mais de 6 anos de domínio da linguagem. Por fim, destaca-se que o terceiro avaliador, responsável por parte das análises, possui mais de seis anos de experiência em desenvolvimento com Python, além de formação em pós-graduação e atuação profissional na área, o que reforça a confiabilidade do estudo.

Os dados indicam que a maioria dos participantes trabalha e estuda simultaneamente, possui pós-graduação e apresenta experiência consolidada em desenvolvimento de software, sendo que 13 têm mais de seis anos de atuação. Em relação à linguagem Python, predomina o grupo de iniciantes (21 com até 3 anos de prática), embora haja também um contingente intermediário relevante (10 entre 3 e 6 anos).

[1] `https://drive.google.com/drive/folders/1FB8g2V5UH0tjsfYn3zB3ENL6wQ94Wr9u`

5.4.2 Resultados Quantitativos

Os resultados da avaliação de legibilidade de códigos gerados por LLMs, demonstram que esses modelos apresentam um potencial promissor para refatorar código Python funcional, tornando-o mais claro e acessível. Conforme mostra a Figura 5.5, dos 80 votos coletados, 81,25% dos avaliadores preferiram o código refatorado (B). Apenas 17,5% optaram pelo código original (A), e somente um caso (1,25%) indicou preferência por "ambos os códigos", nenhum resposta final teve com resultado a opção "não sei".

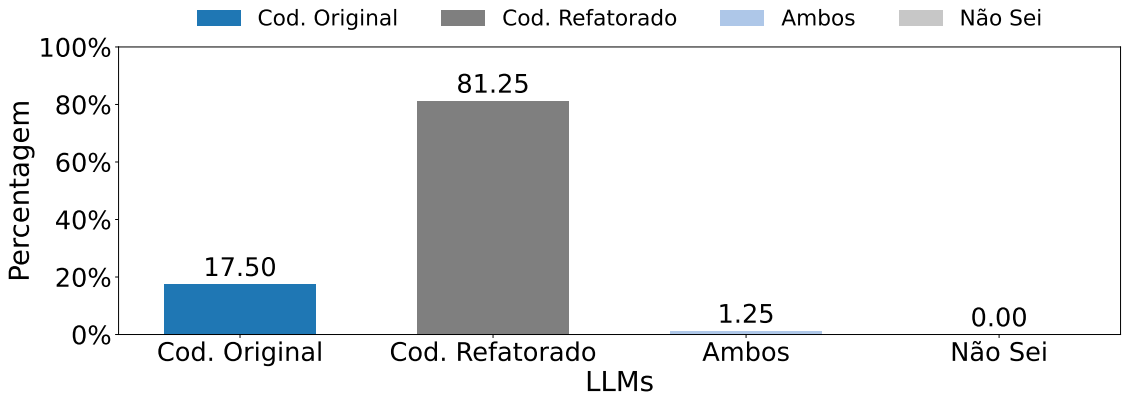

Figura 5.5: Resultado da Avaliação Humana

A Figura 5.6 apresenta os resultados da avaliação humana para cada LLM, indicando a quantidade de casos em que os participantes consideraram mais legível o código original, o código refatorado, ambos ou quando não souberam responder.

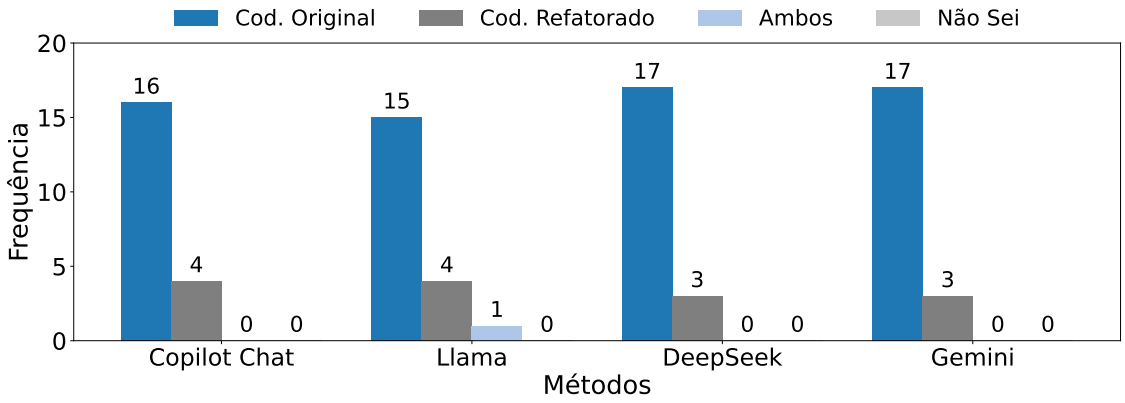

Figura 5.6: Resultado Avaliação Humana Por LLM

Os avaliadores demonstraram um padrão consistente de preferência pelas versões refatoradas em todos os modelos avaliados. No caso do Copilot Chat, 16 das 20 refatorações foram consideradas mais legíveis do que suas respectivas versões originais.

Para o LLaMA, 15 códigos refatorados foram preferidos, 3 avaliadores escolheram a versão original e 1 indicou ambos como igualmente legíveis. Já no DeepSeek e no Gemini, 17 das 20 refatorações foram escolhidas como mais legíveis em comparação ao código original.

Esse comportamento uniforme sugere que, independentemente do modelo utilizado, as refatorações tendem a gerar códigos mais claros e compreensíveis. A consistência entre os resultados dos diferentes modelos reforça a confiabilidade das conclusões obtidas na avaliação humana.

5.4.3 Percepção Subjetiva dos Avaliadores - Uma Avaliação Qualitativa

Além dos dados quantitativos sobre a legibilidade dos códigos gerados por LLMs, também coletamos justificativas dos participantes, totalizando 160 respostas. Essa análise qualitativa buscou compreender os critérios subjetivos que orientaram as escolhas, complementando a simples contagem de votos (Código A, Código B, Ambos ou Não Sei). Enquanto os resultados numéricos indicam a frequência de cada opção, as justificativas revelam os padrões de julgamento adotados e os fatores que influenciaram a percepção de legibilidade e qualidade do código.

A Tabela 5.21 e a Figura 5.7 mostram o número e a proporção de escolhas. O código A (original) recebeu 28 preferências (17,5%), o código B (refatorado) foi escolhido em 100 casos (62,5%), enquanto 27 avaliações (16,9%) indicaram ambos e 5 (3,1%) responderam "não sei".

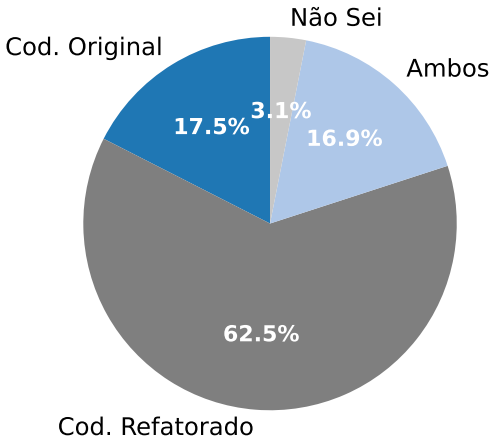

Figura 5.7: Distribuição de Respostas Por Opção

Nota: Esses dados se referem ao número total de avaliações coletadas, e não ao resultado do confronto direto entre duas avaliações, conforme explicado anteriormente. Apresentamos esses dados com o objetivo de contextualizar os resultados que discutiremos a seguir.

Resposta	Quantidade
A(Código Original)	28
B(Código Refatorado)	100
Ambos	27
Não Sei	5

Tabela 5.21: Distribuição das Respostas por Opção

Na Tabela 5.22, apresentamos as categorias utilizadas para agrupar as percepções dos avaliadores com base nas justificativas fornecidas em cada comparação entre os códigos.

Além disso, a contagem por categoria, segmentada por tipo de escolha (A, B, Ambos, Não Sei), permitiu observar quais critérios pesaram mais fortemente nas preferências por determinado código. Por exemplo, códigos do tipo B foram frequentemente justificados com base em categorias como ”organização”, ”princípios de projeto”e ”simplicidade”, o que pode indicar uma percepção de que códigos refatorados apresentam melhores práticas.

Tabela 5.22: Categorias utilizadas na análise qualitativa das justificativas

Categoria	Descrição
Clareza / Legibilidade	Comentários que destacam a facilidade de leitura do código, seja pela simplicidade sintática ou pela estrutura visual.
Nomenclatura	Justificativas que mencionam a escolha de nomes de variáveis ou funções como fator decisivo.
Organização	Referências à estrutura do código, modularidade, separação de responsabilidades ou tamanho das funções.
Reutilização / Eficiência	Quando a justificativa menciona o reaproveitamento de trechos de código ou eficiência na implementação.
Simplicidade / Estilo	Comentários sobre códigos mais enxutos, diretos, ou que evitam redundâncias. Também inclui menções à conformidade com guias de estilo, como a PEP8, especialmente quando a justificativa envolve formatação, clareza visual ou concisão do código..
Princípios de Projeto	Citações explícitas ou implícitas de boas práticas como SRP (Princípio da Responsabilidade Única), validação antecipada, separação de preocupações etc.
Similaridade	Justificativas que apontam que os dois códigos são muito parecidos.
Diferença Irrelevante	Quando a diferença apontada pelo avaliador não afeta significativamente a legibilidade ou a qualidade.
Decisão Neutra	Quando o participante afirma que ambos os códigos são equivalentes ou que não consegue escolher um melhor.

Para avaliar a justificativa apresentada por um participante, buscamos identificar a percepção que se alinha a uma ou mais categorias pré-definidas. Por exemplo, na seguinte justificativa:

> *"O código **B**, porque ele divide as responsabilidades, além de tornar claro o objetivo das execuções de cada parte do código através das declarações de cada função. No código **A** é necessário ler o código como um todo para entender o contexto."*

É possível extrair com clareza a intenção de destacar aspectos como *clareza e legibilidade*, ao mencionar que o código **B** "torna claro o objetivo das execuções de cada parte". Também se percebe a organização do código quando o participante aponta a "divisão de responsabilidades", indicando uma estrutura mais modular. Além disso, a reutilização pode ser inferida de forma indireta, uma vez que a modularização, ao separar funcionalidades em funções específicas, favorece a reutilização futura. Por fim, há uma referência direta a um princípio de projeto, o da responsabilidade única, ao mencionar explicitamente a "divisão de responsabilidades", o que reforça a adoção de boas práticas de design.

A Tabela 5.23 apresenta a distribuição das justificativas fornecidas pelos avaliadores, classificadas de acordo com as categorias previamente definidas. Cada justificativa foi associada a uma escolha específica: **código A, código B, ambos ou não sei**. Ao todo, foram analisadas 160 justificativas provenientes dos formulários, com a possibilidade de múltiplas categorias por justificativa.

Nota: o código A refere-se ao código original, enquanto o código B representa a versão refatorada.

Tabela 5.23: Distribuição das Justificativas por Categoria e escolha dos Avaliadores

Categoria	**A(Original)**	**B(Refatorado)**	**Ambos**	**Não Sei**	**Total**
Clareza/Legibilidade	25	99	20	2	146
Nomenclatura	13	42	5	1	61
Organização	12	71	2	0	85
Reutilização/Eficiência	1	30	2	0	33
Princípios de Projeto	10	78	3	1	92
Simplicidade/Estilo	17	85	12	1	115
Similaridade	4	11	24	1	40
Diferença Irrelevante	6	2	47	5	60
Total	88	418	115	11	**632**

As categorias mais frequentes foram *Clareza/Legibilidade* (146 ocorrências), *Simplicidade/Estilo* (115) e *Princípios de Projeto* (92), indicando que os avaliadores valorizam a compreensão do código e boas práticas de desenvolvimento. A maioria das justificativas dessas categorias foi atribuída ao código B, refatorado pelo LLM, sugerindo preferência por soluções mais legíveis, simples e estruturadas (Martin, 2008; Fowler, 2019).

Em contraste, *Similaridade* (40) e *Diferença Irrelevante* (60) apareceram em respostas em que avaliadores não perceberam diferenças significativas ou consideraram-nas pouco relevantes, evidenciando que, em alguns casos, os modelos geraram códigos com distinções sutis, dificultando a avaliação objetiva.

A seguir, apresentamos alguns exemplos de justificativas atribuídas a diferentes categorias, a fim de ilustrar a classificação realizada e dar maior transparência ao processo analítico. Diversos avaliadores destacaram a clareza como fator determinante em sua decisão. Um dos participantes que optou pelo código B (refatorado) justificou sua escolha da seguinte forma:

> "Código B (refatorado) é mais legível devido à utilização de nomes claros para variáveis e à organização dos blocos de forma que facilita a leitura e compreensão."

Outro avaliador também preferiu o código B, enfatizando a divisão de responsabilidades e a clareza nas funções:

> *"O código B porque ele divide as responsabilidades, além de tornar claro o objetivo das execuções de cada parte do código através das declarações de cada função; no código A (original), é necessário ler o código como um todo para entender o contexto."*

Essas justificativas reforçam a percepção de que práticas como a separação de responsabilidades e a escolha de nomes adequados contribuem para a legibilidade e compreensibilidade do código, aspectos valorizados por esses avaliadores.

Por outro lado, um dos participantes optou pelo código A (original):

> *"'A' é geralmente considerado mais legível devido aos nomes de variáveis mais descritivos, apesar de ambos os códigos serem funcionais e seguirem a sintaxe correta do Python."*

Esse comentário evidencia que, mesmo em códigos funcionais e semanticamente equivalentes, fatores subjetivos como a percepção dos nomes podem influenciar a decisão do avaliador.

