Universidade Federal de Pernambuco

Centro de Informática

Engenharia da Computação

Automatização da Refatoração *Extract Method* com DeepSeek R1

Trabalho de Conclusão de Curso

por

Bianca Carneiro da Cunha Nunes Ferreira

Orientador: Prof. Márcio Lopes Cornélio

Recife, Abril / 2025

Bianca Carneiro da Cunha Nunes Ferreira

Automatização da Refatoração *Extract Method* com DeepSeek R1

Monografia apresentada ao Centro de Informática da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do Título de Bacharel em Engenharia da Computação.

Orientador: Prof. Márcio Lopes Cornélio

Recife

2025

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

Ferreira, Bianca Carneiro da Cunha Nunes.
Automatização da Refatoração Extract Method com DeepSeek R1 / Bianca Carneiro da Cunha Nunes Ferreira. - Recife, 2025.
44 p : il., tab.

Orientador(a): Márcio Lopes Cornélio
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.
Inclui referências.

1. Refatoração de código. 2. DeepSeek-R1. 3. Extract Method. 4. Large Language Models. 5. Chain of Thought. 6. Code Smells. I. Cornélio, Márcio Lopes. (Orientação). II. Título.

000 CDD (22.ed.)

BIANCA CARNEIRO DA CUNHA NUNES FERREIRA

Automatização da Refatoração *Extract Method* com DeepSeek R1

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

Aprovado em: 04 de Abril de 2025

Banca Examinadora:

Prof. Dr. Prof. Márcio Lopes Cornélio (Orientador)
Centro de Informática da UFPE

Prof. Dr. Breno Miranda
Centro de Informática da UFPE

Recife

2025

Agradecimentos

O desenvolvimento desse trabalho contou com o apoio e a ajuda de diversas pessoas; agradeço especialmente:

Ao professor Márcio Lopes Cornélio, por sua orientação ao longo dos quatro meses de desenvolvimento deste projeto, sempre se mostrando disponível para discutir ideias e esclarecer dúvidas.

Aos professores do curso de Engenharia da Computação da UFPE, cujo ensino e suporte foram fundamentais para que eu adquirisse os conhecimentos necessários para concretizar este projeto.

Aos meus pais, Fábio Nunes e Elza Carneiro da Cunha, por me proporcionarem educação, valores e, acima de tudo, pelo incentivo constante na busca pelo conhecimento.

Às minhas irmãs, Anna Flávia e Amanda Carneiro da Cunha, cuja trajetória e exemplo me inspiraram a vida toda, moldando a mulher determinada que sou hoje, pronta para enfrentar desafios e conquistar meus objetivos.

Aos meus amigos, Gabriel Freitas, Vinícius Araújo, Matheus Carvalheira, Renan Melo e Eduarda Gomes, por tornarem meus momentos felizes ainda mais especiais e minhas dificuldades mais leves, sempre oferecendo apoio, compreensão e ajudando a aliviar minha ansiedade nos últimos meses.

Ao meu amigo e mentor Felipe dos Santos, por guiar minha trajetória profissional, compartilhar insights valiosos para este projeto e se dispor a assistir aos meus treinos de defesa.

Aos amigos do trabalho, que tornaram meus dias em São Paulo mais alegres. Em especial, à Paula Hemsi, pela parceria e por se disponibilizar como plateia nos meus treinos.

A todos, meu mais sincero agradecimento.

A inteligência artificial está em toda parte.
Não é aquela coisa grande e assustadora do futuro.
A IA está aqui conosco.

Fei-Fei Li

RESUMO

Este trabalho investiga o uso de Large Language Models (LLMs) para a automatização da refatoração Extract Method. O estudo utiliza modelos da família DeepSeek-R1, especificamente as variantes Qwen 1.5B, Qwen 7B e LLaMa 8B, para sugerir refatorações em projetos open-source Java. Além disso, é analisado o Chain of Thought (CoT) gerado pelos modelos para identificar se os modelos relacionam code smells à necessidade de refatoração.

Para validar a eficácia das refatorações, são realizados testes estatísticos com métricas como Levenshtein Ratio, número de palavras repetidas no CoT e proximidade das sugestões ao baseline. Os resultados indicam que os modelos sugeriram refatorações, em sua maioria, desnecessárias. O Qwen 7B apresentou o CoT mais conciso e menos redundante, mas os CoT não demonstraram uma associação entre code smells e as refatorações sugeridas.

Esse estudo destaca tanto o potencial quanto as limitações dos LLMs na refatoração automática de código. Para trabalhos futuros, é proposto aprimorar os prompts, explorar modelos mais avançados e executar os testes em outros datasets, com outras linguagens de programação, ou com foco nos maus cheiros ou em outras refatorações.

Palavras-chave: Refatoração de código, Extract Method, Large Language Models, DeepSeek-R1, Chain of Thought, Code Smells.

ABSTRACT

This study investigates the use of Large Language Models (LLMs) for automating the Extract Method refactoring. The research utilizes models from the DeepSeek-R1 family, specifically the Qwen 1.5B, Qwen 7B, and LLaMa 8B variants, to suggest refactorings in open-source Java projects. Additionally, the Chain of Thought (CoT) generated by the models is analyzed to determine whether they associate code smells with the need for refactoring.

To validate the effectiveness of the refactorings, statistical tests are conducted using metrics such as the Levenshtein Ratio, the number of repeated words in the CoT, and the proximity of suggestions to the baseline. The results indicate that the models mostly suggested unnecessary refactorings. The Qwen 7B model produced the most concise and least redundant CoT; however, the CoT did not demonstrate an association between code smells and the suggested refactorings.

This study highlights both the potential and limitations of LLMs in automatic code refactoring. For future work, it is proposed to improve the prompts, explore more advanced models, and conduct tests on other datasets, with different programming languages, or focus on code smells or other refactorings.

Keywords: Code Refactoring, Extract Method, Large Language Models, DeepSeek-R1, Chain of Thought, Code Smells.

LISTA DE FIGURAS

Figura 1 Softmax 17
Figura 2 Representação do mecanismo de atenção single head com janela de contexto de três tokens. 18
Figura 3 Arquitetura do tranformers 20
Figura 4 Representação de todos os modelos DeepSeek-R1 e as etapas para seu treinamento 21
Figura 5 Mapeamento das refatorações sugeridas ao baseline 31
Figura 6 Número de refatorações por classe 34
Figura 7 Número de linhas extraídas 35
Figura 8 Levenshtein *ratio* dos nomes de função 36
Figura 9 Diferença do início e do final da extração comparado ao baseline 37
Figura 10 Nuvem de Palavras do CoT 38
Figura 11 Número de palavras do CoT 39
Figura 12 Repetições de tokens no CoT 40
Figura 13 Distribuições analisadas no teste de hipótese 41

LISTA DE TABELAS

Tabela 1 Conceitos fundamentais em PLN 16
Tabela 2 Code Smells mais populares relacionados ao extract method segundo [1].. 24
Tabela 3 Limites de CPU, MEM e GPU da partição short do apuana 29
Tabela 4 Número de refatorações 32
Tabela 5 Precision/Recall por modelo 32
Tabela 6 Métricas do número de refatorações por classe 33
Tabela 7 Testes de Mann-Whitney do *Levenshtein ratio* dos resultados 35
Tabela 8 Testes de D'Agostino-Pearson 37
Tabela 9 Testes de Wilcoxon para diferença do line_start entre o baseline e os experimentos 37
Tabela 10 Testes de Wilcoxon para diferença do line_end entre o baseline e os experimentos 37
Tabela 11 Dez palavras mais comuns do CoT 38
Tabela 12 Palavras mais repetidas no CoT e suas repetições máximas 40
Tabela 13 Testes de D'Agostino-Pearson para distribuições do número de palavras e número de palavras repetidas do CoT 41
Tabela 14 Testes de Mann-Whitney para distribuições do número de palavras do CoT 41
Tabela 15 Testes de Mann-Whitney para distribuições do número de palavras repetidas do CoT 41

LISTA DE SIGLAS

CoT	Chain of Though
CPU	Central Processing Unit
GPU	Graphics Processing Unit
GRU	Gated Recurrent Unit
PLN	Processamento de Linguagem Natural
LLM	Large Language Model
LSTM	Long Short-Term Memory
MoE	Mixture of Experts
RNN	recurrent neural network (Redes Neurais Recorrentes)
UFPE	Universidade Federal de Pernambuco

SUMÁRIO

1 **INTRODUÇÃO** 13
1.1 **Objetivos** 13
1.1.1 Objetivo Geral 13
1.1.2 Objetivos Específicos 13
1.2 **Organização de texto** 14
2 **REVISÃO BIBLIOGRÁFICA** 15
2.1 **Processamento de Linguagem natural (PLN)** 15
2.1.1 Mecanismo de atenção 15
2.1.2 Transformers 18
2.1.3 DeepSeek 19
2.2 **Code smells** 22
2.2.1 Code Smells Relacionados à Extract Method 22
2.2.2 Detecção de *Code Smells* 23
2.3 **Refatoração** 24
2.3.1 Extrair Função 25
2.3.2 Extrair método e LLMs 25
3 **SOLUÇÃO PROPOSTA E METODOLOGIA** 27
3.1 **Solução Proposta** 27
3.2 **Metodologia** 27
3.2.1 Dataset 28
3.2.2 Prompt 28
3.2.3 Apuana 29
3.2.4 Métricas 29
3.2.4.1 Levenshtein Ratio 29
3.2.4.2 Estatísticas 30
3.2.5 Pre-processamento dos Dados 30
3.2.5.1 Refatorações válidas 30

4 **ANÁLISE DE RESULTADOS** 33
4.1 **Extrair Método** 33
4.1.1 Análise estatística 33
4.2 **Chain of Thought (CoT)** 36
4.2.1 Tamanho e repetições por CoT 38
5 **CONCLUSÃO E TRABALHOS FUTUROS** 42
5.1 **Trabalhos futuros** 43

1 INTRODUÇÃO

Refatoração é o processo de modificar o código-fonte de um software sem alterar seu comportamento funcional, com o objetivo de melhorar o design e a estrutura do código. Embora o conceito de refatoração exista desde antes da década de 1990, ele foi consolidado com o livro *Refatoração: Aperfeiçoando o Design de Códigos Existentes* [2], de Martin Fowler.

Code smells (ou "Maus Cheiros"), um conceito introduzido por Kent Beck e popularizado por Martin Fowler [2], referem-se a indícios no código que indicam problemas de design ou manutenção. Segundo Fowler, os code smells auxiliam os desenvolvedores a identificar quando é necessário iniciar uma refatoração e quando ela está concluída.

Atualmente, existem diversos métodos para detectar code smells e realizar refatorações automaticamente. O objetivo deste estudo é propor uma solução para automatizar o processo da refatoração Extract Method a partir do uso de LLMs (Large Language Models) da família do DeepSeek R1 e analisar o Chain Of Thought das respostas geradas. Essa análise será realizada em busca de contextos relacionados a code smells, a fim de verificar se os modelos os relacionam às refatorações.

1.1 Objetivos

1.1.1 Objetivo Geral

O objetivo deste estudo é explorar o uso de Large Language Models (LLMs) da família DeepSeek R1, aplicando *zero-shot prompt engineering* para automatizar a refatoração de extração de método. Além disso, analisaremos as etapas seguidas pelo modelo para identificar a necessidade dessa refatoração em uma determinada classe por meio do Chain of Thought (CoT) retornado na resposta do DeepSeek.

