Uma abordagem iterativa baseada em LLMs para melhoria de código a partir de recomendações de análise estática

João Carlos Gonçalves

Universidade Federal de Uberlândia
Faculdade de Computação
Programa de Pós-Graduação em Ciência da Computação

Uberlândia
2025

João Carlos Gonçalves

Uma abordagem iterativa baseada em LLMs para melhoria de código a partir de recomendações de análise estática

Dissertação de mestrado apresentada ao Programa de Pós-graduação da Faculdade de Computação da Universidade Federal de Uberlândia como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Ciência da Computação

Orientador: Prof. Dr. Marcelo de Almeida Maia

Uberlândia
2025

Ficha Catalográfica Online do Sistema de Bibliotecas da UFU
com dados informados pelo(a) próprio(a) autor(a).

G635 Gonçalves, João Carlos, 1991-
2025 Uma abordagem iterativa baseada em LLMs para melhoria de código a partir de recomendações de análise estática [recurso eletrônico] / João Carlos Gonçalves. - 2025.

Orientador: Marcelo de Almeida Maia.
Dissertação (Mestrado) - Universidade Federal de Uberlândia, Pós-graduação em Ciência da Computação.
Modo de acesso: Internet.
DOI http://doi.org/10.14393/ufu.di.2025.481
Inclui bibliografia.

1. Computação. I. Maia, Marcelo de Almeida,1973-, (Orient.). II. Universidade Federal de Uberlândia. Pós-graduação em Ciência da Computação. III. Título.

CDU: 681.3

Bibliotecários responsáveis pela estrutura de acordo com o AACR2:
Gizele Cristine Nunes do Couto - CRB6/2091
Nelson Marcos Ferreira - CRB6/3074

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
Coordenação do Programa de Pós-Graduação em Computação
Av. João Naves de Ávila, 2121, Bloco 1A, Sala 243 - Bairro Santa Mônica, Uberlândia-MG, CEP 38400-902
Telefone: (34) 3239-4470 - www.ppgco.facom.ufu.br - cpgfacom@ufu.br

ATA DE DEFESA - PÓS-GRADUAÇÃO

Programa de Pós-Graduação em:	Ciência da Computação				
Defesa de:	Dissertação, 13/2025, PPGCO				
Data:	01 de Agosto de 2025	Hora de início:	14:00	Hora de encerramento:	16:30
Matrícula do Discente:	12312CCP014				
Nome do Discente:	João Carlos Gonçalves				
Título do Trabalho:	Uma abordagem iterativa baseada em LLMs para melhoria de código a partir de recomendações de análise estática				
Área de concentração:	Ciência da Computação				
Linha de pesquisa:	Engenharia de Software				
Projeto de Pesquisa de vinculação:	Projeto 1: Smart Developer - Assistentes Automatizados para Apoio ao Desenvolvedor de Software Projeto 2: Parece funcionar, mas... Qualidade do Teste de Software na era dos Modelos de Linguagem				

Reuniu-se por videoconferência, a Banca Examinadora, designada pelo Colegiado do Programa de Pós-graduação em Ciência da Computação, assim composta: Professores Doutores: Stéphane Julia - FACOM/UFU , Eduardo Figueiredo - UFMG e Marcelo de Almeida Maia - FACOM/UFU, orientador do candidato.

Os examinadores participaram desde as seguintes localidades: Eduardo Figueiredo - Ottawa/Canadá. Os outros membros da banca e o aluno participaram da cidade de Uberlândia.

Iniciando os trabalhos o presidente da mesa, Prof. Dr. Marcelo de Almeida Maia , apresentou a Comissão Examinadora e o candidato, agradeceu a presença do público, e concedeu ao Discente a palavra para a exposição do seu trabalho. A duração da apresentação da Discente e o tempo de arguição e resposta foram conforme as normas do Programa.

A seguir o senhor presidente concedeu a palavra, pela ordem sucessivamente, aos examinadores, que passaram a arguir ao candidato. Ultimada a arguição, que se desenvolveu dentro dos termos regimentais, a Banca, em sessão secreta, atribuiu o resultado final, considerando o candidato:

Aprovado

Esta defesa faz parte dos requisitos necessários à obtenção do título de Mestre.

O competente diploma será expedido após cumprimento dos demais requisitos, conforme as normas do Programa, a legislação pertinente e a regulamentação interna da UFU.

Nada mais havendo a tratar foram encerrados os trabalhos. Foi lavrada a presente ata que após lida e achada conforme foi assinada pela Banca Examinadora.

Documento assinado eletronicamente por **Eduardo Magno Lages Figueiredo**, **Usuário Externo**, em 04/08/2025, às 11:38, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Marcelo de Almeida Maia**, **Professor(a) do Magistério Superior**, em 04/08/2025, às 11:39, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

Documento assinado eletronicamente por **Stéphane Julia**, **Professor(a) do Magistério Superior**, em 05/08/2025, às 09:41, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site https://www.sei.ufu.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **6488727** e o código CRC **40AD30FD**.

Referência: Processo nº 23117.046331/2025-41 SEI nº 6488727

Dedico este trabalho aos meus pais, Lázaro (in memoriam) e Rosângela.

Agradecimentos

Agradeço, em primeiro lugar, à minha mãe, Rosângela, por acreditar que eu seria capaz de abrir mão da estabilidade do cargo de servidor público para atuar no mercado de tecnologia, mesmo sem nenhuma experiência prévia, e por me apoiar na realização do sonho de estudar na Universidade Federal de Uberlândia.

Ao Professor Marcelo de Almeida Maia, agradeço pela paciência e pela orientação ao longo do mestrado, especialmente nos momentos em que conciliar as atividades profissionais com as acadêmicas se mostrou desafiador.

Aos meus gestores, agradeço pela compreensão e pelo apoio ao flexibilizarem minha carga horária, permitindo que eu participasse das atividades do programa de pós-graduação.

Aos amigos Rafael, Clênio e Douglas, meu sincero agradecimento por tornarem mais fácil minha adaptação em Uberlândia/MG. Vocês foram fundamentais não apenas para a conclusão deste curso, mas também para outras conquistas importantes ao longo desse período.

Aos professores do Instituto Federal de Educação, Ciência e Tecnologia do Triângulo Mineiro – Campus Patrocínio, agradeço pela oportunidade de conhecer o universo da pesquisa científica ainda no ensino técnico, por meio das atividades de iniciação científica.

Por fim, agradeço a todas as pessoas que, de forma direta ou indireta, contribuíram para que eu superasse os desafios impostos pela baixa visão.

"É preciso força pra sonhar e perceber que a estrada vai além do que se vê"
(Los Hermanos)

Resumo

A manutenção e evolução de sistemas de *software* frequentemente demanda mais esforço do que seu desenvolvimento inicial. Melhorar a qualidade do código-fonte por meio de suporte automatizado pode reduzir significativamente a dívida técnica, aumentar a manutenibilidade e aprimorar a produtividade dos desenvolvedores. Esta pesquisa apresenta uma abordagem que integra análise estática com Modelos de Linguagem de Grande Escala (LLMs) para automatizar a melhoria de código-fonte. A abordagem proposta processa iterativamente classes Java, extraindo problemas detectados pelo SonarQube e transformando-os em *prompts* para os LLMs, que geram versões aprimoradas do código. Cada versão é reanalisada, e o processo se repete até a convergência ou até atingir um limite predefinido de iterações. A abordagem foi avaliada experimentalmente e a configuração experimental inclui múltiplas combinações, envolvendo dois LLMs (GPT-4-mini e Gemini), variações de temperatura, estilo de *prompt* e número de iterações. As avaliações foram conduzidas utilizando múltiplos conjuntos de dados Java, com três execuções repetidas sobre o repositório Commons Lang para identificar padrões de comportamento. A análise se concentra na redução do número de problemas, na diminuição da dívida técnica (medida por uma métrica do SonarQube) e na evolução da severidade das *issues*. A correção funcional foi verificada manualmente por meio da inspeção do código aprimorado, a fim de garantir a preservação do comportamento. Os resultados demonstram que a combinação de SonarQube com LLMs é eficaz na redução de problemas no código — alcançando mais de 58% de redução média em cenários-chave — ao mesmo tempo em que preserva a funcionalidade. O processo iterativo mostrou-se bem-sucedido em guiar os modelos para melhorar incrementalmente a qualidade do código com base em *feedbacks* reais da análise estática. Este trabalho contribui com uma *pipeline* reprodutível e extensível, demonstrando o impacto de diferentes configurações dos LLMs e apoiando futuras pesquisas na integração entre IA e engenharia de qualidade de *software*.

Palavras-chave: Melhoria Automática de Código-Fonte. LLM. Análise Estática.

Abstract

Maintaining and evolving software systems often demands more effort than their initial development. Improving source code quality through automated support can significantly reduce technical debt, increase maintainability, and enhance developer productivity. This research presents an approach that integrates static analysis with Large Language Models (LLMs) to automate source code improvement. The proposed pipeline iteratively processes Java classes by extracting issues detected by SonarQube and transforming them into prompts for LLMs, which generate improved code versions. Each version is reanalyzed, and the process repeats until convergence or a predefined iteration limit is reached. The approach was empirically assessed and the experimental setup includes multiple configurations combining two LLMs (GPT-4-mini and Gemini), variation in temperature, prompt style, and number of iterations. Evaluations were conducted using multiple Java datasets, with three repeated runs for the Commons Lang repository to identify behavioral patterns. The analysis focuses on the number of issues reduction, decrease in technical debt (measured in a SonarQube metric), and the evolution of issue severity. Functional correctness was assessed manually by inspecting the improved code to ensure behavior preservation. The results demonstrate that combining SonarQube with LLMs is effective in reducing code issues—achieving over 58% average reduction in key scenarios—while preserving functionality. The iterative process proved successful in guiding the models to incrementally improve code quality based on real static analysis feedback. This work contributes a reproducible and extensible pipeline, offering insights into the impact of LLM configurations and supporting further research in the integration of AI and software quality engineering.

Keywords: Automatic source code improvement. LLM. Static Analysis.

Lista de ilustrações

Lista de tabelas

Lista de siglas

API Application Programming Interface

FFD Fractional Factorial Design

IDE Integrated Development Environment

LLMs Large Language Models

TD Technical Debt

Sumário

Capítulo 1

Introdução

A manutenção e a evolução são atividades centrais no ciclo de vida do desenvolvimento de software, frequentemente consumindo mais recursos do que a construção inicial do sistema (DEHAGHANI; HAJRAHIMI, 2013). Nesse contexto, a melhoria contínua da qualidade do código-fonte torna-se essencial para garantir legibilidade, mantenibilidade e redução da dívida técnica. Entre as estratégias amplamente adotadas para alcançar esses objetivos destacam-se as ferramentas de análise estática, como o SonarQube (SonarSource, 2024), e, mais recentemente, o uso de Modelos de Linguagem de Grande Escala (Large Language Models (LLMs)) como suporte às tarefas de melhoria e correção de código (ZUBAIR; AL-HITMI; CATAL, 2024; ETEMADI et al., 2022; SIMÕES; VENSON, 2024).

LLMs especificamente treinados com dados de programação, como Codex (OpenAI, 2021), CodeGen (CodeGen, 2024), StarCoder (Hugging Face, 2023) e Code LLaMA (Meta, 2023), têm apresentado desempenho promissor em tarefas de geração, explicação e correção de código-fonte (CHEN et al., 2021; OPENAI, 2023b; ZHONG et al., 2023; LU et al., 2023). Estudos recentes exploram o uso dessas ferramentas como assistentes no desenvolvimento de software, demonstrando avanços na compreensão contextual, identificação de defeitos e aplicação automatizada de correções (ROSS et al., 2023). Apesar do potencial, a adoção prática dos LLMs em ambientes reais ainda enfrenta desafios relacionados à confiabilidade de suas sugestões, à preservação da funcionalidade original e à integração com ferramentas tradicionais de análise de qualidade de código (CAI et al., 2025).

Adicionalmente, o SonarQube permanece como uma das ferramentas de análise estática mais utilizadas, com ampla adoção em projetos públicos e privados (YEBOAH; POPOOLA, 2023). Sua abordagem baseada em regras permite a detecção de falhas de segurança, vulnerabilidades, problemas de mantenibilidade e code smells. No entanto, apesar de oferecer diagnósticos detalhados e contextualizados, o SonarQube não realiza correções automáticas, exigindo que os desenvolvedores apliquem manualmente as melhorias recomendadas.

Diante da relevância do SonarQube para o desenvolvimento de software, é pertinente

investigar abordagens que combinem a precisão da análise estática com as capacidades generativas dos LLMs.

Este trabalho propõe uma abordagem para melhoria automática de código-fonte por meio da integração entre SonarQube e LLMs em um processo iterativo, automatizado e controlado. Para operacionalizar essa proposta, foi desenvolvida uma pipeline capaz de orquestrar a análise estática, a geração de prompts para os modelos, a substituição do código, a reanálise e o registro dos resultados em ciclos sucessivos. Os diagnósticos produzidos pelo SonarQube servem como instruções base para a construção dos prompts submetidos aos LLMs, que retornam versões refinadas do código, com a preservação do comportamento funcional original. Cada versão gerada é reanalisada, estabelecendo um ciclo contínuo de melhorias, que prossegue até a convergência ou o atingimento do número máximo de iterações.

