UNIVERSIDADE FEDERAL DE OURO PRETO
INSTITUTO DE CIÊNCIAS EXATAS E BIOLÓGICAS
DEPARTAMENTO DE COMPUTAÇÃO

GABRIEL HENRIQUE MOREIRA ROCHA

DETECÇÃO AUTOMÁTICA DE REFATORAÇÕES: UM ESTUDO EXPLORATÓRIO UTILIZANDO LLMS

Ouro Preto
2026

GABRIEL HENRIQUE MOREIRA ROCHA

DETECÇÃO AUTOMÁTICA DE REFATORAÇÕES: UM ESTUDO EXPLORATÓRIO UTILIZANDO LLMS

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como parte dos requisitos necessários para a obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Aline Norberta de Brito

Ouro Preto
2026

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE OURO PRETO
REITORIA
INSTITUTO DE CIENCIAS EXATAS E BIOLOGICAS
DEPARTAMENTO DE COMPUTACAO

FOLHA DE APROVAÇÃO

Gabriel Henrique Moreira Rocha

Detecção Automática de Refatorações: Um Estudo Exploratório Utilizando LLMs

Monografia apresentada ao Curso de Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para obtenção do título de Bacharel em Ciência da Computação

Aprovada em 23 de Julho de 2026.

Membros da banca

Aline Norberta de Brito (Orientadora) - Doutora - Universidade Federal de Ouro Preto
Saul Emanuel Delabrida Silva (Examinador) - Doutor - Universidade Federal de Ouro Preto
Ricardo de Sousa Job (Examinador) - Doutor - Instituto Federal de Educação, Ciência e Tecnologia da Paraíba

Aline Norberta de Brito, Orientadora do trabalho, aprovou a versão final e autorizou seu depósito na Biblioteca Digital de Trabalhos de Conclusão de Curso da UFOP em 23/07/2026.

Documento assinado eletronicamente por **Aline Norberta de Brito**, **PROFESSOR DE MAGISTERIO SUPERIOR**, em 26/07/2026, às 15:06, conforme horário oficial de Brasília, com fundamento no art. 6º, § 1º, do Decreto nº 8.539, de 8 de outubro de 2015.

A autenticidade deste documento pode ser conferida no site http://sei.ufop.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0 , informando o código verificador **1142654** e o código CRC **2C85FB92**.

R. Diogo de Vasconcelos, 122, - Bairro Pilar Ouro Preto/MG, CEP 35402-163
Telefone: 3135591692 - www.ufop.br

Agradecimentos

Agradeço, primeiramente, à minha família, que foi a minha base durante toda essa caminhada: sem ela, nada disso seria possível. Em especial, à minha mãe, Elaine Andreia Gonçalves Moreira da Rocha, ao meu pai, David Vieira da Rocha, e ao meu irmão, Vinicius Moreira da Rocha, que estiveram ao meu lado em todos os momentos, nas decisões difíceis e nas conquistas, mesmo com a distância.

À minha namorada, Laura Muschioni de Castro, pelo companheirismo, pela paciência e pelo apoio constante, principalmente nos períodos mais intensos deste trabalho.

Aos meus companheiros da República Nadavê, que fizeram de Ouro Preto uma segunda casa e foram parte essencial da minha formação, dentro e fora da sala de aula. Aos meus amigos de Belo Horizonte, que mesmo de longe se fizeram presentes durante toda a graduação, pelo incentivo e pela amizade de sempre.

À minha orientadora, Professora Aline Norberta de Brito, pela orientação dedicada, pela disponibilidade, pela paciência nas revisões e pela confiança depositada em mim ao longo de todo o desenvolvimento deste trabalho.

Ao Professor Saul Emanuel Delabrida Silva, um agradecimento especial por ter me introduzido no meio científico por meio do laboratório XR4GOOD, experiência que despertou meu interesse pela pesquisa e teve papel importante na minha trajetória acadêmica.

À banca examinadora, pela disponibilidade em avaliar este trabalho e pelas contribuições para a sua versão final.

Por fim, agradeço à Universidade Federal de Ouro Preto, ao Instituto de Ciências Exatas e Biológicas e ao Departamento de Computação, e a todos os professores e servidores que fizeram parte da minha formação, pela oportunidade de um ensino público, gratuito e de qualidade.

Resumo

A detecção de refatorações é uma atividade fundamental para a compreensão da evolução de sistemas de *software* e para a manutenção da qualidade do código-fonte. Ferramentas tradicionais para esse fim, como RefDiff e RefactoringMiner, apresentam alta precisão, porém dependem fortemente de análises sintáticas e estruturais, o que pode limitar sua aplicação em cenários mais complexos ou em múltiplas linguagens de programação. Nesse contexto, o avanço recente dos *Large Language Models* (LLMs) tem despertado interesse na Engenharia de Software, especialmente devido à sua capacidade de interpretar informações semânticas e contextuais em código-fonte. Este trabalho investiga a viabilidade do uso de LLMs na detecção automática de operações de refatoração em nível de método. Para isso, é proposto um método baseado em *prompt engineering*, no qual o LLM analisa versões anterior e posterior de trechos de código-fonte e identifica refatorações do tipo *Extract Method*, *Move Method* e *Rename Method*. A abordagem foi avaliada sobre um conjunto de 150 operações reais extraídas de um oráculo de refatorações validadas manualmente e amplamente utilizado na literatura, empregando dois LLMs de famílias distintas, o GPT-5.4 e o Claude Haiku 4.5, e confrontando os resultados com os reportados por ferramentas consolidadas no estado da arte. Os resultados revelam perfis complementares entre os LLMs: um deles favorece a precisão (0,73) e o outro, a cobertura (*recall* de 0,72), enquanto a combinação de suas saídas eleva o *recall* de detecção a 0,77. Observou-se ainda que as falhas decorrem majoritariamente de omissão, e não de confusão de tipo, e que a maioria dos falsos positivos corresponde a transformações reais no código. Embora o desempenho permaneça abaixo do das ferramentas dedicadas, os LLMs mostram-se uma técnica complementar promissora, com a vantagem de justificar cada detecção em linguagem natural e de prescindir de *parser* específico por linguagem. Por fim, o trabalho discute as limitações observadas e aponta direções para investigações futuras sobre a integração de LLMs em ferramentas de apoio à manutenção de *software*.

Palavras-chave: Refatoração. LLM. Detecção de refatorações. Prompt Engineering.

Abstract

Refactoring detection is a fundamental activity for understanding software evolution and maintaining source code quality. Traditional refactoring detection tools, such as RefDiff and RefactoringMiner, achieve high accuracy but rely heavily on syntactic and structural analyses, which may limit their applicability in more complex scenarios or across multiple programming languages. In this context, the recent advances in *Large Language Models* (LLMs) have attracted significant attention in Software Engineering due to their ability to capture semantic and contextual information from source code. This work investigates the feasibility of using LLMs for the automatic detection of method-level refactoring operations. A prompt-based approach is proposed, in which an LLM analyzes the differences between pre-change and post-change versions of source code and identifies refactorings such as *Extract Method*, *Move Method*, and *Rename Method*. The approach was evaluated on a set of 150 real refactoring operations drawn from a manually validated refactoring oracle widely adopted in the literature, using two LLMs from distinct families, one from the GPT family and Claude Haiku 4.5, and comparing the results with those reported by state-of-the-art tools. The results reveal complementary profiles between the LLMs: one favors precision (0.73) and the other favors coverage (0.72 recall), while combining their outputs raises the detection recall to 0.77. Moreover, failures stem mostly from omission rather than type confusion, and most false positives correspond to real code transformations. Although performance remains below that of dedicated tools, LLMs prove to be a promising complementary technique, with the advantage of justifying each detection in natural language and not requiring a language-specific parser. Finally, this work discusses the observed limitations and outlines directions for future research on the integration of LLMs into software maintenance tools.

Keywords: Refactoring. LLM. Refactoring detection. Prompt Engineering.

Lista de Figuras

Listings

Lista de Abreviaturas e Siglas

ABNT	Associação Brasileira de Normas Técnicas
DECOM	Departamento de Computação
UFOP	Universidade Federal de Ouro Preto
LLM	Large Language Model
NLP	Processamento de Linguagem Natural
AST	Árvore de Sintaxe Abstrata
API	Application Programming Interface
JSON	JavaScript Object Notation
TCC	Trabalho de Conclusão de Curso
GPT	Generative Pre-trained Transformer

Sumário

1 Introdução

A refatoração é uma prática fundamental no desenvolvimento de *software*, visando melhorar a legibilidade, e facilitar a correção de defeitos e adição de novas funcionalidades (SILVA; TSANTALIS; VALENTE, 2016; VALENTE, 2020). Trata-se de uma atividade recorrente na manutenção e na evolução de sistemas, na qual a estrutura interna do código é alterada sem que o seu comportamento externo seja modificado.

Dessa forma, ferramentas têm sido desenvolvidas para auxiliar os desenvolvedores neste processo. Dentre estas ferramentas e técnicas modernas, destacam-se as ferramentas para detecção de operações de refatoração, como RefactoringMiner (TSANTALIS et al., 2018; TSANTALIS; KETKAR; DIG, 2020) e RefDiff (SILVA; VALENTE, 2017; SILVA et al., 2021; BRITO; VALENTE, 2020). Ao identificar automaticamente que uma refatoração ocorreu, essas ferramentas apoiam a revisão de código, a navegação pelo histórico de versões e os estudos empíricos sobre evolução de *software*.

Embora elas apresentem ótimos resultados, existem limitações importantes em cenários reais. Por exemplo, elas requerem a implementação de um *parser* específico por linguagem de programação, demandando esforço e tempo. Essa dependência restringe a sua aplicação a um conjunto limitado de linguagens e torna custosa a extensão para novos contextos.

Nesse contexto, abordagens baseadas em LLMs surgem como alternativa promissora, capazes de capturar intenção e significado em alterações de código (SILVA et al., 2024), além de facilitar a portabilidade para outras linguagens de programação (ZHONG; WANG, 2024).

O avanço recente dos *Large Language Models* (LLMs) tem redefinido o cenário da Engenharia de Software, ampliando o alcance de tarefas automatizadas relacionadas à compreensão e geração de código. Estudos recentes demonstram que LLMs têm alcançado alto desempenho em atividades como sumarização, explicação de trechos e interpretação de mudanças (ZAN et al., 2023; MELO; ALMEIDA; GARCéS, 2025; AMARAL et al., 2025; SILVA et al., 2024). Portanto, isso sugere um potencial para apoiar processos tradicionalmente complexos no desenvolvimento e análise de *software*.

Este trabalho de conclusão de curso propõe e avalia um método baseado em LLM para identificar e classificar operações de refatoração a partir de mudanças no código. O objetivo é comparar sua efetividade com ferramentas estáticas, analisando sua capacidade de interpretação semântica e precisão na detecção das refatorações. Assim, o objetivo é compreender se LLMs podem complementar ou superar abordagens estruturais existentes, contribuindo para o avanço das ferramentas modernas para manutenção e evolução de *software*.

O método foi avaliado sobre um conjunto de 150 operações de refatoração reais, extraídas

de um oráculo de refatorações validadas por especialistas (TSANTALIS et al., 2022). Considera-se as refatorações *Extract Method*, *Move Method* e *Rename Method*, empregando dois LLMs de famílias distintas, um da família GPT e o Claude Haiku 4.5.

Os resultados evidenciam perfis complementares entre os LLMs: enquanto um modelo favorece a precisão, o outro apresenta maior cobertura, e a combinação de suas saídas eleva o *recall* de detecção acima do obtido por qualquer um deles isoladamente. A principal contribuição deste trabalho é fornecer evidências empíricas de que os LLMs podem constituir uma técnica complementar para a detecção de operações de refatoração. Além disso, no melhor do nosso conhecimento, este é o primeiro estudo a investigar o uso de LLMs na detecção automática de operações de refatoração sob essa perspectiva.

1.1 Justificativa

Nos últimos anos, LLMs têm se tornado um componente chave na Engenharia de Software contemporânea, impulsionando avanços significativos, como detecção de *code smells* (ZHENG et al., 2025; SILVA et al., 2024), migração entre bibliotecas (ALMEIDA; XAVIER; VALENTE, 2024), e geração de testes (SIDDIQ et al., 2024). LLMs das famílias GPT e Claude têm se mostrado extremamente avançados e competentes quanto às suas capacidades de compreensão e geração de código, tendo assim, ganhado grande visibilidade (AQUINO et al., 2025).

Dada a ótima performance dessas ferramentas, o foco em explorar seu potencial em tarefas de engenharia de *software* tem aumentado significativamente, buscando novas oportunidades de solucionar desafios desse campo (ZAN et al., 2023). Esse cenário reforça a necessidade de investigações que avaliem o papel dos LLMs em tarefas mais sofisticadas e menos exploradas, como a análise automatizada de refatorações.

Apesar dos avanços significativos em ferramentas como RefactoringMiner e RefDiff, a literatura aponta limitações estruturais que dificultam a detecção precisa de refatorações em cenários reais. Tsantalis et al. (2018), por exemplo, afirmam que essas ferramentas baseadas majoritariamente em regras sintáticas e análise estática apresentam dificuldades frente a mudanças complexas, múltiplas operações encadeadas e modificações que envolvem interpretação semântica do código. Tais limitações indicam uma lacuna importante no estado da arte e justificam a busca por novas abordagens capazes de capturar aspectos semânticos e contextuais ausentes em técnicas tradicionais.

Diante desse contexto, integrar LLMs à mecanismos de detecção de refatorações representa uma oportunidade promissora para superar as limitações atuais e avançar o estado da arte. Assim, o trabalho atual busca investigar e avaliar se LLMs podem atuar como uma alternativa viável e confiável na identificação automatizada de operações de refatoração.

1.2 Objetivos

Esta seção descreve os objetivos geral e específicos, necessários para a conclusão do trabalho proposto.