1.1.2 Objetivos Específicos

Para atingir o objetivo principal, foram definidos os seguintes objetivos específicos:

- Identificar quais são os maus cheiros relacionados à refatoração Extract Method

- Verificar na literatura quais soluções para a automatização da refatoração Extract Method com LLMs existem:
- Verificar o comportamento dos modelos de baixo custo computacional da família do DeepSeek R1 na tarefa de refatoração Extract Method
- Analisar se as sugestões do Extract Method foram úteis
- Analisar o CoT dos modelos antes de aplicar a refatoração Extract Method

1.2 **Organização de texto**

O texto será organizado da seguinte forma:

- No Capítulo 2 (Revisão Bibliográfica), apresentaremos um panorama sobre o DeepSeek R1, o estado da arte da refatoração de código e os *code smells* que indicam a necessidade da refatoração Extrair Método.
- No Capítulo 3 (Solução proposta e Metodologia), descreveremos qual a solução proposta por esse estudo e o passo a passo para a sua reprodução.
- No Capítulo 4 (Análise de Resultados), examinaremos os resultados obtidos nos experimentos, discutindo suas implicações.
- Por fim, na Conclusão e Trabalhos Futuros, retomaremos as principais conclusões da análise e exploraremos possíveis direções para pesquisas futuras.

2 REVISÃO BIBLIOGRÁFICA

Nas próximas seções, abordaremos conceitos fundamentais de Processamento de Linguagem Natural (PLN), incluindo a evolução histórica que levou ao desenvolvimento do DeepSeek. Além disso, discutiremos os princípios da refatoração de código, os maus cheiros (*code smells*) que indicam a necessidade de melhorias no código e o estado da arte na automatização da refatoração.

2.1 Processamento de Linguagem natural (PLN)

O Processamento de Linguagem Natural (PLN) é um campo de pesquisa que utiliza técnicas estatísticas e de aprendizado de máquina para analisar, compreender e gerar linguagem humana [3]. A Tabela 1 lista alguns conceitos fundamentais para PLN que foram utilizados nesse estudo.

Os primeiros modelos de PLN adotavam abordagens estatísticas e utilizavam Redes Neurais Recorrentes (RNNs) [5] e suas variantes, como Long Short-Term Memory (LSTM) [5] e Gated Recurrent Units (GRU) [5]. No entanto, essas arquiteturas apresentavam limitações, especialmente no tratamento de sequências longas, pois perdiam o contexto conforme a distância entre os tokens aumentava.

Por exemplo, considere a sequência: "Eu estudo Engenharia da Computação. Eu gosto de longas caminhadas. Eu trabalho com Software."Em abordagens clássicas, a informação sobre "Engenharia da Computação"poderia se perder quando o modelo tentasse inferir o significado de "Software". Esse problema foi mitigado com a introdução da arquitetura Transformer, apresentada no artigo "Attention is All You Need" [6].

Nos próximas seções, serão discutidos o funcionamento dos Transformers, suas limitações e as otimizações propostas pelo DeepSeek R1 [7].

2.1.1 Mecanismo de atenção

O mecanismo de atenção (attention), introduzido no artigo "Attention is All You Need" [6], permite que um modelo de linguagem identifique e atribua pesos distintos a diferentes partes de uma sequência de entrada, destacando as informações mais relevantes para o contexto.

Conceito	Descrição
Tokens	Unidades mínimas de texto, como palavras ou subpalavras [3]
Embeddings	Representações vetoriais multidimensionais que capturam aspectos semânticos e sintáticos das palavras (tokens) [3]
Stopwords	Palavras comuns em um idioma, como preposições e artigos, geralmente removidas em tarefas de PLN [3]
Prompt	Entrada fornecida a um modelo de linguagem para guiar sua resposta, podendo variar de uma simples pergunta a um comando estruturado com contexto específico [3]
Few-shot	Prompt com alguns exemplos para que o modelo de linguagem consiga entender o padrão desejado da tarefa [4]
One-shot	Prompt com um exemplo para que o modelo de linguagem consiga entender o padrão desejado da tarefa [4]
Zero-shot	Prompt com nenhum exemplo [4]

Tabela 1: Conceitos fundamentais em PLN

O mecanismo inicia quando são calculados os vetores Q (query), K (key) e V (value), que podem ser interpretados como uma "pergunta"e uma "resposta", como:

- Q: "Esse substantivo está sendo modificado por alguma palavra?"
- K: "Sim, por mim"

Os vetores de Query (Q) e Key (K) são calculados a partir da multiplicação dos embeddings da entrada pelas matrizes obtidas durante o treinamento do modelo W_q, W_k e W_v, respectivamente. Em outras palavras, os vetores Q, K e V são calculados segundo as equações 2.1, 2.2 e 2.3, onde i é cada token da janela de contexto.

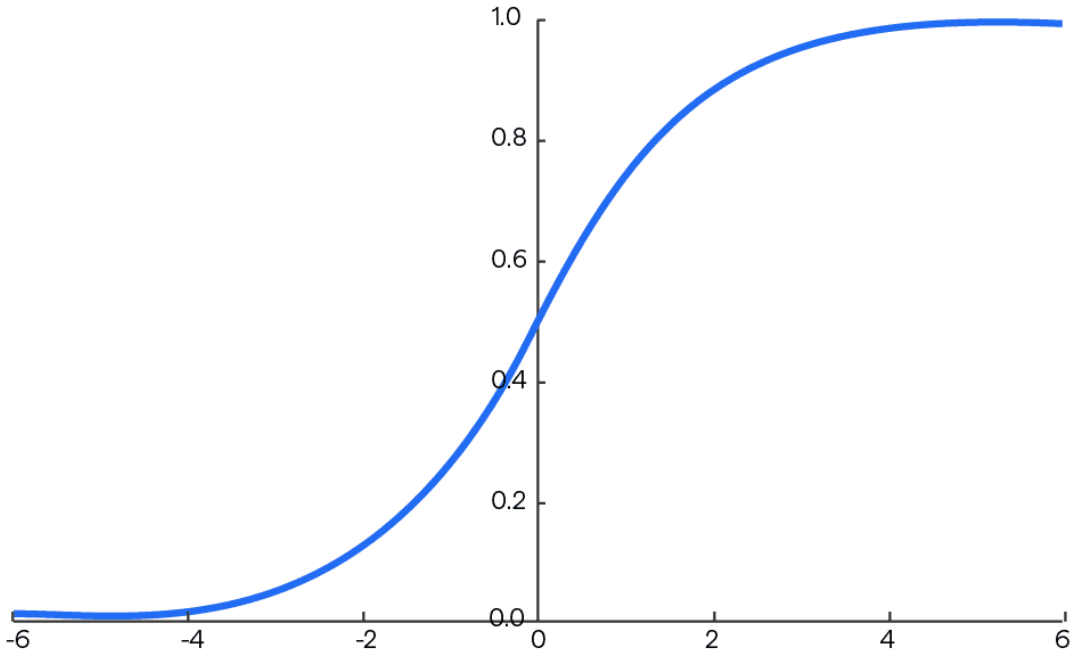
$$\vec{Q_i} = W_q \vec{E_i} \tag{2.1}$$

$$\vec{K_i} = W_k \vec{E_i} \tag{2.2}$$

$$\vec{V_i} = W_v \vec{E_i} \tag{2.3}$$

Para sabermos se uma Key é uma "resposta"para uma Query, calculamos a similaridade entre os vetores K e Q. A similaridade entre K e Q é computada utilizando o produto escalar, resultando em uma matriz de scores. É utilizado o produto escalar como uma simplificação da similaridade do cosseno, retirando o denominador. Esses scores são

Figura 1: Softmax



Fonte: botpenguin [8], 2025 (adaptada)

normalizados pela raiz quadrada da dimensão dos embeddings d_k para estabilizar os gradientes durante o treinamento. Em seguida, a função softmax (Figura 1) é aplicada sobre os scores para normalizá-los, convertendo-os em uma distribuição de probabilidades que determina a importância de cada token na sequência. Por fim, os vetores V são multiplicados com os valores resultantes da softmax, produzindo a saída da atenção, como descrito na equação 2.4.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{2.4}$$

O cálculo de atenção é computacionalmente custoso, pois W_q, W_k e W_v são parâmetros do modelo, e os modelos contemporâneos geralmente possuem múltiplas camadas de atenção, empregando o mecanismo de atenção multi-head. Modelos recentes, como o LLaMa 3.1 [9], possuem aproximadamente 405 bilhões de parâmetros, tornando o treinamento intensivo em recursos, apesar da possibilidade de paralelização.

Todo o processo do cálculo de atenção *single headed* pode ser visualizado na Figura 2. Em 1, os tokens são convertidos em suas representações vetoriais, conhecidas como

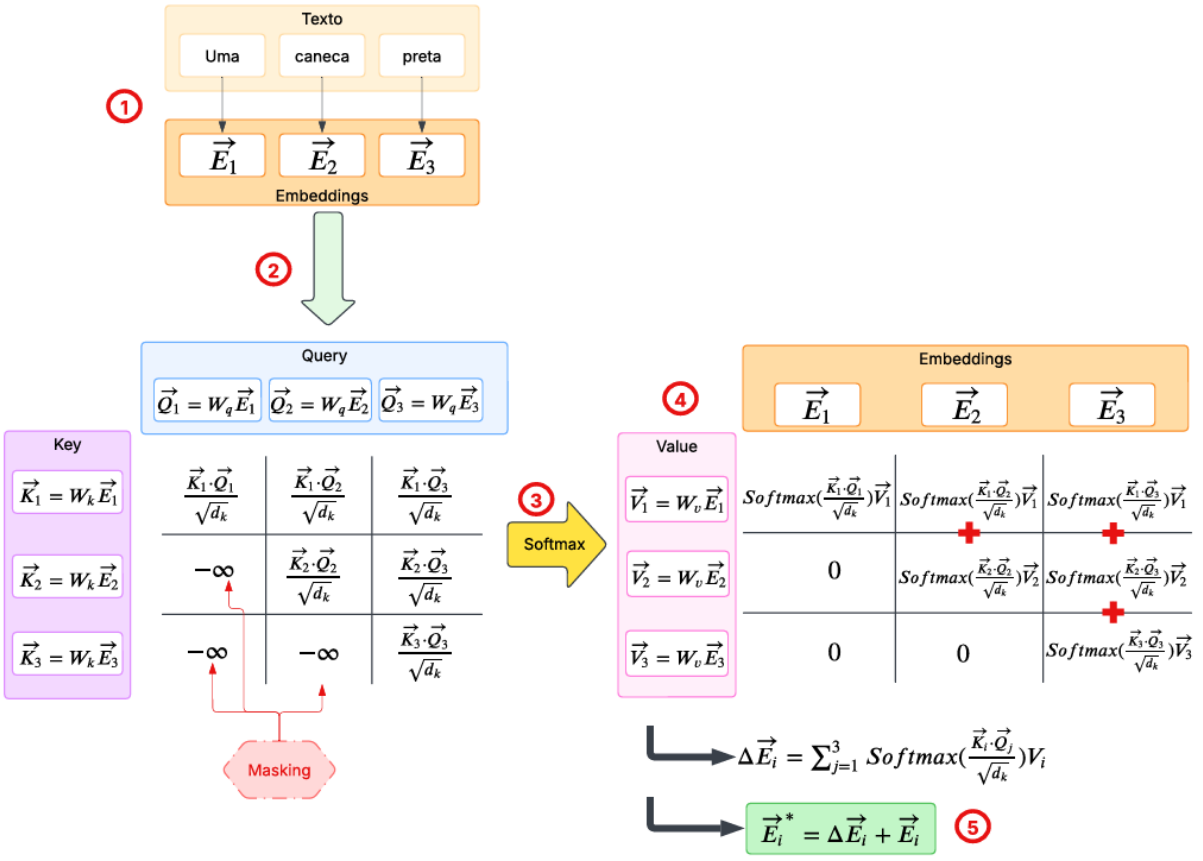


Figura 2: Representação do mecanismo de atenção single head com janela de contexto de três tokens.

embeddings. Em 2, são calculados os vetores de Key e Query, e o produto escalar entre eles é normalizado por $\sqrt{d_k}$. Nesse processo, palavras futuras são mascaradas com o valor $-\infty$ para evitar que influenciem o cálculo, já que, após a aplicação da softmax, esses valores se tornarão zero. Em 3, a função softmax é aplicada nos resultados obtidos em 2.. Em 4, os vetores de Value são calculados e multiplicados pelo resultado da etapa anterior. Por fim, em 5, todos os valores são somados e utilizados como modificadores dos embeddings originais.

