Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Monografia em Sistemas de Informação II

Avaliação da Efetividade de LLMs ao Resolver Code Smells Relacionados à Manutenibilidade de Software

Pesquisa Científica

por Caio Simões da Silva Ferreira

Orientador: Eduardo Figueiredo

Belo Horizonte, MG
27 de junho de 2025

Resumo

A crescente complexidade dos sistemas de software modernos torna a manutenção uma tarefa desafiadora e crítica para garantir a longevidade e adaptabilidade desses sistemas. Code smells, que são indicadores de potenciais problemas no código-fonte, podem comprometer a manutenibilidade de software ao dificultar modificações, correções e melhorias. A identificação e correção desses code smells são essenciais para reduzir custos operacionais e melhorar a eficiência do desenvolvimento. No entanto, as abordagens tradicionais de análise e refatoração de código podem ser demoradas. Com o avanço dos Modelos de Linguagem de Grande Escala (LLMs), surge a oportunidade de automatizar e potencialmente aprimorar a detecção e correção de code smells, oferecendo uma solução promissora para esse campo da Engenharia de Software.

Este estudo tem como objetivo explorar a eficácia dos Modelos de Linguagem de Grande Escala (LLMs) na resolução de code smells relacionados à manutenibilidade de software, utilizando a LLM DeepSeek. A pesquisa propõe um método que envolve a seleção de projetos .NET no GitHub, análise estática com SonarQube, e a aplicação de prompts no DeepSeek para correção dos bad smells detectados. Os resultados esperados incluem uma discussão sobre a eficácia do DeepSeek na resolução de problemas de manutenibilidade, além da análise de quais tipos de erro a LLM mais comete ao realizar esse tipo de tarefa.

1 Introdução

A qualidade das soluções de software tem se tornado cada vez mais crucial com a integração dessas soluções em diversos aspectos da vida moderna. Nunca foi tão importante garantir que os sistemas funcionem de maneira eficiente, segura e confiável. Um dos aspectos fundamentais da qualidade de software é a manutenibilidade [30] [26], que se refere à facilidade com que um sistema ou componente de software pode ser modificado para corrigir falhas, melhorar o desempenho e outros atributos, ou adaptar-se a um ambiente modificado [26]. A manutenibilidade não apenas contribui para a longevidade do software, mas também para sua adaptabilidade às mudanças tecnológicas e de requisitos, reduzindo custos operacionais a longo prazo e melhorando a eficiência geral do desenvolvimento.

Para a garantia da manutenibilidade, uma das soluções é a utilização de ferramentas de análise estática de código. Elas inspecionam o código em busca de problemas como erro de sintaxe, vulnerabilidades, violações de padrões, dentre outros [32]. Para resolução desses problemas de forma

rápida, os desenvolvedores podem recorrer aos Modelos de Linguagem de Grande Escala (LLMs), que têm se destacado em tarefas como a geração de código [12]. No entanto, devido à natureza emergente dessa tecnologia, pouco se sabe sobre a sua real eficácia nesse contexto. Por isso, este projeto teve como objetivo principal avaliar a eficiência da LLM DeepSeek na resolução de code smells relacionados à manutenibilidade de software detectados pelo SonarQube.

2 Trabalhos Relacionados

O artigo de Nunes et al. [23] investiga a eficácia dos Modelos de Linguagem de Grande Escala (LLMs) na resolução de problemas de manutenibilidade de código em projetos de software do mundo real. Mais especificamente, o estudo foca em avaliar se o Copilot Chat, LLM da Microsoft, é capaz de corrigir problemas de manutenibilidade (code smells) detectados em projetos Java pela ferramenta de análise estática de código SonarQube, sem introduzir novos erros. Um outro trabalho [31] avalia a eficiência das LLMs GPT-4 e Llma 3.1 na refatoração de código escrito em Python para descobrir sua aptidão para corrigir problemas de manutenibilidade, identificar limitações e propor melhorias. A diferença principal deste projeto é que iremos utilizar a LLM DeepSeek, para correção de code smells detectados pelo SonarQube, além de trabalhar com projetos .NET ao invés de projetos feitos em Java ou Python.

3 Referencial Teórico

3.1 Qualidade de Software

Segundo Pressman [26], qualidade de software se refere a um processo de desenvolvimento de software bem-sucedido que resulta na criação de um produto valioso, beneficiando tanto os desenvolvedores, quanto os usuários finais. Tal processo deve especificar uma infraestrutura capaz de suportar quaisquer esforços necessários para a construção de um software de alta qualidade, e o software construído deve atender aos requisitos especificados pelos stakeholders de forma robusta e sem falhas. Ao beneficiar ambos os lados, o desenvolvedor se ocupa menos com manutenção e mais com a criação de novas funcionalidades, enquanto o usuário final desfruta de um produto que atende às suas expectativas.

3.1.1 Fatores de qualidade de McCall

McCall, Richards e Walters [20] definem fatores de qualidade de software e os organizam em três categorias principais: operação, revisão e transição. A categoria de operação abrange os atributos do software que impactam diretamente sua execução e o desempenho percebido pelos usuários finais, assegurando que o software funcione corretamente, com confiabilidade e eficiência durante seu uso. A categoria de revisão concentra-se nos aspectos que facilitam a manutenção e evolução do software ao longo do tempo, sendo essenciais para garantir que ele possa ser atualizado e aprimorado de maneira eficaz. Por último, a categoria de transição trata da capacidade do software de se adaptar a diferentes ambientes e contextos de uso, garantindo sua reutilização e integração em diversos sistemas.

3.1.2 Fatores de qualidade ISO25010

A ISO/IEC 25010 [16] é uma norma internacional que faz parte da série SQuaRE (*Software Product Quality Requirements and Evaluation*), desenvolvida para substituir e expandir a ISO/IEC 9126. Publicada pela *International Organization for Standardization* (ISO) e pela *International Electrotechnical Commission* (IEC), a ISO/IEC 25010 fornece um modelo para avaliar a qualidade de produtos de software, refletindo as necessidades modernas do desenvolvimento de software. A norma define dois modelos principais: o modelo de qualidade do produto e o modelo de qualidade em uso. O modelo de qualidade do produto foca nas características internas e externas do software, que são fundamentais para garantir que ele atenda aos requisitos funcionais e não funcionais. Já o modelo de qualidade em uso avalia a qualidade do software do ponto de vista do usuário final durante o uso real, considerando o impacto direto na experiência do usuário.

3.2 Code Smells

Segundo Fowler [13], code smells são sintomas no código-fonte que podem indicar problemas mais profundos de design, sugerindo a necessidade de refatoração, mesmo quando o código funciona corretamente. Embora um code smell não seja necessariamente um erro, ele frequentemente aponta para uma fragilidade no design que pode tornar o código mais difícil de manter, entender e evoluir no futuro. Além disso, Fowler [13] destaca que os code smells são indicadores, não regras absolutas, e que a decisão de refatorar deve ser baseada no contexto e no impacto do code smell no projeto. Os principais bad smells definidos por Fowler [13] são:

- *Duplicated code*: se trata da repetição do mesmo trecho de código. Para resolver, em geral, deve-se implementar um método cuja chamada substitua tais trechos. No entanto, em casos onde há código duplicado em classes que não se relacionam, deve-se considerar a implementação de uma nova classe;

- *Long method*: se trata de um método que é muito longo e que pode ser difícil de entender e manter. Na maior parte dos casos, a solução se encontra na divisão do método atual em outros métodos menores. No entanto, se tratando de um método com um grande número de parâmetros, também deve-se considerar a implementação de um objeto que agrupe os parâmetros relacionados;

- *Large class*: se trata de uma classe que tenta fazer muito e torna-se difícil de compreender e modificar. A solução se encontra na implementação de uma nova classe que contenha métodos e atributos que se relacionam;

- *Feature envy*: ocorre quando um método em uma classe parece estar mais interessado em outra classe do que na própria, acessando muitos de seus dados ou métodos. Para resolve esse code smell, basta mover o método para a classe mais relacionada;

- *Data clumps*: se trata de conjuntos de variáveis que sempre aparecem juntos. Isso indica que tais variáveis podem ser encapsuladas em sua própria classe;

- *Refused bequest*: ocorre quando a classe filha nunca utiliza um ou mais métodos herdados da sua classe pai. Isso pode indicar um problema no design da hierarquia de classes, onde a subclasse está forçada a assumir responsabilidades que não são relevantes para ela. A resolução varia entre a redefinição dos métodos herdados e a criação de uma subclasse que contenha os métodos não usados;

- *Speculative generality*: se trata de código escrito para acomodar futuras necessidades que nunca se materializam, resultando em complexidade desnecessária. Deve-se considerar a sua remoção;

- *Temporary field*: se trata de atributos que só são usados em certas circunstâncias, tornando o estado da classe difícil de entender. A solução se baseia na implementação de uma classe que contenha tais atributos;

- *Shotgun surgery*: ocorre quando uma pequena mudança requer modificações em muitas classes diferentes, indicando que a funcionalidade

está espalhada pelo sistema. É solucionado com a transferência de métodos e atributos de forma que as modificações sejam feitas somente numa classe;

- *Lazy class*: se trata de uma classe que não faz o suficiente para justificar sua existência. Deve-se considerar fundi-la com outra classe relacionada;

- *Divergent change*: ocorre quando uma única classe é frequentemente modificada por diferentes razões ou para acomodar alterações em múltiplas funcionalidades. A solução se baseia na implementação de classes que agrupem métodos e atributos relacionados a uma mesma responsabilidade.

3.3 Métricas de Software

Segundo Sommerville [30], métricas de produto são métricas de previsão que têm o objetivo de medir atributos internos de software. Elas são divididas em métricas dinâmicas e métricas estáticas. Métricas dinâmicas são coletadas durante a execução do programa e ajudam a avaliar a eficiência e a confiabilidade de um sistema. Por exemplo, é possível medir a taxa de utilização da CPU e o consumo de memória durante a execução, o que se relaciona diretamente com a eficiência do sistema. Além disso, a frequência de erros de execução e o tempo médio entre falhas podem ser monitorados, fornecendo uma relação direta com a confiabilidade do software.

