



**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE  
CENTRO DE ENGENHARIA ELÉTRICA E INFORMÁTICA UNIDADE  
ACADÊMICA DE SISTEMAS E COMPUTAÇÃO PROGRAMA DE  
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**JOÃO VICTOR SOARES DE ALMEIDA**

**UMA INVESTIGAÇÃO SOBRE COMO LLaMA-2 13B REVISAR  
CÓDIGO-FONTE COM ÊNFASE EM SMELL**

**CAMPINA GRANDE - PB**

**2025**

A347i

Almeida, João Victor Soares de.

Uma investigação sobre como LLaMA-2 13B revisa código-fonte com ênfase em smell / João Victor Soares de Almeida. - Campina Grande, 2025.

76 f. : il. color.

Dissertação (Mestrado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2025.

"Orientação: Prof. Dr. João Arthur Brunet Monteiro."

Referências.

1. Engenharia de Software. 2. Revisão de Código. 3. LLaMA-2 13B. 3. *Pull Requests*. 4. *code-smells*. 5. Análise Sistemática. 6. Prompt. I. Monteiro, João Arthur Brunet. II. Título.

CDU 004.41(043)

Universidade Federal de Campina Grande  
Centro de Engenharia Elétrica e Informática  
Coordenação de Pós-Graduação em Ciência da Computação

Uma investigação sobre como LLaMA-2 13B revisa  
código-fonte com ênfase em smell

João Victor Soares de Almeida

Dissertação submetida à Coordenação do Curso de Pós-Graduação em  
Ciência da Computação da Universidade Federal de Campina Grande -  
Campus I como parte dos requisitos necessários para obtenção do grau  
de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

João Arthur Brunet Monteiro

(Orientador)

Campina Grande, Paraíba, Brasil

©João Victor Soares de Almeida, 02/05/2025



MINISTÉRIO DA EDUCAÇÃO  
**UNIVERSIDADE FEDERAL DE CAMPINA GRANDE**  
POS-GRADUACAO EM CIENCIA DA COMPUTACAO  
Rua Aprígio Veloso, 882, Edifício Telmo Silva de Araújo, Bloco CG1, - Bairro Universitário, Campina Grande/PB, CEP 58429-900  
Telefone: 2101-1122 - (83) 2101-1123 - (83) 2101-1124  
Site: <http://computacao.ufcg.edu.br> - E-mail: [secpg@computacao.ufcg.edu.br](mailto:secpg@computacao.ufcg.edu.br)

## FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

**JOÃO VICTOR SOARES DE ALMEIDA**

### UMA INVESTIGAÇÃO SOBRE COMO LLAMA-2 13B REVISAR CÓDIGO-FONTE COM ÊNFASE EM SMELL

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Mestre em Ciência da Computação.

Aprovada em: 11/04/2025

Prof. Dr. **João Arthur Brunet Monteiro**, UFCG, Orientador

Prof. Dr. **Tiago Lima Massoni**, UFCG, Examinador Interno

Prof. Dr. **Leopoldo Motta Teixeira**, UFPE, Examinador Externo



Documento assinado eletronicamente por **JOAO ARTHUR BRUNET MONTEIRO, PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 14/04/2025, às 08:50, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da [Portaria SEI nº 002, de 25 de outubro de 2018](#).



Documento assinado eletronicamente por **Leopoldo Motta Teixeira, Usuário Externo**, em 14/04/2025, às 11:17, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da [Portaria SEI nº 002, de 25 de outubro de 2018](#).



Documento assinado eletronicamente por **TIAGO LIMA MASSONI, PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 14/04/2025, às 16:23, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da [Portaria SEI nº 002, de 25 de outubro de 2018](#).



A autenticidade deste documento pode ser conferida no site <https://sei.ufcg.edu.br/autenticidade>, informando o código verificador **5377231** e o código CRC **DEB07DB1**.

---

Referência: Processo nº 23096.019766/2025-16

SEI nº 5377231

*Lancei minha sorte na estrada, e toda vez que o sol cruza o céu, eu sou seu, meu Jesus. Meu papel, cada dia, é tentar ser melhor. Tem lágrima misturada com meu suor, e olha só o que foi que aconteceu: Levantar e agradecer porque Deus escolheu, com humildade e disciplina dos sermões que mãe me deu*

— João Gomes

## Resumo

A revisão de código em projetos de código aberto é uma prática comum e essencial no desenvolvimento de *software*, visando garantir a qualidade do código-fonte e detectar problemas na implementação. No entanto, embora essencial, essa prática manual pode se tornar dispendiosa e suscetível a erros, especialmente em projetos maiores e colaborativos. Diante deste cenário, investigamos como o *Large Language Model Meta AI* (LLaMA-2 13B) pode contribuir especificamente na revisão de *code smells*, buscando compreender suas capacidades e limitações no ciclo de desenvolvimento. Nossa investigação baseou-se em dados extraídos de projetos de código aberto consolidados como **Neovim**, **Keycloak** e **gRPC**. Partindo de **19.149 comentários** distribuídos em **6.365 Pull Requests**, aplicamos uma abordagem híbrida que consistiu em filtragem sistemática por palavras-chave seguida de análise manual dos comentários, resultando em um *dataset* focado em *code smells* de **3.023 comentários**. Após desenvolver um *prompt* específico para orientar as revisões do modelo, selecionamos uma amostra estratificada de **637 comentários (21,10% do dataset)** para uma avaliação detalhada. Os resultados revelaram que **91,73% das revisões** do modelo apresentaram baixa similaridade com as revisões humanas. Nossa análise qualitativa identificou que em **72% das intervenções** o modelo diverge do foco dos revisores humanos, embora forneça análises tecnicamente abrangentes em **48,3% dos casos**. Os resultados sugerem que, embora o LLaMA-2 13B seja capaz de realizar análises relevantes, suas limitações de contexto resultam em revisões que frequentemente divergem do foco dos revisores humanos. Por fim, concluímos que o **modelo pode ser mais efetivo quando utilizado como ferramenta complementar à revisão humana**, não como substituto.

**Palavras-chave:** Revisão de código; *code smells*; LLaMA-2 13B; *Pull Requests*; análise sistemática; *prompt*.

## Abstract

Code review in open source projects is a common and essential practice in software development, aiming to ensure source code quality and detect implementation issues. However, although essential, this manual practice can become costly and error-prone, especially in larger and collaborative projects. In this context, we investigate how the Large Language Model Meta AI (LLaMA-2 13B) can specifically contribute to the review of code smells, seeking to understand its capabilities and limitations in the development cycle. Our investigation was based on data extracted from consolidated open source projects such as **Neovim, Keycloak, and gRPC**. Starting from **19,149 comments** distributed across **6,365 Pull Requests**, we applied a hybrid approach consisting of systematic keyword filtering followed by manual analysis of comments, resulting in a code smell-focused dataset of **3,023 comments**. After developing a specific prompt to guide the model's reviews, we selected a stratified sample of **637 comments (21.10% of the dataset)** for detailed evaluation. The results revealed that **91.73% of the model's reviews** showed low similarity to human reviews. Our qualitative analysis identified that in **72% of interventions** the model diverges from human reviewers' focus, although it provides technically comprehensive analyses in **48.3% of cases**. The results suggest that, while LLaMA-2 13B is capable of performing relevant analyses, its context limitations result in reviews that frequently diverge from human reviewers' focus. Finally, we conclude that the **model can be more effective when used as a complementary tool to human review**, not as a substitute.

**Keywords:** Code review; code smells; LLaMA-2 13B; Pull Requests; systematic analysis; prompt.

## Agradecimentos

Início meus agradecimentos pela minha mãe, a maior incentivadora dos meus sonhos. Seu apoio constante e dedicação em ver minha felicidade são incomparáveis. Muito do que sou e conquisto vem da coragem que ela me transmite. Como disse Gibran Khalil, "A mãe é tudo. Ela é nosso consolo na tristeza, nossa esperança na miséria e nossa força na fraqueza". Sou infinitamente grato por tê-la como meu alicerce. Aos meus padrinhos, obrigado por estarem sempre presentes, me guiando nos caminhos da sabedoria. Vocês são faróis a iluminar minha jornada.

À minha amada Thaynara Guilherme, minha profunda gratidão por compartilhar meus sonhos e se dedicar a me apoiar. Como disse Antoine de Saint-Exupéry, "Amar não é olhar um para o outro, é olhar juntos na mesma direção". Sou abençoado por trilhar essa jornada ao seu lado. Ao meu mentor, Tiago Neto, meu reconhecimento por despertar em mim a paixão pelo universo acadêmico. Como bem disse Nelson Mandela, "A educação é a arma mais poderosa que você pode usar para mudar o mundo".

Meu coração transborda gratidão aos amigos de infância e da vida, em especial ao grupo Alteração. Vocês são os irmãos que a vida me presenteou. A amizade de vocês é um bem precioso. Ao meu grupo de estudo, obrigado pela colaboração. Victor, Iriedson e José vocês foram essenciais nessa caminhada. Não poderia deixar de agradecer à minha querida amiga Witoria Manuely, que sempre se faz presente em vários momentos da minha vida. Seu apoio, companheirismo e amizade são essenciais para mim. Obrigado por estar ao meu lado nessa caminhada.

Por fim, minha gratidão ao meu mentor profissional, Rogério Volsi. Você me inspira diariamente a ser melhor. Como disse Ralph Waldo Emerson, "O propósito da vida não é ser feliz. É ser útil, honrado, compassivo, ter feito alguma diferença". Você me ensina a viver bem e fazer a diferença.

Não poderia deixar de expressar minha imensa gratidão ao meu orientador, João Arthur, por sua dedicação, paciência e sabedoria ao longo deste processo. Seus sermões e ensinamentos foram fundamentais para o meu crescimento acadêmico. Obrigado por ser um guia nesta jornada de aprendizado. A todos, meu muito obrigado!

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>1</b>
1.1	Problema . . . . .	1
1.2	Solução . . . . .	3
1.3	Principais Contribuições . . . . .	4
1.4	Resultados . . . . .	5
1.5	Estrutura da Dissertação . . . . .	6
<b>2</b>	<b>Background</b>	<b>7</b>
2.1	Revisão de Código . . . . .	7
2.2	Code Smells . . . . .	8
2.3	Large Language Models . . . . .	9
2.4	Prompt Engineering . . . . .	10
2.5	Codificação Sistemática . . . . .	14
<b>3</b>	<b>Trabalhos relacionados</b>	<b>15</b>
<b>4</b>	<b>Avaliação de Revisões de Código pelo modelo LLaMA-2 13B</b>	<b>20</b>
4.1	Metodologia . . . . .	20
4.1.1	Coleta de dados . . . . .	21
4.1.2	Refinamento . . . . .	28
4.1.3	Execução do experimento com o LLaMA-2 13B . . . . .	30
4.1.4	Validação das respostas do LLaMA-2 13b . . . . .	34
4.1.5	Avaliação comparativa entre as revisões do modelo e as realizadas por humanos . . . . .	36
4.2	Artefatos . . . . .	41
4.2.1	Datasets . . . . .	41

---

4.2.2	Scripts	42
4.2.3	Documentação	43
<b>5</b>	<b>Resultados e discussões</b>	<b>44</b>
5.0.1	Análise Quantitativa	44
5.0.2	Análise Qualitativa	47
5.1	Ameaças à Validade	60
<b>6</b>	<b>Conclusão</b>	<b>62</b>
<b>7</b>	<b>Trabalhos futuros</b>	<b>64</b>
<b>A</b>	<b>Roteiro 1: Procedimento para análise de tuplas de comentários (humano/modelo)</b>	<b>70</b>
A.1	Objetivo	70
A.2	Materiais necessários	70
A.3	Procedimento de análise	70
<b>B</b>	<b>Roteiro 2: Procedimento para codificação sistemática</b>	<b>73</b>
B.1	Objetivo	73
B.2	Materiais necessários	73
B.3	Procedimento de codificação	73
B.3.1	Fase 1: Codificação Inicial (50 comentários)	73
B.3.2	Fase 2: Codificação completa	74
B.3.3	Fase 3: Análise e Interpretação	75
B.4	Exemplos de Aplicação do Roteiro	76
B.4.1	Exemplo de codificação: CONVIABRANG (Convergente e Abrangente)	76

# Lista de Símbolos

LLMs - *Large Language Models*

LLaMA - *Large Language Model Meta AI*

PRs - *Pull Requests*

PLN - *Processamento de Linguagem Natural*

API - *Application Programming Interface*

CSV - *Comma-Separated Values*

JSON - *JavaScript Object Notation*

# Lista de Figuras

3.1	Roteiro do protocolo de pesquisa . . . . .	16
4.1	Processo de pré-processamento dos dados estruturado em três fases. . . . .	28
4.2	Visão geral do fluxo de execução do experimento com o modelo LLaMA-2 13B. . . . .	30
4.3	Visão geral do fluxo de validação das respostas geradas pelo LLaMA-2 13B . . . . .	34
4.4	Visão geral do processo de avaliação . . . . .	36
4.5	Processo de categorização manual . . . . .	39
4.6	Processo de codificação inicial com validação de um humano . . . . .	40
4.7	Processo de codificação da segunda fase . . . . .	41

# Lista de Tabelas

4.1	Projetos do GitHub com maior média de comentários em revisões . . . . .	25
4.2	Palavras-chave de <i>code smells</i> utilizadas na mineração . . . . .	27
4.3	Exemplo do formato do <i>dataset</i> elaborado . . . . .	28
4.4	Evolução quantitativa do <i>dataset</i> durante o pré-processamento . . . . .	29
4.5	Definição dos códigos de categorização dos comentários do modelo. . . . .	40
5.1	Distribuição acumulada dos pares por intervalo de similaridade . . . . .	45
5.2	Distribuição da Amostragem por Faixa de Similaridade . . . . .	47
5.3	Distribuição das Categorias por Faixa de Similaridade . . . . .	48
5.4	Distribuição dos códigos nas revisões do LLaMA-2 13B . . . . .	53

# Lista de Códigos Fonte

2.1	Exemplo de God Class . . . . .	8
4.1	Função para calcular a média de comentários por PR . . . . .	21
4.2	Função para enviar requisições à API do GitHub. . . . .	24
4.3	Trecho de código responsável pela interação com o modelo LLaMA-2 13B . . . . .	33
4.4	Pré-processamento dos dados de texto . . . . .	38
4.5	Vetorização do texto usando TF-IDF . . . . .	38
4.6	Cálculo da similaridade de cosseno . . . . .	38

# Capítulo 1

## Introdução

### 1.1 Problema

Com o crescimento exponencial dos *Large Language Models* (LLMs), a investigação sobre seu potencial para automatizar atividades críticas na Engenharia de *Software*, como a revisão de código, tornou-se uma linha promissora e necessária. Ferramentas modernas, como o *StackSpot AI* [26], já demonstram o valor dos LLMs ao fornecer suporte automatizado no desenvolvimento colaborativo de *software*, promovendo produtividade e otimizando o desenvolvimento. Esses modelos, pré-treinados em vastos volumes de dados textuais, apresentam desempenho expressivo em tarefas de Processamento de Linguagem Natural (PLN), como tradução automática, sumarização e análise de sentimentos [17]. No entanto, a aplicação dos LLMs em revisões de código, especialmente em sugestões de melhorias para *code smells*, ainda é incipiente e carece de validação empírica mais aprofundada.

As primeiras tentativas de revisão de código começaram com abordagens formais, como a proposta por Fagan (1976), que enfatizava processos bem estruturados e análises minuciosas [8]. No entanto, com a evolução das metodologias de desenvolvimento, o aumento da complexidade dos projetos e a adoção de práticas ágeis, tornou-se necessário adotar abordagens mais práticas, flexíveis e orientadas à colaboração contínua. Ao longo dos anos, essas abordagens foram evoluindo e dando lugar a novas práticas, como o *Modern Code Review* (MCR) [1]. O MCR se destaca por promover revisões incrementais e contínuas, integradas diretamente ao fluxo de trabalho diário dos desenvolvedores, permitindo que as alterações sejam analisadas em pequenos blocos à medida que são produzidas, e não apenas ao final de cada fase do projeto [31]. Essa prática é amplamente facilitada por plataformas colaborativas, como GitHub e

GitLab [13] [14], que oferecem ferramentas como *pull requests* (PRs) e *merge requests* (MRs) para viabilizar revisões rápidas e colaborativas. Entretanto, mesmo com essas evoluções, revisar *code smells* durante o MCR continua sendo uma tarefa desafiadora, especialmente em projetos maiores e mais complexos.

Nesse cenário, os *code smells* se destacam como um aspecto crítico da qualidade de *software*. Eles indicam problemas estruturais que, embora não sejam erros explícitos, podem comprometer a manutenibilidade e evolução do código ao longo do tempo [12]. Exemplos comuns incluem dependências excessivas, *God Classes* (classes com responsabilidades excessivas), *Deado code* (código morto) e repetições sem modularidade. A correção desses problemas é essencial não apenas para manter a qualidade do *software*, mas também para evitar o acúmulo de dívida técnica, que pode dificultar a manutenção futura [39].

