Avaliando LLMs para Detecção de Refatorações: Um Estudo Preliminar

Gabriel Henrique Rocha
Departamento de Computação (DECOM), Instituto de Ciências Exatas e Biológicas (ICEB), Universidade Federal de Ouro Preto (UFOP)
Ouro Preto, Brasil
gabriel.hmr@aluno.ufop.edu.br

Diego Demétrio
Departamento de Computação (DECOM), Instituto de Ciências Exatas e Biológicas (ICEB), Universidade Federal de Ouro Preto (UFOP)
Ouro Preto, Brasil
diego.demetrio@aluno.ufop.edu.br

Aline Brito
Departamento de Computação (DECOM), Instituto de Ciências Exatas e Biológicas (ICEB), Universidade Federal de Ouro Preto (UFOP)
Ouro Preto, Brasil
aline.brito@ufop.edu.br

Rodrigo Brito
Departamento de Computação (DCC), Instituto de Ciências Exatas (ICEX), Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, Brasil
brito.rodrigo@dcc.ufmg.br

Resumo

Ferramentas para detecção de refatorações, como RefactoringMiner e RefDiff, apresentam alta precisão, porém dependem de análises sintáticas e estruturais, o que requer esforço e tempo para extensão para múltiplas linguagens de programação. Visando melhorar este cenário, este trabalho investiga a viabilidade do uso de *Large Language Models* (LLMs) na detecção automática de refatoração. Para tanto, propõe-se um método no qual LLMs analisam commits e detectam refatorações dos tipos *Extract Method*, *Move Method* e *Rename Method*. A abordagem foi avaliada sobre um conjunto de 150 operações reais extraídas de um oráculo de refatorações amplamente utilizado na literatura. Utilizam-se dois LLMs, GPT e Claude Haiku 4.5. Os resultados revelam perfis complementares: um modelo favorece a precisão (0,73) e o outro, a cobertura (*recall* de 0,72), enquanto a combinação de suas saídas eleva o *recall* a 0,77. Esses achados sugerem que os LLMs constituem uma abordagem promissora.

Palavras-chave

Refatoração, LLM, Detecção de refatorações, Prompt Engineering

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 [12, 18]. 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 [15, 17] e RefDiff [3, 11, 13]. 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.

Nesse contexto, abordagens baseadas em LLMs surgem como alternativa promissora, capazes de capturar intenção e significado em alterações de código [14], além de facilitar a portabilidade para outras linguagens de programação [20]. Estudos recentes demonstram que LLMs têm alcançado alto desempenho em atividades como sumarização, explicação de trechos e interpretação de mudanças [1, 10, 14, 19]. Esses dados sugerem um potencial para apoiar processos tradicionalmente complexos no desenvolvimento e análise de *software*.

Este trabalho 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 [16]. 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: enquanto um modelo favorece a precisão, o outro apresenta maior cobertura, e a combinação de suas saídas eleva o *recall*. A principal contribuição deste trabalho é fornecer evidências empíricas de que os LLMs podem constituir uma técnica complementar para a detecção de refatorações. Além disso, no melhor do nosso conhecimento, este é o primeiro estudo a investigar o uso de LLMs sob essa perspectiva.

Especificamente, o estudo responde a três questões de pesquisa: *QP1*, como os LLMs se comparam quanto ao desempenho; *QP2*, como o desempenho varia entre os tipos de operação; e *QP3*, quais as principais razões para a imprecisão observada.

O restante deste artigo está organizado da seguinte forma. A Seção 2 apresenta os trabalhos relacionados. A Seção 3 descreve a metodologia adotada, incluindo a construção do *ground truth*, o *promp* proposto, o processo de detecção via LLMs e o protocolo de avaliação. A Seção 4 responde às três questões de pesquisa. A Seção 5 discute os principais achados do estudo. Por fim, a Seção 6 apresenta as ameaças à validade deste estudo e Seção 7 conclui o trabalho, prospectando futuras linhas de pesquisa.