A abordagem foi avaliada experimentalmente em um processo conduzido com diferentes modelos (GPT-4-mini e Gemini) e configurações (temperatura, prompt, número de iterações), aplicados a repositórios Java amplamente utilizados, como Commons Lang, Commons IO e Google Guava. A análise dos resultados considerou a redução no número total de issues, a diminuição da dívida técnica estimada e a evolução da severidade das issues ao longo das iterações. A preservação funcional foi verificada manualmente por meio da inspeção e comparação do código gerado.

Os resultados obtidos evidenciam o potencial da combinação entre análise estática e LLMs para automatizar o processo de melhoria de código, reduzindo o esforço manual associado à correção e contribuindo para o aumento da qualidade do software. A abordagem proposta busca suprir lacunas existentes na literatura, especialmente no que tange ao uso iterativo de LLMs guiado por ferramentas de análise estática, à avaliação do impacto de diferentes configurações e à execução sistemática de experimentos com validação funcional.

1.1 Motivação

A qualidade do código-fonte é um fator essencial para garantir a manutenibilidade, legibilidade e evolução sustentável de sistemas de software (PRESSMAN; MAXIM, 2021). Estudos clássicos e contemporâneos indicam que atividades de manutenção e evolução frequentemente representam entre 60% e 90% do custo total do desenvolvimento de software ao longo de seu ciclo de vida (TELANG; WATTAL, 2007). Esse custo tende a crescer com o tempo, especialmente quando a base de código acumula problemas estruturais e técnicos que não são tratados oportunamente (SANTOS et al., 2016).

Além disso, o acúmulo de dívida técnica (Technical Debt (TD)) pode ter impactos negativos sobre o time de desenvolvimento, impactando em sua produtividade e entusiasmo com o projeto (BESKER et al., 2020). Quando a qualidade de software não é devida-

mente aplicada na fase de construção, este esforço será necessário no futuro, pois as falhas do sistema irão criar demandas urgentes e impactar nas atividades dos desenvolvedores, além de implicar na confiança dos usuários (FERNÁNDEZ-SÁNCHEZ; GARBAJOSA; YAGÜE, 2015).

A Figura 1 ilustra o comportamento típico da curva de custo de manutenção, representando o impacto do acúmulo de dívida técnica ao longo dos anos, destacando que o custo da manutenção corretiva é sempre o pior dos casos e deve ser evitada. O custo ideal de manutenção deve ser uma divisão entre corretiva, preditiva e corretiva. Embora os custos com manutenção sempre estejam presentes no processo de desenvolvimento de software, estes podem ser minimizados quando a construção conta com adoção de boas práticas, por exemplo, presença de testes unitários que garantem que alterações não impactam outras funcionalidades.

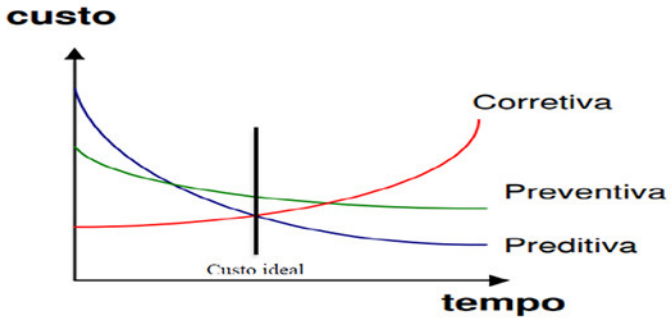

Figura 1 – Crescimento do custo de manutenção ao longo do tempo (DEHAGHANI; HAJRAHIMI, 2013).

Em projetos reais, a presença de problemas como duplicações, complexidade excessiva e violações de boas práticas tende a gerar essas dívidas técnicas, dificultando a evolução contínua e aumentando os custos operacionais (AKAIKINE, 2010). Ferramentas de análise estática, como o SonarQube, têm sido amplamente utilizadas para identificar esses problemas de forma automatizada. No entanto, a aplicação das correções continua majoritariamente dependente da intervenção manual de desenvolvedores (PELLEGRINI; JANES; TAIBI, 2018; CAMPBELL; PAPAPETROU, 2013).

Paralelamente, os LLMs vêm demonstrando avanços significativos em tarefas relacionadas à geração, explicação e correção de código-fonte, tornando-se ferramentas promissoras para apoio ao desenvolvimento de software (LIU et al., 2024; CHEN et al., 2023; MADAAN et al., 2023). A combinação entre a capacidade diagnóstica das ferramentas de análise estática e o poder generativo dos LLMs oferece uma oportunidade para automatizar partes do processo de melhoria de código, mantendo a precisão técnica e, ao mesmo tempo, reduzindo o esforço humano envolvido.

Nesse contexto, esta pesquisa busca explorar o potencial da integração entre SonarQube e LLMs em um processo iterativo e controlado de melhoria automática de código-fonte. A proposta é investigar como essa combinação pode contribuir para elevar a qualidade do software, ao mesmo tempo em que oferece um arcabouço sistemático

de experimentação com foco na redução de problemas técnicos, validação funcional e análise de impacto de diferentes configurações.

1.2 Objetivos

Os objetivos gerais e específicos deste trabalho são:

❑ Geral

- Investigar a eficácia de LLMs na melhoria automatizada de código-fonte com base em recomendações emitidas por ferramentas de análise estática, por meio da implementação e avaliação de uma pipeline iterativa, reprodutível e escalável aplicada a projetos reais de código aberto.

❑ Específicos

1. Projetar e implementar uma pipeline automatizada que integre ferramentas de análise estática com LLMs, orquestrando a extração de problemas, geração de sugestões de melhoria e reavaliação contínua do código-fonte.
2. Elaborar estratégias de construção de prompts que traduzam, com clareza e objetividade, as mensagens do SonarQube em instruções compreensíveis e acionáveis pelos modelos de linguagem.
3. Executar experimentos controlados sobre repositórios Java reais, avaliando diferentes configurações de modelo, temperatura, estilo de prompt e número de iterações.
4. Mensurar os impactos das melhorias aplicadas, utilizando métricas extraídas do SonarQube, como redução do número de *issues*, severidade, dívida técnica e evolução ao longo das iterações.
5. Verificar a preservação da funcionalidade do código após as melhorias aplicadas, identificando limitações da abordagem atual e propondo mecanismos de validação automatizada como trabalhos futuros.
6. Discutir a aplicabilidade e generalização da abordagem, destacando ameaças à validade, desafios enfrentados e potenciais extensões para outros domínios, linguagens ou ferramentas de análise.

1.3 Perguntas de Pesquisa e Hipóteses

Esta pesquisa busca avaliar o uso de LLMs como agentes de apoio à melhoria automatizada de código-fonte, orientada por recomendações provenientes de ferramentas de

análise estática. A seguir, apresentam-se as perguntas de pesquisa que guiam este estudo, juntamente com as hipóteses formuladas para investigá-las.

1.3.1 RQ1: Qual é o percentual de redução das *issues*?

Hipótese H1: A aplicação da pipeline com LLMs reduz o número total de *issues* detectadas pelo SonarQube.

Justificativa: Assume-se que, ao aplicar as recomendações de melhoria com apoio de LLMs, o código-fonte será modificado de forma a corrigir parte relevante dos problemas identificados pela análise estática. Assim, espera-se uma redução mensurável no número total de *issues* reportadas após as iterações.

1.3.2 RQ2: A redução das *issues* impacta proporcionalmente na diminuição da dívida técnica?

Hipótese H2: A redução das *issues* está correlacionada com uma diminuição proporcional na dívida técnica estimada.

Justificativa: Como o SonarQube associa a cada *issue* uma estimativa de esforço (em minutos) para correção — representando a dívida técnica —, pressupõe-se que a eliminação de problemas também resulte em uma redução proporcional da dívida acumulada no projeto.

1.3.3 RQ3: Como a configuração experimental impacta a qualidade das melhorias de código?

Hipótese H3: Diferentes configurações experimentais (modelo, temperatura, estilo de prompt e número de iterações) impactam a qualidade das melhorias aplicadas.

Justificativa: Como os LLMs são sensíveis às variações nos parâmetros de execução, é esperado que determinadas combinações influenciem a efetividade da melhoria do código. A qualidade será avaliada por métricas como número de *issues* resolvidas, severidade das correções e redução de dívida técnica.

1.3.4 RQ4: O processo iterativo de melhoria compromete a funcionalidade do código?

Hipótese H4: O processo iterativo de melhoria não compromete a funcionalidade observável do código modificado.

Justificativa: Apesar das alterações realizadas no código, espera-se que os modelos preservem o comportamento funcional original. Essa hipótese será verificada por meio de

execução direta dos métodos aprimorados e inspeção manual, considerando a ausência de quebras perceptíveis de funcionalidade.

1.4 Organização da Dissertação

Esta dissertação está estruturada em capítulos, organizados da seguinte maneira:

- Capítulo 2 – Fundamentação Teórica. Apresentação dos conceitos fundamentais para entendimento do propósito do trabalho e seus impactos em pesquisas envolvendo o uso de Modelos de Linguagem na melhoria automática de código-fonte.
- Capítulo 3 – Metodologia. Apresenta a abordagem metodológica adotada no desenvolvimento da pesquisa, destacando a pipeline proposta para orquestrar as atividades de melhoria de código.
- Capítulo 4 – Resultados. Apresenta os resultados alcançados.
- Capítulo 5 – Trabalhos Relacionados. Apresentação de trabalhos relacionados com a temática abordada e que foram motivadores para este estudo.
- Capítulo 6 – Conclusão. Apresenta a conclusão deste estudo e destaca oportunidades de trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este capítulo apresenta conceitos fundamentais para entendimento do trabalho, destacando a importância da qualidade do código-fonte, o funcionamento da análise estática e as oportunidades advindas dos LLMs.

2.1 Qualidade de Código e Refatoração

A qualidade do código-fonte é um dos aspectos mais importantes para garantir a manutenção, evolução e confiabilidade de sistemas de software. Código de qualidade é aquele que é claro, legível, eficiente, modular e aderente às boas práticas de desenvolvimento, facilitando o entendimento por parte dos desenvolvedores e a realização de modificações futuras com baixo risco de introdução de erros (FOWLER, 2018). A qualidade do código impacta diretamente nos custos de manutenção, na produtividade da equipe e na satisfação do cliente, uma vez que um código bem estruturado reduz o esforço necessário para a correção de defeitos e para a implementação de novas funcionalidades (MENS; TOURWÉ, 2004).

Entretanto, durante o ciclo de vida de um software, é comum que o código sofra degradação de sua qualidade, fenômeno conhecido como "code rot" ou "software decay" (BROWN et al., 1998). Essa degradação ocorre devido a mudanças contínuas, pressões para entregas rápidas e falta de disciplina no processo de desenvolvimento, resultando em código complexo, duplicado, com baixa coesão e alta acoplamento, dificultando a manutenção e aumentando o risco de falhas (FOWLER, 2018).

A refatoração é uma prática fundamental para mitigar esses problemas e melhorar a qualidade do código ao longo do tempo. Segundo Fowler (FOWLER, 2018), refatoração é o processo de modificar a estrutura interna do código sem alterar seu comportamento externo, visando torná-lo mais legível, compreensível e fácil de manter. Por meio de pequenas transformações incrementais, a refatoração promove a remoção de duplicações, a simplificação de expressões complexas, a melhoria da modularidade e a adequação a

padrões de projeto, contribuindo para um código mais sustentável (PRESSMAN; MAXIM, 2021).

Diversos estudos indicam que a adoção sistemática da refatoração melhora a qualidade do software e reduz a ocorrência de defeitos (FOWLER, 2018; CHEN et al., 2016; MOTOGNA; BERCIU; MOLDOVAN, 2024). Além disso, a refatoração pode ser potencializada por ferramentas automatizadas que detectam smells de código — indícios de problemas estruturais — e recomendam ou aplicam melhorias de forma assistida (KHALEEL; MAHMOOD, 2024). Essa automação é essencial para viabilizar a prática em projetos de grande porte, nos quais a análise manual seria inviável.

Portanto, a combinação entre avaliação da qualidade do código e a aplicação contínua de refatoração configura-se como uma estratégia eficaz para manter a saúde dos sistemas de software e garantir sua evolução sustentável ao longo do tempo.

2.2 Análise Estática

A análise estática de código é uma técnica amplamente utilizada para inspecionar o código-fonte sem executá-lo, com o objetivo de identificar defeitos, más práticas e potenciais vulnerabilidades (FOSTER; HICKS; PUGH, 2007). Essa abordagem oferece uma forma eficiente de detectar problemas de qualidade nas fases iniciais do desenvolvimento, contribuindo para a redução dos custos de manutenção e o aumento da confiabilidade do software (MENS; TOURWÉ, 2004).

Ferramentas de análise estática baseiam-se em um conjunto de regras e heurísticas para examinar o código-fonte e identificar padrões potencialmente problemáticos. Entre os principais tipos de problemas que podem ser detectados por essas ferramentas, destacam-se:

- **Erros lógicos e defeitos comuns (bugs)**: problemas que podem resultar em falhas de execução ou comportamentos incorretos, como desreferenciamento de ponteiros nulos, divisões por zero, ou uso de variáveis não inicializadas.

- **Más práticas de codificação (code smells)**: padrões que, embora não representem defeitos imediatos, indicam fragilidades estruturais no código, como métodos longos, classes excessivamente complexas ou violação de princípios de coesão e acoplamento.

- **Vulnerabilidades de segurança**: construções que podem ser exploradas por atacantes, como injeção de código, uso inseguro de APIs criptográficas ou ausência de validações de entrada.

- **Métrica de complexidade e manutenibilidade**: indicadores como complexidade ciclomática, duplicação de código, e cobertura de testes automatizados que ajudam a avaliar a qualidade estrutural do software.