Além disso, houve respostas neutras, como no exemplo abaixo:

> *"Tanto o Code A quanto o Code B apresentam semelhanças próximas."*

Esse tipo de resposta foi classificado como decisão neutra, pois o participante não apontou preferências claras ou critérios distintos para justificar sua escolha.

Resumo Seção 5.4

Realizamos uma avaliação com participantes humanos para verificar se refatorações feitas por LLMs melhoram a legibilidade e corrigem problemas de manutenibilidade. Cada avaliador comparou pares de códigos e justificou sua escolha; em caso de discordância, um terceiro avaliador realizou o desempate. A maioria dos participantes possui experiência em desenvolvimento e formação acadêmica consolidada, embora a prática em Python varie, reforçando a confiabilidade dos resultados. Houve clara preferência pelos códigos refatorados, percebidos como mais legíveis, organizados e alinhados a boas práticas. Critérios como clareza, simplicidade, organização, princípios de projeto, nomenclatura e modularidade influenciaram as escolhas. Respostas neutras ocorreram quando as diferenças eram sutis, indicando códigos funcionalmente equivalentes. Assim, as refatorações feitas pelos LLMs tendem a produzir códigos mais compreensíveis e estruturados.

5.5 Considerações Finais

Neste capítulo apresentamos os resultados do experimento, avaliando a eficácia dos LLMs na refatoração de código Python com problemas de manutenibilidade. Comparamos o desempenho dos modelos e, por meio de testes estatísticos, confirmamos a superioridade do few-shot em relação ao zero-shot. Analisamos ainda as refatorações com base em regras de manutenibilidade, identificando padrões de erros e técnicas empregadas, o que permitiu delinear o perfil de cada modelo. Por fim, a avaliação humana sobre a legibilidade dos códigos complementou a análise automática, oferecendo uma visão mais completa da qualidade das refatorações.

Capítulo 6

Discussão

Neste capítulo, discutimos os resultados obtidos neste trabalho, com base nas questões de pesquisa apresentadas no início do Capítulo 4. A análise concentra-se na comparação da eficácia dos modelos de linguagem na refatoração de código Python, avaliando sua capacidade de resolver problemas de manutenibilidade, melhorar a legibilidade e obter sucesso nos testes automatizados.

- A Seção 6.1 apresenta uma visão geral dos resultados;
- A Seção 6.2 traz uma comparação dos resultados com estudos anteriores;
- A Seção 6.3 apresenta tendências entre os modelos;
- A Seção 6.4 discute os impacto das técnicas de prompts;
- A Seção 6.5 apresenta os tipos e refatorações realizadas;
- A seção 6.6 traz discussão sobre as percepção humana;
- A seção 6.7 discute limitações e implicações práticas dos LLMs;
- A seção 6.8 mostra alguns exemplos de refatoração;
- A seção 6.9 apresenta exemplos de refatoração;
- A seção 6.10 considerações finais;

6.1 Visão Geral dos Resultados

Os resultados do experimento indicam que os LLMs apresentam potencial relevante para a refatoração de código Python com problemas de manutenibilidade, produzindo versões geralmente mais legíveis e alinhadas às boas práticas de desenvolvimento. Entre os quatro modelos avaliados, três apresentaram desempenhos semelhantes tanto em zero-shot quanto em few-shot, sugerindo que, no contexto deste

estudo, o desempenho entre eles foi estável independentemente da técnica empregada. O modelo LLaMA, embora tenha obtido resultados ligeiramente mais modestos, também foi capaz de realizar refatorações adequadas. Esses achados foram corroborados pela análise estatística, que não identificou diferenças significativas entre os modelos em cada técnica. Assim, embora haja uma tendência favorável a três modelos, não existem evidências robustas para afirmar superioridade estatística com o tamanho atual da amostra.

Apesar de apresentarem taxas de sucesso promissoras, chegando a ultrapassar 60% na abordagem few-shot, os LLMs também cometeram erros relevantes. Entre eles, destacam-se a introdução de novos problemas de manutenibilidade, falhas de execução, inconsistências na lógica de testes e casos em que, embora o código fosse executado normalmente, o code smell não havia sido resolvido. Esses aspectos indicam que, embora os modelos demonstrem potencial, ainda não estão prontos para uso em escala industrial sem supervisão humana criteriosa.

De modo geral, os modelos avaliados mostraram-se capazes de aplicar refatorações úteis, mas também introduziram erros de execução e apresentaram variações de comportamento. Esses resultados fornecem uma visão inicial do desempenho dos LLMs e orientam as análises mais detalhadas apresentadas nas seções seguintes, além de fundamentarem a discussão crítica que será desenvolvida nas próximas seções.

6.2 Comparação com estudos anteriores

Liu et al. (2024) avaliaram recentemente o uso de LLMs na refatoração de código Java, considerando apenas GPT-4 e Gemini, mas com objetivo semelhante ao nosso. Nesse estudo, a refatoração era considerada bem-sucedida quando o código continuava passando nos testes unitários e quando as soluções geradas eram julgadas superiores às de humanos. Em nosso caso, ampliamos os critérios, exigindo também ausência de erros de sintaxe, execução correta, não introdução de novos problemas e solução efetiva do problema identificado. Além disso, expandimos o escopo para quatro modelos (Copilot Chat, LLaMA, DeepSeek e Gemini), testamos *zero-shot* e *few-shot* e incluímos avaliação humana de legibilidade.

Os resultados de Liu et al. (2024) foram promissores (63,6% para GPT-4 e 56,2% para Gemini), próximos aos nossos achados, em que o desempenho variou de 44% (LLaMA, *zero-shot*) a 64,67% (Gemini, *few-shot*), com destaque também para Copilot Chat (63,33%) e DeepSeek (64,00%). De modo geral, a técnica *few-shot* mostrou ganhos consistentes. Apesar de resultados semelhantes, as pesquisas diferem quanto à linguagem (Java vs. Python), ferramentas (SonarQube neste estudo), número de instâncias avaliadas (180 contra 1.200) e complexidade dos códigos. Assim, embora limitados, ambos os trabalhos reforçam a eficácia promissora das LLMs na refatoração de código.

A pesquisa de Al Madi (2022) investigou a legibilidade de códigos gerados pelo GitHub Copilot em comparação com códigos escritos por humanos, aproximando-

se do nosso estudo por também avaliar a legibilidade de soluções produzidas por LLMs. Diferentemente de nós, os autores ampliaram o escopo ao considerar não apenas a clareza do código, mas também fatores como viés de automação e nível de revisão crítica aplicada às sugestões. O experimento contou com 21 programadores, envolvendo análise estatística de legibilidade, rastreamento ocular durante a inspeção visual e o uso de uma ferramenta própria para coleta desses dados.

Os resultados indicaram que o código gerado por LLMs apresenta legibilidade e complexidade comparáveis aos de humanos. Nosso estudo reforça essa conclusão, já que mais de 75% dos avaliadores consideraram os códigos refatorados mais legíveis que os originais. Contudo, Al Madi (2022) mostraram, via rastreamento ocular, que os participantes dedicaram menos atenção ao código gerado automaticamente, sugerindo excesso de confiança ou menor engajamento na revisão. Assim, embora os LLMs sejam capazes de produzir códigos legíveis, ambos os trabalhos convergem ao destacar a importância da revisão humana antes da adoção plena das sugestões geradas por IA

O estudo de Nunes et al. (2025) avaliou a refatoração de código Java com LLMs, analisando Copilot Chat e LLaMA 3.1, com técnicas de *zero-shot* e *few-shot*. Assim como em nosso trabalho, o objetivo foi medir a eficácia dos modelos em corrigir problemas de manutenibilidade, utilizando SonarQube e avaliação humana de legibilidade. Nosso estudo ampliou o número de modelos (Copilot Chat 4.0, LLaMA 3.3 70B Instruct, DeepSeek V3 e Gemini 2.5 Pro), o número de instâncias (1.200 contra 127–150) e aplicou *few-shot* em todos os modelos. Nos resultados, Nunes et al. (2025) reportaram 30% de sucesso do LLaMA em *zero-shot* e 32,29% em *few-shot*, valores inferiores aos nossos: LLaMA 44% (*zero-shot*) e 55,33% (*few-shot*), Copilot Chat 52,67% e 63,33%, respectivamente. Diferenças de linguagem (Java vs Python), número de instâncias e regras de manutenibilidade explicam, em parte, essas variações.

Apesar das diferenças, ambos os estudos indicam limitações no uso de LLMs, como falhas em testes, introdução de novos problemas e situações em que o problema original não foi resolvido. Os resultados reforçam a necessidade de validação humana, mostrando que LLMs têm potencial para apoiar a refatoração automática, mas exigem supervisão crítica.

Cordeiro et al. (2025) avaliaram o uso de LLMs, como o StarCoder2, para automatizar a refatoração de código, buscando melhorias estruturais sem alterar a funcionalidade. Semelhante ao nosso trabalho, avaliamos a eficácia de LLMs na preservação da integridade funcional, mas com quatro modelos. O estudo adotou abordagem qualitativa, baseada em revisão de literatura e reflexão sobre limitações e oportunidades em IDEs, sem avaliação quantitativa. Foram observadas limitações semelhantes às nossas: (i) compreensão contextual restrita, (ii) risco de alucinações, e (iii) dependência de prompts claros.

Nosso estudo reforça a importância de prompts mais elaborados (few-shot), que melhoram o desempenho dos modelos, evidenciando que, apesar do potencial, am-

bos os trabalhos apontam para a necessidade de revisão humana e refinamento das instruções para resultados mais confiáveis.

O estudo de AlOmar et al. (2024) se assemelha ao nosso ao avaliar o uso de LLMs na refatoração de código. Assim como em nossa pesquisa, eles verificaram que prompts detalhados e contextuais (*few-shot*) resultam em desempenho superior, mostrando que a interação do desenvolvedor com o modelo impacta a qualidade da refatoração. Há diferenças metodológicas: AlOmar et al. (2024) adotou abordagem qualitativa, analisando interações reais entre usuários e ChatGPT, enquanto nosso estudo é quantitativo, avaliando execução, testes automatizados, ausência de novos problemas e legibilidade. O estudo também identificou problemas recorrentes nas respostas do modelo, como variáveis não definidas, falhas de documentação, uso inadequado de recursos, tratamento incompleto de exceções e questões de segurança, alinhando-se às limitações observadas em nosso experimento.

Ambos os trabalhos convergem na conclusão de que, apesar do potencial dos LLMs, a revisão humana é essencial. Nosso estudo comprova experimentalmente os benefícios de prompts elaborados, enquanto AlOmar et al. (2024) mostra que desenvolvedores já aplicam essa prática empiricamente, reforçando a relevância prática de nossos achados.

O estudo de Ságodi et al. (2024) avaliou LLMs na geração de código e na qualidade das soluções produzidas, enquanto nosso trabalho foca na refatoração de código existente para resolver problemas de manutenibilidade sem comprometer a funcionalidade. Ambos os estudos incorporam avaliações humanas e analisam aspectos como code smells, embora os critérios de sucesso sejam diferentes: eles consideram tarefas específicas e presença de falhas, enquanto nós avaliamos execução correta, testes automatizados, ausência de novos problemas e legibilidade. Quanto aos modelos, Ságodi et al. (2024) testaram ChatGPT-3.5, GitHub Copilot, Gemini 1.5-Pro e AlphaCode; nosso estudo incluiu Copilot Chat 4.0, Gemini 2.5 Pro, LLaMA 3.3 70B Instruct e DeepSeek V3, avaliando também estratégias de prompting (zero-shot e few-shot). As linguagens diferem: eles usaram C++, Java e Python; nós trabalhamos apenas com Python.

Apesar das diferenças, ambos reforçam o potencial dos LLMs para apoiar desenvolvedores e a importância de avaliação humana e métricas objetivas. Os resultados sugerem que a estratégia few-shot tende a melhorar o desempenho dos modelos em ambos os contextos: no estudo deles, ChatGPT (GPT-4 Turbo) alcançou 87,2% no HumanEval, Gemini 1.5-Pro 74,9%, AlphaCode 45–54% no Codeforces e GitHub Copilot 75,7% no LeetCode; em nosso estudo, Copilot Chat, Gemini e DeepSeek atingiram 63,33%, 64% e 64,67%, respectivamente, na estratégia few-shot, indicando a eficácia dessa abordagem também em refatoração de código.

Pomian et al. (2024) avaliaram o impacto dos LLMs na refatoração de código com a ferramenta EM-Assist, integrada ao IntelliJ e baseada no GPT-3.5-turbo, focando na extração de métodos. Assim como nosso trabalho, realizou experimentos com projetos reais e incluiu avaliação humana da legibilidade e qualidade do código.

Difere do nosso estudo por seu foco em uma técnica específica, enquanto analisamos diversos problemas de manutenibilidade identificados pelo SonarQube, testando quatro modelos de LLMs e estratégias de prompting (*zero-shot* e *few-shot*). Os resultados de Pomian et al. (2024) mostraram recall de 53,4% em 1.752 refatorações, superior à versão anterior (39,4%), e alta aceitação entre desenvolvedores (94,4%). Nosso estudo também apresentou taxas expressivas de sucesso, especialmente com prompting *few-shot*.

Ambos os trabalhos identificaram limitações, como alucinações e problemas introduzidos inadvertidamente, reforçando a necessidade de supervisão humana. Juntos, evidenciam o potencial das LLMs para apoiar a refatoração e a manutenção de software, assim como os desafios a serem considerados para uso confiável em ambientes reais.