Por outro lado, métricas estáticas são obtidas através de medições realizadas no código-fonte ou na documentação. As métricas estáticas de produto discutidas por Sommerville [30] são:

- *Fan-in/Fan-out*: *fan-in* refere-se ao número de funções ou módulos que chamam uma determinada função ou módulo. Um alto *fan-in* pode indicar que a função é amplamente reutilizada, mas também pode sugerir um ponto crítico que, se alterado, pode impactar muitas partes do sistema . Já o *fan-out* é o número de funções ou módulos chamados por uma determinada função ou módulo. Um alto *fan-out* pode indicar complexidade, pois a função depende de muitos outros componentes;

- *Lines of code* (LOC): mede o tamanho do código-fonte, geralmente em termos de linhas de código. Essa métrica fornece uma indicação básica do tamanho do projeto e pode influenciar outras métricas, como esforço de desenvolvimento e manutenção;

- *Cyclomatic complexity*: mede o número de caminhos lineares independentes através de um programa. É calculada com base no grafo de controle de fluxo do programa e ajuda a identificar áreas do código que podem ser difíceis de testar ou manter;

- *Identifier length*: refere-se ao tamanho dos nomes usados para variáveis, funções, classes, etc. Identificadores mais longos estão associados a uma melhor legibilidade e a compreensibilidade do código, enquanto identificadores muito curtos podem dificultar a compreensão;

- *Depth of conditional nesting*: mede o nível de aninhamento de estruturas condicionais (como *if-else*) dentro do código. Uma profundidade de aninhamento excessiva pode tornar o código difícil de entender e seguir;

- *Fog index*: avalia o quão difícil é a compreensão de um documento por meio do comprimento médio de suas palavras e sentenças.

Chidamber e Kemerer [4] propõem um conjunto de métricas estáticas para programas orientados a objetos. São elas:

- *Weighted methods per class* (WMC): calcula a soma das complexidades dos métodos de uma classe. A complexidade pode ser medida de várias maneiras, como pela complexidade ciclomática. Um WMC alto pode indicar que a classe é complexa e difícil de entender, testar e manter;

- *Depth of inheritance tree* (DIT): mede a profundidade máxima da hierarquia de herança de uma classe, ou seja, o número de níveis de herança até a raiz da árvore. Um DIT maior pode indicar maior reutilização de código, mas também pode aumentar a complexidade devido à dificuldade de prever o comportamento da classe;

- *Number of children* (NOC): conta o número de subclasses diretamente derivadas de uma classe. Um NOC alto pode indicar uma classe que é altamente reutilizada, mas também pode sugerir que mudanças na classe base terão um impacto significativo nas subclasses;

- *Coupling between object classes* (CBO): mede o grau de acoplamento entre classes, contando o número de outras classes às quais uma classe está acoplada. Um CBO alto pode indicar dependências excessivas, tornando o sistema mais difícil de modificar e manter;

- *Response for a class* (RFC): calcula o número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto de

uma classe. Isso inclui métodos internos e aqueles chamados por outros métodos. Um RFC alto pode indicar complexidade, pois a classe tem muitos caminhos de execução possíveis;

- *Lack of cohesion in methods* (LCOM): mede a falta de coesão entre os métodos de uma classe, avaliando quantos métodos acessam diferentes subconjuntos de variáveis de instância. Uma LCOM alta sugere que a classe realiza muitas tarefas não relacionadas e pode se beneficiar de uma divisão em classes menores e mais focadas.

3.4 Ferramentas de Análise Estática

A análise de código estática é uma técnica fundamental no processo de desenvolvimento de software, utilizada para examinar o código-fonte sem a necessidade de executá-lo. Essa abordagem permite identificar potenciais problemas, como vulnerabilidades de segurança e bad smells, que podem comprometer a manutenibilidade e a qualidade geral do software [32]. A seguir, são discutidas algumas das principais ferramentas de análise de código estática.

3.4.1 SonarQube

O SonarQube [9] é uma plataforma de análise de código estática que oferece suporte a uma ampla variedade de linguagens de programação. Ele se integra aos *pipelines* de Integração Contínua (CI) e plataformas *DevOps*, permitindo a verificação automatizada do código em relação a um conjunto abrangente de regras. Essas regras cobrem diversos aspectos da qualidade do código, incluindo manutenibilidade, confiabilidade e segurança.

A solução Sonar visa identificar qualquer problema no código que o impeça de ser considerado *Clean Code*. Para isso, cada atributo do *Clean Code* é avaliado para uma determinada linguagem com base em uma série de regras. Cada regra possui as seguintes características:

- Associação com o atributo do *Clean Code* que ela avalia: Por exemplo, uma regra pode estar associada à facilidade de manutenção do código;
- Associação com a qualidade de software à qual esse atributo do *Clean Code* contribui: As qualidades de software consideradas são segurança, confiabilidade e manutenibilidade;
- Nível de severidade atribuído (crítico, alto, médio, baixo ou informativo): Essa severidade determina o quanto a qualidade do software é impactada quando a regra é violada.

Quando uma regra é violada, um problema de código é levantado. Esse problema de código afeta uma ou mais qualidades de software com diferentes níveis de severidade, herdados da regra associada.

3.4.2 PMD

PMD [11] é uma ferramenta *open source* de análise estática de código voltada principalmente para Java, mas que também suporta outras linguagens, como JavaScript e XML. Ela identifica uma variedade de problemas, desde más práticas de codificação até possíveis bugs e code smells. Para cada linguagem suportada, o PMD possui um conjunto de regras que pode ser estendido pelo usuário. Para Java, por exemplo, a ferramenta possui os conjuntos:

- *Best practices*: regras geralmente aceitas como boas práticas de programação Java;
- *Code style*: regras que impõe um estilo específico de codificação;
- Design: regras que auxiliam a descoberta de problemas de design;
- *Documentation*: regras relacionadas à documentação do código;
- *Error prone*: regras para detectar código que está muito confuso ou propenso a erros em tempo de execução;
- *Multithreading*: regras que apontam problemas ao lidar com várias *threads* de execução;
- Performance: regras que sinalizam código não eficiente;
- *Security*: regras que evidenciam falhas de segurança potenciais.

3.4.3 Checkstyle

Checkstyle [7] é outra ferramenta popular para projetos Java, focada em garantir que o código siga padrões de estilo predefinidos. Embora seu foco principal seja a conformidade com convenções de codificação, ela também é capaz de identificar os code smells *large class*, *long method*, *long parameter list* e *duplicated code*.

3.4.4 FindBugs

SpotBugs [10], antiga FindBugs, é uma ferramenta *open source* que utiliza técnicas de análise de *bytecode* na detecção de bugs em programas Java. A ferramenta divide os bugs detectados nas seguintes categorias:

- *Bad practices*: violação de boas práticas de programação;
- *Correctness*: situações em que o comportamento do código provavelmente não é o desejado;
- Experimental: bugs que não foram totalmente verificados;
- *Internationalization*: problemas de código relacionados a internacionalização e localidade;
- *Malicius code vulnerability*: código vulnerável a ataques;
- *Multithread correctness*: problemas de códigos relacionados a *threads* e *locks*;
- Performance: código que pode ser ineficiente;
- *Security*: código que possui uma entrada que pode ser explorada em ataques;
- *Dodgy code*: código confuso ou escrito de uma forma que leva a erros.

3.4.5 JDeodorant

JDeodorant [8] é um plugin *open source* do Eclipse capaz de identificar e propor refatorações para os code smells *feature envy*, *type checking*, *long method*, *god class* e *duplicated code*. A ferramenta é produto de pesquisas desenvolvidas no *Software Refactoring Lab*, na Universidade da Concordia e *Software Engineering Group*, na Universidade da Macedônia.

3.5 Modelos de Linguagem de Grande Escala (LLMs)

Desde o lançamento do ChatGPT em novembro de 2022, os Modelos de Linguagem de Grande Escala (LLMs) ganharam significativa popularidade, devido à sua capacidade de retornar resultados a diversos tipos de prompts, que poderiam ser escritos por qualquer pessoa. Os LLMs são modelos de rede neural treinados com uma quantidade massiva de dados, como afirmam Zhao et al. [38]; e são projetados para aprender padrões estatísticos e relações

semânticas inerentes à linguagem humana, permitindo-lhes realizar uma ampla gama de tarefas de Processamento de Linguagem Natural (NLP), como a geração e a sumarização de código, tradução de idiomas e tomada de decisão [12].

3.5.1 GPT

O GPT (*Generative Pre-trained Transformer*) é um modelo de linguagem desenvolvido pela OpenAI, baseado na arquitetura *Transformer* [34], uma rede neural com capacidade de processamento de dados sequenciais, como séries temporais ou texto. O GPT-3 é considerado um marco importante na evolução de PLMs (*Pre-trained Language Models*) para LLMs por ter mostrado que um grande aumento da capacidade do modelo pode ser alcançado ao escalar as redes neurais para um tamanho significativo. Tal modelo, pré-treinado apenas com texto simples, apresentava limitações na realização de tarefas complexas como completação de código e resolução de problemas matemáticos. No entanto, a inclusão de dados de treinamento contendo código no desenvolvimento do GPT-3.5 resultou em melhorias significativas na capacidade de raciocínio do modelo. O GPT-4, lançado em março de 2023 supera o GPT-3.5 em termos de resolução de tarefas complexas, mostrando um desempenho aprimorado em várias avaliações. Além disso, a inclusão de um componente adicional de segurança no treinamento por reforço com feedback humano (RLHF) aumentou a eficácia do GPT-4 ao responder de forma segura a consultas maliciosas ou provocativas [38]. O GPT-4 Omini, o modelo mais recente lançado em maio de 2024, introduziu uma série de inovações significativas que aprimoram os LLMs anteriores. Este modelo conta com um número impressionante de parâmetros, estimado em mais de um trilhão, superando o GPT-3, que possui 175 bilhões de parâmetros. O GPT-4 Omini possui protocolos éticos e de segurança aprimorados, abordando eficazmente a questão das saídas prejudiciais ou incorretas e oferece um tratamento mais avançado para consultas ambíguas e complexas, reduzindo potenciais mal-entendidos entre o usuário e o modelo [29]. Uma das principais aplicações dos modelos GPT é o modelo de conversação desenvolvido pela OpenAI, o ChatGPT, que possui a capacidade de interagir de forma conversacional com os usuários, respondendo a prompts de forma detalhada [38] [15].

3.5.2 Claude

O Claude é um modelo de linguagem criado pela Anthropic, lançado nos EUA em março de 2023 e no Brasil em agosto de 2024. Dentre as suas capacidades estão a geração de código, transcrição e análise de imagem, geração

de texto e reconhecimento de padrões, e tradução de idiomas em tempo real. A "família" Claude 3 é composta por três modelos: o Haiku, o mais rápido e de menor custo; o Sonnet, que oferece a melhor combinação de performance e velocidade em tarefas de alta vazão; e o Opus, o mais dispendioso, que oferece a melhor performance e se destaca em tarefas complexas [5]. Sua arquitetura é baseada nos *transformers* de [34], trazendo melhorias, como por exemplo, um mecanismo de atenção escalável que mantém a eficiência mesmo para entradas muito extensas, que são comuns em tarefas como escrita de artigos e análise e documentos complexos [6]. O Claude-3.5 Sonnet, lançado em 20 de junho de 2024, apresentou melhorias significativas na velocidade operacional e na eficiência de custos em comparação com o Claude-3, operando a uma velocidade duas vezes maior e com custos inferiores ao Claude-3 Opus. Conforme detalhado pela Anthropic, o modelo foi treinado com dados até abril de 2024 e supera o GPT-4o e o Gemini 1.5 Pro em diversos aspectos. Destaca-se que, no LiveBench, um *benchmark* para LLMs, o Claude-3.5 Sonnet foi considerado o modelo de melhor desempenho e demonstrou uma performance em codificação aproximadamente 25% superior ao do GPT-4o [3].