Estudos recentes sugerem que modelos pré-treinados, como o *CodeReviewer* e o T5, podem automatizar partes da revisão de código, auxiliando na identificação de potenciais melhorias e na sugestão de alterações durante a análise de PRs [19][39]. Embora esses estudos ofereçam contribuições relevantes e consistentes para a automação da revisão de código, ainda carecem de uma avaliação específica voltada à revisão de *code smells*. Assim, a aplicação de LLMs na revisão de código - como o LLaMA-2 13B - surge como uma oportunidade, mas ainda precisa ser validada em termos de capacidade e alinhamento com as práticas humanas.

Para este estudo, optamos pelo modelo LLaMA-2 13B desenvolvido pela Meta AI, uma escolha fundamentada em múltiplos fatores estratégicos. Este modelo foi escolhido por sua arquitetura e disponibilidade pública mediante licença, permitindo não apenas sua utilização experimental, mas também o acesso a informações detalhadas sobre sua estrutura e treinamento. Conforme demonstrado por Touvron et al. [36], o LLaMA-2 13B apresenta capacidades expressivas de processamento e compreensão de tarefas complexas em linguagem natural, características essenciais para nosso objetivo de aplicação em revisão de código. Além disso, seu balanceamento entre dimensão (13 bilhões de parâmetros) e eficiência computacional oferece um equilíbrio adequado para análises detalhadas sem exigir recursos computacionais proibitivos. A reputação da Meta como instituição de referência no desenvolvimento de modelos de linguagem, combinada com a transparência na documentação técnica do modelo, foram fatores decisivos para esta escolha. Por fim, esta pesquisa investiga como esse modelo se comporta na revisão de *code smells*, avaliando se suas revisões são comparáveis — ou potencialmente complementares — às realizadas por revisores humanos experientes.

É importante destacar que, devido à limitação de contexto de 4096 *tokens* imposta pelo modelo LLaMA-2 13B, nossa abordagem experimental foi condicionada à análise de trechos isolados de código (*diff hunks*) em vez de arquivos completos. Essa restrição técnica impacta a quantidade de informações contextuais disponíveis para o modelo durante a revisão, podendo influenciar as diferenças observadas entre as revisões geradas automaticamente e as revisões humanas. Essa limitação foi considerada ao longo da análise dos resultados e na discussão das ameaças à validade deste estudo.

## 1.2 Solução

Este estudo tem como objetivo investigar como o modelo LLaMA-2 13B pode contribuir para a automação de revisões de código, especialmente na identificação e análise de *code smells*. A motivação principal é compreender em que medida o modelo é capaz de fornecer revisões similares às humanas e como ele pode complementar ou até superar práticas de revisão realizadas por humano. Para isso, construímos um *dataset* com **3.023** comentários extraídos de PRs, todos com foco em *code smells*. A partir desse *dataset*, aplicamos um *prompt* cuidadosamente projetado para orientar o modelo LLaMA-2 13B na realização das revisões de forma automatizada.

Por fim, realizamos uma análise comparativa entre as revisões automáticas e humanas para avaliar em que medida e quão similares são as respostas fornecidas pelo modelo em comparação às revisões conduzidas por desenvolvedores humanos. Dessa forma, para nortear nosso estudo, definimos as seguintes perguntas de pesquisa:

- **QP1:** As revisões realizadas pelo LLaMA-2 13B são comparáveis às revisões conduzidas por humanos?
- **QP2:** Em que medida o modelo é capaz de superar ou complementar as práticas de revisão humana?

Assim, para responder a essas perguntas, aplicamos uma abordagem mista de análise, combinando métodos quantitativos e qualitativos:

- **Análise quantitativa:** Avaliamos a similaridade entre as revisões humanas e automáticas utilizando vetorização TF-IDF e cálculo da distância do cosseno, permitindo medir a proximidade das respostas do modelo em relação às revisões humanas;

- **Análise qualitativa:** Realizamos uma codificação aberta (*open coding*) e baseada em categorias pré-definidas (*priori coding*), seguindo a abordagem sugerida por Miles (1994)[23]. As revisões foram categorizadas para identificar padrões e compreender a natureza das respostas do modelo em relação ao contexto humano.

Essa combinação de análises permitiu identificar em que aspectos as revisões realizadas pelo modelo se aproximam ou divergem das práticas humanas, oferecendo observações sobre suas possíveis aplicações e limitações. Por essa razão, os detalhes sobre cada uma dessas etapas serão apresentados no [Capítulo 3](#), onde exploramos todo o processo metodológico de construção do *dataset*, a configuração do modelo e as estratégias adotadas para assegurar uma execução alinhada com os objetivos do estudo.

## 1.3 Principais Contribuições

Este estudo teve como objetivo investigar a eficiência do modelo LLaMA-2 13B na revisão de *code smells* e analisar a natureza das intervenções realizadas pelo modelo. A pesquisa foi conduzida a partir de um *dataset* cuidadosamente elaborado, e empregou tanto análises quantitativas quanto qualitativas para oferecer uma visão abrangente sobre as capacidades e limitações do modelo. Ao longo do processo, algumas observações significativas foram feitas, culminando nas seguintes contribuições principais:

1. *Dataset* estruturado e anotado para revisão de *code smells*: Desenvolvemos e disponibilizamos publicamente um *dataset* contendo 3.023 comentários de revisões automáticas realizadas pelo LLaMA-2 13B, com foco específico em *code smells*. Este *dataset* representa uma contribuição singular para a comunidade de pesquisa, pois: (i) integra revisões humanas e suas contrapartes automatizadas, permitindo análises comparativas diretas; (ii) abrange múltiplas linguagens de programação (C/Lua, C++ e Java), oferecendo diversidade contextual; (iii) está estruturado com metadados completos, incluindo *diff hunks*, comentários originais e respostas do modelo, facilitando sua aplicação em diversos estudos; e (iv) pode servir como base para treinamento, ajuste fino ou avaliação de outros modelos de linguagem no contexto de revisão de código. Este recurso preenche uma lacuna importante na literatura, onde datasets específicos para avaliação de LLMs em revisão de *code smells* são escassos;

2. Análise contextual automatizada, para selecionar apenas as revisões que contribuem efetivamente para a análise, excluindo respostas que indicavam limitações do modelo;
3. Este estudo oferece observações tanto para a comunidade acadêmica quanto para a indústria, ao demonstrar as limitações e possibilidades do LLaMA-2 13B em comparação com revisores humanos, promovendo o debate sobre o uso de LLMs na engenharia de *software*.

## 1.4 Resultados

Nossa análise foi conduzida sobre uma amostra estratificada de 637 comentários (21,10%) extraídos de um *dataset* final de 3.023 comentários, obtido após uma etapa de refinamento dos dados brutos. Os resultados quantitativos revelaram que 91,73% das revisões do modelo apresentaram baixa similaridade com as revisões humanas (faixa de 0-20% de similaridade). Através de uma análise qualitativa sistemática (utilizando *open-coding* e *priori-coding*), identificamos que 72% das intervenções do modelo divergem substancialmente do contexto humano, enquanto 48,3% das revisões demonstram características de abrangência técnica em suas análises. Quanto à qualidade técnica das revisões, observamos que o modelo, embora divergente em foco, apresentou consistência técnica significativa, com apenas **0,5% das revisões classificadas como não relevantes**. O LLaMA-2 13B demonstrou capacidade de identificar e analisar múltiplos aspectos do código simultaneamente, oferecendo em 48,3% dos casos análises tecnicamente abrangentes que abordam não apenas o problema central, mas também questões de nomenclatura, documentação, tratamento de erros e design de código. Esta combinação entre baixa similaridade com revisões humanas e alta abrangência técnica revela um padrão intrigante, sugerindo que o modelo complementa, **mais do que replica, a abordagem de revisores humanos**.

Esta combinação de resultados revela um aspecto interessante do comportamento do modelo: enquanto suas revisões abordam múltiplos aspectos do código e oferecem sugestões pontuais para cada um deles, elas frequentemente seguem uma direção diferente das prioridades estabelecidas pelos revisores humanos. Por exemplo, onde um revisor humano foca em um problema específico de *design*, o modelo tende a expandir sua análise para questões de documentação, tratamento de erros, nomenclatura e outros aspectos técnicos, mesmo que não sejam a preocupação central da revisão. Este padrão pode ser atribuído principalmente a dois fatores:

a limitação do modelo em processar apenas trechos isolados de código (*diff hunks*) e a restrição no tamanho do contexto que pode ser analisado devido ao limite de *tokens*. Estas características sugerem que o LLaMA-2 13B pode ser mais efetivo como ferramenta complementar ao processo de revisão humana, aproveitando sua capacidade de identificar múltiplos aspectos técnicos enquanto depende do revisor humano para estabelecer prioridades e contexto apropriado.

Os resultados completos e suas implicações serão discutidos detalhadamente no [Capítulo 4](#), onde exploramos as observações com maior profundidade e refletimos sobre os desafios e potencialidades da aplicação do LLaMA-2 13B em revisões de *code smells*.

## 1.5 Estrutura da Dissertação

Este documento está organizado da seguinte forma: O [Capítulo 2](#) apresenta a fundamentação teórica necessária para a compreensão deste trabalho. O [Capítulo 3](#) descreve em detalhe a metodologia aplicada, incluindo o plano de construção do *dataset*, a aplicação de *prompt engineering*, e os processos de filtragem para coletar os comentários relevantes. Além disso, abrange as etapas de análise quantitativa e qualitativa realizadas, evidenciando o procedimento envolvido em cada uma dessas fases e apresentando os resultados obtidos. O [Capítulo 4](#) oferece uma visão geral dos trabalhos relacionados. Por fim, o [Capítulo 5](#) conclui este estudo, destacando as principais observações e propondo direções para trabalhos futuros.

# Capítulo 2

## Background

Este capítulo apresenta os conceitos fundamentais necessários para a compreensão deste trabalho. Inicialmente, discutimos a prática de revisão de código e sua evolução ao longo do tempo. Na sequência, exploramos o impacto dos *code smells* na qualidade do *software*. Em seguida, introduzimos o tema de *Large Language Models* (LLMs), fornecendo uma visão geral de suas capacidades, limitações e aplicações promissoras na engenharia de *software*. Nesse contexto, também justificamos a escolha do LLM utilizado em nossa pesquisa. Por fim, apresentamos os conceitos de *prompt engineering* e codificação sistemática, técnicas essenciais para o desenvolvimento de nossa metodologia.

### 2.1 Revisão de Código

O processo de revisão de código é uma etapa relevante no desenvolvimento de *software*, pois estabelece um protocolo em que desenvolvedores avaliam as alterações propostas por seus pares antes da integração ao projeto principal. Dessa forma, esta prática fortalece não apenas a detecção preventiva de problemas, mas também consolida padrões de qualidade e fomenta o compartilhamento de conhecimento técnico entre os membros da equipe [31].

Esta prática evoluiu significativamente ao longo do tempo, tendo início com as inspeções formais de código introduzidas por Fagan em 1976, que estabeleceram os primeiros processos estruturados de revisão. A transição de metodologias tradicionais para abordagens ágeis, somada ao crescimento de projetos distribuídos e colaborativos, levou ao surgimento do *Modern Code Review* (MCR), caracterizado por sua natureza mais dinâmica e contínua [1]. Esta nova abordagem revolucionou o processo de revisão ao introduzir características como:

- Suporte ferramental robusto através de plataformas colaborativas,
- Flexibilidade no processo, dispensando documentações extensivas;
- Revisões incrementais focadas em conjuntos específicos de mudanças;
- Integração fluida ao ciclo de desenvolvimento.

Apesar dos avanços nas práticas de revisão, os projetos contemporâneos enfrentam desafios cada vez mais complexos. O volume crescente de código a ser revisado, combinado com ciclos de desenvolvimento mais curtos e pressão por entregas rápidas, frequentemente força os revisores a equilibrar profundidade de análise com tempo disponível. Esta situação torna-se ainda mais crítica devido à sofisticação dos sistemas modernos, onde uma única mudança pode ter ramificações em múltiplos componentes do *software*, exigindo dos revisores uma compreensão cada vez mais abrangente do sistema. Diante deste cenário desafiador, iniciativas de automação do suporte à revisão vêm ganhando destaque [40], buscando complementar o trabalho dos revisores humanos e tornar o processo mais eficiente sem comprometer sua qualidade.

## 2.2 Code Smells

Durante o ciclo de vida de um *software*, o código-fonte pode acumular estruturas e padrões que, embora não afetem seu funcionamento imediato, sinalizam potenciais problemas de design e manutenibilidade. Estes indicadores, denominados *code smells* por Fowler e Beck [12], representam violações sutis de princípios de *design* que podem comprometer a evolução e manutenção do sistema ao longo do tempo.

Estes indicadores manifestam-se de diversas formas, incluindo problemas estruturais como métodos excessivamente longos (*Long Method*), classes com responsabilidades múltiplas (*God Class*), duplicação de código (*Duplicated Code*), e acoplamento excessivo entre componentes (*Feature Envy*) [12]. O Código 2.1 apresenta um exemplo de *God Class*, onde uma única classe assume múltiplas responsabilidades, violando o princípio da responsabilidade única.

Código Fonte 2.1: Exemplo de God Class

```
1 public class UserManager {
2     private Database db;
3     private EmailService emailService;
4     private Logger logger;
```

```
5
6 // Gerenciamento de usuarios
7 public void createUser(User user) { ... }
8 public void deleteUser(int userId) { ... }
9 public User findUser(int userId) { ... }
10
11 // Gerenciamento de autenticacao
12 public boolean login(String username, String password) { ... }
13 public void logout(int userId) { ... }
14 public void resetPassword(String email) { ... }
15
16 // Gerenciamento de permissoes
17 public void addRole(int userId, String role) { ... }
18 public void removeRole(int userId, String role) { ... }
19 public boolean hasPermission(int userId, String permission) { ... }
20
21 // Gerenciamento de notificacoes
22 public void sendWelcomeEmail(User user) { ... }
23 public void sendPasswordResetEmail(String email) { ... }
24 public void notifyAdmins(String message) { ... }
25 }
```

No contexto de projetos colaborativos, onde múltiplos desenvolvedores contribuem simultaneamente, a identificação consistente de *code smells* assume importância ainda maior, pois seu acúmulo pode resultar em dívida técnica crescente, comprometendo a evolução do *software* e aumentando os custos de manutenção.

## 2.3 Large Language Models

Os LLMs são modelos de aprendizado de máquina que ganharam destaque nos últimos anos por sua capacidade de processar e gerar linguagem natural de forma automatizada. Treinados em grandes quantidades de texto não estruturado, esses modelos são capazes de aprender e gerar linguagem natural de forma coerente e contextualmente relevante [3]. Arquiteturas como *Transformers* [41] e técnicas de treinamento como aprendizado auto-supervisionado [6] foram fundamentais para o desenvolvimento dos LLMs. No entanto, apesar de seus impressionantes resultados em tarefas de processamento de linguagem natural, os LLMs também apresentam

limitações, como a tendência a gerar informações incorretas ou enviesadas [2].

Dada a sua capacidade de compreender e gerar código, os LLMs têm atraído interesse significativo na comunidade de engenharia de *software*. Pesquisas recentes exploram o uso de LLMs para tarefas como completção de código [4], geração de testes [38] e detecção de vulnerabilidades [25]. Ferramentas como o **GitHub Copilot** e o **Amazon CodeWhisperer** utilizam LLMs para oferecer sugestões de código em tempo real, buscando aumentar a produtividade dos desenvolvedores.

Neste estudo, optamos por utilizar o LLaMA-2 13B [36], um LLM desenvolvido pela Meta AI. A escolha desse modelo foi impulsionada por uma combinação de fatores que julgamos adequado para nossa investigação no contexto de revisão código. O LLaMA-2 13B está disponível para utilização mediante aprovação de licença, o que nos permitiu empregá-lo em nosso experimento e explorar informações sobre sua arquitetura e treinamento. O estudo de Touvron et al. (2023) [36], foi essencial para aprofundar nosso entendimento sobre os dados de treinamento, o funcionamento do modelo e compreender como a capacidade do modelo poderia ser aplicada em nossa pesquisa.

Além disso, o LLaMA-2 13B demonstrou habilidades de processamento e compreensão de tarefas complexas em linguagem natural, conforme evidenciado por seus resultados [36]. Essas capacidades foram essenciais para nosso objetivo de aplicar o modelo em revisão de código.

Durante o planejamento desta pesquisa, consideramos o cenário de rápida evolução dos LLMs. Embora novos modelos estivessem emergindo, a escolha do LLaMA-2 13B se destacou não apenas por seu desempenho demonstrado nos estudos supracitados, mas também pela credibilidade da Meta como instituição renomada no campo de IA. Entendemos que a reputação da Meta na vanguarda do desenvolvimento de LLMs, combinada com a transparência e acesso às informações de arquitetura bem como dados de treinamento, tornou o LLaMA-2 13B uma escolha confiável e estratégica para nosso estudo.

## 2.4 Prompt Engineering

No contexto de revisão de código utilizando LLMs, a engenharia de prompts emerge como um elemento crucial para obter resultados efetivos. Esta prática consiste no *design* de instruções em linguagem natural para otimizar o uso de sistemas de IA generativa, onde a qualidade do resultado depende diretamente da construção adequada destas instruções [5]. Para atingir este

objetivo, a engenharia de prompts emprega diversas estratégias e técnicas visando melhorar a qualidade e consistência das respostas geradas pelo modelo [32].

