Universidade Federal de Pernambuco
Centro de Informática

Caio Bezerra Londres Barreto

SonarAutoFix: Automatização da Correção de Código com Base em Análise Estática de Código

Recife
2025

Caio Bezerra Londres Barreto

SonarAutoFix: Automatização da Correção de Código com Base em Análise Estática de Código

Trabalho apresentado ao Programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Paulo Freitas de Araújo Filho

Recife

2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Barreto, Caio Bezerra Londres.
SonarAutoFix: Automatização da Correção de Código com Base em Análise Estática de Código / Caio Bezerra Londres Barreto. - Recife, 2025.
46 p. : il., tab.

Orientador(a): Paulo Freitas de Araújo Filho
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2025.
Inclui referências, apêndices.

1. LLM. 2. Análise estática. 3. Correção automática de issues. 4. Segurança de código. I. Araújo Filho, Paulo Freitas de. (Orientação). II. Título.

000 CDD (22.ed.)

Caio Bezerra Londres Barreto

SonarAutoFix: Automatização da Correção de Código com Base em Análise Estática de Código

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Ciênca da Computação.

Aprovado em: 02/04/2025

BANCA EXAMINADORA

_____

Prof. Paulo Freitas de Araújo Filho (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Filipe Carlos de Albuquerque Calegário (Examinador Interno)

Universidade Federal de Pernambuco

RESUMO

A qualidade do código-fonte é um fator crítico no desenvolvimento de software, impactando diretamente a segurança, a manutenibilidade e a eficiência dos sistemas. No entanto, a identificação e correção de problemas em código costumam ser tarefas demoradas e que exigem conhecimento especializado. Ferramentas de análise estática, como o SonarQube, são amplamente utilizadas para detectar problemas automaticamente, mas a correção ainda depende da intervenção manual dos desenvolvedores.

Diante desse cenário, este trabalho propõe a automação da correção de *issues* detectadas por ferramentas de análise estática utilizando modelos de linguagem (*LLMs – Large Language Models*). A solução desenvolvida se integra ao fluxo de desenvolvimento através de um script de *pre-commit*, permitindo que, ao identificar problemas no código, um modelo de inteligência artificial gere sugestões de correção automaticamente. O desenvolvedor pode, então, revisar e aplicar as alterações, reduzindo o tempo necessário para corrigir falhas sem comprometer a qualidade do software.

Para avaliar a eficácia da proposta, foram analisados 11 repositórios *open-source* em JavaScript e TypeScript, com um total de 8.581 *issues*. Os resultados demonstraram uma taxa média de correção 64.33%, mostrando que a abordagem pode reduzir significativamente a carga de trabalho dos desenvolvedores ao lidar com problemas comuns de código.

Conclui-se que a aplicação de *LLMs* na correção automática de código tem grande potencial para otimizar processos de revisão e manutenção de software. No entanto, desafios como a validação das correções e a adaptação a diferentes linguagens de programação ainda precisam ser aprofundados para aprimorar a precisão e confiabilidade da solução.

Palavras-chaves: Correção automática de código, análise estática de código, Large Language Models (LLMs), SonarQube, automação no desenvolvimento de software.

ABSTRACT

The quality of source code is a critical factor in software development, directly impacting security, maintainability, and system efficiency. However, identifying and fixing issues in the code are often time-consuming tasks that require specialized knowledge. Static analysis tools, such as SonarQube, are widely used to automatically detect problems, but corrections still depend on manual intervention by developers.

In this context, this study proposes the automation of fixing *issues* detected by static analysis tools using language models (*LLMs – Large Language Models*). The developed solution integrates into the development workflow through a *pre-commit* script, allowing an AI model to automatically generate correction suggestions when identifying problems in the code. The developer can then review and apply the changes, reducing the time required to fix issues without compromising software quality.

To evaluate the effectiveness of the proposal, 11 *open-source* repositories in JavaScript and TypeScript were analyzed, with a total of 8,581 *issues*. The results showed an average correction rate of 64.33%, demonstrating that the approach can significantly reduce developers' workload when dealing with common code problems.

It is concluded that the application of *LLMs* in automatic code correction has great potential to optimize code review and maintenance processes. However, challenges such as validating corrections and adapting to different programming languages still need to be further explored to improve the accuracy and reliability of the solution.

Keywords: Automatic code correction, static code analysis, *Large Language Models* (LLMs), SonarQube, software development automation. .

LISTA DE FIGURAS

Figura 1 – Diagrama de blocos da solução proposta. 22
Figura 2 – Fluxograma da solução proposta. 25
Figura 3 – Taxa de Correção Geral de Issues por Repositório 36
Figura 4 – Porcentagem de Correção de Code Smells e Bugs por Repositório 38

LISTA DE TABELAS

Tabela 1 – Resumo dos repositórios e suas características principais 32
Tabela 2 – Quantidade de *Code Smells*, *Bugs* e Vulnerabilidades por Repositório 33
Tabela 3 – Correção Total de Issues por Repositório 35
Tabela 4 – Correção de Code Smells por Repositório 37
Tabela 5 – Correção de Bugs por Repositório 37
Tabela 6 – Top 10 Issues Corrigidas 39
Tabela 7 – Tempo de Execução do SonarAutoFix por Repositório (em segundos) 40

LISTA DE ABREVIATURAS E SIGLAS

API	Application Programming Interface
CoT	Chain of thought
IA	Inteligência artificial
LLM	Large Language Models

SUMÁRIO

1 INTRODUÇÃO 11
1.1 CONTEXTUALIZAÇÃO E MOTIVAÇÃO 11
1.2 OBJETIVOS 12
2 FUNDAMENTAÇÃO TEÓRICA 13
2.1 MODELOS DE LINGUAGEM DE GRANDE ESCALA (LLMS) 13
2.2 ESTRATÉGIAS DE INTERAÇÃO COM LLMS 14
2.2.1 Prompt Engineering 14
2.2.2 Chain of Thought 15
2.3 ANÁLISE DE CÓDIGO 15
2.3.1 Análise Estática de Código 16
2.4 VERSIONAMENTO DE CÓDIGO 17
2.5 APIS (APPLICATION PROGRAMMING INTERFACES) 17
3 TRABALHOS RELACIONADOS 18
4 PROPOSTA 22
4.1 MÓDULO DE EXECUÇÃO 22
4.2 MÓDULO DE ANÁLISE ESTÁTICA 22
4.3 MÓDULO DE GERAÇÃO DE CORREÇÕES 23
4.4 MÓDULO DE APLICAÇÃO DE CORREÇÕES 23
4.5 MÓDULO DE VALIDAÇÃO 24
4.6 FLUXO DE FUNCIONAMENTO 24
5 METODOLOGIA 27
5.1 TECNOLOGIAS UTILIZADAS 27
5.1.1 Ferramenta de Análise Estática: SonarQube 27
5.1.2 Modelo de Linguagem: Claude AI 27
5.1.3 Git e Husky: Integração e Automação 29
5.2 MINERAÇÀO DE ISSUES E CONSTRUÇÃO DOS CONJUNTOS DE DADOS 30
5.3 DESCRIÇÃO DOS REPOSITÓRIOS 31
5.4 IMPLEMENTAÇÃO DA SOLUÇÃO 32
5.5 EXPERIMENTOS 33

5.6 METODOLOGIA DE AVALIAÇÃO 34
6 RESULTADOS E DISCUSSÕES 35
6.1 ANÁLISE GERAL DOS RESULTADOS POR REPOSITÓRIO 35
6.2 CORREÇÃO DE CODE SMELLS 36
6.3 CORREÇÃO DE BUGS 37
6.4 COMPARAÇÃO ENTRE CODE SMELLS E BUGS 38
6.5 TAXA DE CORREÇÃO GERAL 39
6.6 DESEMPENHO TEMPORAL 39
6.7 CUSTO OPERACIONAL 40
6.8 DISCUSSÃO E IMPLICAÇÕES 41
7 CONCLUSÃO 42
REFERÊNCIAS 43

1 INTRODUÇÃO

Esta seção introdutória aborda a contextualização da qualidade do código-fonte e os desafios na detecção e correção de falhas, apresenta as motivações para o uso de soluções automatizadas baseadas em inteligência artificial e define os objetivos desta pesquisa.

1.1 CONTEXTUALIZAÇÃO E MOTIVAÇÃO

A qualidade do código-fonte é um fator essencial no desenvolvimento de software, influenciando diretamente a segurança, a manutenção e o desempenho dos sistemas. No entanto, a identificação e resolução de bugs continuam sendo desafios significativos, frequentemente exigindo tempo e expertise dos desenvolvedores. A crescente complexidade dos sistemas de software tem demandado soluções que automatizam e otimizam o processo de detecção e correção de falhas, reduzindo o esforço manual e os custos associados (ALLAMANIS et al., 2018).

Nesse cenário, a análise estática emerge como uma abordagem relevante, consistindo na avaliação do código-fonte sem sua execução, o que permite identificar precocemente problemas com base em regras predefinidas e padrões de codificação. Ferramentas de análise estática, como SonarQube (SonarQube, 2025) e PMD (PMD, 2025), são amplamente utilizadas para detectar de forma automatizada desde violações de boas práticas, como o uso de variáveis globais desnecessárias, até potenciais vulnerabilidades, como falhas de injeção de SQL. O SonarQube opera integrando análises estáticas e métricas de qualidade, fornecendo relatórios detalhados que abrangem desde duplicação de código até vulnerabilidades de segurança, com suporte a múltiplas linguagens de programação. Já o PMD foca na análise de código em nível granular, identificando problemas como variáveis não utilizadas e práticas inadequadas, sendo especialmente configurável para regras específicas de projetos em Java.

Apesar de sua eficácia na detecção, a correção desses issues identificados pelas ferramentas de análise estática ainda depende majoritariamente da intervenção humana, o que pode gerar empecilhos no fluxo de desenvolvimento, especialmente em projetos de grande escala. Embora essas ferramentas sejam valiosas para localizar falhas, a ausência de suporte para correções automáticas limita sua capacidade de atender demandas de agilidade nos ciclos modernos de desenvolvimento (MONPERRUS, 2019).

Nesse contexto, a aplicação de Modelos de Linguagem de Grande Escala (Large Language

Models (LLM)) surge como uma abordagem promissora para enfrentar essa problemática. Estudos recentes, como o de Kang, Yoon e Yoo (2023), investigam o potencial desses modelos para geração automática de testes capazes de reproduzir bugs a partir de descrições mínimas, demonstrando sua capacidade de compreender e interagir com o código-fonte de forma contextual. Além disso, o uso de LLMs para localizar e reparar *bugs* em nível de *token*, identificando e corrigindo elementos mínimos do código, como palavras-chave, identificadores ou operadores. Por exemplo, um LLM pode identificar que um bug ocorre devido a um operador lógico incorreto (como um && em vez de um ||) em uma expressão condicional e sugerir a correção adequada com base no contexto do código ao redor, como variáveis e intenções implícitas na lógica. (HOSSAIN et al., 2024).