3.5.3 Gemini

O Gemini AI, lançado em março de 2023 pela Google para concorrer com a dominância do ChatGPT no mercado de inteligências artificiais generativas [25], apresenta uma série de inovações significativas. A primeira versão, Gemini 1.0, está disponível em três configurações: Ultra, Pro e Nano, cada uma adaptada para diferentes necessidades computacionais e de aplicação. O modelo Ultra apresenta ótimo desempenho em atividades de alta complexidade; o Pro oferece um bom desempenho numa grande variedade de tarefas, equilibrado com custo e latência; enquanto o Nano é ideal para dispositivos móveis e se destaca pela capacidade de rodar em dispositivos com diferentes capacidades de memória [33]. Recentemente, a Google lançou os modelos Gemini 1.5 Flash e Gemini 1.5 Pro, que aceitam entradas de imagens, áudio e vídeo, além de texto. O Gemini 1.5 Flash se destaca pela baixa latência, sendo ideal para aplicações que requerem respostas rápidas, enquanto a nova versão do Pro entrega qualidade comparável ao Ultra usando menos recursos computacionais [22] [14].

A arquitetura do Gemini, construída sobre decodificadores *Transformer* [34] aprimorados, foi desenvolvida para processar uma ampla variedade de dados, incluindo texto, imagens, áudio e vídeo. Isso é possível graças a um codificador multimodal exclusivo, juntamente com uma rede de atenção e um decodificador multimodais. Essa estrutura permite que o Gemini supere as

limitações dos sistemas de codificação única, associando conceitos textuais a regiões de imagem e alcançando uma compreensão composicional de cenas. Outro fator de destaque do Gemini é seu foco na explicação do resultado: ao fornecer justificativas das respostas, promove-se uma maior compreensão do processo de raciocínio da IA e, consequentemente, há um aumento da confiança do usuário [21].

3.5.4 DeepSeek

O DeepSeek é uma LLM desenvolvida pela empresa chinesa DeepSeek com o objetivo de competir diretamente com os modelos de raciocínio da OpenAI. Lançado em janeiro de 2025, o modelo gerou um impacto significativo na comunidade global de inteligência artificial por ser um modelo de código aberto e apresentar custos de desenvolvimento reduzidos. O DeepSeek-V3 adota a arquitetura *Mixture-of-Experts* (MoE), que ativa apenas uma fração dos seus parâmetros durante o processo de inferência. Essa abordagem proporciona maior eficiência no uso de recursos computacionais e reduz significativamente os custos relacionados ao treinamento [27] [35]. Além disso, outras técnicas inovadoras, como *Multi-Head Latent Attention*, *Multi-Token Prediction*, *DualPipe* e *FP8 Mixed Precision Training*, foram fundamentais para alcançar o sucesso e a eficiência de custo desse modelo [35] [2].

Enquanto o DeepSeek-V3 foi projetado como um modelo de propósito geral, cobrindo uma ampla gama de domínios de conhecimento, o DeepSeek-R1 foi criado para tarefas mais complexas, como a solução de problemas matemáticos e de raciocínio lógico [17]. Diferentemente das LLMs tradicionais, que geralmente produzem respostas não estruturadas, o DeepSeek-R1 oferece um raciocínio sequencial que simula etapas cognitivas humanas, como auto-verificação e reflexão. Para isso, o DeepSeek-R1 emprega técnicas modernas de treinamento, como o *supervised fine-tuning* (SFT) e o *reinforcement learning from verifiable rewards* (RLVR). Essas técnicas permitiram que o modelo desenvolvesse comportamentos sofisticados de raciocínio utilizando uma quantidade relativamente modesta de dados supervisionados [36].

3.5.5 Aplicações de modelos de linguagem de grande escala na Engenharia de Software

Zheng et al. [39] afirmam que a busca por maneiras de se promover um aumento no nível de automação em tarefas da engenharia de software para se alcançar um desenvolvimento e manutenção de software mais eficientes tem sido um tópico amplamente explorado, em especial com a crescente exploração do potencial das LLMs. Os seguintes campos se destacam:

- Engenharia de requisitos: os LLMs têm mostrado um potencial considerável na automação e aprimoramento de tarefas relacionadas à engenharia de requisitos, como obtenção, classificação, geração, elaboração de especificações e avaliação de qualidade [18]. Uma pesquisa revelou que os LLMs são promissores na geração de requisitos. Por exemplo, ao utilizar o ChatGPT para gerar e coletar requisitos dos usuários, observou-se que participantes com experiência profissional conseguem usar o ChatGPT de maneira mais eficaz, destacando a influência do conhecimento do domínio na eficácia da elicitação de requisitos assistida por LLM. Além disso, um outro estudo demonstrou que o uso de LLMs ajustados pode ser benéfico na geração de Especificações de Requisitos de Software (SRS), aumentando a produtividade geral do projeto [18].

- Geração de código e desenvolvimento de software: recentemente, a aplicação de LLMs na geração de código e no desenvolvimento de software avançou de forma considerável, transformando a forma como os desenvolvedores trabalham e promovendo uma mudança nos processos automatizados de desenvolvimento. Os LLMs atuais exibem capacidades notáveis de geração de código, alcançando satisfatórias taxas de precisão e compilação, com intervenção humana adequada [37]. A eficácia dos LLMs na geração de código é atribuída ao pré-treinamento em grandes volumes de dados textuais, especialmente corpus extensos de código, embora o fenômeno de "alucinação"possa, às vezes, comprometer a qualidade do código gerado [19]. Deve-se ressaltar que a qualidade do código gerado está diretamente relacionada aos prompts fornecidos pelos usuários: prompts vagos ou gerais podem dificultar a compreensão dos requisitos reais pelo LLM, prejudicando a geração precisa do código em uma única tentativa [18].

- Geração de testes de software: a aplicação de inteligência artificial na geração automática de casos de teste tem sido objeto de pesquisa desde meados dos anos 2000. Inicialmente, técnicas mais simples de IA e aprendizado de máquina foram utilizadas para automatizar partes desse processo. Com o avanço da tecnologia, modelos mais sofisticados, como o processamento de linguagem natural e modelos de aprendizado profundo, passaram a ser empregados, resultando em uma maior precisão e abrangência na geração de casos de teste [18]. As LLMs têm mostrado uma capacidade promissora na geração de testes, incluindo a criação de código de teste, entradas de teste e oráculos de teste [19]. Um exemplo notável é o TESTPILOT, uma ferramenta que gera automaticamente testes unitários para JavaScript utilizando modelos de

linguagem natural. Esta abordagem demonstrou ser altamente eficaz, superando significativamente a ferramenta Nessie, considerada o estado da arte da geração de testes unitários para JavaScript. Os resultados indicam que o uso de modelos de linguagem natural pode servir como uma alternativa eficaz às técnicas tradicionais de geração de testes [28]. No entanto, Jin et al. [18] afirmam que ao gerar testes de alta cobertura, os LLMs podem necessitar de prompts e etapas de pós-processamento mais elaboradas para alcançar os resultados desejados, sendo que a qualidade dos resultados depende fortemente do design e da qualidade do prompt utilizado.

- Segurança e manutenção de software: a compreensão semântica permite que os LLMs identifiquem falhas lógicas e problemas linguísticos que podem resultar em vulnerabilidades [39]. Jin et al. [18] afirmam que os LLMs têm demonstrado ser uma ferramenta promissora na área de segurança e manutenção ao automatizar a detecção e correção de vulnerabilidades. Sua utilização não apenas agiliza os processos de segurança, mas também contribui para a melhoria da qualidade do software, reduzindo o tempo e os custos associados à identificação e correção de falhas.

- Detecção estática de bugs: para Liu et al. [19], os LLMs podem ajudar a identificar potenciais problemas de qualidade em código-fonte. Ajustar LLMs em códigos com bugs ou simplesmente solicitar sua análise mostrou-se promissor na identificação de bugs, vulnerabilidades ou bad smells. No entanto, devido à diversidade e complexidade das causas-raiz de diferentes problemas de código e aos longos contextos, os LLMs apresentam precisão limitada na verificação estática de código em cenários reais.

- Documentação: Ozkaya [24] acredita que a geração automática de documentação de software é uma das aplicações promissoras dos LLMs. Esses modelos podem ser treinados para gerar diversos tipos de documentos, como contratos de software, requisitos funcionais e documentação de código, a partir de informações estruturadas ou não estruturadas.

- Tradução de linguagens de programação: A modernização de sistemas legados frequentemente impõe a necessidade de traduzir códigos de uma linguagem de programação para outra. LLMs apresentam um potencial promissor para auxiliar nessa tarefa, embora sua aplicação em larga escala ainda seja limitada. Estudos demonstram que LLMs podem

ser eficazes na tradução de fragmentos de código, mas a complexidade e a especificidade de sistemas legados exigem uma abordagem mais refinada [24].

4 Metodologia

Nesta seção, apresentamos a metodologia adotada no estudo, que segue uma abordagem quantitativa para avaliar a eficácia da LLM DeepSeek-V3 na resolução de problemas de manutenibilidade identificados pela ferramenta de análise estática SonarQube [9]. A escolha do DeepSeek foi motivada por sua relevância atual no campo da inteligência artificial, além da escassez de estudos que explorem seu uso, devido à sua recente introdução no mercado. Optamos especificamente pela versão "V3"em detrimento da versão "R1", pois, apesar de possuir capacidade de raciocínio passo a passo e ser mais adequada para tarefas complexas [36], a versão "R1"tem um tempo de resposta consideravelmente alto. Além disso, testes preliminares constataram um desempenho similar entre as duas versões no contexto de refatoração de code smells. Por fim, escolhemos o SonarQube [9] para identificar os problemas de manutenibilidade, devido à sua reputação como uma das ferramentas mais utilizadas e estudadas no campo da análise estática de código [1].

4.1 Questões de Pesquisa

1. **RQ1. Quais tipos de erros o DeepSeek comete com mais frequência ao tentar corrigir problemas de manutenibilidade em código C#?** Para responder a essa questão, durante o processo de análise, sempre que a LLM repetia o mesmo tipo de erro ao refatorar o código com uma determinada violação de regra, registrávamos essa informação em uma tabela para posterior discussão.