6.3 Tendências de desempenho entre modelos

Embora este estudo não aponte a superioridade definitiva de nenhum modelo, os resultados sugerem uma tendência de desempenho ligeiramente melhor nos modelos Gemini, Copilot Chat e DeepSeek em comparação ao LLaMA. Essa diferença deve ser interpretada com cautela, pois não se mostrou estatisticamente significativa, possivelmente em função do tamanho limitado da amostra utilizada. Assim, os achados sugerem que, no contexto atual, não há um modelo claramente dominante para tarefas de refatoração de código Python com foco em manutenibilidade, embora o LLaMA tenha apresentado desempenho ligeiramente inferior aos demais nos valores numéricos.

Diante desse cenário, a escolha do modelo mais adequado tende a depender menos de diferenças intrínsecas de desempenho e mais de fatores externos, como o contexto de aplicação, a facilidade de integração com fluxos de trabalho existentes, a disponibilidade da ferramenta ou a preferência da equipe de desenvolvimento. Considerando a constante evolução dos LLMs, é provável que a semelhança de desempenho observada entre os modelos neste estudo se mantenha ou até se torne mais consistente em versões futuras. Em termos práticos, os resultados reforçam que os LLMs podem atuar como ferramentas promissoras de apoio à refatoração, embora ainda careçam de evidências robustas que comprovem superioridade consistente em ambientes reais de desenvolvimento de software.

6.4 Impacto das Técnicas de Prompts

Ao avaliarmos as técnicas de prompt *zero-shot* e *few-shot*, buscamos verificar se a inclusão de exemplos no prompt (*few-shot*) contribui para aprimorar o desempenho dos LLMs na refatoração de código Python com problemas de manutenibilidade. Os resultados indicam que, ao analisar valores absolutos em comparações individuais, a técnica *few-shot* tende a gerar refatorações mais eficazes, com menor incidência de

erros, em comparação à abordagem *zero-shot*, que apresentou desempenho ligeiramente inferior em situações semelhantes.

Essas diferenças foram confirmadas quando aplicados testes estatísticos sobre os resultados agregados dos modelos, embora não tenham se mostrado significativas nas análises individuais. Em consonância com estudos anteriores (Brown et al., 2020), esses achados sugerem que LLMs tendem a apresentar melhor desempenho quando prompts mais refinados, como no *few-shot*, são empregados, evidenciando o impacto positivo de técnicas de estímulo estruturadas na qualidade das refatorações.

Na prática, o estudo sugere que a adoção de técnicas de prompts mais consistentes e contextualizadas, como o *few-shot*, pode melhorar os resultados das refatorações realizadas por LLMs em ambientes reais de desenvolvimento (Brown et al., 2020). Apesar desse desempenho promissor, os modelos ainda cometem erros relevantes e não são totalmente autônomos. A escolha do modelo e o design do prompt podem influenciar significativamente a eficácia das refatorações, incluindo a estratégia de intervenção adotada pelo LLM.

6.5 Tipos de refatorações realizadas

A análise das técnicas de refatoração aplicadas pelos LLMs revela elementos importantes sobre as capacidades e limitações desses modelos ao lidar com problemas de manutenibilidade em código Python. Observou-se que as técnicas mais frequentes foram renomeação de variáveis, extração de métodos e consolidação de condicionais, indicando que os modelos tendem a priorizar ajustes estruturais de maior impacto na legibilidade e manutenção do código.

Mesmo apresentando erros significativos ou ocasionalmente respostas incorretas, os LLMs conseguem identificar pontos críticos de melhoria, contribuindo para refatorações úteis na prática. Modelos como DeepSeek se destacaram em refatorações estruturais no modo zero-shot, enquanto Gemini e Copilot mostraram melhor desempenho em tarefas de nomenclatura e remoção de TODOs no modo few-shot, evidenciando que a inclusão de exemplos no prompt aumenta a consistência e a aplicação de boas práticas.

Na prática, esses achados sugerem que LLMs podem apoiar tarefas de refatoração frequentes e estruturais, economizando tempo e padronizando melhorias de código. Entretanto, a supervisão humana continua essencial, principalmente em técnicas menos comuns ou em sistemas críticos, garantindo que erros relevantes sejam identificados antes da aplicação em produção.

6.6 Percepções dos Desenvolvedores

A análise das percepções dos desenvolvedores mostrou que, de forma geral, as refatorações geradas pelos LLMs foram vistas como um ganho em termos de legibilidade.

Muitos participantes destacaram atributos como clareza, simplicidade, organização estrutural e aderência a princípios de projeto como diferenciais positivos. Isso sugere que, além de corrigirem problemas de manutenibilidade, os modelos também podem contribuir para tornar o código mais fácil de compreender e mais próximo de boas práticas de desenvolvimento.

Por outro lado, nem todas as refatorações foram recebidas de maneira positiva. Uma parte dos avaliadores considerou o código original mais legível, enquanto alguns não notaram diferença relevante entre a versão original e a refatorada. Esses casos mostram que os LLMs ainda podem produzir trechos com pouca clareza ou até mesmo soluções que se afastam dos padrões de desenvolvimento esperados.

Outro ponto importante diz respeito ao perfil dos avaliadores. Como a maioria tinha experiência limitada com Python, é possível que detalhes mais sutis de legibilidade tenham passado despercebidos. Para reduzir possíveis vieses, nos casos de empate foi realizada uma avaliação adicional por um terceiro desenvolvedor mais experiente, garantindo maior consistência na análise.

Ao cruzar os dados quantitativos e qualitativos, foi possível obter uma visão mais completa dos resultados. A predominância da escolha pelo código refatorado (81,25%) se conecta diretamente às justificativas mais recorrentes, como maior clareza, simplicidade e melhor organização. Já entre os que preferiram o código original (17,5%), o argumento principal foi que certas refatorações diminuíram a legibilidade ou fugiram dos padrões esperados. Nos poucos casos de equivalência, a percepção foi de que a refatoração não trouxe mudanças significativas para a leitura do código. Esses resultados indicam que, embora os LLMs tendam a favorecer a legibilidade, os ganhos não são uniformes e variam tanto em função da qualidade da refatoração quanto da experiência do avaliador com a linguagem.

De forma prática, esses achados trazem duas lições importantes. Em primeiro lugar, os LLMs podem ser usados como aliados em tarefas de manutenção rotineira, oferecendo sugestões que facilitam a leitura e incentivam boas práticas. Em segundo lugar, é fundamental manter a revisão humana, sobretudo em trechos críticos ou complexos, onde escolhas inadequadas podem comprometer a qualidade ou a segurança do software.

Em resumo, a avaliação humana reforça o potencial dos LLMs como assistentes no desenvolvimento, mas também deixa claro que o uso dessas ferramentas exige supervisão especializada e integração cuidadosa ao fluxo de trabalho, para que os ganhos de legibilidade se convertam em resultados consistentes e confiáveis.

6.7 Limitações e Implicações Práticas

Este estudo apresenta algumas limitações que precisam ser consideradas. Identificar os erros cometidos pelos LLMs foi fundamental para entendermos como esses modelos se comportam ao refatorar código, mesmo quando buscam corrigir problemas

de manutenibilidade. Esses achados nos mostram que, embora os LLMs consigam gerar refatorações úteis e, muitas vezes, melhorar a legibilidade do código, eles ainda não oferecem uma solução definitiva para o processo de manutenção de software. .

6.7.1 Limitações dos LLMs

Os resultados indicam que, em diversas situações, os modelos conseguiram melhorar a estrutura do código e corrigir problemas de manutenibilidade. Entretanto, surgiram erros que não podem ser ignorados, como falhas de execução (NameError, TypeError, AttributeError), falhas em testes (AssertionError), a introdução de novos problemas de manutenibilidade e casos em que a refatoração não solucionou completamente o problema original. Isso evidencia que, embora os modelos consigam compreender partes do código, nem sempre captam o contexto completo, podendo gerar consequências inesperadas que afetam a funcionalidade, a confiabilidade e a integridade do software.

Na prática, o código gerado pelos LLMs pode parecer organizado e funcional, mas ainda apresenta riscos que vão desde pequenas falhas de legibilidade até problemas que comprometem o funcionamento do programa. Algumas refatorações nem sequer resolveram o problema inicial e, em certos casos, chegaram a introduzir novas dificuldades. Esses achados reforçam a importância de manter a revisão humana e a execução de testes automatizados como parte essencial do fluxo de desenvolvimento, garantindo que o uso dos LLMs seja seguro e confiável.

Por outro lado, os LLMs se mostraram bastante úteis em tarefas mais simples ou repetitivas, como renomeação de variáveis, extração de métodos e reorganização de trechos de código. Eles podem acelerar o trabalho do desenvolvedor e servir como um bom ponto de partida, principalmente quando utilizados de forma consciente e integrada a processos de revisão e testes.

6.7.2 Alucinações

Durante o experimento, observamos que os quatro modelos de LLM avaliados, independentemente do tipo de prompt utilizado, apresentaram alucinações. Embora conseguissem resolver problemas de manutenibilidade, os modelos ainda geravam erros persistentes, muitas vezes sutis e difíceis de detectar mesmo com testes automatizados. A inclusão de exemplos de código com problemas de manutenibilidade (*few-shot*) contribuiu para reduzir a frequência dessas alucinações, evidenciando a importância de técnicas de prompting mais refinadas e contextualizadas para aumentar a confiabilidade das refatorações.

Em alguns casos, os LLMs corrigiam problemas ou refatoravam o código, mas não consideravam o contexto completo do método, o que podia afetar o resultado final. Situações aparentemente simples, como renomear uma variável para seguir o padrão *snake_case* do Python, poderiam resultar em inconsistências se o modelo não atualizasse todas as ocorrências da variável dentro do método. Esse comportamento

indica que, embora os LLMs apresentem potencial para auxiliar na refatoração, suas decisões nem sempre são totalmente confiáveis quando analisadas em um contexto mais amplo.

Esses achados evidenciam uma limitação crucial dos LLMs: mesmo quando melhoram a estrutura e a legibilidade do código, podem introduzir inconsistências ou alterações inesperadas no comportamento. Além disso, o padrão observado reforça que a eficácia das refatorações não pode ser avaliada apenas pela aparência do código ou pela correção sintática.

Alguns exemplos de alucinações identificados neste estudo podem ser consultados no Apêndice G. Esses casos demonstram a necessidade de integração entre modelos automatizados, boas práticas de programação e mecanismos de validação contínua, destacando a importância de uma abordagem crítica e interpretativa ao utilizar LLMs na refatoração de código.

6.7.3 Implicações Práticas

O estudo revela que os LLMs têm um potencial promissor para apoiar a refatoração de código Python com problemas de manutenibilidade. Além de resolverem esses problemas, os modelos foram capazes de produzir versões mais legíveis, claras e bem estruturadas do código, o que pode trazer benefícios práticos para desenvolvedores e equipes de software que buscam aumentar a qualidade e a manutenibilidade de seus sistemas.

Na prática, este estudo sugere que, ao solicitar a refatoração de código aos LLMs, é fundamental validar os resultados gerados. Nem sempre as alterações correspondem ao que realmente se deseja ou preservam o comportamento original do código. Por isso, é imprescindível complementar as refatorações automatizadas com testes, execução do código e revisão humana, como realizado neste experimento. Diante disso, o estudo aponta que a adoção de LLMs por profissionais de engenharia de software requer cautela, especialmente em sistemas críticos, já que esses modelos ainda cometem erros que podem comprometer sua aplicação no dia a dia. Apesar do potencial para apoiar tarefas de refatoração e contribuir para a melhoria da qualidade do código, o uso direto, sem mecanismos de validação, pode introduzir riscos relevantes, como inconsistências ou alterações indesejadas no comportamento do sistema.

O grande aprendizado deste estudo é que essas ferramentas ainda não substituem a supervisão humana, mas podem atuar como ferramentas auxiliares em tarefas de programação, especialmente na refatoração de códigos com problemas de manutenibilidade. Quando utilizadas de forma estratégica, contribuem para aumentar a produtividade e a consistência do código. Integrá-las aos fluxos de trabalho existentes, aproveitando a agilidade que oferecem sem abrir mão da confiabilidade, mostra-se como o caminho mais seguro e promissor.

6.8 Limitações e Ameaças à Validade

Esta pesquisa apresenta algumas limitações que podem afetar os resultados e a confiabilidade do estudo. Por isso, é importante relatramos de forma clara as ameaças à validade, a fim de contextualizar adequadamente as nossas conclusões obtidas.

6.8.1 Ameaças Internas

As ameaças internas estão relacionadas a fatores que podem comprometer a execução do estudo, influenciando diretamente os resultados.

Em relação à validade interna, destaca-se que apenas quatro modelos de LLM foram avaliados, o que pode limitar a generalização dos achados para outros modelos. Além disso, a análise foi conduzida no nível de método, e análises em contextos mais amplos, como o nível de classe, poderiam fornecer insights adicionais. O conjunto de regras aplicado também foi restrito, com 12 regras selecionadas aleatoriamente a partir de um conjunto predefinido, o que pode influenciar os resultados. Variações nos prompts ou nos exemplos *few-shot* também podem afetar os resultados; para mitigar essa influência, os prompts foram validados por dois autores adicionais. A avaliação humana, embora padronizada, pode introduzir subjetividade, que foi parcialmente mitigada pela participação de um terceiro avaliador. Por fim, apesar de os scripts de automação terem sido testados várias vezes, erros residuais podem ainda impactar os resultados.

6.8.2 Ameaças Externas

As ameaças externas dizem respeito a fatores que podem limitar a generalização dos resultados.