Neste contexto, esta pesquisa investiga como as tecnologias de Inteligência artificial (IA) podem contribuir para a automação da correção de issues, tornando o processo de manutenção mais ágil e preciso. Ao alinhar-se às necessidades da indústria por soluções mais eficientes e menos dependentes de intervenção manual, espera-se que o uso de LLMs represente um avanço significativo na forma como defeitos de software são identificados e corrigidos.

1.2 OBJETIVOS

Diante desse cenário, esta pesquisa propõe a implementação de uma solução automatizada para a detecção e correção de problemas em código-fonte, integrando análise estática, geração de sugestões de correção e aplicação automatizada dessas modificações.

Tem-se, portanto, como objetivo, desenvolver um sistema que facilite e acelere o processo de melhoria contínua do código, reduzindo a necessidade de intervenção manual e garantindo que as correções sejam aplicadas de maneira sistemática e eficiente. Deseja-se otimizar a qualidade do software desde as primeiras etapas do ciclo de desenvolvimento, promovendo maior aderência às boas práticas de codificação e reduzindo a incidência de erros antes mesmo que o código seja integrado ao repositório. Assim, buscamos contribuir para um desenvolvimento mais ágil e confiável, minimizando retrabalho e aumentando a produtividade de equipes de engenharia de *software*.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos fundamentais para o entendimento do trabalho. Dessa forma, serão abordados tópicos sobre Modelos de Linguagem de Grande Escala (LLM), análise de código, versionamento de código e Application Programming Interface (API)s.

2.1 MODELOS DE LINGUAGEM DE GRANDE ESCALA (LLMS)

Modelos de Linguagem de Grande Escala (LLMs) são sistemas de inteligência artificial projetados para entender e gerar linguagem humana de forma avançada, que pertencem à categoria de modelos de linguagem - algoritmos desenvolvidos para prever ou produzir sequências de palavras com base em padrões extraídos de dados textuais. O diferencial dos LLMs está na sua escala em massa: são treinados em enormes conjuntos de dados, como textos da internet, artigos científicos e livros, utilizando bilhões de parâmetros para capturar desde regras gramaticais básicas até relações contextuais complexas (RADFORD et al., 2019). Esse avanço é resultado de melhorias em técnicas de aprendizado profundo e da crescente disponibilidade de recursos computacionais (BENDER et al., 2021).

O funcionamento dos LLMs se baseia na previsão probabilística de palavras em uma sequência ou no preenchimento de lacunas textuais. Por exemplo, ao analisar "O céu está...", o modelo pode sugerir "azul" devido à alta frequência desse padrão nos dados de treinamento. Essa capacidade de generalização, combinada com o processamento de grandes volumes de texto, torna os LLMs ferramentas poderosas em diversas áreas teóricas e práticas (BOMMASANI et al., 2021).

A arquitetura predominante dos LLMs modernos é baseada nos Transformers, uma estrutura de redes neurais profundas que utiliza o mecanismo de atenção para processar sequências de texto (VASWANI et al., 2017). Esse mecanismo permite ao modelo focar em partes relevantes de uma entrada, mesmo em contextos distantes, sendo definido pela equação:

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

onde Q, K e V representam matrizes de consultas, chaves e valores, e d_k é a dimensão das chaves, usada para estabilizar os cálculos. Modelos como GPT-3, descrito por Brown et al. (2020), e LLaMA, apresentado por Touvron et al. (2023), utilizam essa arquitetura em larga escala, com bilhões de parâmetros ajustados para maximizar a generalização. O pré-treinamento

em grandes quantidades de texto é essencial para o desempenho dessas arquiteturas, permitindo que os LLMs capturem nuances linguísticas complexas (DEVLIN et al., 2019).

2.2 ESTRATÉGIAS DE INTERAÇÃO COM LLMS

A interação com LLMs depende de técnicas específicas que orientam o modelo a produzir respostas alinhadas aos objetivos do usuário. Diferentemente de sistemas tradicionais, nos quais comandos rígidos determinam o comportamento, os LLMs respondem a estímulos textuais flexíveis, exigindo estratégias que explorem sua capacidade de interpretar nuances linguísticas (ZHANG et al., 2023). Essas técnicas evoluíram para atender à demanda por saídas mais precisas e contextualmente relevantes, especialmente em cenários que requerem raciocínio ou criatividade.

A comunicação eficaz com LLMs começa com a estruturação clara das instruções, mas vai além, incorporando abordagens que ajustam o tom, o nível de detalhamento e até o formato da resposta. Por exemplo, solicitar que o modelo "explique como um especialista em um tom formal" pode gerar saídas distintas de uma instrução como "dê uma explicação rápida e informal". Essa flexibilidade é um diferencial destacado por Chen, Liu e Wang (2023), que aponta que a adaptação do estilo de entrada influencia diretamente a qualidade da interação.

2.2.1 Prompt Engineering

O *prompt engineering* emerge como uma técnica central para refinar a comunicação com LLMs, consistindo na elaboração estratégica de instruções que direcionam o modelo para resultados específicos. Essa prática envolve não apenas a escolha de palavras, mas também a inclusão de contexto, exemplos ou restrições que moldam a saída (ZHAO; WALLACE; FENG, 2022). Um *prompt* bem projetado pode, por exemplo, transformar uma resposta genérica em uma análise detalhada, como ao pedir: "Liste três benefícios da energia solar, justificando cada um com base em dados econômicos recentes".

A eficácia do *prompt engineering* reside em sua capacidade de contornar limitações inerentes aos LLMs, como a tendência a respostas vagas em perguntas abertas. Sun, Zhou e Li (2023) sugere que prompts estruturados com perguntas específicas e exemplos práticos (*few-shot prompts*) aumentam a precisão em até 30% em tarefas de classificação ou geração de texto.

Um exemplo ilustrativo seria:

Listing 2.1 – Exemplo de prompt estruturado - Categorizacao de ideias

```
Entrada: "Categorize a ideia como positiva ou negativa. Exemplo: 'Trabalhar em
    equipe e bom' -> positiva; 'Falta de dialogo atrapalha' -> negativa. Ideia: '
    Aprender algo novo e legal.'"
Saida esperada: positiva
```

2.2.2 Chain of Thought

A técnica de Chain of thought (CoT) amplia o *prompt engineering* ao incentivar os LLMs a articular um raciocínio sequencial, detalhando cada etapa antes de apresentar uma conclusão. Essa abordagem é particularmente valiosa em tarefas que demandam lógica ou resolução de problemas, pois permite ao modelo simular um processo de pensamento humano (NYE; ANDREASSEN; GUR-ARI, 2023). Diferente de respostas diretas, o CoT revela o caminho lógico percorrido, aumentando a confiabilidade e a interpretabilidade das saídas.

Por exemplo, ao resolver uma questão como “Se um carro percorre 240 km em 3 horas, qual é sua velocidade média?”, um *prompt* com CoT poderia ser:

Listing 2.2 – Exemplo de chain of thought - Escolha de presente

```
Entrada: "Decida qual presente dar a um amigo e explique cada passo."
Saida esperada: "Primeiro, penso no que o amigo gosta: ele prefere livros e jogos
    . Segundo, vejo o orcamento: posso gastar ate 50 reais. Terceiro, comparo
    opcoes: um livro custa 40 reais e um jogo, 45 reais. Escolho o livro porque
    cabe no orcamento e combina com o gosto dele. Portanto, o presente sera um
    livro."
```

Pesquisas como a de Wang, Li e Zhang (2023) mostram que o CoT melhora o desempenho em tarefas de raciocínio em até 40% em comparação com prompts convencionais, especialmente em modelos avançados. Além disso, a técnica facilita a identificação de erros no processo lógico, sendo uma ferramenta promissora em contextos educacionais e analíticos (LI; CHEN; HUANG, 2023).

2.3 ANÁLISE DE CÓDIGO

A análise de código é uma prática central na engenharia de software que visa examinar o código-fonte para identificar defeitos, melhorar sua qualidade ou assegurar aderência a padrões estabelecidos. Essa análise pode ser conduzida de diferentes maneiras, dependendo do

momento e do método aplicado, sendo geralmente classificada em estática, dinâmica e híbrida (CHESS; WEST, 2007). A análise estática examina o código sem executá-lo, permitindo a detecção precoce de erros e a verificação de conformidade com normas, embora possa gerar falsos positivos. Já a análise dinâmica ocorre durante a execução do programa, oferecendo insights sobre o comportamento real do software, mas depende de casos de teste bem definidos e pode não cobrir todos os cenários. A abordagem híbrida combina elementos das duas anteriores, buscando equilibrar a precisão da análise estática com a contextualização da dinâmica, embora exija mais recursos para sua implementação. Cada tipo oferece uma perspectiva única para avaliar o software, contribuindo para a robustez dos sistemas desenvolvidos.

2.3.1 Análise Estática de Código

A análise estática de código consiste em inspecionar o código-fonte por meio de regras predefinidas ou algoritmos, buscando revelar possíveis defeitos. Essa técnica destaca-se por identificar erros sintáticos, vulnerabilidades de segurança e práticas inadequadas ainda antes da fase de testes, sendo especialmente valorizada em sistemas críticos por sua eficácia (CHESS; WEST, 2007). Um exemplo típico é a detecção de variáveis não inicializadas, como ilustrado no trecho em C apresentado a seguir:

Listing 2.3 – Erro detectado por análise estática

```
int calcular(int x) {
    int y;
    return x + y; // Error: y not initialized
}
```

Diversas ferramentas suportam essa análise, como o SonarQube, que analisa código em várias linguagens e identifica bugs, vulnerabilidades e más práticas; o ESLint, especializado em JavaScript para verificar padrões de codificação; e o Coverity, focado em C/C++ e eficiente na detecção de problemas de memória e concorrência (JOHNSON et al., 2013). A eficiência da análise estática reside em sua rapidez e escalabilidade, permitindo processar grandes bases de código em minutos, ao contrário da análise dinâmica, que depende de casos de teste específicos. Segundo Boehm e Basili (2001), essa abordagem reduz significativamente os custos de correção ao antecipar defeitos, enquanto Beller et al. (2018) mostram que projetos com análise estática contínua têm uma queda média de 15% nos bugs pós-lançamento.