2. **RQ2. Até que ponto o DeepSeek é eficaz na correção de problemas de manutenibilidade em códigos C#?** Para responder a essa pergunta, cada problema de manutenibilidade refatorado pela LLM foi analisado para verificar se o problema foi resolvido, se o código compilava, se os testes unitários passavam e se nenhum novo problema de manutenibilidade foi gerado no processo.

4.2 Etapa 1: Coleta de Projetos no Github

Nesta etapa, utilizamos a plataforma de hospedagem de códigos-fonte GitHub para a seleção e obtenção dos projetos que seriam utilizados no estudo. A

escolha dos projetos foi guiada por critérios previamente definidos, conforme descrito abaixo:

- Popularidade: consideramos apenas projetos com mais de 1000 estrelas, indicando um nível significativo de relevância e aceitação pela comunidade;
- Predominância de código em C#: selecionamos projetos cujo código fosse composto majoritariamente por C#, correspondendo a pelo menos 90% do total;
- Presença de testes de unidade: os projetos deveriam ter testes de unidade para permitir a verificação de que o comportamento dos métodos refatorados não foi alterado;
- Atualização recente: para garantir que os projetos utilizados no estudo não estivessem defasados, consideramos apenas aqueles que haviam sido atualizados nos últimos quatro anos;
- Compilação: durante o processo de seleção, verificamos se os projetos eram capazes de compilar corretamente no ambiente utilizado para a pesquisa. Projetos que não atendiam a esse requisito foram descartados, pois poderiam comprometer a validade dos resultados.

Esses critérios foram estabelecidos para assegurar a relevância dos projetos selecionados, bem como para viabilizar a análise proposta neste trabalho. A Tabela 1 mostra os projetos escolhidos.

Tabela 1: Projetos escolhidos

Projeto	Estrelas	Contribuidores
jackett	13200	442
jellyfin	38800	1220
moq	6100	105
newtonsoftjson	11000	117
octokit	2800	282
questpdf	12700	31
swagger	5400	230
telegram bot	3400	44
tweetinvi	1000	20
closedxml	5000	87

4.3 Etapa 2: Identificação dos Problemas de Manutenibilidade

Nesta fase, utilizamos o SonarQube [9] para identificar problemas relacionados à manutenibilidade nos projetos selecionados na fase anterior. Primeiramente, foi necessário criar um projeto dentro da ferramenta, conforme ilustrado na Figura 1. Em seguida, geramos um token de autenticação específico para cada projeto, como mostrado na Figura 2. Esse token é essencial para vincular o projeto ao ambiente de análise.

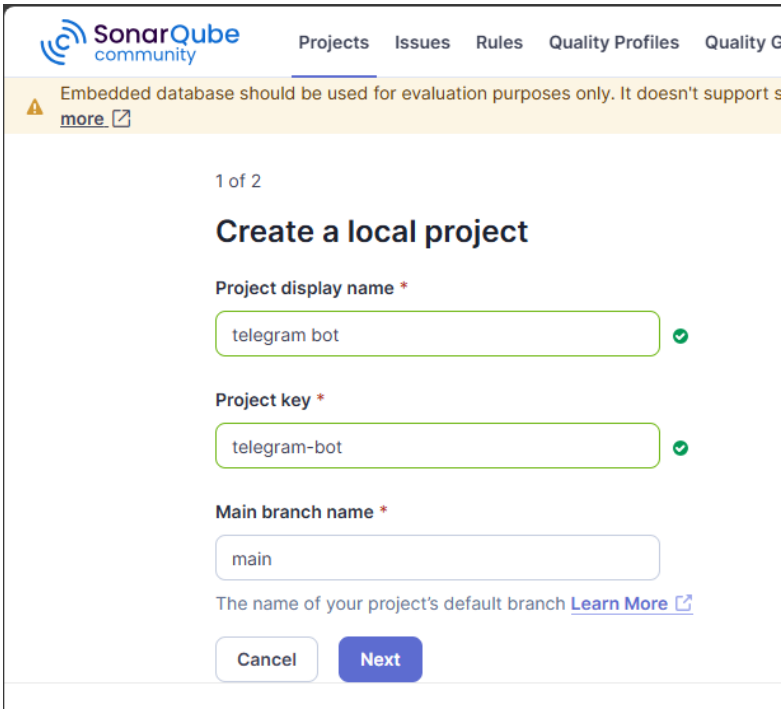

Figura 1: Tela de criação de projeto no SonarQube

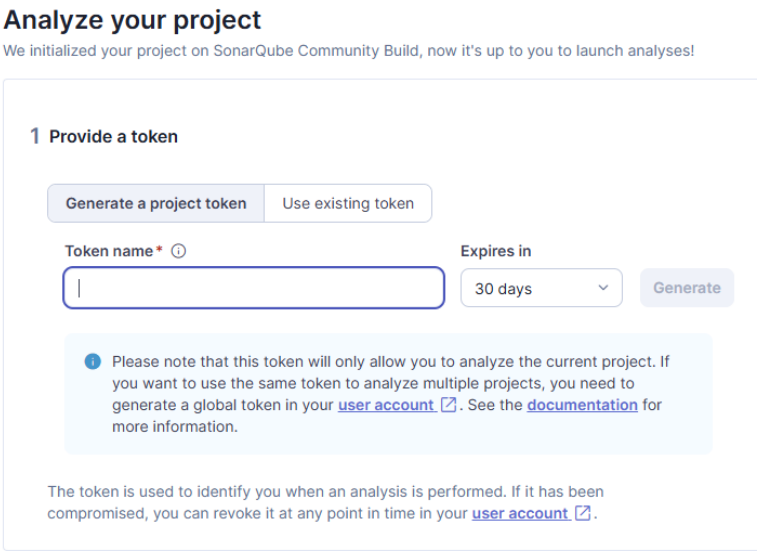

Figura 2: Tela de criação do token do projeto

Posteriormente, configuramos o tipo e o framework do projeto a ser analisado. Para os projetos avaliados, utilizamos as opções ".NET"e ".NET Core", como evidenciado na Figura 3. Após essa configuração inicial, executamos os comandos necessários no terminal do projeto, conforme apresentado na Figura 4.

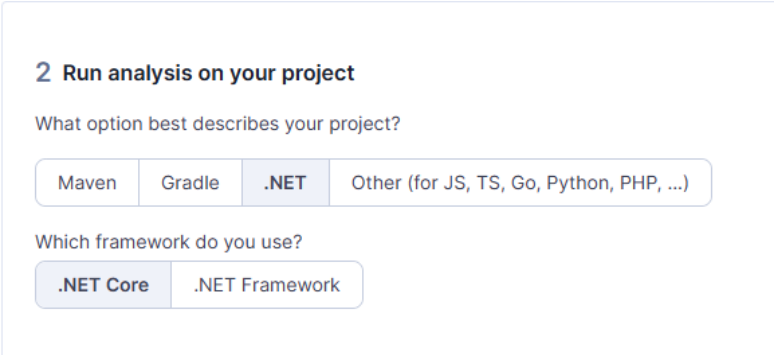

Figura 3: Tela de escolha do framework do projeto a ser analisado

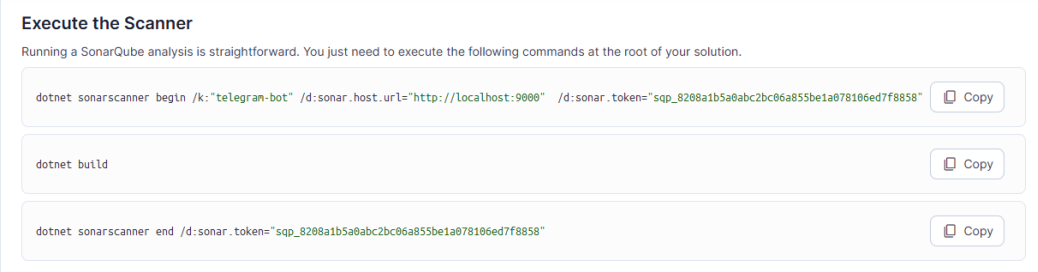

Figura 4: Tela de comandos a serem executados no terminal do projeto

Ao término da análise estática realizada pelo SonarQube [9], a ferramenta exibe uma interface detalhada com os resultados obtidos. Dentre as informações fornecidas, destacam-se os problemas de manutenibilidade identificados, que podem ser acessados por meio de uma das seções de navegação disponíveis na tela de resultados (Figura 5).

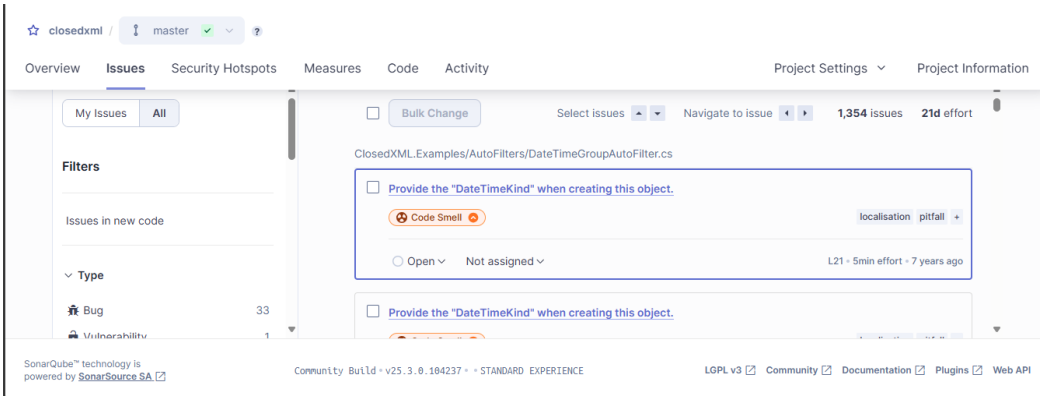

Figura 5: Tela de problemas de manutenibilidade do projeto

4.4 Etapa 3: Seleção de Problemas de Manutenibilidade

Com as 16.123 violações de regras identificadas na etapa anterior, foi necessário realizar uma filtragem para selecionar as regras e os problemas de manutenibilidade que seriam refatorados pela LLM. Para isso, aplicamos critérios específicos, o que resultou na seleção final de oito regras e 126 problemas de manutenibilidade.

4.4.1 Escolha de regras

A seleção das regras foi realizada de forma aleatória, considerando apenas aquelas classificadas com severidade "*major*"ou "*critical*", ou seja, regras cuja violação representa problemas de maior seriedade. Além disso, as regras selecionadas deveriam atender aos seguintes critérios:

- Ter violações detectadas em pelo menos cinco projetos diferentes;
- Apresentar um mínimo de 18 violações detectadas entre todos os projetos analisados.

A Tabela 3 apresenta as oito regras escolhidas com base nesses critérios. Com essa filtragem inicial, chegamos a um total de 1351 problemas de manutenibilidade.