Quanto à validade externa, o foco exclusivo em Python limita a generalização dos achados para outras linguagens. Para contornar essa limitação, foram selecionadas regras comuns a outras linguagens, sendo oito das doze compatíveis com Java ou JavaScript. Além disso, a análise foi realizada em dez projetos específicos, cada um com 15 problemas diversificados, buscando reduzir limitações na extrapolação para outros contextos de programação.

6.8.3 Validade de Construto e Conclusão

A validade de construto refere-se à adequação entre o que foi planejado e o que foi efetivamente avaliado. Já a validade de conclusão está relacionada ao quanto as inferências feitas a partir dos resultados são consistentes e justificáveis. Assim, é necessário avaliar se os instrumentos e métricas utilizados realmente medem os aspectos pretendidos.

Em relação à validade de construto e de conclusão, o estudo se restringiu a problemas de manutenibilidade, sem considerar outras dimensões da qualidade de software,

como segurança ou desempenho. A diversidade de projetos e regras utilizadas contribuiu para mitigar, em parte, essas limitações, mas é importante reconhecer que os resultados se aplicam especificamente ao escopo investigado

6.8.4 Confiabilidade

A confiabilidade refere-se ao grau em que este estudo pode ser replicado em outros contextos e obter resultados semelhantes. Fatores como a documentação detalhada do processo experimental e a definição clara dos critérios de análise contribuem para aumentar a confiabilidade da pesquisa.

Por fim, no que se refere à confiabilidade, o estudo buscou garantir a capacidade de replicação em outros contextos. Entretanto, a replicabilidade depende das versões específicas das ferramentas utilizadas, incluindo LLMs e SonarQube, e como os modelos proprietários passam por atualizações frequentes, pode não ser possível reproduzir o experimento exatamente nas mesmas condições. Para reduzir esse risco, foram disponibilizados documentação completa, versões dos projetos, scripts de automação, códigos originais e refatorados, bem como os formulários e respostas coletadas na avaliação humana, permitindo transparência e maior consistência na replicação do estudo.

6.9 Exemplos de Refatoração

No Apêndice G, apresentamos exemplos de refatorações bem-sucedidas, assim como casos em que ocorreram erros, incluindo alucinações. Incluímos esses exemplos para destacar como os LLMs se comportaram ao tentar resolver problemas de manutenibilidade, mostrando o problema identificado e a solução gerada. Em seguida, discutimos situações em que os modelos falharam, introduzindo erros ou produzindo soluções alucinadas.

6.10 Considerações Finais

Neste capítulo, discutimos os principais achados, evidenciando que LLMs podem apoiar a refatoração de código Python, produzindo versões mais legíveis e aderentes a boas práticas. Copilot Chat, Gemini e DeepSeek apresentaram desempenho consistente, enquanto LLaMA foi ligeiramente inferior, sem diferenças estatisticamente significativas. A técnica few-shot melhorou a qualidade e reduziu erros. As refatorações mais frequentes envolveram renomeação de variáveis, extração de métodos e consolidação de condicionais. A avaliação humana confirmou maior legibilidade na maioria dos códigos, embora algumas inconsistências persistissem. As principais limitações incluíram falhas de execução, novos problemas e alucinações, reforçando a necessidade de supervisão humana. Em síntese, os LLMs mostram potencial para acelerar e padronizar melhorias, mas exigem revisão crítica, prompts adequados e testes rigorosos para uso confiável.

Capítulo 7

Considerações Finais

Neste capítulo, apresentamos as conclusões deste estudo, assim como as suas contribuições e trabalhos futuros.

- A Seção 7.1 apresenta as conclusões do estudo;
- A Seção 7.2 descreve as contribuições finais da pesquisa.
- A Seção 7.3 propõe direções para trabalhos futuros;

7.1 Conclusões

Este estudo investigou a eficácia dos Modelos de Linguagem de Grande Escala (LLMs) na refatoração de código Python, com foco na correção de métodos com problemas de manutenibilidade detectados automaticamente pelo SonarQube. O objetivo foi avaliar o desempenho de diferentes modelos, como Copilot Chat 4o, Llama 3.3 70B Instruct, DeepSeek V3 e Gemini 2.5 Pro, na geração de refatorações que eliminem tais problemas, mantendo a funcionalidade original e promovendo maior legibilidade e qualidade estrutural.

A análise empírica, baseada em 1.200 instâncias de refatoração, revelou que os LLMs possuem um potencial promissor para propor soluções tecnicamente válidas na maioria dos casos. Modelos como DeepSeek V3, Copilot Chat 4o e Gemini 2.5 Pro obtiveram as maiores taxas de sucesso, especialmente sob estratégias de prompting mais elaboradas, como o *few-shot*. No entanto, os testes estatísticos por modelo não indicaram diferenças significativas entre as técnicas de prompting, sugerindo que a vantagem observada pode não ser estatisticamente robusta em análises segmentadas.

Por outro lado, a análise conjunta dos resultados, considerando todos os modelos, apontou uma diferença estatisticamente significativa a favor do *few-shot*, evidenciando que estratégias de prompting mais refinadas tendem a melhorar o desempenho geral dos modelos. Essa diferença pode ser atribuída ao maior poder estatístico

obtido com a agregação dos dados, capaz de revelar padrões não perceptíveis em amostras menores.

Complementando a análise quantitativa, uma avaliação qualitativa com 32 participantes confirmou melhorias nos códigos refatorados quanto à clareza, organização e legibilidade. Essa percepção humana reforça o potencial dos LLMs para contribuir positivamente na qualidade e manutenibilidade do código.

Apesar dos resultados promissores, foram observadas limitações importantes, como falhas de execução, erros em testes automatizados, manutenção do problema original e introdução de novos problemas, inclusive alucinações. A comparação com estudos anteriores destaca os avanços deste trabalho, especialmente no que se refere à escala dos experimentos e à diversidade de modelos e técnicas avaliadas.

Concluímos que os LLMs apresentam grande potencial para auxiliar na refatoração e correção de problemas de manutenibilidade em código Python. Em consonância com estudos anteriores, esses modelos podem contribuir significativamente para a melhoria da qualidade do código, especialmente em práticas de refatoração estruturais. No entanto, foram identificadas limitações relevantes que ainda impedem o uso pleno dessas ferramentas em pipelines de produção, principalmente em sistemas críticos ou legados, pois erros não detectados podem ter consequências significativas sem a supervisão de um desenvolvedor. Com isso, ressaltamos a necessidade de acompanhamento humano, com validação criteriosa e revisão cuidadosa dos resultados para garantir maior segurança e confiabilidade na refatoração do código.

7.2 Contribuições

Este estudo traz as seguintes contribuições originais para a engenharia de software:

- **Análise do desempenho de LLMs na refatoração de código Python:** Este estudo avalia o desempenho de diferentes LLMs na refatoração de código Python, identificando acertos, falhas e limitações, como erros de nomenclatura, uso incorreto de atributos e falhas de execução. A partir desses achados, propõe-se orientar o uso responsável e eficaz dessas ferramentas, promovendo práticas mais seguras e alinhadas às boas práticas da engenharia de software.
- **Proposição de boas práticas de *prompting*:** Com base nas observações dos resultados, são oferecidas recomendações para a construção de *prompts* mais eficazes, com o objetivo de melhorar a qualidade das refatorações e reduzir a ocorrência de erros ao utilizar ferramentas de IA.
- **Identificação de limitações dos modelos:** O estudo detalha os problemas que os LLMs tiveram dificuldade em resolver, como métodos longos, complexos e análise de contexto global. Foram observadas falhas recorrentes, como *TypeError*, *AttributeError* e *NameError*. Registrar esses erros fornece subsídios para melhorar modelos e técnicas de *prompting*, além de orientar desenvolvedores para um uso mais consciente e eficaz dos LLMs em cenários reais.

- **Integração de avaliação empírica e humana:** Ao combinar análises quantitativas e qualitativas, o estudo evidencia que os modelos são capazes de gerar e refatorar código com qualidade razoável. Essa abordagem também contribui para aprimorar práticas de avaliação da legibilidade e compreensibilidade na engenharia de software.
- **Ampliação do escopo de avaliação:** Diferente de trabalhos anteriores, este estudo avalia um conjunto mais amplo de modelos e estratégias de *prompting*, aplicando as técnicas a um volume significativo de exemplos reais, com foco exclusivo na linguagem Python.
- **Abertura para futuras pesquisas:** O estudo sugere direções para novas investigações, como o aprimoramento de técnicas de *prompting*, a avaliação de novos modelos emergentes e o desenvolvimento de ferramentas que integrem avaliação automática e humana, promovendo o uso responsável e eficaz de LLMs na engenharia de software.
- **Catálogo de exemplos de refatoração assistida por LLMs:** Organização de um repositório contendo pares de código-fonte com problemas de manutenibilidade identificados via análise estática e suas respectivas refatorações geradas por diferentes LLMs. Esse catálogo pode ser utilizado como base para estudos comparativos, benchmarks e validação de abordagens futuras em engenharia de software orientada por IA.

Portanto, este trabalho busca contribuir para o avanço das pesquisas na interseção entre inteligência artificial e engenharia de software, fornecendo insights sobre o papel dos LLMs no aprimoramento da qualidade do código. Ao explorar novas técnicas de prompting e ampliar a gama de ferramentas testadas, esperamos oferecer subsídios para a criação de metodologias mais robustas e práticas para a refatoração assistida por IA, contribuindo para o desenvolvimento de sistemas de software mais sustentáveis e adaptáveis às demandas do mercado e da academia.

7.3 Trabalhos Futuros

Embora os resultados desta pesquisa evidenciem o potencial dos LLMs na refatoração de código Python, há diversas oportunidades para aprimorar os experimentos, ampliar os cenários de aplicação e desenvolver soluções mais robustas. Destacam-se algumas direções relevantes:

- **Expansão metodológica:** Avaliar o uso de técnicas como mais refininadas como, *chain-of-thought* (Wei et al., 2022; Wang et al., 2022), *step-by-step prompting* (Zhou et al., 2023), e abordagens *multi-turn* (Lu et al., 2020), que permitam ao modelo manter o contexto em múltiplas interações.
- **Aprimoramento dos modelos:** Investigar o desempenho de novas versões dos LLMs, incluindo modelos proprietários e open-source, bem como explorar *fine-*

tuning (Chen et al., 2021; Yu et al., 2024) específico para engenharia de software e agentes autônomos com validação automatizada do código.

- **Avaliação e métricas:** Incorporar métricas específicas para avaliar qualidade, legibilidade e manutenibilidade, como complexidade ciclomática e índice de manutenibilidade. Utilizar outras ferramentas de análise estática para ampliar a abrangência da avaliação.
- **Desenvolvimento de ferramentas e integração:** Criar plugins focados em refatoração orientada por LLMs, com suporte à análise de qualidade, validação automatizada e personalização. Integrar essas funcionalidades a pipelines de CI/CD para promover melhoria contínua do código.
- **Ampliação do escopo:** Avaliar outras categorias de problemas detectados pelo SonarQube, como bugs e vulnerabilidades de segurança, e investigar a geração e adaptação de testes automatizados pelos modelos.
- **Considerações éticas e de segurança:** Estudar os riscos de introdução de vulnerabilidades na refatoração automática e desenvolver diretrizes para verificação do código gerado. Abordar questões de privacidade com técnicas de anonimização e ambientes controlados.

Essas direções ampliam as possibilidades para pesquisas futuras e aplicações práticas, fortalecendo o papel dos LLMs na engenharia de software.

Referências

Al Madi, N. (2022). How readable is model-generated code? examining readability and visual inspection of github copilot. *Proceedings of the 37th IEEE/ACM international conference on automated software engineering*, páginas 1–5.

AlOmar, E. A., Venkatakrishnan, A., Mkaouer, Mohamed, W., Newman, C., e Ouni, A. (2024). How to refactor this code? an exploratory study on developer-chatgpt refactoring conversations. *Proceedings of the 21st International Conference on Mining Software Repositories*, páginas 202–206.

Beazley, D. M. e Jone, B. K. (2013). *Python Cookbook: Recipes for Mastering Python 3*. O'Reilly Media, 3th edição.

Beck, K. (2004). *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2th edição.

Bracarense, P. A. (2010). *Estatística Aplicada às Ciências Sociais*. IESDE Brasil S.A., Florianópolis, 2ª ed. edição.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, et al. (2020). Language models are few-shot learners. *Curran Associates, Inc.*, 33:1877–1901.

Brunnet, E. S., Flemmer, R. C., e Flemmer, C. L. (2009). A review of artificial intelligence. *2009 4th International Conference on Autonomous Robots and Agents*, páginas 385–392.

Campbell, G. A. e Papapetrou, P. P. (2013). *SonarQube in Action*. Manning, 1th edição.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, d. O. P. H., Kaplan, J., e et al. (2021). Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Chollet, F. (2021). *Deep Learning with Python*. Manning, 2th edição.

Coello, C. E. A. e Kouatly, M. N. A. R. (2024). Effectiveness of chatgpt in coding: A comparative analysis of popular large language models. *Digital*, 4:114–125.

Cohen, J. (1988). *Statistical power analysis for the behavioral sciences*. Lawrence Erlbaum Associates, Hillsdale, NJ, 2th edição.

Cordeiro, J., Noei, S., e Zou, Y. (2025). Llm-driven code refactoring: Opportunities and limitations. *2025 IEEE/ACM Second IDE Workshop (IDE), Ottawa, ON, Canada, 2025*, páginas 32–36.

Feathers, M. C. (2004). *Working Effectively with Legacy Code*. Prentice Hall, Upper Saddle River, NJ, 1st edição.

Fowler, M. (2019). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2th edição.