2 Trabalhos Relacionados

A detecção de operações de refatoração permite o desenvolvimento de ferramentas e técnicas modernas. Por exemplo, a identificação de refatorações pode ajudar no processo de revisão de código, melhorando a visualização do *diff textual* em plataformas como o GitHub [4] e a navegação no histórico do código-fonte [2].

Neste contexto, RefDiff analisa o histórico de versões e aplica heurísticas para identificar 13 tipos de refatorações em Java e JavaScript, com uma precisão de 100% e *recall* de 88% [11, 13]. A extensão RefDiff4Go, desenvolvida para projetos implementados na linguagem Go, é capaz de detectar 13 tipos de refatoração. A ferramenta obteve uma precisão de 92% e 80% de *recall* [3].

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 [17].

Dois anos depois, o RefactoringMiner 2.0 acrescentou suporte à detecção em nível de submétodo, dispensando limites de similaridade de código e identificando operações aninhadas em um único *commit*. Nos experimentos, atingiu precisão de 99,6% e *recall* de 94%, sendo em média 2,6 vezes mais rápida que a segunda ferramenta mais eficiente [15].

Entretanto, essas ferramentas requerem um esforço significativo para serem estendidas a outras linguagens, exigindo a análise de classes específicas e a implementação de *parsers*. No melhor do nosso conhecimento, estratégias baseadas em LLMs têm sido pouco exploradas neste contexto da refatoração de código.

Cordeiro et. al. [6] 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 permanecem pouco exploradas com LLMs.

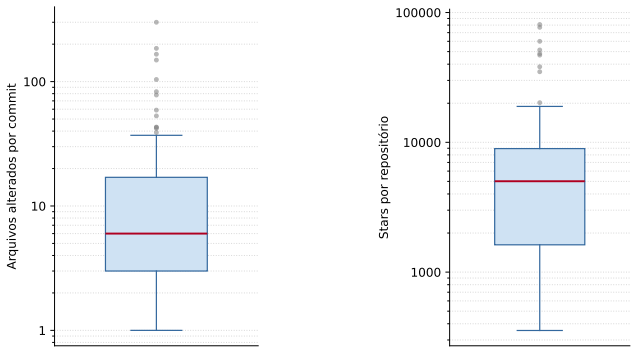
3 Metodologia

3.1 Definição Do *Ground Truth* De Refatorações

Para a avaliação deste estudo, utilizou-se um oráculo de refatorações amplamente adotado na literatura [13, 15], composto por operações reais realizadas por desenvolvedores em projetos GitHub e posteriormente validadas manualmente por Tsantalis e colaboradores [16]. A Figura 1 resume as principais características do *dataset*.

A mediana do número de estrelas é de 5.013, com o terceiro quartil em 8.947 (Figura 1b). Em relação ao tamanho dos *commits*, a mediana é de 6 arquivos modificados por *commit*, 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 conjunto de dados inclui projetos relevantes, como `spring boot` (80.860 estrelas), `elasticsearch` (76.937), `spring framework` (60.012) e `google guava` (51.475).

Desse oráculo extraiu-se um subconjunto balanceado, composto apenas por operações classificadas como *True Positive* (TP): 50 de cada um dos três tipos analisados, priorizando a diversidade de projetos e *commits*.



(a) Arquivos por *commit* **(b) Estrelas por repositório**

Figura 1: Caracterização do conjunto de dados (eixo vertical em escala logarítmica).

O conjunto final reúne 150 operações de refatoração, em 150 *commits* de 81 projetos distintos e 141 desenvolvedores. A maioria dos projetos (72%) contribui com apenas um *commit*, reduzindo a concentração de casos em poucos repositórios e aumentando a heterogeneidade das operações analisadas.

Como o oráculo registra apenas a descrição textual de cada refatoração, um *script* infere o caminho dos arquivos da operação-alvo e recupera seu conteúdo, via API do GitHub. O modelo recebe os arquivos de interesse, eliminando os demais arquivos do commit que não envolvem a refatoração.