2.1.2 Transformers

A arquitetura de Transformers, como pode-se observar na Figura 3, é baseada em mecanismos de atenção e é composta por múltiplas camadas de encoder e decoder. O encoder transforma a sequência de entrada em uma representação vetorial, preservando relações contextuais entre os tokens. Para isso, cada camada do encoder contém dois componentes principais: um mecanismo de *multi-head self-attention* e uma rede neural

feed-forward, que aprimora a representação aprendida.

O decoder tem o papel de gerar sequências de saída, como texto ou código. Ele possui uma estrutura semelhante ao encoder, mas com a adição de mecanismos que garantem a causalidade na geração sequencial, impedindo que a previsão de um token futuro dependa de tokens ainda não gerados. Além disso, cada camada do decoder inclui um classificador linear seguido de uma função softmax, permitindo a modelagem da distribuição de probabilidade sobre o vocabulário e a escolha da próxima palavra ou token.

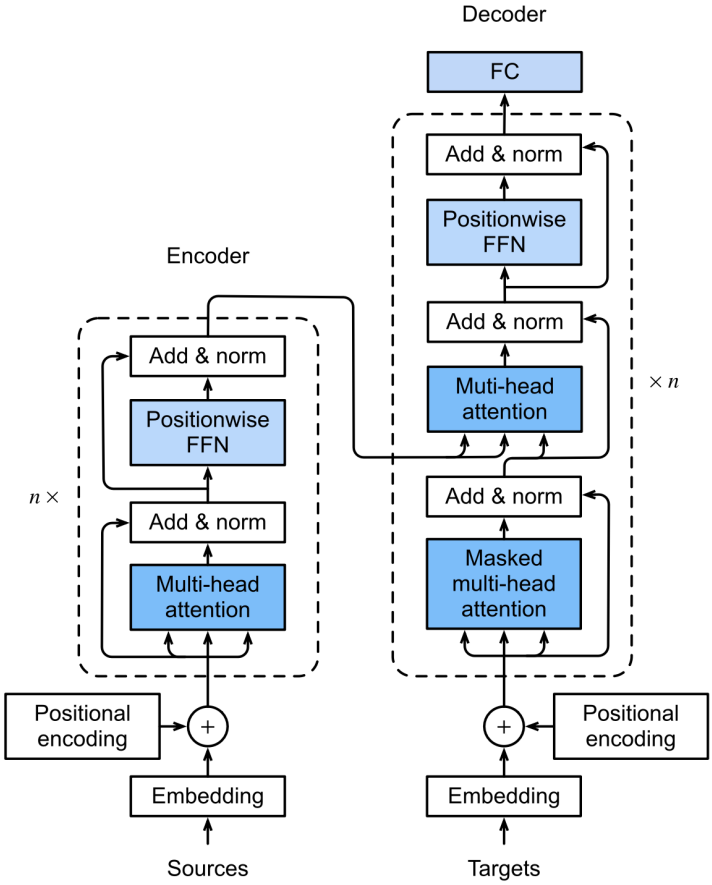
Os modelos de linguagem em grande escala, conhecidos como Large Language Models (LLMs), são construídos sobre a arquitetura Transformer e atingiram o estado da arte em diversas tarefas de PLN. Modelos como GPT-4 [10] e LLaMA 3 [9] exigem vastos recursos computacionais para treinamento e inferência, demandando milhares de GPUs e investimentos na casa de milhões de dólares. Mesmo modelos *open-source*, como os da família LLaMA, frequentemente requerem hardware especializado, tornando inviável a execução em dispositivos convencionais. Essa barreira técnica e financeira tem levado à concentração do desenvolvimento de LLMs em grandes corporações, levantando preocupações sobre acessibilidade e monopólio no avanço da inteligência artificial.

2.1.3 DeepSeek

O DeepSeek-R1 [7], introduzido em 2025, apresenta uma série de modelos que incorporam a técnica de *chain-of-thought* (CoT), na qual o modelo explicita sua linha de raciocínio ao formular uma resposta. Dentre esses modelos, há o DeepSeek-R1, o DeepSeek-R1-Zero e seis versões destiladas do DeepSeek-R1 com 1.5B, 7B, 8B, 14B, 32B e 70B parâmetros, baseadas em arquiteturas *open-source*, como Qwen [11] e LLaMa [9]. Na figura Figura 4, pode-se observar todos os modelos DeepSeek-R1 e as etapas para treiná-los.

O DeepSeek-R1 tem como base o DeepSeek V3 [12], um modelo de *mixture of experts* (MoE). Essa abordagem difere das arquiteturas monolíticas tradicionais ao treinar múltiplos modelos especializados em diferentes domínios ou sintaxes. O DeepSeek V3 possui um total de 671 bilhões de parâmetros, mas apenas 37 bilhões são ativados por entrada, tornando-o significativamente mais eficiente em termos computacionais. Comparativamente, o LLaMa 3.1, com 405 bilhões de parâmetros, foi treinado em 30,84 milhões de horas de GPU, enquanto o DeepSeek V3, apesar de ter 671 bilhões de parâmetros,

Figura 3: Arquitetura do tranformers



Fonte: Dive into deep learning [5], 2025

exigiu apenas 2,788 milhões de horas de GPU—aproximadamente 11 vezes menos que o LLaMa.

A partir do DeepSeek V3, foi desenvolvido o DeepSeek-R1. No artigo "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning" [7], os autores propuseram uma abordagem inovadora para aprimorar a capacidade de raciocínio (chain of thought) de modelos de linguagem utilizando aprendizado por reforço (*Reinforcement Learning*, RL). Até então, os modelos de *chain-of-thought* de alto desempenho eram restritos a corporações privadas. O DeepSeek-R1-Zero foi a primeira iteração resultante desse processo, apresentando resultados promissores, mas ainda demonstrando inconsistências na estruturação do raciocínio e, ocasionalmente, introduzindo termos em idiomas mistos ou gerando respostas mal formatadas.

Dado o desempenho inicial encorajador do DeepSeek-R1-Zero, os pesquisadores aplicaram *fine-tuning* supervisionado utilizando um conjunto de dados de alta qualidade

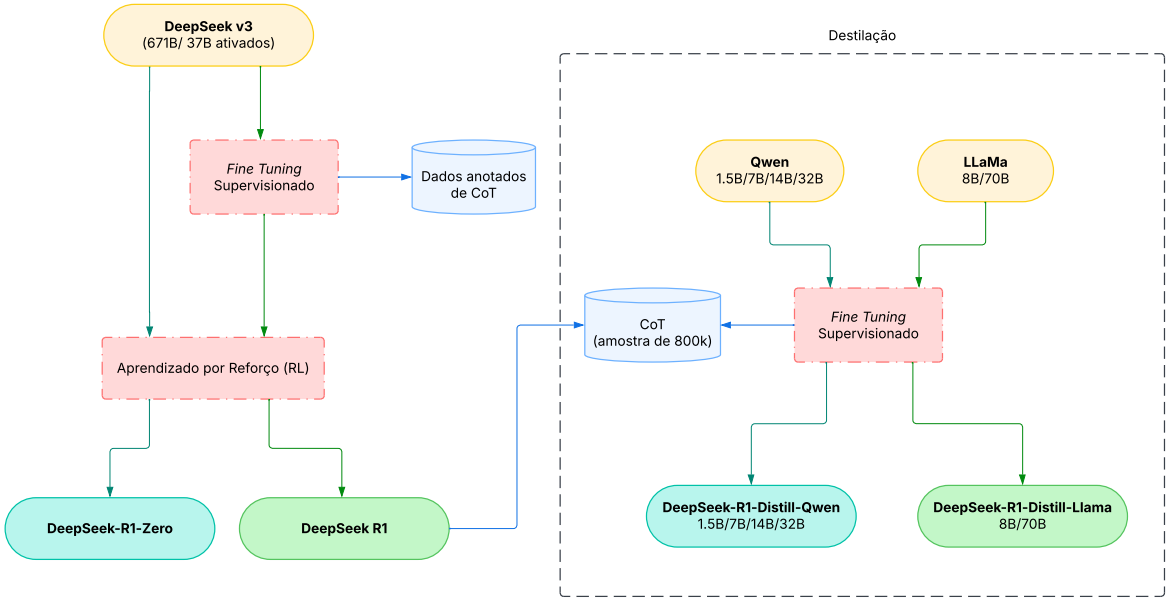


Figura 4: Representação de todos os modelos DeepSeek-R1 e as etapas para seu treinamento

contendo exemplos anotados de *chain-of-thought*. Em seguida, o treinamento foi continuado com aprendizado por reforço, resultando no DeepSeek-R1. Esse processo permitiu um refinamento significativo na capacidade do modelo de gerar raciocínios coerentes e bem estruturados.

É importante destacar que, para realizar inferência com o DeepSeek-R1, é necessário alocar toda a arquitetura do DeepSeek V3 na memória. Embora apenas 37 bilhões de parâmetros sejam ativados por entrada, diferentes entradas podem ativar subconjuntos distintos dos parâmetros, impossibilitando a execução local do modelo completo de 671 bilhões de parâmetros.

Para viabilizar o uso do DeepSeek-R1 em ambientes com recursos limitados, os pesquisadores aplicaram um processo de destilação do modelo, criando versões menores compatíveis com execução local. O processo de destilação envolve a utilização de um modelo professor estado da arte, nesse caso o DeepSeek-R1, que passa o seu conhecimento para os modelos alunos. Essa técnica permite que os modelos reduzidos preservem parte do desempenho do modelo original, tornando-os mais acessíveis e eficientes.

Por fim, conforme destacado nas conclusões do artigo do DeepSeek-R1, os autores observaram que o modelo não apresenta desempenho ideal em *few-shot prompting*. Portanto, neste estudo, seguiremos as recomendações dos pesquisadores e adotaremos

zero-shot prompting como abordagem principal.

2.2 Code smells

Maus Cheiros, do inglês, *Code Smells*, é um conceito criado por Kent Beck e popularizado por Martin Folwer em 1999, no livro Refatoração, Aperfeiçoando o Design de Códigos Existentes [2]. Code Smells são indicativos de que o código está difícil de manter e compreender. Segundo Fowler, é por meio de Code Smells que os desenvolvedores sabem quando devem começar e terminar a refatoração.