Um exemplo amplamente utilizado de ferramenta de análise estática é o SonarQube, que permite integrar a inspeção de código ao processo de desenvolvimento contínuo, fornecendo feedback automático por meio de dashboards e relatórios de qualidade. No entanto, uma limitação significativa da análise estática tradicional reside no fato de que, embora os problemas sejam detectados automaticamente, a correção das falhas geralmente exige intervenção manual dos desenvolvedores, o que pode ser oneroso, especialmente em bases de código extensas ou legadas (CAMPBELL; PAPAPETROU, 2013).

2.2.1 SonarQube

O SonarQube é um software de código aberto para análise estática de código-fonte. Desenvolvido pela empresa SonarSource, atualmente é uma das soluções para revisão de código mais usadas na área científica e na indústria (SonarSource, 2024).

O SonarQube pode ser usado/implantado de diversas formas, seja através de uma imagem docker, seja em um ambiente de nuvem ou através de um servidor privado (LOMIO; MORESCHINI; LENARDUZZI, 2022). Suas análises permitem identificar diversas anomalias, como: bugs, código duplicado, complexidade ciclomática, entre diversos outros problemas que podem ocorrer durante o processo de desenvolvimento de software.

A solução de análise de código-fonte é compatível com mais de 30 linguagens de programação, possuindo em torno de 500 regras de análise para cada uma (CAMPBELL; PAPAPETROU, 2013). Mais que garantir um código que compile sem falhas, tais análises permitem construir um código limpo, confiável, seguro, de fácil entendimento e modular, flexibilizando a aplicação para atender demandas de novas funcionalidades, que surgem constantemente na dinâmica do desenvolvimento ágil (LOMIO; MORESCHINI; LENARDUZZI, 2022; MARCILIO et al., 2019).

As regras do SonarQube identificam falhas de conformidade e as classificam nos níveis bloqueador, crítico, maior, menor e informacional. Tal classificação é muito valiosa, visto que podemos ter uma quantidade elevada de itens fora do padrão, assim, precisamos priorizar tal dívida técnica, logo, começar por trechos classificados com maior severidade, pode ser uma estratégia válida para lidar com prazos críticos ou falta de recursos (ALFAYEZ et al., 2023).

2.3 Modelos de Linguagem de Grande Escala

Os Modelos de Linguagem de Grande Escala, mais conhecidos como LLMs (Large Language Models), são modelos de aprendizado de máquina (machine learning) que usam

técnicas de aprendizado profundo (deep learning) para entender e processar linguagem natural (WU et al., 2023).

LLMs são treinados em grandes quantidades de dados para que possam aprender padrões de uso de frases e palavras, assim, quando recebem uma nova entrada, conseguem gerar uma saída que faça sentido (WU et al., 2023). Estes modelos geralmente são usados em tarefas diversas, como tradução de texto, geração de resumos, escrita de redações, conversas ou em outras ações que sejam baseadas no processamento de linguagem natural (OUYANG et al., 2022).

Dentre tais possíveis aplicações dos LLMs, também estão presentes capacidades como traduzir código-fonte entre diferentes linguagens de programação ou até mesmo criar código com base em uma descrição textual fornecida. Todavia, LLMs podem apresentar saídas com falhas, no caso de código-fonte, o mesmo ser incompatível com os requisitos do usuário ou apresentar características incompatíveis com boas práticas de desenvolvimento (JIANG; WANG; WANG, 2023).

Assim, torna-se válido adotar estratégias para melhorar as saídas dos LLMs, conforme abordado no trabalho aqui proposto.

2.3.1 GPT

Os modelos de linguagem baseados na arquitetura Transformer têm revolucionado a área de Processamento de Linguagem Natural (PLN) nos últimos anos, proporcionando avanços significativos na geração, compreensão e manipulação de texto (VASWANI et al., 2017). Entre esses modelos, destaca-se a série GPT (Generative Pre-trained Transformer), desenvolvida pela OpenAI, que utiliza um enfoque de pré-treinamento não supervisionado seguido de ajuste fino para diversas tarefas específicas (RADFORD et al., 2018).

O GPT é um modelo autoregressivo que gera texto prevendo a próxima palavra em uma sequência com base no contexto anterior. Sua arquitetura é fundamentada no mecanismo de atenção do Transformer, que permite capturar dependências de longo alcance no texto de forma eficiente (VASWANI et al., 2017). O pré-treinamento do GPT consiste na exposição do modelo a grandes volumes de dados textuais para que ele aprenda padrões linguísticos, sem a necessidade de rótulos manuais, o que aumenta sua escalabilidade e capacidade de generalização (RADFORD et al., 2019).

Após o pré-treinamento, o modelo pode ser ajustado (fine-tuned) para tarefas específicas, como tradução, sumarização, resposta a perguntas, entre outras, apresentando resultados competitivos com abordagens supervisionadas tradicionais (BROWN et al., 2020). Além disso, versões mais recentes, como o GPT-3 e GPT-4, possuem bilhões de parâmetros e são capazes de realizar tarefas complexas, inclusive com poucos exemplos (few-shot learning), o que amplia seu potencial de aplicação em diversos domínios (BROWN et al., 2020; OPENAI, 2023a; OpenAI, 2024b).

Na área de engenharia de software, modelos como o GPT têm sido explorados para auxiliar na geração automática de código, revisão, documentação e até na refatoração, demonstrando-se uma ferramenta promissora para aumentar a produtividade dos desenvolvedores e a qualidade do software produzido (CHEN et al., 2021).

2.3.2 Gemini

O Gemini é um modelo multimodal de inteligência artificial desenvolvido pela Google DeepMind, projetado para integrar e processar diferentes tipos de dados, como texto, imagens e outros formatos, utilizando uma arquitetura baseada em Transformers (Google, 2023). Essa capacidade multimodal permite que o Gemini realize tarefas complexas que envolvem a compreensão e geração de conteúdo em múltiplas modalidades, o que representa um avanço em relação a modelos focados exclusivamente em texto.

Inspirado nas arquiteturas de modelos de linguagem como o GPT, o Gemini combina aprendizado profundo e atenção para oferecer melhor compreensão contextual e raciocínio, incluindo habilidades que envolvem múltiplas fontes de dados simultaneamente (Google, 2023). Isso o torna particularmente útil para aplicações que exigem integração entre linguagem natural e outras formas de dados, como visão computacional, interpretação multimodal e assistentes inteligentes.

Além do desempenho avançado em benchmarks multimodais, o Gemini incorpora técnicas para melhoria do alinhamento com os objetivos humanos e mitigação de vieses, seguindo as tendências atuais em desenvolvimento ético e responsável de IA (Google DeepMind, 2024). O modelo está sendo explorado em diversas áreas, incluindo engenharia de software, onde pode auxiliar na geração e melhoria de código a partir de descrições em linguagem natural e outros contextos multimodais.

Capítulo 3

Metodologia

Esta pesquisa adotou uma abordagem experimental, utilizando análise de dados quantitativos e qualitativos para investigar a efetividade de Modelos de Linguagem de Grande Escala (LLMs) na melhoria automatizada de código-fonte com base em recomendações emitidas por uma ferramenta de análise estática. O estudo foi conduzido em quatro etapas principais: (i) o desenvolvimento de uma *pipeline* iterativa automatizada de aprimoramento de código-fonte; (ii) a execução sistemática de experimentos sobre repositórios de projetos reais; (iii) a análise quantitativa dos resultados com base em métricas extraídas de avaliações realizadas pelo SonarQube sobre as versões modificadas; e (iv) a análise qualitativa da preservação da funcionalidade após as melhorias aplicadas ao código.

3.1 Pipeline de Melhoria Iterativa Automatizada

A *pipeline* automatizada desenvolvida no contexto desta pesquisa teve como papel central a orquestração de todas as etapas do processo de melhoria automatizada do código-fonte, integrando componentes de análise estática, geração de sugestões com modelos de linguagem e avaliação iterativa dos resultados. Sua concepção foi orientada por princípios de reprodutibilidade, escalabilidade e automação completa, de modo a viabilizar a condução de experimentos de forma sistemática e padronizada, com diferentes combinações de parâmetros experimentais previamente definidos.

Um esquema representativo da *pipeline* é apresentado na Figura 2, onde é ilustrado um fluxograma com as macro etapas de processamento realizadas. Este modelo de orquestração é capaz de lidar com repositórios com inúmeras classes, realizando a melhoria do código de cada uma de maneira iterativa, conforme valores de parâmetros definidos.

A arquitetura pode ser organizada em 6 itens principais – detalhados abaixo -, sendo: (i) Análise Estática Inicial; (ii) Extração e Filtragem de Issues; (iii) Geração do Prompt; (iv) Interação com o Modelo de Linguagem; (v) Atualização e Reanálise e (vi) Critérios de Parada.

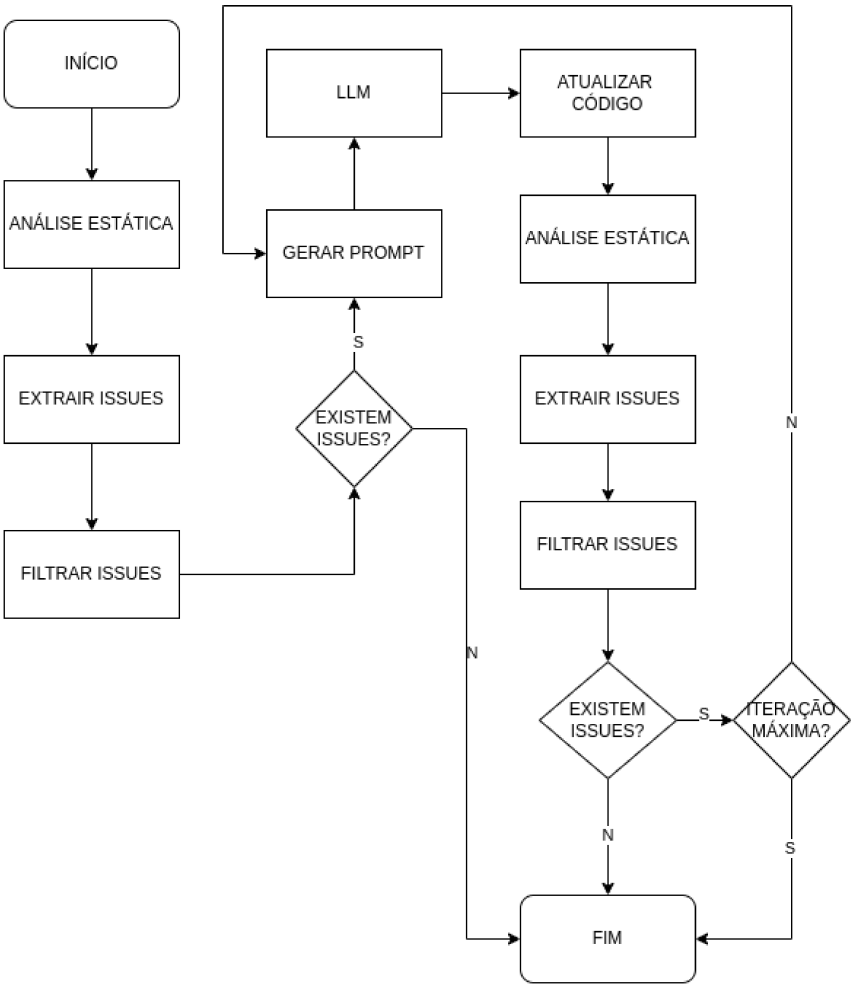

Figura 2 – Esquema representativo da pipeline desenvolvida.

3.1.1 Análise Estática Inicial

Após listar todas as classes de determinado repositório, a pipeline realiza o processamento de cada uma, sendo que o primeiro passo necessário é a verificação se existem issues – problemas de padronização, segurança ou falha - no código. Essa verificação é realizada por meio do uso SonarScanner em conjunto com o SonarQube. O SonnarScanner foi usado para verificar a presença de issues no código dada sua capacidade de realizar análise do código de determinada classe sem a necessidade de compilação do projeto. Após realizar verificação da qualidade do código, os resultados da análise são encaminhados ao SonarQube, que executa em um container Docker.

Para apuração das issues de cada classe, foi criado um projeto base usando Apache Maven, onde a classe principal é substituída pelo código produzido em cada uma das iterações. Essa estratégia foi adotada para garantir análise do código em um ambiente controlado e sem impacto de dependências.

3.1.2 Extração e Filtragem de Issues

Após avaliar a qualidade do código e enviar os resultados ao SonarQube - por meio do SonarScanner - a pipeline realiza um delay de 60 segundas – para que as métricas do Sonar sejam atualizadas – e, então, realiza a coleta das issues via API. O endpoint *api/issues/search* permite obter todos as issues da última avaliação realizada. Na Figura 3 é apresentado um exemplo de resultado de avaliação de qualidade de código obtido via API do SonarQube.

```
"issues": [
  {
    "key": "AXxF9L5j8T1eXK4l8gYz",
    "rule": "java:S2095",
    "severity": "MAJOR",
    "component": "com.example:myproject:src/main/java/com/example
       /DatabaseManager.java",
    "project": "com.example:myproject",
    "line": 58,
    "hash": "ad0234829205b9033196ba818f7a872b",
    "textRange": {
      "startLine": 58,
      "endLine": 58,
      "startOffset": 12,
      "endOffset": 45
    },
    "flows": [],
    "status": "OPEN",
    "message": "Close this 'Connection' object or handle it
       inside a try-with-resources statement.",
    "effort": "30min",
    "debt": "30min",
    "author": "developer@example.com",
    "tags": ["bug", "resource-leak"],
    "creationDate": "2025-05-27T13:45:21+0000",
    "updateDate": "2025-05-27T13:45:21+0000",
    "type": "BUG",
    "scope": "MAIN"
  }
]
```

Figura 3 – Exemplo de issue do tipo `MAJOR` no formato JSON retornado pela API do SonarQube.