1.2.1 Objetivo Geral

Considerando a importância de ferramentas para detecção de refatoração e os desafios existentes no desenvolvimento de técnicas baseadas na análise estática do código, pretende-se, neste trabalho, atingir o seguinte objetivo geral:

> Analisar a eficácia de LLMs na detecção automática de refatorações, avaliando a sua capacidade de identificar e classificar diferentes tipos de operações.

1.2.2 Objetivos Específicos

Com base em nosso objetivo geral, os seguintes objetivos específicos são propostos:

- Investigar técnicas recentes para análise de código via LLMs, com foco em qualidade de *software* e refatorações.
- Desenvolver um método baseado em LLM para detecção e classificação de refatorações em Java, visando explorar a capacidade de LLMs para analisar mudanças no código.
- Discutir a efetividade do método proposto para detectar refatorações, visando identificar oportunidades de melhorias e limitações.

1.3 Organização do Trabalho

O restante deste documento está organizado da seguinte forma: o Capítulo 2 inclui detalhes técnicos sobre os conceitos de LLM, modelos baseados na arquitetura *Transformer*, refatorações discutidas na literatura e ferramentas consolidadas para análise de refatorações. O Capítulo 3 apresenta a metodologia adotada, descrevendo o processo de detecção de refatorações via LLMs e os critérios utilizados para avaliação dos resultados. O Capítulo 4 reúne e discute os resultados obtidos com a aplicação do método proposto, comparando-os com os das ferramentas consolidadas de detecção. Por fim, o Capítulo 5 apresenta as considerações finais e as propostas de trabalhos futuros.

2 Revisão Bibliográfica

Este capítulo apresenta a revisão bibliográfica sobre os assuntos abordados no documento e os trabalhos relacionados ao tema escolhido. A Seção 2.1 apresenta o referencial teórico, com os conceitos de *Large Language Models* e de refatoração necessários à compreensão do estudo, enquanto a Seção 2.2 descreve os trabalhos relacionados, abrangendo as ferramentas consolidadas de detecção de refatorações e as abordagens recentes baseadas em LLMs.

2.1 Fundamentação Teórica

Nesta seção são apresentados os conceitos relacionados a Large Language Models (LLMs) e Refatoração na literatura.

2.1.1 Large Language Models (LLM)

Large Language Models (LLMs) constituem uma classe de modelos de Inteligência Artificial baseados em técnicas de aprendizado profundo, treinados com grandes volumes de dados textuais, normalmente extraídos da internet (ALBERTS et al., 2023). O objetivo de um LLM é ser capaz de compreender e processar a linguagem natural humana de maneira contextual e coerente (SALLAM, 2023).

Um LLM eficaz deve apresentar quatro características-chave (YAO et al., 2024): (i) profunda compreensão do contexto da linguagem natural; (ii) habilidade para gerar textos semelhantes aos produzidos por humanos; (iii) entendimento de contextos especializados, especialmente em domínios que exigem alto nível de conhecimento; (iv) elevada capacidade de seguir instruções complexas, o que é útil para a resolução de problemas e a tomada de decisões.

Para atingir esse objetivo, o LLM aprende padrões e associações entre palavras durante o treinamento, de modo a prever a próxima palavra na sequência. Essa capacidade probabilística permite que o modelo gere saídas coerentes. Embora modelos probabilísticos existam há décadas, os LLMs revolucionaram o campo de Processamento de Linguagem Natural (NLP) (BHAYANA, 2024) ao empregar a arquitetura baseada em atenção, capaz de capturar dependências de longo alcance e contextos complexos com alta eficiência, conhecida como *Transformer* (VASWANI et al., 2017).

A arquitetura *Transformer* oferece representações contextuais robustas e elimina a necessidade de mecanismos recorrentes ou convolucionais durante o treinamento (OTTER; MEDINA; KALITA, 2020). Composta por blocos empilhados de *encoders* e *decoders* com múltiplos mecanismos de autoatenção, ela se tornou a base de diversos modelos que definem o estado da arte em aprendizado profundo.

Entre os exemplos mais notáveis está o ChatGPT, desenvolvido pela OpenAI, amplamente reconhecido por sua capacidade de compreender e gerar linguagem natural de maneira contextual e adaptável (TAECHARUNGROJ, 2023). Modelos como esse têm extrapolado o domínio do texto, alcançando aplicações relevantes no campo da Engenharia de Software, especialmente em tarefas de manutenção e evolução de código (VAITHILINGAM; ZHANG; GLASSMAN, 2022; ZHENG et al., 2025; SILVA et al., 2024) e no âmbito educacional (AQUINO et al., 2025).

Outro exemplo de destaque é o Claude, família de LLMs desenvolvida pela Anthropic e treinada com ênfase em segurança e alinhamento por meio da abordagem de *Constitutional AI*, na qual o comportamento do modelo é guiado por um conjunto explícito de princípios durante o ajuste, reduzindo a dependência de rótulos humanos para conteúdos indesejados (BAI et al., 2022). A família organiza-se em variantes com diferentes compromissos entre capacidade, custo e latência, entre elas a variante Haiku, voltada a maior velocidade e menor custo de inferência (Anthropic, 2024). Assim como o ChatGPT, o Claude tem sido avaliado de forma recorrente em tarefas de compreensão e geração de código (ZAN et al., 2023; ZHONG; WANG, 2024), o que motiva sua adoção em estudos empíricos de Engenharia de Software.

O comportamento de um LLM na geração de texto é regulado por um conjunto de parâmetros, entre os quais se destaca a temperatura. Como a geração é um processo probabilístico, no qual o próximo *token* é amostrado a partir de uma distribuição de probabilidade sobre o vocabulário, a temperatura atua justamente sobre essa distribuição, tornando-a mais concentrada ou mais dispersa. Valores baixos aproximam a seleção do *token* mais provável, produzindo saídas mais determinísticas e conservadoras, ao passo que valores altos aumentam a diversidade e a variabilidade das respostas, com maior risco de inconsistências e alucinações (LI et al., 2025). Por esse motivo, tarefas que exigem respostas mais estáveis e reprodutíveis, como a análise automatizada de código, costumam adotar temperaturas baixas, decisão retomada e justificada na metodologia deste trabalho (Capítulo 3).

2.1.2 Refatoração

Refatorações consistem em alterações no código que não modificam o comportamento externo do sistema (VALENTE, 2020; FOWLER, 2018; FOWLER, 1999). Desenvolvedores refatoram o código por diversos motivos, como por exemplo, para remoção de duplicações, melhorar a legibilidade, simplificação do *design* e aumento da flexibilidade estrutural (WEBER et al., 2011). A ideia central consiste em redistribuir classes, variáveis e métodos dentro da hierarquia de classes, facilitando futuras adaptações e extensões do sistema (MENS; TOURWÉ, 2004). A literatura recente mostra que existem também refatorações oportunísticas (SILVA; TSANTALIS; VALENTE, 2016), que ocorrem quando o desenvolvedor refatora o código enquanto está adicionando novas funcionalidades ou corrigindo bugs.

Fowler (1999) em seu livro cataloga 72 tipos distintos de operações de refatoração em Java, sendo atualizado recentemente considerando a linguagem JavaScript (FOWLER, 2018). Existem

operações complexas envolvendo movimentação de partes do código, assim como operações triviais que geram pouco impacto.

As Figuras 2.1, 2.2 e 2.3 ilustram, respectivamente, as operações de refatoração em que um bloco de código é extraído para uma função mais geral e reutilizável, uma operação em que um método é apenas movido de local dentro do código, e, por fim, uma operação na qual um método tem apenas o seu nome alterado. Todas essas operações são classificadas como refatorações em nível de método e estão entre as mais comumente aplicadas na prática.

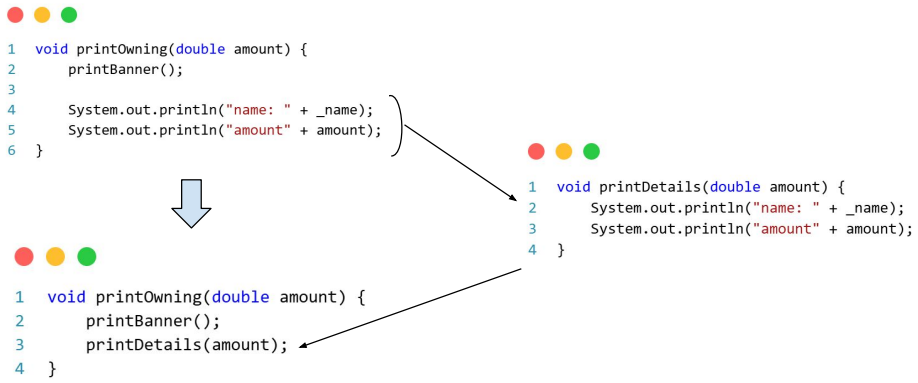


Figura 2.1 – Exemplo de *Extract Method* (Fowler (1999))

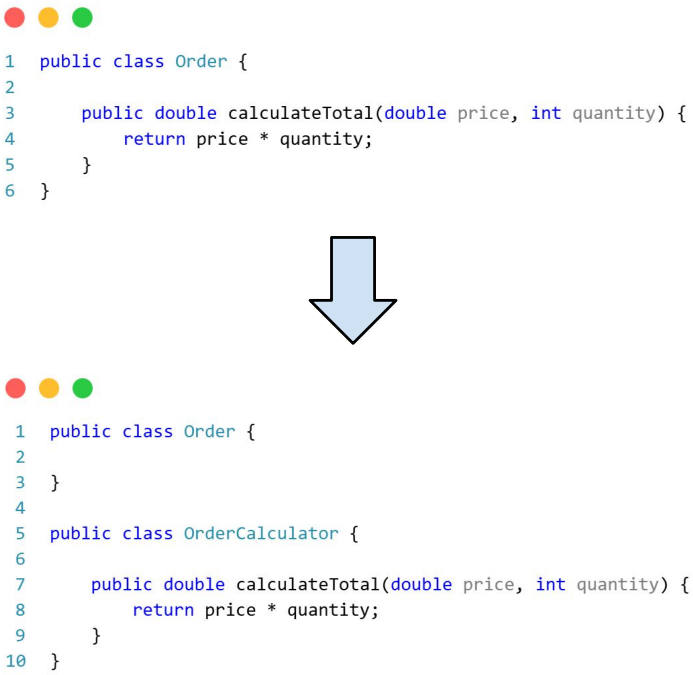


Figura 2.2 – Exemplo sintético da operação de *Move Method*.

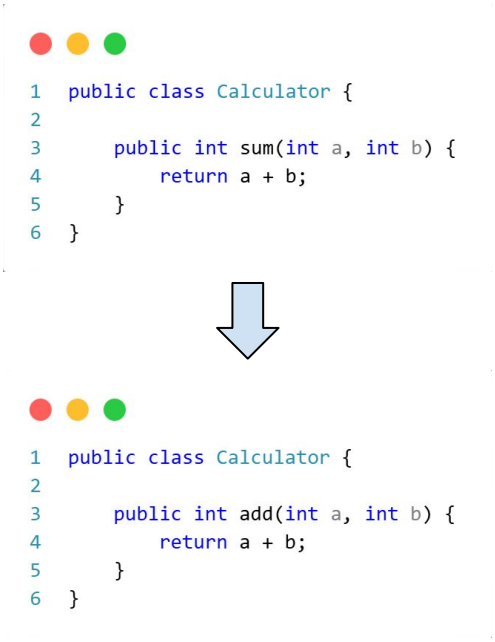


Figura 2.3 – Exemplo sintético da operação de *Rename Method*.

A refatoração é um processo importante na construção de sistemas de software de qualidade. Conforme a escala e a complexidade de um projeto cresce, a ausência de refatorações periódicas tende a provocar a deterioração da qualidade interna do código (PANTIUCHINA et al., 2020). Trata-se, portanto, de uma prática indispensável para preservar o *design*, a compreensibilidade e a manutenibilidade do sistema, evitando o acúmulo de dívida técnica e garantindo um desenvolvimento sustentável e contínuo (FOWLER, 2018; FOWLER, 1999; VALENTE, 2020).

2.2 Trabalhos Relacionados

A detecção de operações de refatoração permite o desenvolvimento de ferramentas e técnicas modernas. Por exemplo, a identificação de refatorações pode auxiliar no processo de revisão de código, melhorando a visualização do *diff* textual em plataformas como o GitHub (BRITO; VALENTE, 2021), e na navegação pelo histórico do código-fonte (BRITO et al., 2023; BRITO; HORA; VALENTE, 2021).

2.2.1 Ferramentas para Detecção de Refatoração

Neste contexto, o RefDiff é uma ferramenta automatizada para detecção de refatorações, que analisa o histórico de versões de um sistema de *software*. Em sua versão original, voltada à linguagem Java, aplica combinações de heurísticas baseadas em análises estatísticas e em similaridade de código para identificar 13 tipos de refatorações definidas na literatura, alcançando precisão de 100% e *recall* de 88% (SILVA; VALENTE, 2017). Além disso, existe a RefDiff4Go, uma extensão desenvolvida para projetos implementados na linguagem Go, capaz de detectar

13 tipos de atividades de refatoração, que obteve precisão de 92% e *recall* de 80% (BRITO; VALENTE, 2020).

Posteriormente, foi proposta a RefDiff 2.0, uma evolução da ferramenta concebida para a detecção de refatorações em múltiplas linguagens de programação (SILVA et al., 2021). Sua principal contribuição é um modelo intermediário e independente de linguagem, denominado *Code Structure Tree* (CST), que representa de forma uniforme os elementos estruturais do código-fonte, como classes, funções e blocos. Com essa abstração, o algoritmo de detecção permanece o mesmo para todas as linguagens, e o suporte a uma nova linguagem passa a exigir apenas a implementação de um componente que traduza o seu código para o CST, em vez da reescrita de todo o mecanismo de detecção. Na versão 2.0, a ferramenta oferece suporte às linguagens Java, JavaScript e C, mantendo altos níveis de precisão e *recall* na detecção. Ainda assim, mesmo com esse projeto voltado à extensibilidade, a inclusão de cada nova linguagem continua a depender da construção de um *parser* próprio e do respectivo componente de tradução.