2.4 VERSIONAMENTO DE CÓDIGO

O versionamento de código é um processo essencial na engenharia de software que permite rastrear mudanças, gerenciar versões e facilitar a colaboração entre desenvolvedores. Sem um sistema adequado, projetos podem enfrentar inconsistências, perda de histórico ou dificuldades na integração de alterações. Os Sistemas de Controle de Versão (VCS) são as ferramentas que suportam esse processo, sendo divididos em dois tipos principais: centralizados, como o Subversion (SVN), onde um servidor único armazena todas as versões, e distribuídos, como o Git, onde cada desenvolvedor mantém um repositório completo com o histórico de versões (SPINELLIS, 2005).

O Git é um dos sistemas distribuídos mais utilizados, oferecendo operações eficientes como *commit*, *branch* e *merge*, enquanto o GitHub complementa essa funcionalidade com repositórios remotos e recursos como *pull requests* e integração contínua. Um exemplo básico de inicialização de um repositório Git é:

Listing 2.4 – Inicialização de um repositório Git

```
$ git init
$ git add .
$ git commit -m "Primeiro commit"
```

2.5 APIS (APPLICATION PROGRAMMING INTERFACES)

APIs (Interfaces de Programação de Aplicações) são estruturas fundamentais no desenvolvimento de software moderno, permitindo a comunicação entre diferentes sistemas ou serviços. Podem ser classificadas em diversas categorias conforme seu escopo e funcionamento: públicas, acessíveis a qualquer desenvolvedor e frequentemente usadas para integrar plataformas como Google Maps e Twitter; privadas, restritas a um grupo específico dentro de uma organização; REST, que seguem o modelo arquitetural Representational State Transfer, utilizando métodos HTTP para operações; e GraphQL, que oferecem maior flexibilidade, permitindo que o cliente especifique exatamente os dados desejados (MASSE, 2011). Um exemplo prático é a API do SonarQube, utilizada para coletar dados de análise estática, como mostrado abaixo:

Listing 2.5 – Consulta a issues via API do SonarQube

```
curl -u admin:password "http://localhost:9000/api/issues/search?componentKeys=
    meuprojeto"
```

3 TRABALHOS RELACIONADOS

A geração automática de correções para alertas emitidos por ferramentas de análise estática tem sido um tema amplamente explorado na literatura, resultando em diversas abordagens que combinam técnicas de aprendizado de máquina, mineração de repositórios e métodos baseados em regras. Diferentes ferramentas foram propostas com o objetivo de reduzir o esforço manual necessário para a manutenção da qualidade do código e para mitigar vulnerabilidades de segurança. Nesta seção, discutimos algumas dessas abordagens, destacando suas contribuições e diferenças metodológicas.

O Sorald (ETEMADI et al., 2023) é uma ferramenta desenvolvida para gerar automaticamente sugestões de correção para violações detectadas pelo SonarQube. Para isso, emprega mineração de repositórios públicos e análise sintática de código-fonte, identificando padrões de correção recorrentes aplicados por desenvolvedores em projetos reais. O processo utilizado, denominado mineração de transformações de código (*Mining Code Change Patterns*), extrai modificações feitas por humanos para corrigir violações do SonarQube e generaliza essas transformações para aplicação automática em novos códigos. Ao se integrar diretamente ao *pipeline* de desenvolvimento, a ferramenta possibilita a aplicação das correções antes da submissão do código ao repositório, minimizando a necessidade de intervenção manual. Diferentemente de abordagens baseadas em modelos de aprendizado de máquina, utiliza análises estruturais de código-fonte e técnicas de refatoração baseada em padrões, garantindo que as correções sejam aplicadas de maneira precisa e confiável. Dessa forma, reduz significativamente o esforço necessário para lidar com problemas detectados por análise estática. Por outro lado, o Sorald apresenta algumas limitações, como sua especificidade para código Java, o que restringe sua aplicabilidade a projetos desenvolvidos nessa linguagem. Além disso, sua abordagem baseada na mineração de padrões pode não abranger todos os tipos de violações detectadas pelo SonarQube, especialmente aquelas que exigem compreensão semântica mais profunda do código (ETEMADI et al., 2023).

Seguindo uma linha semelhante, mas com um enfoque distinto, o SpongeBugs (MARCILIO et al., 2020) combina transformações baseadas em regras com aprendizado de máquina supervisionado para sugerir correções mais precisas. Diferente do Sorald, que se baseia na identificação de padrões recorrentes, o SpongeBugs emprega um modelo de classificação que seleciona a melhor estratégia de reparo para cada alerta. Esse modelo é treinado com base em dados de

históricos de correções anteriores, permitindo que a ferramenta refine progressivamente suas sugestões. Assim, enquanto o Sorald prioriza a automação de correções a partir de padrões amplamente utilizados, o SpongeBugs busca aprimorar a precisão das recomendações por meio da adaptação a cada contexto específico (MARCILIO et al., 2020). Contudo, o Sorald apresenta limitações ao depender exclusivamente da aplicação de padrões recorrentes, o que pode resultar em correções genéricas e menos eficazes em contextos específicos.

O StaticFixer (JAIN et al., 2023) apresenta uma abordagem voltada para a conversão automática de alertas de análise estática em reparos contextualizados. O StaticFixer se diferencia por empregar um *pipeline* estruturado que combina análise de fluxo de dados e validação por testes de regressão, garantindo que as modificações propostas não introduzam novos defeitos. A ferramenta inicia sua operação coletando alertas de análise estática e, em seguida, aplica um mecanismo de análise semântica para compreender o impacto da correção sugerida. Esse diferencial permite que o StaticFixer tenha um nível mais alto de precisão na geração de reparos, reduzindo a introdução de erros colaterais. Apesar dessas vantagens, sua implementação pode ser mais custosa devido à necessidade de execução de testes automatizados para validar cada alteração (JAIN et al., 2023).

Outro sistema que se destaca no cenário de reparo automático é o PyTy (CHOW; GRAZIA; PRADEL, 2024) que foca especificamente na correção de erros de tipagem em Python. Dada a natureza dinâmica da linguagem, a inferência de tipos é um desafio relevante para garantir a qualidade do código. O PyTy utiliza aprendizado de máquina para sugerir anotações de tipo consistentes, permitindo que desenvolvedores aprimorem a segurança do código sem comprometer sua flexibilidade. Para isso, a ferramenta coleta um grande conjunto de código Python anotado e não anotado, treinando um modelo que prevê e recomenda a inserção de tipos apropriados. Essa abordagem facilita a adoção gradual da tipagem estática no Python, reduzindo a incidência de erros relacionados à inferência de tipos incorreta. No entanto, sua eficácia pode ser impactada pela necessidade de constante adaptação a novas bibliotecas e padrões de desenvolvimento (CHOW; GRAZIA; PRADEL, 2024).

O SAW-BOT (SERBAN et al., 2021) é um mecanismo para corrigir automaticamente alertas de análise estática diretamente no GitHub. Diferente das abordagens anteriores, o SAW-BOT foca na automação do processo dentro do ambiente de controle de versão, extraindo padrões de correção de um grande conjunto de repositórios públicos. A ferramenta opera monitorando os alertas de análise estática em repositórios ativos e sugerindo correções diretamente nas *pull requests*. O diferencial dessa abordagem é a capacidade de inserir sugestões automatizadas de

correção dentro do fluxo de desenvolvimento, permitindo que os desenvolvedores revisem e aceitem as sugestões com facilidade. Entretanto, sua dependência de padrões previamente observados limita sua eficácia em cenários onde as correções exigem um entendimento semântico mais profundo, além de exigir repositórios suficientemente grandes para que os padrões possam ser extraídos com confiabilidade (SERBAN et al., 2021).

O SecureFalcon (FERRAG et al., 2025), introduzido por explora a detecção e correção de vulnerabilidades de segurança utilizando LLMs. O sistema combina aprendizado profundo com aprendizado por reforço, permitindo a adaptação contínua a novas ameaças. Para isso, a ferramenta analisa relatórios de vulnerabilidades conhecidas e treina seus modelos a partir de grandes bases de código vulnerável e corrigido. Além de corrigir problemas detectados, o SecureFalcon busca identificar padrões emergentes de vulnerabilidades, o que o diferencia de outras abordagens. No entanto, sua eficácia depende fortemente da qualidade dos dados de treinamento, e a validação manual das sugestões continua sendo um desafio, especialmente em ambientes críticos onde uma correção inadequada pode introduzir novas falhas de segurança (FERRAG et al., 2025).

O DeepCode AI Fix (BERABI et al., 2024), emprega aprendizado profundo para identificar e corrigir vulnerabilidades no código-fonte. Diferente do SecureFalcon, que combina LLMs com aprendizado por reforço, o DeepCode AI Fix adota uma abordagem supervisionada, analisando relatórios de análise estática para propor correções alinhadas a padrões de segurança. A ferramenta utiliza um grande conjunto de exemplos anotados para treinar seus modelos, permitindo a identificação de vulnerabilidades que poderiam passar despercebidas por métodos convencionais. Sua capacidade de aprendizado contínuo permite refinar as sugestões ao longo do tempo, aumentando a precisão das correções. No entanto, sua eficácia pode ser limitada por vieses nos dados de treinamento, levando a sugestões inadequadas para vulnerabilidades emergentes ou pouco documentadas (BERABI et al., 2024).

Por fim, o RepairLLaMA Silva, Fang e Monperrus (2024), descrito por propõe um modelo otimizado para reparo automático de código, utilizando representações especializadas e adaptadores ajustados para gerar correções eficientes. Diferentemente de abordagens genéricas baseadas em LLMs, o RepairLLaMA busca reduzir o custo computacional associado à geração de sugestões, tornando-se uma alternativa viável para aplicações em larga escala. A ferramenta se destaca pela capacidade de manter a estrutura e legibilidade do código corrigido, algo que pode ser um problema em modelos convencionais que apenas geram substituições textuais diretas. No entanto, sua eficácia depende da disponibilidade de dados de treinamento de qualidade,

e sua adaptação a diferentes linguagens de programação ainda representa um desafio técnico significativo Silva, Fang e Monperrus (2024).

4 PROPOSTA

Neste trabalho, introduzimos o SonarAutoFix, uma solução para a automação da correção de código, reduzindo a necessidade de ajustes manuais. Esta seção descreve sua arquitetura, destacando os módulos que o compõem e seu fluxo de funcionamento. O sistema é estruturado em cinco módulos interconectados, que operam de forma sequencial e se integram ao fluxo de desenvolvimento de software: execução, análise estática, geração de correções, aplicação de correção e validação. A Figura 1 apresenta um diagrama de blocos da solução proposta.