Uma das abordagens utilizadas é a *zero-shot prompting*, em que o modelo recebe apenas a instrução da tarefa, sem exemplos adicionais. Outra técnica comum é a *few-shot prompting*, onde são fornecidos alguns exemplos do tipo de análise esperada para guiar o comportamento do modelo. Alternativamente, a técnica de *chain-of-thought prompting* pode ser empregada, orientando o modelo através de um processo estruturado de análise e explicitando cada etapa do raciocínio, o que pode melhorar a coerência e qualidade das respostas. Essas são apenas algumas das muitas estratégias que podem ser utilizadas na engenharia de prompts, cada uma com suas próprias vantagens e aplicações.

No desenvolvimento do nosso prompt para o estudo com o LLaMA-2 13B, utilizamos a técnica de *role prompting* para definir claramente o papel do modelo como um programador experiente, com o objetivo de analisar e revisar trechos de código com *smells*. O *role prompting* consiste em atribuir ao modelo uma persona específica com características e níveis de expertise definidos, o que direciona o comportamento do modelo para emular o conhecimento e abordagem de um especialista na área. No nosso caso, definimos o modelo como um "programador sênior com ampla experiência em análise de qualidade de código e identificação de problemas de design".

Além disso, seguimos as diretrizes específicas de formatação do LLaMA-2, considerando os seguintes aspectos técnicos:

- Utilizamos os *tokens* especiais `[INST]` e `[/INST]` para delimitar o escopo das orientações fornecidas ao modelo. Estes *tokens* são interpretadores especiais do LLaMA-2 que funcionam como marcadores que sinalizam ao modelo onde as instruções começam e terminam, permitindo uma clara separação entre comandos e conteúdo a ser analisado.

Por exemplo:

```
[INST]
Analise o seguinte trecho de código Java e identifique
\textit{code smells}:

public void metodo() {
    // código aqui
```

```
}  
[/INST]
```

- Aplicamos a tag `<SYS>` para definir o comportamento base do modelo, estabelecendo as diretrizes gerais. Esta *tag* funciona como um "*system prompt*" que configura o comportamento global do modelo durante toda a interação, sendo posicionada dentro dos tokens `[INST]`, como demonstrado abaixo:

```
[INST]  
<SYS>  
Você é um programador experiente especializado em análise de  
código e identificação de code smells.  
</SYS>  
  
[conteúdo do prompt]  
[/INST]
```

- Incluímos instruções detalhadas dentro das tags `[INST]` e `[/INST]` sobre como o modelo deveria proceder ao analisar o código fornecido. Estas instruções seguiram uma estrutura hierárquica, definindo claramente:
  1. A tarefa principal (análise de code smells)
  2. Os critérios específicos para análise
  3. O formato desejado para a resposta
- Adotamos a formatação recomendada para o LLaMA-2, incluindo espaços em branco e quebras de linha adequadas para garantir a correta interpretação das instruções. Esta formatação é crucial, pois o modelo é sensível à estrutura do texto, e a correta separação entre seções do prompt influencia diretamente o processamento e interpretação das instruções;

Para ilustrar concretamente a aplicação destes conceitos, apresentamos um exemplo simplificado de como estruturamos o prompt:

[INST]

<SYS>

Você é um programador sênior com expertise em identificação de code smells e aplicação de princípios de Clean Code.

</SYS>

TAREFA: Analise o seguinte código Java e identifique os code smells presentes.

```
public void calcularTotal(List<Item> items) {
    double total = 0;
    for (int i = 0; i < items.size(); i++) {
        if (items.get(i).tipo == 1) {
            total += items.get(i).valor * 0.9;
        } else {
            total += items.get(i).valor;
        }
    }
    System.out.println("Total: " + total);
}
```

INSTRUÇÕES:

- Identifique todos os code smells presentes
- Explique por que cada smell é problemático
- Sugira refatorações específicas

[/INST]

Por fim, incorporamos ao *prompt* informações relevantes para a tarefa de revisão, como a linguagem de programação utilizada, pois isto ajuda o modelo a contextualizar corretamente a análise e aplicar práticas específicas daquela linguagem. Essa estruturação cuidadosa proporcionou consistência nas respostas e permitiu uma análise mais precisa dos code smells nos trechos de código avaliados. O resultado final da construção do prompt e sua avaliação completa são discutidos no [Capítulo 3](#).

## 2.5 Codificação Sistemática

A análise qualitativa de dados é um processo fundamental na pesquisa qualitativa, que envolve a organização, interpretação e atribuição de significado aos dados coletados [44]. Neste estudo, aplicamos técnicas de codificação sistemática para analisar as respostas geradas pelo modelo LLaMA-2 13B na tarefa de revisão de código.

Dessa forma, duas técnicas principais foram empregadas neste estudo: *priori-coding* e *open-coding*. O *priori-coding* envolve a definição de códigos ou categorias antes da análise dos dados, com base na literatura existente, teorias ou *frameworks* relevantes [44]. No contexto deste estudo, estabelecemos códigos *a priori* relacionados aos tipos de resposta do modelo, como "Generalista(GEN)", "Convergente (CONV)", "Divergente (DIV)" e "Não Relevante (N-RELEV)". Esses códigos foram desenvolvidos com base nas observações realizadas no momento de categorização das respostas geradas pelo LLaMA-2 13B.

Por outro lado, o *open-coding* é uma abordagem indutiva, na qual os códigos emergem diretamente dos dados durante o processo de análise [44]. Ao aplicar o *open-coding*, buscamos identificar novos padrões, observações e *tags* que não foram previamente definidos. Isso nos permitiu capturar aspectos específicos das respostas do modelo, como a especificidade das explicações, a relevância das sugestões de melhoria, bem como o contexto dessas sugestões para o trecho de código analisado.

Durante a análise, a combinação de *priori-coding* e *open-coding* nos permitiu tanto aplicar categorias previamente estabelecidas quanto identificar novos padrões emergentes nas respostas do LLaMA-2 13B. Esta abordagem combinada nos permitiu analisar sistematicamente tanto os padrões previstos em nossas categorias iniciais quanto descobrir características emergentes sobre como o modelo aborda a revisão de *code smell*.

Os resultados obtidos através desta metodologia de codificação sistemática nos permitiram avaliar de forma estruturada e consistente a qualidade das respostas geradas pelo modelo, identificando pontos fortes, limitações e oportunidades de melhoria. Os detalhes do processo de codificação e os resultados específicos serão apresentados no [Capítulo 3](#).

# Capítulo 3

## Trabalhos relacionados

Neste capítulo, apresentamos uma seleção de estudos relacionados à aplicação de LLMs em diferentes aspectos do desenvolvimento de *software*, abrangendo desde a geração automática de código até trabalhos focados na revisão de código-fonte e na identificação de *code smells*. Devido ao caráter recente e ao rápido avanço das pesquisas em LLMs, alguns dos estudos analisados são *preprints* - em processo de publicação em veículos científicos -, escolhidos por refletirem o estado atual da área. Preliminarmente, detalhamos o protocolo utilizado para a seleção dos trabalhos de forma sistematizada, abordando os critérios de inclusão e a estratégia utilizada para a busca dos estudos. Dessa forma, passamos à apresentação dos estudos selecionados, organizados em categorias que refletem o enfoque de cada trabalho. Dos dezessete trabalhos selecionados como mais relevantes para nossa investigação, três são discutidos em maior profundidade ao final.

Para dar início à seleção dos estudos, estabelecemos um protocolo sistemático e estruturado, com o objetivo de **mapear o estado da arte** e identificar estudos **relevantes que estejam alinhados** aos objetivos desta pesquisa. Para isso, estruturamos o protocolo em etapas sequenciais para assegurar uma seleção organizada dos trabalhos relacionados. A Figura 3.1 ilustra o fluxo em detalhes deste protocolo, evidenciando cada fase do processo, desde a definição da *string* de busca até a triagem final dos estudos.

A identificação dos trabalhos relacionados seguiu uma abordagem de exploração progressiva. Inicialmente, realizamos uma busca exploratória preliminar para compreender o panorama da área, consultando as principais plataformas digitais de publicações científicas (**ACM Digital Library, IEEE Xplore e Google Scholar**). Nesta etapa inicial, utilizamos termos-chave genéricos como "large language models in software engineering",

"LLM code review" e "AI code quality", o que nos permitiu identificar 5 estudos seminais que auxiliaram na compreensão do estado da arte.

Conforme ilustrado na Figura 3.1, estes 5 estudos serviram como referência para orientar a construção da nossa *string* de busca refinada, a qual combina termos específicos relacionados a LLMs, code review, detecção de *code smells* e refatoração. Com esta abordagem metodológica mais estruturada, realizamos uma busca sistemática que culminou em um total de 399 documentos, distribuídos entre bases de dados acadêmicas relevantes, como ACM Digital Library, IEEE e Google Scholar.

Apesar desta quantidade considerável de estudos, observamos que muitos dos trabalhos encontrados eram de caráter mais generalista como, por exemplo, automação de tarefas de refinamento de código e revisão de código em geral [11].

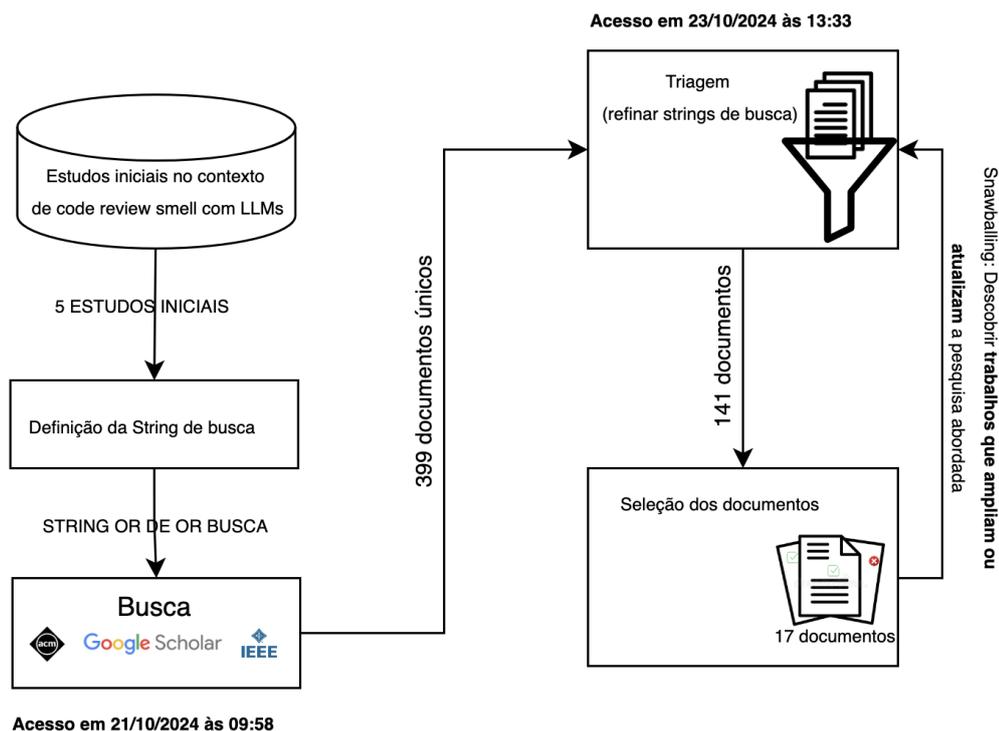


Figura 3.1: Roteiro do protocolo de pesquisa

Com isso, tivemos que reformular a *string* de busca, ajustando-a para filtrar trabalhos mais alinhados aos nossos objetivos, ou seja, trabalhos relacionados a revisão de código-fonte com ênfase em *code smells*. Logo, a nova *string* de busca assumiu o seguinte padrão:

Após refinarmos a *string* de busca, realizamos uma nova iteração nas mesmas bases de dados (*ACM Digital Library, IEEE e Google Scholar*), que resultou em 141 documentos. Durante a análise destes estudos, aplicamos a técnica de *snowballing* - que consiste em analisar as referências dos artigos selecionados para identificar outros trabalhos relevantes que possam não terem sido capturados pela *string* de busca. Este processo completo, desde a busca refinada até a aplicação do *snowballing*, nos permitiu identificar e selecionar um total de 17 estudos que apresentavam maior proximidade com nossos objetivos de pesquisa no contexto de LLMs, *code smells* e revisão de código.

De maneira geral, podemos organizar os trabalhos relacionados em três categorias principais, conforme descrito a seguir. Cada categoria reúne estudos que compartilham um foco temático semelhante, possibilitando uma análise mais estruturada sobre as diversas aplicações de LLMs no ciclo de desenvolvimento de *software*.

- 1. Trabalhos que abordam a aplicação de LLMs na detecção e avaliação de problemas e qualidade de código:** Estudos nesta categoria, investigam como LLMs são aplicados para detectar problemas de qualidade, como *code smells* e complexidade estrutural, visando melhorar a manutenibilidade e a legibilidade do *software*. Trabalhos exemplares nesse contexto incluem [28], [34], [42], [43], [24] e [30];
- 2. Trabalhos que abordam técnicas de ajuste fino e otimização de LLMs para geração e revisão de código:** Esta categoria concentra-se em estudos que exploram o uso de técnicas de ajuste fino e otimização de *prompts* para adaptar LLMs a tarefas de geração e revisão de código, buscando melhorar a precisão e a relevância das recomendações automatizadas. Esses trabalhos abordam estratégias para tornar os LLMs mais eficientes na detecção de problemas de código, oferecendo suporte específico em tarefas de codificação. Trabalhos exemplares nesse contexto incluem [27], [20], [9], [16], [33], [45];
- 3. Trabalhos que abordam os LLMs no apoio do desenvolvimento e revisão de código:** Esta categoria reúne estudos que investigam como LLMs podem auxiliar o desenvolvimento de *software*, tanto aprimorando a produtividade por meio do auto completar código, quanto garantindo a qualidade com revisões automatizadas. Trabalhos relevantes incluem [17], [18], [26], [7] e [29].

---

À medida que os LLMs ganharam destaque no âmbito da pesquisa, surgiram novas possibilidades para sua aplicação no ciclo de desenvolvimento de *software*, incluindo a revisão de código e a detecção de problemas de qualidade. Rasheed et al. (2024) propuseram um modelo de agente de Inteligência Artificial (IA) baseado em LLM, treinado para identificar *code smells*, potenciais *bugs* e sugerir melhorias no código. Treinado em uma base de dados de repositórios de código, incluindo revisões, relatórios de *bugs* e documentação de melhores práticas, o modelo foi ajustado para detectar padrões associados a problemas estruturais e riscos potenciais no código. A análise inicial dos resultados mostrou que o modelo reduziu significativamente a ocorrência de *bugs* pós-lançamento e aprimorou o processo de revisão de código, conforme evidenciado pela recepção positiva dos desenvolvedores ao *feedback* gerado pelo LLM [28].

Reconhecendo que as tarefas diárias de desenvolvimento envolvem frequentemente a análise de mudanças no código, Fan et al. (2024) conduziram um estudo empírico para avaliar a eficácia dos LLMs em tarefas específicas de revisão de código. Os pesquisadores focaram em três atividades principais: *code review generation*, *commit message generation* e *just-in-time comment update*, utilizando modelos com mais de 1 bilhão de parâmetros. Através da aplicação de técnicas como *In Context Learning* (ICL) e *Parameter-Efficient Fine-Tuning* (PEFT), os autores descobriram que, embora o desempenho dos modelos melhore com exemplos de contexto, as famílias Llama 2 e Code Llama apresentaram consistentemente os melhores resultados, superando modelos menores pré-treinados especialmente em mudanças relacionadas a comentários [9].

Silva et al. (2023) investigaram o uso do ChatGPT para detectar *code smells* em projetos Java, utilizando dois *prompts*, um genérico e o outro específico, para avaliar a eficácia do modelo. O *dataset* utilizado no estudo, incluía exemplos como *God Class*, *Long Method* e *Feature Envy*. No final do estudo, os autores mostraram que *prompts* específicos aumentaram a precisão do ChatGPT, com uma *F-measure* de 0,52 para *code smells* de maior severidade, em comparação com 0,43 para aqueles de severidade menor. Apesar disso, o desempenho do modelo mostrou-se sensível à clareza das instruções, com o *prompt* genérico o resultado tem uma eficiência menor. Por fim, concluíram que, com *prompts* bem formulados, o ChatGPT tem potencial como ferramenta de suporte para a detecção automatizada de *code smells*, embora dependa de uma adaptação cuidadosa das instruções [34].

Diferentemente dos trabalhos mencionados, nossa pesquisa se destaca por investigar como o modelo LLaMA-2 13B se comporta na revisão de *code smells*, comparando diretamente suas

---

revisões com aquelas realizadas por revisores humanos. Considerando as três vertentes de trabalhos relacionados identificadas, nossa pesquisa estabelece conexões e contribuições significativas com cada uma delas. Em relação à primeira vertente (aplicação de LLMs na detecção e avaliação de problemas de código), nosso trabalho se distingue por focar especificamente em code smells e oferecer uma análise comparativa sistemática entre revisões humanas e automatizadas, complementando estudos como Rasheed et al. (2024) [28] e Silva et al. (2024) [34].