Em 2018 foi proposta a ferramenta RefactoringMiner, que funciona com base em um algoritmo de correspondência de instruções baseado em Árvore de Sintaxe Abstrata (AST). A ferramenta oferece alta precisão, eficiência e escalabilidade na detecção de refatorações ao longo de históricos de *commits* de um projeto, sem a necessidade de reconstruir cada *commit* individualmente (TSANTALIS et al., 2018).

Dois anos depois foi proposto o RefactoringMiner 2.0, uma versão aprimorada da ferramenta, com suporte à detecção em nível de submétodo, um diferencial muito grande em relação a grande parte das ferramentas existentes. Além disso, a RefactoringMiner 2.0 apresenta características únicas, como: não depender de limites de similaridade de código, oferecer suporte a refatorações de baixo nível que ocorrem dentro do corpo dos métodos, e é capaz de detectar operações de refatoração aninhadas em um único *commit*. Nos experimentos realizados, foi atingida uma precisão de 99,6% e *recall* de 94%, sendo em média 2,6 vezes mais rápida que a segunda ferramenta mais eficiente de detecção de refatorações (TSANTALIS; KETKAR; DIG, 2020).

Entretanto, conforme mencionado anteriormente, mesmo as ferramentas projetadas para múltiplas linguagens ainda requerem um esforço significativo para serem estendidas a novas linguagens, sendo necessário analisar construções específicas de cada uma e implementar *parsers* próprios.

2.2.2 Abordagens Baseadas em LLMs

Paralelamente à evolução dessas ferramentas, os LLMs vêm sendo aplicados a uma variedade crescente de tarefas de Engenharia de Software que exigem interpretação semântica do código, com resultados promissores. Trabalhos recentes exploram seu uso na detecção de *code smells* (SILVA et al., 2024), na migração entre bibliotecas (ALMEIDA; XAVIER; VALENTE,

2024), na geração automatizada de testes (SIDDIQ et al., 2024) e na avaliação da qualidade de código gerado (AQUINO et al., 2025). Esses estudos indicam que os LLMs são capazes de capturar intenção e contexto em alterações de código, aspectos difíceis de tratar por heurísticas puramente sintáticas. No campo específico da refatoração, Cordeiro, Noei e Zou (2025) avaliaram empiricamente a capacidade de LLMs em realizar operações de refatoração em projetos Java, obtendo reduções de *code smells* superiores às de desenvolvedores em determinadas categorias. Tais esforços, contudo, concentram-se em aplicar refatorações, ao passo que a detecção e a classificação automáticas de operações já realizadas, tarefa deste trabalho, permanecem pouco exploradas com LLMs. Até onde temos conhecimento, portanto, e somando-se ao fato de que essas estratégias dispensam o ferramental específico por linguagem, essa lacuna motiva o presente trabalho.

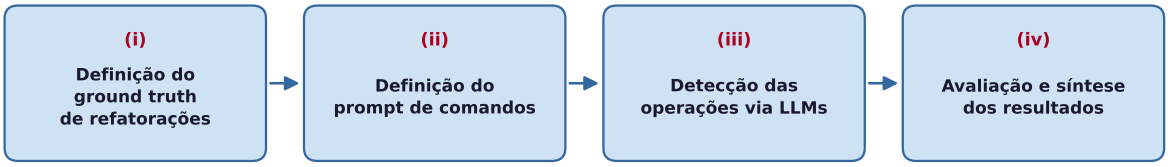
3 Metodologia

3.1 Visão Geral da Metodologia

Este trabalho adota uma abordagem experimental e aplicada, com o objetivo de avaliar a utilização de LLMs na detecção e classificação de operações de refatoração. A abordagem combina aspectos quantitativos, relacionados à avaliação dos resultados focada em métricas, e qualitativos, associados à interpretação semântica das refatorações identificadas.

A Figura 3.1 mostra a visão geral da metodologia empregada neste estudo, que inclui quatro etapas principais: (i) definição do *ground truth* de refatorações, a partir de operações validadas de um oráculo consolidado na literatura; (ii) definição do *prompt* de comandos para detecção de refatorações; (iii) detecção das operações de refatoração via LLMs; e (iv) avaliação e síntese dos resultados. As próximas seções detalham cada uma dessas etapas. A detecção (etapa iii) foi conduzida com dois LLMs distintos, analisados de forma independente sobre exatamente o mesmo conjunto de casos, o que permite, além da avaliação individual, uma comparação direta entre eles (Capítulo 4).

Figura 3.1 – Visão geral da metodologia.



Fonte: elaborada pelo autor.

3.2 Definição do *ground truth* de refatorações

Para a avaliação deste estudo, utilizou-se um oráculo de refatorações amplamente adotado na literatura, composto por operações reais realizadas por desenvolvedores em projetos GitHub e posteriormente validadas manualmente por Tsantalis e colaboradores (TSANTALIS et al., 2022). Esse oráculo é frequentemente utilizado como *ground truth* na avaliação de ferramentas de detecção de refatorações (SILVA; VALENTE, 2017; TSANTALIS; KETKAR; DIG, 2020).

Os *commits* do oráculo provêm de projetos reais e podem conter uma ou mais operações de refatoração em um mesmo *commit*, refletindo cenários práticos de evolução de código-fonte. Eles vêm principalmente de projetos Java *open-source* cujas operações foram validadas manualmente segundo o catálogo de Fowler (1999, 2018). O escopo deste trabalho considera as operações *Extract Method*, *Move Method* e *Rename Method*. A escolha dessas operações deve-se à sua

frequência e relevância das mesmas em atividades de manutenção e evolução de software (SILVA; TSANTALIS; VALENTE, 2016; VALENTE, 2020; MURPHY-HILL; PARNIN; BLACK, 2011).

A partir desse oráculo, extraiu-se um subconjunto experimental segundo quatro critérios: (i) inclusão exclusiva de operações validadas como *True Positive* (TP), de modo a garantir maior confiabilidade experimental e evitar que falsos positivos do próprio oráculo comprometessem a análise; (ii) distribuição equilibrada entre os três tipos analisados; (iii) inclusão de 50 operações de cada tipo; e (iv) priorização da diversidade de projetos e *commits*, evitando repetições excessivas de um mesmo repositório. O conjunto final reúne 150 operações de refatoração, distribuídas em 150 *commits* (um para cada operação) pertencentes a 81 projetos distintos, com a distribuição por tipo apresentada na Tabela 3.1.

Tabela 3.1 – Distribuição das operações de refatoração no conjunto experimental.

Tipo de refatoração	**Operações (TP)**
Extract Method	50
Move Method	50
Rename Method	50
Total	**150**

Fonte: elaborada pelo autor.

Para ancorar a discussão em código real, os parágrafos a seguir ilustram cada um dos três tipos com um exemplo extraído do oráculo.

Extract Method. Esta operação isola parte da lógica de um método em um novo método com responsabilidade específica, preservando o comportamento externo. O Código 3.1, do projeto crashub/crash,[1] mostra a conversão de um `PEMKeyPair` em `KeyPair`, antes embutida no laço do método `loadKeys()`, sendo extraída no novo método `convertPemKeyPair`, que passa a ser invocado em seu lugar.

Listing 3.1 – Exemplo de Extract Method (crashub/crash).

```
// Antes
} else if (o instanceof PEMKeyPair) {
    PEMKeyPair keyPair = (PEMKeyPair) o;
    JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
    keys.add(new KeyPair(converter.getPublicKey(keyPair.getPublicKeyInfo()), null));
}

// Depois
} else if (o instanceof PEMKeyPair) {
    keys.add(convertPemKeyPair((PEMKeyPair) o));
}

private KeyPair convertPemKeyPair(PEMKeyPair pemKeyPair) throws PEMException {
    JcaPEMKeyConverter converter = new JcaPEMKeyConverter();
    return new KeyPair(converter.getPublicKey(pemKeyPair.getPublicKeyInfo()), null);
```

[1] Disponível em: <https://github.com/crashub/crash/commit/2801269c7e47bd6e243612654a74cee809d20959>.

```
}
```

Rename Method. Esta refatoração altera o nome de um método, preservando sua assinatura e seu comportamento. O Código 3.2, do projeto GoClipse,[2] traz o caso mais puro possível: a correção de uma digitação no nome do método (`getInvocationOffest` para `getInvocationOffset`), sem qualquer outra mudança.

Listing 3.2 – Exemplo de Rename Method (GoClipse).

```
// Antes
public int getInvocationOffest() {
    return context.getInvocationOffset();
}

// Depois
public int getInvocationOffset() {
    return context.getInvocationOffset();
}
```

Move Method. Por fim, *Move Method* desloca um método de uma classe para outra, sendo o único dos três que envolve dois arquivos. No Código 3.3, do projeto apache/drill,[3] o método `executeQuery` deixa a classe `HiveTestDataGenerator` e passa a residir em `HiveTestUtilities`, onde é promovido a público e estático para ser reutilizado por outras classes de teste.

Listing 3.3 – Exemplo de Move Method (apache/drill).

```
// Antes (classe HiveTestDataGenerator)
private void executeQuery(Driver hiveDriver, String query) {
    CommandProcessorResponse response = null;
    // ...
    response = hiveDriver.run(query);
    // ...
}

// Depois (classe HiveTestUtilities)
public static void executeQuery(Driver hiveDriver, String query) {
    CommandProcessorResponse response = null;
    // ...
    response = hiveDriver.run(query);
    // ...
}
```

Diferentemente do oráculo original, que registra apenas a descrição textual de cada refatoração, a abordagem proposta requer o código-fonte completo das versões anterior e posterior à modificação. Assim, cada operação foi enriquecida por meio de um *script* que, a partir da descrição da refatoração, infere o caminho do arquivo afetado e recupera, via API do GitHub, o conteúdo correspondente nos *snapshots* do *commit* e de seu *commit* pai. Quando o arquivo da classe foi renomeado ou movido de pacote no próprio *commit*, o caminho derivado da descrição

[2] Disponível em: <https://github.com/GoClipse/goclipse/commit/851ab757698304e9d8d4ae24ab75be619ddae31a>.
[3] Disponível em: <https://github.com/apache/drill/commit/c1b847acdc8cb90a1498b236b3bb5c81ca75c044>.

(que reflete o estado posterior) não existe no *commit* pai; nesses casos, o *script* recupera a versão anterior a partir dos metadados de renomeação do *commit*, garantindo a recuperação do código correto.

Ao final do processo, todas as 150 operações dispõem de ambas as versões de código (anterior e posterior), sem qualquer caso de arquivo irrecuperável. Para Extract Method e Rename Method, que envolvem um único arquivo, todas as 50 operações de cada tipo tiveram as duas versões recuperadas integralmente. Para Move Method, que envolve dois arquivos (a classe de origem e a classe de destino), 34 das 50 operações têm ambos os arquivos integralmente presentes nos dois *snapshots*. Nas 16 operações restantes, um dos arquivos não existe em um dos *snapshots* por uma razão intrínseca à própria refatoração: em 13 casos a classe de destino é criada no *commit* (não existia na versão anterior) e em 5 casos a classe de origem é removida após a movimentação do método. Esses casos não representam dados ausentes ou degradados; o estado real do arquivo (criado ou removido) é registrado explicitamente por um marcador textual fornecido ao LLM, que assim recebe a informação da evolução do código. A Tabela 3.2 detalha esses números.

Tabela 3.2 – Disponibilidade do código-fonte por tipo de refatoração.

Tipo	**Total**	**Ambos arquivos**	**Classe criada/removida**	**Irrecuperáveis**
Extract Method	50	50	0	0
Rename Method	50	50	0	0
Move Method	50	34	16	0
Total	**150**	**134**	**16**	**0**

Fonte: elaborada pelo autor.

Dessa forma, as 150 operações foram submetidas aos LLMs com informação de código completa e fiel ao estado real de cada *snapshot*. A criação ou remoção de classes no Move Method, longe de ser uma limitação de coleta, constitui um sinal estrutural característico dessa refatoração e é apresentada ao LLM como tal.

Para a construção do conjunto, foram desenvolvidos *scripts* auxiliares responsáveis por: (i) filtrar, no oráculo, os *commits* que continham operações compatíveis com os critérios definidos; (ii) selecionar e reordenar o subconjunto experimental; e (iii) coletar metadados adicionais dos repositórios envolvidos, como número de *stars*, *forks* e linguagem predominante, utilizados para aferir a relevância e a maturidade dos projetos.

3.2.1 Caracterização do conjunto experimental

A diversidade de projetos foi preservada: dos 81 repositórios, 58 (aproximadamente 72%) contribuem com um único *commit*, ao passo que apenas um pequeno número de projetos de maior porte concentra mais ocorrências, como `facebook/buck` (11 *commits*), `apache/hive` (9), `JetBrains/intellij-community` (8), `apache/cassandra` (7) e `hazelcast/hazelcast`

(7). As operações são oriundas de 141 autores distintos, reforçando a heterogeneidade das modificações analisadas.

A relevância dos projetos foi aferida pelo número de *stars* no GitHub, apresentada na Figura 3.2c: a mediana é de 5.013 *stars* por repositório, com o terceiro quartil em 8.947 e diversos projetos de altíssima visibilidade na cauda superior, como `spring-projects/spring-boot` (80.860 *stars*), `elastic/elasticsearch` (76.937), `spring-projects/spring-framework` (60.012) e `google/guava` (51.475). Os repositórios são predominantemente escritos em Java. Essa característica sugere maior maturidade do código-fonte, manutenção contínua e relevância prática das refatorações selecionadas.