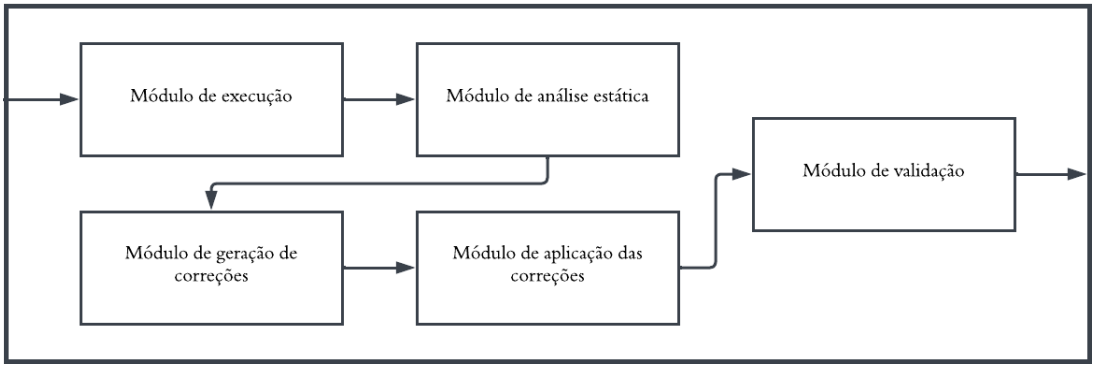

Figura 1 – Diagrama de blocos da solução proposta.

4.1 MÓDULO DE EXECUÇÃO

O módulo de execução é responsável pela orquestração do fluxo de trabalho, instanciando os módulos subsequentes e chamando seus métodos com os devidos parâmetros. Ele é acionado sempre que há uma submissão no código-fonte. A principal função desse módulo é coordenar a execução eficiente do pipeline de análise e correção, encaminhando as saídas de cada etapa como entradas para a subsequente.

4.2 MÓDULO DE ANÁLISE ESTÁTICA

A análise estática do código-fonte visa identificar problemas sem a necessidade de execução do programa. Essa análise é realizada por meio de ferramentas especializadas, como SonarQube (SonarQube, 2025), ESLint (ESLint, 2025) e PMD (PMD, 2025), que detectam padrões inadequados e más práticas no desenvolvimento de software. O uso dessas ferramentas é vantajoso, pois permite uma verificação automatizada, reduzindo a intervenção manual e garantindo maior con-

sistência na identificação de falhas. Os erros identificados podem ser classificados em diversas categorias, incluindo erros de lógica, como uma condição que sempre resulta em verdadeiro ou falso e impacta o fluxo da aplicação; más práticas de programação, como o uso de variáveis não inicializadas, laços infinitos e redundância de código; vulnerabilidades de segurança, como injeção de SQL, falhas de autenticação e manipulação inadequada de dados sensíveis; e problemas de desempenho, como alocação excessiva de memória ou uso ineficiente de estruturas de dados. O módulo retorna um relatório estruturado contendo as falhas detectadas, classificadas por gravidade e impacto no código.

4.3 MÓDULO DE GERAÇÃO DE CORREÇÕES

Com base na lista de problemas identificados pelo módulo de análise estática, este módulo retorna correções automatizadas geradas por meio de LLM. Esses modelos se destacam pela capacidade de interpretar padrões complexos e gerar código alinhado às boas práticas da indústria, garantindo tanto legibilidade quanto eficiência. Além disso, a adaptabilidade dos LLMs a diferentes linguagens e domínios amplia sua aplicabilidade, reduzindo o esforço manual dos desenvolvedores e, consequentemente, elevando a qualidade do software produzido.

Para otimizar essa interação, utilizam-se técnicas de *prompt engineering*, que consistem na formulação precisa de solicitações, assegurando respostas altamente contextualizadas ao código analisado. Complementarmente, a estratégia de *chain of thought*, que orienta o modelo a raciocinar em etapas lógicas antes de propor soluções, possibilita sugestões mais precisas e fundamentadas, alinhadas às necessidades específicas do código.

4.4 MÓDULO DE APLICAÇÃO DE CORREÇÕES

Após a geração das sugestões, este módulo substitui automaticamente cada arquivo de código-fonte por sua versão corrigida, conforme gerado pelo módulo de correções. A substituição é realizada de forma integral para cada arquivo, garantindo que todas as melhorias sugeridas sejam aplicadas de maneira consistente e padronizada.

4.5 MÓDULO DE VALIDAÇÃO

Por fim, o módulo de validação permite que o desenvolvedor revise as alterações realizadas. Se necessário, ele pode realizar ajustes manuais ou rejeitar determinadas modificações. Esse processo é essencial para garantir que as correções aplicadas não comprometam a lógica do software e estejam alinhadas com as expectativas do projeto.

4.6 FLUXO DE FUNCIONAMENTO

O fluxo de funcionamento da solução proposta pode ser representado por meio de um fluxograma, como ilustrado na Figura 2. Este diagrama oferece uma visão abstrata das etapas envolvidas, destacando a interação entre os módulos e o papel do desenvolvedor no processo.

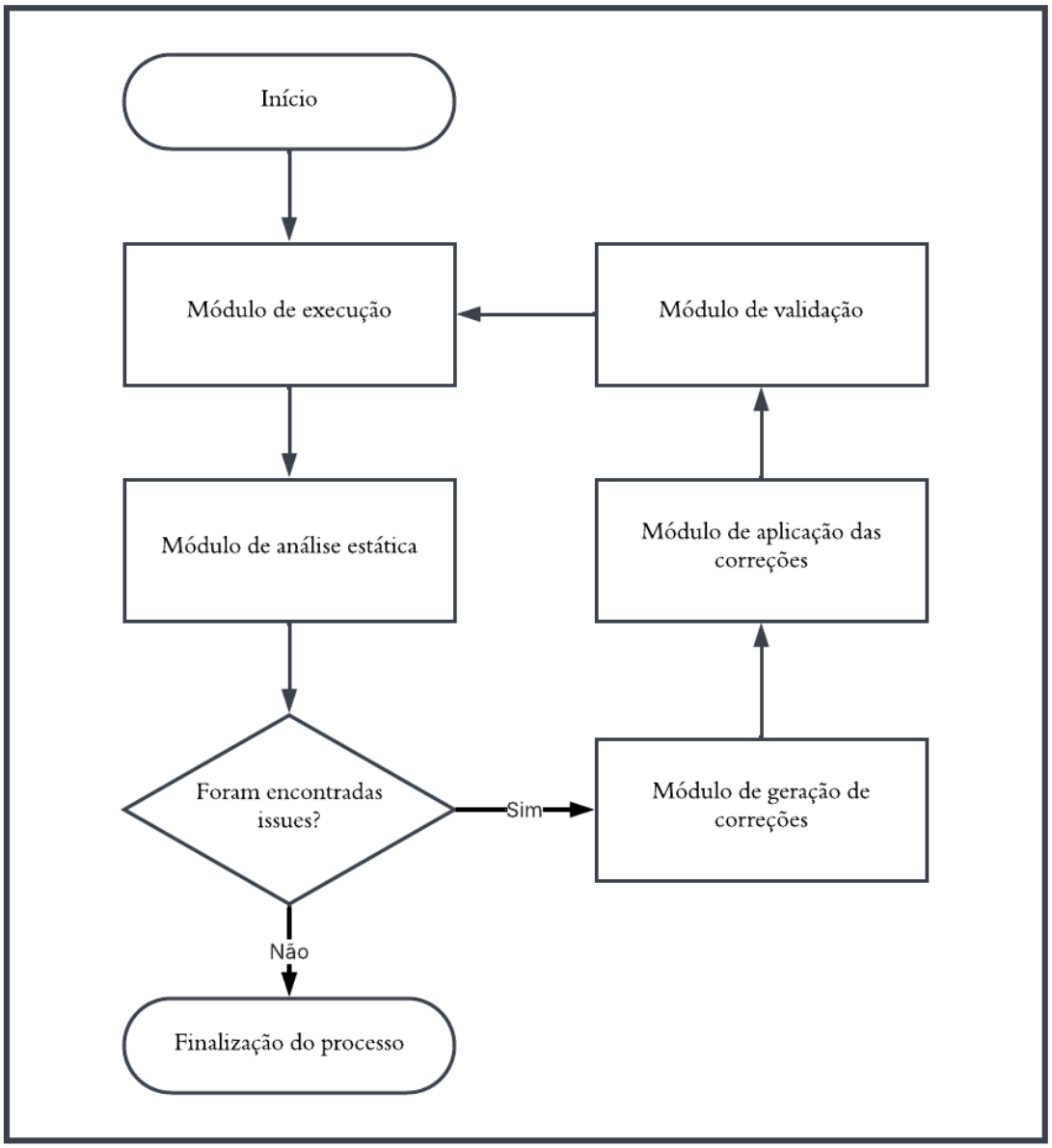

Figura 2 – Fluxograma da solução proposta.

O processo inicia com a submissão de alterações no código, acionando o Módulo de Execução automaticamente. O Módulo de Análise Estática examina o código-fonte e detecta problemas, como erros de lógica ou más práticas. As informações seguem para o Módulo de Geração de Correções, que propõe soluções específicas. O Módulo de Aplicação de Correções atualiza o código, e o Módulo de Validação submete as alterações ao desenvolvedor para revisão. Após aprovação, o código é integrado ao repositório; caso o desenvolvedor faça ajustes, o processo pode ser reiniciado.

O pseudocódigo a seguir descreve a lógica da solução proposta, demonstrando a interação entre os módulos e a estrutura do fluxo de comunicação:

A lógica em *loop* reflete a capacidade do sistema de reagir a ajustes manuais, reiniciando o

Algorithm 1 Algoritmo da Solução Proposta

Entrada: Código-fonte submetido pelo desenvolvedor
Saída: Código-fonte corrigido e validado
Iniciar Módulo de Execução ao detectar submissão ou alteração no código
loop
    $problemas \leftarrow$ Executar Módulo de Análise Estática no código
    if $problemas \neq \emptyset$ **then**
        $sugestoes \leftarrow$ Executar Módulo de Geração de Correções($problemas$)
        $codigo_corrigido \leftarrow$ Executar Módulo de Aplicação de Correções($codigo, sugestes$)
        Disponibilizar $codigo_corrigido$ ao Módulo de Validação
        Aguardar revisão do desenvolvedor
        if Correções aprovadas **then**
            Integrar $codigo_corrigido$ ao repositório
        else
            if Desenvolvedor realiza ajustes manuais **then**
                Reiniciar Módulo de Execução com o código ajustado
            end if
        end if
    else
        Integrar código diretamente ao repositório
    end if
    Sair do loop se não houver ajustes manuais
end loop
Finalizar processo

Módulo de Execução para garantir que o código revisado pelo desenvolvedor permaneça livre de problemas antes da integração.

5 METODOLOGIA

Esta seção descreve os procedimentos adotados para a implementação e avaliação da solução proposta, detalhando as escolhas tecnológicas, os métodos de coleta de dados e os critérios utilizados para análise dos resultados.