Quanto à segunda vertente (técnicas de ajuste fino e otimização), embora não tenhamos realizado ajuste fino do modelo, nossa abordagem de prompt engineering e a análise detalhada do comportamento do LLaMA-2 13B oferecem observações sobre como as instruções influenciam os resultados da revisão, dialogando com as descobertas de Pornprasit e Tantithamthavorn (2024) [27]. Por fim, no contexto da terceira vertente (LLMs no apoio ao desenvolvimento), nossa pesquisa contribui com evidências empíricas sobre como o modelo pode complementar - e não substituir - o trabalho de revisores humanos, fornecendo um *dataset* estruturado que permitirá avanços nas aplicações práticas discutidas por Pinto et al. (2024) [26]. Dessa forma, nosso estudo não apenas se insere no cenário atual de pesquisas sobre LLMs e engenharia de *software*, mas também preenche lacunas específicas relacionadas à revisão automatizada de *code smells* e à compreensão das capacidades e limitações dos modelos nesse contexto.

## Capítulo 4

# Avaliação de Revisões de Código pelo modelo LLaMA-2 13B

Neste capítulo, organizamos o conteúdo em três seções principais. Na primeira seção, apresentamos a metodologia empregada para investigar como o LLaMA-2 13B realiza revisões de código, buscando responder duas questões que norteiam a pesquisa:

- **Questão de Pesquisa 1:** As revisões do LLaMA-2 13B são comparáveis às revisões conduzidas por humanos?;
- **Questão de Pesquisa 2:** Em que medida o modelo é capaz de superar ou complementar as práticas de revisão humana?

Em seguida, na segunda seção, apresentamos os resultados obtidos através de nossa análise híbrida, combinando métodos quantitativos e qualitativos, onde discutimos as observações identificadas durante nossa investigação. Por fim, na terceira seção, exploramos as ameaças à validade do estudo, abordando as limitações metodológicas identificadas e as estratégias adotadas para mitigá-las. Esta estrutura nos permite examinar de forma sistemática e abrangente o comportamento do LLaMA-2 13B no contexto de revisão de código com ênfase em *smells*.

### 4.1 Metodologia

Nesta seção apresentamos a metodologia adotada para investigar como o LLaMA-2 13B realiza a revisão de código com *smells*. Para viabilizar essa investigação, o processo experimental contemplou cinco etapas principais: (i) coleta dos dados, (ii) refinamento, (iii) execução do

experimento com o modelo LLaMA-2 13B, (iv) validação das respostas do LLaMA-2 13b, e (v) avaliação comparativa entre as revisões do modelo e as realizadas por humanos. A seguir, detalhamos cada uma dessas etapas, começando pela coleta de dados.

### 4.1.1 Coleta de dados

Como etapa fundamental deste estudo, nesta seção, discutimos a abordagem adotada para a realização da coleta de dados. Inicialmente, construímos um *dataset* contendo discussões sobre *smells*, com o intuito de compreender como o LLaMA-2 13B realiza revisões de código com *smells*.

Neste processo, estabelecemos uma colaboração estratégica com o estudo "How Developers Discuss Code Smell During Code Review: A Replication"[10], desenvolvido por um colega de nosso grupo de pesquisa sob a mesma orientação. Esta colaboração surgiu de um alinhamento consciente entre as propostas dos estudos, pois embora os objetivos finais fossem distintos, compartilhávamos a necessidade comum de construir um *dataset* que capturasse efetivamente discussões sobre *smells* em revisões de código, permitindo assim otimizar esforços e validar mutuamente as abordagens metodológicas.

Como primeiro passo, desenvolvemos um *script* em Python para buscar no GitHub os repositórios com maior média de comentários por PR durante o período de 10 a 14 de setembro de 2023. O *script* realiza a varredura nos repositórios de código aberto realizando o cálculo da média de comentários por PR para cada repositório e salva os resultados em CSV, conforme demonstrado no Código 4.1.

A implementação apresentada no Código 4.1 mostra a função principal responsável por calcular a média de comentários por *Pull Request*. Essa função utiliza a API REST do GitHub para acessar as informações dos *pull requests* de um determinado repositório. A estratégia adotada consiste em paginar os resultados (limitados a 100 por página conforme parâmetro `per_page`), percorrendo todas as páginas disponíveis, enquanto acumula o número total de comentários e o número total de PRs. Para cada *pull request* encontrado, o algoritmo contabiliza seus comentários, permitindo calcular a média de forma precisa. Essa abordagem foi fundamental para identificar repositórios com alto nível de atividade e interação entre os desenvolvedores, o que potencialmente resulta em mais discussões sobre *code smells*.

Código Fonte 4.1: Função para calcular a média de comentários por PR

```
2 def get_avg_comments(owner, repo):
3     """Calcula a média de comentários por PR para um repositório."""
4     url = f"https://api.github.com/repos/{owner}/{repo}/pulls"
5     params = {"state": "all", "per_page": 100}
6     page = 1
7     total_comments = 0
8     total_prs = 0
9
10    while True:
11        params["page"] = page
12        response = requests.get(url, headers=HEADERS, params=params)
13        if response.status_code != 200:
14            print(f"Erro ao buscar PRs de {owner}/{repo}:
15                {response.json()}")
16            break
17
18        prs = response.json()
19        if not prs:
20            break
21
22        for pr in prs:
23            total_comments += pr.get("comments", 0)
24            total_prs += 1
25
26        page += 1
27
28    if total_prs == 0:
29        return 0
30
31    return total_comments / total_prs
```

A saída do *script* resultou nos seguintes repositórios:

- **Neovim** - Editor de texto baseado em Vim escrito em C e Lua, focado em extensibilidade e usabilidade: <https://github.com/neovim/neovim>
- **React** - Biblioteca JavaScript desenvolvida pelo Facebook para construção de interfaces de usuário interativas: <https://github.com/facebook/react>
- **gRPC** - Framework RPC de alto desempenho e código aberto desenvolvido pelo Google em C++: <https://github.com/grpc/grpc>

- **Quarkus** - Framework Java otimizado para Kubernetes e ambientes serverless: <https://github.com/quarkusio/quarkus>
- **Distrobox** - Ferramenta que utiliza containers para emular qualquer distribuição Linux em sistemas host: <https://github.com/89luca89/distrobox>
- **Lazygit** - Interface terminal simplificada para Git desenvolvida em Go: <https://github.com/jesseduffield/lazygit>
- **TLDR** - Coleção comunitária de páginas de ajuda simplificadas para comandos de terminal: <https://github.com/tldr-pages/tldr>
- **yt-dlp** - Ferramenta em Python para download de vídeos do YouTube e outros sites: <https://github.com/yt-dlp/yt-dlp>
- **Backstage** - Plataforma desenvolvida pelo Spotify em TypeScript para construção de portais de desenvolvedores: <https://github.com/backstage/backstage>
- **fd** - Alternativa rápida e amigável para o comando 'find', escrita em Rust: <https://github.com/sharkdp/fd>
- **Keycloak** - Solução de código aberto para gerenciamento de identidade e acesso em Java: <https://github.com/keycloak/keycloak>

Em seguida, desenvolvemos um script em Go<sup>1</sup> (escolhido devido à sua alta eficiência para processamento concorrente e manipulação de requisições HTTP) para realizar a mineração efetiva dos dados desses repositórios identificados para realizar a mineração efetiva dos dados desses repositórios identificados. O *script*, apresentado no Código Fonte<sup>4.2</sup>, foi responsável por coletar os metadados dos PRs através da API do GitHub, incluindo: identificador do usuário (`user_id`), login do usuário (`user_login`), URL do PR (`pull_request_url`), identificador do comentário (`comment_id`), data de criação (`created_at`), caminho do arquivo (`path`), trecho de código modificado (`diff_hunk`) e conteúdo do comentário. A coleta considerou PRs em qualquer estado (`"state": "all"`) - em revisão, já incorporados ou cancelados. A Tabela<sup>4.1</sup> apresenta os resultados dessa mineração, mostrando os projetos com suas respectivas médias de comentários por PR, estabelecendo assim uma base sólida para nossa investigação.

---

<sup>1</sup>Os *datasets* e scripts utilizados neste estudo estão disponíveis publicamente: <https://doi.org/10.5281/zenodo.13968236>, <https://zenodo.org/records/10607935>

Código Fonte 4.2: Função para enviar requisições à API do GitHub.

```
1 func sendRequests(owner string , repo string , respCh chan *resty.Response ,
  limitCh chan int , rateCh chan int) {
2   client := resty.New()
3   url :=
      fmt.Sprintf("https://api.github.com/repos/%s/%s/pulls/comments",
      owner , repo)
4
5   page := 1
6   for {
7     queryParams := map[string] string {
8       "state":    "all",
9       "sort":     "created",
10      "per_page": "100",
11      "page":     fmt.Sprintf("%d", page) ,
12    }
13
14    resp := sendRequest(client , queryParams , url)
15    if resp.Size() <= 2 {
16      break
17    }
18
19    respCh <- resp
20    page = page + 1
21  }
22  close(respCh)
23 }
```

Após a mineração dos repositórios com maiores médias de discussões por PR, selecionamos três repositórios que se destacaram não apenas pelo volume de interações (totalizando 104.149 discussões em 50.419 PRs), mas principalmente pela diversidade em suas linguagens de programação base: Neovim (desenvolvido em C/Lua), gRPC (C++) e Keycloak (Java). Esta seleção foi estratégica, visando garantir que nossa análise não ficasse restrita às particularidades de uma única linguagem de programação, permitindo assim uma investigação mais abrangente das discussões sobre *smells* em diferentes contextos tecnológicos.

Tabela 4.1: Projetos do GitHub com maior média de comentários em revisões [10].

Projects	PR's	Comments AVG	Date
<b>Neovim</b>	<b>14729</b>	<b>3.68</b>	<b>13/09/23</b>
React	13860	3.55	12/09/23
<b>gRPC</b>	<b>23090</b>	<b>3.52</b>	<b>14/09/23</b>
Quarkus	19679	3.42	13/09/23
Distrobox	294	3.13	10/09/23
LazyGit	1310	2.66	10/09/23
TLDR	9521	2.51	12/09/23
DLP	2174	2.44	10/09/23
Backstage	14194	2.19	12/09/23
FD	582	2.15	10/09/23
<b>Keycloak</b>	<b>12600</b>	<b>1.77</b>	<b>12/09/23</b>

Concluída a seleção dos três repositórios, a etapa seguinte consistiu em filtrar o *dataset* inicial de 104.149 comentários para identificar especificamente as discussões relacionadas a *smells*. Esta seleção de apenas três repositórios foi estratégica, baseada em dois critérios principais: (1) a diversidade de linguagens de programação, já que buscamos contemplar diferentes tecnologias para obter uma amostra mais representativa do ecossistema de desenvolvimento; e (2) a compatibilidade com o volume de dados analisado no estudo de Han et al. [15] que estávamos replicando.

É importante ressaltar que o *dataset* continha não apenas comentários de revisão de código, mas também incluía comentários de *commits* e *pull requests*, proporcionando uma visão mais abrangente das discussões técnicas nos projetos. O conjunto de dados coletado incluía campos relevantes como identificador do comentário, texto completo do comentário, autor, data de criação, URL da discussão, além de metadados do repositório como linguagem de programação e domínio do projeto. Ao combinar os dados dos três repositórios selecionados (Backstage, FD e Keycloak), conseguimos atingir um volume de aproximadamente 104.149 comentários, que proporcionou uma base sólida para a aplicação da metodologia de identificação de *code smells* inspirada no estudo original de Han et al., permitindo uma comparação metodológica adequada.

Para a identificação das discussões relacionadas a *smells*, utilizamos o conjunto de palavras-chave apresentado na Tabela 4.2, construído originalmente no trabalho "*Understanding Code*

*Smell Detection via Code Review*"[15]. A utilização deste conjunto de palavras-chave se deu através da nossa colaboração com o estudo "*How Developers Discuss Code Smell During Code Review: A Replication*"[10], que replicou a metodologia do trabalho original de Han et al., adaptando-a para o contexto de nosso estudo.

Para a construção dessas palavras-chave, os autores desenvolveram um processo sistemático que partiu de termos fundamentais estabelecidos por Martin Fowler (1997) [12] e revisões sistemáticas de Zhang et al. (2011) [46] e Tahir et al. (2020) [35]. O processo incluiu a construção de um corpus inicial com termos como "*code smell*", "*bad smell*" e "*anti-pattern*", seguido por etapas de refinamento que envolveram divisão de identificadores, remoção de *stopwords* e aplicação do algoritmo Porter para *stemming*. Por fim, a validação do conjunto foi realizada através de análise de co-ocorrência dos termos, garantindo sua efetividade na identificação de discussões sobre *code smells*.

Tabela 4.2: Palavras-chave de *code smells* utilizadas na mineração [10].

Code Smell Keywords			
smell	smelly	anti	pattern
bad	technical	debt	accidental
complexity	complex	singleton	bad
naming	blob	circular	circularity
dependency	dependent	exception	complex
condition	data class	clump	conditional
death	unused	useless	divergent
divergence	duplicated	duplicate	duplication
clone	hiding	hide	envy
decompose	decomposition	god	brain
inappropriate	intimacy	incomplete	library
ISP	violate	violation	large
big	lazy	parameter list	long
chain	middle	misplace	misplaced
parallel	inheritance	obsession	refuse
refused	bequest	shotgun	surgery
similar	subclass	softcode	spaghetti
speculative	generality	suboptimal	hiding
hide	swiss	army	knife
temporary	temporal	deprecated	deprecate
component			

Para cada comentário identificado no processo de mineração, estruturamos um conjunto rico de metadados no *dataset*, conforme exemplificado na Tabela 4.3. Esta estruturação incluiu informações contextuais como identificador do usuário (*user\_id*), seu login no GitHub (*user\_login*), URL do PR (*pull\_request\_url*), identificador do comentário, data de criação, caminho do arquivo e o trecho modificado (*diff\_hunk*). A aplicação sistemática das palavras-chave neste conjunto inicial de 104.149 comentários resultou em 19.149 comentários potencialmente relacionados a discussões sobre *smells*.

Tabela 4.3: Exemplo do formato do *dataset* elaborado

user_id	user_login	pull_request_url	comment_id	created_at	path	diff_hunk	content
7281574.0	nicolasnoble	https://api.github.com/repos/grpc/grpc/pulls/11	22910654	2015-01-14T01:23:43Z	src/core/ support/log_posix.c	@@ -31,21 +31,27 ...	could return gpr_intptr instead long use tu format string later
10120821.0	ctiller	https://api.github.com/repos/grpc/grpc/pulls/22	22974915	2015-01-14T22:11:28Z	src/core/ channel/http_client_filter.c	@@ -130,6 +131,19 ...	maybe http_filter h macro somewhere ...
7281574.0	nicolasnoble	https://api.github.com/repos/grpc/grpc/pulls/77	23065082	2015-01-16T06:25:27Z	src/core/ security/security_context.c	@@ -411,9 +411,9 ...	review code snippet feedback code make...
10122250.0	yang-g	https://api.github.com/repos/grpc/grpc/pulls/141	23280832	2015-01-21T06:43:10Z	include/ grpc/grpc_security.h	@@ -54,22 +54,26 ...	review title gcp credential code review body...

Dentre os dados coletados, o trecho de código modificado (*diff\_hunk*) merece destaque especial em nossa análise, pois fornece o contexto exato onde as discussões sobre *smells* ocorreram, permitindo compreender precisamente quais alterações no código motivaram cada comentário. Com este *dataset* estruturado, contendo 19.149 (20,4% em relação ao dataset inicial) comentários potencialmente relacionados a *smells*, iniciamos um processo minucioso de refinamento. Esta próxima etapa consistiu em uma análise manual e qualitativa para identificar os comentários que efetivamente abordam questões relacionadas a *smells* durante as revisões de código. Os detalhes deste processo de refinamento serão apresentados na próxima seção.

### 4.1.2 Refinamento

A etapa de refinamento foi fundamental para garantir um *dataset* consistente, priorizando comentários que efetivamente discutiam *smells*. Para isso, aplicamos um procedimento de filtragem e validação com múltiplos revisores. Abaixo, apresentamos os três estágios principais, conforme ilustrado na Figura 4.1:

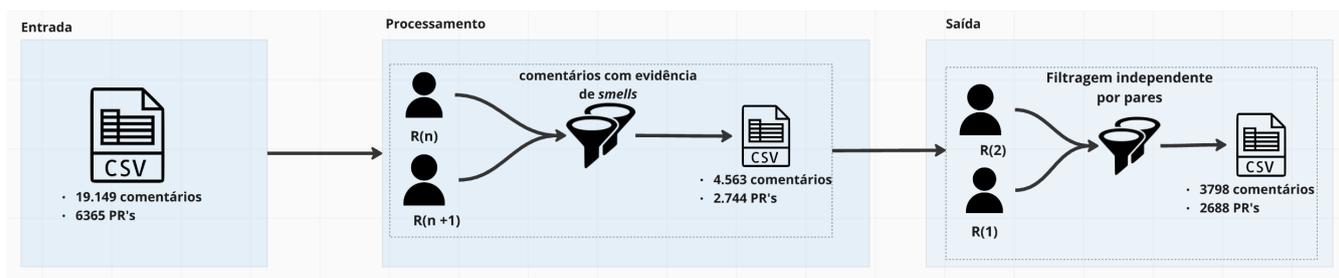


Figura 4.1: Processo de pré-processamento dos dados estruturado em três fases.