A Figura 3.2 sintetiza ainda outras duas dimensões do conjunto. O *boxplot* a mostra o número de arquivos alterados por *commit*: a mediana é de 6 arquivos, mas a distribuição é fortemente assimétrica, com o terceiro quartil em 17 arquivos e *commits* extremos que modificam mais de uma centena (até 300) de arquivos. O *boxplot* b mostra o número de operações de refatoração dos três tipos estudados, confirmadas como TP no oráculo, presentes em cada um dos 150 *commits*: embora cada *commit* contribua com uma única operação rotulada, a mediana de operações desses tipos efetivamente presentes é de 1, com terceiro quartil em 3 e *commits* que chegam a conter até 13. De fato, 69 dos 150 *commits* (46%) contêm mais de uma operação validada como TP dentre os três tipos estudados. Considerando todos os tipos de refatoração catalogados pelo oráculo (não apenas os três estudados), os *commits* selecionados contêm uma mediana de 9 operações confirmadas, com média de 20,3 e máximo de 507.

A Tabela 3.3 reúne as estatísticas dessas três dimensões. Como a amplitude entre elas é muito distinta, a escala do eixo vertical na Figura 3.2 foi definida por dimensão: logarítmica para arquivos por *commit* e *stars* por repositório, cujos valores variam duas ordens de grandeza, e linear para operações-alvo por *commit*, cuja faixa vai de 1 a 13 e ficaria comprimida contra o eixo em escala logarítmica.

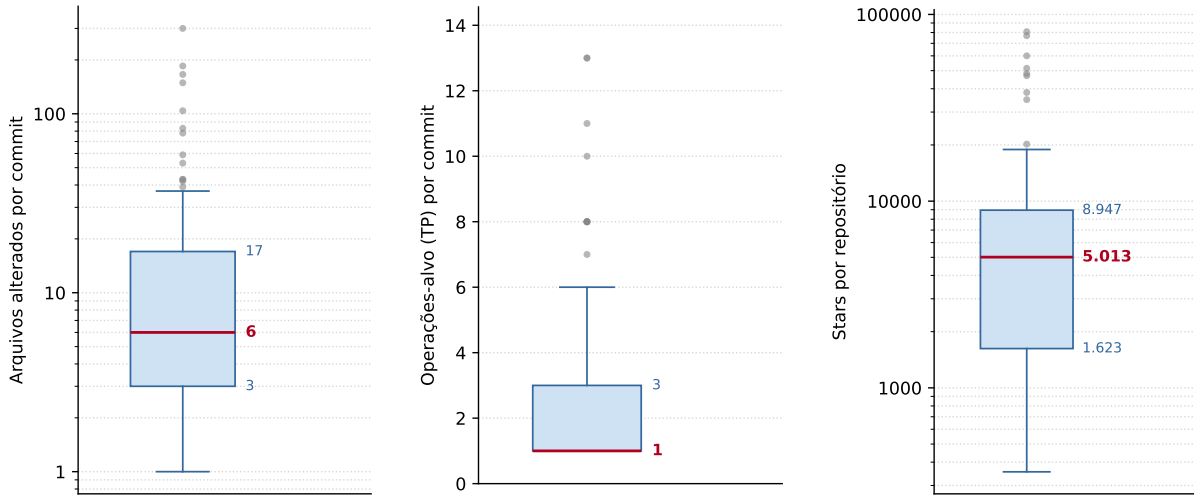
Tabela 3.3 – Estatísticas descritivas do conjunto experimental.

Dimensão	**Mínimo**	**1º quartil**	**Mediana**	**3º quartil**	**Máximo**
Arquivos alterados por *commit*	1	3	6	17	300
Operações-alvo por *commit*	1	1	1	3	13
Stars por repositório	355	1.623	5.013	8.947	80.860

Fonte: elaborada pelo autor.

Esse último dado é central para a interpretação dos resultados: o recorte selecionado rotula uma operação por *commit*, ainda que o *commit* contenha, com frequência, várias outras refatorações legítimas. Trata-se de uma decisão deliberada de amostragem, tomada para balancear os três tipos, e não de uma propriedade do oráculo original. É também o que motiva a adoção de duas formulações complementares de *recall* no protocolo de avaliação, apresentadas na Seção 3.5.

Figura 3.2 – Caracterização do conjunto experimental. A mediana e os quartis estão indicados em cada *boxplot*; o eixo vertical está em escala logarítmica nos painéis (a) e (c), e linear no painel (b).



(a) Arquivos por *commit*. (b) Operações-alvo por *commit*. (c) *Stars* por repositório.

Fonte: elaborada pelo autor.

3.3 Definição do prompt de comandos para detecção de refatorações

O *prompt* enviado ao LLM é estruturado em sete componentes principais (CAMPOS, 2026), compreendendo contexto, persona, tarefa, requisitos, restrições, formato de entrada, saída, e ação. Cada um deles é detalhado a seguir.

Componente I: Definição do contexto. O Código 3.4 apresenta a primeira parte do *prompt*, na qual se define o `#CONTEXT`, isto é, fornecemos ao LLM as informações necessárias para compreender a tarefa, dando um contexto para ele da situação. É informado que serão apresentados os dois trechos de código (antes e depois da refatoração). A seção também define o objetivo da tarefa, que consiste em verificar se as mudanças referem-se a uma operação de refatoração segundo as definições de Fowler (1999, 2018).

Listing 3.4 – Prompt para detecção de refatorações (contexto).

```
#CONTEXT

    Two source code snapshots are provided: a BEFORE version and an AFTER version of a
    transformation applied to one or more Java source files. The goal is to determine whether
    the transformation constitutes a refactoring according to Martin Fowler's formal definitions
    .
```

Componente II: Persona. Na segunda parte do prompt, define-se a `#PERSONA` do LLM, sendo isso o papel que ele irá desempenhar. Neste caso, o prompt define um especialista em Engenharia de Software com conhecimento em evolução de código-fonte e no catálogo de refatorações de

Fowler (1999, 2018), como pode-se observar no Código 3.5.

Listing 3.5 – Prompt para detecção de refatorações (persona).

```
#PERSONA

    Act as a Software Engineering expert specializing in source code evolution, with deep
    knowledge of Martin Fowler's formal refactoring catalogue.
```

Componente III: Tarefa. A seção `#TASK` define a ação principal que o LLM deve executar, conforme apresentado no Código 3.6. Neste estudo, a tarefa consiste em identificar quais operações de refatoração, se houver, foram aplicadas na transformação do código `BEFORE` para o código `AFTER`.

Listing 3.6 – Prompt para detecção de refatorações (tarefa).

```
#TASK

    Identify which refactoring operations, if any, were applied when transforming the BEFORE
    code into the AFTER code.
```

Componente IV: Requisitos. A seção `#REQUIREMENTS` fornece ao LLM a base conceitual da tarefa: a definição formal de refatoração e a descrição dos três tipos de interesse, todas extraídas de Fowler (1999, 2018). O Código 3.7 apresenta uma visão geral da definição de refatoração e a definição formal de cada tipo. Para cada um, o *prompt* completo inclui ainda a motivação, a mecânica e um exemplo sintético, reproduzidos no Apêndice A.

Listing 3.7 – Prompt para detecção de refatorações (requisitos).

```
#REQUIREMENTS

Formal definition of refactoring (Fowler, 1999):
"Refactoring is the process of changing a
software system in such a way that it does not
alter the external behavior of the code yet
improves its internal structure."

Recognized refactoring types, their formal
definitions, mechanics, and structural signals (...)

1. EXTRACT METHOD
[Definition], [Mechanics], [Synthetic example]

2. RENAME METHOD
[Definition], [Mechanics], [Synthetic example]

3. MOVE METHOD
[Definition], [Mechanics], [Synthetic example]
```

Componente V: Restrições. A seção `#CONSTRAINTS` define as restrições que a resposta deve seguir. Conforme apresentado no Código 3.8, foram definidas restrições relacionadas a definição de refatoração, visando maior consistência dos resultados. Dessa forma, além dos ajustes nos

parâmetros de temperatura, as restrições também orientam o LLM visando mitigar ao máximo as alucinações.

Listing 3.8 – Prompt para detecção de refatorações (restrições).

```
#CONSTRAINTS

    - Report only refactorings that strictly match the formal definitions above.
    - Do not report a change as a refactoring if it introduces new logic, fixes a bug, alters
    behavior, or modifies the observable output of the method.
    - Do not infer behavior or intent beyond what is provided in the code.
    - If the same transformation can be explained by multiple types, choose the one that best
    represents the primary structural change.
    - If no refactoring is detected, return an empty list.
```

Componente VI: Formatos de entrada e saída. Em seguida, especificam-se os formatos de saída e de entrada, pelas seções de `#OUTPUT` e `#INPUT FORMAT` respectivamente. A saída é padronizada em JSON, o que facilita a análise automatizada das respostas, conforme apresentado no Código 3.9. O Código 3.10 mostra a definição da entrada, que admite um ou múltiplos arquivos Java concatenados, com marcadores de estado para as classes criadas ou removidas no *commit*.

Listing 3.9 – Prompt para detecção de refatorações (formato de saída).

```
#OUTPUT

    Return ONLY valid JSON, with no explanations or text outside it, in the following format:

    {
      "refactorings": [
        {
          "type": "<Extract Method | Rename Method | Move Method>",
          "method_before": "<full method signature as it appears in BEFORE, or null if the
    method did not exist>",
          "method_after": "<full method signature as it appears in AFTER, or null if the method
    was removed>",
          "description": "<one sentence explaining what structural transformation occurred and
    why it matches Fowler's definition>"
        }
      ]
    }
```

Listing 3.10 – Prompt para detecção de refatorações (formato de entrada).

```
#INPUT FORMAT

    Each of the BEFORE and AFTER sections below may contain ONE or MULTIPLE Java source files.
    When multiple files are present, they are concatenated in the following format:

      Arquivo 1 (FileName1.java): <full source of file 1>, Arquivo 2 (FileName2.java): <full
    source of file 2>, ...
```

Componente VII: Ação. Por fim, a seção `#ACTION` (Código 3.11) instrui o LLM a analisar os trechos e a produzir o JSON de saída. É nessa seção que o código `BEFORE` e `AFTER` de cada operação é efetivamente injetado no *prompt*.

Listing 3.11 – Prompt para detecção de refatorações (ação).

```
#ACTION

    Analyze the code snippets below and produce the output JSON.

    BEFORE Code:
    <BEFORE source code>

    AFTER Code:
    <AFTER source code>
```

Duas decisões de projeto guiaram a criação do *prompt*. A primeira foi descrever cada tipo de refatoração isoladamente, sem definições por contraste (por exemplo, sem instruir que um *Move Method* não é um *Extract Method*). Definições contrastivas tendem a induzir a classificação por eliminação e a enviesar o LLM em direção a um rótulo; por isso, optou-se por caracterizar cada operação apenas por seus próprios sinais estruturais. A segunda foi o aprofundamento simétrico das descrições: os tipos que concentravam erros tiveram sua motivação e mecânica detalhadas a partir de Fowler (1999) (como o tratamento de variáveis locais na extração e o ajuste de referências na movimentação de um método já existente), preservando profundidade equivalente para os três tipos, de modo a não favorecer nenhum deles. O *prompt* foi redigido em inglês, idioma em que foi submetido aos LLMs. Dada a sua extensão, ele é reproduzido na íntegra no Apêndice A.

Essa versão do *prompt* não foi obtida de uma só vez, mas resultou de um processo de refinamento conduzido sobre um conjunto de validação de 9 operações (3 de cada tipo), extraído do mesmo oráculo de Tsantalis et al. (2022), porém de *commits* e repositórios que não integram as 150 operações de avaliação. Essa separação garante que os ajustes no *prompt* não tenham sido guiados pelos casos sobre os quais o método é posteriormente medido. Como a aferição desse refinamento depende de medições, seus resultados são reportados junto aos demais, na Seção 4.1, que compara o desempenho no conjunto de validação antes e depois dos ajustes.

3.4 Detecção de operações de refatoração via LLMs

A detecção das operações de refatoração é realizada por meio de LLMs acessados pelas APIs oficiais de seus fornecedores. Foram avaliados dois LLMs de famílias distintas: um modelo da família *Generative Pre-trained Transformer* (GPT), especificamente o GPT-5.4, da OpenAI, e o Claude Haiku 4.5, da Anthropic. A integração é feita pelas bibliotecas clientes oficiais para Node.js (`openai` e `@anthropic-ai/sdk`), possibilitando a execução automatizada dos experimentos.

O processo de detecção consiste no envio, a cada LLM, do *prompt* previamente definido (Seção 3.3), acompanhado das versões anterior e posterior do código. Cada LLM é então responsável por analisar semanticamente as diferenças entre essas versões e identificar as operações de

refatoração presentes. Os dois LLMs recebem exatamente o mesmo *prompt* e são consultados de forma independente, sem que a saída de um influencie a do outro, o que assegura uma comparação justa entre eles.

Entre os parâmetros que influenciam o comportamento dos LLMs, destaca-se a temperatura, responsável por controlar o grau de aleatoriedade das respostas: valores mais baixos tendem a gerar saídas mais determinísticas, enquanto valores mais altos aumentam a variabilidade (LI et al., 2025). Neste trabalho, a temperatura foi fixada em 0,2 para ambos os LLMs, buscando equilíbrio entre estabilidade e capacidade de interpretação semântica (LI et al., 2025): a escolha reduz variações indesejadas e mitiga alucinações, ao mesmo tempo em que preserva a habilidade de identificar refatorações que não dependem exclusivamente de padrões sintáticos rígidos. Os demais parâmetros foram mantidos em seus valores padrão, à exceção do limite de *tokens* de saída do Claude, ajustado para acomodar respostas mais longas.

O Código 3.12 ilustra a configuração das chamadas às duas APIs, com a seleção do LLM e o controle da temperatura. Essas chamadas são realizadas, de forma repetida e independente, para cada operação do conjunto experimental, compondo o *pipeline* de detecção automática proposto.