Na Figura 3, percebe-se a quantidade de informações relevantes retornadas pela API, incluindo mensagem de erro, linha, tipo, severidade e as estimativas de débito e esforço necessário para o ajuste, calculados com base em regras do Sonar para a regra violada.

Após obter a lista de issues, a aplicação realizada um filtro, considerando alguns critérios, como: status, tipo e severidade. São consideradas somente issues com o status

em aberto, tipo definido como Bug, Vulnerabilidade e Code Smell, além das severidades Blocker, Critical e Major. Tais critérios foram definidos com base nos itens mais críticos que podem ser apontados pelas métricas do Sonar, além disso, itens classificados como informacional foram desconsiderados, visto que seu conteúdo não relata um problema.

A função de filtrar issues é importante para determinar se a classe em questão deve ser processada. Tal abordagem foi utilizada buscando reduzir a aplicação de ajustes que não impactam na funcionalidade e manutenibilidade do código. Caso sejam retornadas issues pelo filtro, é formada uma lista com a mensagem de sugestão de ajuste e o número da linha de cada problema, sendo estes elementos usados na construção do prompt.

3.1.3 Geração do Prompt

Após a identificação das *issues*, a *pipeline* constrói dinamicamente um *prompt* em linguagem natural e o envia ao modelo de linguagem (LLM) selecionado para orientar a melhoria do código. O objetivo do *prompt* é comunicar de forma clara as *issues* encontradas no código e especificar as melhorias que devem ser aplicadas. Dois modelos de prompt foram elaborados e são destacados abaixo:

Prompt 1 (Disparo Zero)

Aplique as seguintes melhorias no código e retorne apenas o código Java: {issues_message}.

Prompt 2 (Baseado em Papeis)

Você está atuando como um assistente de desenvolvimento de software e deve aplicar melhorias no código utilizando a seguinte lista de issues: {issues_message}. Você deve modificar apenas o código para corrigir as issues reportadas, sem remover nenhuma funcionalidade, a menos que isso seja explicitamente indicado. Somente o código Java deve ser retornado, sem quaisquer explicações, textos adicionais ou perda de qualquer comportamento previamente implementado.

Neste caso, foram usadas duas estratégias de engenharia de prompt: disparo zero e baseado em papéis. O conceito de prompt de disparo zero, ou zero-shot prompting, refere-se a uma técnica de interação com modelos de linguagem na qual se solicita a execução de uma tarefa sem fornecer exemplos explícitos dessa tarefa no próprio prompt. O modelo realiza a inferência apenas com base na descrição textual da tarefa, apoiando-se no conhecimento adquirido durante sua fase de treinamento (WANG et al., 2023; LIU et al., 2024).

O prompt baseado em papéis é uma técnica de engenharia de prompt que estrutura a interação com o LLM atribuindo papéis – roles – durante a conversa (SHA; ZHANG, 2024). Tal metodologia de trabalho busca fornecer contexto semântico e funcional ao

modelo, buscando reduzir eventuais desvios e melhorar os resultados. Ao definir instruções específicas por meio do papel system, por exemplo, é possível orientar o comportamento do modelo para simular personas específicas, controlar o tom da resposta ou restringir o escopo da tarefa (DAS; SRIHARI, 2024). Essa abordagem é especialmente útil em contextos mais complexos ou de múltiplas interações, como agentes autônomos ou sistemas de diálogo, nos quais a manutenção de consistência e intenção é fundamental.

3.1.4 Interação com o Modelo de Linguagem e Reanálise

Após elaboração do prompt, este é enviado ao *LLM* parametrizado por meio de *Application Programming Interface (API).* No corpo da requisição são enviados dois parâmetros, *system* e *user*. No parâmetro *system* é armazenado a solicitação com a lista de *issues* encontradas na iteração corrente, ao passo que no user é armazenado o código-fonte.

Além desses atributos, também são enviados os valores de temperatura, o modelo de linguagem a ser utilizado e a quantidade máxima de tokens. O número máximo de tokens impacta diretamente os resultados, pois está relacionado ao tamanho máximo da resposta que o modelo pode gerar. Durante os experimentos iniciais com a *pipeline* foi observado que parte das respostas era truncada quando se utilizava um valor muito baixo para o parâmetro *max_tokens*. Por esse motivo, optou-se por definir seu valor como 5000. Além disso, foi realizado um pré-processamento nos *datasets* para remover os comentários de todas as classes.

Após obter a resposta do LLM, o código é extraído e inserido na classe main do projeto base, sendo submetido em uma nova análise do Sonar. Caso existem *issues*, a execução do processamento continua até que a quantidade máxima de iterações seja atingida ou que não existe mais problemas no código.

3.2 Design Experimental

Os experimentos foram sistematicamente planejados para avaliar a eficácia da melhoria automática de código assistida por LLM sob diferentes combinações de parâmetros, buscando compreender como a variação destes influencia os resultados.

Os cenários foram definidos com base em um *desenho fatorial completo* (*Full Factorial Design*) (MONTGOMERY, 2017), considerando quatro fatores principais: o modelo de linguagem utilizado (*LLM*), a temperatura de geração, o tipo de prompt e o número máximo de iterações. Cada fator foi avaliado com dois níveis distintos, sendo i) LLM (GP4-4-mini ou gemini); ii) Temperatura - 0.1 para comportamento determinítico e 0.3 para comportamento exploratório; iii) prompt (direto e baseado em papeis) e iv) quantidade de iterações - 2 ou 3 por classe.

Em um arranjo fatorial completo com quatro fatores binários, o número total de combinações possíveis seria de 16 (2^4), No entanto, optou-se pela execução de 8 cenários

experimentais, previamente definidos de forma a manter o equilíbrio entre os níveis dos fatores e garantir diversidade nas combinações. Essa decisão foi motivada por questões de viabilidade computacional e pela necessidade de reduzir o custo experimental, sem comprometer a análise dos efeitos principais. Ainda que essa escolha limite a avaliação completa de interações de ordem superior entre os fatores, os cenários selecionados foram suficientes para identificar padrões relevantes e avaliar de forma comparativa a influência das variáveis sobre o processo de melhoria automática.

As combinações executadas foram rotuladas como Cenários 1 a 8, conforme detalhado na Tabela 1.

Tabela 1 – Relação de Cenários Experimentais.

Experimento	LLM	Iterações	Temperatura	Prompt
1	GPT-4-mini	2	0.1	1
2	GPT-4-mini	2	0.3	2
3	GPT-4-mini	5	0.1	2
4	GPT-4-mini	5	0.3	1
5	Gemini-2.0-flash-lite	2	0.1	2
6	Gemini-2.0-flash-lite	2	0.3	1
7	Gemini-2.0-flash-lite	5	0.1	1
8	Gemini-2.0-flash-lite	5	0.3	2

Cada um dos experimentos foi executado em um conjunto de classes Java. Para o conjunto de dados `Commons Lang`, todas as oito configurações foram executadas três vezes para garantir a robustez estatística, permitindo a análise de médias, desvios-padrão e padrões de comportamento. Para a fase exploratória, usando `Commons IO` e `Guava`, cada configuração foi executada uma vez para fornecer uma perspectiva complementar sobre como a diversidade de código e o estilo do projeto influenciam a eficácia das melhorias.

Todo o processo experimental foi totalmente automatizado por meio do pipeline desenvolvido, que cuidou da leitura de entrada, análise estática via SonarQube, geração de prompts, interação com o modelo, substituição de código e registro de métricas — incluindo o número de problemas, níveis de gravidade, esforço estimado (dívida técnica) e contagem de iterações. Os dados de saída foram armazenados em arquivos estruturados `.csv`, e cada versão de código gerada ao longo das iterações foi salva no formato `.txt` para permitir inspeção, validação e rastreabilidade histórica.

3.2.1 Questões de Pesquisa

Este trabalho busca responder às seguintes questões de pesquisa:

- **RQ1: Qual é o percentual de redução das *issues*?** Esta questão avalia a capacidade dos Modelos de Linguagem de Grande Escala (LLMs) em reduzir

a quantidade de defeitos identificados por ferramentas de análise estática após a aplicação das melhorias automatizadas.

- **RQ2: A redução das *issues* impacta proporcionalmente na diminuição da dívida técnica?** Esta questão examina se a redução observada nos defeitos resulta, de forma proporcional, em uma diminuição na estimativa de dívida técnica do projeto, conforme apontada pelas métricas do SonarQube.
- **RQ3: Como a configuração experimental impacta a qualidade das melhorias de código?** Esta questão analisa o efeito da variação dos parâmetros experimentais — como o modelo de linguagem utilizado, a temperatura, a formulação do *prompt* e o número de iterações — na qualidade das melhorias geradas.
- **RQ4: O processo iterativo de melhoria compromete a funcionalidade do código?** Esta questão investiga se o processo iterativo de aprimoramento introduz quebras funcionais, regressões ou falhas semânticas que comprometam o comportamento original do código.

3.2.2 Repositórios e Fontes de Dados

Os experimentos utilizaram repositórios Java reais extraídos de projetos populares de código aberto com históricos de manutenção ativos. Os principais repositórios selecionados foram Apache Commons Lang, Apache Commons IO e Google Guava.

O Apache Commons Lang estende as funcionalidades principais do Java com utilitários para manipulação de strings, números e objetos. O Apache Commons IO oferece utilitários de E/S para operações com arquivos. O Google Guava é um conjunto abrangente de bibliotecas principais do Google, com coleções, imutáveis, suporte a gráficos e utilitários de simultaneidade, hash e strings.

Esses repositórios foram selecionados devido à sua diversidade de classes, cobertura de código e compatibilidade com a análise estática do SonarQube. Todos os arquivos foram armazenados e processados localmente pelo pipeline em um ambiente controlado.

3.2.3 Métricas de Avaliação

Para avaliar a efetividade das melhorias propostas, foram extraídas quatro métricas principais a cada iteração de refatoração, com base nos resultados da ferramenta SonarQube:

Quantidade total de issues: representa o número absoluto de problemas identificados pelo SonarQube em uma determinada versão do código. Essa métrica permite observar a evolução bruta do volume de defeitos detectados ao longo das iterações.

Percentual de redução de issues: calcula-se pela razão entre a quantidade de issues eliminadas em relação à quantidade inicial. Essa métrica normaliza o impacto das

melhorias, permitindo comparações proporcionais entre classes com diferentes tamanhos ou níveis iniciais de complexidade.

Redução por severidade: consiste na análise da diminuição de issues agrupadas por severidade (*BLOCKER*, *CRITICAL*, *MAJOR*), conforme a taxonomia do SonarQube. Essa distinção é relevante pois prioriza a eliminação de problemas com maior impacto potencial na confiabilidade ou segurança do sistema.

Redução da dívida técnica: mensura a diminuição do esforço estimado (em minutos) para correção das issues reportadas, com base nos cálculos fornecidos pelo SonarQube. Essa métrica oferece uma estimativa do ganho prático em termos de manutenção e evolução do código.

Além das métricas automatizadas, a preservação da funcionalidade foi avaliada manualmente por meio da comparação dos resultados dos códigos gerados em cada iteração. Essa análise foi conduzida em um Ambiente de Desenvolvimento Integrado (Integrated Development Environment (IDE)), o que permitiu identificar, com precisão, os pontos de modificação realizados em cada ciclo do processo iterativo. Embora não tenham sido observadas falhas funcionais nas amostras analisadas, reconhece-se que a ausência de um mecanismo sistemático de validação semântica representa uma limitação deste estudo.

Capítulo 4

Resultados

Este capítulo apresenta os resultados obtidos a partir de aplicação de Modelos de Linguagem de Grande Escala (LLMs) na tarefa de melhoria automatizada de código-fonte com base nas issues reportadas pelo SonarQube. Os resultados estão estruturados de maneira que sejam capazes de responder as questões de pesquisa levantadas durante o desenvolvimento deste trabalho. A RQ1 discorre sobre a redução de issues no processo de melhoria iterativa, a RQ2 apresenta os resultados da correlação de redução de issues e dívida técnica, a RQ3 descreve como as configurações de cada experimento impactam os resultados alcançados e, finalmente, a RQ4 retrata os impactos da melhoria iterativa e a preservação de funcionalidade do código-fonte.

4.1 RQ1 - O uso de LLMs reduz efetivamente as issues?

Para responder à RQ1, foi avaliada a porcentagem de issues identificadas pelo SonarQube que foram corrigidas com sucesso por Modelos de Linguagem de Grande Escala (LLMs) por meio de ciclos iterativos de melhorias. A análise considerou a diferença percentual entre o número de issues na iteração inicial (0) e na última iteração registrada em cada execução. Devido a limitações financeiras e de recursos computacionais, a maioria dos experimentos foi executada apenas uma vez; assim, a análise prioriza a variação entre diferentes datasets e configurações experimentais, oferecendo uma visão comparativa da efetividade dos modelos.

A Figura 4 apresenta o percentual de redução de issues por experimento para o conjunto de dados Apache Commons Lang. Observa-se que sete dos oito experimentos obtiveram reduções superiores a 50%, sendo que um deles ultrapassou a marca de 80%.