2.2.1 Code Smells Relacionados à Extract Method

O Extract Method é capaz de corrigir *Code Smells* diferentes. Segue na lista abaixo os principais *Code Smells* corrigidos pelo *Extract Method* [13] e como isso é feito:

- Duplicate Code (Código Duplicado): Trechos de código idênticos ou muito semelhantes que aparecem em mais de um lugar. O Extract Method permite que essses trechos sejam substituídos por uma chamada de função.
- Long Method (Função (Método) Longa): Métodos muito extensos, que realizam muitas operações com mais de uma responsabilidade. Com o Extract Method trechos de código com uma responsabilidade única podem ser extraídos, tornando o método principal mais legível e claro.
- Feature Envy (Inveja de Recursos): Um método depende excessivamente de dados ou métodos de outra classe. Se apenas parte do método possui inveja, este deve ser extraído e transferido para a classe com os dados.
- Switch Statement: Um código utiliza uma estrutura de decisão switch com muitos ramos, geralmente envolvendo a mesma lógica de processamento em diferentes condições. A refatoração Extract Method corrige esse problema extraindo o switch para uma função própria.
- Message Chains (Cadeias de Mensagens): Um método faz várias chamadas em cadeia, como por exemplo obj.getA().getB().getC(). A lógica associada à cadeia pode ser extraída para um método com um nome descritivo.

- Comments (Comentários): Um código que contém muitos comentários explicativos, geralmente é porque o código é complexo ou confuso, e os comentários são usados para explicar o que o código faz em vez de quais regras de negócio eram vigentes em sua criação. Caso haja um comentário explicando o que um trecho de um código está fazendo, usar o *Extract Method* com um nome de função significativo pode corrigir esse problema.

- Data Class (Classe de Dados): Classes que apenas têm atributos e métodos de leitura e escrita. Esse *mau cheiro* pode ser corrigido observando quais métodos chamam os métodos de leitura e escrita dessa classe de dados e extraindo a lógica para um método dentro da Classe de Dados.

2.2.2 Detecção de *Code Smells*

A survey *Code Smells Detection and Visualization: A Systematic Literature Review* [1] analisa as soluções de detecção de maus cheiros existentes, em 2022, e as separa em sete abordagens:

1. Baseado em busca: utiliza algoritmos de Inteligência Artificial, fazendo a detecção de forma automática, mas depende da qualidade dos dados.

2. Baseado em métricas: cria uma regra baseada em métricas de detecção e define limiares para cada code smells, mas não há um consenso dos valores dos limiares já que a existência de um Code Smell pode ser arbitrária.

3. Baseado em sintomas: Procura detectar um code smell pelo seu sintoma, a survey classifica essa abordagem como lenta e não acurata.

4. Baseado em visualização: Transfere métricas para uma representação visual para ajudar desenvolvedores a detectar code smells, o que faz com que essa abordagem precise de muito esforço humano e seja suscetível a erros.

5. Probabilístico: Determina a probabilidade de ser um code smell.

6. Baseado em cooperação: Utiliza a cooperação de duas ou mais abordagens diferentes.

7. Manual: Realiza o processo de detecção manualmente.

Code Smell	Número de Estudos	% Estudos
Inveja de Recursos	28	33.7
Função (Método) Longa	22	26.5
Classe de Dados	18	21.7
Código Duplicado	9	10.8
Switch Statement	4	4.8

Tabela 2: Code Smells mais populares relacionados ao extract method segundo [1]

A survey também apresenta os dez *code smells* mais estudados em cada artigo, dentre esses code smells, os que são relacionados ao Extract Method estão destacados na Tabela 2. Os Code smells Cadeias de Mensagem e Comentários não tiveram mais que três publicações para serem considerados.

O artigo "Multi-label Learning for Identifying Co-occurring Class Code Smells" [14] propõe uma abordagem de aprendizado de máquina para detectar code smells que ocorrem simultaneamente em classes. O objetivo desse artigo foi desenvolver um modelo baseado em aprendizado de máquina multi-rótulo, utilizando métricas de código como features. Nesse artigo, o único code smell estudado relacionado Extract Method foi o Cadeias de Mensagens.

O artigo "Improving Accuracy of Code Smells Detection Using Machine Learning with Data Balancing Techniques" [15] explora como técnicas de aprendizado de máquina combinadas com estratégias de balanceamento de dados podem melhorar a precisão na detecção de code smells. O trabalho aborda o problema do desequilíbrio em conjuntos de dados, onde a baixa ocorrência de code smells dificulta o treinamento de modelos de aprendizado. Para mitigar isso, técnicas como oversampling (aumentar exemplos da classe minoritária) e undersampling (reduzir exemplos da classe majoritária) foram aplicadas antes do treinamento dos algoritmos LSTM e GRU. Na avaliação dos resultados, os autores realizaram um t-test pareado e a acurácia média foi dois pontos percentuais mais alta para ambos os modelos treinados com o dataset balanceado. Dentre os code smells relacionados a extract method, eles analisam Classe de Dados, Inveja de Recursos e Método longo.

2.3 **Refatoração**

A refatoração é o processo de modificar o código-fonte de um software sem alterar seu comportamento externo, com o objetivo de melhorar seu design, estrutura e legibilidade. O conceito de refatoração já era discutido antes de 1990, mas foi popularizado e

consolidado com a publicação do livro "Refatoração: Aperfeiçoando o Design de Códigos Existentes" [2], de Martin Fowler. Nas seções a seguir, será discutido o que é a refatoração *Extract Method* (Extrair Método), que será abordada nesse estudo, e o que temos na literatura atual para automatização da refatoração do *Extract Method*.

2.3.1 Extrair Função

A refatoração Extrair Função (*Extract function/method*) se inicia assim que nota-se que existe um trecho numa função (ou método, caso a linguagem seja orientada a objetos) que possui significado próprio. Esse trecho deve ser extraído e direcionado para uma função que tenha um nome que descreva qual deve ser o comportamento do trecho.

No livro "Refatoração: Aperfeiçoando o Design de Códigos Existentes" [2], Fowler propõe que o procedimento do Extrair Função siga o seguinte passo a passo:

1. Criar uma função nova com um nome de acordo com o seu propósito
2. Copiar o código a ser extraído para a função criada
3. Verificar no código extraído referências a variáveis locais da função original. Se existirem, passar essas variáveis como parâmetros da nova função
4. Compilar o código
5. Substituir o código extraído por uma chamada da função criada
6. Testar
7. Procurar códigos iguais ou parecidos ao que acabou de ser extraído e aplicar a refatoração "Substituir código internalizado por chamada de função"neles

2.3.2 Extrair método e LLMs

O artigo "Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method" [16] propõe a utilização de few-shot prompt engineering para realizar a refatoração de um código. A técnica aplicada nesse artigo segue o seguinte passo a passo:

1. Faz a chamada para a LLM para gerar sugestões de Extrair Método para um determinado método

2. Remove alucinações e mantém apenas as refatorações que são livres de erro

3. Remove as refatorações que não são úteis

O primeiro passo inicia com o *prompt engineering*, onde os autores aplicam *in-context learning*, que consiste em prover na entrada do modelo todas as instruções necessárias para a realização de uma tarefa. Para enriquecer ainda mais o prompt, eles aplicam *few-shot leaning*, adicionando dois exemplos no prompt. Depois, os autores fazem um array de sugestões, executando um mesmo prompt várias vezes para dada entrada pois resultados de LLMs são não-determinísticos.

Seguindo para o segundo passo, os autores removem sugestões de *Extract Method* inválidas, removendo sugestões que levariam a um código não compilável. Em seguida, eles removem as refatorações que não são úteis.

No artigo "Refactoring Programs Using Large Language Models with Few-Shot Examples" [17], os autores investigam o uso do modelo GPT-3.5 para refatorações em programas Python, com o objetivo de reduzir a complexidade do código e aprimorar sua legibilidade. A abordagem proposta baseia-se em few-shot prompting. Para cada programa analisado, o modelo gera 10 versões refatoradas e são mantidas apenas aquelas que são funcionalmente corretas.

Os experimentos demonstraram que 95,68% dos programas puderam ser refatorados com sucesso, resultando em uma redução média de 17,35% na complexidade ciclomática e 25,84% na quantidade de linhas de código. No entanto, os autores destacam que, em alguns casos, a abordagem pode gerar modificações desnecessárias em códigos já bem estruturados.

3 SOLUÇÃO PROPOSTA E METODOLOGIA

Neste capítulo, serão apresentadas a solução proposta e a metodologia dos experimentos conduzidos durante a execução deste projeto, detalhando suas etapas para reprodução.

3.1 Solução Proposta

A solução proposta neste Trabalho de Conclusão de Curso (TCC) consiste em analisar a refatoração de códigos Java *open source* utilizando modelos de linguagem com um *prompt* pré-definido para guiar a execução do Extract Method. O estudo será conduzido por meio de experimentos utilizando modelos destilados do DeepSeek-R1, mais especificamente as variantes DeepSeek-R1 Qwen 1.5B, DeepSeek-R1 Qwen 7B e DeepSeek-R1 LLaMa 8B. O principal motivo da escolha desses modelos é sua otimização em relação aos modelos maiores, como o DeepSeek R1, que tem 671B de parâmetros e não pode ser executado sem o auxílio de um servidor de placas de vídeo.

A metodologia adotada visa não apenas avaliar a capacidade dos modelos de identificar e sugerir refatorações adequadas, mas também compreender o raciocínio subjacente a essas sugestões, explorando o chain-of-thought retornado nas respostas dos modelos. Além disso, a análise incluirá métricas quantitativas, como o ratio de Levenshtein entre os nomes das funções extraídas e o nome das funções do baseline, além da diferença entre os trechos de código selecionados pelo baseline e pelos modelos. Também serão observados padrões de erros comuns, como a repetição excessiva de tokens ou a sugestão de refatorações desnecessárias. Esses fatores permitirão avaliar o desempenho dos modelos e identificar oportunidades para aprimoramento na aplicação de LLMs para refatoração automática de código.

3.2 Metodologia

De modo geral, o processo experimental envolveu a refatoração de códigos Java open source, utilizando um prompt pré-definido. Os experimentos foram conduzidos com modelos destilados do DeepSeek-R1, DeepSeek-R1 Qwen 1.5B, DeepSeek-R1 Qwen 7B e DeepSeek-R1 LLaMa 8B. Por fim, os resultados foram analisados com foco no raciocínio

seguido pelo modelo (chain-of-thought) e na qualidade das refatorações sugeridas.

3.2.1 Dataset

Neste estudo, foram utilizados três projetos open-source nos quais a refatoração Extract Method foi previamente mapeada: JHotDraw 5.2, MyWebMarket e wikidev-filters. Esses projetos fazem parte do conjunto de dados utilizado no artigo "Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method" [16]. A versão refinada desse dataset contém 106 sugestões de refatoração Extract Method, distribuídas em 50 classes diferentes, extraídas de um total de 224 arquivos Java.

Para a execução dos experimentos, os arquivos Java foram processados por projeto. Durante a leitura do arquivo, cada linha de código foi enumerada, permitindo que o modelo identificasse o trecho a ser extraído.