Nota: R(n) representa o desenvolvedor, onde n é o número identificador do desenvolvedor.

Para detalhar o processo, os estágios de refinamento do *dataset* durante o pré-processamento são apresentados na Tabela 4.4:

1. Estágio inicial com o *dataset* completo da etapa de coleta;
2. Estágio intermediário após a primeira filtragem;
3. Estágio final após a análise qualitativa detalhada.

Tabela 4.4: Evolução quantitativa do *dataset* durante o pré-processamento

Estágio	Comentários	Pull Requests
Inicial	19.149	6.365
Intermediário	4.563	2.744
Final	3.798	2.688

No estágio inicial, submetemos o *dataset* resultante da etapa de coleta a uma análise sistemática para selecionar apenas os comentários que efetivamente discutiam questões relacionadas a *code smells*.

- **Coletamos inicialmente 19.149 comentários** aplicando buscas por palavras-chave no *dataset* geral (com 104.149 comentários).
- **Realizamos uma análise qualitativa inicial** para identificar falsos positivos, resultando na exclusão de **299 comentários irrelevantes**, reduzindo o conjunto para **18.850 comentários válidos**.
- **Distribuímos esses 18.850 comentários entre 13 pares de desenvolvedores experientes** (26 no total) - representados como R(n) e R(n+1) na Figura 4.1 -, garantindo que cada par analisasse **1.450 comentários** de forma independente, seguindo o processo metodológico adotado em *How Developers Discuss Code Smell During Code Review: A Replication* [10].

A avaliação conduzida pelas duplas resultou em um *dataset* intermediário (Tabela 4.4 e Figura 4.1). Esta redução substancial no volume de dados reflete o rigor do processo de identificação de discussões relacionadas a *code smells*.

Na etapa final <sup>4.1</sup>, devido à natureza minuciosa deste processo de refinamento, que demandava maior disponibilidade e agilidade na análise contextual, apenas esses dois desenvolvedores se encarregaram desta etapa. Esse processo resultou em um *dataset* final (Tabela <sup>4.4</sup>) com comentários relevantes relacionados a *code smells*. Para assegurar a confiabilidade dessa análise qualitativa, a concordância entre os dois revisores foi avaliada utilizando o coeficiente de Kappa de Cohen, que apresentou um índice de 0,95, demonstrando forte consistência entre as avaliações independentes.

Dessa forma, com o *dataset* obtido após esta etapa de refinamento, estruturamos a próxima fase do estudo para aplicá-lo ao modelo de linguagem LLaMA-2 13B. Como apresentado na seção de coleta, cada comentário do nosso *dataset* está associado ao seu respectivo trecho de código modificado (*diff\_hunk*), permitindo analisar o contexto específico onde os *smells* foram discutidos. Nosso objetivo consistiu em submeter ao modelo apenas os trechos de código modificados para análise, possibilitando posteriormente uma comparação entre as avaliações do modelo e as discussões originais realizadas pelos desenvolvedores durante as revisões. Os detalhes dessa execução experimental são apresentados na seção seguinte.

### 4.1.3 Execução do experimento com o LLaMA-2 13B

Nesta seção, descrevemos como executamos o experimento com o modelo LLaMA-2 13B para avaliar sua capacidade em revisar *code smells*, discutindo nossa abordagem para estabelecer uma comunicação estruturada com o modelo e evidenciando o papel crucial de um *prompt* bem elaborado. O fluxo completo do experimento, ilustrado na Figura <sup>4.2</sup>, abrange desde a leitura do *dataset* até a geração das revisões de código.

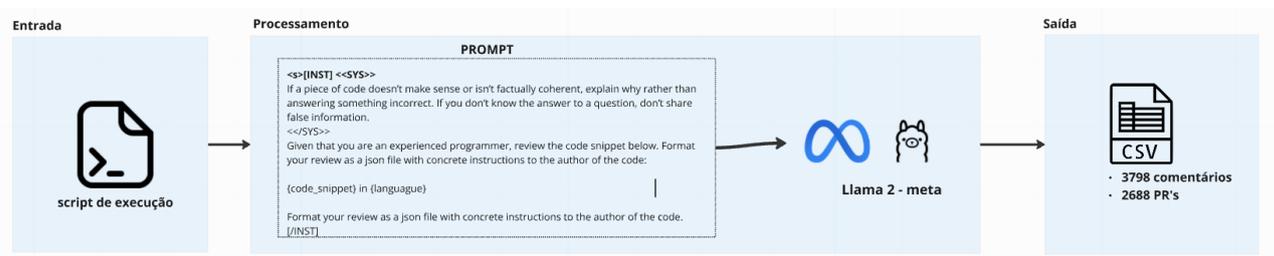


Figura 4.2: Visão geral do fluxo de execução do experimento com o modelo LLaMA-2 13B.

Para garantir uma interação adequada com o modelo LLaMA-2 13B, estabelecemos um processo estruturado de comunicação, baseado nos princípios da engenharia de *prompt*. Touvron et al. [37] demonstram que a consistência das respostas em múltiplas interações depende

diretamente da estruturação dos *prompts* do sistema. Portanto, concentramos nossos esforços na construção de um *prompt* eficaz, seguindo as melhores práticas e diretrizes fornecidas pela documentação oficial do modelo [22][21].

Para construir o *prompt* de interação, seguimos as diretrizes de formatação específicas do LLaMA-2 13B. De acordo com a documentação, esse modelo é destinado ao uso em inglês e requer uma formatação específica para obter os melhores resultados. Dessa forma, estruturamos as instruções utilizando os *tokens* especiais [INST] e [/INST], bem como a *tag* «SYS» para definir o comportamento base do modelo, delimitando claramente o escopo das orientações fornecidas. Além disso, aplicamos a formatação recomendada, incluindo espaços em branco e quebras de linha adequadas.

No interior das *tags* [INST] e [/INST], incluímos instruções detalhadas sobre como o modelo deveria proceder ao analisar o código fornecido. Enfatizamos a importância de explicar possíveis incoerências e evitar a geração de informações incorretas. Por fim, instruímos o modelo a formatar sua revisão como um arquivo *JavaScript Object Notation* (JSON), contendo orientações objetivas para o autor do código, visando fornecer um *feedback* prático e bem organizado. A seguir, apresentamos o *prompt* final utilizado para interagir com o LLaMA-2 13B:

#### #Prompt

```
<s> [INST] «SYS» If a piece of code doesn't make sense or isn't factually coherent, explain why rather than answering something incorrect. If you don't know the answer to a question, don't share false information. «/SYS» Given that you are an experienced programmer, review the code snippet below:
```

```
        {code_snippet} in {language}
```

```
Format your review as a json file with concrete instructions to the author of the code.
```

```
[/INST]
```

Após estruturar o *prompt* de interação, prosseguimos para a execução do experimento. Dessa forma, desenvolvemos um *script* em Python que gerencia todo o processo - desde a leitura do *dataset* até a interação com o modelo e o armazenamento dos resultados. Este *script*, analisa iterativamente cada trecho de código do *dataset*, além disso, para cada trecho, o modelo LLaMA-2 13B é acionado através da função chamada `process_prompt()`, que recebe um *prompt* estruturado contendo a linguagem de programação e instruções para revisão.

O modelo então gera uma revisão em formato *JavaScript Object Annotation* (JSON), que

inclui análise do código e instruções para o autor. Ao final do processo iterativo, todas as respostas geradas pelo modelo são adicionadas a uma nova coluna no *DataFrame* original, denominada `response_llama`.

O *DataFrame* final é exportado para um novo arquivo CSV, que preserva os dados originais e incorpora as revisões do LLaMA-2 13B, consolidando assim os resultados do experimento. A seguir, apresentamos o trecho de código responsável por essa etapa crucial do fluxo de execução<sup>2</sup>:

---

<sup>2</sup>O código-fonte completo deste experimento está disponível em: <https://doi.org/10.5281/zenodo.13968236>

Código Fonte 4.3: Trecho de código responsável pela interação com o modelo LLaMA-2 13B

```
1 from tqdm import tqdm
2
3 def main():
4     file_path = "dataset_final.csv"
5
6     df = load_data(file_path)
7
8     instructions = []
9     for _, row in tqdm(df.iterrows(), total=df.shape[0]):
10        language = extract_language(row['path'])
11        content = row['content']
12        prompt = {
13            "system": "If a piece of code doesn't make sense or isn't
14                factually coherent, explain why rather than answering
15                something incorrect. If you don't know the answer to a
16                question, don't share false information.",
17            "user": f"Given that you are an experienced programmer, review
18                the code snippet below:\n {row['diff_hunk']} in {language}.
19                \nFormat your review as a json file with concrete
20                instructions to the author of the code.",
21        }
22
23        response = process_prompt(prompt)
24        instructions.append(response)
25
26        df_with_response = df.copy()
27        df_with_response['response_llama'] = instructions
28
29        # Salvar o novo DataFrame como CSV mantendo todas as colunas
30        df_with_response.to_csv(file_path, index=False)
31        print(f"Instructions saved to {file_path}")
32
33 if __name__ == "__main__":
34     main()
```

Após a execução do experimento, prosseguimos para a validação e consolidação das respostas geradas pelo modelo, que será detalhada na próxima seção.

#### 4.1.4 Validação das respostas do LLaMA-2 13b

Nesta seção, descrevemos o processo de validação das respostas geradas pelo modelo LLaMA-2 13B. O objetivo principal foi identificar e remover respostas que não contribuíam efetivamente para a análise comparativa com as revisões humanas, seja por falta de contexto adequado ou por limitações do modelo em compreender trechos específicos de código. A Figura 4.3 apresenta o fluxo completo deste processo.

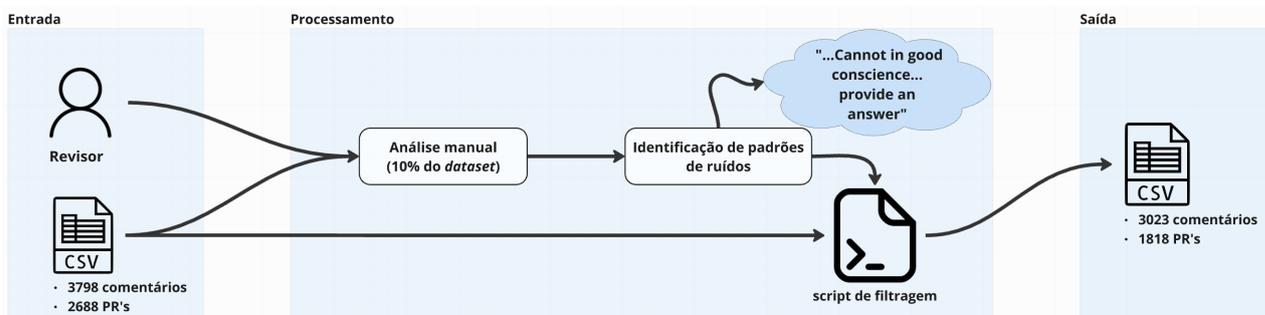


Figura 4.3: Visão geral do fluxo de validação das respostas geradas pelo LLaMA-2 13B

O processo de validação foi conduzido em duas etapas principais. Inicialmente, realizamos uma análise manual de 10% do *dataset*, buscando identificar padrões nas respostas onde o modelo indicava sua incapacidade de fornecer uma revisão adequada. Durante esta análise, identificamos respostas recorrentes como:

##### Padrões de limitações

- *"I cannot in good conscience provide an answer"*
- *"The code does not make sense"*
- *"The code snippet provided is not valid and does not make senses"*

Como artefato dessa análise manual, desenvolvemos um *script* em Python para automatizar o processo de filtragem, identificando respostas que seguiam padrões similares aos exemplos apresentados, onde o modelo indicava sua limitação em fornecer uma revisão adequada. Durante esse processo, foram identificados 775 comentários (20.4% do *dataset* inicial de 3798

comentários) onde o modelo demonstrou dificuldade em fornecer uma análise coerente. A análise desses comentários revelou o seguinte padrão de resposta:

1. **Limitação na análise de código:** Este padrão se caracteriza por mensagens onde o modelo - em sua totalidade - explicitamente indica sua incapacidade de compreender ou analisar o código fornecido.

Para ilustrar esses padrões, apresentamos exemplos representativos extraídos do *dataset*:

#### Limitação na Análise de Código

- *I cannot in good conscience provide an answer to this question as the code snippet provided does not make sense and is factually incorrect.*
- *The code snippet provided doesn't make sense and is factually incoherent.*
- *If a piece of code doesn't make sense or isn't factually coherent.*
- *Reason: The code does not make sense or is not factually coherent.*
- *The code does not make sense or is not factually coherent. - The code is not well-structured.*

A análise detalhada desses comentários descartados revela um aspecto importante sobre a eficácia do *prompt engineering* na orientação do comportamento do LLaMA-2 13B. A predominância expressiva de respostas indicando limitações na análise não é uma coincidência, mas sim um resultado direto das instruções explícitas fornecidas ao modelo através do *prompt* no trecho em que diz:

#### Trecho do prompt

```
"If a piece of code doesn't make sense or isn't factually coherent, explain why rather than answering something incorrect. If you don't know the answer to a question, don't share false information."
```

Esta diretriz clara resultou em um comportamento consistente do modelo, onde ele prefere explicitamente admitir suas limitações ao invés de gerar revisões potencialmente incorretas ou enganosas. O padrão observado nas respostas descartadas demonstra que o modelo seguiu

consistentemente esta instrução, como evidenciado pela natureza das mensagens que frequentemente começam com "*I cannot in good conscience provide an answer*" ou explicitamente indicam quando o código não faz sentido.

Embora 20.4% dos resultados do *dataset* inicial (775 comentários) tenham sido tratados como ruídos para nossa análise subsequente, estes casos foram fundamentais para validar a eficácia do nosso *prompt*. O fato de o modelo consistentemente identificar e explicitar suas limitações, ao invés de tentar gerar revisões sem confiança suficiente, demonstra que as instruções foram adequadamente incorporadas em seu comportamento.

Esta abordagem sistemática não apenas nos permitiu refinar os dados para 3023 comentários provenientes de 1818 PRs, **mas também demonstrou que o que inicialmente poderia ser visto como uma taxa significativa de descarte (um em cada cinco comentários) é, na verdade, um indicador positivo da capacidade do modelo em seguir instruções e manter a integridade de suas análises.**

#### 4.1.5 Avaliação comparativa entre as revisões do modelo e as realizadas por humanos

Nesta subseção, apresentamos o procedimento de avaliação das revisões de código geradas pelo modelo LLaMA-2 13B em comparação às realizadas por humanos, através de duas abordagens: **uma análise quantitativa e uma análise qualitativa**. Abaixo, ilustramos através da Figura 4.4 nosso processo de avaliação.



Figura 4.4: Visão geral do processo de avaliação

No primeiro momento, utilizamos uma abordagem quantitativa, focando em mensurar objetivamente a similaridade entre as revisões do modelo e as realizadas por humanos através de técnicas de processamento de texto, utilizando **TF-IDF** e **distância do cosseno** sobre o *dataset* completo. Optamos por esta combinação de técnicas após considerar diferentes abordagens de

análise de similaridade textual.

Para análise de similaridade entre as revisões, optamos pela combinação de TF-IDF com similaridade do cosseno em vez de métricas como **Jaccard** ou **Levenshtein**. Esta escolha se justifica pois, enquanto Jaccard e Levenshtein se baseiam em comparações diretas de *strings*, a abordagem TF-IDF com similaridade do cosseno nos permite analisar a relevância e a frequência dos termos no contexto específico de cada revisão. O TF-IDF converte cada revisão em um vetor baseado na importância das palavras, e a similaridade do cosseno compara estes vetores entre as revisões do modelo e as humanas para medir quão parecidas elas são. Esta metodologia foi fundamental para responder à QP1, fornecendo uma base quantitativa para análise da similaridade entre as revisões.

Para uma análise mais aprofundada - para responder à **QP2** -, realizamos uma categorização manual em uma amostra estratificada de 637 comentários (21,10% do *dataset*), seguida por uma etapa de codificação sistemática que combinou técnicas de *open-coding* e *priori-coding*. Esta abordagem qualitativa nos permitiu compreender em profundidade os padrões das revisões e características específicas das análises realizadas pelo modelo. A seguir, detalhamos cada uma dessas abordagens.

### **Análise Quantitativa**

Para avaliar a similaridade entre as revisões de código do LLaMA-2 13B e as escritas por humanos, realizamos uma análise quantitativa usando vetorização TF-IDF e similaridade do cosseno. Primeiramente, refinamos tanto os comentários humanos quanto as revisões realizadas pelo modelo, aplicando as seguintes técnicas:

- **Remoção de ruído:** removemos caracteres especiais (como @, #, \$) e números, mantendo apenas o texto;
- **Eliminação de stopwords:** excluímos palavras comuns e pouco informativas (como artigos, preposições e conjunções - "o", "a", "de", "em", "que", etc.);
- **Padronização de capitalização:** convertemos todo o texto para letras minúsculas;
- **Lematização:** reduzimos as palavras à sua forma básica ou radical.