O Experimento 7 destacou-se por obter a maior média de redução (81,29%), superando em mais de 20 pontos percentuais os demais experimentos do mesmo conjunto. Por apresentar a diferença mais expressiva, esse cenário foi executado três vezes, com

resultados consistentes: 82,14%, 72,95% e 88,77%, reforçando sua eficácia na redução de problemas identificados. Esses dados posicionam o Experimento 7 como a configuração mais promissora para o Commons Lang em termos de redução de *issues*.

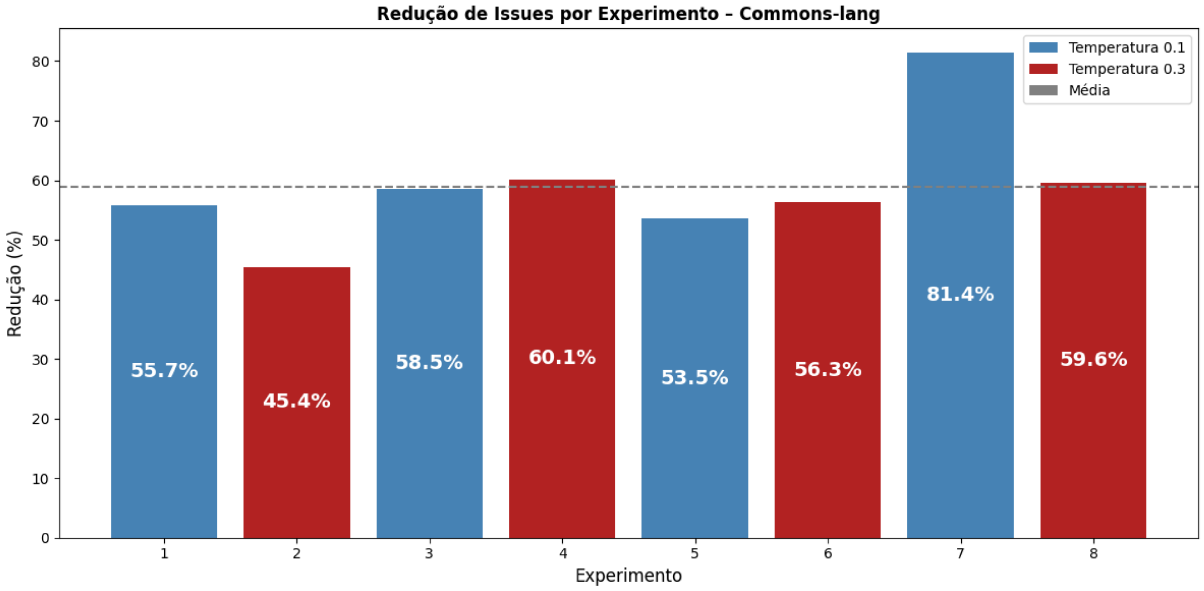

Figura 4 – Redução de issues observada no conjunto de dados Apache Commons-Lang.

A Tabela 2 apresenta os resultados de redução percentual de *issues* no dataset Apache Commons-Lang para as oito configurações experimentais avaliadas. Os dados demonstram que todas as combinações de parâmetros resultaram em reduções relevantes, variando entre aproximadamente 49,49% (Experimento 2) e 81,29% (Experimento 7), o que indica que a *pipeline* foi capaz de aplicar melhorias efetivas nos arquivos analisados. De modo geral, os Experimentos 3, 4, 6, 7 e 8 apresentaram médias superiores a 60%, com destaque para o Experimento 7, que obteve a maior média de redução (81,29%) e também o maior valor individual registrado (88,78%). Esses resultados evidenciam que certas configurações experimentais podem ser mais eficazes do que outras na eliminação de problemas reportados pelo SonarQube.

Ao observar o desvio padrão dos experimentos, é possível identificar variações relevantes na estabilidade dos resultados. O Experimento 3 apresentou o menor desvio padrão (1,93), indicando resultados mais consistentes entre as três execuções. Por outro lado, os Experimentos 4 e 7 apresentaram os maiores desvios (7,51 e 7,94, respectivamente), sugerindo maior sensibilidade às variações nos modelos ou ao comportamento não determinístico dos LLMs. Isso reforça a importância da realização de múltiplas execuções por configuração para capturar a variabilidade do processo, além de mostrar que combinações com médias elevadas nem sempre garantem consistência. No geral, os resultados indicam que a abordagem iterativa com LLMs é capaz de melhorar significativamente a qualidade interna do código, embora o impacto dependa fortemente da configuração aplicada.

Tabela 2 – Redução de Issues por Experimento no Dataset Apache Commons-Lang

Experimento	Exec. 1	Exec. 2	Exec. 3	Média	Desvio Padrão
1	54,08	56,12	62,76	57,65	4,53
2	44,90	47,96	55,61	49,49	5,52
3	61,22	61,73	64,80	62,59	1,93
4	62,24	71,94	77,04	70,41	7,52
5	53,57	46,43	60,20	53,40	6,89
6	57,14	67,35	61,22	61,90	5,14
7	82,14	72,96	88,78	81,29	7,94
8	62,24	57,14	65,31	61,56	4,12

A Figura 5 ilustra os resultados obtidos a partir dos experimentos com o conjunto de dados Apache Commons IO. Neste caso, foi alcançada uma redução média superior a 70%, com variações entre 55% e 83%, dependendo da configuração.

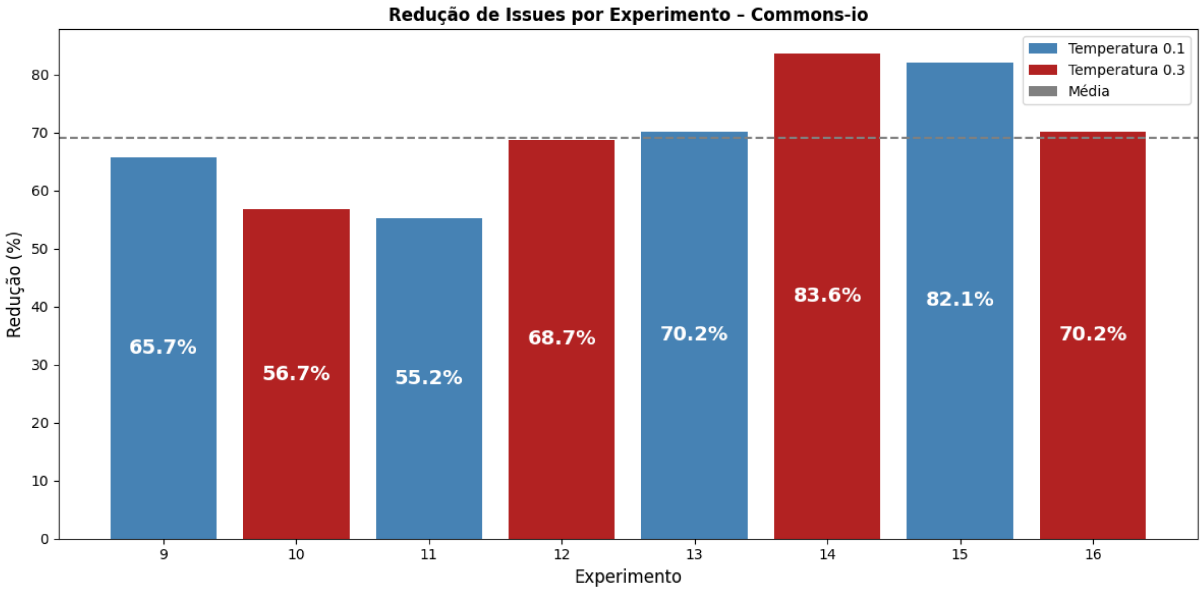

Figura 5 – Redução de issues observada no conjunto de dados Apache Commons IO.

A Figura 6 apresenta os resultados obtidos a partir dos experimentos com o conjunto de dados Google Guava, nos quais foi observada uma redução média de 46%, com variações entre 41% e 51%, dependendo da configuração adotada.

As diferenças nas porcentagens médias de redução entre os datasets podem ser explicadas pela quantidade e gravidade das issues presentes em cada caso. Embora tais diferenças existam, observou-se que a configuração impacta diretamente os resultados. Por exemplo, os parâmetros (temperatura, tipo de prompt e número de iterações) utiliza-

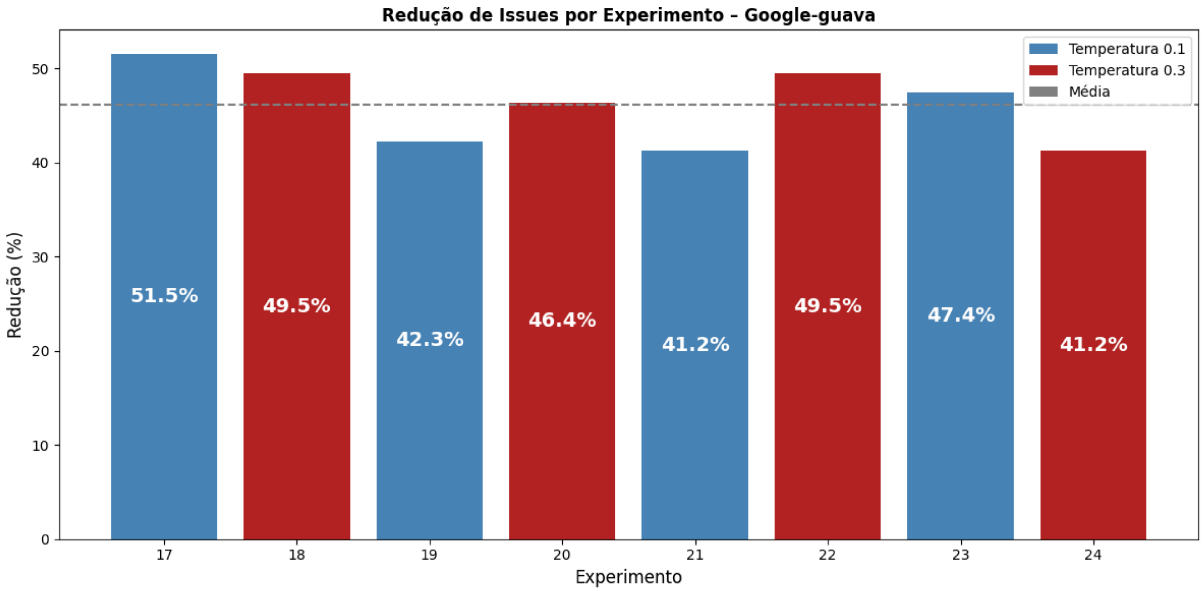

Figura 6 – Redução de issues observada no conjunto de dados Google Guava.

dos nos Experimentos 1 e 2 para o conjunto Commons Lang são os mesmos empregados nos Experimentos 9, 10, 17 e 19, resultando em tendências semelhantes entre as execuções.

Outro aspecto importante na análise da redução de issues é a gravidade dos problemas, uma vez que certos tipos de defeitos são, por natureza, mais complexos de corrigir. A complexidade ciclomática, por exemplo, exige a divisão de uma função em várias outras, mantendo o comportamento original. A Tabela 3 apresenta a redução média por nível de severidade, considerando todos os experimentos.

Tabela 3 – Redução de Issues por Severidade

Dataset	Redução Blocker (%)	Redução Critical (%)	Redução Major (%)
Commons Lang	-1.25	67.45	60.00
Commons IO		50.66	76.30
Google Guava		42.61	47.17

Esses resultados indicam que os modelos são mais eficazes na correção de issues de severidade intermediária e alta (CRITICAL e MAJOR), mas não conseguiram lidar de forma eficiente com issues do tipo BLOCKER, que inclusive apresentaram um leve aumento em um dos experimentos. Isso sugere que melhorias estruturais mais profundas ainda representam um desafio significativo para os LLMs.

Assim, embora os LLMs tenham se mostrado eficazes na resolução de uma parcela substancial dos problemas detectados por análise estática — frequentemente superando a marca de 50% de redução de issues — sua efetividade varia de acordo com a severidade do problema e com a configuração experimental, destacando a necessidade de parâmetros

cuidadosamente ajustados para alcançar resultados ideais.

4.2 RQ2 - A redução de issues impacta proporcionalmente a redução da dívida técnica?

Para avaliar a proporcionalidade entre a redução de issues e a redução da dívida técnica, foi calculada a porcentagem de redução de cada métrica nos experimentos realizados com três conjuntos de dados distintos: Commons Lang, Commons IO e Google Guava. A análise considerou a comparação entre os valores da primeira e da última iteração de cada execução experimental, utilizando uma única execução por configuração.

A Tabela 4 resume os valores médios de redução de issues, redução de dívida técnica, a diferença absoluta entre essas métricas e a proporção relativa da redução da dívida técnica em relação à redução de issues.

Tabela 4 – Comparação da Redução de Issues e da Redução da Dívida Técnica por Dataset

Dataset	Redução de Issues (%)	Redução de Dívida Técnica (%)	Diferença Absoluta (%)	Proporção Relativa (x)
Commons IO	69.03	63.33	5.70	0.91
Commons Lang	58.81	42.08	16.73	0.71
Google Guava	46.13	42.14	3.99	0.92

Os dados indicam que, embora a redução de issues esteja fortemente correlacionada com a redução da dívida técnica, essa relação não é proporcional em todos os casos. O conjunto de dados Commons IO apresentou a maior eficiência relativa, com uma redução média de 69,03% em issues e 63,33% em dívida técnica, resultando em uma diferença absoluta de apenas 5,70 pontos percentuais e uma proporção de 0,91, indicando que praticamente cada issue corrigida contribuiu para a redução da dívida técnica.