Gil, A. C. (2008). *Métodos e Técnicas de Pesquisa Social*. Editora Atlas, São Paulo, 6 edição.

GitHub Copilot Documentation (2024). Github copilot docs. `https://docs.github.com/copilot`. Acesso em: 17 jul. 2025.

Google DeepMind (2024). Gemini: A new era of ai. `https://deepmind.google/technologies/gemini/`. Acesso em: 30 jun. 2025.

Google Gemini (2024). Gemini: Chat to supercharge your ideas. `https://gemini.google/`. Acesso em: 30 nov. 2024.

Guo, D., Yang, D., Zhang, H., et al. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Guo, Q., Cao, J., Xie, X., Liu, et al. (2024). Exploring the potential of chatgpt in automated code refinement: An empirical study. *ICSE '24: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, páginas 1–13.

Hou, X., Zhao, Y., Liu, Y., et al. (2024). Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 33(8):1–79.

Jurafsky, D. e Martin, J. H. (2025). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Draft available at Stanford, Stanford, CA, 3th edição.

Kim, G., Behr, K., e Spafford, G. (2018). *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*. IT Revolution, 5th edição.

Kruchten, P., Nord, R., e Ozkaya, I. (2012). *Managing Technical Debt: Reducing Friction in Software Development*. Addison-Wesley, 1th edição.

Larson, R. e Farber, B. (2015). *Estatística Aplicada: Retratando o Mundo*. Editora Pearson, Porto Alegre, Brasil, 6th edição. Tradução e adaptação da obra original *Elementary Statistics: Picturing the World*.

Lenarduzzi, V., Lomio, F., Huttunen, H., e Taibi, D. (2020). Are sonarqube rules inducing bugs? *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, páginas 501–511.

Liang, J., Chen, J., Xia, L., Wang, M., et al. (2025). Deepseek-v3 technical report. `https://github.com/deepseek-ai/DeepSeek-V3`. Acesso em: 30 jun. 2025.

Liu, B., Jiang, Y., Zhang, Y., Niu, N., Li, G., e Liu, H. (2024). An empirical study on the potential of LLMs in automated software refactoring. *arXiv preprint arXiv:2411.04444v1*.

Lu, J., Ren, X., Ren, Y., Liu, A., e Xu, Z. (2020). Improving contextual language models for response retrieval in multi-turn conversation. *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*, páginas 1805–1808.

Lu, J., Yu, L., Li, X., Yang, L., e Zuo, C. (2023). Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. páginas 647–658.

Lutz, M. (2013). *Learning Python*. O'Reilly Media, Sebastopol, CA, USA, 5th edição.

Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., e Pinto, G. (2019). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. *Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, páginas 209–219.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education, 1th edição.

McCorduck, P., Minsky, M., Selfridge, O. G., e Simon, H. A. (1977). History of artificial intelligence. *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI)*, páginas 951–954.

Meszaros, G. (2007). *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley Professional, Boston, MA, 1th edição.

Meta AI (2023). Llama: Large language model by meta ai. `https://ai.meta.com/blog/large-language-model-llama-meta-ai/`. Acesso em: 22 jun. 2025.

Morettin, P. A. e de Oliveira Bussab, W. (2010). *Estatística Básica*. Saraiva, São Paulo, 6th edição.

Nguyen, N. e Nadi, S. (2022). An empirical evaluation of github copilot's code suggestions. *ACM - MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories*, páginas 1–5.

Nunes, H., Figueredo, E., Rocha, L., Nadi, S., e Ferreira, G. E. F. (2025). Evaluating the effectiveness of llms in fixing maintainability issues in real-world projects. *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, páginas 669–680.

OpenAI (2024). Introducing gpt-4o. `https://openai.com/research/gpt-4`. Acesso em: 11 jul. 2025.

Pichai, S. (2024). Google gemini: Advancing ai capabilities. `https://blog.google/technology/ai/google-gemini-ai/#sundar-note`. Acesso em: 30 nov. 2024.

Pomian, D., Bellur, A., Dilhara, M., Kurbatova, Z., et al. (2024). Em-assist: Safe automated extract method refactoring with llms. *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, páginas 582–586.

Pressman, R. S., Maxim, B. R., Arakaki, J., Arakaki, R., de Andrade, R., e Costa, F. (2021). *Engenharia de Software*. AMGH, 1th edição.

Pytest Development Team (2024). Pytest documentation. `https://docs.pytest.org/en/latest/`. Acesso em: 15 jul. 2025.

Python Software Foundation (2024a). Built-in exceptions. https://docs.python.org/3/library/exceptions.html. Acesso em: 15 jul. 2025.

Python Software Foundation (2024b). unittest — unit testing framework. `https://docs.python.org/3/library/unittest.html`. Acesso em: 15 jul. 2025.

Rossum, G. V., Warsaw, B., e Coghlan, A. (2001). Pep 8 – style guide for python code. `https://peps.python.org/pep-0008/`. Acesso em: 15 jun. 2025.

Ruohonen, J., Hjerppe, K., e Rindell, K. (2021). A large-scale security-oriented static analysis of python packages in pypi. *2021 18th International Conference on Privacy, Security and Trust (PST)*, páginas 1–10.

Shirafuji, A., Oda, Y., Suzuki, J., Morishita, M., e Watanobe, Y. (2023). Refactoring programs using large language models with few-shot examples. *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*, páginas 151–160.

Siam, M. K., Gu, H., e Cheng, J. Q. (2024). Programming with ai: Evaluating chatgpt, gemini, alphacode, and github copilot for programmers. *Proceedings of the 3rd International Conference on Computing Advancements*, páginas 346–354.

Siddiq, M. L., Roney, L., Zhang, J., e Santos, J. C. S. (2024). Quality assessment of chatgpt generated code and their use by developers. *Proceedings of the 21st International Conference on Mining Software Repositori*, páginas 152–156.

Sommerville, I. (2016). *Software Engineering*. Pearson, 10th edição.

SonarSource (2024). Sonarqube: Continuous inspection of code quality. `https://www.sonarsource.com/`. Acesso em: 29 jun. 2025.

SonarSource (2024). Sonarqube documentation. `https://docs.sonarsource.com/sonarqube/latest/`. Acesso em: 16 jul. 2025.

Ságodi, Z., Siket, I., e Ferenc, R. (2024). Methodology for code synthesis evaluation of llms presented by a case study of chatgpt and copilot. *IEEE Access*, 12:72303–72316.

TIOBE Software (2024). Tiobe index. `https://www.tiobe.com/tiobe-index/`. Acesso em: 06 novembro. 2024.

Touvron, H. et al. (2024). Llama 3 technical report. `https://ai.meta.com/blog/meta-llama-3/`. Acesso em: 11 jul. 2025.

Touvron, H., Lavril, T., Martinet, X., e et al. (2023). Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971v1*.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., et al. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.

Wang, B., Wei, J., Dong, L., Le, Q., e Zhou, D. (2022). Towards understanding chain-of-thought prompting: An empirical study of what matters. *arXiv preprint arXiv:2212.10001*.

Wei, J., Wang, X., Schuurmans, D., Bosma, et al. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Xueying, D., Mingwei, L., Kaixin, W., Hanlin, W., Junwei, L., et al. (2024). Evaluating large language models in class-level code generation. *ICSE '24: Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, páginas 1–13.

Yetistiren, B., Ozsoy, I., e Tuzun, E. (2022). Assessing the quality of github copilot's code generation. *ACM DL - International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '22)*, páginas 62–71.

Yu, Y., Rong, G., Shen, H., Zhang, H., et al. (2024). Fine-tuning large language models to improve accuracy and comprehensibility of automated code review. *ACM Transactions on Software Engineering and Methodology*, 34(1):1–26.

Zhou, D., Schärli, N., Hou, L., et al. (2023). Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*.

Zhu, Q., Guo, D., Shao, Z., Yang, D., et al. (2024). Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., e Aftandilian, E. (2024). Measuring github copilot's impact on productivity. *Communications of the ACM*, 67(3):54–63.

Apêndice A

Cálculo dos Quartis Utilizados na Classificação dos Projetos

Este apêndice apresenta o cálculo dos quartis para a classificação dos projetos.

Método de cálculo dos quartis. Os quartis foram obtidos pela fórmula $P = (n+1)p$, em que n representa o número de observações e p a fração correspondente ao quartil desejado (0,25, 0,50 e 0,75). Os valores intermediários foram determinados por interpolação linear entre as posições adjacentes na lista ordenada.

1. Número de Estrelas no GitHub

Valores ordenados:

9400, 25400, 31000, 35860, 43200, 48800, 52000, 52300, 75360, 79100

Posição Q1 $= 0.25 \times 11 = 2.75$

Q1 $= 25400 + 0.75 \times (31000 - 25400) = 25400 + 4200 = \boxed{29600}$

Posição Q2 $= 0.5 \times 11 = 5.5$

$$\textbf{Q2} = \frac{43200 + 48800}{2} = \boxed{46000}$$

Posição Q3 $= 0.75 \times 11 = 8.25$

Q3 $= 52300 + 0.25 \times (75360 - 52300) = 52300 + 5765 = \boxed{58065}$

Tabela A.1: Classificação por Quartis – Número de Estrelas

Quartil	Intervalo	Projeto(s)
Q1	$\leq 29\,600$	SQLAlchemy, Django-Rest
Q2	$29\,601 - 46\,000$	Poetry, Mitmproxy, Pandas
Q3	$46\,001 - 58\,065$	Rich, Requests, Scrapy
Q4	$> 58\,065$	FastAPI, Django

2. Número de Colaboradores

Valores ordenados:

$$257,\ 486,\ 560,\ 582,\ 631,\ 646,\ 752,\ 1261,\ 1700,\ 3152$$

Posição Q1 $= 0.25 \times 11 = 2.75$

$$\textbf{Q1} = 486 + 0.75 \times (560 - 486) = 486 + 55.5 = \boxed{541.5}$$

Posição Q2 $= 0.5 \times 11 = 5.5$

$$\textbf{Q2} = \frac{631 + 646}{2} = \boxed{638.5}$$

Posição Q3 $= 0.75 \times 11 = 8.25$

$$\textbf{Q3} = 1261 + 0.25 \times (1700 - 1261) = 1261 + 109.75 = \boxed{1370.75}$$

Tabela A.2: Classificação por Quartis – Colaboradores

Quartil	Intervalo	Projeto(s)
Q1	≤ 541.5	Rich, Mitmproxy
Q2	$541.6 - 638.5$	Scrapy, Poetry, SQLAlchemy
Q3	$638.6 - 1370.75$	Requests, FastAPI, Django-Rest
Q4	> 1370.75	Django, Pandas

3. Cobertura de Testes (%)

Valores ordenados:

$$75, 79, 81, 82, 91, 94, 94, 95, 96, 98$$

$$\textbf{Posição Q1} = 0.25 \times 11 = 2.75$$
$$\textbf{Q1} = 79 + 0.75 \times (81 - 79) = 79 + 1.5 = \boxed{80.5}$$
$$\textbf{Posição Q2} = 0.5 \times 11 = 5.5$$
$$\textbf{Q2} = \frac{91 + 94}{2} = \boxed{92.5}$$
$$\textbf{Posição Q3} = 0.75 \times 11 = 8.25$$
$$\textbf{Q3} = 95 + 0.25 \times (96 - 95) = 95 + 0.25 = \boxed{95.25}$$

Tabela A.3: Classificação por Quartis – Cobertura de Testes

Quartil	Intervalo	Projeto(s)
Q1	$\leq 80.5\%$	Django, Scrapy
Q2	$80.51\% - 92.5\%$	Requests, Mitmproxy, Rich
Q3	$92.51\% - 95.25\%$	Poetry, Pandas
Q4	$> 95.25\%$	Django-Rest, SQLAlchemy, FastAPI

4. Linhas de Código (LOC)

Valores ordenados:

3478, 15092, 22533, 29401, 74358, 76607, 119485, 124359, 191201, 271684

$$\textbf{Posição Q1} = 0.25 \times 11 = 2.75$$
$$\textbf{Q1} = 15092 + 0.75 \times (22533 - 15092) = 15092 + 5580.75 = \boxed{20672.75}$$
$$\textbf{Posição Q2} = 0.5 \times 11 = 5.5$$
$$\textbf{Q2} = \frac{74358 + 76607}{2} = \boxed{75482.5}$$
$$\textbf{Posição Q3} = 0.75 \times 11 = 8.25$$
$$\textbf{Q3} = 124359 + 0.25 \times (191201 - 124359) = 124359 + 16710.5 = \boxed{141069.5}$$

Tabela A.4: Classificação por Quartis – LOC

Quartil	Intervalo	Projeto(s)
Q1	$\leq 20\,672.75$	Requests, Poetry
Q2	$20\,672.76 - 75\,482.5$	Django-Rest, FastAPI, Rich
Q3	$75\,482.6 - 141\,069.5$	Mitmproxy, Django, SQLAlchemy
Q4	$> 141\,069.5$	Pandas, Scrapy

Apêndice B

Regras do SonarQube

Este apêndice apresenta as 47 regras do SonarQube.