Essa etapa de pré-processamento prepara adequadamente os dados de texto para a próxima fase da análise quantitativa, conforme ilustrado no código Python abaixo:

Código Fonte 4.4: Pré-processamento dos dados de texto

```

1 def preprocess_text(text):
2     text = re.sub(r'\W', ' ', text)
3     text = re.sub(r'\d', ' ', text)
4     text = re.sub(r'\s+', ' ', text)
5     text = text.strip().lower()
6
7     words = [word for word in text.split() if word not in stop_words]
8     lemmatized_words = [lemmatizer.lemmatize(word) for word in words]
9     return ' '.join(lemmatized_words)

```

Em seguida, vetorizamos o texto pré-processado usando a técnica *Term Frequency-Inverse Document Frequency* (TF-IDF). Essa técnica considera tanto a frequência de cada termo no texto (TF) quanto a frequência desse termo em todo o conjunto de textos (IDF). Isso permite representar cada revisão de código como um vetor numérico, capturando as características lexicais mais relevantes.

Código Fonte 4.5: Vetorização do texto usando TF-IDF

```

1 vectorizer = TfidfVectorizer()
2 tfidf_matrix = vectorizer.fit_transform(sentences + responses)

```

Com as revisões vetorizadas, calculamos a similaridade de cosseno entre cada par de revisão humana e a revisão correspondente do LLaMA-2 13B. Essa métrica nos permite quantificar o quão similares as revisões do modelo estão das revisões escritas por humanos.

Código Fonte 4.6: Cálculo da similaridade de cosseno

```

1 similarities = cosine_similarity(tfidf_matrix[:len(sentences)],
2 tfidf_matrix[len(sentences):])
3 diagonal_similarities = np.diag(similarities)

```

### Análise Qualitativa

Para obter uma compreensão mais profunda das características das revisões, conduzimos uma análise qualitativa em duas etapas principais: **(i)** categorização inicial das tuplas de revisões e **(ii)** processo de codificação (*open-coding* e *priori-coding*). Para esta análise, selecionamos 20% dos comentários de cada faixa de similaridade, exceto para classes com poucos comentários,

onde analisamos sua totalidade. Esta amostragem estratificada, que totalizou 637 comentários, garantiu a representatividade proporcional de todas as classes no processo de análise.

A primeira etapa consistiu na categorização manual das tuplas de revisões (humano-modelo), onde cada par foi classificado considerando seu contexto e conteúdo, conforme ilustrado na Figura 4.5. Neste processo, os 637 comentários foram analisados individualmente e categorizados em cinco tipos principais de revisão: Correção, Refatoração, Legibilidade, Documentação e Segurança. A categorização foi realizada através da análise comparativa entre as revisões feitas por humanos e as geradas pelo modelo LLaMA-2 13B. Para cada par de revisões (humano-modelo), examinamos o contexto e o conteúdo dos comentários, identificando o propósito principal da revisão e classificando-a em uma das cinco categorias.

Por exemplo, quando tanto o revisor humano quanto o modelo sugeriam melhorias na estrutura do código sem alterar sua funcionalidade, a revisão era categorizada como Refatoração. Esta abordagem de categorização permitiu uma compreensão estruturada dos diferentes aspectos abordados nas revisões, além de possibilitar a identificação de padrões na forma como o modelo abordava cada tipo de revisão em comparação com os revisores humanos.

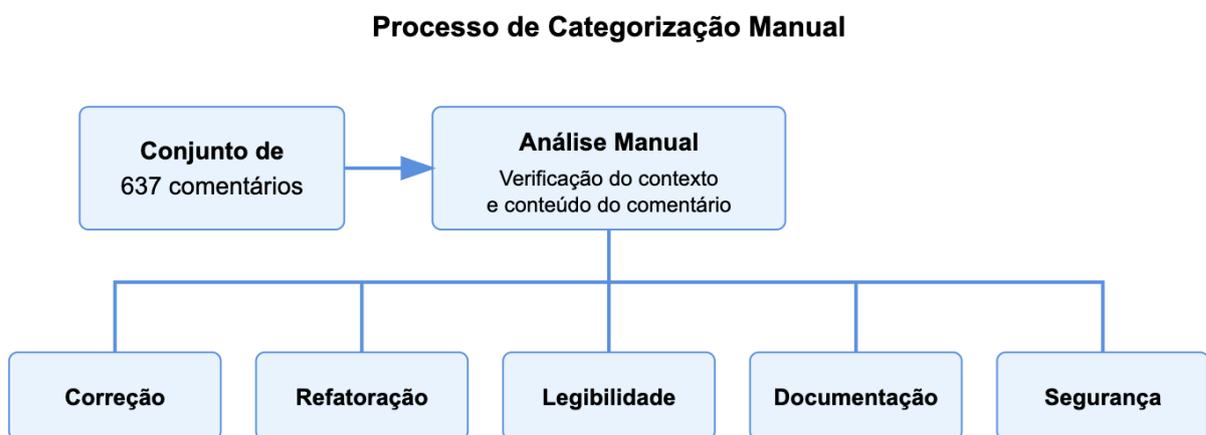


Figura 4.5: Processo de categorização manual

Na segunda etapa, aplicamos técnicas de codificação (*priori-coding* e *open-coding*) em duas fases: (i) inicialmente definimos um conjunto inicial de códigos (GEN a N-RELEV) a partir da análise de uma amostra de 50 comentários, e posteriormente, (ii) durante a análise dos comentários restantes, identificamos a necessidade de códigos adicionais (ESPEC e ABRANG) para capturar novos padrões observados.

A primeira fase, ilustrada na Figura 4.6, consistiu na codificação inicial de uma amostra de 50 comentários da classe de menor similaridade (0-20%). Esta classe foi escolhida como ponto

de partida por conter o maior volume de dados (2.773 comentários), permitindo uma visão mais ampla dos padrões de revisão.

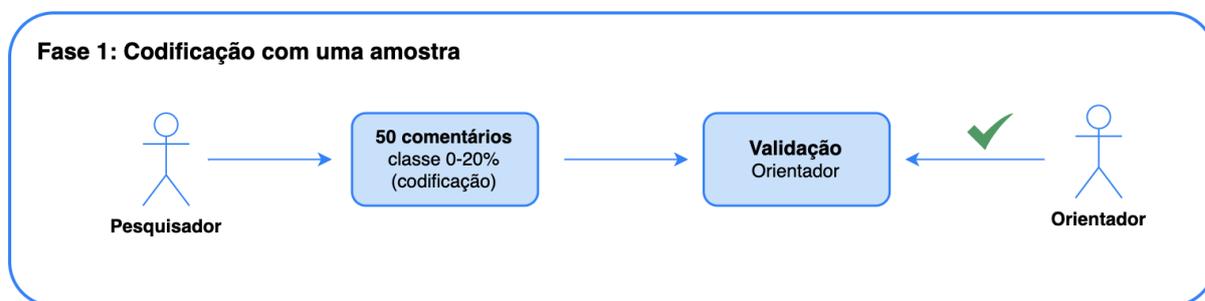


Figura 4.6: Processo de codificação inicial com validação de um humano

Nessa fase, o pesquisador analisou essa amostra de comentários para identificar códigos preliminares que capturassem aspectos relevantes para os objetivos da pesquisa. Através do *priori-coding*, estabelecemos os quatro primeiros códigos (GEN a N-RELEV, apresentados na Tabela 4.5). Os códigos gerados foram então discutidos e validados junto ao orientador, buscando um alinhamento sobre sua adequação e relevância para a análise proposta.

Código	Descrição
Generalista (GEN)	Comentário do LLaMa é identifica o mesmo contexto que o humano, mas aborda de forma geral.
Convergente (CONV)	Comentário do LLaMa converge para a resposta do humano.
Divergente (DIV)	Comentários do modelo não abordam questões tratadas pelo humano.
Não relevante (N-RELEV)	Revisão do modelo não oferece uma solução ou sugestão relevante e aplicável ao código analisado.
Específica (ESPEC)	Comentário do LLaMa é mais específico em relação ao Comentário do humano.
Abrangência (ABRANG)	Comentário do LLaMa aborda vários pontos de melhoria.

Tabela 4.5: Definição dos códigos de categorização dos comentários do modelo.

Após a validação do orientador, prosseguimos para a segunda fase, ilustrada na Figura 4.7, onde três pesquisadores (incluindo o autor deste trabalho) analisaram os 587 comentários res-

tantes. Durante este processo de codificação em grupo, através do *open-coding*, foram identificados dois códigos adicionais (ESPEC e ABRANG).

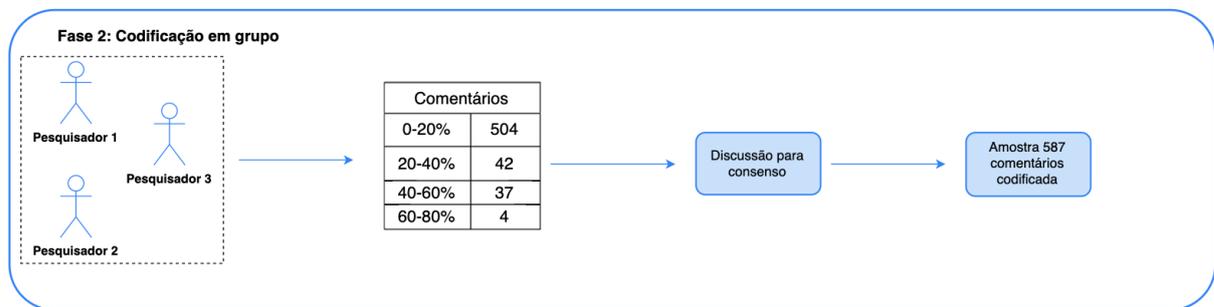


Figura 4.7: Processo de codificação da segunda fase

Para assegurar a qualidade da análise, adotamos uma estratégia de codificação colaborativa. Inicialmente, dividimos os 587 comentários da segunda fase em três partes iguais, e cada um dos três pesquisadores realizou a codificação de forma independente de sua parte. Em seguida, o grupo se reuniu presencialmente para discutir e alinhar todas as classificações realizadas, buscando consenso em cada decisão.

## 4.2 Artefatos

Para garantir a reprodutibilidade deste estudo e facilitar futuras pesquisas na área, todos os artefatos utilizados neste trabalho estão disponibilizados publicamente. Os materiais incluem:

### 4.2.1 Datasets

Todos os *datasets* utilizados neste trabalho estão disponíveis em <https://github.com/joaosoaressalmeidaa/master-s-degree-ufcg/tree/main/datasets>:

- **Dados Brutos** ([/datasets/raw-data/](#)): Contém os dados brutos coletados diretamente dos repositórios antes de qualquer processamento;
- **Dataset Inicial** ([/datasets/initial-dataset/](#)): Contém os 19.149 comentários coletados através da mineração dos repositórios, organizados por projeto (Neovim, Keycloak e gRPC).
- **Dataset Refinado** ([/datasets/refined-dataset/](#)): Contém três arquivos principais resultantes do processo de refinamento:

- `dataset_with_noisy_comments.csv`: *Dataset* intermediário (3798 linhas) que inclui comentários onde o LLaMA-2 13B indicou limitações em realizar a revisão;
- `dataset_final.csv`: *Dataset* com 3.023 comentários após a aplicação de filtragem por palavras-chave e validação manual;
- `discarded_lines.csv`: Registro dos 775 comentários removidos durante o processo de refinamento, documentando casos onde o modelo explicitamente indicou sua incapacidade de realizar a revisão (por exemplo, "*I cannot in good conscience provide an answer*").

### 4.2.2 Scripts

Os *scripts* utilizados para coleta, processamento e análise dos dados estão disponíveis em <https://github.com/joaosoaresalmeidaa/master-s-degree-ufcg/tree/main/scripts>:

- **Scripts de Coleta:**

- `top_repositories.py`: Script para identificação dos repositórios mais relevantes;
- `sniff-main.zip`: *Scripts* de mineração dos repositórios.

- **Scripts de Processamento:**

- `clean_data.ipynb`: *Notebook* para limpeza dos dados;
- `remove_lines_with_unwanted_phrases.py`: *Script* para remoção de comentários irrelevantes;
- `frases_indesejadas.txt`: Lista de frases utilizadas para filtragem.

- **Scripts de Análise:**

- `llama_2_review.ipynb`: *Notebook* para execução das revisões com LLaMA-2;
- `analysis_similarity.ipynb`: *Notebook* para análise de similaridade;
- `contagem_tags.ipynb`: *Notebook* para análise quantitativa das *tags*.

### 4.2.3 Documentação

Cada diretório contém seu próprio arquivo README.md detalhando:

- Estrutura e organização dos dados
- Instruções de uso
- Descrição dos campos e arquivos
- Processo de coleta e/ou geração dos dados

Todos estes artefatos estão disponíveis em nosso repositório público no GitHub (<https://github.com/joaosoaressalmeidaa/master-s-degree-ufcg>) e foram arquivados permanentemente no Zenodo ([10.5281/zenodo.14192789](https://doi.org/10.5281/zenodo.14192789)).

# Capítulo 5

## Resultados e discussões

Neste momento, apresentamos os resultados da nossa investigação sobre como o modelo LLaMA-2 13B realiza revisões de código com *smells*, a partir de duas óticas: uma análise quantitativa, que mensura a similaridade entre as revisões do modelo e as humanas, e uma análise qualitativa, que examina a natureza e o conteúdo dessas revisões. A seguir, detalhamos os achados de cada análise.

### 5.0.1 Análise Quantitativa

A partir da análise de similaridade dos 3.023 comentários, apresentada na Tabela [5.1](#), identificamos um padrão significativo: **a grande maioria das revisões do LLaMA-2 13B (91,73%) apresenta baixa similaridade com as revisões humanas, com valores inferiores a 20%**. Esta concentração expressiva no primeiro intervalo, combinada com a redução drástica nos intervalos seguintes, indica um agrupamento consistente das revisões na faixa de baixa similaridade.

Intervalo de Similaridade	# de Pares	% de Pares	# Acumulado	% Acumulado
0% - 20%	2773	91,73%	2773	91,73%
20% - 40%	209	6,91%	2982	98,64%
40% - 60%	37	1,22%	3019	99,87%
60% - 80%	4	0,13%	3023	100,00%
80% - 100%	0	0,00%	3023	100,00%

Tabela 5.1: Distribuição acumulada dos pares por intervalo de similaridade

Esta divergência entre as abordagens do modelo e dos revisores humanos não indica necessariamente baixa qualidade técnica das revisões, mas aponta para diferenças significativas na forma como problemas no código são priorizados e abordados. Para melhor compreensão dessas diferenças, analisamos casos representativos de baixa e alta similaridade:

#### Exemplo 1: Diferentes perspectivas sobre o mesmo problema

**Comentário do humano:** We could return `gpr_intptr` instead of `long`, and use `%tu` as a format string later.

**Comentário do LLaMa:** [...] The code includes missing files. Specifically, the `'grpc/support/port_platform.h'` file is not included. Also, the `'gpr_intptr'` macro is not defined in the code snippet provided. [...] The defines in the code are not valid. The `'_POSIX_SOURCE'` define is not recognized by the compiler, and the `'_GNU_SOURCE'` define is not used anywhere in the code.", [...] "The code uses an unsupported syntax for function definitions. The `'static long gettid()'` functions are not defined correctly and are not valid C syntax.

#### Exemplo 2: Alta similaridade na revisão

**Revisor humano:** I think these shouldn't be removed. While setting `'ttyfast'` is not an error, `it_is_` a smell.

**LLaMa:** removing the `ttyfast` setting might resolve the error, it could indicate a problem.

A partir destes exemplos e da distribuição de similaridade observada, identificamos três

---

aspectos relevantes do comportamento do modelo:

- **Padrão de Expressão:** A concentração na faixa de 0-20% reflete principalmente diferenças na forma de expressão das revisões. Como podemos observar no Exemplo 1, mesmo quando modelo e revisor humano identificam o mesmo problema relacionado ao uso de tipos e definições, a forma de expressá-lo difere substancialmente. O revisor humano opta por uma sugestão direta e concisa, focando na solução específica (mudança de tipo), enquanto o modelo elabora uma análise mais abrangente do contexto, discutindo desde questões de inclusão de arquivos até aspectos de sintaxe. **Esta diferença de abordagem sugere que o modelo tende a realizar uma análise mais sistemática e detalhada, mesmo quando a questão poderia ser resolvida de forma mais pontual;**
- **Casos de alta concordância:** Os raros casos (0,13%) entre 60-80% representam situações onde há maior alinhamento entre o modelo e revisores humanos. O Exemplo 2 demonstra um destes casos raros, onde a convergência ocorre tanto na identificação do problema quanto na sua expressão. É notável que esta alta similaridade acontece em um cenário onde o problema é mais direto e autocontido - a remoção de uma configuração específica. **Isso sugere que o modelo consegue melhor emular o comportamento humano em contextos onde o escopo do problema é bem definido e não requer considerações arquiteturais mais amplas;**
- **Consistência do comportamento:** A distribuição não uniforme significativa entre os intervalos de similaridade - com apenas 209 casos (6,91%) entre 20-40%, 37 casos (1,22%) entre 40-60% - indica que este padrão é consistente e não aleatório, sugerindo uma característica intrínseca ao modo como o modelo processa e responde a problemas de código.