3.2.2 Prompt

Como o DeepSeek R1 não apresenta um desempenho satisfatório em abordagens *few-shot* e *one-shot* [7], o *prompt* do artigo "Next-Generation Refactoring: Combining LLM Insights and IDE Capabilities for Extract Method" [16] foi ajustado para um formato *zero-shot*. Dessa forma, as inferências foram realizadas utilizando o seguinte *prompt*:

```
You are a skilled software developer. You have immense knowledge on
    software refactoring.
You communicate with a remote server that sends you code of functions
    (one function in a message) that it wants to simplify by applying
    extract method refactoring.
In return, you send a JSON object with suggestions of helpful extract
    method refactorings. It is important for suggestions to not
    contain the entire function body.
Each suggestion consists of the start line, end line, and name for
    the extracted function.
The JSON should have the following format: [{"function_name": <new
    function name>, "line_start": <line start>, "line_end": <line end
    >}, ..., ].),
Your answer should only consist of this json.
```

CPU	MEM	GPU
32	64	2

Tabela 3: Limites de CPU, MEM e GPU da partição short do apuana

The code is listed bellow:

3.2.3 Apuana

Os experimentos foram executados no *cluster* Apuana [18], disponibilizado pelo Centro de Informática da UFPE. Cada execução teve uma duração aproximada de dois dias, utilizando a partição *short*, cujos limites de CPU, memória e GPU estão detalhados na Tabela 3.

Para executar os *jobs*, foi necessário criar um *script* em Python, contendo o modelo a ser utilizado, a lógica de leitura dos arquivos e o *prompt*, além de um *script* em Bash, com os comandos para alocação de recursos e a referência ao *script* em Python. Após a preparação, o trabalho foi submetido utilizando o SLURM [19] por meio do comando *sbatch*.

3.2.4 Métricas

Nas subseções a seguir, serão discutidas as métricas utilizadas para analisar a performance dos resultados obtidos pelos modelos.

3.2.4.1 Levenshtein Ratio

A distância de Levenshtein quantifica a diferença entre duas strings, calculando o número mínimo de operações necessárias para transformar uma string na outra. As operações permitidas incluem inserção, remoção e substituição de caracteres. Para normalizar a distância em relação ao número de caracteres do nome da função, foi utilizada a função *ratio* da biblioteca Levenshtein [20], que aplica a Equação 3.1.

$$ratio(s_1, s_2) = 1 - (\frac{distance}{(len(s_1) + len(s_2))}) \tag{3.1}$$

Nesse contexto, o nome da função gerado pelo Extract Method estará mais próximo do nome da função do *baseline* quanto mais próximo o valor do *ratio* estiver de 1. Assim,

a análise foi realizada maximizando o valor do *ratio* de Levenshtein para verificar se o modelo sugeriu o nome da função do método extraído adequadamente.

3.2.4.2 Estatísticas

Para medir a precisão das sugestões do modelo, foi feita uma análise estatística da diferença entre os trechos extraídos do modelo versus do baseline. Essa análise foi feita por meio dos testes não-paramétricos de hipótese de Mann-Whitney e de Wilcoxon, já que os dados não seguiam uma distribuição normal.

3.2.5 Pre-processamento dos Dados

A resposta do DeepSeek foi retornada ao job do Apuana em formato de texto, utilizando o token </think> para separar o CoT da refatoração gerada pelo modelo. No processamento das respostas para análise, a coluna do CoT foi preenchida com todo o texto gerado pelo modelo até o token </think>. Para a análise do CoT, todas as palavras foram convertidas para minúsculas, e as *stopwords* foram removidas. Os resultados do Extract Method foram adicionados na coluna dos métodos extraídos apenas quando apresentados no formato especificado pelo prompt, independentemente se eram na posição inicial da resposta ou não.

3.2.5.1 Refatorações válidas

Para considerar uma refatoração válida, essa deveria poder ser mapeada à uma refatoração do baseline. Porém, tanto o baseline quanto os modelos poderiam sugerir mais de uma refatoração. Assim, para mapear a refatoração do modelo à refatoração do baseline corretamente, foi considerado uma refatoração o conjunto *(nome da função, linha de início da extração, linha do final da extração)* e a seleção da refatoração pareada ao baseline foi feita seguindo as seguintes regras:

1. Uma classe não pode ter uma refatoração sugerida ligada a duas refatorações diferentes do baseline

2. Uma classe não pode ter uma refatoração do baseline ligada a duas refatorações sugeridas diferentes

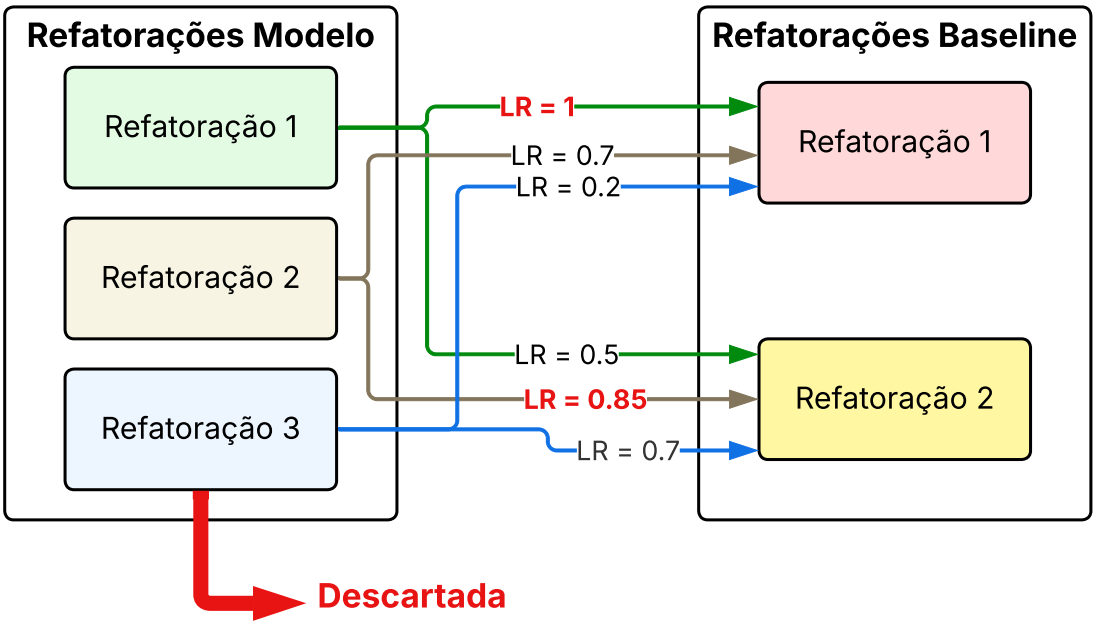


Figura 5: Mapeamento das refatorações sugeridas ao baseline

Ou seja, todas as refatorações têm que ser únicas ou nulas. A seleção da refatoração a ser mantida na deduplicação foi feita de modo a maximizar o *Levenshtein ratio*. Na Figura 5, pode-se observar como o algoritmo de mapeamento funciona para o caso em que o modelo sugere três refatorações e o baseline duas. Nesse exemplo, a Refatoração 1 do modelo é mapeada para a Refatoração 1 do baseline, pois o *Levenshtein ratio* dela é maior em comparação às outras duas. Pelo mesmo motivo, a Refatoração 2 do modelo é mapeada para a Refatoração 2 do baseline. Como não há outras refatorações para realizar o mapeamento, a Refatoração 3 é descartada.

Na Tabela 4, observa-se que o modelo com o maior número de Extract Methods válidos foi o LLaMa 8B, embora também tenha sido o modelo com o maior número de refatorações descartadas.

Na Tabela 5, esse valor foi convertido em precision e recall. Nesse sentido, o LLaMa 8B encontrou o maior número de Extract Method que deveriam acontecer (recall 77,45%), o Qwen 7B obteve a pior precisão (6,96%) e o Qwen 1.5B obteve a melhor precisão (11,41%), mas o valor foi próximo à precisão LLaMa 8B (10.01%).

	Baseline	**Qwen 1.5B**	**Qwen 7B**	**LLaMa 8B**
Extract Method Sugeridos	102	184	273	789
Extract Method Descartados	0	163	254	710
Extract Method	102	21	19	79

Tabela 4: Número de refatorações

	Qwen 1.5B	**Qwen 7B**	**LLaMa 8B**
Precision	11,41%	6,96%	10,01%
Recall	20,59%	18,83%	77,45%

Tabela 5: Precision/Recall por modelo

4 ANÁLISE DE RESULTADOS

Neste capítulo, serão avaliados os resultados do *Chain of Thought* (CoT) nos testes, bem como as refatorações sugeridas pelos modelos. Além disso, analisamos a proximidade dessas refatorações ao dataset *source of truth*.

4.1 Extrair Método

Nessa seção, será feita uma análise dos resultados em relação à sugestão de *Extract Method* do modelo. Serão analisados o nome da função sugerida e o trecho de código escolhido para ser refatorado.

Ao analisar a Figura 6, observa-se que o número de refatorações por classe do baseline está mais próximo de zero ou um, em comparação com os modelos, com exceção de alguns outliers. Isso indica que os modelos sugeriram refatorações desnecessárias. Os valores exatos dessa diferença podem ser consultados na Tabela 6. Nesse contexto, o Qwen 1.5B apresentou resultados mais próximos aos do baseline.

Além disso, na Figura 7, observa-se que o modelo que mais se aproximou do baseline em termos de número de linhas extraídas durante o processo de realização do Extrair Método foi o Qwen 1.5B. Vale destacar que esse modelo apresentou o maior outlier, de aproximadamente 100. Também foi investigado se algum modelo gerou um Extrair Método com o final antes do começo, mas nenhum modelo apresentou esse tipo de anomalia.

4.1.1 Análise estatística

Como apresentado na Figura 8, a mediana dos valores do *Levenshtein ratio* foi ligeiramente maior para o Qwen 1.5B, embora o valor tenha sido muito semelhante ao dos outros modelos.