Em contraste, o conjunto Commons Lang apresentou uma diferença mais significativa: apesar de alcançar uma redução de 58,81% em issues, a redução na dívida técnica foi de apenas 42,08%, resultando em uma diferença de 16,73 pontos percentuais e uma proporção de 0,71. Isso sugere que, nesse contexto, os modelos de linguagem tendem a priorizar a correção de problemas com menor impacto técnico, deixando parcialmente ou totalmente sem tratamento issues com maior esforço estimado (em minutos).

O conjunto de dados Google Guava, por sua vez, apresentou as menores taxas absolutas de redução (46,13% para issues e 42,14% para dívida técnica), mas manteve uma alta proporção relativa (0,92), demonstrando um comportamento mais equilibrado entre as duas métricas, ainda que com impacto geral mais modesto.

Esses resultados destacam que, embora a redução de issues possa ser um bom indicativo de melhoria de código, ela não garante, por si só, uma redução proporcional da dívida técnica. A discrepância observada entre as métricas ressalta a importância de considerar

ambos os aspectos em avaliações automatizadas da qualidade de software assistidas por LLMs.

4.3 RQ3 - Como a configuração experimental impacta a melhoria do código?

Esta questão de pesquisa investiga em que medida as configurações experimentais — compostas pela combinação do modelo de LLM, temperatura de geração, tipo de prompt e número de iterações — influenciam a efetividade da melhoria automatizada de código. A análise foi conduzida com base em três execuções por experimento, a fim de obter estimativas mais estáveis do comportamento médio de cada configuração.

A Tabela 5 apresenta a média percentual de redução de issues para cada experimento, juntamente com os respectivos desvios padrão.

Tabela 5 – Resultados Médios da Redução Percentual de Issues por Experimento

Experimento	Modelo	Temperatura	Prompt	Iterações	Média de Redução (%)	Desvio Padrão (%)
1	GPT-4-mini	0.1	1	2	57.65	4.54
2	GPT-4-mini	0.3	2	2	49.49	5.52
3	GPT-4-mini	0.1	2	5	62.58	1.94
4	GPT-4-mini	0.3	1	5	70.41	7.52
5	Gemini	0.1	2	2	53.40	6.89
6	Gemini	0.3	1	2	61.90	5.14
7	Gemini	0.1	1	5	81.29	7.94
8	Gemini	0.3	2	5	70.26	5.91

Os resultados demonstram que as configurações experimentais influenciam substancialmente a efetividade da melhoria automatizada de código. Dentre todas as combinações avaliadas, o Experimento 7 — que utilizou o modelo Gemini com temperatura 0,1, prompt do tipo 1 e cinco iterações — obteve o melhor desempenho médio, alcançando uma redução de 81,29% nas issues. Esse resultado sugere que a combinação de um modelo robusto, uma temperatura mais conservadora, maior profundidade iterativa e um prompt bem formulado proporciona um ambiente favorável para melhorias eficazes.

No extremo oposto, o Experimento 2, configurado com o modelo GPT-4-mini, temperatura 0,3, prompt do tipo 2 e apenas duas iterações, resultou na menor redução média (49,49%). Essa configuração evidencia os riscos associados a combinações subótimas de parâmetros, em que uma temperatura mais elevada — embora potencialmente benéfica para a diversidade das respostas —, quando combinada com um prompt menos direcionado e menor número de ciclos, compromete a capacidade do modelo de convergir para soluções qualificadas.

Além disso, os resultados revelam consistência interna entre as configurações com múltiplas iterações, particularmente nos Experimentos 3, 4, 7 e 8, reforçando a noção de que a repetição controlada do ciclo de análise e melhoria tende a produzir melhores

resultados. No entanto, a variabilidade observada entre os modelos e a interação entre os parâmetros demonstram que o desempenho não depende de um único fator isoladamente, mas sim da sinergia entre todos os elementos experimentais.

4.3.1 Impacto do Modelo

A comparação entre os modelos — GPT-4-mini e Gemini — revela diferenças relevantes tanto no desempenho médio quanto na estabilidade entre as execuções. O modelo Gemini, especialmente nas configurações com cinco iterações (Experimentos 7 e 8), alcançou os melhores resultados gerais, com destaque para o Experimento 7, que obteve a maior redução média de issues (81,29%) entre todas as configurações testadas. Além disso, mesmo em configurações menos otimizadas (como o Experimento 6, com apenas duas iterações), o Gemini manteve desempenho competitivo (61,90%), indicando maior robustez e resistência a variações nos parâmetros experimentais.

Por outro lado, o GPT-4-mini demonstrou maior sensibilidade à temperatura, tipo de prompt e número de iterações. Seu desempenho médio variou significativamente entre os experimentos, indo de 49,49% (Experimento 2) a 70,41% (Experimento 4). Essa amplitude revela que o modelo pode apresentar tanto desempenho excelente quanto resultados limitados, dependendo do ajuste dos parâmetros experimentais. Em particular, a combinação de temperatura mais alta com prompt menos eficaz (como no Experimento 2) impactou negativamente o desempenho, mesmo em comparação com configurações mais simples do modelo Gemini.

Esses achados indicam que, embora ambos os modelos sejam capazes de melhorar código com certo grau de sucesso, o modelo Gemini tende a ser mais consistente e confiável, especialmente em cenários com maior profundidade iterativa. Em contrapartida, o GPT-4-mini apresenta maior potencial de variabilidade, exigindo maior cuidado no desenho experimental para alcançar resultados satisfatórios.

4.3.2 Temperatura e Prompt: Uma Interação Determinante

A análise dos experimentos mostra que a temperatura de geração e o tipo de prompt não devem ser avaliados de forma independente, mas como componentes interdependentes que, quando bem combinados, podem maximizar — ou comprometer — a qualidade da melhoria automática.

De modo geral, temperaturas mais altas, como 0,3, aumentam a diversidade e a fluência das respostas do modelo, o que pode ser benéfico para tarefas que requerem criatividade, como reescrita de código. Entretanto, essa maior liberdade demanda prompts mais claros, objetivos e bem estruturados, capazes de guiar o modelo apesar do espaço de geração mais amplo. Isso fica evidenciado ao comparar os Experimentos 2 e 4, ambos utilizando GPT-4-mini com temperatura 0,3, mas com prompts diferentes: o Experimento 4 (Prompt

1) alcançou uma redução de 70,41% nas issues, enquanto o Experimento 2 (Prompt 2) atingiu apenas 49,49%. A diferença de mais de 20 pontos percentuais, apesar do uso do mesmo modelo e temperatura, destaca que o prompt foi determinante para orientar o processo de melhoria.

Esse comportamento também é observado com o modelo Gemini, embora com menor variabilidade. A comparação entre os Experimentos 7 e 8, ambos com cinco iterações, mas prompts diferentes, mostra que o Prompt 1, quando combinado com temperatura 0,1 (Experimento 7), levou ao melhor desempenho geral (81,29%), enquanto o Prompt 2 (Experimento 8) resultou em 70,26%. Apesar de ambos apresentarem altas taxas de redução, a diferença reforça o papel moderador do prompt sobre os efeitos da temperatura.

Assim, a evidência empírica demonstra que a temperatura aumenta a flexibilidade da geração, mas só produz ganhos reais quando associada a prompts que contextualizam e delimitam adequadamente o escopo da tarefa. Modelos expostos a prompts vagos ou genéricos, mesmo com maior liberdade criativa, tendem a produzir resultados inconsistentes. Portanto, o efeito da temperatura não é linear, e sua efetividade está intrinsecamente ligada à qualidade da engenharia de prompt empregada.

4.3.3 Número de Iterações: Impacto Incremental e Condicionado

A análise dos dados revela que o número de iterações exerce um efeito incremental positivo na qualidade da melhoria, mas esse impacto está condicionado à presença de outras boas decisões experimentais. Configurações com cinco iterações, como nos Experimentos 3 e 4, alcançaram reduções médias maiores (62,58% e 70,41%, respectivamente) em comparação com configurações com apenas duas iterações, como nos Experimentos 1, 2 e 5.

Contudo, aumentar apenas o número de iterações não é suficiente para garantir melhores resultados. Por exemplo, o Experimento 2, com apenas duas iterações, apresentou o pior desempenho (49,49% de redução média) — resultado atribuído à combinação da baixa iteratividade com um prompt ineficaz. Em contrapartida, o Experimento 4, que combinou mais iterações com um prompt mais claro e uma temperatura que favoreceu a diversidade, alcançou um dos melhores resultados.

Esses achados sugerem que iterações adicionais possibilitam um processo progressivo de refinamento, no qual o modelo pode abordar problemas residuais deixados pelos ciclos anteriores. Entretanto, quando a base experimental está mal configurada — por exemplo, com instruções vagas ou parâmetros inadequados de geração — a repetição pode simplesmente propagar falhas ou produzir apenas ajustes superficiais.

Assim, o número de iterações deve ser interpretado como um fator amplificador da qualidade, cujo impacto é mais expressivo quando inserido em um contexto experimental

favorável. Trata-se, portanto, de uma variável estratégica que pode ser explorada para intensificar o processo iterativo de melhoria, desde que os demais elementos do experimento estejam devidamente calibrados.

De forma geral, a análise demonstra que a configuração experimental impacta direta e substancialmente a efetividade da melhoria automatizada conduzida por LLMs. O desempenho do modelo não pode ser dissociado dos parâmetros acompanhantes. O modelo Gemini mostrou-se mais robusto e consistente, enquanto o GPT-4-mini apresentou maior variabilidade, sendo mais sensível à qualidade do prompt, temperatura e número de iterações.

Além disso, a interação entre temperatura e prompt mostrou-se determinante, com evidências claras de que prompts bem estruturados são essenciais para extrair o máximo potencial de modelos que operam com maior liberdade criativa. Por sua vez, o número de iterações atua como um amplificador, contribuindo para a maturação das melhorias ao longo de múltiplos ciclos.

Esses resultados reforçam que a efetividade da melhoria baseada em LLM depende da orquestração de múltiplos fatores, exigindo planejamento sistemático e desenho experimental para selecionar a configuração ideal conforme os objetivos e o contexto do projeto.

4.4 RQ4 - O processo iterativo mantém a funcionalidade original do código?

Esta questão de pesquisa investigou se o processo iterativo de melhoria assistido por LLMs compromete a funcionalidade original do código-fonte em algum momento. Como o objetivo do estudo é aplicar melhorias baseadas nas recomendações do SonarQube, é crucial garantir que tais modificações não introduzam regressões ou removam comportamentos esperados.

Para responder a essa questão, foi realizada uma avaliação manual e exploratória das versões do código geradas ao longo das iterações. A análise consistiu na inspeção visual das alterações feitas pelos modelos. O objetivo principal foi identificar indícios de falhas funcionais, como remoção inadequada de trechos de código, mudanças sem justificativa semântica ou modificações inesperadas na lógica de controle. Os resultados indicam que, nas amostras avaliadas, não foram observadas quebras funcionais explícitas. De modo geral, os LLMs mantiveram um comportamento conservador, realizando melhorias pontuais, como reorganização de código, renomeações e extração de métodos auxiliares, sem impactar a lógica funcional das classes.

Um exemplo representativo pode ser observado nas Figuras 7 e 8. A Figura 7 apresenta a versão original da função `lastIndexOf(...)`, enquanto a Figura 8 mostra a versão aprimorada, na qual um método auxiliar denominado `handleStringLastIndexOf(...)`

foi extraído. Essa modificação preserva integralmente o comportamento funcional original, ao mesmo tempo que melhora a legibilidade, promove a modularização e facilita a manutenção futura do código.

```
1  static int lastIndexOf(final CharSequence cs, final CharSequence
       searchChar, int start) {
2      if (searchChar == null || cs == null) {
3          return NOT_FOUND;
4      }
5      if (searchChar instanceof String) {
6          if (cs instanceof String) {
7              return ((String) cs).lastIndexOf((String) searchChar, start)
                   ;
8          }
9          if (cs instanceof StringBuilder) {
10             return ((StringBuilder) cs).lastIndexOf((String) searchChar,
                    start);
11         }
12         if (cs instanceof StringBuffer) {
13             return ((StringBuffer) cs).lastIndexOf((String) searchChar,
                   start);
14         }
15     }
16     ...
17 }
```

Figura 7 – Código antes da melhoria.

```
1  static int lastIndexOf(final CharSequence cs, final CharSequence
       searchChar, int start) {
2      if (searchChar == null || cs == null) {
3          return NOT_FOUND;
4      }
5      if (searchChar instanceof String) {
6          return handleStringLastIndexOf(cs, (String) searchChar, start);
7      }
8      ...
9  }
10
11 private static int handleStringLastIndexOf(final CharSequence cs, final
       String searchChar, int start) {
12     if (cs instanceof String) {
13         return ((String) cs).lastIndexOf(searchChar, start);
14     }
15     if (cs instanceof StringBuilder) {
16         return ((StringBuilder) cs).lastIndexOf(searchChar, start);
17     }
18     if (cs instanceof StringBuffer) {
19         return ((StringBuffer) cs).lastIndexOf(searchChar, start);
20     }
21     return NOT_FOUND;
22 }
```

Figura 8 – Código após melhoria por extração de método.