Tabela 2: Regras e suas siglas

Regra	Sigla
Cognitive Complexity of methods should not be too high	CCM
Methods should not have too many parameters	MSP
Mergeable "if"statements should be combined	MSC
Utility classes should not have public constructors	UCC
Unused method parameters should be removed	UMP
LINQ expressions should be simplified	LES
Properties should not make collection or array copies	PCA
Empty arrays and collections should be returned instead of null	EAC

Tabela 3: Número de problemas identificados e analisados por regra

Sigla	Recorrência em projetos	Problemas encontrados	Problemas analisados
CCM	10	650	54
MSP	8	263	25
MSC	8	278	26
UCC	7	47	8
UMP	6	21	1
LES	5	46	9
PCA	5	19	1
EAC	5	27	2
		1351	126

4.4.2 Escolha de violações de regra

Para a seleção dos problemas de manutenibilidade, foram excluídos códigos relacionados a testes, uma vez que o foco do estudo é identificar padrões generalizados em código de produção. Considerando que este é um estudo exploratório, adotamos uma amostra estatisticamente representativa com 90% de confiança e 7% de margem de erro, o que resultou na escolha de 126 violações. O processo de seleção seguiu os passos descritos abaixo:

1. Análise do código do projeto no SonarQube para identificação de violações;
2. Exportação dos dados no formato .csv;
3. Criação de planilha com os todos os problemas de manutenibilidade que atendessem os critérios;
4. Seleção aleatória das violações de regras utilizando a função "ALEATORIOENTRE"do Google Sheets;
5. Verificação da cobertura de testes nos métodos selecionados. Somente métodos cobertos por testes seriam utilizados;

A Tabela 4 mostra a quantidade de violações escolhidas por regra e projeto.

A abordagem adotada nesta etapa garantiu a representatividade da seleção dos problemas de manutenibilidade. A escolha aleatória das violações reduziu possíveis vieses no processo, enquanto o tamanho da amostra (126 violações) foi estatisticamente representativo. Além disso, a filtragem das regras, baseada na severidade ("*major*"ou "*critical*") e no número mínimo de ocorrências (ao menos 18 violações detectadas em cinco ou mais projetos), assegurou que os problemas selecionados fossem relevantes. Por fim, a variabilidade foi assegurada ao garantir que cada violação de regra estivesse presente múltiplos projetos.

Tabela 4: Número de problemas analisados por regra e projeto

Projeto	CCM	MSP	MSC	UCC	UMP	LES	PCA	EAC	Total
jackett	10	1	2	0	1	2	1	0	17
jellyfin	17	9	14	0	0	1	0	0	41
moq	0	0	1	0	0	1	0	0	2
newtonsoftjson	14	0	5	0	0	2	0	0	21
octokit	1	6	0	0	0	0	0	0	7
questpdf	0	0	0	3	0	0	0	0	3
swagger	2	0	0	4	0	0	0	0	6
telegram	2	9	0	0	0	0	0	0	11
tweetinvi	0	0	1	1	0	0	0	1	3
closedxml	8	0	3	0	0	3	0	1	15

4.5 Etapa 4: Uso do DeepSeek para Resolver Problemas de Manutenibilidade

Nesta etapa, utilizamos o DeepSeek para refatorar os problemas de manutenibilidade selecionados na etapa anterior. O acesso à LLM foi realizado por meio do link `https://www.deepseek.com/`, onde inseríamos os prompts no campo de texto disponível.

4.5.1 Prompt

Adotamos a estratégia de prompting conhecida como *zero-shot*, que consiste em fornecer ao modelo uma tarefa com instruções explícitas, sem exemplos prévios ou treinamento adicional. O prompt utilizado incluía a assinatura do método e o nome da regra violada pelo problema de manutenibilidade, além de uma referência ao código original do método. A estrutura do prompt era: "In the following class, the method ASSINATURA_METODO has the issue NOME_REGRA. Can you identify and fix it? Original code: ", sendo que os trechos "ASSINATURA_METODO"e "NOME_REGRA"eram substituídos, respectivamente, pela assinatura do método a ser refatorado e pelo título da violação detectada. Esse processo foi realizado de forma semi-automatizada, utilizando a planilha criada na etapa anterior (Figura 6), que já continha uma coluna dedicada aos prompts. Na planilha, o campo "NOME_REGRA"já estava preenchido automaticamente com o título correto da violação, enquanto o trecho "ASSINATURA_METODO"era preenchido manualmente para cada caso a ser analisado.

E
prompt
In the following class, the method "ASSINATURA_METODO" has the issue "Cognitive Complexity of methods should not be too high". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Cognitive Complexity of methods should not be too high". Can you identify and fix it? Original code:
In the following class, the method "ToHtml(string message, MessageEntity[]? entities)" has the issue "Cognitive Complexity of methods should not be too high". Can you identif
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Cognitive Complexity of methods should not be too high". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Cognitive Complexity of methods should not be too high". Can you identify and fix it? Original code:
In the following class, the method "SetWebhook(this ITelegramBotClient botClient, string url, InputFileStream? certificate = default, string? ipAddress = default, int? maxConne
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:
In the following class, the method "ASSINATURA_METODO" has the issue "Methods should not have too many parameters". Can you identify and fix it? Original code:

Figura 6: Trecho da planilha de 1351 problemas de manutenibilidade

4.5.2 Aplicação da sugestão de refatoração

Para obter as sugestões de refatoração, utilizamos o site do DeepSeek (`https://www.deepseek.com/`). O processo consistiu em copiar o prompt previamente preparado na planilha de violações e colá-lo no campo de texto do chat da ferramenta. Em seguida, substituímos o trecho "ASSINATURA_METODO"pela assinatura específica do método a ser refatorado e adicionamos o código original do método ao final do prompt, conforme ilustrado na Figura 7. Após enviar o prompt, recebemos a resposta gerada pela LLM contendo a sugestão de refatoração (Figura 8). Com base nessa resposta, substituímos o código original pelo código refatorado sugerido. Esse procedimento foi repetido para cada um dos 126 problemas de manutenibilidade selecionados.

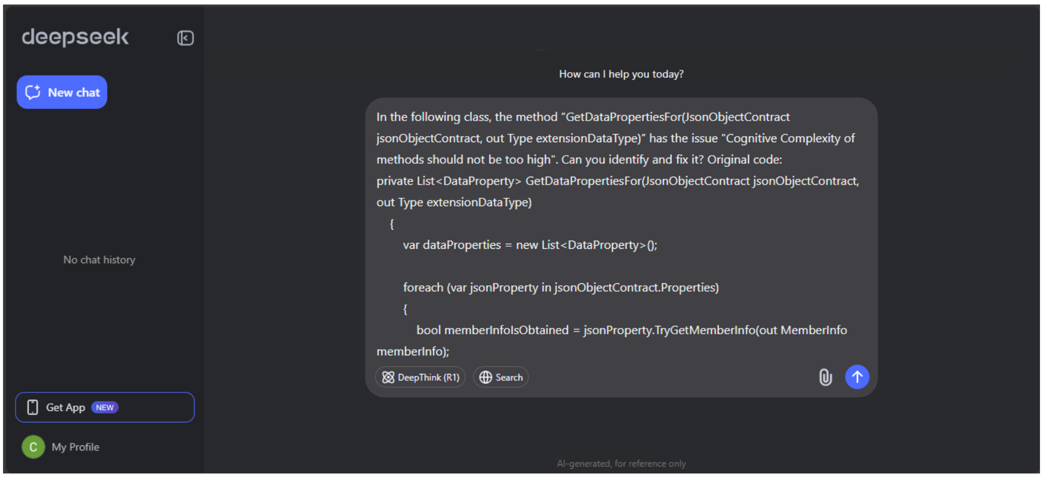

Figura 7: Chat do DeepSeek

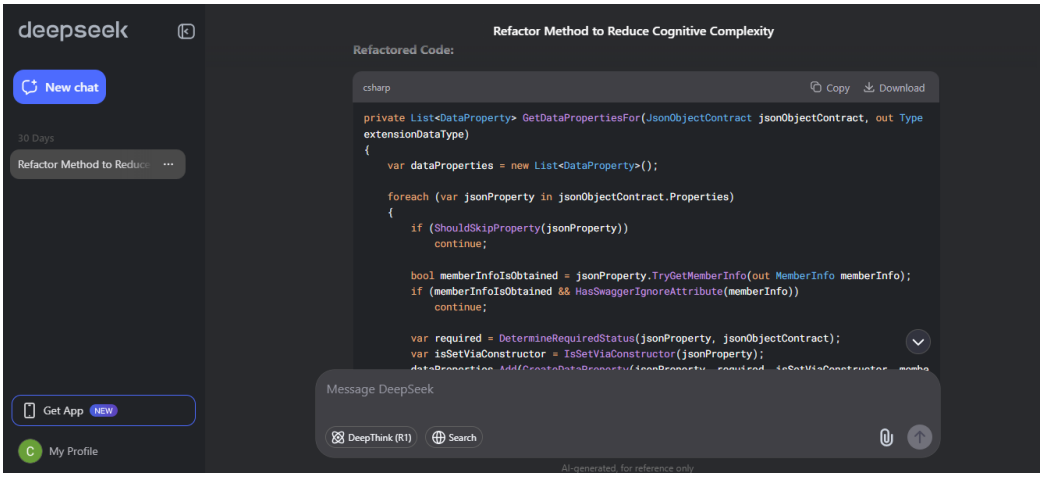

Figura 8: Resposta do DeepSeek

4.6 Etapa 5: Compilação e Teste

O objetivo desta etapa foi assegurar que, após as refatorações realizadas, o projeto permanecesse funcional e com o mesmo comportamento esperado. Para isso, inicialmente, realizamos a compilação do projeto para garantir que não houvesse erros de sintaxe ou problemas estruturais introduzidos pelas alterações. Em seguida, executamos os testes automatizados existentes no projeto, verificando se todos os casos de teste eram aprovados, o que indicaria que o comportamento original do sistema não havia sido alterado.

4.7 Etapa 6: Reanálise do Código Refatorado

Se tivesse sucesso na etapa anterior, o projeto era submetido a uma nova análise estática utilizando o SonarQube, empregando os mesmos comandos apresentados na Figura 4. O objetivo dessa reanálise era verificar se as refatorações realizadas haviam resolvido os problemas de manutenibilidade previamente identificados e garantir que nenhuma nova violação tivesse sido introduzida no código. Para facilitar esse processo, utilizamos a planilha de problemas de manutenibilidade, que continha uma coluna dedicada aos links diretos para cada violação no SonarQube, conforme ilustrado na Figura 9.