3.2 Definição Do Prompt De Comandos Para Detecção De Refatorações

O *prompt* enviado ao LLM é estruturado em sete componentes principais [5], compreendendo contexto, persona, tarefa, requisitos, restrições, formato de entrada, saída e ação. Cada um deles é detalhado a seguir. Duas decisões guiaram sua definição: descrever cada tipo isoladamente, sem definições por contraste, que induzem a classificação por eliminação; e aprofundar motivação e mecânica dos tipos com mais erros [7], mantendo profundidade equivalente entre os três. O refinamento foi conduzido sobre 9 operações de validação (3 por tipo), de *commits* e repositórios disjuntos das 150 avaliadas. Esta etapa elevou o acerto de tipos de refatorações do GPT de 4/9 para 7/9 e o de método de 2/9 para 5/9; o Claude manteve-se estável (8/9 e 6/9).

3.2.1 Componente I: Definição do Contexto. Na primeira parte do *prompt* define-se o contexto, isto é, fornecemos ao modelo as informações necessárias para compreender a tarefa. 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 [7, 8].

3.2.2 Componente II: Persona. Na segunda parte do prompt, define-se a persona, isto é, o papel que modelo irá desempenhar. Neste caso, o *prompt* define um especialista em Engenharia de Software com conhecimento em evolução de código-fonte e refatoração [7, 8].

```
#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 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.
```

3.2.3 Componente III: Tarefa. Em seguida, define-se a ação principal que o modelo deve executar. Neste estudo, a tarefa consiste em identificar quais operações de refatoração, se houver, foram aplicadas na transformação do código.

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

3.2.4 Componente IV: Requisitos. O quarto componente do *prompt* define os requisitos para a detecção das operações de refatoração. Esta seção inclui a definição formal de refatoração e descreve os três tipos considerados neste estudo (*Extract Method*, *Move Method* e *Rename Method*). Com base no catálogo de refatorações proposto por Fowler, para cada tipo de refatoração são fornecidos a definição formal, a mecânica da refatoração e um exemplo didático.

```
#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. [Def.], [Mechanics], [Example]

2. RENAME METHOD. [Def.], [Mechanics], [Example]

3. MOVE METHOD. [Def.], [Mechanics], [Example]
```

3.2.5 Componente V: Restrições. A seção `#CONSTRAINTS` define as restrições que devem ser observadas pelo modelo durante a identificação das operações de refatoração. Especificamente, o modelo é instruído a reportar apenas refatorações que atendam estritamente às definições formais apresentadas, sem inferir intenções ou comportamentos não evidenciados pelo código analisado.

3.2.6 Componente VI: Formatos de Entrada e Saída. Em seguida, são especificados os formatos de saída e de entrada por meio das seções `#OUTPUT FORMAT` e `#INPUT FORMAT`, respectivamente. A saída é padronizada no formato JSON. A entrada é composta por um ou mais arquivos Java concatenados, acompanhados de marcadores que indicam o estado de cada arquivo (por exemplo, criado ou removido) no *commit*.

3.2.7 Componente VII: Ação. Por fim, a seção `#ACTION` instrui o modelo a analisar os trechos de código fornecidos e produzir a saída no formato especificado. É nesta seção que os códigos correspondentes às versões `BEFORE` e `AFTER` de cada operação são efetivamente inseridos no *prompt* para análise.

3.3 Detecção De Refatoração Via LLMs

A detecção foi realizadas pelas APIs oficiais de dois LLMs de famílias distintas: GPT-5.4 (OpenAI, identificador `gpt-5.4`) e Claude Haiku 4.5 (Anthropic, identificador `claude-haiku-4-5-20251001`). A integração foi implementada em Node.js utilizando as bibliotecas clientes oficiais (`openai` e `@anthropic-ai/sdk`). Todas as execuções foram realizadas em junho de 2026.

Para cada operação, o mesmo *prompt*, acompanhado das versões anterior e corrente do código, foi enviado independentemente a cada modelo. As respostas foram geradas de forma isolada, sem influência entre os LLMs, garantindo uma comparação direta.