5.1 TECNOLOGIAS UTILIZADAS

A implementação da solução demandou a seleção de tecnologias específicas para cada etapa do processo. As escolhas foram fundamentadas em critérios como compatibilidade, desempenho e aceitação na indústria.

5.1.1 Ferramenta de Análise Estática: SonarQube

Para a análise estática do código, optou-se pelo SonarQube devido à sua capacidade de suportar múltiplas linguagens, com destaque para JavaScript e TypeScript, e por proporcionar uma classificação detalhada das *issues*. Essas *issues* são categorizadas pelo SonarQube em três tipos principais: *bugs*, que representam erros de código que podem causar falhas de execução ou comportamentos inesperados; *code smells*, que indicam práticas de programação, como duplicação de código ou complexidade excessiva, afetando a manutenibilidade e legibilidade; e vulnerabilidades, que apontam falhas de segurança passíveis de exploração, como injeções de código ou exposições de dados sensíveis. Além disso, a integração do SonarQube com *pipelines* de desenvolvimento e a possibilidade de execução automatizada via API foram fatores determinantes na escolha da ferramenta, reforçando sua adequação ao fluxo de trabalho proposto.

5.1.2 Modelo de Linguagem: Claude AI

A geração de correções foi conduzida por meio da utilização da Claude AI, desenvolvida pela Anthropic. Esse modelo foi selecionado devido à sua capacidade de processar instruções estruturadas e produzir código corrigido a partir de descrições detalhadas das *issues*, exibindo desempenho notável em tarefas de *prompting*. A escolha também foi fundamentada pela facilidade de integração proporcionada pelo pacote NPM (*Node Package Manager*), que viabilizou a comunicação eficiente com o sistema.

A configuração do Claude foi implementada por meio de uma integração com a API da Anthropic, utilizando o pacote `@anthropic-ai/sdk`. Os parâmetros do modelo foram definidos com base em variáveis de ambiente gerenciadas pelo serviço `ConfigService` do framework NestJS. O modelo padrão adotado foi o `claude-3-5-sonnet-20241022`, escolhido por sua capacidade avançada de geração de código, configurado pela constante `CLAUDE_MODEL`. A temperatura, estabelecida em 0.2 (`CLAUDE_TEMPERATURE`), foi ajustada para garantir respostas consistentes e reduzir a aleatoriedade, um fator crítico em tarefas de correção de código onde a precisão é essencial. O parâmetro `CLAUDE_MAX_TOKENS`, fixado em 4000, foi definido para suportar a geração de respostas extensas, permitindo que o modelo processasse trechos de código complexos sem truncamento.

A interação com o Claude foi otimizada pelo uso do método `client.messages.stream`, que implementa o modo de *streaming* para processar respostas de forma incremental. Esse mecanismo foi crucial para lidar com grandes volumes de dados de maneira eficiente, garantindo que o modelo pudesse retornar correções em tempo real. A configuração dos prompts foi estruturada para maximizar a eficácia do Claude, utilizando técnicas de *prompt engineering* e *chain of thought*. O *prompt* do sistema, apresentado a seguir, define o contexto e estabelece diretrizes rigorosas:

Listing 5.1 – Prompt do sistema utilizado no Claude

```
You are an expert TypeScript/JavaScript developer specializing in SonarQube
compliance. Your task is to fix code issues identified by SonarQube. Provide only
the complete fixed code without explanations.
IMPORTANT: Your fixed code MUST NOT introduce any new SonarQube issues.
Follow these strict guidelines:
1. Address only the specific SonarQube issues mentioned in the prompt
2. Maintain the existing code structure where possible
3. Use established patterns from the codebase
4. DO NOT introduce new code smells such as unused imports, unused variables, or
    unused functions
5. DO NOT add new imports unless absolutely necessary for the fix
6. Pay special attention to maintaining consistent styling to avoid SonarQube
    complexity issues
7. Ensure all imports and functions used in the code are necessary; remove any
    that become unused after your fixes
8. DO NOT add comments about potential issues or TODOs
9. Return only clean, production-ready code that will pass SonarQube scans
```

Já o *prompt* do usuário detalha as *issues* específicas do SonarQube e o código a ser corrigido, estruturando o raciocínio em etapas claras:

Listing 5.2 – Prompt do usuário para correção de código

```
Fix the following TypeScript/JavaScript code according to the SonarQube issues:
Issues to fix:
Line X: [descricao da issue] ([severidade] [regra])
Current code:
```typescript
[codigo original]
```
CRITICAL REQUIREMENTS:
Your fixes MUST NOT introduce ANY new SonarQube issues
1. Pay special attention to avoid introducing:
2. Unused imports or variables
    - Unused functions
    - Any new code smells
    - Any new bugs
    - Any new vulnerabilities
3. Fix ONLY the specific issues listed above
4. DO NOT add any new imports unless they are absolutely necessary
5. If your fixes make any existing imports or functions unnecessary, REMOVE them.

Provide the complete fixed version of the code that addresses all issues. Keep
the same formatting style and only make necessary changes to fix the reported
issues. Return only the code without any explanations, wrapped in markdown code
block.
```

Essa abordagem reflete o uso de *chain of thought*, pois segmenta o problema em instruções específicas e sequenciais, orientando o modelo a raciocinar sobre cada etapa da correção, enquanto o *prompt engineering* é evidenciado pela precisão e clareza das diretrizes, evitando ambiguidades e garantindo resultados alinhados aos requisitos do SonarQube. Assim, a integração do Claude ao sistema não apenas automatiza a correção de código, mas também exemplifica uma aplicação prática de técnicas avançadas de interação com modelos de linguagem.

5.1.3 Git e Husky: Integração e Automação

A integração com o Git foi implementada por meio de *hooks*, assegurando a execução automática do SonarAutoFix antes de cada *commit*. Para otimizar esse processo, foi adotado o Husky como ferramenta de gerenciamento de *pre-commit hooks*, selecionado por sua capacidade de proporcionar uma configuração eficiente da automação, além de sua ampla adoção em projetos de desenvolvimento que utilizam Git.

5.2 MINERAÇÀO DE ISSUES E CONSTRUÇÃO DOS CONJUNTOS DE DADOS

Esta etapa do trabalho consistiu na identificação e seleção de repositórios públicos que apresentassem uma quantidade significativa de problemas de código. Para tanto, foram selecionados repositórios *open-source* de tecnologias amplamente reconhecidas no mercado de tecnologia. A escolha baseou-se em experiências prévias de desenvolvimento, priorizando repositórios que utilizam as linguagens JavaScript e TypeScript. Essas linguagens foram adotadas em razão de sua extensa disseminação na comunidade de desenvolvimento e da disponibilidade de ferramentas de análise estática compatíveis.

O processo de mineração seguiu as seguintes etapas:

1. Definição de uma lista inicial de repositórios populares, com ênfase em projetos ativos e amplamente reconhecidos pela comunidade do GitHub, priorizando aqueles com maior engajamento e relevância;

2. Realização de uma análise preliminar por meio da API do SonarQube, visando quantificar as *issues* presentes em cada repositório e obter uma visão inicial de sua complexidade e qualidade de código;

3. Aplicação de um processo de filtragem dos repositórios com base em um limiar mínimo de *issues*, garantindo representatividade e robustez para os experimentos. Nesse sentido, foram priorizados repositórios com maior quantidade de *issues*, a fim de assegurar uma base de dados mais substancial para análise.

Foram selecionados 11 repositórios de software com base em sua relevância e ampla adoção na indústria, priorizando projetos bem documentados e que desempenham papéis fundamentais no desenvolvimento moderno de aplicações. Esses repositórios, amplamente utilizados em projetos baseados em JavaScript, TypeScript e Node.js, oferecem soluções robustas para diversas necessidades, desde a construção de APIs até o gerenciamento de estado e manipulação de dados. A seguir, apresenta-se uma análise detalhada de cada um, destacando suas características e contribuições para a comunidade de desenvolvimento.

5.3 DESCRIÇÃO DOS REPOSITÓRIOS

O repositório Nest.js (NestJS, 2025) é um framework voltado para a criação de aplicações escaláveis em Node.js. Desenvolvido com TypeScript, ele utiliza uma estrutura organizada e inspirada no Angular (Google, 2025), facilitando a gestão de projetos complexos. Complementarmente, o Jest (Jest, 2025), é uma ferramenta de testes para JavaScript, amplamente reconhecida por sua simplicidade e eficácia em testes unitários e de integração. Já o Prisma (Prisma, 2025), um *Object-Relational Mapping* (ORM) moderno para Node.js e TypeScript, torna mais fácil e seguro o trabalho com bancos de dados relacionais.

No contexto de manipulação de dados, o Date-fns (Date-fns, 2025) é uma biblioteca leve e eficiente para lidar com datas em JavaScript, oferecendo funções confiáveis e consistentes para aplicações. Para requisições HTTP, o Axios (Axios, 2025), um cliente baseado em promessas, é muito utilizado por funcionar bem tanto em Node.js quanto em navegadores, sendo ideal para tarefas assíncronas. Além disso, o Zod (Zod, 2025) proporciona uma forma prática de validar e definir tipos em TypeScript, assegurando dados consistentes, enquanto o tRPC (tRPC, 2025) simplifica a criação de APIs seguras, eliminando a complexidade de padrões tradicionais como esquemas REST.

Para *logging*, o Pino (Pino, 2025) se destaca como uma ferramenta de alto desempenho em Node.js, projetada para consumir poucos recursos e atender a aplicações que exigem eficiência. O Express (Express, 2025), por sua vez, é uma das tecnologias mais populares para desenvolvimento web em Node.js, graças à sua estrutura flexível e prática, muito usada na criação de APIs e servidores. No gerenciamento de estado, o Redux (Redux, 2025) é uma referência em aplicações JavaScript, especialmente com React (Meta, 2025) e Angular (Google, 2025), por manter os dados organizados e previsíveis. Por fim, o RxJS (RxJS, 2025) complementa o conjunto com uma abordagem reativa que facilita o controle de informações dinâmicas e operações assíncronas.

A tabela 1 sintetiza as características principais dos repositórios analisados, destacando suas funcionalidades e diferenciais.