A verificação da resolução dos problemas era realizada acessando o link correspondente e observando o informativo exibido na tela, como mostrado na Figura 10. Já para identificar possíveis novos problemas de manutenibilidade, navegávamos até a seção “new code”na visão geral do projeto e verificávamos o valor indicado em “new issues”(Figura 11). Caso novos problemas fossem detectados, eles poderiam ser visualizados clicando no número

correspondente.

component	line	link_issue
telegram:src/Telegram.Bot/Polling/DefaultUpdateReceiver.cs	18	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=6dd26c44-77a0-4780-a196-483f01cf2564
telegram:src/Telegram.Bot/Extensions/FormatExtensions.cs	23	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=818796c5-288f-4ba2-81bf-0d6197c51216
telegram:src/Telegram.Bot/Extensions/FormatExtensions.cs	137	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=61e5c436-ab35-4a4f-b987-c4122aaeda70
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	3131	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=900f98cb-53e6-4cdb-91eb-7fbd72664190
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	3278	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=f706f10f-0d4a-4b10-9b23-1e79cd5d674c
telegram:src/Telegram.Bot/Polling/AsyncEnumerableReceivers/QueuedUpdateReceiver.cs	91	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=c42d31ee-4634-4a12-9dc4-c857663bc17b
telegram:src/Telegram.Bot/TelegramBotClient.cs	99	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=e4e9b790-0efd-4250-a223-d410cef0f327
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	48	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=b751729c-f7ab-428b-b3e6-b3cbb703408d
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	220	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=2da85f22-e020-4cae-a515-5ac617b95f2e
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	303	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=0a2e343d-75df-480d-8160-0390fbfaa4ff
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	817	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=c4e9300a-9c1a-41d8-917d-78fc21c2bcd6
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	1315	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=5c49bc44-f892-4d3e-99d1-ec617d52e719
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	1477	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=dc96b5d2-e043-4240-a466-824daccd5a5a
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	2280	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=f1237a26-b11d-4f5f-aa7c-25b2e63d882c
telegram:src/Telegram.Bot/TelegramBotClientExtensions.ApiMethods.cs	2313	http://localhost:9000/issues?issueStatuses=OPEN%2CCONFIRMED&open=93dffd1d-801f-4f3f-b1ff-0442468e7613

Figura 9: Trecho da planilha de problemas de violações - link para issue

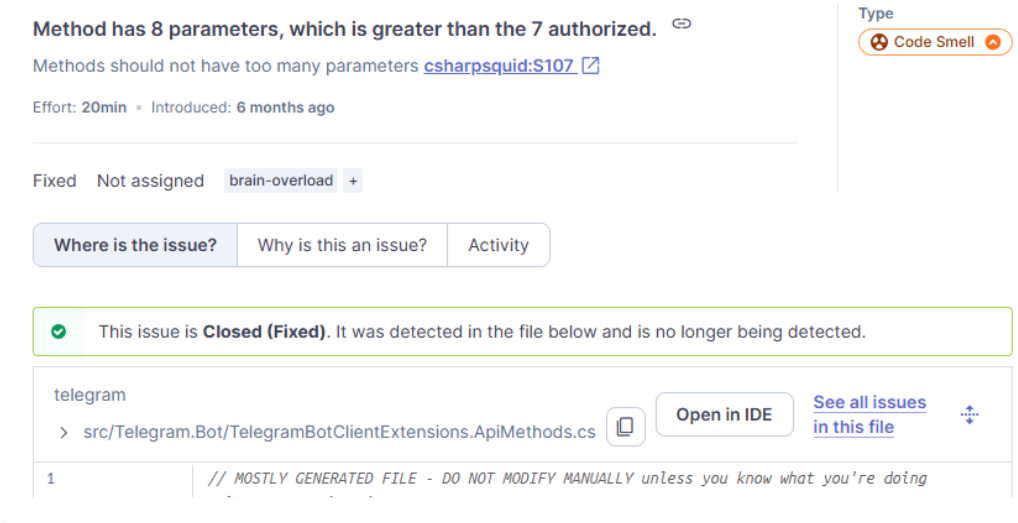

Figura 10: Visualização de um problema de manutenibilidade resolvido

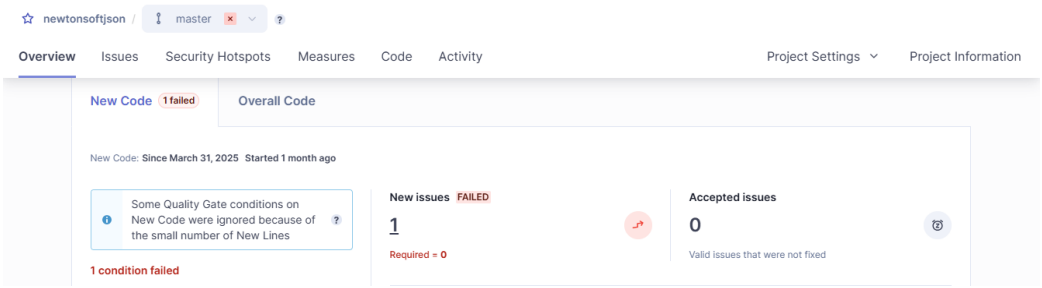

Figura 11: Seção "new code"de um projeto com um novo problema de manutenibilidade introduzido

4.8 Considerações Finais

A metodologia adotada neste trabalho mostrou-se eficaz para alcançar os objetivos propostos, permitindo a identificação, refatoração e validação de problemas de manutenibilidade em projetos escritos em C#. Desde a seleção dos projetos até a aplicação das refatorações sugeridas pela LLM e a reanálise do código, cada fase contribuiu para uma boa avaliação da capacidade da ferramenta em melhorar a qualidade de código.

Entretanto, alguns empecilhos foram enfrentados ao longo do processo. Uma das principais dificuldades foi encontrar projetos que atendessem aos critérios iniciais estabelecidos, como presença de testes unitários e atualização recente, além de garantir que esses projetos compilassem corretamente no computador utilizado. Além disso, a execução dos testes automatizados se mostrou um processo demorado, especialmente em projetos maiores.

Apesar dessas dificuldades, a metodologia se mostrou consistente e forneceu resultados relevantes. Tais resultados são discutidos na próxima seção.

5 Resultados

Nesta seção, apresentamos os resultados obtidos a partir da avaliação das refatorações realizadas pelo DeepSeek-V3 em métodos que continham diferentes problemas de manutenibilidade identificados pelo SonarQube. Para organização, definimos categorias que classificam os resultados. Além disso, detalhamos os resultados gerais do estudo, exploramos as falhas específicas por categoria e examinamos o desempenho da LLM em relação a cada regra avaliada.

5.1 Categorias de Resultados

Definimos categorias para agrupar os resultados obtidos na análise de cada ocorrência de code smell analisada.

- Resolvido: esta categoria abrange os métodos que foram corrigidos pela LLM sem introduzir novos problemas de manutenibilidade e que passavam nos testes;

- Não resolvido: esta categoria inclui métodos que passaram nos testes, não geraram erros de compilação, mas não solucionaram o problema de manutenibilidade. Também incluímos nessa categoria os casos em que a refatoração solucionou outra ocorrência do mesmo problema de manutenibilidade no método, mas não a desejada;

- Erro de compilação: esta categoria refere-se a métodos que causaram erro de compilação após a intervenção da LLM;
- Erro em testes: esta categoria abrange os casos em que os testes automatizados falharam após a intervenção da LLM;
- Gerou novo problema: esta categoria inclui os casos em que ao menos um novo problema de manutenibilidade foi criado após a intervenção da LLM;
- Não sugeriu: esta categoria se refere aos casos em que a LLM não sugeriu correção para o problema, ou seja, retornou um código idêntico ao original.

5.2 Resultados Gerais

A Figura 12 mostra os resultados gerais do estudo, evidenciando a comparação entre o número de problemas de manutenibilidade resolvidos e não resolvidos pelas refatorações. Dos 126 métodos analisados, o DeepSeek-v3 refatorou com sucesso 62 (49,21%) e falhou ao refatorar 64 (50,79%).

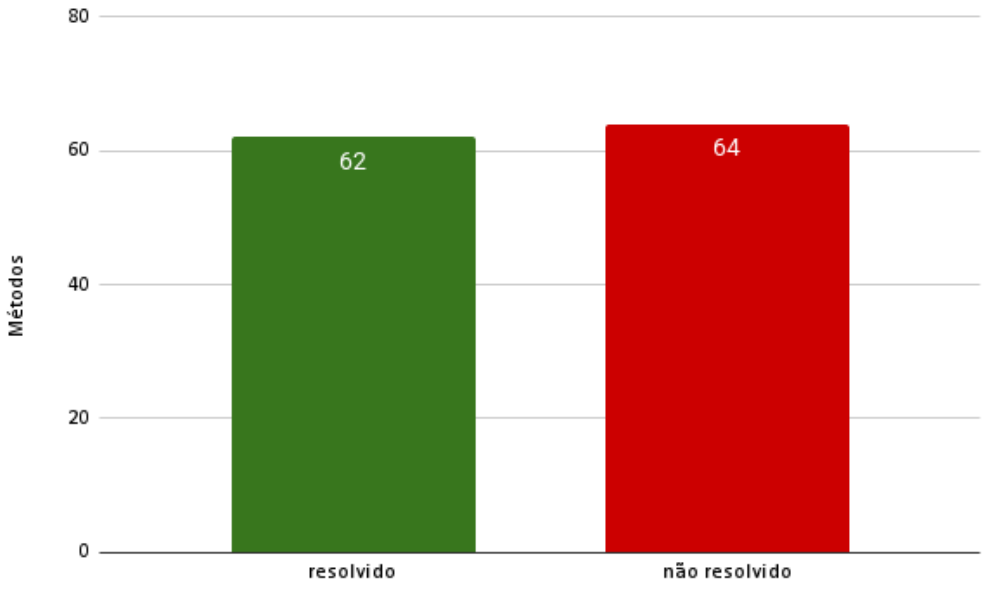

Figura 12: Resultados gerais

5.3 Resultados por Categorias

A Figura 13 apresenta os resultados organizados de acordo com as categorias de análise definidas no estudo, proporcionando uma visão mais clara das situações em que a LLM falhou ao lidar com os code smells. Os dados revelam que, em 28 métodos (22,22%), a refatoração realizada resultou em erros de compilação. Em outros 6 métodos (4,76%), a refatoração alterou o comportamento esperado, causando falhas nos testes automatizados. Além disso, em 8 métodos (6,35%), o problema original de manutenibilidade não foi resolvido, enquanto em 21 métodos (16,66%) a refatoração introduziu novos problemas de manutenibilidade.