Listing 3.12 – Exemplo de chamada às APIs dos dois LLMs avaliados.

```
// OpenAI (GPT)
const respostaGpt = await openaiClient.chat.completions.create({
  model: "gpt-5.4",
  temperature: 0.2,
  messages: [{ role: "user", content: prompt }],
});

// Anthropic (Claude)
const respostaClaude = await claudeClient.messages.create({
  model: "claude-haiku-4-5-20251001",
  max_tokens: 8192,
  temperature: 0.2,
  messages: [{ role: "user", content: prompt }],
});
```

As respostas de ambos os LLMs são retornadas em JSON e processadas por uma rotina de *parsing* tolerante, que extrai o objeto JSON mesmo na presença de eventuais marcações de código ou texto adicional ao redor da resposta. As detecções resultantes são então armazenadas, por LLM, para a etapa de avaliação.

3.5 Avaliação e síntese dos resultados

A avaliação da abordagem consiste na comparação entre as operações identificadas pelos LLMs e aquelas registradas no oráculo de Tsantalis et al. (2022) para os *commits* analisados.

À semelhança da metodologia adotada na construção do oráculo, a análise considera *commits* completos, preservando o contexto real das modificações e evitando a fragmentação artificial das mudanças em blocos isolados; isso permite avaliar os LLMs em cenários próximos da prática, nos quais múltiplas operações podem ocorrer em um mesmo *commit*.

Para cada *commit*, considera-se como verdade o conjunto de operações que, no oráculo, estão validadas como *True Positive* (TP) e pertencem a um dos três tipos estudados. Operações marcadas como CTP (*Commented True Positive*), casos em que os validadores do oráculo registraram ressalvas quanto à classificação do tipo, e como FP não são consideradas verdade; consequentemente, uma detecção que corresponde apenas a uma operação CTP é contabilizada como falso positivo. Trata-se de uma escolha conservadora, que evita creditar ao LLM operações cuja própria classificação é objeto de dúvida. Cada detecção emitida pelos LLMs foi confrontada com o oráculo e classificada manualmente, a fim de evitar os erros de um casamento puramente textual de assinaturas que funcionaria a partir de script que automatizaria a compilação dos resultados. A partir desse confronto, definem-se três categorias:

- *Verdadeiro positivo (TP)*: refatoração detectada pelo modelo que corresponde a uma operação TP do oráculo.
- *Falso positivo (FP)*: refatoração detectada pelo modelo que não corresponde a nenhuma operação TP do oráculo.
- *Falso negativo (FN)*: operação-alvo do *dataset* que o modelo não identificou.

Sobre essas categorias, a avaliação emprega as métricas usuais de detecção. A precisão é definida como $\mathrm{TP}/(\mathrm{TP}+\mathrm{FP})$. Para o *recall*, reportam-se duas formulações complementares, que medem aspectos distintos da cobertura:

- *Recall sobre os alvos*:fração das operações-alvo efetivamente identificadas, calculada como a razão entre os alvos encontrados e o total de alvos (50 por tipo, 150 no total). Essa métrica indica quantas das operações-alvo foram identificadas pelos LLMs.
- *Recall sobre as detecções*: Métrica complementar ao *recall* sobre os alvos, pois considera todas as refatorações corretamente identificadas no *commit*. Como o numerador (TP) inclui também essas detecções adicionais, seus valores tendem a ser superiores aos do *recall* sobre os alvos.

O F_1 reportado é a média harmônica entre a precisão e o *recall* sobre as detecções. Cada detecção é ainda classificada quanto à sua origem: no dataset inicial, quando corresponde à operação-alvo do *commit*, ou detecção nova, quando o LLM aponta uma refatoração que não era o alvo selecionado. Por fim, para caracterizar os erros, cada detecção contabilizada como falso

positivo foi classificada manualmente segundo a transformação real a que de fato correspondia, análise cujos resultados são apresentados no Capítulo 4.

4 Resultados

Este capítulo apresenta e discute os resultados da detecção de operações de refatoração pelos dois LLMs avaliados, seguindo o protocolo descrito no Capítulo 3 e sobre o conjunto experimental caracterizado na Seção 3.2.1. A Seção 4.1 reporta o efeito do refinamento do *prompt* sobre o conjunto de validação. Em seguida, a Seção 4.2 responde às três questões de pesquisa do estudo: o desempenho geral de cada LLM (QP1), a variação desse desempenho por tipo de refatoração (QP2) e as razões da imprecisão observada (QP3), apresentando cada resultado ao lado de sua interpretação. A Seção 4.3 faz a síntese transversal e posiciona a abordagem frente às ferramentas do estado da arte, e a Seção 4.4 encerra com as ameaças à validade. As métricas empregadas (precisão, *recall* sobre os alvos, *recall* sobre as detecções e F_1) foram definidas na Seção 3.5. Nenhuma das execuções resultou em erro de API ou resposta malformada, de modo que as 150 operações foram efetivamente processadas por ambos os LLMs.

4.1 Refinamento do Prompt

Antes da avaliação sobre as 150 operações, o *prompt* foi refinado sobre um conjunto de validação de 9 operações (3 de cada tipo), extraído do mesmo oráculo de Tsantalis et al. (2022), porém de *commits* e repositórios que não integram a avaliação. A Tabela 4.1 compara o desempenho nesse conjunto antes e depois do refinamento, medido de forma simplificada pela contagem de acertos de tipo e de método e por ausência de respostas.

Tabela 4.1 – Efeito do refinamento do *prompt* sobre o conjunto de validação (9 operações, disjunto do conjunto experimental).

	GPT		**Claude**	
Métrica	**Antes**	**Depois**	**Antes**	**Depois**
Recall (tipo)	4/9	7/9	8/9	8/9
Recall (método)	2/9	5/9	6/9	6/9
Silêncios	4	2	0	0

Fonte: elaborada pelo autor.

O ganho concentrou-se no GPT, o LLM mais conservador: o *recall* em nível de tipo subiu de 4/9 para 7/9 e o de método de 2/9 para 5/9, com o número de silêncios reduzido pela metade (de 4 para 2). O Claude manteve-se estável (8/9 e 6/9), sem regressão. A versão refinada do *prompt* é a empregada na avaliação sobre as 150 operações.

4.2 Questões de Pesquisa

Esta seção apresenta os resultados das questões de pesquisa propostas neste trabalho.

4.2.1 (QP1) Como os LLMs avaliados se comparam quanto ao desempenho na detecção de operações de refatoração?

As Tabelas 4.2 e 4.3 consolidam o desempenho de cada LLM, discriminado por tipo de refatoração e no total. Os dois LLMs exibem perfis nitidamente opostos, que as métricas tornam evidentes. O GPT é conservador: emite poucas detecções (104 no total, contra 294 do Claude) e, quando aponta uma refatoração, costuma acertar, o que lhe confere a maior precisão (0,731). Esse rigor tem um custo: ele deixa passar a maioria das operações-alvo, com *recall* sobre os alvos de apenas 0,333, isto é, encontrou 50 dos 150. O Claude segue a lógica inversa: é expansivo e identifica a maior parte dos alvos (*recall* de 0,720, ou seja, 108 dos 150), mas, ao gerar um volume muito maior de detecções, reduz a precisão para 0,537.

Tabela 4.2 – Resultados de detecção do Claude.

Tipo	**TP**	**FP**	**FN**	**Precisão**	**Recall$_{\text{alvos}}$**	**Recall$_{\text{detec}}$**	F_1
Extract Method	73	74	0	0,497	1,000	1,000	0,664
Rename Method	44	24	23	0,647	0,540	0,657	0,652
Move Method	41	38	19	0,519	0,620	0,683	0,590
Total	**158**	**136**	**42**	**0,537**	**0,720**	**0,790**	**0,640**

Fonte: elaborada pelo autor.

Tabela 4.3 – Resultados de detecção do GPT.

Tipo	**TP**	**FP**	**FN**	**Precisão**	**Recall$_{\text{alvos}}$**	**Recall$_{\text{detec}}$**	F_1
Extract Method	26	4	31	0,867	0,380	0,456	0,598
Rename Method	17	3	39	0,850	0,220	0,304	0,447
Move Method	33	21	30	0,611	0,400	0,524	0,564
Total	**76**	**28**	**100**	**0,731**	**0,333**	**0,432**	**0,543**

Fonte: elaborada pelo autor.

O confronto entre as duas formas de *recall* aprofunda esse contraste. No Claude, o *recall* sobre as detecções (0,790) supera o *recall* sobre os alvos (0,720), pois parte de seus acertos recai sobre refatorações reais do *commit* que não eram a operação rotulada. No GPT, a diferença é bem menor (0,432 contra 0,333), coerente com um LLM que quase não emite detecções além do alvo. Essa sobredetecção do Claude é expressiva: das 294 detecções que produziu, 186 vão além do alvo selecionado, contra 54 das 104 do GPT. O F_1, que pondera precisão e cobertura, favorece o Claude (0,640 contra 0,543): sua cobertura ampla compensa a precisão mais baixa em maior grau do que a precisão elevada do GPT compensa sua baixa cobertura.

Resumo: Os LLMs apresentaram perfis complementares: o GPT priorizou precisão (0,731), enquanto o Claude alcançou maior cobertura (*recall* de 0,720). Em termos de equilíbrio entre precisão e cobertura, o Claude obteve o melhor desempenho geral ($F_1 = 0,640$).

4.2.2 (QP2) Como o desempenho dos LLMs varia entre os diferentes tipos de operações de refatoração?

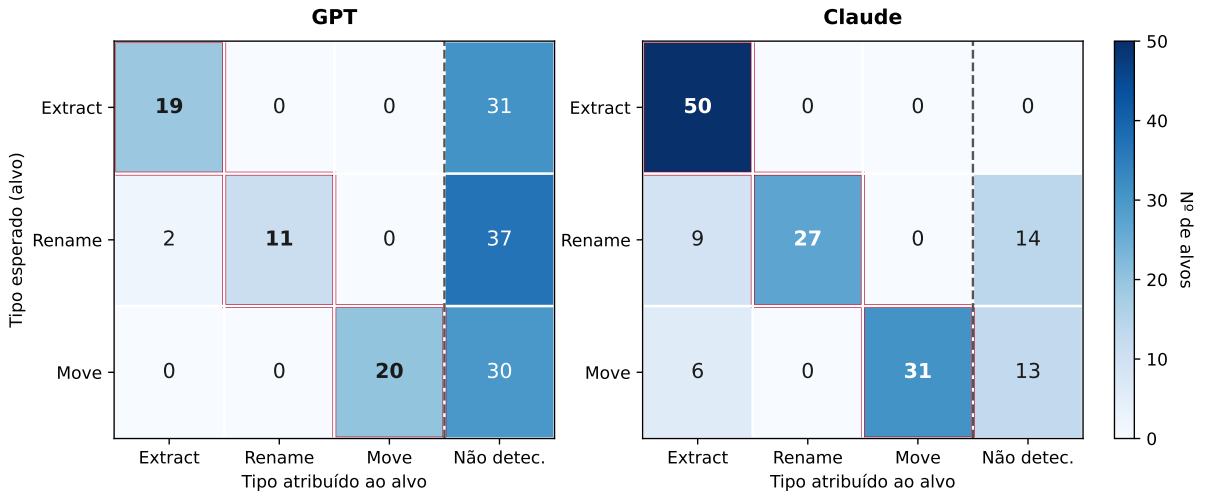
Em relação ao desempenho por tipo de refatoração, a detecção de Extract Method é o tipo em que os dois perfis mais se distanciam. O Claude identificou todas as 50 operações-alvo (*recall* de 1,000), mas com a menor precisão do experimento (0,497): das 147 detecções que emitiu para esse tipo, 74 são falsos positivos. O GPT exibe o retrato inverso, com precisão de 0,867 (apenas 4 falsos positivos) e *recall* de somente 0,380. Esse contraste é coerente com a natureza local da extração: o trecho extraído permanece no mesmo arquivo e o método de origem passa a invocar o novo método, oferecendo um sinal estrutural nítido e contido em um único *snapshot*. Por ser um sinal tão acessível, o Claude o reconhece sem falhas, mas também o projeta em excesso.

O Rename Method foi a operação mais difícil para ambos, com a menor cobertura do conjunto (*recall* de 0,540 no Claude e 0,220 no GPT). Ainda assim, quando se comprometem com uma renomeação, ambos costumam acertar (precisão de 0,647 no Claude e 0,850 no GPT), o que sugere que a dificuldade está em perceber a renomeação, e não em confirmá-la. Uma explicação plausível é que boa parte dos casos do oráculo não é uma renomeação pura: a mudança de nome vem acompanhada de alterações de assinatura (novos parâmetros, mudança de tipo de retorno), o que dilui o sinal. Já o Move Method apresentou o desempenho mais equilibrado entre os LLMs (precisão de 0,519 no Claude e 0,611 no GPT); sua natureza distribuída em dois arquivos exige raciocínio entre múltiplos *snapshots*, o que reduz a cobertura, mas torna as detecções mais criteriosas.

Confusão entre tipos. Para entender se os alvos perdidos foram rotulados com o tipo errado ou simplesmente não detectados, a Figura 4.1 apresenta, para cada LLM, a matriz de confusão focada no alvo: para cada uma das 150 operações-alvo, registra-se se o LLM a detectou com o tipo correto, se lhe atribuiu um dos outros dois tipos ou se não a detectou. As linhas indicam o tipo esperado e as colunas o tipo atribuído ao método-alvo, acrescidas da coluna “Não detectado”; cada linha soma 50 e a diagonal reproduz o *recall* sobre os alvos.

Nota-se que os dois LLMs falharam sobretudo por omissão, e não por confusão de tipo. No GPT, dos 100 alvos não recuperados, 98 sequer foram detectados e apenas 2 receberam um tipo incorreto; sua baixa cobertura é quase inteiramente silêncio, coerente com o perfil conservador. No Claude, dos 42 alvos perdidos, 27 não foram detectados e apenas 15 foram rotulados incorretamente. Mesmo no LLM mais expansivo, portanto, a confusão de tipo é minoritária diante da não-detecção, e a fragilidade central da abordagem é de reconhecer que há uma refatoração ali, não de rotulação do que já foi reconhecido.

Figura 4.1 – Matrizes de confusão focadas no alvo, por LLM. A diagonal indica os acertos de tipo; a coluna “Não detectado” registra os alvos que o LLM não identificou.