Tabela 1 – Resumo dos repositórios e suas características principais

Repositório	**Descrição**	**Principais Características**
Nest.js	Framework para aplicações escaláveis em Node.js	Arquitetura modular, TypeScript, inspirado no Angular
Jest	Framework de testes para JavaScript	Testes unitários e de integração, criado pela Meta
Prisma	ORM moderno para Node.js e TypeScript	Tipagem, simplificação de acesso a bancos relacionais
Date-fns	Biblioteca para manipulação de datas	Funções eficientes, imutáveis, leve
Axios	Cliente HTTP baseado em Promises	Requisições assíncronas, compatível com Node.js e navegadores
Zod	Validação e inferência de tipos em TypeScript	Consistência de dados, integração com TypeScript
tRPC	Framework para APIs tipadas	Elimina esquemas REST, segurança de tipos
Pino	Biblioteca de logging para Node.js	Alto desempenho, baixo consumo de recursos
Express	Framework para aplicações web em Node.js	Baseado em middleware, popular para APIs
Redux	Gerenciamento de estado em JavaScript	Estado previsível, amplo uso com React
RxJS	Programação reativa com observáveis	Manipulação de fluxos assíncronos, dados reativos

5.4 IMPLEMENTAÇÃO DA SOLUÇÃO

A implementação do SonarAutoFix foi estruturada como um conjunto de módulos integrados, desenvolvidos em Typescript e configurados para operar no momento do pre-commit em sistemas de controle de versão como o Git. O sistema foi projetado para seguir o fluxo proposto na seção anterior, com as seguintes etapas práticas:

1. **Módulo de Execução:** Acionado no pre-commit, este módulo coordena a execução dos demais componentes e registra logs detalhados de cada etapa;

2. **Módulo de Análise Estática:** Utiliza o SonarQube para identificar *issues*, organizando-as por arquivo e localização no código;

3. **Módulo de Geração de Código:** Envia as *issues* e trechos de código à Claude AI, que retorna sugestões de correção;

4. **Módulo de Aplicação de Correções:** Substitui os arquivos originais pelos corrigidos, mantendo-os em estado *unstaged* no Git para revisão;

5. **Revisão pelo Desenvolvedor:** Permite validação manual das alterações antes da integração ao repositório.

5.5 EXPERIMENTOS

Inicialmente, foi realizada a execução do SonarQube em todos os repositórios selecionados, com o objetivo de identificar e quantificar as *issues* presentes antes da aplicação do SonarAutoFix. Essa análise preliminar permitiu mapear a quantidade de *code smells*, *bugs* e vulnerabilidades em cada repositório, fornecendo uma base para a avaliação da solução proposta. Os resultados dessa análise inicial são apresentados na Tabela 2.

Tabela 2 – Quantidade de *Code Smells*, *Bugs* e Vulnerabilidades por Repositório

Repositório	**Qtd. de *Code Smells***	**Qtd. de *Bugs***	**Qtd. de Vulnerabilidades**
nest	483	17	0
jest	712	41	0
prisma	971	92	0
date-fns	388	239	0
axios	208	25	1
zod	281	30	0
tRPC	457	56	0
pino	205	1	0
express	1776	2	0
redux	191	23	0
rxjs	2364	19	0

Posteriormente, foram realizados experimentos com o objetivo de validar a efetividade da solução proposta, por meio da implementação de um subprocesso em Python que processou os repositórios selecionados de forma sistemática. O procedimento consistiu na clonagem individual de cada repositório a partir de uma lista previamente definida, seguida pela aplicação do SonarAutoFix, que utiliza a análise estática do SonarQube para identificar e corrigir *issues*. Após cada execução, uma nova análise com o SonarQube foi realizada para quantificar as *issues* remanescentes. Para garantir a independência dos testes, cada repositório foi removido do am-

biente ao final de seu processamento, possibilitando a análise sequencial do próximo repositório da lista.

5.6 METODOLOGIA DE AVALIAÇÃO

A avaliação do desempenho do SonarAutoFix foi baseada na análise da redução de *issues* nos repositórios selecionados. A métrica principal utilizada foi a taxa de correção, calculada conforme a equação 5.1, que expressa a redução percentual de *issues* após a aplicação da solução:

$$\text{Taxa de correção} = \left(1 - \frac{\text{Total de } \textit{Issues} \text{ Finais}}{\text{Total de } \textit{Issues} \text{ Iniciais}}\right) \times 100 \tag{5.1}$$

A análise considerou apenas as categorias de *bugs* e *code smells*, devido à baixa incidência de vulnerabilidades, conforme evidenciado na Tabela 2. Além disso, as *rules* identificadas pelo SonarQube para cada *issue* foram examinadas, possibilitando uma avaliação detalhada do desempenho da solução em diferentes tipos de problemas de código.

6 RESULTADOS E DISCUSSÕES

Este capítulo apresenta os resultados obtidos com a aplicação do SonarAutoFix em repositórios de código aberto, analisando sua eficácia na correção automática de *issues* — abrangendo *code smells* e *bugs* — e na melhoria da qualidade do código. Os dados são organizados em tabelas e gráficos, acompanhados de análises quantitativas e qualitativas, que permitem avaliar o desempenho da ferramenta em diferentes contextos de software. São examinados a redução de defeitos, os tipos de problemas corrigidos, as taxas de correção por repositório e a eficácia geral.

6.1 ANÁLISE GERAL DOS RESULTADOS POR REPOSITÓRIO

A Tabela 3 resume a quantidade inicial de *issues* (combinando *code smells* e *bugs*), os corrigidos pelo SonarAutoFix e os remanescentes após sua aplicação em cada repositório analisado.

Tabela 3 – Correção Total de Issues por Repositório

Repositório	**Qtd. Inicial**	**Qtd. Corrigida**	**Qtd. Final**
nest	500	295	205
jest	753	313	440
prisma	1063	592	471
date-fns	627	349	278
axios	233	67	166
zod	311	154	157
trpc	513	305	208
pino	206	182	24
express	1778	986	792
redux	214	191	23
rxjs	2383	2087	296

Os resultados mostram variações expressivas no desempenho da ferramenta. Em *rxjs*, por exemplo, dos 2383 *issues* iniciais, 2087 (87,57%) foram corrigidos, enquanto em *axios*, dos 233, apenas 67 (28,63%) foram resolvidos. Essa diferença indica que fatores como complexidade do código, tipos predominantes de *issues* e particularidades dos projetos — como padrões de codificação ou dependências — impactam a eficácia do SonarAutoFix. Tal observação sugere a necessidade de investigar mais a fundo os limites contextuais da abordagem.

Para uma visão consolidada, a Figura 3 ilustra as taxas de correção geral por repositório em um gráfico de barras horizontais, expressas em porcentagem.

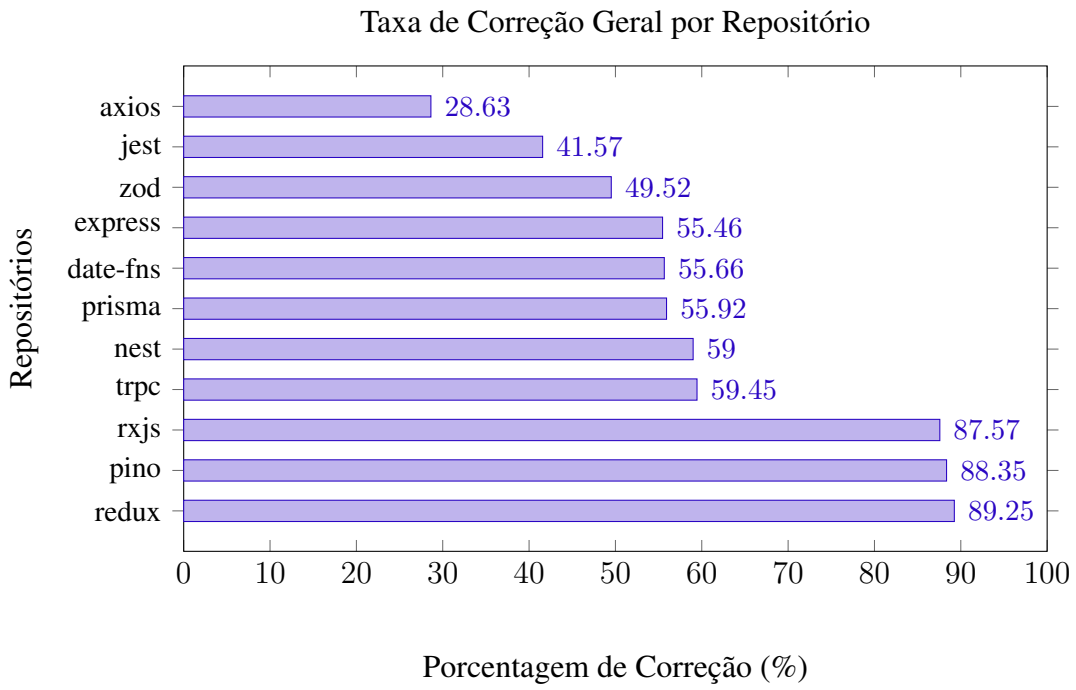

Figura 3 – Taxa de Correção Geral de Issues por Repositório

A análise gráfica confirma os padrões da Tabela 3. Repositórios como *redux* (89,25%), *pino* (88,35%) e *rxjs* (87,57%) alcançam taxas acima de 87%, sugerindo alta eficácia em projetos com *issues* estruturais que demandam menos interpretação semântica. Já a baixa taxa em *axios* (28,63%) pode indicar dificuldades com *issues* mais contextuais ou específicos, apontando para a necessidade de explorar esses desafios em maior detalhe.

6.2 CORREÇÃO DE CODE SMELLS

A Tabela 4 detalha os resultados da correção de *code smells*, que afetam a legibilidade, manutenibilidade e escalabilidade do código.

Os dados mostram eficácia significativa em *redux* (90,05%), *pino* (88,78%) e *rxjs* (87,73%), indicando que o SonarAutoFix lida bem com *code smells* estruturais, como variáveis redundantes ou padrões obsoletos. Taxas menores em *axios* (24,52%) e *date-fns* (39,69%) sugerem limitações em problemas mais contextuais, que podem exigir maior compreensão semântica.

Tabela 4 – Correção de Code Smells por Repositório

Repositório	Qtd. Inicial	Qtd. Corrigida	Qtd. Final
nest	483	282	201
jest	712	288	424
prisma	971	550	421
date-fns	388	154	234
axios	208	51	157
zod	281	130	151
trpc	457	270	187
pino	205	182	23
express	1776	985	791
redux	191	172	19
rxjs	2364	2074	290

6.3 CORREÇÃO DE BUGS

A Tabela 5 foca na correção de *bugs*, defeitos que comprometem a funcionalidade e a estabilidade do software.

Tabela 5 – Correção de Bugs por Repositório

Repositório	Qtd. Inicial	Qtd. Corrigida	Qtd. Final
nest	17	13	4
jest	41	25	16
prisma	92	42	50
date-fns	239	195	44
axios	25	16	9
zod	30	24	6
trpc	56	35	21
pino	1	0	1
express	2	1	1
redux	23	19	4
rxjs	19	13	6

Repositórios como *date-fns* (81,59%), *zod* (80%) e *redux* (82,61%) destacam-se na correção de *bugs*, evidenciando a capacidade do SonarAutoFix em tratar defeitos críticos. A ausência de correção em *pino* (0%) e a baixa taxa em *express* (50%) apontam dificuldades com *bugs* menos frequentes ou de lógica dinâmica.