Embora os resultados sejam positivos, é importante destacar que a avaliação funcional realizada possui limitações. A ausência de uma suíte abrangente de testes automatizados ou validação semântica formal impede uma verificação exaustiva da preservação comportamental em larga escala. Assim, a análise realizada é indicativa, mas não conclusiva quanto à total ausência de regressões. Trabalhos futuros podem incorporar estratégias mais robustas, como testes baseados em especificações, geração automática de oráculos ou análise de mutação, para fortalecer a confiabilidade da validação funcional no contexto de melhorias assistidas por LLMs.

Portanto, pode-se concluir que, nas amostras avaliadas, não foram observadas quebras funcionais perceptíveis, e os modelos mostraram-se capazes de realizar melhorias seguras do ponto de vista comportamental. No entanto, a ausência de testes automatizados ou validação semântica formal representa uma limitação relevante do estudo. Trabalhos futuros podem incorporar mecanismos como verificação de equivalência semântica, testes baseados em especificações ou testes de mutação para validar de forma mais rigorosa a preservação funcional ao longo do processo iterativo de melhoria.

4.5 Considerações Finais

Os resultados obtidos evidenciam que a abordagem proposta, baseada em ciclos iterativos com LLMs guiados por recomendações do SonarQube, é eficaz na redução de *issues* em projetos de código aberto. Em diferentes datasets e configurações, observou-se que a maioria dos experimentos alcançou reduções superiores a 50%, com alguns cenários atingindo mais de 80% de melhoria. Esses resultados reforçam o potencial dos LLMs como agentes de apoio à manutenção automatizada de software, especialmente quando inseridos em pipelines estruturadas.

No entanto, a análise também revela que a efetividade da solução é sensível à configuração experimental adotada, e que a consistência dos resultados pode variar em função do modelo, temperatura, tipo de prompt e número de iterações. Além disso, a dificuldade dos modelos em lidar com *issues* mais severas, como as do tipo *BLOCKER*, indica a necessidade de abordagens complementares para garantir a correção de problemas estruturais mais complexos. Assim, embora os achados desta pesquisa sejam promissores, eles também apontam para oportunidades de refinamento e expansão da abordagem em estudos futuros.

4.6 Ameaças à Validade

Embora os resultados obtidos nesta pesquisa demonstrem o potencial dos Modelos de Linguagem de Grande Escala (LLMs) para a melhoria automatizada de código-fonte, algumas limitações e ameaças à validade precisam ser reconhecidas. Este estudo foi con-

duzido com base em um conjunto específico de repositórios Java de código aberto (Apache Commons Lang, Apache Commons IO e Google Guava), o que pode limitar a generalização dos resultados para outros domínios de aplicação, linguagens de programação ou estilos de codificação.

Além disso, apenas dois modelos de linguagem (GPT-4-mini e Gemini) foram avaliados sob um conjunto restrito de configurações. Apesar da realização de múltiplas execuções visando reduzir a variabilidade, o espaço de parâmetros explorado permanece limitado frente à diversidade de comportamentos possíveis dos LLMs e às diferentes estratégias de engenharia de prompt disponíveis. Outra limitação importante refere-se à avaliação funcional do código aprimorado. Embora o processo iterativo com LLMs tenha reduzido significativamente o número de *issues* relatadas pelo SonarQube, a preservação da funcionalidade original foi verificada manualmente, por meio de inspeção e execução direta dos métodos modificados. A ausência de um mecanismo sistemático e automatizado de validação semântica impede afirmar com total segurança que todas as melhorias mantêm o comportamento esperado do sistema.

Ademais, embora não tenham sido observadas falhas funcionais nas amostras analisadas, a inexistência de testes exaustivos amplia o risco de regressões não identificadas. Também se destaca como ameaça à validade a dependência exclusiva do SonarQube como ferramenta de avaliação das melhorias aplicadas. Apesar de oferecer resultados detalhados por meio de análise estática, o SonarQube pode não abranger todos os aspectos relevantes da qualidade do software, como problemas de projeto arquitetural, aprimoramentos na legibilidade do código não mapeados por regras, ou fatores de manutenibilidade que extrapolam as *issues* detectadas.

Por fim, os LLMs utilizados operam como sistemas de caixa-preta, sem fornecer explicitação do raciocínio nem considerar o contexto arquitetural mais amplo do software. Como consequência, as melhorias sugeridas tendem a ser localizadas, podendo desconsiderar dependências e interações entre componentes. Tal limitação pode comprometer a mantenibilidade a longo prazo e a qualidade da integração, ainda que as métricas estáticas de curto prazo apresentem ganhos expressivos.

Capítulo 5

Trabalhos Relacionados

O uso de Modelos de Linguagem de Grande Escala (LLMs) — como o GPT-3 (OpenAI, 2024a), GPT-4 (OpenAI, 2024b), CodeT5 (WANG et al., 2021) e Codex (OpenAI, 2021) — tem mostrado grande potencial em tarefas como melhoria de código, reparo automatizado e geração de código (ZHANG et al., 2024; HOU et al., 2024). Zhang *et al.* (ZHANG et al., 2024) realizaram uma revisão sistemática sobre o uso de LLMs na Reparação Automatizada de Programas (APR), organizando dezenas de estudos recentes, propondo uma taxonomia das abordagens existentes e explorando cenários como correção de falhas semânticas e mitigação de vulnerabilidades de segurança.

Complementarmente, Hou *et al.*(HOU et al., 2024) discutiram o uso de LLMs com foco especial em reparação de vulnerabilidades, destacando a importância do pré-treinamento dos modelos em conjuntos de dados específicos de código. Modelos como o CodeBERT(FENG et al., 2020) e o CodeT5 (WANG et al., 2021) têm sido aplicados em tarefas supervisionadas de reparo, treinados com padrões explícitos de defeito e solução, ressaltando a necessidade de conjuntos de dados representativos para maximizar a eficácia dos modelos.

Buscando reduzir a dependência de treinamento supervisionado, diversos estudos (MADAAN et al., 2023; CHEN et al., 2023; JIANG; WANG; WANG, 2023) demonstraram que LLMs podem reparar defeitos de código autonomamente por inferência, sem a necessidade de rótulos manuais. No entanto, embora esses modelos consigam refletir sobre o comportamento do código e propor correções, não há garantias formais de correção. Conforme destacado por Cai *et al.*(CAI et al., 2025), essa limitação tem motivado o desenvolvimento de mecanismos complementares para guiar e validar as correções geradas por LLMs, mitigando o risco de degradação funcional ou introdução de novos defeitos(AGRAWAL et al., 2023).

Enfrentando as limitações contextuais frequentemente encontradas por LLMs — especialmente quando o código depende de informações dispersas em múltiplos arquivos ou bibliotecas externas — trabalhos recentes têm explorado mecanismos baseados em análise estática e técnicas avançadas de engenharia de prompts (MOHAJER et al., 2024; CHEN

et al., 2023).

Entre esses, destacam-se os trabalhos de Agrawal *et al.*(AGRAWAL et al., 2023), Ahmed*et al.*(AHMED et al., 2023) e Hao*et al.*(HAO et al., 2023). Agrawal*et al.* propuseram a técnica de *Monitor-Guided Decoding* (MGD), na qual um monitor atua como uma interface com estado entre o LLM e ferramentas de análise estática, como linters e servidores LSP. Durante a geração de código, o monitor intercepta trechos de código em pontos de disparo pré-definidos, consulta essas ferramentas e transforma seu retorno em restrições aplicadas aos passos subsequentes de decodificação.

Ahmed *et al.* (AHMED et al., 2023) aproveitaram saídas de ferramentas como o SonarQube — incluindo regras violadas, severidade e localização dos defeitos — para selecionar exemplos históricos relevantes e compor prompts ricos em contexto e no estilo few-shot, melhorando a capacidade de generalização e a precisão de reparo em projetos não vistos anteriormente.

Em uma linha de pesquisa distinta, Hao *et al.* (HAO et al., 2023) exploraram o potencial de guiar LLMs por meio da execução simbólica de pseudocódigo embutido diretamente nos prompts. Sua abordagem permite que o modelo raciocine sobre variáveis, fluxo de controle e chamadas de métodos, simulando aspectos da análise estática internamente, sem a necessidade de instrumentação externa.

Para além de estratégias pontuais de análise e reparo, estudos recentes têm explorado ciclos de melhoria iterativa, nos quais sugestões geradas por LLMs são aplicadas, avaliadas e refinadas ao longo de múltiplas rodadas. Barke *et al.*(BARKE; JAMES; POLIKARPOVA, 2023) investigaram essa abordagem interativa com LLMs integrados ao GitHub Copilot(GitHub, 2024), enquanto outras propostas projetaram pipelines totalmente automatizadas que combinam análise estática, geração de sugestões e reavaliação contínua (MADAAN et al., 2023; JIANG; WANG; WANG, 2023).

Essas abordagens, em conjunto, evidenciam a crescente maturidade do uso de LLMs na engenharia de software, reforçando a importância da integração com análise estática, curadoria contextual de prompts e iteração contínua para superar limitações inerentes e aumentar a robustez dos modelos em cenários reais de desenvolvimento.

Capítulo 6

Conclusão

Este trabalho investigou o uso de Modelos de Linguagem de Grande Escala (LLMs) para a melhoria automatizada de código-fonte por meio de um processo iterativo guiado pelas recomendações do SonarQube. A proposta central foi operacionalizada por meio do desenvolvimento de uma pipeline experimental automatizada que integra análise estática, geração de prompts, interação com LLMs e reavaliação contínua da qualidade após cada iteração.

A pipeline desenvolvida teve um papel fundamental na condução da pesquisa, permitindo a execução reprodutível, escalável e comparável de múltiplos experimentos. Ao automatizar tarefas desde a leitura dos arquivos de entrada até a coleta dos resultados após as melhorias, a pipeline possibilitou a aplicação consistente do processo iterativo em diferentes repositórios, modelos e configurações, além de facilitar a análise quantitativa dos resultados.

Os dados coletados demonstraram que os modelos foram capazes de promover reduções significativas nas issues identificadas, com reduções médias superiores a 58% ao longo dos experimentos. A análise segmentada por severidade revelou que os modelos tendem a ser mais eficazes na resolução de issues do tipo MAJOR e CRITICAL, que apresentaram as maiores taxas de correção. Esse achado indica que os LLMs são capazes de contribuir com melhorias tecnicamente mais relevantes, contrariando a suposição de que seriam eficazes apenas em correções simples.

A redução da dívida técnica, medida em minutos de esforço estimado, acompanhou parcialmente a redução das issues, embora com variações entre diferentes configurações. Correlações mais fortes foram observadas quando as correções se concentraram em issues de maior severidade. Essa observação reforça a importância de considerar múltiplas métricas de qualidade — tanto quantitativas quanto qualitativas — ao avaliar processos de melhoria automatizada.

A verificação da preservação da funcionalidade no código melhorado foi realizada manualmente por meio da inspeção e comparação direta dos métodos alterados. Embora não tenham sido identificadas falhas de nas amostras avaliadas, essa estratégia não garante

sistematicamente a equivalência semântica com o comportamento original, representando uma limitação importante a ser abordada com abordagens de validação mais robustas em trabalhos futuros.

Quanto às configurações experimentais, os resultados destacam a importância da sinergia entre os parâmetros. O modelo Gemini, combinado com cinco iterações, um prompt bem estruturado e uma configuração conservadora de temperatura (experimento 7), apresentou o melhor desempenho geral. O GPT-4-mini, por sua vez, mostrou maior sensibilidade à configuração, com resultados mais dependentes do ajuste fino da temperatura, elaboração do prompt e número de iterações.

Em resumo, este estudo demonstrou que LLMs podem ser utilizados com sucesso em pipelines automatizadas para a melhoria de código-fonte, alcançando ganhos concretos na redução de problemas identificados por ferramentas de análise estática. A pipeline desenvolvida foi essencial para estruturar o processo de forma sistemática e permitir experimentação iterativa em múltiplos cenários. Apesar das limitações relacionadas à validação funcional, os resultados obtidos reforçam o potencial das soluções baseadas em LLM como ferramentas de apoio ao desenvolvimento e à manutenção de software, desde que acompanhadas de boas práticas de engenharia e mecanismos adequados de validação.

A solução proposta não é apenas eficaz do ponto de vista experimental, mas também prática: pode ser integrada a pipelines de desenvolvimento reais para apoiar refatorações automatizadas baseadas em feedback de análise estática, reduzindo a carga de trabalho dos desenvolvedores e aumentando a consistência nas melhorias de qualidade do código.

6.1 Principais Contribuições

Este trabalho apresenta um conjunto de contribuições para a área de engenharia de software assistida por IA, com ênfase em melhoria automatizada de código. As principais contribuições incluem:

- A proposta e implementação de uma pipeline iterativa, automatizada e reprodutível, que integra ferramentas de análise estática (SonarQube) com modelos de linguagem de grande escala (LLMs), aplicada à melhoria de código-fonte Java;
- O uso de mensagens de análise estática como insumo direto para a construção de prompts em linguagem natural, permitindo direcionamento mais preciso e contextualizado das instruções aos modelos;
- A execução de um plano experimental baseado em Fractional Factorial Design (FFD), que permitiu analisar o impacto de diferentes configurações (modelo, temperatura, estilo de prompt e número de iterações) nos resultados obtidos, com redução do custo experimental e preservação da análise dos principais efeitos;

- ❏ A aplicação da abordagem proposta em três projetos reais amplamente utilizados (Commons Lang, Commons IO e Guava), demonstrando sua viabilidade prática e impacto na qualidade interna do código;

- ❏ A obtenção de resultados que demonstram redução de mais de 50% dos problemas identificados pelo SonarQube, com preservação da funcionalidade observável em todos os casos analisados;

- ❏ A disponibilização pública dos artefatos experimentais, incluindo código-fonte, dados, scripts de execução e resultados consolidados, fomentando a reprodutibilidade e o avanço de pesquisas futuras na área.