Fonte: elaborada pelo autor.

Quando a confusão ocorre, contudo, ela tem direção única: todos os rótulos trocados convergem para o Extract Method (15 casos no Claude, 2 no GPT); não se observou nenhum Move Method rotulado como Rename Method, nem o inverso.

Resumo: O desempenho variou conforme o tipo de refatoração: o Claude alcançou cobertura máxima para *Extract Method*, enquanto o GPT apresentou maior precisão. Já *Rename Method* foi o tipo mais difícil para ambos, e a maioria dos erros decorreu da omissão de refatorações, e não da confusão entre tipos.

4.2.3 (QP3) Quais são as principais razões para a imprecisão dos LLMs na detecção de refatorações?

A precisão, isoladamente, informa o quanto cada LLM erra, mas não por que erra. Investigar as razões dessa imprecisão exige examinar as detecções que não encontram respaldo no oráculo, ou seja, os falsos positivos. Para tanto, cada uma das 164 detecções classificadas como FP (136 do Claude e 28 do GPT) foi inspecionada manualmente e confrontada com o *commit* correspondente, registrando-se a que transformação real cada uma correspondia. Dessa inspeção emergem cinco causas, sintetizadas na Tabela 4.4 e detalhadas por LLM (Figura 4.2) e por tipo atribuído (Figura 4.3):

- *Refatoração real fora do escopo*: a detecção corresponde a uma transformação legítima, porém de um tipo que não integra os três estudados (por exemplo, um Pull Up Method rotulado como Move Method).
- *Sem correspondência no oráculo*: a detecção não encontra respaldo em nenhuma operação do oráculo (sobredetecção sem base).

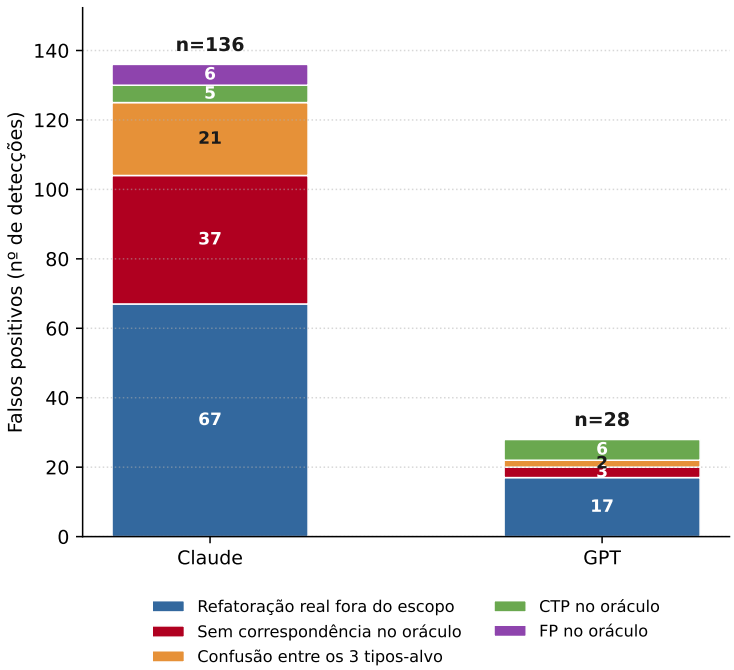
- *Confusão entre os três tipos-alvo*: a operação é real, validada como TP e pertence aos três tipos, mas o LLM lhe atribuiu o rótulo de outro deles.
- *CTP no oráculo*: a detecção corresponde a uma operação classificada no oráculo como CTP (*Commented True Positive*), isto é, um caso em que os validadores registraram ressalvas quanto à classificação da refatoração e, por isso, não a confirmaram de forma definitiva. Em nossa avaliação, essas detecções são contabilizadas como FP, seguindo a regra conservadora adotada.
- *FP no oráculo*: a detecção corresponde apenas a uma operação que o próprio oráculo já havia rejeitado como falso positivo.

Tabela 4.4 – Anatomia dos 164 falsos positivos, por tipo atribuído pelo LLM e causa. Contagem de detecções.

Tipo atribuído	**Fora do escopo**	**Sem base**	**Entre os 3 tipos**	**CTP**	**FP**	**Total**
Extract Method	19	34	20	0	5	78
Rename Method	18	5	3	0	1	27
Move Method	47	1	0	11	0	59
Total	**84**	**40**	**23**	**11**	**6**	**164**

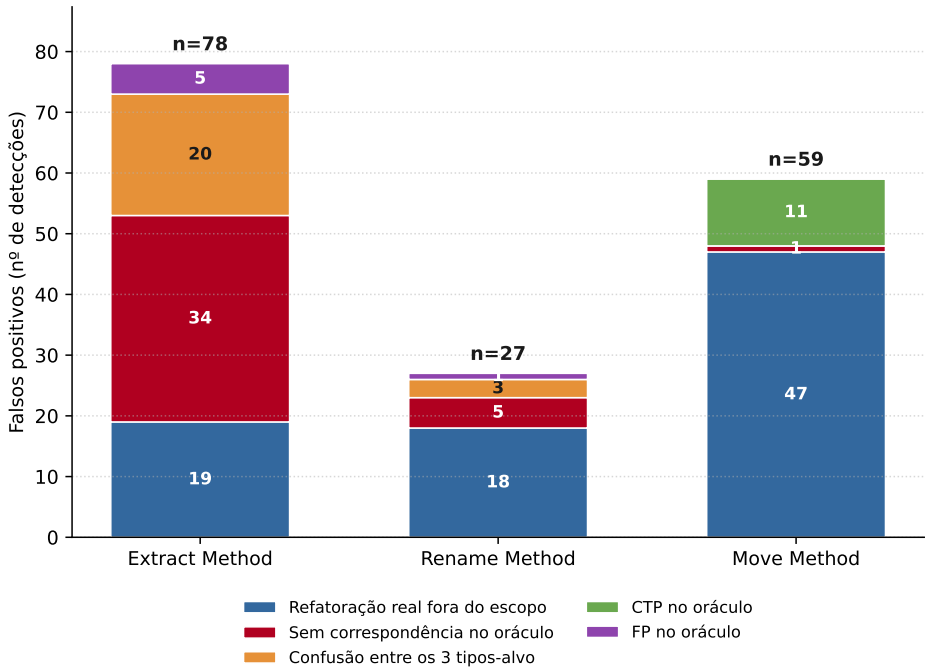
Fonte: elaborada pelo autor.

Figura 4.2 – Composição dos falsos positivos por LLM, segundo a causa.



Fonte: elaborada pelo autor.

Figura 4.3 – Composição dos 164 falsos positivos (ambos os LLMs) por tipo atribuído pelo LLM e causa.



Fonte: elaborada pelo autor.

A interpretação da precisão é que a maior parte dos falsos positivos não decorre de invenção do LLM. Apenas 46 das 164 detecções (28,0%) não correspondem a nenhuma operação validada como TP (40 sem qualquer registro no oráculo e 6 casando com operações já rejeitadas); as 118 restantes (72,0%) correspondem a uma transformação real presente no *commit*: refatorações fora dos três tipos (84), operações dos três tipos com rótulo trocado (23) ou operações marcadas como CTP (11). Em outras palavras, na ampla maioria dos casos o LLM acerta que existe uma refatoração ali e erra apenas a sua catalogação, ou a detecta em um tipo que o recorte experimental deliberadamente excluiu.

Cada tipo falha por um mecanismo próprio (Figura 4.3). Os falsos positivos de Move Method são quase integralmente refatorações reais fora do escopo (47 de 59) ou operações CTP (11 de 59): quando um LLM aponta um Move Method equivocadamente, ele quase sempre viu uma relocação verdadeira de código, apenas de uma variante não catalogada no estudo, o que confirma que a fragilidade do Move está na cobertura, não na fabricação de detecções. Em contraste, o Extract Method concentra tanto a sobredetecção sem base (34 dos 40 casos) quanto a confusão entre os três tipos (20 dos 23), reforçando seu papel de atrator. As 84 detecções fora do escopo (Tabela 4.5) são dominadas por operações compostas ou especializadas de movimentação e extração (56 casos, com destaque para Move and Rename Method e Pull Up Method), em que o LLM capta o componente estrutural, mas não reconhece que a operação completa é composta.

A análise por LLM (Figura 4.2) separa nitidamente os dois perfis. Dos 28 falsos positivos do GPT, apenas 3 são sobredetecção sem base; os demais correspondem a refatorações reais fora

Tabela 4.5 – Composição das 84 confusões com refatorações reais fora do escopo, por família de operação.

Família (operações mais frequentes)	**Casos**
Movimentação/extração compostas e especializadas (Move and Rename 22, Pull Up 10, Extract and Move 7, Inline 7, Move Class 4, Merge 3, Move Attribute 2, Move Code 1)	56
Mudanças de assinatura (Change Return Type 6, Change Method Access Modifier 5, Add Parameter 4, Add Method Modifier 4, Remove Parameter 2, demais 2)	23
Renomeações em outros níveis (Rename Class 2, Rename Variable 2, Rename Attribute 1)	5
Total	**84**

Fonte: elaborada pelo autor.

do escopo (17), operações CTP (6) ou confusões entre os três tipos (2). Quase nenhum falso positivo do GPT é, portanto, uma detecção inventada, de modo que a sua precisão efetiva é ainda superior à reportada. No Claude, ao contrário, concentra-se a quase totalidade da sobredetecção sem base (37 dos 40 casos), que recai quase inteiramente sobre o rótulo Extract Method.

Complementaridade entre os LLMs. Como ambos os LLMs analisaram exatamente os mesmos 150 alvos, é possível medir a cobertura da combinação dos dois. A Tabela 4.6 apresenta o *recall* sobre os alvos de cada LLM e da união, na qual um alvo é considerado identificado se ao menos um dos LLMs o encontra.

Tabela 4.6 – *Recall* sobre os alvos por LLM e da união (GPT ∪ Claude), discriminado por tipo. Alvos identificados em 50 por tipo, 150 no total.

Tipo	**Claude**	**GPT**	**União**	**Recall união**
Extract Method	50	19	50	1,000
Rename Method	27	11	31	0,620
Move Method	31	20	35	0,700
Total	**108**	**50**	**116**	**0,773**

Fonte: elaborada pelo autor.

Os LLMs concordam em apenas 76 dos 150 casos (42 em que ambos acertam e 34 em que ambos falham); nos 74 de desacordo, a assimetria é acentuada, com 66 alvos exclusivos do Claude contra apenas 8 do GPT. Essa baixa concordância tem, contudo, uma leitura positiva: os LLMs são complementares. Combinando-se as duas saídas em um esquema de união, o *recall* sobre os alvos sobe para 116 dos 150 (0,773), acima de qualquer LLM isolado; o ganho concentra-se no Move Method e no Rename Method, ao passo que no Extract Method não há ganho, pois o Claude já identifica sozinho todos os alvos. Os 8 alvos exclusivos do GPT são operações estruturais que o Claude deixou passar, indício de que mesmo o LLM de baixa cobertura recupera casos que o

expansivo não cobre.

Esse ganho tem um custo: como a união soma as detecções dos dois LLMs, ela herda os falsos positivos de ambos e sua precisão fica em 0,560, abaixo da do GPT. Ainda assim, o resultado sugere que, em um fluxo com revisão humana, o uso conjunto de LLMs distintos (um voltado à cobertura, outro à exatidão) pode ampliar a detecção para além do que qualquer um deles alcança isoladamente.

Resumo: A maioria dos falsos positivos (72%) corresponde a refatorações reais presentes no *commit*, principalmente operações fora do escopo do estudo ou classificadas com o tipo incorreto. Apenas 28% das detecções não encontraram correspondência com refatorações validadas no oráculo.

4.3 Discussões e Implicações

Os resultados das seções anteriores convergem para um retrato consistente. Nenhum dos dois LLMs reúne, isoladamente, alta precisão e alta cobertura: o GPT aproxima-se de um detector de alta confiança e baixa sensibilidade, adequado a cenários em que um falso positivo é caro; o Claude, de um detector sensível e menos preciso, adequado a fluxos de triagem em que se prefere não deixar ocorrências passarem. Além disso, a fragilidade dominante de ambos é de cobertura, não de rotulação: falham mais por não perceber a refatoração do que por classificá-la erradamente. E, quando classificam algo incorretamente, raramente inventam: 72,0% dos falsos positivos correspondem a transformações reais no código (Seção 4.2.3).

Desenvolvimento de ferramentas multi-linguagem. Esse conjunto de observações permite posicionar a abordagem frente às ferramentas consolidadas de análise estática. A Tabela 4.7 confronta os LLMs e a sua união com o RefactoringMiner e o RefDiff, cujos valores provêm de suas avaliações originais.

Tabela 4.7 – Posicionamento da abordagem frente às ferramentas do estado da arte. Os valores das ferramentas provêm de suas avaliações originais; os dos LLMs referem-se ao conjunto experimental deste trabalho (*recall* sobre os alvos).

Abordagem	**Precisão**	***Recall***
RefactoringMiner	99,6%	94,0%
RefDiff	100%	88,0%
GPT	73,1%	33,3%
Claude	53,7%	72,0%
União (GPT ∪ Claude)	56,0%	77,3%

Fonte: elaborada pelo autor.

Em termos absolutos, os resultados situam-se abaixo dos reportados por essas ferramentas. O RefactoringMiner reporta precisão de 99,6% e *recall* de 94% (TSANTALIS et al.,

2018), e o RefDiff, precisão de 100% e *recall* de 88% (SILVA; VALENTE, 2017; SILVA et al., 2021). A melhor precisão obtida entre os LLMs (0,731, GPT) e o melhor *recall* (0,720, Claude) permanecem aquém desses patamares, diferença que não deve ser minimizada: no estágio atual, LLMs não substituem ferramentas especializadas para a detecção de refatorações.