	Baseline	**Qwen 1.5B**	**Qwen 7B**	**LLaMa 8B**
Média	0.455357	0.821429	1.218750	3.522321
Desvio Padrão	1.115632	1.225399	2.484284	4.949017
Mínimo	0.000000	0.000000	0.000000	0.000000
Máximo	6.000000	8.000000	14.000000	52.000000

Tabela 6: Métricas do número de refatorações por classe

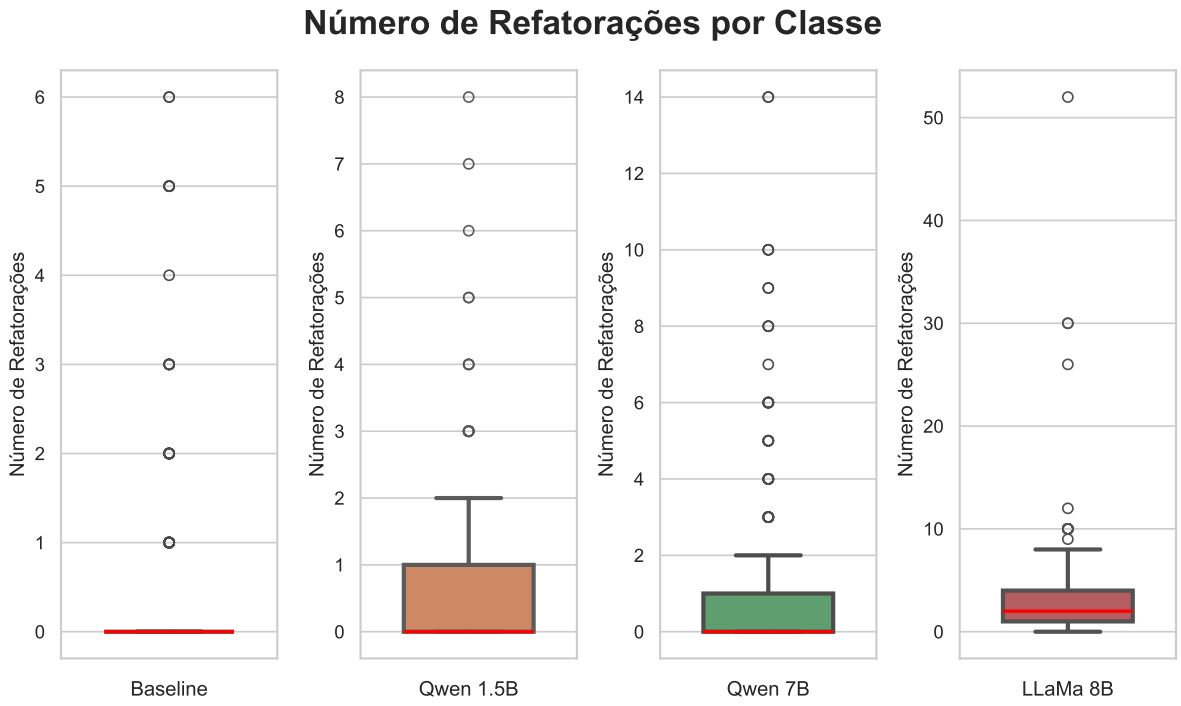


Figura 6: Número de refatorações por classe

A Figura 9 apresenta um boxplot que mostra a diferença entre os valores propostos pelos modelos e pelo *baseline* do line_start e line_end. Como pode ser observado, essa diferença é menor para o modelo Qwen 7B, o que sugere que ele seja mais preciso na definição do local onde a refatoração deve ocorrer. Vale destacar que, ao utilizar a estratégia de maximizar o *ratio* de Levenshtein, não houve nenhum caso em que o modelo tenha retornado uma extração totalmente precisa, em que ambas as diferenças entre o *baseline* e a sugestão fossem zero.

Para verificar se as observações das imagens eram corretas, foram executados dois testes de hipótese. O primeiro, para identificar se a distribuição do *Levenshtein ratio* do Qwen 1.5B é maior que os demais. O segundo, para verificar se existia algum modelo cuja a diferença para o baseline do line_start ou do line_end se aproximava de zero.

Primeiramente, foi analisada a distribuição dos dados referentes ao *Levenshtein ratio*, aplicando o teste de D'Agostino e Pearson para verificar se essas distribuições seguiam um comportamento normal quando o número de amostras é maior que 50 e aplicando Shapiro-Wilk caso contrário. Como mostrado na Tabela 8, apenas Qwen 7B pode ser considerado uma normal, logo, os testes de hipótese executados foram não-paramétricos.

Para o teste referente ao *Levenshtein ratio*, foi utilizado o algoritmo de Mann-Whitney e considerado que o p-valor menor que 5% rejeita a hipótese nula, e suas hipóteses

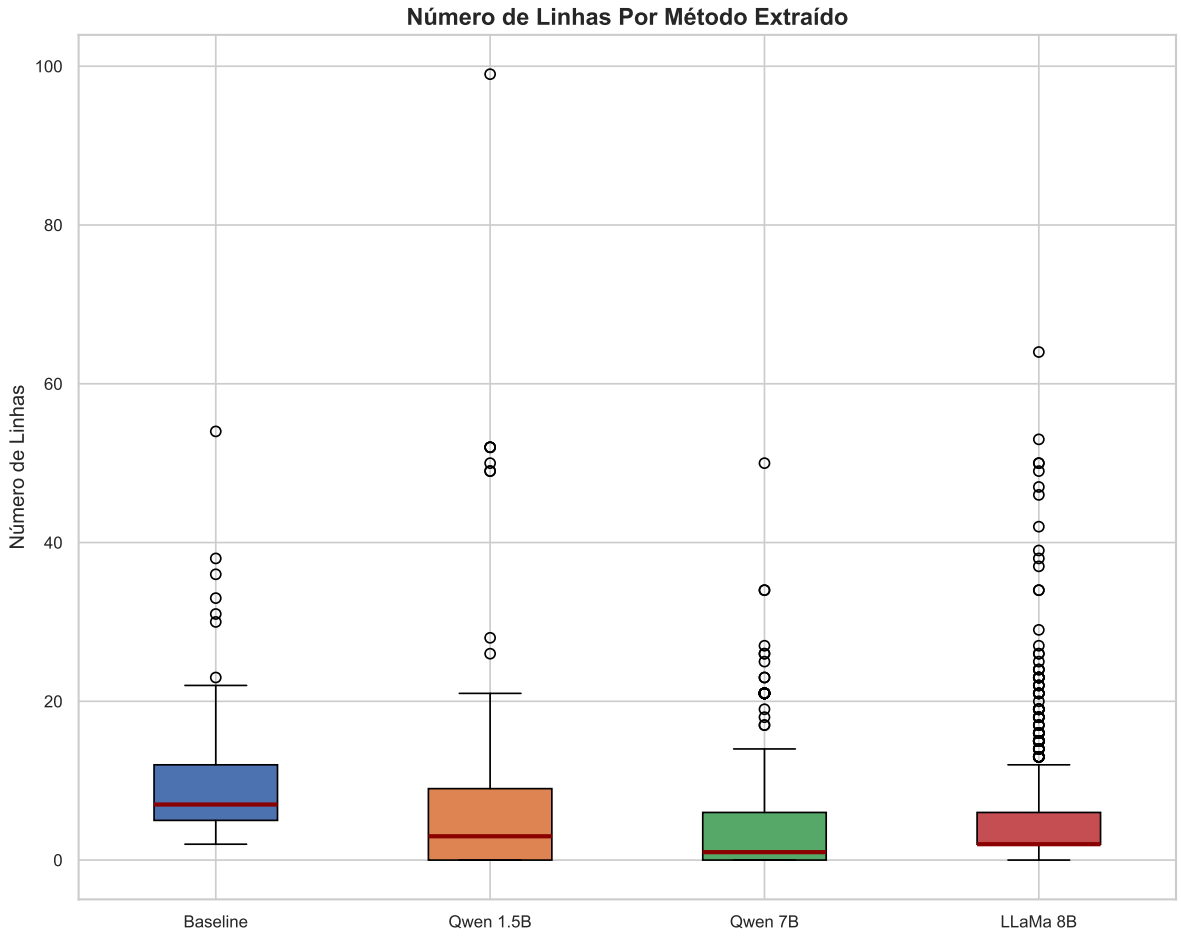


Figura 7: Número de linhas extraídas

Alternativa	**P-valor**	**Conclusão**
Qwen 1.5B >Qwen 7B	$0,02$	Qwen 1.5B **tem** valores maiores que Qwen 7B
LLaMa 8B >Qwen 7B	$0,02$	LLaMa 8B **tem** valores maiores que Qwen 7B

Tabela 7: Testes de Mann-Whitney do *Levenshtein ratio* dos resultados

foram:

- H_0: A distribuição X possui valores menores ou iguais à distribuição Y
- H_1: A distribuição X possui valores maiores à distribuição Y

O teste foi feito com todos os pares de modelo possíveis, mas os únicos pares conclusivos foram o Qwen 1.5B com o Qwen 7B e o LLaMa 8B com o Qwen 7B. Na Tabela 7, observa-se que o teste do que o LLaMa 8B e o Qwen 1.5B têm valores de *Levenshtein ratio* maiores que o Qwen 7B, mas nada se pode concluir entre os dois.

Já para os testes referentes às diferenças entre as linhas da extração, foi utilizado o algoritmo de Wilcoxon, também considerando que o p-valor menor que 5% rejeita a

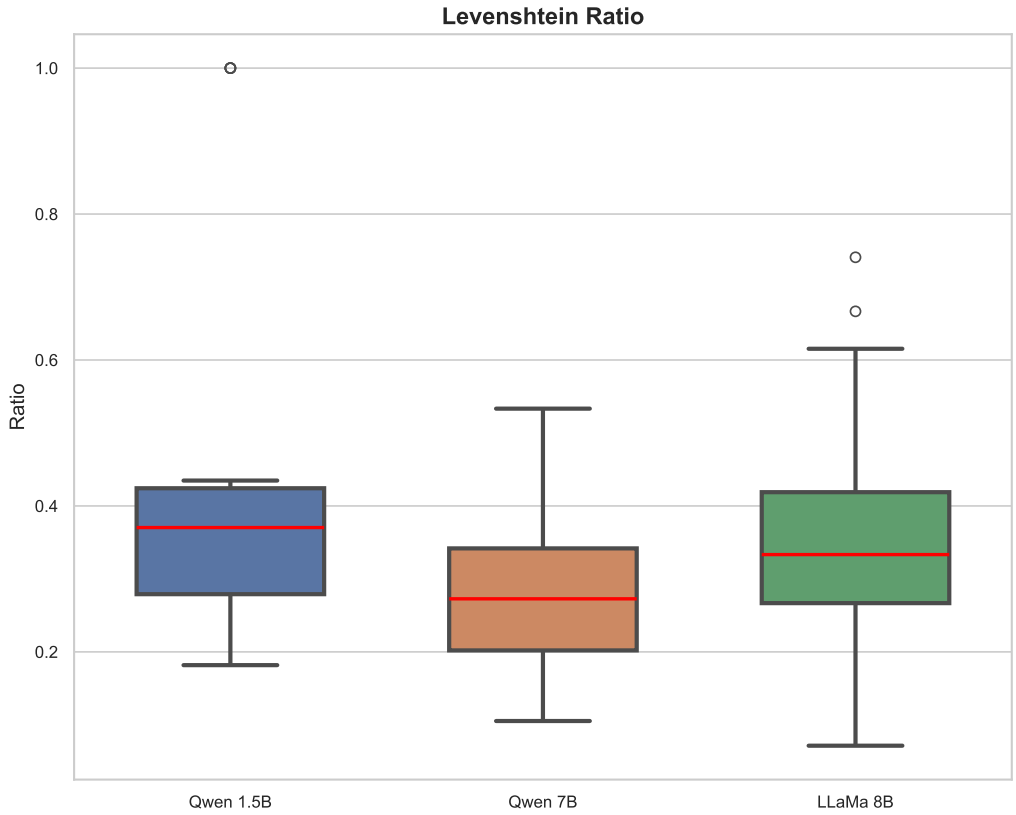


Figura 8: Levenshtein *ratio* dos nomes de função

hipótese nula, e suas hipóteses foram:

- H_0: A distribuição X não é próximos de zero.
- H_1: A distribuição X é próximas de zero.

A partir das Tabela 9 e Tabela 10, observa-se que nenhuma distribuição é suficientemente próxima de zero. Ou seja, nenhum modelo conseguiu inferir as refatorações com pouca diferença das linhas sugeridas pelo baseline.

4.2 **Chain of Thought (CoT)**

Com base na nuvem de palavras da Figura 10, observa-se que o contexto de refatoração foi preservado nas palavras mais comuns do CoT. Essa constatação é reforçada pela Tabela 11, que apresenta as dez palavras mais frequentes no CoT, no qual todas se mantiveram no contexto de código e refatoração.