Tabela B.1: Regras do SonarQube

NOME	CHAVE
Regular expressions should not contain empty groups	S6331
Functions and methods should not have identical implementations	S4144
Regular expressions should not contain multiple spaces	S6326
Regular expression quantifiers should be used concisely	S6353
Function parameters initial values should not be ignored	S1226
Functions returns should not be invariant	S3516
Track lack of copyright and license headers	S1451
”except”clauses should do more than raise	S2737
Method names should comply with a naming convention	S100
Using publicly writable directories is security-sensitive	S5443
String literals should not be duplicated	S1192
Two branches should not have same implementation	S1871
Local variable names should comply with convention	S117
”Exception”should not be raised directly	S112
Comments should not be at end of lines	S139
Functions should not have too many lines	S138
Code should not be commented out	S125
Control flow should not be nested too deeply	S134
Identical conditional branches	S3923
Functions and methods should not be empty	S1186
Mergeable ”if”statements should be combined	S1066
Unused function parameters should be removed	S1172
Unused assignments should be removed	S1854
Conditional expressions should not be nested	S3358
Unused local variables should be removed	S1481
Unnecessary imports should be removed	S1128
Regex patterns should not match empty string	S5842
Reluctant quantifiers in regex	S6019
Wildcard imports should not be used	S2208
Redundant parentheses should be removed	S1110
Cognitive Complexity should not be too high	S3776
Single-character regex classes should be avoided	S6397
Single-character alternations in regex	S6035
Unicode Grapheme Clusters in regex	S5868
Duplicate characters in regex classes	S5869
Break/return in ”finally”blocks	S1143
Named regex groups should be used	S5860
Boolean checks should not be inverted	S1940
Slow regular expressions are security-sensitive	S5852
Grouped regex alternatives with anchors	S5850
Track uses of ”FIXME”tags	S1134
Track uses of ”TODO”tags	S1135

Apêndice C

Regras de Manutenibilidade e sua Relevância

Este apêndice apresenta as 12 regras de manutenibilidade utilizada neste estudo.

Tabela C.1: Regras de Manutenibilidade Selecionadas e sua Relevância

Regra (ID)	**Nome Resumido**	**Descrição e Relevância**
CCM	Cognitive Complexity of functions should not be too high	A Complexidade Cognitiva mede a dificuldade de entender o fluxo de controle de uma unidade de código. Códigos com alta complexidade cognitiva são difíceis de ler, entender, testar e modificar. Como regra geral, alta complexidade cognitiva é um sinal de que o código deve ser refatorado em partes menores e mais fáceis de gerenciar.
FNP	Functions, methods and lambdas should not have too many parameters	Funções, métodos ou lambdas com uma longa lista de parâmetros são difíceis de usar, pois os mantenedores precisam descobrir a função de cada parâmetro e monitorar sua posição.
FNC	Function names should comply with a naming convention	Esta regra verifica se todos os nomes de funções correspondem a uma expressão regular fornecida. De acordo com o PEP8, os nomes de funções devem ser escritos em letras minúsculas, com as palavras separadas por sublinhados, conforme necessário. Essa convenção é conhecida como *snake_case*. Por exemplo: `calcular_área`, `imprimir_olá`, `processar_dados`.
BCI	Boolean checks should not be inverted	É desnecessariamente complexo inverter o resultado de uma comparação booleana. Em vez disso, deve-se fazer a comparação oposta.

Tabela C.1 – continuação

Regra (ID)	**Nome Resumido**	**Descrição e Relevância**
SLD	String literals should not be duplicated	Literais de string duplicados tornam o processo de refatoração complexo e propenso a erros, pois qualquer alteração precisaria ser propagada em todas as ocorrências.
LVN	Local variable and function parameter names should comply with a naming convention	Variáveis locais e parâmetros de função devem ser nomeados de forma consistente para comunicar a intenção e melhorar a manutenibilidade.
UFP	Unused function parameters should be removed	Parâmetros de função não utilizados ocorrem quando a função declara parâmetros, mas não faz uso deles em seu corpo. Embora o código funcione, isso gera confusão e prejudica a clareza da intenção do programador.
MIS	Mergeable "if"statements should be combined	Blocos `if` que podem ser mesclados devem ser combinados. Isso reduz a duplicação e melhora a legibilidade.
ULV	Unused local variables should be removed	Uma variável local não utilizada é uma variável que foi declarada, mas não é usada. É um código morto, contribuindo para complexidade desnecessária.
SES	'startswith' or 'endswith' methods should be used instead of string slicing in condition expressions	Esta regra levanta um problema quando o fatiamento de string é usado em expressões condicionais em vez dos métodos `startswith` ou `endswith`.
BSV	Builtins should not be shadowed by local variables	Essa regra é violada quando há o sombreamento (*shadowing*) de nomes embutidos da linguagem, como `list`, `str`, `sum`, `input`, etc.
TUT	Track uses of "TODO"tags	Desenvolvedores usam tags `TODO` para marcar áreas onde melhorias são necessárias, mas muitas vezes são esquecidas. Essa regra visa evitar código inacabado.

Apêndice D

Cálculo Amostral da Avaliação Humana

Este apêndice apresenta o "Cálculo Amostral da Avaliação Humana" e detalha o procedimento usado para definir o tamanho da amostra de pares de código avaliados pelos participantes, incluindo critérios, fórmulas e pressupostos para garantir representatividade.

Fórmula para População Finita

$$n = \frac{Z^2 \cdot p \cdot (1-p) \cdot N}{e^2 \cdot (N-1) + Z^2 \cdot p \cdot (1-p)} \tag{D.1}$$

Onde:

- n = tamanho da amostra;
- N = tamanho da população (5.585 métodos);
- Z = valor da distribuição normal padrão para o nível de confiança (1,645 para 90%);
- e = margem de erro (0,07);
- p = proporção estimada da característica de interesse (0,5).

Cálculo Detalhado

Numerador:

$$Z^2 \cdot p \cdot (1-p) \cdot N = 1{,}645^2 \cdot 0{,}5 \cdot 0{,}5 \cdot 5585 = 2{,}706025 \cdot 0{,}25 \cdot 5585 = 3.776{,}3 \tag{D.2}$$

Denominador:

$$e^2 \cdot (N-1) + Z^2 \cdot p \cdot (1-p) = 0{,}0049 \cdot 5584 + 0{,}67650625 = 27{,}3616 + 0{,}67650625 = 28{,}0381 \tag{D.3}$$

Resultado final:

$$n = \frac{3.776{,}3}{28{,}0381} \approx 134{,}6 \tag{D.4}$$

Amostra Utilizada

O valor obtido (aproximadamente 135) representa o número mínimo necessário de métodos para garantir um nível de confiança de 90% e margem de erro de 7%.

Observação: Optamos por utilizar uma amostra de 150 métodos (cerca de 11% acima do valor calculado) para:

- Garantir maior robustez diante de possíveis variações ou perdas de dados;
- Facilitar a distribuição equitativa entre projetos e condições analisadas;
- Reduzir a margem de erro para aproximadamente 6,7%, ou aumentar o nível de confiança para cerca de 93%.

Resumo Comparativo dos Parâmetros

Parâmetro	**Valor Original**	**Valor para** $n = 150$
Nível de Confiança	90%	∼93%
Margem de Erro	7%	∼6,7%
Tamanho Amostral	135	150

Tabela D.1: Comparação entre o cálculo teórico e a amostra utilizada

Conclusão

A adoção de 150 métodos na amostra se mostra estatisticamente válida e mais precisa que o mínimo exigido, garantindo maior confiabilidade aos resultados com uma margem de erro reduzida para aproximadamente 6,7%.

Apêndice E

Capturas de Tela das Interações com os Modelos de Linguagem

Este apêndice apresenta capturas das interações com os modelos de linguagem na Etapa 4 do experimento, mostrando o ambiente, os prompts e as respostas geradas. O objetivo é fornecer uma visão clara do uso dos modelos e facilitar a reprodutibilidade da pesquisa.

Copilot Chat (GPT-4o) – Zero-Shot

As interações com o Copilot Chat na abordagem *zero-shot* foram realizadas diretamente no ambiente de desenvolvimento Visual Studio Code (VSCode), utilizando a extensão do GitHub Copilot e a caixa de diálogo.

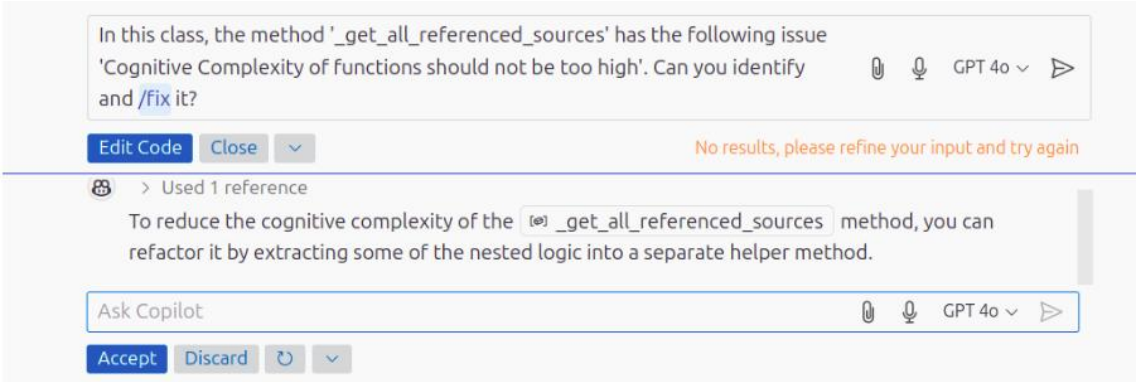

Figura E.1: Tela de interação do Copilot Chat 4o – Zero-Shot

Copilot Chat (GPT-4o) – Few-Shot

As interações com o Copilot Chat na abordagem *few-shot* também foram realizadas no VSCode, por meio da extensão do GitHub Copilot, utilizando o chat para envio dos exemplos e dos prompts.

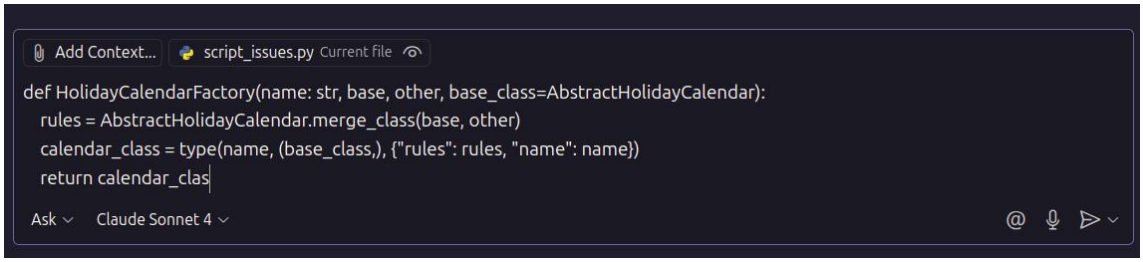

Figura E.2: Tela de interação do Copilot Chat 4o – Few-Shot

LLaMA 3.3 70B Instruct

Para interagir com o modelo LLaMA, utilizamos a plataforma Hugging Face Chat (`https://huggingface.co/chat/settings/meta-llama/Meta-Llama-3.3-70B-Instruct`). As telas incluem o código com o problema detectado, o prompt inserido e a resposta do modelo com a sugestão de refatoração. Destacamos que a execução foi feita online, devido à limitação de recursos locais.

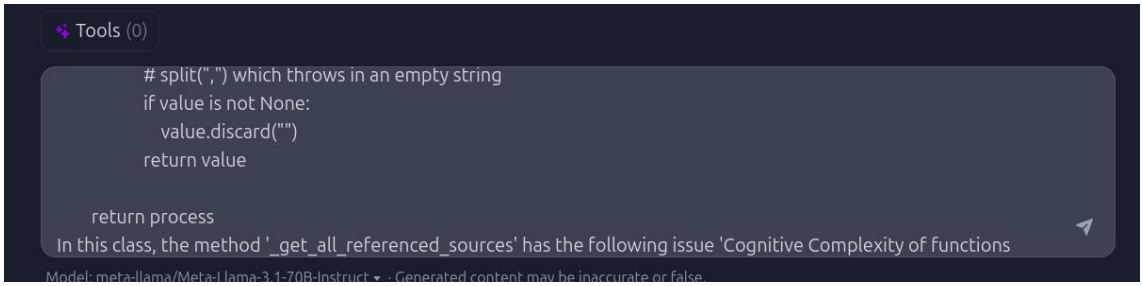

Figura E.3: Tela de interação com o LLaMA 3.3 70B Instruct

DeepSeek V3

As interações com o DeepSeek V3 foram realizadas via interface web, disponível em `https://chat.deepseek.com/`. As capturas demonstram a entrada do código com o respectivo prompt, bem como a resposta do modelo. O processo foi semelhante ao utilizado com o LLaMA, sendo executado completamente em ambiente online.

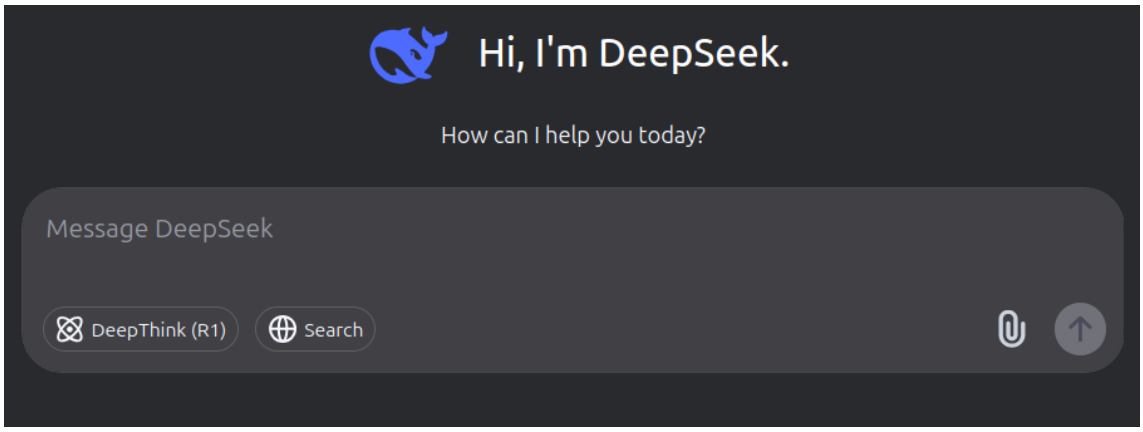

Figura E.4: Tela de interação com o DeepSeek V3

Gemini 2.5 Pro

O modelo Gemini foi acessado por meio da interface da Google, disponível em `https://gemini.google.com/app?hl=pt-BR`. Assim como nos casos anteriores, as telas ilustram o código com problemas, o prompt aplicado e a resposta gerada. Essa interface permitiu avaliar o desempenho do modelo em uma ferramenta amplamente acessível ao público.