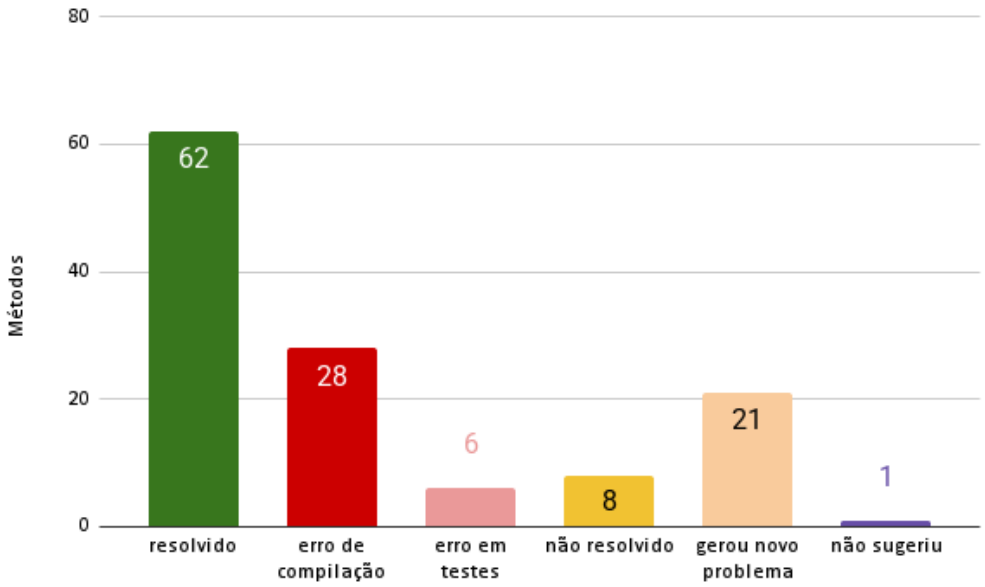

Figura 13: Resultados por categorias

5.4 Resultados por Regras do SonarQube

As Figuras 14 e 15 apresentam os resultados organizados com base nas regras do SonarQube, que, quando violadas, configuram problemas de manutenibilidade.

A regra *The cognitive complexity of methods should not be too high* (CCM) identifica métodos com alta complexidade cognitiva, ou seja, métodos cujo fluxo de controle é difícil de entender, tornando-os mais complicados de ler, testar e modificar. Foram avaliados 54 métodos que violaram essa regra, dos quais a LLM conseguiu resolver 10, mas falhou em 44. Entre os casos de

insucesso, 22 métodos resultaram em erros de compilação, 3 métodos apresentaram erros em testes, um método permaneceu sem solução e 18 métodos tiveram novos problemas de manutenibilidade gerados.

Já a regra *Methods should not have too many parameters* (MSP) alerta sobre métodos com listas longas de parâmetros, o que dificulta a compreensão do papel de cada um e aumenta a complexidade ao utilizá-los. Nesse caso, avaliamos 25 métodos, e a LLM foi capaz de resolver 15 deles, enquanto falhou em 10. Entre os casos de falha, 6 métodos resultaram em erros de compilação, três métodos apresentaram erros em testes e 1 método teve um novo problema de manutenibilidade gerado.

A regra *Mergeable "if"statements should be combined* (MSC) sugere evitar blocos de código aninhados, como declarações if dentro de outras, sempre que possível. A combinação dessas declarações reduz a profundidade do código, tornando-o mais simples e legível. Avaliamos 26 métodos que violaram essa regra, dos quais a LLM conseguiu resolver 18, mas falhou em 8. Entre os casos de insucesso, 6 métodos não tiveram o problema resolvido, um método apresentou novos problemas de manutenibilidade, e, em um caso, a LLM não foi capaz de sugerir alteração.

Por outro lado, a regra *Utility classes should not have public constructors* (UCC) estabelece que classes utilitárias, compostas apenas por membros estáticos, não devem possuir construtores públicos, já que essas classes não precisam ser instanciadas. Avaliamos 8 ocorrências de violação dessa regra e a LLM foi capaz de resolver todas elas.

A regra *Unused method parameters should be removed* (UMP) detecta parâmetros declarados e não utilizados em métodos. Esses parâmetros desnecessários podem causar confusão e prejudicar a clareza do código. Avaliamos 1 método que violou essa regra, e a LLM conseguiu resolvê-lo. Já a regra *LINQ expressions should be simplified* (LES) recomenda simplificar expressões *LINQ* para melhorar a legibilidade do código e, em alguns casos, otimizar seu desempenho. Avaliamos 9 métodos que violaram essa regra, dos quais a LLM conseguiu resolver 7, mas falhou em 2. Entre os casos de falha, 1 método teve novos problemas de manutenibilidade introduzidos e o outro método permaneceu sem resolução.

A regra *Properties should not make collection or array copies* (PCA) destaca que propriedades que retornam cópias de coleções ou *arrays* podem ser ineficientes. Avaliamos 1 ocorrência de violação dessa regra e a LLM foi capaz de resolvê-la. Por fim, a regra *Empty arrays and collections should be returned instead of null* (EAC) recomenda que métodos retornem *arrays* ou coleções vazias em vez de valores nulos, para evitar verificações adicionais de nulidade que tornam o código mais complexo e menos legível. Avaliamos 2 ocorrências de violação dessa regra, e a LLM resolveu ambas.

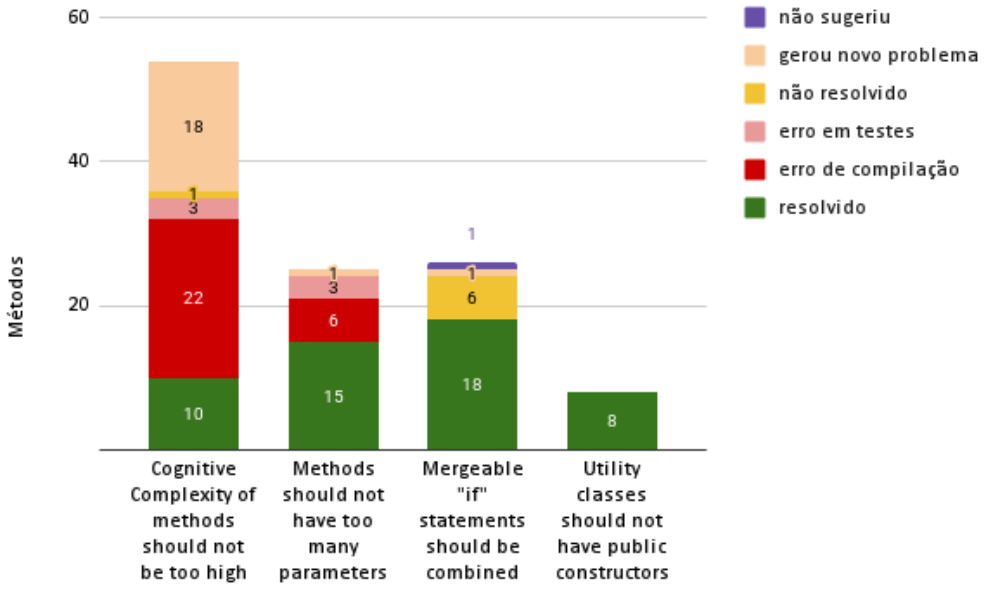

Figura 14: Resultados por regras e categorias - Primeira parte

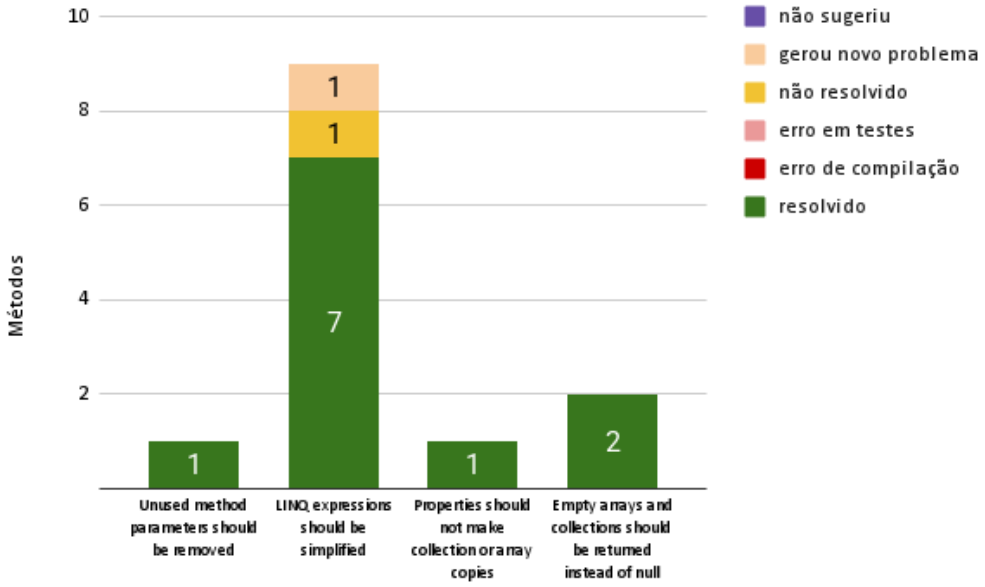

Figura 15: Resultados por regras e categorias - Segunda parte

6 Discussão

Neste seção, apresentamos as discussões sobre a análise dos resultados obtidos nos experimentos de refatoração de código C# utilizando o DeepSeek-V3. A análise foi orientada pelas questões de pesquisa definidas no início da seção de metodologia, com foco na eficácia da LLM em melhorar a qualidade do código, especialmente em termos de manutenibilidade.

6.1 RQ1. Quais tipos de erros o DeepSeek comete com mais frequência ao tentar corrigir problemas de manutenibilidade em código C#?

Ao longo da análise das refatorações propostas pela LLM para diferentes os diferentes problemas de manutenibilidade, foi possível identificar padrões recorrentes de erros, que são detalhados a seguir:

1. Em algumas refatorações para a regra *The cognitive complexity of methods should not be too high* (CCM), a LLM falhou ao não marcar os novos métodos criados como *static*, mesmo quando isso era necessário para manter a consistência do código. Além disso, em alguns casos, embora tenha criado métodos estáticos corretamente, continuou utilizando atributos da classe original, o que resultou em dependências inválidas e erros de compilação.

2. Ao tentar reduzir o número de parâmetros nos métodos para seguir a regra *Methods Should Not Have Too Many Parameters* (MSP), a LLM frequentemente sugeriu a criação de classes para encapsular os parâmetros. No entanto, houve situações em que essas classes continham parâmetros incorretos ou desnecessários, comprometendo a funcionalidade e a clareza do código. Em outros casos, a LLM manteve métodos com parâmetros não utilizados, deixando o problema parcialmente resolvido.

3. Ao tratar das violações da regra *Mergeable "If"Statements Should Be Combined* (MSC), a LLM, em alguns casos, otimizou outras ocorrências do problema no método, mas ignorou a esperada, deixando o problema principal sem solução.