Essa comparação, contudo, deve ser lida sob condições desiguais. As ferramentas dedicadas dispõem de *parsers* específicos da linguagem e operam sobre a árvore sintática completa do código, ao passo que a abordagem avaliada recebe apenas dois *snapshots* textuais, sem treinamento ou ajuste para a tarefa. Soma-se a isso a natureza parcial do oráculo (Seção 3.2.1) e a evidência de que a maioria dos falsos positivos corresponde a transformações reais: a precisão reportada é penalizada por uma condição que não se aplica à validação original das ferramentas.

Abordagens Híbridas para detecção de refatorações. Vários resultados sustentam o potencial do método. O mais expressivo é o Extract Method, em que o Claude igualou a cobertura das ferramentas dedicadas (todas as 50 operações) sem dispor de qualquer *parser*; igualmente relevante é a confiabilidade no Move Method, cuja fragilidade se concentra na cobertura, e não na exatidão do que é apontado. A abordagem agrega ainda duas vantagens qualitativas ausentes na análise estática: cada detecção vem acompanhada de uma justificativa em linguagem natural, útil para revisão assistida, e, por prescindir de *parser* dedicado, é imediatamente aplicável a outras linguagens. Combinando LLMs distintos, a técnica eleva o *recall* sobre os alvos a 77,3%, aproximando-se da faixa das ferramentas especializadas a partir de duas saídas não treinadas. No estágio atual, portanto, LLMs não competem com RefactoringMiner e RefDiff em precisão e *recall* globais para Java, mas constituem uma técnica complementar promissora, especialmente atrativa em cenários com revisão humana ou em linguagens sem ferramental maduro de detecção. Adicionalmente, os modelos exibem perfis complementares: o GPT prioriza precisão, enquanto o Claude alcança maior cobertura. Como mostra Tabela 4.7, a combinação das detecções de ambos eleva o *recall*, indicando que abordagens híbridas baseadas em múltiplos LLMs representam uma direção promissora para aumentar a cobertura da detecção de refatorações.

4.4 Ameaças à Validade

Os resultados apresentados estão sujeitos a um conjunto de ameaças à validade, discutidas a seguir nas dimensões usuais de estudos empíricos em engenharia de *software*.

Validade interna. A inferência por LLM é não determinística, o que poderia introduzir variação entre execuções. Para reduzi-la, empregou-se temperatura 0,2 e submeteram-se ambos os LLMs exatamente ao mesmo *prompt* e *pipeline*. Ainda assim, cada caso foi executado uma única vez, sem repetição que permitisse estimar a variância, de modo que pequenas oscilações nos números não podem ser descartadas. O desempenho também é sensível ao *prompt*; para mitigar esse risco, ele foi refinado sobre um conjunto de validação disjunto do de avaliação (Seção 4.1) e descreve cada operação isoladamente, sem definições por contraste que pudessem enviesar a

classificação. Por fim, nos casos de Move Method em que uma das classes é criada ou removida no *commit*, um dos *snapshots* é um marcador textual de estado, e não código; embora esse marcador transmita fielmente a evolução do arquivo, ele altera o formato da entrada nesses casos, o que pode influenciar a detecção do tipo.

Validade externa (generalização). O estudo restringe-se à linguagem Java e a três tipos de refatoração (Extract Method, Rename Method e Move Method); a generalização para outras linguagens e para o catálogo completo de refatorações não foi verificada empiricamente. Alguns aspectos do desenho atenuam essa limitação, sem eliminá-la. A amostra, embora restrita a 150 operações, é balanceada entre os três tipos e diversa em origem: provém de 81 repositórios e 141 autores distintos, com ampla variação de porte e visibilidade (Seção 3.2.1), o que reduz a dependência de um único projeto ou estilo de código. A avaliação empregou dois LLMs de famílias distintas (o GPT-5.4 e o Claude Haiku 4.5), o que evita conclusões atadas ao comportamento de um único LLM. E, por prescindir de *parser* ou de qualquer componente específico de linguagem, a abordagem é, em princípio, aplicável a outras linguagens sem reimplementação, ainda que essa aplicabilidade permaneça uma conjectura a ser testada. Duas ressalvas subsistem: os projetos são majoritariamente *open-source* maduros e de alta visibilidade, podendo não representar código de menor maturidade ou de domínios fechados; e os dois LLMs correspondem a versões pontuais no tempo, de modo que, dada a rápida evolução dos LLMs, os números devem ser lidos como um retrato do estado atual, não como um limite permanente da técnica.

Validade de conclusão. Os valores de precisão e *recall* atribuídos ao RefactoringMiner e ao RefDiff provêm de suas avaliações originais, conduzidas sobre conjuntos distintos do utilizado neste trabalho, e não foram reexecutados sobre as 150 operações aqui analisadas; a comparação é, portanto, indireta. Para que permaneça informativa, ela é apresentada como referência de magnitude entre as abordagens, e não como *benchmark* pareado sob condições idênticas, e ampara-se no fato de o conjunto experimental derivar do mesmo oráculo de Tsantalis et al. (2022) que fundamenta a avaliação do RefactoringMiner.

5 Considerações Finais

5.1 Síntese dos Resultados

Este trabalho teve como objetivo geral analisar a eficácia de LLMs na detecção automática de refatorações, avaliando a sua capacidade de identificar e classificar diferentes tipos de operações. O estudo concentrou-se nas operações *Extract Method*, *Move Method* e *Rename Method*, em nível de método, na linguagem Java. Para alcançar esse objetivo geral, foram definidos três objetivos específicos, cujo cumprimento é retomado a seguir.

O primeiro objetivo específico consistiu em investigar técnicas recentes para análise de código via LLMs, com foco em qualidade de *software* e refatorações. Ele foi atingido pela revisão apresentada no Capítulo 2, que fundamentou os conceitos de LLMs e da arquitetura *Transformer*, caracterizou a refatoração segundo o catálogo de Fowler (1999) e discutiu as ferramentas consolidadas de detecção, RefactoringMiner e RefDiff, situando a lacuna que motiva o uso de LLMs. O segundo objetivo foi desenvolver um método baseado em LLM para detecção e classificação de refatorações em Java; ele foi cumprido com a construção, descrita no Capítulo 3, de um *prompt* estruturado em seções e fundamentado nas definições de Fowler (1999), aplicado a um conjunto de 150 operações validadas como *True Positive* extraídas do oráculo de Tsantalis et al. (2022), balanceadas entre os três tipos e oriundas de 81 repositórios e 141 autores distintos, e submetidas a dois LLMs de famílias distintas, o GPT-5.4 e o Claude Haiku 4.5. O terceiro objetivo foi discutir a efetividade do método proposto; ele foi atendido pela análise apresentada no Capítulo 4, que mediu as detecções segundo as métricas de precisão, *recall* e F_1, caracterizou os erros dos LLMs e posicionou os resultados frente às ferramentas de análise estática, identificando oportunidades de melhoria e limitações.

A síntese dos resultados, detalhada no Capítulo 4, revelou dois perfis nitidamente opostos: o GPT mostrou-se conservador, com a maior precisão do experimento (0,731), porém baixa cobertura (*recall* sobre os alvos de 0,333); o Claude mostrou-se expansivo, recuperando a maior parte dos alvos (*recall* de 0,720), ao custo de uma precisão menor (0,537). Três achados atravessaram toda a análise. Primeiro, a fragilidade dominante de ambos é de cobertura, e não de rotulação: os LLMs falham mais por não perceber a refatoração do que por classificá-la de forma incorreta. Segundo, quando erram a classificação, raramente inventam, já que 72% dos falsos positivos correspondem a transformações reais presentes no *commit*, apenas fora dos três tipos estudados, com rótulo trocado ou marcadas como CTP no oráculo. Terceiro, os dois LLMs são complementares, e a combinação de suas saídas eleva o *recall* sobre os alvos para 0,773, acima de qualquer um deles isoladamente. Em termos absolutos, o desempenho situou-se abaixo do reportado pelo RefactoringMiner (precisão de 99,6% e *recall* de 94%) e pelo RefDiff (precisão de 100% e *recall* de 88%), diferença que, contudo, deve ser lida sob condições desiguais, uma

vez que as ferramentas dedicadas dispõem de *parsers* específicos da linguagem, enquanto os LLMs receberam apenas dois *snapshots* textuais, sem treinamento para a tarefa.

Considerando o conjunto dos objetivos específicos cumpridos, conclui-se que o objetivo geral foi alcançado: o trabalho analisou a eficácia dos LLMs e demonstrou empiricamente que eles são capazes de identificar e classificar operações de refatoração, ainda que, no estágio atual, não substituam ferramentas especializadas para a linguagem Java. A principal contribuição para o meio acadêmico é a evidência empírica, obtida sobre um conjunto diverso e derivado de um oráculo consolidado, de que os LLMs constituem uma técnica complementar promissora, e não uma substituta, para a detecção de refatorações. Essa contribuição sustenta-se em três aspectos ainda pouco explorados na literatura: a caracterização dos perfis de erro dos LLMs, que falham mais por omissão do que por confusão de tipo; a constatação de que a maioria de seus falsos positivos corresponde a refatorações reais; e a demonstração de vantagens qualitativas ausentes na análise estática, como a justificativa em linguagem natural que acompanha cada detecção e a aplicabilidade imediata a outras linguagens, por prescindir de *parser* dedicado.

5.2 Trabalhos Futuros

Os resultados obtidos abrem diversas frentes de continuidade. As mais imediatas são apresentadas a seguir.

- **Ampliação do catálogo de operações:** A análise dos falsos positivos revelou que boa parte deles corresponde a operações compostas ou especializadas fora do escopo, como *Move and Rename Method*, *Pull Up Method* e *Extract and Move Method*. Estender o estudo a esses e a outros tipos do catálogo de Fowler (1999) permitiria verificar se o mesmo comportamento se mantém em um espaço de classificação maior.

- **Generalização para outras linguagens:** Como a abordagem prescinde de *parser* dedicado, uma continuação natural é aplicá-la a linguagens como Python, JavaScript e Go, transformando em evidência empírica a conjectura de portabilidade discutida neste trabalho.

- **Avaliação da estabilidade das respostas:** Cada operação foi submetida uma única vez aos LLMs. Repetir as execuções permitiria estimar a variância inerente à inferência não determinística e reportar os resultados com intervalos de confiança, fortalecendo a validade das conclusões.

- **Exploração de novos LLMs e configurações:** A rápida evolução dos LLMs torna relevante reavaliar a tarefa com modelos mais recentes, incluindo modelos abertos, bem como investigar o efeito de diferentes valores de temperatura e de estratégias de *prompt*.

- **Combinação de LLMs em *pipeline*:** A complementaridade observada sugere investigar arranjos que unam cobertura e precisão, por exemplo um esquema em dois estágios em

que um LLM sensível levanta candidatos e outro, mais criterioso, os confirma, reduzindo os falsos positivos sem sacrificar a cobertura.

- **Comparação pareada com as ferramentas do estado da arte:** A comparação com o RefactoringMiner e o RefDiff apoiou-se em valores de suas avaliações originais. Executar essas ferramentas sobre exatamente as 150 operações analisadas produziria uma comparação direta, sob condições idênticas, entre a abordagem baseada em LLMs e a análise estática tradicional.

5.3 Publicações & Artefatos

Os resultados obtidos neste Trabalho de Conclusão de Curso deram origem a um artigo científico, que foi submetido a um workshop da área de Engenharia de Software. O artigo encontra-se atualmente em processo de avaliação por pares, aguardando o parecer dos revisores.

Os *scripts* e conjuntos de dados resultantes deste trabalho de conclusão de curso encontram-se publicamente disponíveis em: <https://github.com/gabrielhenrique-123/llm-refactoring-detector>

Referências

ALBERTS, I. L.; MERCOLLI, L.; PYKA, T.; PRENOSIL, G.; SHI, K.; ROMINGER, A.; AFSHAR-OROMIEH, A. Large language models (llm) and chatgpt: what will the impact on nuclear medicine be? *European journal of nuclear medicine and molecular imaging*, Springer, v. 50, n. 6, p. 1549–1552, 2023.

ALMEIDA, A.; XAVIER, L.; VALENTE, M. T. Automatic library migration using large language models: First results. In: *18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. [S.l.: s.n.], 2024. p. 427–433.

AMARAL, L.; GAMA, E.; PAIXAO, M.; AGUIAR, L. *Evaluating Large Language Models on the Classification of Different Technical Debt Types in Stack Overflow Discussions*. 2025. 1–6 p.

Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. [S.l.], 2024. Online.

AQUINO, B.; BRITO, A.; TAVARES, C. S.; SEUFITELLI, D. B.; BATISTELI, J. P. O. Analisando a qualidade e eficácia de códigos gerados por llms: Um estudo com problemas da plataforma leetcode. In: *13th Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM)*. [S.l.: s.n.], 2025. p. 1–8.

BAI, Y.; KADAVATH, S.; KUNDU, S.; ASKELL, A.; KERNION, J.; JONES, A.; CHEN, A.; GOLDIE, A.; MIRHOSEINI, A.; MCKINNON, C. et al. Constitutional AI: Harmlessness from AI feedback. *arXiv preprint arXiv:2212.08073*, 2022.

BHAYANA, R. Chatbots and large language models in radiology: a practical primer for clinical and research applications. *Radiology*, Radiological Society of North America, v. 310, n. 1, p. e232756, 2024.

BRITO, A.; HORA, A.; VALENTE, M. T. Characterizing refactoring graphs in Java and JavaScript projects. *Empirical Software Engineering*, v. 26, 2021.

BRITO, A. N. de et al. Refactoring graphs: reasoning about refactoring over time. Universidade Federal de Minas Gerais, 2023.

BRITO, R.; VALENTE, M. T. RefDiff4Go: Detecting refactorings in Go. In: *14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*. [S.l.: s.n.], 2020. p. 101–110.

BRITO, R.; VALENTE, M. T. RAID: Tool Support for Refactoring-Aware Code Reviews. In: *Proceedings of the 29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. [S.l.]: IEEE/ACM, 2021. p. 265–275.