6.2 Trabalhos Futuros

Para trabalhos futuros, são propostas as seguintes direções:

- ❏ Incorporar métricas mais rigorosas de validação funcional e semântica, incluindo testes diferenciais, testes baseados em especificação e análises de cobertura aprimoradas, com o objetivo de garantir a preservação do comportamento do código após a aplicação das melhorias;

- ❏ Expandir os experimentos para diferentes contextos de software, incluindo outras linguagens de programação, frameworks complexos e bases de código com cobertura de testes insuficiente;

- ❏ Aplicar técnicas de engenharia de prompts adaptativa, permitindo a evolução iterativa das instruções fornecidas ao modelo ao longo do processo de melhoria;

- ❏ Integrar a abordagem com sistemas de controle de versão para avaliar o impacto das melhorias dentro de ciclos reais de desenvolvimento (por exemplo, pull requests, merges, revisões humanas);

- ❏ Avaliar a viabilidade de inclusão da pipeline proposta em um ambiente real de integração e entrega contínua (CI/CD);

- ❏ Por fim, investigar estratégias híbridas que combinem heurísticas estáticas, sugestões humanas e respostas de LLMs para aumentar a confiabilidade e a explicabilidade em processos de melhoria assistida.

Essas direções futuras visam não apenas aumentar a confiabilidade dos LLMs em tarefas de melhoria de código, mas também ampliar sua aplicabilidade prática em cenários de desenvolvimento contínuo e manutenção de sistemas no mundo real.

6.3 Produção Bibliográfica

A pesquisa desenvolvida resultou no desenvolvimento do artigo intitulado *An Empirical Study on the Effectiveness of Iterative LLM-Based Improvements for Static Analysis Issues*, aceito para publicação na Trilha de Pesquisa do XXXIX Simpósio Brasileiro de Engenharia de Software (SBES 2025) (GONCALVES; MAIA, 2025).

Referências

AGRAWAL, L. A. et al. Monitor-guided decoding of code LMs with static analysis of repository context. **Advances in Neural Information Processing Systems**, v. 36, p. 32270–32298, 2023.

AHMED, T. et al. Improving few-shot prompts with relevant static analysis products. **arXiv preprint arXiv:2304.06815**, Apr, 2023.

AKAIKINE, A. **The impact of software design structure on product maintenance costs and measurement of economic benefits of product redesign**. Tese (Doutorado) — Massachusetts Institute of Technology, 2010.

ALFAYEZ, R. et al. How SonarQube-identified technical debt is prioritized: An exploratory case study. **Information and Software Technology**, Elsevier, v. 156, p. 107147, 2023.

BARKE, S.; JAMES, M. B.; POLIKARPOVA, N. Grounded Copilot: How programmers interact with code-generating models. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 7, n. OOPSLA1, p. 85–111, 2023.

BESKER, T. et al. The influence of technical debt on software developer morale. **Journal of Systems and Software**, Elsevier, v. 167, p. 110586, 2020.

BROWN, T. et al. Language models are few-shot learners. **Advances in neural information processing systems**, v. 33, p. 1877–1901, 2020.

BROWN, W. H. et al. **AntiPatterns: refactoring software, architectures, and projects in crisis**. [S.l.]: John Wiley & Sons, Inc., 1998.

CAI, Y. et al. Automated program refinement: Guide and verify code large language model with refinement calculus. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 9, n. POPL, p. 2057–2089, 2025.

CAMPBELL, G. A.; PAPAPETROU, P. P. **SonarQube in action**. [S.l.]: Manning Publications Co., 2013.

CHEN, J. et al. Perspectives on refactoring planning and practice: an empirical study. **Empirical Software Engineering**, Springer, v. 21, p. 1397–1436, 2016.

CHEN, M. et al. Evaluating large language models trained on code. **arXiv preprint arXiv:2107.03374**, 2021.

CHEN, X. et al. Teaching large language models to self-debug. **arXiv preprint arXiv:2304.05128**, 2023.

CodeGen. **CodeGen AI Platform**. 2024. <https://www.codegen.com/>. Accessed: 2024-04-18.

DAS, S.; SRIHARI, R. K. Compos mentis at semeval2024 task6: A multi-faceted role-based large language model ensemble to detect hallucination. In: **Proceedings of the 18th International Workshop on Semantic Evaluation (SemEval-2024)**. [S.l.: s.n.], 2024. p. 1449–1454.

DEHAGHANI, S. M. H.; HAJRAHIMI, N. Which factors affect software projects maintenance cost more? **Acta Informatica Medica**, v. 21, n. 1, p. 63, 2013.

ETEMADI, K. et al. Sorald: Automatic patch suggestions for SonarQube static analysis violations. **IEEE Transactions on Dependable and Secure Computing**, IEEE, v. 20, n. 4, p. 2794–2810, 2022.

FENG, Z. et al. Codebert: A pre-trained model for programming and natural languages. **Findings of the Association for Computational Linguistics: EMNLP 2020**, Association for Computational Linguistics, 2020.

FERNÁNDEZ-SÁNCHEZ, C.; GARBAJOSA, J.; YAGÜE, A. A framework to aid in decision making for technical debt management. In: IEEE. **2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)**. [S.l.], 2015. p. 69–76.

FOSTER, J. S.; HICKS, M. W.; PUGH, W. Improving software quality with static analysis. In: **Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering**. [S.l.: s.n.], 2007. p. 83–84.

FOWLER, M. **Refactoring: improving the design of existing code**. [S.l.]: Addison-Wesley Professional, 2018.

GitHub. **GitHub Copilot**. 2024. <https://github.com/features/copilot>. Accessed: 2024-04-18.

GONCALVES, J. C.; MAIA, M. A. An empirical study on the effectiveness of iterative LLM-based improvements for static analysis issues. In: SBC. **Proc. of the 39th Brazilian Symposium on Software Engineering, SBES'2025**. [S.l.], 2025. p. accepted for publication.

Google. **Google Gemini: Our largest and most capable AI model**. 2023. <https://blog.google/technology/ai/google-gemini-ai/>. Accessed: 2025-06-25.

Google DeepMind. **Google Gemini: December 2024 Update – Building Responsibly**. 2024. <https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/#building-responsibly>. Accessed: 2025-06-25.

HAO, Y. et al. E&v: Prompting large language models to perform static analysis by pseudo-code execution and verification. **arXiv preprint arXiv:2312.08477**, 2023.

HOU, X. et al. Large language models for software engineering: A systematic literature review. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, v. 33, n. 8, p. 1–79, 2024.

Hugging Face. **StarCoder and StarCoderBase: The next generation of code LLMs**. 2023. <https://huggingface.co/blog/starcoder>. Accessed: 2024-04-18.

JIANG, S.; WANG, Y.; WANG, Y. Selfevolve: A code evolution framework via large language models. **arXiv preprint arXiv:2306.02907**, 2023.

KHALEEL, S. I.; MAHMOOD, R. A. Automatic software refactoring to enhance quality: A review. **Jurnal Ilmiah Teknik Elektro Komputer dan Informatika (JITEKI)**, v. 10, n. 4, p. 734–746, 2024.

LIU, Y. et al. Refining ChatGPT-generated code: Characterizing and mitigating code quality issues. **ACM Transactions on Software Engineering and Methodology**, ACM New York, NY, v. 33, n. 5, p. 1–26, 2024.

LOMIO, F.; MORESCHINI, S.; LENARDUZZI, V. A machine and deep learning analysis among SonarQube rules, product, and process metrics for fault prediction. **Empirical Software Engineering**, Springer, v. 27, n. 7, p. 189, 2022.

LU, J. et al. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In: IEEE. **2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)**. [S.l.], 2023. p. 647–658.

MADAAN, A. et al. Self-refine: Iterative refinement with self-feedback. **Advances in Neural Information Processing Systems**, v. 36, p. 46534–46594, 2023.

MARCILIO, D. et al. Are static analysis violations really fixed? a closer look at realistic usage of SonarQube. In: IEEE. **2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)**. [S.l.], 2019. p. 209–219.

MENS, T.; TOURWÉ, T. A survey of software refactoring. **IEEE Transactions on software engineering**, IEEE, v. 30, n. 2, p. 126–139, 2004.

Meta. **Code Llama: An Open Reproduction of Code-Related Large Language Models**. 2023. <https://ai.meta.com/blog/code-llama-large-language-model-coding/>. Accessed: 2024-04-18.

MOHAJER, M. M. et al. Effectiveness of ChatGPT for static analysis: How far are we? In: **Proceedings of the 1st ACM International Conference on AI-Powered Software**. [S.l.: s.n.], 2024. p. 151–160.

MONTGOMERY, D. C. **Design and analysis of experiments**. [S.l.]: John wiley & sons, 2017.

MOTOGNA, S.; BERCIU, L.-M.; MOLDOVAN, V.-A. Artificial intelligence methods in software refactoring: A systematic literature review. In: IEEE. **2024 50th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)**. [S.l.], 2024. p. 309–316.

OpenAI. **OpenAI Codex**. 2021. <https://openai.com/index/openai-codex/>. Accessed: 2024-04-18.

OPENAI. **ChatGPT**. 2023. Acessado em: 12 ago. 2023. Disponível em: <https://openai.com/chatgpt>.

_____. **GPT-4 Technical Report**. 2023. OpenAI Research. <https://openai.com/research/gpt-4>.

OpenAI. **GPT-3 Apps: Applications Powered by OpenAI's GPT-3**. 2024. <https://openai.com/index/gpt-3-apps/>. Accessed: 2024-04-18.

_____. **GPT-4 by OpenAI**. 2024. <https://openai.com/index/gpt-4/>. Accessed: 2024-04-18.

OUYANG, L. et al. Training language models to follow instructions with human feedback. **Advances in neural information processing systems**, v. 35, p. 27730–27744, 2022.

PELLEGRINI, L.; JANES, A. A.; TAIBI, D. On the fault proneness of SonarQube technical debt violations. an empirical study. **Ph. D. dissertation**, 2018.

PRESSMAN, R. S.; MAXIM, B. R. **Engenharia de software-9**. [S.l.]: McGraw Hill Brasil, 2021.

RADFORD, A. et al. Improving language understanding by generative pre-training. San Francisco, CA, USA, 2018.

_____. Language models are unsupervised multitask learners. **OpenAI blog**, v. 1, n. 8, p. 9, 2019.

ROSS, S. I. et al. The programmer's assistant: Conversational interaction with a large language model for software development. In: **Proceedings of the 28th International Conference on Intelligent User Interfaces**. [S.l.: s.n.], 2023. p. 491–514.

SANTOS, C. G. d. et al. Um estudo empírico sobre a gerência de dívida técnica em projetos de desenvolvimento de software que utilizam scrum. Pontifícia Universidade Católica do Rio Grande do Sul, 2016.

SHA, Z.; ZHANG, Y. Prompt stealing attacks against large language models. **arXiv preprint arXiv:2402.12959**, 2024.

SIMÕES, I. R. d. S.; VENSON, E. Evaluating source code quality with large language models: a comparative study. In: **Proceedings of the XXIII Brazilian Symposium on Software Quality**. [S.l.: s.n.], 2024. p. 103–113.

SonarSource. **SonarSource - Continuous Code Quality**. 2024. Accessed: 2024-04-18. Disponível em: <https://www.sonarsource.com/>.

TELANG, R.; WATTAL, S. An empirical analysis of the impact of software vulnerability announcements on firm stock price. **IEEE Transactions on Software engineering**, IEEE, v. 33, n. 8, p. 544–557, 2007.

VASWANI, A. et al. Attention is all you need. **Advances in neural information processing systems**, v. 30, 2017.

WANG, C. et al. Generating variable explanations via zero-shot prompt learning. In: IEEE. **2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)**. [S.l.], 2023. p. 748–760.

WANG, Y. et al. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: **Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing**. [S.l.: s.n.], 2021. p. 8696–8708.

WU, T. et al. A brief overview of ChatGPT: The history, status quo and potential future development. **IEEE/CAA Journal of Automatica Sinica**, IEEE, v. 10, n. 5, p. 1122–1136, 2023.

YEBOAH, J.; POPOOLA, S. Efficacy of static analysis tools for software defect detection on open-source projects. In: IEEE. **2023 International Conference on Computational Science and Computational Intelligence (CSCI)**. [S.l.], 2023. p. 1588–1593.

ZHANG, Q. et al. A systematic literature review on large language models for automated program repair. **CoRR**, 2024.

ZHONG, M. et al. Codegen-test: An automatic code generation model integrating program test information. In: IEEE. **2023 2nd International Conference on Cloud Computing, Big Data Application and Software Engineering (CBASE)**. [S.l.], 2023. p. 341–344.

ZUBAIR, F.; AL-HITMI, M.; CATAL, C. The use of large language models for program repair. **Computer Standards & Interfaces**, Elsevier, p. 103951, 2024.

APÊNDICE **A**

Disponibilidade de Artefatos

Todos os artefatos resultantes desta pesquisa — incluindo o código-fonte da *pipeline*, scripts de execução, dados de entrada e resultados obtidos — estão disponíveis publicamente no repositório Zenodo[1] para fins de replicação e reutilização.

[1] <https://doi.org/10.5281/zenodo.15278368>