6.4 COMPARAÇÃO ENTRE CODE SMELLS E BUGS

A Figura 4 compara as taxas de correção de *code smells* e *bugs*, integrando os resultados.

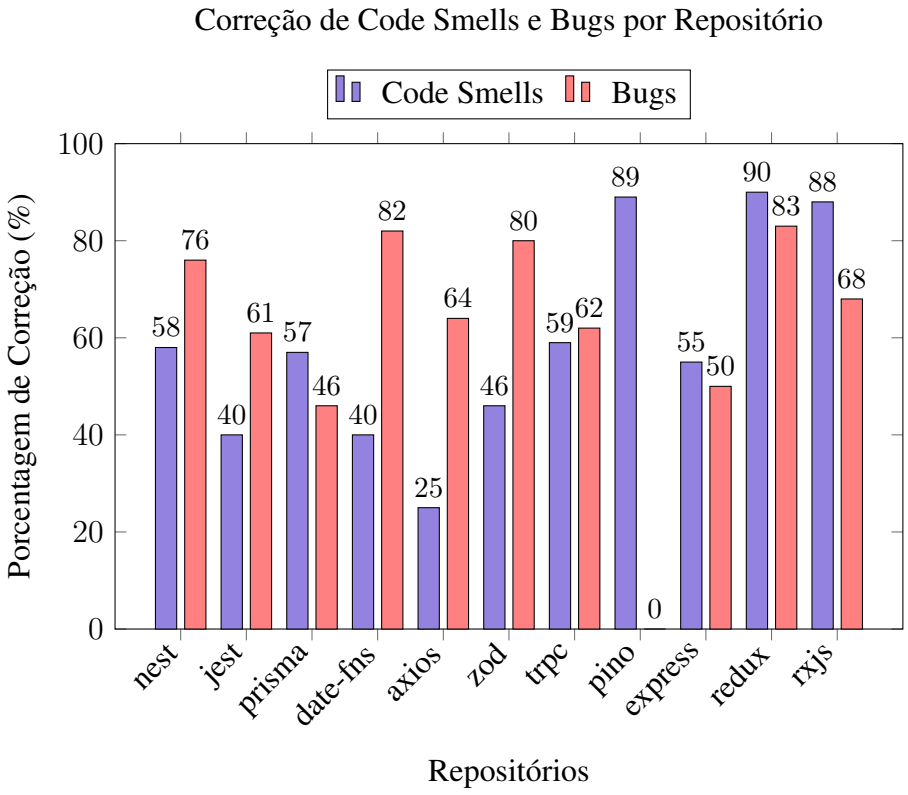

Figura 4 – Porcentagem de Correção de Code Smells e Bugs por Repositório

O gráfico revela que a correção de *bugs* frequentemente supera a de *code smells*, como em *date-fns* e *zod*, sugerindo maior precisão em defeitos funcionais. A discrepância em *pino* (89% para *code smells*, 0% para *bugs*) destaca a necessidade de ajustes para equilibrar o desempenho entre os tipos de *issues*.

A Tabela 6 lista os dez tipos de *issues* mais corrigidos, com base nas regras do SonarQube.

A predominância de *code smells* como S1854 (variáveis não utilizadas) e S3504 (uso inadequado de *var*) reflete a força da ferramenta em questões estruturais. Contudo, a presença de S3504 entre as *issues* remanescentes sugere desafios em casos mais complexos, indicando limites na abordagem atual e oportunidades para ajustes.

Tabela 6 – Top 10 Issues Corrigidas

Regra	Tipo	Severidade	Qtd. Corrigida
S1854	CODE_SMELL	MAJOR	1059
S3504	CODE_SMELL	CRITICAL	1012
S1874	CODE_SMELL	MINOR	681
S1186	CODE_SMELL	CRITICAL	252
S1135	CODE_SMELL	INFO	237
S4325	CODE_SMELL	MINOR	195
S6035	CODE_SMELL	MAJOR	177
S4335	BUG	CRITICAL	123
S3776	CODE_SMELL	CRITICAL	115
S125	CODE_SMELL	MAJOR	99

6.5 TAXA DE CORREÇÃO GERAL

A taxa de correção geral foi calculada pela Equação 5.1, apresentada na Seção 5.6, considerando os totais de *issues* antes (8581) e depois (3060) da aplicação:

$$\text{Taxa de Correção} = \left(1 - \frac{3060}{8581}\right) \times 100 = 64,33\% \tag{6.1}$$

Com 64,33%, o SonarAutoFix demonstra viabilidade com um modelo comercial como Claude, superando expectativas para uma solução genérica. Comparado a ferramentas especializadas como SpongeBugs (MARCILIO et al., 2020) e Sorald (ETEMADI et al., 2023), o resultado reflete um bom equilíbrio entre generalização e eficácia, com os 35% remanescentes apontando caminhos para melhorias, como *prompts* mais elaborados, utilização de técnicas diferentes como *fine-tuning* ou customização do modelo.

6.6 DESEMPENHO TEMPORAL

A Tabela 7 apresenta os tempos de execução do SonarAutoFix em cada etapa do processo — inicialização do SonarAutoFix (Init), obtenção dos arquivos modificados pelo usuário (Files), análise do SonarQube (Análise), obtenção de issues do SonarQube (Issues), geração de correções automáticas (Geração), aplicação das correções (Aplicação) e tempo total (Total) — para os repositórios analisados, com uma linha final indicando a média de cada etapa.

Tabela 7 – Tempo de Execução do SonarAutoFix por Repositório (em segundos)

Rep.	Init	Files	Análise	Issues	Geração	Aplicação	Total
nest	0.02	0.04	115.69	13.06	2032.35	0.07	2161.23
jest	0.02	0.05	141.46	17.57	3576.00	0.10	3735.21
prisma	0.02	0.07	152.22	15.61	4678.26	0.18	4846.35
date-fns	0.01	0.02	54.64	23.78	2646.89	0.12	2725.46
axios	0.02	0.01	25.23	4.39	600.25	0.01	629.91
zod	0.02	0.01	21.88	4.40	669.95	0.03	696.29
trpc	0.01	0.01	44.13	6.54	2426.08	0.12	2476.89
pino	0.02	0.01	37.04	4.75	580.29	0.01	621.84
express	0.02	0.01	20.20	2.63	1155.13	0.04	1178.04
redux	0.01	0.01	18.69	2.24	576.32	0.01	597.28
rxjs	0.01	0.01	60.36	11.18	5600.12	0.16	5671.84
Média	0.015	0.021	61.505	9.83	2267.88	0.078	2355.76

Os tempos apresentados na Tabela 7 referem-se à execução completa do SonarAutoFix em todo o código de cada repositório analisado. Observa-se que a etapa de geração de correções (Geração) é a mais dispendiosa, variando de 576,32 segundos em redux a 4.678,26 segundos em prisma, de acordo com a quantidade de arquivos e linhas a serem analisadas em cada repositório. Em contraste, etapas como inicialização (Init) e aplicação de correções (Aplicação) apresentam tempos irrisórios, com médias de 0,015 e 0,078 segundos, respectivamente.

É importante destacar que esses tempos são relativos ao processamento integral dos repositórios, que frequentemente contêm grandes volumes de código. Em cenários mais comuns no cotidiano de um desenvolvedor, nos quais as quantidades de código analisadas são significativamente menores — como trechos específicos ou módulos isolados —, os tempos de execução tendem a ser proporcionalmente reduzidos. Assim, a aplicação do SonarAutoFix em contextos práticos de desenvolvimento pode oferecer respostas mais rápidas, alinhando-se às necessidades de eficiência em fluxos de trabalho reais.

6.7 CUSTO OPERACIONAL

Um aspecto relevante na avaliação do SonarAutoFix é o custo operacional associado ao uso de LLM comerciais. Neste trabalho, utilizou-se o modelo de linguagem Claude AI para processar os 11 repositórios analisados, que, juntos, totalizam aproximadamente 627,9 mil linhas de código e 8.581 issues identificadas pelo SonarQube.

Notavelmente, o custo total para a execução completa do projeto, abrangendo a tentativa de geração de correções para todas as issues, foi de apenas 50 dólares. Esse valor reflete o consumo de créditos na API do Claude, considerando o volume de *tokens* processados, tanto na entrada (descrição das issues e trechos de código) quanto na saída (sugestões de correção). A eficiência econômica observada destaca uma vantagem significativa do uso de LLMs comerciais em tarefas de refatoração automática, especialmente quando comparada ao esforço manual que seria necessário para tratar um volume equivalente de código e issues.

Esse custo reduzido sugere que o SonarAutoFix, integrado ao Claude, apresenta um modelo viável para projetos de grande escala, com uma relação custo-benefício favorável. Por exemplo, considerando uma média aproximada de 0,0058 dólares por issue corrigida.

6.8 DISCUSSÃO E IMPLICAÇÕES

O SonarAutoFix destaca o potencial de LLMs comerciais na automação de refatoração, especialmente para *code smells* estruturais e *bugs* detectáveis estaticamente. Durante o processo de substituição, não houve nenhum erro de estrutura de página decorrente dos códigos gerados pelo Claude, evidenciando a robustez do modelo na preservação da integridade estrutural do código. A variação de desempenho entre repositórios sugere que limitações contextuais podem ser mitigadas com ajustes, como expansão para outras linguagens ou integração de aprendizado contínuo. Comparado a soluções especializadas, o SonarAutoFix mostra eficácia promissora, com espaço para refinamentos em engenharia de *prompts* ou estratégias de otimização. Contudo, uma das limitações pode ser a introdução de erros dinâmicos, visto que a lógica do código pode ter sido alterada, o que poderia levar à quebra de testes unitários ou de integração. Ainda assim, esses aspectos consolidam seu valor na redução do esforço de manutenção de software.

7 CONCLUSÃO

Este trabalho introduziu o SonarAutoFix, uma solução inovadora para a automação da correção de código, impulsionada pela necessidade de otimizar os processos de desenvolvimento e manutenção de software em um contexto de crescente complexidade de projetos. A proposta consistiu em um sistema estruturado em cinco módulos interdependentes — execução, análise estática, geração de correções, aplicação de correções e validação —, integrando ferramentas consolidadas como SonarQube e LLMs comerciais, como o Claude AI. Os testes em 11 repositórios de código aberto revelaram uma taxa de correção geral de 64,33%, com desempenhos destacados na resolução de *code smells* estruturais (e.g., 90,05% em *redux*) e *bugs* funcionais (e.g., 82,61% em *redux*), processando 8.581 *issues* a um custo operacional de apenas 50 dólares. Esses resultados sublinham a capacidade da ferramenta de reduzir significativamente o esforço manual dos desenvolvedores, elevar a qualidade do código e oferecer uma abordagem economicamente viável para a refatoração automatizada, com potencial de integração em fluxos de trabalho reais.