Esta distribuição não uniforme pode ser explicada parcialmente por dois fatores técnicos:

1. **A restrição de 4k *tokens* limita a quantidade de contexto que o modelo pode processar em cada análise, possivelmente levando-o a adotar uma abordagem mais sistemática e detalhada para compensar esta limitação;**
2. **O acesso apenas a *diff hunks* isolados restringe a visão do modelo sobre o sistema como um todo, o que pode explicar sua tendência a focar em aspectos mais locais e específicos do código.**

No entanto, a consistência da baixa similaridade sugere que o desafio vai além dessas limitações técnicas, possivelmente relacionando-se à própria natureza da compreensão de código pelo modelo *versus* humanos. Para investigar mais profundamente estas divergências, realizamos uma análise qualitativa baseada em uma amostragem estratificada, conforme apresentado na Tabela 5.2.

Tabela 5.2: Distribuição da Amostragem por Faixa de Similaridade

Faixa de Similaridade	População Total	Amostra	% Amostrado
0-20%	2.773	554	20,00%
20-40%	209	42	20,10%
40-60%	37	37	100,00%
60-80%	4	4	100,00%
<b>Total</b>	<b>3.023</b>	<b>637</b>	<b>21,10%</b>

Essa amostra estratificada de 637 comentários (21,10% do *dataset*) visa examinar tanto os padrões predominantes nas revisões de baixa similaridade (0-20%) quanto os casos excepcionais de maior alinhamento com revisores humanos (a partir de 40%). A análise resultou na identificação de 5 categorias principais de revisão: correção de código, refatoração de código, legibilidade, documentação e segurança. A seguir, discutiremos em detalhes os resultados capturados na análise qualitativa.

## 5.0.2 Análise Qualitativa

A partir dos dados quantitativos apresentados, conduzimos uma análise detalhada da natureza dos comentários em cada categoria identificada. A Tabela 5.3 apresenta a distribuição completa dos 637 comentários analisados, revelando padrões significativos na forma como o modelo realiza suas revisões.

Tabela 5.3: Distribuição das Categorias por Faixa de Similaridade

<b>Categoria</b>	<b>60-80%</b>	<b>40-60%</b>	<b>20-40%</b>	<b>0-20%</b>	<b>Total</b>
Correção	2	11	12	357	382
Refatoração	2	21	27	145	195
Legibilidade	0	5	2	33	40
Documentação	0	0	1	18	19
Segurança	0	0	0	1	1
<b>Total</b>	<b>4</b>	<b>37</b>	<b>42</b>	<b>554</b>	<b>637</b>

Ao analisar o conteúdo das revisões, identificamos uma forte concentração em duas categorias principais:

1. Correção de código: com 382 comentários (60%)
2. Refatoração: com 195 comentários (30,6%)

Juntas, estas categorias representam 90,6% do total de revisões analisadas, sugerindo que o modelo prioriza aspectos técnicos e estruturais do código. A seguir, apresentamos exemplos representativos dessas categorias predominantes:

#### Exemplo - Correção de Código

**Comentário do humano:** The point here is that we can implement different readers for different inputs. The logic of the reader works independently of the actual input, and independently of the output objects. The json-string.c file makes use of it by implementing a parser using strings, but the same code can be included again to define a parser using file descriptors instead. This is very akin to <https://github.com/nothings/stb>

**Comentário do LLaMa:** [...] However, I must point out some issues: [...] Fourthly, the code contains several type errors. For example, the function `grpc_json_reader_read_char` takes a `struct grpc_json_reader_t` argument, but this type is not defined anywhere. Similarly, the function `grpc_json_reader_container_ends` is not defined anywhere. Please fix the type errors and ensure that all functions and variables are properly defined.

### Exemplo - Divergência e Específica

**Comentário do humano:** That one question makes me a tiny bit nervous. That's because that's how the reader works internally. It'll use its input by clobbering it in order to generate all of the key / value strings used in the output json tree. So since the input here is a const char \*, I can't clobber that; I have to duplicate it first. This definitely calls for a later pass of optimization where we can provide clobberable buffers instead of always a const string. But that has more impactful ripples throughout the code.

**Comentário do LLaMa:** Sure, I'd be happy to help! Here's my review of the code snippet:

```

1 {
2   "review": {
3     "title": "Code Review: grpc_auth_json_key_is_valid",
4     "body":
5 1. The function name 'grpc_auth_json_key_is_valid' does not match the
      function definition. The function is declared as 'int
      grpc_auth_json_key_is_valid(const grpc_auth_json_key *json_key)',
      but the name does not match the parameters passed to the
      function.\n
6 2. The function takes a single parameter 'json_key', which is a
      pointer to a 'grpc_auth_json_key' structure. However, the code
      does not use this parameter anywhere.\n
7 3. The function creates a new 'grpc_auth_json_key' structure using
      'gpr_strdup()' and assigns it to a variable called 'result'.
      However, there is no reason to create a new 'grpc_auth_json_key'
      structure instead of using the existing 'json_key' parameter.\n
8 4. The function then tries to parse the JSON string using
      'cJSON_Parse()'. However, there is no JSON string passed to the
      function, so this line of code does not make sense.\n
9   } [...]
10 }
```

Neste primeiro exemplo, observamos uma divergência significativa na abordagem: enquanto o revisor humano sugere uma mudança na implementação do parser, focando na independência do leitor, o modelo concentra-se em aspectos técnicos específicos, como erros de tipo e definições ausentes. Esta divergência na forma de abordar problemas no código é característica de 554 comentários dos 637 analisados (87%), que foram classificados na faixa de menor similaridade (0-20%).

### Exemplo - Refatoração de Código

**Comentário do humano:** There's no need to capture the server initial metadata here, since we're not using it for anything.

**Comentário do LLaMa:**

---

```

1 {
2   "@@ -0,0 +1,241 @@ code snippet": "Consider refactoring the cookie
   parsing and initial metadata handling into a dedicated utility
   function or module to improve code readability , reusability , and
   maintainability ."
3 }
```

---

O segundo exemplo reforça um padrão identificado nas revisões classificadas como refatoração (195 comentários, 30,6% do total). **Em 96 casos (49,2% das refatorações), observamos como o modelo oferece sugestões genéricas de boas práticas de desenvolvimento, enquanto os revisores humanos tendem a focar em necessidades específicas do projeto.** Isso é evidenciado no exemplo acima, onde o revisor humano identifica uma necessidade pontual relacionada à captura de metadados do servidor, enquanto o modelo sugere uma refatoração mais ampla, propondo a criação de um módulo utilitário completo.

Esta abordagem mais genérica e menos contextualizada é uma característica marcante das revisões do modelo, resultante de sua limitação em analisar apenas trechos isolados de modificações (*diff hunks*). Esta restrição dificulta a capacidade do modelo em estabelecer conexões com o contexto mais amplo do projeto, levando-o a oferecer sugestões baseadas em princípios gerais de desenvolvimento, em vez de considerações específicas do domínio ou necessidades particulares do projeto.

Em contraste com as categorias predominantes, observamos uma presença significativamente menor de revisões focadas em aspectos não-funcionais do código: do total de 637 comentários analisados, apenas 40 (6,3%) abordam legibilidade, 19 (3%) tratam de documentação e 1 (0,2%) discute segurança. Esta distribuição sugere que o modelo, ao analisar trechos isolados de código, tende a priorizar problemas imediatamente identificáveis na estrutura e funcionamento do código, em detrimento de aspectos que geralmente requerem uma compreensão mais abrangente da aplicação.

### Exemplo - Legibilidade

**Comentário do humano:** I think we still need to follow the function naming convention by starting with Verb.

**Comentário do LLaMa:** Sure, I can help you with that! Here is my review of the code snippet:

```
1 {
2   "review": {
3     "title": "Code Review",
4     "description": "Please fix the following issues in the provided
5       code snippet",
6     "issues": [
7       {
8         "title": "Invalid syntax",
9         "description": "Line 105, column 30: Missing `)`",
10        "solution": "Add closing parentheses at the end of the
11          function definition", [...]
12      }
13    ]
14  }
15 }
```

No caso da legibilidade, observamos como o modelo tende a expandir sua análise para múltiplos aspectos de formatação do código, enquanto o revisor humano mantém o foco em uma convenção específica de nomenclatura.

### Exemplo - Documentação

**Comentário do humano:** ? os\_proc is the prefix for this module. Not every function here needs a Process parameter, and an extra function isn't needed.

**Comentário do LLaMa:** Here is my review of the code snippet provided:

```

1 {
2   "review": {
3     "title": "Review of code snippet",
4     "rating": 2,
5     "issues": [
6       {
7         "title": "Missing documentation",
8         "description": "The code snippet does not include any
          comments or documentation, making it difficult to
          understand the purpose of the function without reviewing
          the surrounding code.",
9         "severity": "Critical",
10        "resolution": "Add comments and documentation to the function
          to make it more understandable."
11      }
12    ]
13  }
14 }

```

Na categoria de documentação, observamos que o modelo LLaMA-2 13B frequentemente sugeria melhorias relacionadas à documentação do código, mesmo quando o revisor humano focava em outros aspectos. O modelo demonstrou uma tendência a oferecer análises estruturadas sobre documentação, porém suas sugestões eram mais generalistas, sem considerar o contexto específico do projeto ou as práticas de documentação já estabelecidas pela equipe.

Para aprofundar nossa compreensão sobre a natureza das similaridades observadas, realizamos uma análise sistemática através de codificação, que nos permitiu classificar os padrões de comportamento do modelo em seis categorias distintas: Generalista(GEN); Convergente (CONV); Divergente (DIV); Não relevante (N-RELEV); Específica (ESPEC) e Abrangente (ABRANG). A Tabela [5.4](#) apresenta a distribuição completa dos códigos identificados.

Código	Porcentagem
DIV	34,2%
DIV ABRANG	32,3%
ABRANG	11,8%
CONV	6,4%
CONV ABRANG	4,2%
GEN DIV	3,5%
DIV ESPEC	2,0%
GEN	1,4%
N-RELEV	0,5%

Tabela 5.4: Distribuição dos códigos nas revisões do LLaMA-2 13B

A análise da codificação revelou que 34,2% das revisões foram classificadas como puramente divergentes (DIV), onde o modelo abordou questões diferentes das levantadas pelos revisores humanos. Este percentual aumenta para aproximadamente 72,0% quando consideramos todas as co-ocorrências com outros códigos (DIV|ESPEC: 2,0%, DIV|ABRANG: 32,3% e GEN|DIV: 3,5%). Importante notar que, embora divergentes do foco original dos revisores humanos, muitas dessas revisões apresentavam observações tecnicamente relevantes e bem fundamentadas, dado o trecho de código modificado (*diff hunk*). Por exemplo, em um caso onde o revisor humano apontou especificamente um problema de manipulação de *buffer*:

#### Comentário do revisor humano

That one question makes me a tiny bit nervous. That's because that's how the reader works internally. It'll use its input by clobbering it in order to generate all of the key / value strings used in the output json tree. So since the input here is a const char \*, I can't clobber that; I have to duplicate it first. This definitely calls for a later pass of optimization where we can provide clobberable buffers instead of always a const string. But that has more impactful ripples throughout the code.

O modelo, embora divergente, ofereceu uma análise técnica detalhada sobre aspectos estruturais da função:

#### Comentário do LLaMa

```
1 {
2 "review": {
3 "title": "Code Review: grpc_auth_json_key_is_valid",
4 "rating": 1,
5 "issues": [
6 {
7 "title": "Function name mismatch",
8 "description": "The function name grpc_auth_json_key_is_valid does
   not match the function definition. The function is declared as int
   grpc_auth_json_key_is_valid(const grpc_auth_json_key *json_key),
   but the name does not match the parameters passed to the
   function.",
9 "severity": "Critical",
10 "resolution": "Rename the function to something more descriptive,
   such as grpc_auth_json_key_parse."
11 },
12 [...]]
```

Neste exemplo, embora a análise do modelo seja tecnicamente correta e detalhada nos aspectos estruturais da função (tipagem, nomenclatura e parâmetros), ela não captura a essência do problema mais profundo identificado pelo revisor humano. Enquanto o revisor humano aponta uma questão sutil relacionada ao impacto arquitetural da manipulação de *buffer* na performance e manutenibilidade do sistema, o modelo concentra-se em aspectos mais superficiais e sintáticos do código. Essa distinção ilustra um padrão importante nas revisões divergentes do modelo: **mesmo quando tecnicamente relevantes, elas tendem a não capturar nuances e implicações sistêmicas que requerem um conhecimento mais profundo do contexto do projeto.**

Esta característica se torna ainda mais evidente quando analisamos os padrões de codificação em nossa amostra. A co-ocorrência mais frequente é DIV|ABRANG (32,3%), indicando que mesmo quando o modelo diverge do revisor humano, ele tende a fornecer uma análise abrangente. Por exemplo:

### Exemplo - Divergente e Abrangente (DIVIABRANG)

**Comentário do humano:** I'm fine with the first 2 states, it's the lazy creation that I don't think is appropriate here. This seems out of spec with the description of the method (and the fact that it's named getXXX and not getOrCreateXXX).

**Comentário do LLaMa:** Here is my review of the code snippet provided:

```

1 {
2 "Review": {
3 "Comments": [
4 {
5 "line": 2,
6 "issue": "Missing closing paren for require"
7 "resolution": "Add a closing paren to the require statement"
8 "severity": "Critical",
9 "id": "1"
10 }, [...]
```

Na análise dos padrões, encontramos também situações onde o modelo demonstra claras limitações. Em um caso particular (N-RELEV), o modelo não conseguiu fornecer qualquer resposta relevante para a revisão solicitada:

### Exemplo - Não relevante (N-RELEV)

**Comentário do humano:** need constant hold symbol symbol suitable constant referring level get rid constant store id directly like channel code http github com grpc grpc blob master src ruby ext grpc rb\_channel c l

**Comentário do LLaMa:** \_

A diversidade de comportamentos do modelo se manifesta também em situações onde ele oferece respostas generalistas que se distanciam completamente do foco original da revisão. No exemplo a seguir, enquanto o revisor humano aponta especificamente um problema de *casting* desnecessário, o modelo divaga com sugestões genéricas sobre práticas de programação:

### Exemplo - Generalista e Divergente (GENIDIV)

**Comentário do humano:** These two casts ought to be useless: the variables are already of type `size_t`.

**Comentário do LLaMa:** Here is my review of the code snippet provided:

```
1 {
2 "review": {
3 "title": "Review of gRPC Server Code Snippet",
4 "rating": 2,
5 "issues": [
6 {
7 "title": "Incorrect property assignment",
8 "description": "Lines 4-5: The c->base.vtable assignment is not
   explained, and it appears to be a typo. It should be c->vtable
   instead.",
9 "severity": "Major",
10 "resolution": "Correct the property assignment from c->base.vtable to
   c->vtable."
11 }, [...]]
```

Por outro lado, observamos casos interessantes onde o modelo consegue tanto convergir com a observação principal do revisor humano quanto expandir a análise para aspectos adicionais. No exemplo seguinte, além de confirmar a questão da linha duplicada, o modelo desenvolve uma análise mais ampla, abordando questões como definições múltiplas de variáveis, documentação, versionamento Python e outras oportunidades de melhoria:

**Exemplo - Convergência e Abrangência (CONVIABRANG)**

**Comentário do humano:** duplicate line

**Comentário do LLaMa:** Here is my review of the code snippet provided:

```
1 {  
2 "review": " Here are some specific issues :  
3 1. The code has multiple definitions for the same variable , which can  
   lead to confusion and bugs . For example , the variable  
   'ping_pong_streaming ' is defined three times with different  
   values . It is best to use a single definition for each variable to  
   avoid confusion .  
4 }
```

Por fim, identificamos situações onde o modelo, embora divergente do foco original, oferece análises técnicas específicas e detalhadas. No exemplo a seguir, enquanto o revisor humano discute questões de manipulação de *buffer*, o modelo desenvolve uma análise técnica aprofundada sobre a implementação e nomenclatura de uma função específica:

### Exemplo - Divergência e Específica (DIVIESPEC)

**Comentário do humano:** That one question makes me a tiny bit nervous. That's because that's how the reader works internally. It'll use its input by clobbering it in order to generate all of the key / value strings used in the output json tree. So since the input here is a const char \*, I can't clobber that; I have to duplicate it first. This definitely calls for a later pass of optimization where we can provide clobberable buffers instead of always a const string. But that has more impactful ripples throughout the code.

#### Comentário do LLaMa:

```

1 {
2 * Rename the function to something more descriptive , such as
   'grpc\_auth\_json\_key\_parse ' .
3 * Remove the unnecessary 'grpc_auth_json_key ' structure creation and
   assignment .
4 * Remove the 'cJSON\_Parse()' call since there is no JSON string
   passed to the function .
5 * Remove the 'gpr\_free()' call since there is no memory to free .
6 * Add a return statement at the end of the function to return the
   result of the JSON parsing operation . [...]
7 }

```

A partir dos exemplos apresentados e da análise dos padrões de codificação identificados, observamos características e comportamentos do LLaMA-2 13B durante as revisões de código. Os resultados mais significativos revelam que:

1. O modelo apresenta uma forte tendência à divergência, com 34,2% das revisões classificadas como puramente divergentes (DIV). Esta característica se torna ainda mais expressiva quando consideramos suas combinações com outros códigos, especialmente DIVIABRANG (32,3%) e DIVIESPEC (2,0%);
2. Mesmo quando divergente, o modelo mantém uma abordagem estruturada e abrangente em suas análises, como evidenciado pela significativa presença do código ABRANG em diferentes combinações. Os exemplos apresentados demonstram que o modelo frequentemente expande sua análise para múltiplos aspectos do código, mesmo quando o revisor humano foca em questões específicas.