Teste	P-valor	Conclusão
Levenshtein Ratio Qwen 1.5B	$5,13 \times 10^{-5}$	Não segue a distribuição normal
Levenshtein Ratio Qwen 7B	$0,46$	Pode seguir a distribuição normal
Levenshtein Ratio LLaMa 8B	$0,03$	Não segue a distribuição normal
Diferença da Linha Inicial Qwen 1.5B	$2,25 \times 10^{-5}$	Não segue a distribuição normal
Diferença da Linha Inicial Qwen 7B	$7,15 \times 10^{-6}$	Não segue a distribuição normal
Diferença da Linha Inicial LLaMa 8B	$5,09 \times 10^{-9}$	Não segue a distribuição normal
Diferença da Linha Final Qwen 1.5B	$2,29 \times 10^{-5}$	Não segue a distribuição normal
Diferença da Linha Final Qwen 7B	$7,06 \times 10^{-6}$	Não segue a distribuição normal
Diferença da Linha Final LLaMa 8B	$5,12 \times 10^{-9}$	Não segue a distribuição normal

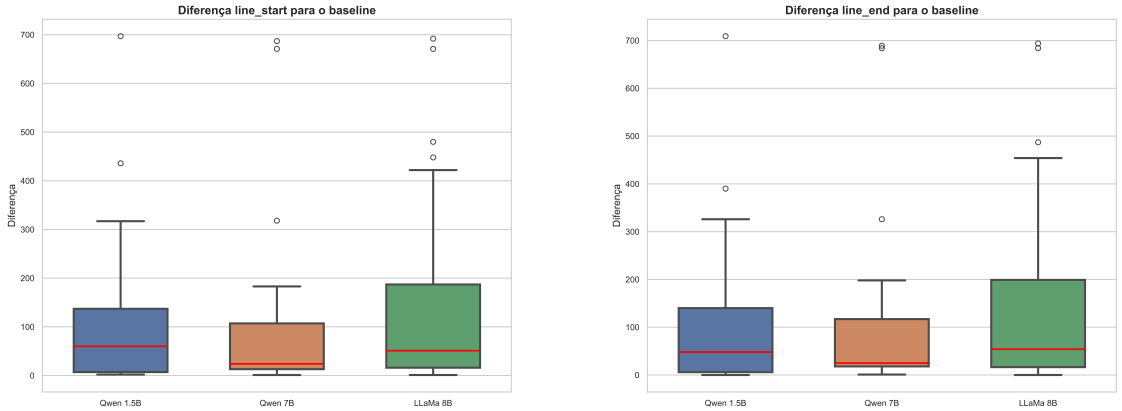
Tabela 8: Testes de D'Agostino-Pearson

Alternativa	P-valor	Conclusão
Qwen 1.5B $\neq 0$	$9,34 \times 10^{-7}$	Qwen 1.5B **não tem** valores próximos ao zero
Qwen 7B $\neq 0$	$3,81 \times 10^{-6}$	LLaMa 8B **não tem** valores próximos ao zero
LLaMa 8B $\neq 0$	$1,15 \times 10^{-14}$	Qwen 1.5B **não tem** valores próximos ao zero

Tabela 9: Testes de Wilcoxon para diferença do line_start entre o baseline e os experimentos

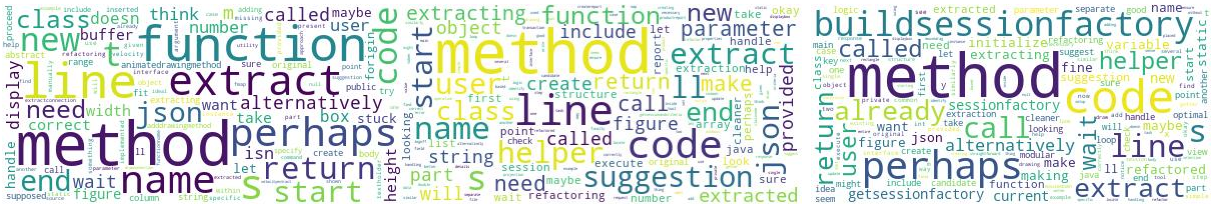
Alternativa	P-valor	Conclusão
Qwen 1.5B $\neq 0$	$8,86 \times 10^{-5}$	Qwen 1.5B **não tem** valores próximos ao zero
Qwen 7B $\neq 0$	$3,81 \times 10^{-6}$	LLaMa 8B **não tem** valores próximos ao zero
LLaMa 8B $\neq 0$	$1,67 \times 10^{-14}$	Qwen 1.5B **não tem** valores próximos ao zero

Tabela 10: Testes de Wilcoxon para diferença do line_end entre o baseline e os experimentos



(a) Início (b) Final

Figura 9: Diferença do início e do final da extração comparado ao baseline



(a) Qwen 1.5B (b) Qwen 7B (c) LLaMa 8B

Figura 10: Nuvem de Palavras do CoT

Qwen 1.5B	Qwen 7B	LLaMa 8B
method	method	method
function	code	code
line	line	perhaps
extract	methods	buildsessionfactory
perhaps	name	already
name	helper	extract
code	extract	call
new	json	methods
start	user	called
end	class	helper

Tabela 11: Dez palavras mais comuns do CoT

Em relação ao contexto de maus cheiros, apenas um CoT citou maus cheiros, o LLaMa 8B, o que significa que os modelos não levaram em consideração maus cheiros ao fazer a refatoração. O trecho do único CoT que menciona maus cheiros foi:

> Looking at lines 45–47, there's a try–catch block where it adds the delay. That's a bit of a code smell. Maybe we can simplify that by extracting the sleep into a helper method.

4.2.1 Tamanho e repetições por CoT

Embora o contexto de refatoração tenha sido mantido, alguns modelos geraram CoTs com tamanhos superiores à mediana. Isso pode ser observado na Figura 11, que apresenta uma comparação entre o número de tokens dos CoTs gerados pelos diferentes modelos.

O modelo Qwen 1.5B apresenta outliers com número de tokens superiores a 6000 com maior frequência do que os outros modelos, o que leva à hipótese que ele tende a se desorganizar durante a formulação do raciocínio, repetindo trechos de forma excessiva. Esse comportamento é corroborado pela Figura 12, que ilustra o número máximo de

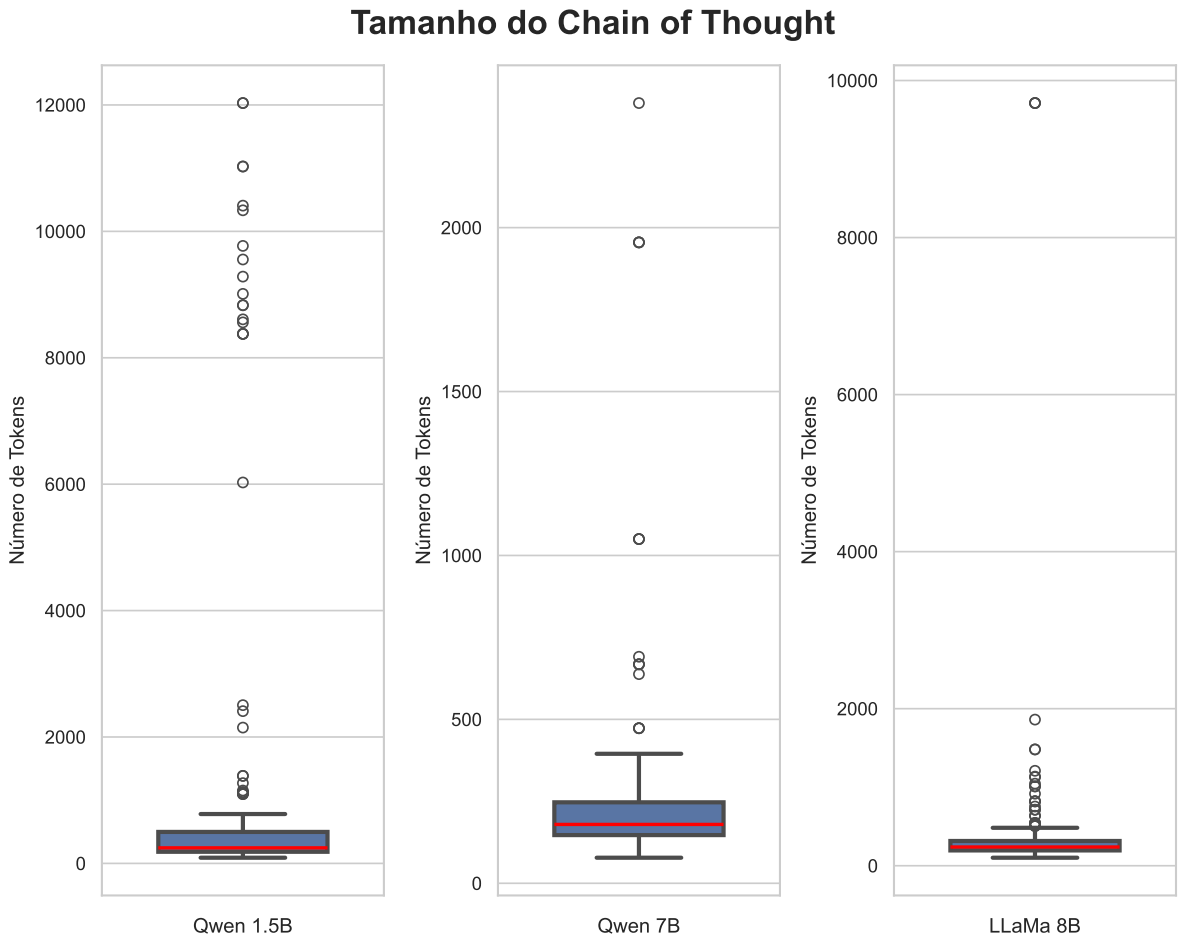


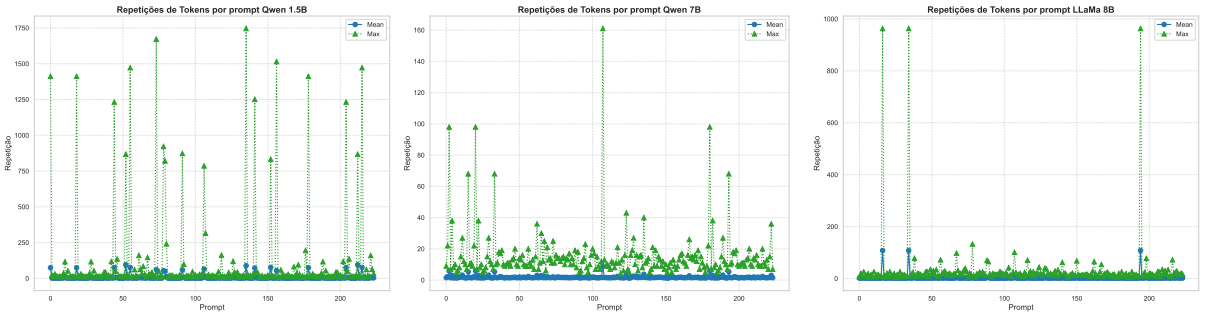
Figura 11: Número de palavras do CoT

tokens repetidos nos CoTs gerados pelo modelo. O Qwen 1.5B exibiu o maior número de repetições em comparação com os demais modelos, indicando que seu CoT tenha pior qualidade em comparação aos demais.