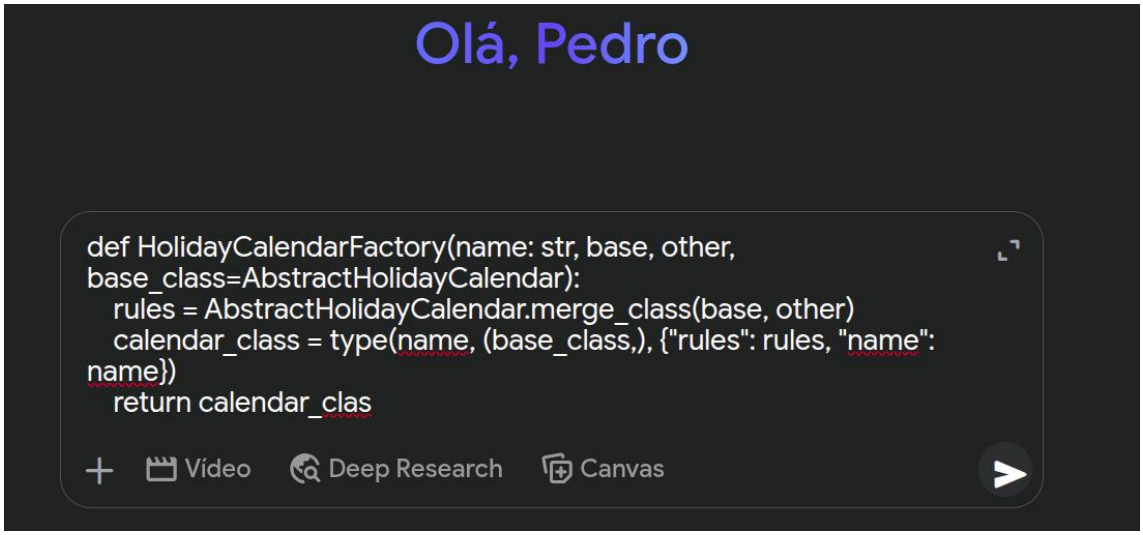

Figura E.5: Tela de interação com o Gemini 2.5 Pro

Apêndice F

Cálculo Amostral da Quantidade de Métodos com Problemas de Manutenibilidade Filtrados no Estudo

Este apêndice detalha o cálculo amostral usado para definir o número de pares de código avaliados na etapa humana do experimento, explicando critérios, fórmulas e pressupostos para garantir transparência e representatividade.

F.1 Parâmetros Considerados

Para o cálculo do tamanho da amostra, foram adotados os seguintes parâmetros, com base em práticas consolidadas em pesquisa quantitativa para populações finitas:

- **Tamanho da População (N):** 371 pares de código gerados com técnica *few-shot prompting*.
- **Nível de Confiança (z):** 95%, correspondente ao valor z = 1,96.
- **Margem de Erro (e):** 10%.
- **Proporção esperada (p):** 0,5 (valor conservador, utilizado quando não há estimativa prévia, maximizando a variabilidade).

F.2 Fórmula Utilizada

A fórmula aplicada para populações finitas é:

$$n = \frac{N \times z^2 \times p(1-p)}{e^2 \times (N-1) + z^2 \times p(1-p)}$$

Onde:

- n = tamanho da amostra
- N = tamanho da população
- z = valor da distribuição normal correspondente ao nível de confiança desejado
- p = proporção estimada da característica na população
- e = margem de erro tolerável

F.3 Cálculo Aplicado

Substituindo os valores:

$$n = \frac{371 \times (1,96)^2 \times 0,5 \times 0,5}{(0,1)^2 \times (371-1) + (1,96)^2 \times 0,5 \times 0,5}$$

$$n = \frac{371 \times 3,8416 \times 0,25}{0,01 \times 370 + 3,8416 \times 0,25}$$

$$n = \frac{357,3764}{3,7 + 0,9604} = \frac{357,3764}{4,6604} \approx 76,65$$

Tabela F.1: Resumo dos Parâmetros do Cálculo Amostral

Parâmetro	**Valor**	**Justificativa**
Tamanho da população (N)	371	Total de pares de código disponíveis.
Margem de erro (e)	10%	Adequada para estudos exploratórios (ajustada para 9.7% com $n = 80$).
Nível de confiança (z)	95% ($z = 1.96$)	Padrão em pesquisas quantitativas.
Proporção esperada (p)	0.5	Valor conservador que maximiza a variabilidade ($p(1-p)$).

F.4 Definição Final da Amostra

O valor obtido foi de aproximadamente 77 pares de código. Para garantir maior robustez e facilitar a divisão dos exemplos nos formulários de avaliação, decidiu-se trabalhar com uma amostra de **80 pares**.

Este valor atende aos critérios estatísticos estabelecidos, assegurando representatividade da amostra em relação ao universo de pares de código disponíveis.

Apêndice G

Exemplos de Refatorações

Neste apêndice, apresentamos exemplos de refatorações bem-sucedidas, detalhando o problema identificado e como ele foi resolvido. Em seguida, discutimos casos em que os modelos falharam, introduzindo erros ou gerando soluções alucinadas.

G.0.1 Exemplos de Refatoração Bem Sucedida

1. **UFP (Unused function parameters should be removed):** Parâmetros de função não utilizados devem ser removidos para melhorar a manutenibilidade e clareza do código, conforme a PEP 8 (Rossum et al., 2001). No Código G.1, o SonarQube identificou o parâmetro ***kw: Any* não utilizado na linha 5. A LLM o removeu sem alterar o comportamento, e os testes confirmaram a integridade do código.

Código G.1: Exemplo de Refatoração - UFP

```
1  # Original
2  def adapt(
3          self,
4          cls: Type[Union[TypeEngine[Any], TypeEngineMixin]],
5          **kw: Any
6      ) -> TypeEngine[Any]:
7      ...
8  # ------------------------------------------------------------
9  # Refatorada pelo LLM
10 def adapt(
11         self,
12         cls: Type[Union[TypeEngine[Any], TypeEngineMixin]]
13     ) -> TypeEngine[Any]:
```

2. **UFL (Unused local variables should be removed):** Variáveis locais não utilizadas devem ser removidas para melhorar a clareza e reduzir a complexidade. No exemplo G.2, a variável *discriminator* na linha 3 não era usada. A LLM a eliminou, substituindo por *underscore* (_) (linha 10) conforme a PEP 8 Rossum et al. (2001).

Código G.2: Exemplo de Alucinação - UFL

```
# Original
 def value(self):
        fieldname, discriminator = self.type_map[self.type]
        if fieldname is None:
            return None
        ...
# ------------------------------------------------------------
# Refatorado pelo LLM
 def value(self):
        fieldname, _ = self.type_map[self.type]
        if fieldname is None:
        ...
```

3. **CCF (Cognitive complexity of functions should not be too high):** Esta regra determina que funções devem ser simples e de fácil compreensão, evitando ramificações e aninhamentos excessivos SonarSource (2024). No exemplo do Código G.3, o SonarQube identificou uma violação na linha 2, no método *resolve_redirects*. A refatoração sugerida pela LLM dividiu o método em funções auxiliares menores (linhas 17, 19, 21 e 23), tornando o código mais modular e menos complexo.

Código G.3: Exemplo de Alucinação - CCF

```
# Original
def resolve_redirects(
     ...
    ):
        """Receives a Response. Returns a generator of
           Responses or Requests."""

        hist = []  # keep track of history

        url = self.get_redirect_target(resp)
      ...
# ------------------------------------------------------------
# Refatorado pelo LLM
 def resolve_redirects(
      ...,
    ):
       ...
 def _consume_response_content(self, resp):
      ...
 def _handle_redirect_url(self, url, resp, previous_fragment):
       ...
 def _purge_headers(self, prepared_request, resp):
       ...
 def _rewind_body_if_needed(self, prepared_request):
       ...
```

4. **LVN (Local variable and function parameter names should comply with a naming convention:)** A regra é violada quando a nomenclatura de variáveis

ou parâmetros não segue a PEP 8. No Código G.16, o SonarQube identificou na linha 2 a variável `bodyBytes` em CamelCase, contrariando a convenção. A LLM refatorou para `body_bytes` (linha 7), seguindo o estilo snake_case. Embora a mudança não afete a funcionalidade, ao atualizar todas as ocorrências no código, a correção melhora a legibilidade, a consistência e facilita a manutenção.

Código G.4: Exemplo de Refatoraçãp - LVN

```
1 # Original
2  def dataReceived(self, bodyBytes: bytes) -> None:
3          ...
4      ...
5 ------------------------------------------------------------
6 # Refatorado por LLM
7 def dataReceived(self, body_bytes: bytes) -> None:
8          ...
9      ...
```

5. **MSI (Mergeable "if" statements should be combined:)** A regra S1066 é violada quando blocos if aninhados ou consecutivos poderiam ser combinados em uma única instrução condicional. No Código G.5, isso ocorre nas linhas 3 e 4, onde dois ifs aninhados podem ser unificados. A LLM refatorou as condições em uma única expressão (linha 9) com o operador `and`, tornando o código mais conciso e legível. O aninhamento desnecessário foi eliminado, mantendo o comportamento funcional original.

Código G.5: Exemplo de Refatoração - MSI

```
1 # Original
2 def insert(self, event_key: _EventKey[_ET], propagate: bool) ->
      None:
3          if event_key.prepend_to_list(self, self.listeners):
4              if propagate:
5                  self.propagate.add(event_key._listen_fn)
6 ------------------------------------------------------------
7 # Refatorado por LLM
8 def insert(self, event_key: _EventKey[_ET], propagate: bool) ->
      None:
9      if event_key.prepend_to_list(self, self.listeners) and
          propagate:
10         self.propagate.add(event_key._listen_fn)
```

6. **BCI (Boolean checks should not be inverted):** Esta regra é violada quando se usa negação indireta em comparações booleanas, tornando o código menos legível. No Código G.6, o SonarQube identificou o problema S1940 na linha 5: `if not hashed_password == user.hashed_password`. Embora correta, essa expressão dificulta a leitura. A LLM refatorou para `if hashed_password != user.hashed_password` (linha 14), tornando o código mais direto. A mudança melhora a semântica, legibilidade e consistência. O trecho refatorado é mais

fácil de compreender e manter. Comparações explícitas reduzem erros de interpretação e facilitam a manutenção. A funcionalidade do código é preservada, garantindo clareza e segurança.

Código G.6: Exemplo de Refatoração - BSI

```
# Original
@app.post("/token")
async def login(form_data: OAuth2PasswordRequestForm = Depends
    ()):
     ,,,
     if not hashed_password == user.hashed_password:
         raise HTTPException(status_code=400, detail="Incorrect␣
             username␣or␣password")

     return {"access_token": user.username, "token_type": "
         bearer"}
-------------------------------------------------------------
# Refatorado por LLM
@app.post("/token")
async def login(form_data: OAuth2PasswordRequestForm = Depends
    ()):
    ...
     if hashed_password != user.hashed_password:  # Refactored
         to avoid inverted Boolean check
         raise HTTPException(status_code=400, detail="Incorrect␣
             username␣or␣password")

     return {"access_token": user.username, "token_type": "
         bearer"}
```

7. **SES (startswith‘ or ‘endswith‘ methods should be used instead of string slicing in condition expressions:)** A regra S6659 alerta contra o uso desnecessário de fatiamento de strings quando métodos nativos mais claros estão disponíveis. No Código G.7, o problema ocorre na linha 4 e 5. A LLM refatorou o código para usar o método `endswith` (linh 10), tornando a intenção mais explícita, melhorando a legibilidade e facilitando a manutenção.

Código G.7: Exemplo de Refatoraçãp - LVN

```

# Original
def write(self, s: str) -> int:
         if s[-1:] == "\n":
             s = s[:-1]
         ...
-------------------------------------------------------------
# Refatorado por LLM
def write(self, s: str) -> int:
     if s.endswith("\n"):  # Replaced slicing with endswith
         s = s[:-1]
     ...
```

8. **BSV (Builtins should not be shadowed by local va- riables:)** A regra S1192 é violada quando literais de string são duplicadas no código, em vez de serem reutilizadas por constantes. No Código G.9, isso ocorre nas linhas 3, 4 e 5, com a string ”header”repetida no método. A LLM refatorou o código extraindo as strings para constantes nomeadas, como `HEADER_PREFIX` (linhas 8, 9 e 10) e `HEADER_SUFFIX`. Assim, alterações futuras podem ser feitas em um único local, melhorando a legibilidade e a manutenibilidade.