A `temperatura` foi fixada em 0,2 para ambos os modelos, por favorecer respostas mais determinísticas e reduzir a ocorrência de variações e alucinações [9]. Os demais parâmetros permaneceram em seus valores padrão, exceto o limite de *tokens* de saída do Claude, ampliado para `max_tokens` = 8192.

As respostas, em formato JSON, foram processadas por uma rotina de *parsing* tolerante e armazenadas separadamente para cada LLM, sendo posteriormente utilizadas na etapa de avaliação.

3.4 Avaliação E Síntese Dos Resultados

A avaliação consiste na comparação entre as refatorações detectadas pelos modelos e aquelas registradas no oráculo. A partir dessa análise, 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.

A precisão é definida como $TP/(TP+FP)$. Para o *recall*, reportam-se duas formulações, 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). Essa métrica indica quantas das operações-alvo foram identificadas pelos LLMs.

- *Recall sobre as detecções*: TP/(TP + FN). 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 corresponde à média harmônica entre a precisão e o *recall* sobre as detecções. Além disso, cada detecção foi classificada quanto à sua origem: operação-alvo, quando corresponde à refatoração selecionada do *commit*, ou detecção nova, quando o LLM identifica outra refatoração presente no mesmo *commit*. Por fim, os falsos positivos foram classificados manualmente de acordo com a transformação efetivamente realizada, permitindo caracterizar os principais tipos de erro.

4 Resultados

Esta seção apresenta e discute os resultados da detecção de operações de refatoração pelos dois LLMs avaliados, considerando o desempenho geral e por tipo. Adicionalmente, realiza-se uma inspeção manual para discussão das limitações da abordagem.

4.1 (QP1) Como Os LLMs Avaliados Se Comparam Quanto Ao Desempenho Na Detecção De Operações De Refatoração?

As Tabelas 1 e 2 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.

As duas formas de *recall* aprofundam 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.

Tabela 1: Detecção de refatorações pelo Claude Haiku 4.5

Tipo	TP	FP	FN	Prec.	Rec._{alv}	Rec._{det}	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**

Tabela 2: Detecção de refatorações pelo GPT-5.4

Tipo	TP	FP	FN	Prec.	Rec._{alv}	Rec._{det}	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**

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 (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 resultado 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 um resultado inverso, com precisão de 0,867 (apenas 4 falsos positivos) e *recall* de somente 0,380.

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).

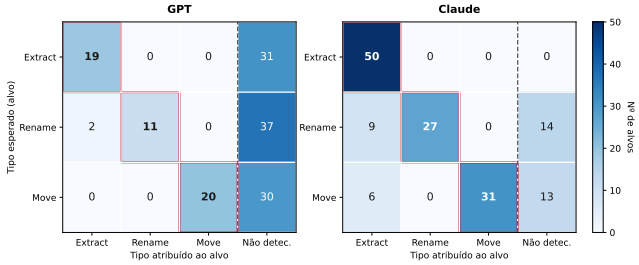
Já o Move Method apresentou o desempenho mais equilibrado com precisão de 0,519 no Claude e 0,611 no GPT; sua natureza distribuída em dois arquivos exige raciocínio entre múltiplos *snapshots*, o que reduz a cobertura, mas torna as detecções mais criteriosas.

4.2.1 Confusão entre Tipos. Para entender se os alvos perdidos foram rotulados com o tipo errado ou não detectados, a Figura 2 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 ausência de detecção. No Claude, dos 42 alvos perdidos, 27 não foram detectados e apenas 15 foram rotulados incorretamente.

Quando a confusão ocorre, contudo, ela tem direção única: todos os rótulos trocados convergem para o Extract Method (15 casos

Figura 2: Matrizes de confusão por LLM. A coluna "Não detectado" registra os alvos omitidos.



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.

4.3 (QP3) Quais São As Principais Razões Para Imprecisão Dos LLMs Na Detecção De Refatorações?