Para validar essa hipótese, foram realizados dois testes estatísticos: o primeiro para verificar se o Qwen 1.5B possui um CoT maior em comparação aos demais modelos, e o segundo para avaliar se ele apresenta um maior número de palavras repetidas por CoT.

Inicialmente, foi analisada a distribuição dos dados referentes ao tamanho do CoT e ao número de repetições (Figura 13), aplicando o teste de D'Agostino e Pearson para verificar se essas distribuições seguiam um comportamento normal. Como mostrado na Tabela 13, nenhuma das distribuições foi considerada próxima a uma distribuição normal, logo, os testes de hipótese executados foram não-paramétricos.

Ambos os testes consideravam que o p-valor menor que 5% rejeita a hipótese nula, e suas hipóteses foram:



(a) Qwen 1.5B (b) Qwen 7B (c) LLaMa 8B

Figura 12: Repetições de tokens no CoT

Qwen 1.5B		**Qwen 7B**		**LLaMa 8B**	
Palavras	**Repetições**	**Palavras**	**Repetições**	**Palavras**	**Repetições**
function	1746	token	161	code	963
method	1672	string	113	buildsessionfactory	913
extract	1662	method	98	perhaps	846
code	1515	parameters	92	method	667
name	1450	helper	85	already	533
line	1412	return	74	getsessionfactory	370
forigin	1251	user	68	sessionfactory	369
animatedrawingmethod	1229	line	68	call	368
display	1003	msg	62	return	364
box	1001	ioexception	61	wait	324

Tabela 12: Palavras mais repetidas no CoT e suas repetições máximas

- H_0: A distribuição X possui valores menores ou iguais à distribuição Y
- H_1: A distribuição X possui valores maiores à distribuição Y

Na Tabela 14, pode-se observar que o Qwen 1.5B e o LLaMa 8B têm tamanhos de CoT maiores que o Qwen 7B, mas não se pode confirmar que o Qwen 1.5B tem tamanhos maiores que o LLaMa 8B. Já na Tabela 15, verifica-se que o Qwen 1.5B apresenta o maior número de palavras repetidas, seguido do LLaMa 8B, enquanto o Qwen 7B tem o menor número de repetições. Combinando essa análise com a Tabela 12, pode-se concluir que o CoT do Qwen 1.5B e do LLaMa 8B apresentam menor qualidade na tarefa de refatoração de código em comparação Qwen 7B, devido ao número excessivo de palavras desnecessárias geradas.

Teste	P-valor	Conclusão
Número de palavras por CoT do Qwen 1.5B	$2,46 * 10^{-36}$	Não segue a distribuição normal
Número de palavras por CoT do Qwen 7B	$1,06 * 10^{-58}$	Não segue a distribuição normal
Número de palavras por CoT do LLaMa 8B	$1,72 * 10^{-77}$	Não segue a distribuição normal
Número de palavras repetidas por CoT do Qwen 1.5B	0	Não segue a distribuição normal
Número de palavras repetidas por CoT do Qwen 7B	0	Não segue a distribuição normal
Número de palavras repetidas por CoT do LLaMa 8B	0	Não segue a distribuição normal

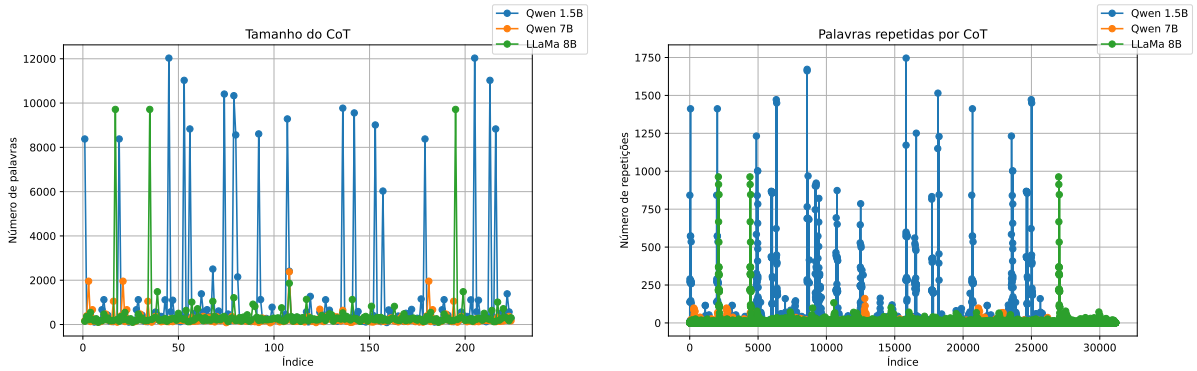
Tabela 13: Testes de D'Agostino-Pearson para distribuições do número de palavras e número de palavras repetidas do CoT

Alternativa	P-valor	Conclusão
Qwen 1.5B >Qwen 7B	$8,01 \times 10^{-14}$	Qwen 1.5B **tem** valores maiores que Qwen 7B
LLaMa 8B >Qwen 7B	$6,42 \times 10^{-14}$	LLaMa 8B **tem** valores maiores que Qwen 7B
Qwen 1.5B >LLaMa 8B	$0,15$	Qwen 1.5B **não tem** valores maiores que o LLaMa 8B

Tabela 14: Testes de Mann-Whitney para distribuições do número de palavras do CoT

Alternativa	P-valor	Conclusão
Qwen 1.5B >Qwen 7B	$1,49 \times 10^{-136}$	Qwen 1.5B **tem** valores maiores que Qwen 7B
LLaMa 8B >Qwen 7B	$5,60 \times 10^{-17}$	LLaMa 8B **tem** valores maiores que Qwen 7B
Qwen 1.5B >LLaMa 8B	$6,86 \times 10^{-73}$	Qwen 1.5B **tem** valores maiores que o LLaMa 8B

Tabela 15: Testes de Mann-Whitney para distribuições do número de palavras repetidas do CoT



(a) Tamanho do CoT

(b) Palavras Repetidas por CoT

Figura 13: Distribuições analisadas no teste de hipótese

5 CONCLUSÃO E TRABALHOS FUTUROS

Este estudo explorou a aplicação de Large Language Models (LLMs) na refatoração automática de Extrair Método em projetos open-source Java, utilizando modelos da família DeepSeek-R1. Os experimentos analisaram as sugestões geradas considerando a Levenshtein Ratio do nome da função sugerida em relação ao baseline e as diferenças nos trechos extraídos.

Inicialmente, foi conduzida uma análise dos principais *code smells* relacionados à refatoração Extrair Método, investigando o estado da arte na detecção automática desses problemas. Em seguida, foi feita uma revisão da literatura para identificar as soluções existentes para a automatização dessa refatoração com LLMs.

O comportamento dos modelos de baixo custo computacional da família do DeepSeek R1 foi analisado a partir dos experimentos em projetos Java. Nesses experimentos, observou-se que a maioria das sugestões de refatoração fornecidas pelos modelos foram descartadas por não se alinharem às refatorações do baseline, resultando em muitas sugestões desnecessárias.

Ao avaliar a utilidade das sugestões, constatou-se que, no teste de hipótese sobre a diferença dos trechos extraídos, nenhum modelo apresentou proximidade suficiente ao baseline, indicando que as sugestões não foram satisfatórias. Adicionalmente, no quesito nomeação das funções, os modelos LLaMa 8B e Qwen 1.5B superaram o desempenho do Qwen 7B. Além disso, ao analisar os Chains of Thought (CoT) dos modelos antes da aplicação da refatoração, verificou-se que, embora os CoTs mantivessem o contexto de refatoração, apenas o gerado pelo LLaMa 8B mencionou explicitamente code smells. Notou-se também que os CoTs do LLaMa 8B e do Qwen 1.5B eram estatisticamente maiores que os do Qwen 7B, sugerindo que esses modelos são menos concisos e diretos na tarefa de refatoração Extrair Método.

Em geral, os resultados dos experimentos foram inferiores ao "Next-generation refactoring: Combining llm insights and ide capabilities for extract method" [16], artigo base para este estudo. Os principais motivos para isso foi a falta de tempo para a execução dos testes diversas vezes, a falta de mão de obra para realizar a análise dos teses e a rotulação dos dados e a impossibilidade de executar os testes com um modelo mais robusto, como o DeepSeek R1 base.

5.1 **Trabalhos futuros**

Com base nos resultados obtidos e nas limitações identificadas, há direções que podem ser seguidas para dar continuação a esse estudo, dentre elas:

- Executar os testes com modelos mais robustos, como o Qwen 14B, Qwen 32B, Llama 70B e até o DeepSeek R1, com 671B de parâmetros
- Executar os testes em outros datasets com outras linguagens de programação
- Alterar o prompt a fim de minimizar as refatorações descartadas
- Executar os testes com o foco em outras refatorações
- Executar os testes com o foco em Maus Cheiros

REFERÊNCIAS

[1] REIS, J. Pereira dos et al. Code smells detection and visualization: a systematic literature review. *Archives of Computational Methods in Engineering*, Springer, v. 29, n. 1, p. 47–94, 2022.

[2] FOWLER, M. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999. ISBN 0-201-48567-2.

[3] NUNES, H. de Medeiros Caseli e Maria das G. V. *Processamento de Linguagem Natural: Conceitos, Técnicas e Aplicações em Português – 2a. Edição*. [S.l.]: BPLN, 2024.

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

[5] ZHANG, A. et al. *Dive into Deep Learning*. [S.l.]: Cambridge University Press, 2023. `https://D2L.ai`.

[6] WASWANI, A. et al. Attention is all you need. In: *NIPS*. [S.l.: s.n.], 2017.

[7] GUO, D. et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[8] SOFTMAX Function: Advantages and Applications — BotPenguin — botpenguin.com. `https://botpenguin.com/glossary/softmax-function`. [Accessed 23-01-2025].

[9] DUBEY, A. et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[10] ACHIAM, J. et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[11] YANG, A. et al. Qwen2. 5-1m technical report. *arXiv preprint arXiv:2501.15383*, 2025.

[12] LIU, A. et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

[13] CODE Smells — refactoring.guru. `https://refactoring.guru/refactoring/smells`. [Accessed 25-01-2025].

[14] HADJ-KACEM, M.; BOUASSIDA, N. Multi-label learning for identifying co-occurring class code smells. *Computing*, Springer, p. 1–28, 2024.

[15] KHLEEL, N. A. A.; NEHÉZ, K. Improving accuracy of code smells detection using machine learning with data balancing techniques. *The Journal of Supercomputing*, Springer, p. 1–46, 2024.

[16] POMIAN, D. et al. Next-generation refactoring: Combining llm insights and ide capabilities for extract method. In: IEEE. *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. [S.l.], 2024. p. 275–287.

[17] SHIRAFUJI, A. et al. Refactoring programs using large language models with few-shot examples. In: IEEE. *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.], 2023. p. 151–160.

[18] DOCUMENTA&XE7;&XE3;O do Cluster Apuana &x2014; documenta&xE7;&xE3;o Cluster Cin latest — apuana.cin.ufpe.br. `https://apuana.cin.ufpe.br/`. [Accessed 27-03-2025].

[19] SLURM Workload Manager - Documentation — slurm.schedmd.com. `https://slurm.schedmd.com/documentation.html`. [Accessed 27-03-2025].

[20] LEVENSHTEIN module &x2014; Levenshtein 0.23.0 documentation — rapidfuzz.github.io. `https://rapidfuzz.github.io/Levenshtein/levenshtein.html#ratio`. [Accessed 23-01-2025].