Embora os avanços sejam promissores, o estudo identificou limitações que orientam os próximos passos da pesquisa. A variação de eficácia entre repositórios, como a taxa de apenas 28,63% em *axios*, aponta para dificuldades em lidar com *issues* mais contextuais ou dependentes de especificidades do projeto, enquanto o foco exclusivo em JavaScript restringe a aplicabilidade a outros paradigmas de programação. Adicionalmente, vulnerabilidades de segurança complexas, como injeções de SQL, não foram completamente contempladas na abordagem atual. Para superar esses desafios, trabalhos futuros incluirão a expansão do SonarAutoFix para suportar outras linguagens de programação, o aprimoramento da geração de correções por meio de técnicas como *fine-tuning* de LLMs e a incorporação de módulos específicos para identificar e corrigir vulnerabilidades de segurança, visando tornar a ferramenta mais abrangente e robusta para atender às demandas diversificadas do desenvolvimento de software moderno.

REFERÊNCIAS

ALLAMANIS, M.; BARR, E. T.; DEVANBU, P.; SUTTON, C. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, v. 51, n. 4, p. 81:1–81:37, 2018.

Axios. *Axios - Promise based HTTP client for the browser and Node.js*. 2025. Disponível em: <https://axios-http.com/>.

BELLER, M.; BHOLANATH, R.; MCINTOSH, S.; ZAIDMAN, A. Analyzing the state of static analysis in open-source software. *IEEE Transactions on Software Engineering*, v. 44, n. 10, p. 949–966, 2018.

BENDER, E. M.; GEBRU, T.; MCMILLAN-MAJOR, A.; SHMITCHELL, S. On the dangers of stochastic parrots: Can language models be too big? *Proceedings of the 2021 ACM Conference on Fairness, Accountability, and Transparency*, p. 610–623, 2021.

BERABI, B.; GRONSKIY, A.; RAYCHEV, V.; SIVANRUPAN, G.; CHIBOTARU, V.; VECHEV, M. *DeepCode AI Fix: Fixing Security Vulnerabilities with Large Language Models*. 2024. Disponível em: <https://arxiv.org/abs/2402.13291>.

BOEHM, B. W.; BASILI, V. R. *Software Defect Reduction Top 10 List*. Los Alamitos: IEEE Computer Society, 2001.

BOMMASANI, R.; HUDSON, D. A.; ADELI, E.; ALTMAN, R.; ARORA, S.; ARX, S. von; BERNSTEIN, M. S.; BOHG, J.; BOSSELUT, A. et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J.; DHARIWAL, P.; NEELAKANTAN, A.; SHYAM, P.; SASTRY, G.; ASKELL, A. et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, v. 33, p. 1877–1901, 2020.

CHEN, Y.; LIU, Z.; WANG, H. Exploring effective communication strategies with large language models. *Journal of Artificial Intelligence Research*, v. 76, p. 123–145, 2023.

CHESS, B.; WEST, J. *Secure Programming with Static Analysis*. Boston: Addison-Wesley, 2007.

CHOW, Y. W.; GRAZIA, L. D.; PRADEL, M. Pyty: Repairing static type errors in python. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ACM, 2024. (ICSE '24), p. 1–13. Disponível em: <http://dx.doi.org/10.1145/3597503.3639184>.

Date-fns. *date-fns - Modern JavaScript Date Utility Library*. 2025. Disponível em: <https://date-fns.org/>.

DEVLIN, J.; CHANG, M.-W.; LEE, K.; TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, p. 4171–4186, 2019.

ESLint. *ESLint: Pluggable JavaScript Linting Utility*. 2025. Disponível em: <https://eslint.org>.

ETEMADI, K.; HARRAND, N.; LARSéN, S.; ADZEMOVIC, H.; PHU, H. L.; VERMA, A.; MADEIRAL, F.; WIKSTRöM, D.; MONPERRUS, M. Sorald: Automatic patch suggestions for sonarqube static analysis violations. *IEEE Transactions on Dependable and Secure Computing*, Institute of Electrical and Electronics Engineers (IEEE), v. 20, n. 4, p. 2794–2810, jul. 2023. ISSN 2160-9209. Disponível em: <http://dx.doi.org/10.1109/TDSC.2022.3167316>.

Express. *Express - Fast, unopinionated, minimalist web framework for Node.js*. 2025. Disponível em: <https://expressjs.com/>.

FERRAG, M. A.; BATTAH, A.; TIHANYI, N.; JAIN, R.; MAIMUT, D.; ALWAHEDI, F.; LESTABLE, T.; THANDI, N. S.; MECHRI, A.; DEBBAH, M.; CORDEIRO, L. C. *SecureFalcon: Are We There Yet in Automated Software Vulnerability Detection with LLMs?* 2025. Disponível em: <https://arxiv.org/abs/2307.06616>.

Google. *Angular - The web development framework for building modern apps*. 2025. Acessado em 19 de março de 2025. Disponível em: <https://angular.dev>.

HOSSAIN, S. B.; DU, N.; RAJBHANDARI, A. et al. A deep dive into large language models for automated bug localization and repair. *Proceedings of the ACM on Software Engineering*, v. 1, p. 1531–1554, 2024.

JAIN, N.; GANDHI, S.; SONWANE, A.; KANADE, A.; NATARAJAN, N.; PARTHASARATHY, S.; RAJAMANI, S.; SHARMA, R. *StaticFixer: From Static Analysis to Static Repair*. 2023. Disponível em: <https://arxiv.org/abs/2307.12465>.

Jest. *Jest - Delightful JavaScript Testing*. 2025. Disponível em: <https://jestjs.io/>.

JOHNSON, B.; SONG, Y.; CHOI, H.; MURPHY-HILL, E. Roles and impacts of static analysis tools in software development. *Proceedings of the 2013 ACM Conference on Empirical Methods in Software Engineering*, p. 23–32, 2013.

KANG, S.; YOON, J.; YOO, S. Large language models are few-shot testers: Exploring llm-based general bug reproduction. *arXiv preprint arXiv:2209.11515*, 2023. Disponível em: https://arxiv.org/abs/2209.11515.

LI, J.; CHEN, X.; HUANG, R. Enhancing interpretability in large language models through structured reasoning. *Computational Linguistics*, v. 49, n. 2, p. 301–325, 2023.

MARCILIO, D.; FURIA, C. A.; BONIFÁCIO, R.; PINTO, G. Spongebugs: Automatically generating fix suggestions in response to static code analysis warnings. *J. Syst. Softw.*, v. 168, p. 110671, 2020. Disponível em: <https://doi.org/10.1016/j.jss.2020.110671>.

MASSE, M. *REST API Design Rulebook*. Sebastopol: O'Reilly Media, 2011.

Meta. *React - The library for web and native user interfaces*. 2025. Acessado em 19 de março de 2025. Disponível em: <https://react.dev>.

MONPERRUS, M. Automatic software repair: A bibliography. *ACM Computing Surveys*, v. 51, n. 1, p. 17:1–17:24, 2019.

NestJS. *NestJS - A progressive Node.js framework*. 2025. Disponível em: <https://nestjs.com/>.

NYE, M.; ANDREASSEN, A.; GUR-ARI, G. Improving reasoning in language models with chain-of-thought prompting. *arXiv preprint arXiv:2301.04567*, 2023. Disponível em: <https://arxiv.org/abs/2301.04567>.

Pino. *Pino - Extremely fast node.js logger*. 2025. Disponível em: <https://getpino.io/>.

PMD. *PMD: An Extensible Cross-Language Static Code Analyzer*. 2025. Disponível em: <https://pmd.github.io>.

Prisma. *Prisma - Next-generation Node.js and TypeScript ORM*. 2025. Disponível em: <https://www.prisma.io/>.

RADFORD, A.; WU, J.; CHILD, R.; LUAN, D.; AMODEI, D.; SUTSKEVER, I. Language models are unsupervised multitask learners. *OpenAI Blog*, 2019. Disponível em: <https://openai.com/blog/better-language-models/>.

Redux. *Redux - A Predictable State Container for JavaScript Apps*. 2025. Disponível em: <https://redux.js.org/>.

RxJS. *RxJS - Reactive Extensions for JavaScript*. 2025. Disponível em: <https://rxjs.dev/>.

SERBAN, D.; GOLSTEIJN, B.; HOLDORP, R.; SEREBRENIK, A. Saw-bot: Proposing fixes for static analysis warnings with github suggestions. In: *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)*. [S.l.: s.n.], 2021. p. 26–30.

SILVA, A.; FANG, S.; MONPERRUS, M. *RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair*. 2024. Disponível em: <https://arxiv.org/abs/2312.15698>.

SonarQube. *SonarQube: Static Code Analysis*. 2025. Disponível em: <https://www.sonarsource.com/products/sonarqube/>.

SPINELLIS, D. Version control systems. *IEEE Software*, v. 22, n. 5, p. 108–109, 2005.

SUN, H.; ZHOU, L.; LI, X. Prompt engineering for precision: Enhancing large language model outputs. In: *Proceedings of the 2023 International Conference on Machine Learning*. [S.l.: s.n.], 2023. p. 567–579.

TOUVRON, H.; LAVRIL, T.; IZACARD, G.; MARTINET, X.; LACHAUX, M.-A.; LACROIX, T.; ROZIèRE, B.; GOYAL, N.; HAMBRO, E.; AZHAR, F. et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

tRPC. *tRPC - End-to-end typesafe APIs made easy*. 2025. Disponível em: <https://trpc.io/>.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, ; POLOSUKHIN, I. Attention is all you need. *Advances in Neural Information Processing Systems*, v. 30, 2017.

WANG, Z.; LI, Y.; ZHANG, Q. Reasoning step-by-step: Advances in chain-of-thought techniques. *Advances in Neural Information Processing Systems*, v. 36, 2023.

ZHANG, X.; LIU, W.; CHEN, P. et al. A survey on large language models: Applications and challenges. *arXiv preprint arXiv:2303.12345*, 2023. Disponível em: <https://arxiv.org/abs/2303.12345>.

ZHAO, T.; WALLACE, E.; FENG, S. Calibrating prompts for better performance in large language models. *arXiv preprint arXiv:2210.09876*, 2022. Disponível em: <https://arxiv.org/abs/2210.09876>.

Zod. *Zod - TypeScript-first schema validation with static type inference*. 2025. Disponível em: <https://zod.dev/>.