A precisão isolada informa quantos falsos positivos cada LLM produz, mas não por que eles ocorrem. Para responder a essa pergunta, 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 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.
- *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 errado.
- *CTP no oráculo*: a detecção corresponde a uma operação classificada no oráculo como CTP (*Commented True Positive*), sobre a qual os validadores registraram ressalvas; e não confirmaram em definitivo. Pela regra conservadora adotada, conta como FP.
- *FP no oráculo*: a detecção corresponde apenas a uma operação já classificada pelo próprio oráculo como falso positivo.

Cada detecção foi comparada às operações que o oráculo registra para o mesmo *commit*: havendo correspondência, a causa decorre do rótulo já atribuído pelo oráculo; não havendo, a detecção foi inspecionada no *diff*. A classificação é, portanto, objetiva em 124 das 164 detecções e depende de julgamento nas 40 restantes (24,4%). O processo foi conduzido por um dos autores, e a planilha resultante integra o pacote de artefatos.

Tabela 3: Anatomia dos 164 falsos positivos por tipo de refatoração atribuído pelo LLM e causa

Tipo atribuído	Fora do escopo	Sem cor.	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**

A interpretação da precisão é de 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 resumo, 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 excluiu.

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.

5 Discussão

Desempenho de LLMs para detecção de refatorações. Conforme reportado anteriormente, nenhum dos dois LLMs reúne, isoladamente, alta precisão e alta cobertura, quando comparado a ferramentas especializadas, como RefDiff e RefactoringMiner. Ainda assim, os resultados evidenciam o potencial de evolução dessa abordagem com modelos e *prompts* mais avançados.

Desenvolvimento de ferramentas multi-linguagem. Por dispensarem *parsers* específicos, os LLMs são em princípio aplicáveis a novas linguagens sem reimplementação, ao passo que ferramentas tradicionais exigem um *parser* e regras próprias para cada linguagem. Ainda que essa portabilidade permaneça uma conjectura a ser testada. O estudo sugere ainda que ferramentas existentes podem se beneficiar da integração com LLMs, desde que as definições formais e a especificação da tarefa sejam fornecidas com clareza.

Abordagens Híbridas para detecção de refatorações. Os resultados mostram que, isoladamente, os LLMs ainda apresentam desempenho inferior ao de ferramentas especializadas, como RefDiff e RefactoringMiner, não constituindo um substituto para essas abordagens. Entretanto, os modelos exibem perfis complementares: o GPT prioriza precisão, enquanto o Claude alcança maior cobertura. Como mostra Tabela 4, 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.

Tabela 4: Precisão e Recall por tipo de abordagem

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%

6 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. Para reduzir a variação, empregou-se temperatura 0,2 e o mesmo *prompt* e *pipeline* para ambos os LLMs; ainda assim, cada caso foi executado uma única vez, sem estimativa de variância. O desempenho é sensível ao *prompt*, refinado sobre um conjunto de validação distinto do de avaliação.

Validade externa. O estudo restringe-se a Java e a três tipos de refatoração (Extract, Rename e Move Method) e a detecção analisa apena os arquivos de interesse no commit. Portanto, a generalização para outras linguagens e para o catálogo completo não foi verificada empiricamente. A amostra, embora restrita a 150 operações, é balanceada e diversa, provindo de 81 repositórios e 141 autores distintos. O uso de dois LLMs de famílias distintas evita conclusões atadas a um único LLM, e a abordagem, por prescindir de *parser* específico, é em princípio aplicável a outras linguagens. Ressalte-se que os projetos são majoritariamente *open-source* maduros e que os LLMs correspondem a versões pontuais no tempo.

Validade de conclusão. Os valores de precisão e *recall* do RefactoringMiner e do RefDiff provêm de suas avaliações originais, sobre conjuntos distintos, e não foram reexecutados sobre as 150 operações aqui analisadas; a comparação é, portanto, indireta, apresentada como referência de magnitude e amparada no fato de o conjunto derivar do mesmo oráculo de refatorações [16].