Código G.8: Exemplo de Refatoraçãp - SLD

```
1 # Original
2 def headername(spec: str):
3     if not (spec.startswith("header[") and spec.endswith("]")):
4         raise exceptions.CommandError("Invalid␣header␣spec:␣%s"
              % spec)
5     return spec[len("header[") : -1].strip()
6 ------------------------------------------------------------
7 # Refatorado por LLM
8 HEADER_PREFIX = "header["
9 HEADER_SUFFIX = "]"
10 INVALID_HEADER_SPEC = "Invalid␣header␣spec:␣%s"
11
12 def headername(spec: str):
13     if not (spec.startswith(HEADER_PREFIX) and spec.endswith(
          HEADER_SUFFIX)):
14         raise exceptions.CommandError(INVALID_HEADER_SPEC %
              spec)
15     return spec[len(HEADER_PREFIX) : -len(HEADER_SUFFIX)].strip
          ()
```

9. **SLD (String literals should not be duplicated:)** No Código G.9, a violação ocorre nas linhas 5, onde o identificador *complex* é utilizado, conflitando com uma palavra reservada do Python. A LLM corrigiu o problema renomeando a variável para *complex_conflict* (linha 17), tornando o código mais claro, seguro e conforme as boas práticas de programação.

Código G.9: Exemplo de Refatoraçãp - SLD

```
1 # Original
2  def _is_collapsible(self, incompatibility: Incompatibility) ->
       bool:
3         ...
4
5         complex = (
6             cause.conflict
7             if isinstance(cause.conflict.cause, ConflictCause)
8             else cause.other
9         )
10
11        return complex not in self._line_numbers
12 ------------------------------------------------------------
13 # Refatorado por LLM
```

```
14 def _is_collapsible(self, incompatibility: Incompatibility) ->
       bool:
15         ...
16
17         complex_conflict = (
18             cause.conflict
19             if isinstance(cause.conflict.cause, ConflictCause)
20             else cause.other
21         )
22
23         return complex_conflict not in self._line_numbers
```

G.0.2 Exemplos de Alucinações

1. **UFP (Unused function parameters should be removed):** Parâmetros de função não utilizados devem ser removidos para melhorar legibilidade e manutenção. No Código G.10, a assinatura do método na linha 3 inclui o parâmetro *url=*, não usado internamente. A LLM sugeriu removê-lo na linha 10 do código refatorado, mas ele pode ser relevante para chamadas externas ou casos específicos, e sua remoção pode comprometer a funcionalidade.

Código G.10: Exemplo de Alucinação - UFP

```
1
2     # Original
3     def prepare_auth(self, auth, url=""):
4         """Prepares the given HTTP auth data."""
5         ...
6
7     ------------------------------------------------------------
8
9     # Refatorado por LLM
10    def prepare_auth(self, auth):
11        """Prepares the given HTTP auth data."""
12        ...
```

2. **CCF (Cognitive complexity of functions should not be too high):** No Código G.11, LLMs refatoraram usando *extração de método*, mas partes do código original foram removidas. O LLM criou novos métodos sem preservar a assinatura de *def render(self):* (linha 3), gerando inconsistências. Adicionou parâmetros desnecessários (*size*, *focus=False*, linha 20, e usou variáveis não declaradas (*parts*). Esses erros mostram que alterações automáticas podem gerar falhas. Embora úteis, LLMs ainda têm dificuldade em compreender o contexto completo. Refatorações automáticas devem ser revisadas manualmente para evitar aumento de complexidade. Além disso, a análise sugere que esses erros ocorrem com mais frequência em métodos longos ou com múltiplas responsabilidades, onde o modelo tende a simplificar excessivamente a lógica.

Código G.11: Exemplo de Alucinação - CCF

```
1
2  # Original
3  def render(self):
4          parts, remaining = self.master.commands.parse_partial(
               self.text)
5          ret = []
6          ...
7  # ------------------------------------------------------------
8  # Refatorada pela LLM
9  def _render_parts(self, parts, remaining):
10         ...
11
12     def _render_valid_parts(self, parts):
13         ...
14     def _render_remaining_parts(self, parts, remaining):
15         ...
16
17 # ------------------------------------------------------------
18
19 # Refatorada pela LLM
20 def render(self, size, focus=False) -> urwid.Canvas:
21     ...
22 def _render_parts(self, parts, remaining, focus):
23     ...
24 def _render_leader(self):
25     return [("text", "")]
26 def _render_valid_parts(self, parts):
27   ...
28 def _render_invalid_parts(self, parts):
29    ...
30 def _render_hint(self, remaining):
31     if remaining:
32         if parts[-1].type!= mitmproxy.types.Space:
33             ret.append(("text", "␣"))
34         for param in remaining:
35             ret.append(("commander_hint", f"{param}␣"))
36         return ret
37     return []
```

3. **LVN (Local variable and function parameter names should comply with a naming convention:** Variáveis locais e parâmetros devem seguir convenções de nomenclatura para garantir legibilidade e consistência. Nesta pesquisa, foram avaliados sete casos por projeto/prompt. No Código G.12, a LLM corrigiu corretamente *threadId* (linha 6) para *thread_id*, mas modificou desnecessariamente outras variáveis (*signal*, *frame*, *file*) (linha 16), gerando inconsistências. Por exemplo, renomeou *file* para *output_file* na assinatura, mas manteve *file* na linha 18, ocasionando erro de execução. Essas alterações desnecessárias configuram uma alucinação da LLM.

Código G.12: Exemplo de Alucinação - LVN

```
1
```

```
2     # Original
3     def dump_stacks(signal=None, frame=None, file=sys.stdout):
4         id2name = {th.ident: th.name for th in threading.
              enumerate()}
5         code = []
6         for threadId, stack in sys._current_frames().items():
7             ...
8         print("\n".join(code), file=file)
9         if os.getenv("MITMPROXY_DEBUG_EXIT"):  # pragma: no
              cover
10            sys.exit(1)
11
12    ------------------------------------------------------------
13    # Refatorado por LLM
14    def dump_stacks(signal_parameter=None, frame_parameter=None
          , output_file=sys.stdout):
15        ...
16        for thread_id, stack in sys._current_frames().items():
17            ...
18        print("\n".join(stack_code_lines), file=file)
19        if os.getenv("MITMPROXY_DEBUG_EXIT"):  # pragma: no
              cover
20            sys.exit(1)
```

4. **TUT (Track uses of "TODO"tags):** A regra TUT é violada quando comentários *TODO* permanecem no código de produção. No Código G.13, há um *TODO* na linha 4 sobre argumentos do tipo *float*. O LLM, em vez de apenas remover o comentário, adicionou parâmetros (*precision*, *scale*, *asdecimal*) (linha12) e lógica não presente, caracterizando uma **alucinação**. A refatoração extrapolou o escopo, incluindo implementações não especificadas no código original.

Código G.13: TUT

```
1
2     # Original
3     def __init__(self, **kw):
4         # TODO: float arguments?
5         sqltypes.Float.__init__(self)
6         sqltypes.MatchType.__init__(self)
7
8     ------------------------------------------------------------
9
10    # Refatorado por LLM
11    def __init__(self, **kw):
12        # Handle float arguments if provided
13        self.precision = kw.pop("precision", None)
14        self.scale = kw.pop("scale", None)
15        sqltypes.Float.__init__(self, precision=self.precision,
              **kw)
16        sqltypes.MatchType.__init__(self)
```

5. **FMP (Functions, methods and lambdas should not have too many parameters):** Métodos com muitos parâmetros dificultam a leitura. No método *add_item* (linha 3), o LLM consolidou parâmetros e *kwargs* em *item_data* (linha 27), incluindo valores extras e alterando o comportamento original. Essa **alucinação funcional** permite dados inesperados sem validação, dificultando a manutenção. Exemplos específicos podem gerar refatorações mais claras.

Código G.14: Exemplo de Alucinação - FMP

```
1
2  # Original
3  def add_item(
4          self,
5          title,
6          link,
7          ...
8          **kwargs,
9      ):
10
11     ...
12     self.items.append(
13         {
14             "title": to_str(item_data.get('title')),
15             ...
16             **item_data.get('kwargs', {}),
17         }
18     )
19 ------------------------------------------------------------
20 # Refatorado por LLM
21 def add_item(self, item_data):
22
23     ...
24     self.items.append(
25         {
26             ...
27             **item_data.get('kwargs', {}),
28         }
29     )
```

6. **ULV (Unused local variables should be removed):** A regra S1481 é violada quando uma variável é declarada, mas não utilizada no corpo do método. No Código G.15, embora os parâmetros `stream` e `media_type` (linha 2) não fossem utilizados dentro do método, a LLM os removeu, presumindo que eram desnecessários. Além disso, a variável `disposition`, não utilizada após a extração dos parâmetros, foi substituída por `_`. No entanto, o método era invocado em outro ponto do código com a assinatura original, o que resultou em uma exceção durante a execução. Esse caso representa uma *alucinação contextual*, em que o modelo aplica uma refatoração localmente coerente, mas globalmente incorreta, por não considerar as dependências estruturais do programa.

Código G.15: Exemplo de Refatoração - ULV

```
# Original
def get_filename(self, stream, media_type, parser_context):
    meta = parser_context['request'].META
    disposition, params = parse_header_parameters(meta['
        HTTP_CONTENT_DISPOSITION'])
    return params['filename']
#
    ------------------------------------------------------------------------

# Refatorado por LLM
def get_filename(self, parser_context):
    meta = parser_context['request'].META
    _, params = parse_header_parameters(meta['
        HTTP_CONTENT_DISPOSITION'])
    return params['filename']
```

7. **MIS (Mergeable "if"statements should be combined):** Instruções *if* que podem ser combinadas devem ser unificadas para reduzir complexidade (linhas 3 e 4). No Código G.16, a LLM atribuiu *soft_wrap = self.soft_wrap* (linha 14) sem verificar o estado anterior. Isso pode gerar inconsistências com *no_wrap* e *overflow* (linhas 19 e 21), tornando o comportamento da função imprevisível.

Código G.16: Exemplo de Alucinação - MIS

```
# Original
    ...
        if soft_wrap is None or soft_wrap:
            if no_wrap is None:
                no_wrap = True
            if overflow is None:
                overflow = "ignore"
            crop = False
    ...

    ------------------------------------------------------------

    # Refatorado por LLM
    ...
        if soft_wrap is None:
            soft_wrap = self.soft_wrap
        if soft_wrap:
            if no_wrap is None:
                no_wrap = True
            if overflow is None:
                overflow = "ignore"
            crop = False
    ...
```

8. **SLD (String literals should not be duplicated):** Essa violação ocorre ao duplicar literais de strings, gerando inconsistências e dificultando a manutenção. No Código G.17, o SonarQube identificou a duplicação na linha 5. O LLM tentou usar uma constante `ERROR_MSG`, mas não alterou todas as ocorrências

da string. Além disso, nas linhas 29-30, substituiu o operador bit a bit ~ por `not`, mudando o comportamento do código. Essa alteração é uma **alucinação**, pois compromete a funcionalidade sem justificar a prática.

Código G.17: Exemplo de Alucinação - SLD

```
# Original
def kleene_or(
    ...
    if not isinstance(left, np.ndarray):
        raise TypeError("Either␣‘left‘␣or␣‘right‘␣need␣to␣be␣a␣
            np.ndarray.")

    ...

    if right_mask is not None:
        # output is unknown where (False & NA), (NA & False), (
            NA & NA)
        left_false = ~(left | left_mask)
        right_false = ~(right | right_mask)
        ...

    return result, mask
------------------------------------------------------------
# Refatorado por LLM
ERROR_MSG = "Either␣‘left‘␣or␣‘right‘␣need␣to␣be␣a␣np.ndarray."

def kleene_or(
   ...

    if not isinstance(left, np.ndarray):
        raise TypeError(ERROR_MSG)

    ...
    if right_mask is not None:
        # output is unknown where (False & NA), (NA & False), (
            NA & NA)
        left_false = not (left | left_mask)
        right_false = not (right | right_mask)
        mask = (
            (left_false & right_mask)
            | (right_false & left_mask)
            | (left_mask & right_mask)
        )
   ...

    return result, mask
```

9. **BCI (Boolean checks should not be inverted):** Esta regra é violada quando uma condição booleana é negada, dificultando a compreensão (linha 5). No Código G.18, a LLM refatorou corretamente a condição para *if len(args) != 1 or not is_url(args[0]):* (linha 18). No entanto, na linha 26, alterou

print_results(opts) para *self.print_results* sem o argumento *opts*, introduzindo um erro funcional. Essa alucinação pode gerar *TypeError* ou resultados incorretos se *print_results* depender de *opts*.

Código G.18: Exemplo de Alucinação - BCI

```
1
2 # Original
3 def run(self, args: List[str], opts: argparse.Namespace) ->
    None:
4         # parse arguments
5         if not len(args) == 1 or not is_url(args[0]):
6             ...
7         ...
8
9         ...
10        if self.spidercls and opts.depth > 0:
11            self.start_parsing(url, opts)
12            self.print_results(opts)
13
14 # ------------------------------------------------------------
15
16 # Refatorado por LLM
17 def run(self, args: List[str], opts: argparse.Namespace) ->
    None:
18     # parse arguments
19     if len(args)!= 1 or not is_url(args[0]):
20             ...
21         ...
22   ...
23     if self.spidercls and opts.depth > 0:
24         self.start_parsing(url, opts)
25         self.print_results
```

Esses exemplos evidenciam que os LLMs podem gerar refatorações válidas e úteis, capazes de melhorar a clareza e a manutenibilidade do código. No entanto, também revelam que o mesmo modelo pode cometer alucinações sutis — como remoções indevidas ou alterações de contexto, quando não considera dependências externas ao trecho analisado. Assim, a adoção dessas ferramentas deve ser acompanhada de revisão humana criteriosa, garantindo que as modificações tragam benefícios reais sem comprometer a funcionalidade do sistema.