4. Os erros de compilação foram uma categoria significativa de falhas observadas nas soluções geradas pela LLM. As causas de tais erros que mais se repetiram foram a implementação de blocos *if* sem chaves "{}",

violando a regra do Visual Studio que exige seu uso explícito; e o uso de variáveis inacessíveis no contexto atual, seja por falta de inicialização ou por referência a atributos removidos ou deslocados durante a refatoração.

6.2 RQ2. Até que ponto o DeepSeek é eficaz na correção de problemas de manutenibilidade em códigos C#

O DeepSeek demonstrou uma eficácia moderada na correção de problemas de manutenibilidade em códigos C#, com uma taxa de sucesso geral de 49,21% (62 métodos resolvidos entre 126 analisados). Apesar disso, 50,79% dos casos apresentaram falhas, sendo que 22,22% resultaram em erros de compilação, 4,76% causaram falhas nos testes automatizados, 6,35% não resolveram o problema original e 16,66% introduziram novos problemas de manutenibilidade. Esses resultados evidenciam tanto o potencial da ferramenta quanto suas limitações, especialmente em cenários mais complexos.

A análise dos resultados por regras do SonarQube revelou que a eficácia do DeepSeek varia conforme a complexidade do problema. Em regras simples e bem definidas, como *Utility classes should not have public constructors* (UCC), *Empty arrays and collections should be returned instead of null* (EAC), *Properties should not make collection or array copies* (PCA) ou *LINQ expressions should be simplified* (LES), a LLM apresentou um bom desempenho. Por outro lado, em problemas mais complexos, como aqueles relacionados à regra *Cognitive Complexity of methods should not be too high* (CCM), o desempenho foi significativamente inferior, com taxas de sucesso abaixo de 20%. Isso sugere que a LLM tem maior dificuldade em lidar com refatorações que exigem compreensão profunda do contexto ou mudanças estruturais mais amplas.

Os resultados indicam que o DeepSeek é eficaz na resolução de problemas de baixa complexidade, mas apresenta limitações significativas ao refatorar por completo problemas de manutenibilidade complexos. Ainda assim, a ferramenta pode ser útil no processo de refatoração desses problemas: em muitos casos em que não consegue solucionar integralmente o code smell, o DeepSeek contribui ao fornecer sugestões iniciais ou parciais que encurtam o caminho para a solução final, reduzindo o tempo e o esforço necessários por parte do desenvolvedor.

7 Limitações do Estudo

A pesquisa possui algumas limitações que devem ser ponderadas. Primeiramente, o estudo foi conduzido utilizando apenas uma linguagem de programação, o C#, o que restringe a generalização dos resultados para outras linguagens. Além disso, apesar da disponibilidade da versão "R1"do DeepSeek - mais adequada para tarefas de raciocínio avançado - foi utilizada a versão "V3"do DeepSeek, que é uma LLM de propósito geral. Outra limitação está na escolha das regras do SonarQube: foram consideradas apenas 8 regras, sendo que 2 delas são específicas para a linguagem C#.

Adicionalmente, embora o tamanho da amostra utilizada (126 métodos) seja significativo, um conjunto maior de dados poderia fornecer uma análise mais representativa. Por fim, a estratégia de prompting adotada considerou apenas o contexto local da classe em que o método estava inserido, sem explorar informações de outros arquivos relacionados. Essas limitações abrem espaço para estudos futuros que explorem cenários mais amplos e diversificados, bem como estratégias mais avançadas de interação com a ferramenta.

8 Conclusões

Este estudo teve como objetivo avaliar a eficácia do DeepSeek ao resolver code smells relacionados à manutenibilidade de software em código C#. Os resultados mostraram que o DeepSeek possui um bom potencial para refatorar problemas de manutenibilidade, demonstrando-se um avanço importante para a Engenharia de Software: sua capacidade de, no mínimo, encurtar o caminho dos desenvolvedores nessas tarefas permite que haja mais tempo a ser gasto em aspectos mais estratégicos do projeto.

No entanto, o estudo também identificou limitações importantes no uso do DeepSeek. Em diversos casos, a LLM introduziu novos problemas de manutenibilidade, alterou o comportamento esperado dos métodos ou gerou erros de compilação, especialmente ao lidar com problemas mais complexos. Esses resultados evidenciam que, embora promissora, a ferramenta ainda requer a supervisão humana no processo de refatoração. Por isso, seu uso deve ser encarado como complementar, exigindo revisão por parte dos desenvolvedores para garantir que as alterações propostas mantenham a integridade do código.

Referências

[1] G. Annamalai e Patroklos P. Papapetrou. *SonarQube in Action*. Shelter Island, NY, USA: Manning Publications, 2013. ISBN: 978-1617290954. URL: https://www.manning.com/books/sonarqube-in-action.

[2] Omer Aydin et al. "Generative AI in Academic Writing: A Comparison of DeepSeek, Qwen, ChatGPT, Gemini, Llama, Mistral, and Gemma". Em: *arXiv preprint arXiv:2503.04765* (2025).

[3] Jaehyeon Bae, Seoryeong Kwon e Seunghwan Myeong. "Enhancing software code vulnerability detection using gpt-4o and claude-3.5 sonnet: A study on prompt engineering techniques". Em: *Electronics* 13.13 (2024), p. 2657.

[4] S.R. Chidamber e C.F. Kemerer. "A metrics suite for object oriented design". Em: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. DOI: 10.1109/32.295895.

[5] *Claude*. https://www.anthropic.com/claude/. 2024.

[6] *Claude architecture*. https://claude3.pro/claude-3-5-sonnet-architecture/. 2024.

[7] *Documentação do Checkstyle*. https://checkstyle.sourceforge.io/. 2024.

[8] *Documentação do JDeodorant*. https://marketplace.eclipse.org/content/jdeodorant. 2024.

[9] *Documentação do SonarQube*. https://docs.sonarsource.com/. 2024.

[10] *Documentação do SpotBugs*. https://spotbugs.readthedocs.io/en/stable/. 2024.

[11] *Documentação PMD*. https://docs.pmd-code.org/latest/. 2024.

[12] Chongzhou Fang et al. "Large language models for code analysis: Do {LLMs} really do their job?" Em: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 829–846.

[13] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] *Gemini 1.5*. https://blog.google/intl/pt-br/novidades/nosso-modelo-de-proxima-geracao-gemini-15/. 2025.

[15] *Introducing ChatGPT*. https://openai.com/index/chatgpt/. 2024.

[16] ISO/IEC 25010. *ISO/IEC 25010:2011, Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. 2011.

[17] Qile Jiang, Zhiwei Gao e George Em Karniadakis. "DeepSeek vs. ChatGPT vs. Claude: A comparative study for scientific computing and scientific machine learning tasks". Em: *Theoretical and Applied Mechanics Letters* 15.3 (2025), p. 100583.

[18] Haolin Jin et al. "From llms to llm-based agents for software engineering: A survey of current, challenges and future". Em: *arXiv preprint arXiv:2408.02479* (2024).

[19] Junwei Liu et al. "Large language model-based agents for software engineering: A survey". Em: *arXiv preprint arXiv:2409.02977* (2024).

[20] Jim A. McCall, Paul K. Richards e Gene F Walters. *Factors in Software Quality*. Rel. técn. General Electric Company, 1977.

[21] Timothy R. McIntosh et al. *From Google Gemini to OpenAI Q* (Q-Star): A Survey of Reshaping the Generative Artificial Intelligence (AI) Research Landscape*. 2023. arXiv: 2312.10868 [cs.AI]. URL: https://arxiv.org/abs/2312.10868.

[22] *Modelos do Gemini*. https://ai.google.dev/gemini-api/docs/models/gemini?hl=pt-br. 2025.

[23] Henrique Nunes et al. *Evaluating the Effectiveness of LLMs in Fixing Maintainability Issues in Real-World Projects*. 2025. arXiv: 2502.02368 [cs.SE]. URL: https://arxiv.org/abs/2502.02368.

[24] Ipek Ozkaya. "Application of large language models to software engineering tasks: Opportunities, risks, and implications". Em: *IEEE Software* 40.3 (2023), pp. 4–8.

[25] Kartheek Kalluri e Abhilash Kokala. *PERFORMANCE BENCHMARKING OF GENERATIVE AI MODELS: CHATGPT-4 VS. GOOGLE GEMINI AI*.

[26] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc, 2010.

[27] Anichur Rahman et al. "Comparative Analysis Based on DeepSeek, ChatGPT, and Google Gemini: Features, Techniques, Performance, Future Prospects". Em: *arXiv preprint arXiv:2503.04783* (2025).

[28] Max Schäfer et al. “An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation”. Em: *IEEE Transactions on Software Engineering* 50.1 (2024), pp. 85–105. DOI: 10.1109/TSE.2023.3334955.

[29] Sakib Shahriar et al. “Putting gpt-4o to the sword: A comprehensive evaluation of language, vision, speech, and multimodal proficiency”. Em: *Applied Sciences* 14.17 (2024), p. 7782.

[30] Ian Sommerville. *Engenharia de Software*. 9ª ed. São Paulo: Pearson Prentice Hall, 2011.

[31] Pedro Carneiro de Souza. “Avaliando a Eficácia de Modelos de Linguagem de Grande Escala (LLMs) na Refatoração de Código Python em Projetos do Mundo Real”. Diss. de mestr. Universidade Estadual de Feira de Santana, 2025.

[32] Darko Stefanović et al. “Static code analysis tools: A systematic literature review”. Em: *Ann. DAAAM Proc. Int. DAAAM Symp.* Vol. 31. 1. 2020, pp. 565–573.

[33] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2024. arXiv: 2312.11805 [cs.CL]. URL: https://arxiv.org/abs/2312.11805.

[34] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[35] Chengen Wang e Murat Kantarcioglu. *A Review of DeepSeek Models' Key Innovative Techniques*. 2025. arXiv: 2503.11486 [cs.LG]. URL: https://arxiv.org/abs/2503.11486.

[36] Chong Zhang et al. *100 Days After DeepSeek-R1: A Survey on Replication Studies and More Directions for Reasoning Language Models*. 2025. arXiv: 2505.00551 [cs.CL]. URL: https://arxiv.org/abs/2505.00551.

[37] Quanjun Zhang et al. “A critical review of large language model on software engineering: An example from chatgpt and automated program repair”. Em: *arXiv preprint arXiv:2310.08879* (2023).

[38] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2024. arXiv: 2303.18223 [cs.CL]. URL: https://arxiv.org/abs/2303.18223.

[39] Zibin Zheng et al. *Towards an Understanding of Large Language Models in Software Engineering Tasks*. 2024. arXiv: 2308.11396 [cs.SE]. URL: https://arxiv.org/abs/2308.11396.