CAMPOS, I. M. *How to Write Well-Structured Prompts for LLMs*. 2026. <https://imcampos195.github.io/MyMumblings/Tutorial--EN-US.html>. Online; accessed April 2026.

CORDEIRO, J.; NOEI, S.; ZOU, Y. An empirical study on the code refactoring capability of large language models. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2025.

FOWLER, M. *Refactoring: Improving the Design of Existing Code*. [S.l.]: Addison-Wesley, 1999.

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

LI, L.; SLEEM, L.; NICHIL, G.; STATE, R. et al. Exploring the impact of temperature on large language models: Hot or cold? *Procedia Computer Science*, Elsevier, v. 264, p. 242–251, 2025.

MELO, A.; ALMEIDA, L.; GARCéS, L. Investigating the use of large language models in software security requirements: Results of a literature review. In: *IV Workshop Brasileiro de Engenharia de Software Inteligente (ISE)*. [S.l.: s.n.], 2025. p. 37–42.

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

MURPHY-HILL, E.; PARNIN, C.; BLACK, A. P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, IEEE, v. 38, n. 1, p. 5–18, 2011.

OTTER, D. W.; MEDINA, J. R.; KALITA, J. K. A survey of the usages of deep learning for natural language processing. *IEEE transactions on neural networks and learning systems*, IEEE, v. 32, n. 2, p. 604–624, 2020.

PANTIUCHINA, J.; ZAMPETTI, F.; SCALABRINO, S.; PIANTADOSI, V.; OLIVETO, R.; BAVOTA, G.; PENTA, M. D. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, ACM New York, NY, USA, v. 29, n. 4, p. 1–30, 2020.

SALLAM, M. The utility of chatgpt as an example of large language models in healthcare education, research and practice: Systematic review on the future perspectives and potential limitations. *MedRxiv*, Cold Spring Harbor Laboratory Press, p. 2023–02, 2023.

SIDDIQ, M. L.; SANTOS, J. C. D. S.; TANVIR, R. H.; ULFAT, N.; RIFAT, F. A.; LOPES, V. C. Using large language models to generate junit tests: An empirical study. In: *28th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. [S.l.: s.n.], 2024. p. 313–322.

SILVA, D.; SILVA, J. P. da; SANTOS, G.; TERRA, R.; VALENTE, M. T. Refdiff 2.0: A multi-language refactoring detection tool. *IEEE Transactions on Software Engineering*, v. 47, n. 12, p. 2786–2802, 2021.

SILVA, D.; TSANTALIS, N.; VALENTE, M. T. Why we refactor? confessions of GitHub contributors. In: *24th International Symposium on the Foundations of Software Engineering (FSE)*. [S.l.: s.n.], 2016. p. 858–870.

SILVA, D.; VALENTE, M. T. RefDiff: Detecting refactorings in version histories. In: *14th International Conference on Mining Software Repositories (MSR)*. [S.l.: s.n.], 2017. p. 269–279.

SILVA, L. L.; SILVA, J. R. d.; MONTANDON, J. E.; ANDRADE, M.; VALENTE, M. T. Detecting code smells using chatgpt: Initial insights. In: *18th International Symposium on Empirical Software Engineering and Measurement*. [S.l.: s.n.], 2024. p. 400–406.

TAECHARUNGROJ, V. "what can chatgpt do?" analyzing early reactions to the innovative ai chatbot on twitter. *Big Data and Cognitive Computing*, MDPI, v. 7, n. 1, p. 35, 2023.

TSANTALIS, N.; KETKAR, A.; DIG, D. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering (TSE)*, 2020.

TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L.; MAZINANIAN, D.; KETKAR, A. *Refactoring oracle*. 2022. <http://refactoring.encs.concordia.ca/oracle>. Online; accessed January 2022.

TSANTALIS, N.; MANSOURI, M.; ESHKEVARI, L. M.; MAZINANIAN, D.; DIG, D. Accurate and efficient refactoring detection in commit history. In: *40th International Conference on Software Engineering (ICSE)*. [S.l.: s.n.], 2018. p. 483–494.

VAITHILINGAM, P.; ZHANG, T.; GLASSMAN, E. L. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In: *Chi conference on human factors in computing systems extended abstracts*. [S.l.: s.n.], 2022. p. 1–7.

VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. [S.l.]: Independente, 2020.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, Ł.; POLOSUKHIN, I. Attention is all you need. *Advances in neural information processing systems*, v. 30, 2017.

WEBER, B.; REICHERT, M.; MENDLING, J.; REIJERS, H. A. Refactoring large process model repositories. *Computers in industry*, Elsevier, v. 62, n. 5, p. 467–486, 2011.

YAO, Y.; DUAN, J.; XU, K.; CAI, Y.; SUN, Z.; ZHANG, Y. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing*, Elsevier, v. 4, n. 2, p. 100211, 2024.

ZAN, D.; CHEN, B.; ZHANG, F.; LU, D.; WU, B.; GUAN, B.; YONGJI, W.; LOU, J.-G. Large language models meet nl2code: A survey. In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. [S.l.: s.n.], 2023. p. 7443–7464.

ZHENG, Z.; NING, K.; ZHONG, Q.; CHEN, J.; CHEN, W.; GUO, L.; WANG, W.; WANG, Y. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, Springer, v. 30, n. 2, p. 50, 2025.

ZHONG, L.; WANG, Z. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In: *Proceedings of the AAAI conference on artificial intelligence*. [S.l.: s.n.], 2024. v. 38, n. 19, p. 21841–21849.

Apêndices

APÊNDICE A – Prompt completo utilizado na detecção

Este apêndice reproduz, na íntegra, o *prompt* submetido a ambos os modelos durante a etapa de detecção (Seção 3.4). Ele foi redigido em inglês e mantido idêntico para os dois modelos, de modo a assegurar uma comparação justa. Os marcadores `${codigoAntes}` e `${codigoDepois}`, ao final, são substituídos em tempo de execução pelas versões anterior e posterior do código-fonte analisado.

Listing A.1 – Prompt completo utilizado na detecção de refatorações.

```
#CONTEXT

Two source code snapshots are provided: a BEFORE version and an AFTER version of a
    transformation applied to one or more Java source files. The goal is to determine whether
    the transformation constitutes a refactoring according to Martin Fowler's formal definitions
    .

#PERSONA

Act as a Software Engineering expert specializing in source code evolution, with deep knowledge
    of Martin Fowler's formal refactoring catalogue.

#TASK

Identify which refactoring operations, if any, were applied when transforming the BEFORE code
    into the AFTER code.

#REQUIREMENTS

Formal definition of refactoring (Fowler, 1999):
"Refactoring is the process of changing a software system in such a way that it does not alter
    the external behavior of the code yet improves its internal structure."

Recognized refactoring types, their formal definitions, mechanics, and structural signals:


Each operation below is described using Fowler's own material: his definition, the motivation
he gives for applying it, the mechanics he prescribes, and a synthetic example illustrating it.


1. Extract Method

Definition:
"You have a code fragment that can be grouped together. Turn the fragment into a method whose
    name explains the purpose of the method."

Motivation (Fowler):
Fowler favours short methods whose names announce their purpose. He extracts a fragment
whenever a method is longer than it ought to be, or whenever a piece of code needs a comment
```

to explain what it does -- in which case the comment becomes the name of the new method. The aim is to separate the INTENTION of a piece of code (what it accomplishes, captured by the method's name) from its IMPLEMENTATION (how it accomplishes it, now tucked inside the new method). Code grouped this way reads at a higher level of abstraction, the named fragment can be reused, and the smaller method is easier to understand and to override.

Mechanics:

1. Create a new method and name it after the intention of the code (name it by what it does, not by how it does it).
2. Copy the fragment of code from the source method into the body of the new method.
3. Examine the fragment for references to variables that are local in scope to the source method:
   - a variable that the fragment only reads becomes a parameter of the new method;
   - a variable that is declared inside the fragment and still used by the source method afterwards must be returned by the new method;
   - if the fragment assigns to more than one variable that is needed later, the fragment is not straightforwardly extractable as a single method and needs further reworking first.
4. Replace the fragment in the source method with a call to the new method, passing it the parameters and assigning back any returned value.
5. Test that behaviour is unchanged.

Synthetic example:

Before refactoring:

```
class Order {

    public double computeTotal(List<Item> items) {
        double total = 0;
        for (Item item : items) {
            total += item.getPrice() * (1 - item.getDiscount());
        }
        System.out.println("Total: " + total);
        return total;
    }
}
```

After refactoring:

```
class Order {

    public double computeTotal(List<Item> items) {
        double total = sumDiscountedPrices(items);
        System.out.println("Total: " + total);
        return total;
    }

    private double sumDiscountedPrices(List<Item> items) {
        double total = 0;
        for (Item item : items) {
            total += item.getPrice() * (1 - item.getDiscount());
        }
        return total;
    }
}
```

2. Rename Method

Definition:
"The name of a method does not reveal its purpose. Change the name of the method."

Motivation (Fowler):
For Fowler, a method's name should reveal its intention, so a reader understands what it does without reading its body. When the name communicates poorly -- it is cryptic, too generic, or out of step with what the body actually does -- the name is changed to one that better describes the method's purpose. (In the 2nd edition this is folded into "Change Function Declaration", where Fowler notes that improving a declaration may also involve adjusting the parameter list.)

Mechanics:
1. Declare a method with the new name.
2. Copy the body of the old method into the new one.
3. Find all callers of the old method and adjust them to call the new one.
4. Remove the old method (if it is part of a published interface, keep it as a delegating method instead).

Synthetic example A:

Before refactoring:

```
class Account {
    public double calc(double amount, double rate) {
        return amount * rate / 100;
    }
}
```

After refactoring:

```
class Account {
    public double calculateInterest(double amount, double rate) {
        return amount * rate / 100;
    }
}
```

Synthetic example B:

Before refactoring:

```
class Report {
    public List<Row> getData() {
        return rows.stream().filter(Row::isVisible).collect(toList());
    }
}
```

After refactoring:

```
class Report {
    public List<Row> fetchVisibleRows() {
        return rows.stream().filter(Row::isVisible).collect(toList());
    }
}
```

3. Move Method

Definition:
"A method is using or used by more features of another class than the class on which it is defined. Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether."

Motivation (Fowler):
Fowler moves a method when it is, or will be, using or being used by the features of another class more than those of the class on which it is currently defined. The guiding idea is to reduce coupling by putting a method together with the data and behaviour it most depends on. He chooses the destination by looking at where the features the method calls, and the data it reads, actually live: the method belongs in the class it talks to most. The method itself is already a complete, named method; what the refactoring changes is the class that hosts it.

Mechanics:

1. Examine all the features (fields and methods) that the source method uses, and consider which of them should be moved along with it.
2. Check whether the method is also declared in superclasses or subclasses of the source class.
3. Declare the method in the target class.
4. Copy the body of the source method into the target method and adjust it to work in its new home; where the method relied on data or methods of its original class, reach them through a field of the target class or through a parameter passed in.
5. Decide how the target method will obtain the object it needs to operate on.
6. Turn the source method into a delegating method that forwards to the target, or remove it entirely and redirect its callers to the target method.
7. Test that behaviour is unchanged.

Synthetic example (two files provided):

Before refactoring:

```
// File 1: Customer.java
class Customer {

    private Account account;

    public double calculateMonthlyCharge() {
        return account.getBalance() * account.getRate() / 12;
    }
}

// File 2: Account.java
class Account {

    private double balance;
    private double rate;

    public double getBalance() { return balance; }
    public double getRate()    { return rate; }
}
```

After refactoring:

```
// File 1: Customer.java
class Customer {

    private Account account;

    // removed entirely, or left as a delegation:
    // public double calculateMonthlyCharge() { return account.calculateMonthlyCharge(); }
}
```

```
    // File 2: Account.java
    class Account {

        private double balance;
        private double rate;

        public double getBalance() { return balance; }
        public double getRate()    { return rate; }

        public double calculateMonthlyCharge() {
            return balance * rate / 12;
        }
    }

#CONSTRAINTS

- Report only refactorings that strictly match the formal definitions above.
- Do not report a change as a refactoring if it introduces new logic, fixes a bug, alters
    behavior, or modifies the observable output of the method.
- Do not infer behavior or intent beyond what is provided in the code.
- If the same transformation can be explained by multiple types, choose the one that best
    represents the primary structural change.
- If no refactoring is detected, return an empty list.

#OUTPUT

Return ONLY valid JSON, with no explanations or text outside it, in the following format:

{
  "refactorings": [
    {
      "type": "<Extract Method | Rename Method | Move Method>",
      "method_before": "<full method signature as it appears in BEFORE, or null if the method
    did not exist>",
      "method_after": "<full method signature as it appears in AFTER, or null if the method was
    removed>",
      "description": "<one sentence explaining what structural transformation occurred and why
    it matches Fowler's definition>"
    }
  ]
}

#INPUT FORMAT

Each of the BEFORE and AFTER sections below may contain ONE or MULTIPLE Java source files.
When multiple files are present, they are concatenated in the following format:

  Arquivo 1 (FileName1.java): <full source of file 1>, Arquivo 2 (FileName2.java): <full source
    of file 2>, ...

#ACTION

Analyze the code snippets below and produce the output JSON.

BEFORE Code:
${codigoAntes}
```

```
AFTER Code:
${codigoDepois}
```

Declaração de Uso de Inteligência Artificial Generativa

Durante a preparação deste documento, eu, Gabriel Henrique Moreira Rocha, estudante de graduação em Ciência da Computação da Universidade Federal de Ouro Preto, declaro o uso de IAGen Claude Opus 4.8, da Anthropic para revisão textual e geração de códigos LaTeX.

Após o uso desta ferramenta/modelo/serviço, eu revisei e editei o conteúdo em conformidade com os princípios éticos para uso de IAGen (Resolução CONPEP nº 144) e com os acordos estabelecidos com o(a) orientador(a) desta pesquisa. Dessa forma, assumo total responsabilidade pelo conteúdo da publicação.