7 Conclusões

Este trabalho investigou a viabilidade do uso de LLMs na detecção automática de refatorações em nível de método, avaliado 150 refatorações reais extraídas de um oráculo consolidado da literatura. Os resultados evidenciaram perfis complementares entre os modelos avaliados: o GPT-5.4 apresentou maior precisão, enquanto o Claude Haiku 4.5 obteve maior cobertura. Além disso, verificou-se que a principal limitação dos LLMs é a omissão de refatorações, e não sua classificação incorreta. A combinação das respostas aumenta o *recall*, indicando potencial para abordagens híbridas.

Embora o desempenho ainda seja inferior ao de ferramentas especializadas, os resultados indicam que os LLMs constituem uma abordagem complementar promissora. No melhor do nosso conhecimento, é o primeiro estudo a avaliar LLMs nesse contexto. Como trabalhos futuros, pretende-se investigar novas estratégias de *prompt engineering*, utilizando o *prompt* proposto como *baseline*, ampliar a avaliação para outras linguagens de programação e expandir o conjunto de operações de refatoração analisadas.

Disponibilidade de Artefatos.

Os *scripts*, *dataset* e *prompt* completo encontram-se disponíveis em: https://github.com/alinebrito/ise-cbsoft-llm-refactoring-detector

Referências

[1] Lucas Amaral, Eliakim Gama, Matheus Paixao, and Lucas Aguiar. 2025. Evaluating Large Language Models on the Classification of Different Technical Debt Types in Stack Overflow Discussions. In *IV Workshop Brasileiro de Engenharia de Software Inteligente (ISE)*.

[2] Aline Brito, Andre Hora, and Marco Tulio Valente. 2021. Characterizing Refactoring Graphs in Java and JavaScript Projects. *Empirical Software Engineering* 26 (2021).

[3] Rodrigo Brito and Marco Tulio Valente. 2020. RefDiff4Go: Detecting Refactorings in Go. In *14th Brazilian Symposium on Software Components, Architectures, and Reuse (SBCARS)*. 101–110.

[4] Rodrigo Brito and Marco Tulio Valente. 2021. RAID: Tool Support for Refactoring-Aware Code Reviews. In *29th IEEE/ACM International Conference on Program Comprehension (ICPC)*. 265–275.

[5] Ivan Moura Campos. 2026. How to Write Well-Structured Prompts for LLMs. https://imcampos195.github.io/MyMumblings/Tutorial--EN-US.html. Online; accessed April 2026.

[6] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2025. An Empirical Study on the Code Refactoring Capability of Large Language Models. *ACM Transactions on Software Engineering and Methodology (TOSEM)* (2025).

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

[8] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley.

[9] Lujun Li, Lama Sleem, Geoffrey Nichil, Radu State, et al. 2025. Exploring the impact of temperature on large language models: Hot or cold? *Procedia Computer Science* 264 (2025), 242–251.

[10] Amália Melo, Lucas Almeida, and Lina Garcés. 2025. 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)*. 37–42.

[11] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802.

[12] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *24th International Symposium on the Foundations of Software Engineering (FSE)*. 858–870.

[13] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *14th International Conference on Mining Software Repositories (MSR)*. 269–279.

[14] Luciana Lourdes Silva, Janio Rosa da Silva, Joao Eduardo Montandon, Marcus Andrade, and Marco Tulio Valente. 2024. Detecting Code Smells using ChatGPT: Initial Insights. In *18th International Symposium on Empirical Software Engineering and Measurement*. 400–406.

[15] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2020. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering (TSE)* (2020).

[16] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and A. Ketkar. 2022. Refactoring oracle. http://refactoring.encs.concordia.ca/oracle. Online; accessed January 2022.

[17] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *40th International Conference on Software Engineering (ICSE)*. 483–494.

[18] Marco Tulio Valente. 2020. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade.* Independente.

[19] Daoguang Zan, Bei Chen, Fengji Zhang, Dianjie Lu, Bingchao Wu, Bei Guan, Wang Yongji, and Jian-Guang Lou. 2023. Large language models meet nl2code: A survey. In *61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 7443–7464.

[20] Li Zhong and Zilong Wang. 2024. Can llm replace stack overflow? a study on robustness and reliability of large language model code generation. In *AAAI conference on artificial intelligence*, Vol. 38. 21841–21849.