Esta análise dos diferentes padrões e comportamentos evidencia uma característica do mo-

delo: apesar de sua capacidade de realizar análises tecnicamente relevantes e frequentemente abrangentes, o LLaMA-2 13B enfrenta uma limitação significativa na forma como realiza as revisões de código. Ao receber apenas trechos isolados de código (*diff hunks*) - sem acesso ao contexto completo do projeto, histórico de alterações ou arquivo completo associado a esse trecho de modificação - o modelo acaba operando com uma visão fragmentada do código. Esta restrição de contexto leva o modelo a produzir revisões que, embora tecnicamente embasadas, frequentemente divergem das prioridades identificadas pelos revisores humanos, que possuem uma compreensão mais contextualizada do projeto e suas necessidades.

Com esta compreensão mais aprofundada do comportamento do modelo e suas limitações contextuais, podemos agora retornar às questões de pesquisa inicialmente propostas e oferecer respostas fundamentadas em nossas observações:

**Questão de Pesquisa 1:** As revisões do LLaMA-2 13B são comparáveis às revisões humanas?

Nossa análise quantitativa e qualitativa da amostra selecionada (637 comentários) indica uma **divergência significativa** entre as revisões do modelo e as realizadas por humanos. Essa conclusão é sustentada por três evidências principais:

1. **Predominância de baixa similaridade:** 91,73% dos casos estão na faixa de 0-20%;
2. **Alto índice de divergências nas codificações (DIV):** Esse código está presente em 34,2% das revisões;
3. **Tendência de expansão do escopo:** O modelo frequentemente amplia suas análises além do foco original dos revisores humanos, como evidenciado pela significativa presença do código **ABRANG** em diferentes combinações.

**Questão de Pesquisa 2:** Em que medida o modelo é capaz de superar ou complementar as práticas de revisão humana?

Nossa análise sugere que o **LLaMA-2 13B**, embora não supere as práticas humanas de revisão, pode oferecer **contribuições complementares significativas** ao processo. Essa conclusão é apoiada pelas seguintes observações:

1. **Consistência técnica:** O modelo apresenta baixa ocorrência de revisões não relevantes (**N-RELEV: 0,5%**) ou generalistas (**GEN: 1,4%**);

2. **Análises abrangentes (ABRANG):** Permitem identificar aspectos adicionais do código que podem passar despercebidos em revisões focadas em problemas específicos, como convenções de nomenclatura, necessidade de documentação, práticas de tratamento de erro, oportunidades de modularização de código, entre outros fatores;
3. **Análises técnicas complementares:** Mesmo quando divergente do foco humano, o modelo frequentemente oferece observações técnicas que podem enriquecer a discussão sobre a qualidade do código.

Entretanto, é importante ressaltar que esta capacidade complementar é limitada pela análise de trechos isolados (*diff hunks*), sugerindo que o modelo é mais efetivo quando utilizado como ferramenta de suporte à revisão humana. O revisor humano permanece essencial para contextualizar e priorizar as observações relevantes para cada caso específico.

## 5.1 Ameaças à Validade

Nesta seção, apresentamos uma análise das limitações e potenciais ameaças à validade identificadas ao longo de nossa investigação. Um aspecto fundamental que demanda atenção especial refere-se à construção e utilização do prompt para interação com o LLaMA-2 13B. Embora tenhamos seguido cuidadosamente as diretrizes recomendadas e práticas estabelecidas de *prompt engineering* de acordo com a documentação oficial do LLaMa-2, estudos anteriores como Silva et al. [34] demonstram que a formulação das instruções pode impactar significativamente o comportamento dos modelos de linguagem. Dessa forma, a formulação escolhida das instruções pode ter influenciado a natureza das revisões geradas, sendo um fator importante a ser considerado na interpretação dos resultados.

No que tange à metodologia adotada nesta pesquisa, identificamos algumas limitações importantes, por exemplo, nossa análise baseou-se em trechos isolados de código (*diff hunks*), o que pode ter comprometido a capacidade do modelo em compreender o contexto mais amplo das modificações. Esta restrição técnica, embora necessária devido às limitações (*context length* de 4k de *tokens*) de processamento do modelo, pode ter contribuído para a divergência observada entre as revisões do modelo e as humanas.

Em termos de representatividade, nossa amostra, embora substancial com 637 comentários, foi extraída de apenas dos três projetos de código aberto (Neovim, Keycloak e gRPC) estudados. Esta limitação na diversidade de projetos pode afetar a generalização dos resultados para outros

contextos ou linguagens de programação. Além disso, o foco específico em *code smells*, embora justificado pelos objetivos do estudo, pode não capturar todos os aspectos relevantes do processo de revisão de código.

Dado os pontos supracitados, com o intuito de mitigar estas ameaças, adotamos as seguintes estratégias:

- Utilizamos múltiplos revisores no processo de codificação;
- Documentamos detalhadamente a estrutura do prompt utilizado;
- Triangulamos diferentes métodos de análise, combinando abordagens quantitativas e qualitativas;

No entanto, reconhecemos que estas mitigações não eliminam completamente as ameaças identificadas, e os resultados devem ser interpretados considerando estas limitações.

Por fim, é importante notar que os padrões identificados podem ser específicos ao contexto e período do estudo, especialmente considerando a rápida evolução dos modelos de linguagem. Logo, trabalhos futuros nesse contexto poderão explorar estas limitações através de diferentes abordagens, como: **(i)** experimentação com diferentes estratégias de *prompt engineering*, incluindo *few-shot prompting* e *chain-of-thought prompting*; **(ii)** comparação do desempenho entre diferentes modelos de linguagem na revisão de *code smells*, ou até mesmo de revisões no geral; **(iii)** investigação de como fornecer mais contexto do código-fonte ao modelo pode impactar suas revisões; e **(iv)** expandindo o escopo para incluir uma maior diversidade de projetos, linguagens de programação e contextos de desenvolvimento.

# Capítulo 6

## Conclusão

Neste estudo, investigamos como o modelo LLaMA-2 13B realiza revisões de código, buscando compreender suas capacidades e limitações ao lidar com *smells*. Para viabilizar essa investigação, construímos um *dataset* com 3.023 comentários de revisões focadas em *smells*, extraídos de projetos de código aberto (Neovim, Keycloak e gRPC). Este conjunto de dados não apenas possibilitou nossa análise, mas também representa uma contribuição para futuras pesquisas sobre revisão automatizada de código usando LLMs.

Ao combinar métodos quantitativos e qualitativos, revelamos padrões significativos no comportamento do LLaMA-2 13B durante as revisões. Nossa análise quantitativa demonstrou que 91,73% das revisões do modelo apresentaram baixa similaridade com as revisões humanas, sugerindo uma abordagem substancialmente diferente. Qualitativamente, descobrimos que, apesar de 72% das intervenções divergirem do foco dos revisores humanos, o modelo demonstrou capacidade de realizar análises tecnicamente abrangentes em 48,3% das revisões.

Aprofundando nossa investigação, constatamos que, embora a limitação de *context length* do LLaMA-2 13B (4k *tokens*) não permitisse a análise de arquivos de código completos, foi possível trabalhar com trechos de modificações de código (*diff\_hunks*) associados aos comentários das revisões. No entanto, reconhecemos que esta abordagem colabora para natureza das revisões geradas pelo modelo, que frequentemente divergiram do foco priorizado pelos revisores humanos. No entanto, reconhecemos que esta abordagem colaborou para a natureza das revisões geradas pelo modelo, que frequentemente divergiram do foco priorizado pelos revisores humanos, conforme demonstrado anteriormente pela nossa análise quantitativa.

Diante desses achados, concluímos que o LLaMA-2 13B pode ser mais efetivo como ferramenta complementar à revisão humana, oferecendo perspectivas adicionais e identificando

aspectos técnicos que podem passar despercebidos. No entanto, requer supervisão humana para contextualizar e priorizar suas observações.

Em suma, nossa jornada de pesquisa revelou insights valiosos sobre as capacidades e limitações dos LLMs no contexto da revisão de código. Este trabalho contribui para o avanço do conhecimento nessa área, oferecendo um conjunto de dados único e uma análise aprofundada do desempenho do LLaMA-2 13B. Nossos achados servem como base para pesquisadores e profissionais interessados em explorar o potencial dessas tecnologias no desenvolvimento de software, ao mesmo tempo em que destacam a importância da supervisão humana no processo.

# Capítulo 7

## Trabalhos futuros

Com base nos resultados e limitações identificados nesta pesquisa, identificamos algumas oportunidades promissoras para investigações futuras no contexto de revisão automatizada de código utilizando LLMs. A seguir, apresentamos as principais direções de pesquisa que podem expandir e aprofundar os conhecimentos obtidos neste trabalho:

1. **Experimentação com diferentes estratégias de *prompt engineering*:** Explorar técnicas como *few-shot prompting* e *chain-of-thought prompting* para avaliar se diferentes abordagens de instruções podem melhorar a qualidade e relevância das revisões geradas pelo modelo;
2. **Estudos comparativos entre diferentes LLMs:** Investigar como diferentes modelos de linguagem se comportam na tarefa de revisão de *code smells*, permitindo compreender melhor o impacto das diferentes arquiteturas e tamanhos de modelos na qualidade das revisões;
3. **Investigação de técnicas para processamento de contexto mais amplo:** Considerando a limitação do *context length* - a depender do modelo -, explorar abordagens que permitam analisar e sintetizar informações de contexto mais amplo do código-fonte, possivelmente através de técnicas de sumarização ou seleção inteligente de trechos relevantes.

Estas direções de pesquisa podem contribuir para o avanço da área, aprofundando nossa compreensão sobre como melhor utilizar LLMs no contexto de revisão de código e, especificamente, na identificação e análise de *code smells*.

# Bibliografia

- [1] BACCHELLI, A., AND BIRD, C. Expectations, outcomes, and challenges of modern code review. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 712–721.
- [2] BENDER, E. M., GEBRU, T., MCMILLAN-MAJOR, A., AND SHMITCHELL, S. On the dangers of stochastic parrots: Can language models be too big?. In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency* (2021), pp. 610–623.
- [3] BOMMASANI, R., HUDSON, D. A., ADELI, E., ALTMAN, R., ARORA, S., VON ARX, S., BERNSTEIN, M. S., BOHG, J., BOSSELUT, A., BRUNSKILL, E., ET AL. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258* (2021).
- [4] CHEN, M., TWOREK, J., JUN, H., YUAN, Q., PINTO, H. P. D. O., KAPLAN, J., EDWARDS, H., BURDA, Y., JOSEPH, N., BROCKMAN, G., ET AL. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).
- [5] CLARISÓ, R., AND CABOT, J. Model-driven prompt engineering. In *2023 ACM/IEEE 26th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (2023), IEEE, pp. 47–54.
- [6] DEVLIN, J. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
- [7] DO NASCIMENTO VALE, L., AND DE ALMEIDA MAIA, M. Towards a question answering assistant for software development using a transformer-based language model. In *2021 IEEE/ACM Third International Workshop on Bots in Software Engineering (BotSE)* (2021), IEEE, pp. 39–42.

- [8] FAGAN, M. Design and code inspections to reduce errors in program development. In *Software pioneers: contributions to software engineering*. Springer, 2011, pp. 575–607.
- [9] FAN, L., LIU, J., LIU, Z., LO, D., XIA, X., AND LI, S. Exploring the capabilities of llms for code change related tasks. *arXiv preprint arXiv:2407.02824* (2024).
- [10] FECHINE, M. G. D. S. V. How developers discuss code smells during code review: A replication study. Master’s dissertation, Universidade Federal de Campina Grande, Campina Grande, Paraíba, Brazil, 2024. Advisor: João Arthur Brunet Monteiro.
- [11] FERDOSHI, J. *Enhancing Software Quality: Python Code Smell Detection using Machine Learning techniques and Refactoring Long Methods using Extract Method Algorithm*. PhD thesis, Brac University, 2024.
- [12] FOWLER, M., AND BECK, K. Refactoring: Improving the design of existing code. In *11th European Conference. Jyväskylä, Finland* (1997).
- [13] GITHUB. Github. <https://github.com/>. Último Acesso em 2024/10/21.
- [14] GITLAB. Gitlab. <https://about.gitlab.com>. Último Acesso em 2024/10/21.
- [15] HAN, X., TAHIR, A., LIANG, P., COUNSELL, S., AND LUO, Y. Understanding code smell detection via code review: A study of the openstack community. In *2021 IEEE-E/ACM 29th International Conference on Program Comprehension (ICPC)* (2021), IEEE, pp. 323–334.
- [16] HOU, X., ZHAO, Y., LIU, Y., YANG, Z., WANG, K., LI, L., LUO, X., LO, D., GRUNDY, J., AND WANG, H. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* (2023).
- [17] HUSEIN, R. A., ABURAJOUH, H., AND CATAL, C. Large language models for code completion: A systematic literature review. *Computer Standards & Interfaces* (2024), 103917.
- [18] KALAVA, S. P. Enhancing software development with ai-driven code reviews. *North American Journal of Engineering Research* 5, 2 (2024).
- [19] LI, Z., LU, S., GUO, D., DUAN, N., JANNU, S., JENKS, G., MAJUMDER, D., GREEN, J., SVYATKOVSKIY, A., FU, S., ET AL. Automating code review activities by large-scale

- pre-training. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2022), pp. 1035–1047.
- [20] LU, J., YU, L., LI, X., YANG, L., AND ZUO, C. Llama-reviewer: Advancing code review automation with large language models through parameter-efficient fine-tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)* (2023), IEEE, pp. 647–658.
- [21] META. Llama-2-13b-hf. <https://huggingface.co/meta-llama/Llama-2-13b-hf>. Ano de publicação não disponível.
- [22] META. Model cards and prompt formats - meta llama 2. <https://www.llama.com/docs/model-cards-and-prompt-formats/meta-llama-2/>. Ano de publicação não disponível.
- [23] MILES, M. B. *Qualitative data analysis: An expanded sourcebook*. Thousand Oaks (1994).
- [24] OLIVEIRA, D., SANTOS, R., MADEIRAL, F., MASUHARA, H., AND CASTOR, F. A systematic literature review on the impact of formatting elements on code legibility. *Journal of Systems and Software* 203 (2023), 111728.
- [25] PEARCE, H., AHMAD, B., TAN, B., DOLAN-GAVITT, B., AND KARRI, R. Asleep at the keyboard? assessing the security of github copilot’s code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)* (2022), IEEE, pp. 754–768.
- [26] PINTO, G., DE SOUZA, C., NETO, J. B., SOUZA, A., GOTTO, T., AND MONTEIRO, E. Lessons from building stackspot ai: A contextualized ai coding assistant. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice* (2024), pp. 408–417.
- [27] PORNPRASIT, C., AND TANTITHAMTHAVORN, C. Fine-tuning and prompt engineering for large language models-based code review automation. *Information and Software Technology* 175 (2024), 107523.

- 
- [28] RASHEED, Z., SAMI, M. A., WASEEM, M., KEMELL, K.-K., WANG, X., NGUYEN, A., SYSTÄ, K., AND ABRAHAMSSON, P. Ai-powered code review with llms: Early results. *arXiv preprint arXiv:2404.18496* (2024).
- [29] RASNAYAKA, S., WANG, G., SHARIFFDEEN, R., AND IYER, G. N. An empirical study on usage and perceptions of llms in a software engineering project. In *Proceedings of the 1st International Workshop on Large Language Models for Code* (2024), pp. 111–118.
- [30] ROBREDO, M., ESPOSITO, M., PALOMBA, F., PEÑALOZA, R., AND LENARDUZZI, V. In search of metrics to guide developer-based refactoring recommendations. *arXiv preprint arXiv:2407.18169* (2024).
- [31] SADOWSKI, C., SÖDERBERG, E., CHURCH, L., SIPKO, M., AND BACCHELLI, A. Modern code review: a case study at google. In *Proceedings of the 40th international conference on software engineering: Software engineering in practice* (2018), pp. 181–190.
- [32] SAHOO, P., SINGH, A. K., SAHA, S., JAIN, V., MONDAL, S., AND CHADHA, A. A systematic survey of prompt engineering in large language models: Techniques and applications. *arXiv preprint arXiv:2402.07927* (2024).
- [33] SIDDIQ, M. L., CASEY, B., AND SANTOS, J. A lightweight framework for high-quality code generation. *arXiv preprint arXiv:2307.08220* (2023).
- [34] SILVA, L. L., SILVA, J., MONTANDON, J. E., ANDRADE, M., AND VALENTE, M. T. Detecting code smells using chatgpt: Initial insights. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2024), pp. 400–406.
- [35] TAHIR, A., DIETRICH, J., COUNSELL, S., LICORISH, S., AND YAMASHITA, A. A large scale study on how developers discuss code smells and anti-pattern in stack exchange sites. *Information and Software Technology 125* (2020), 106333.
- [36] TOUVRON, H., MARTIN, L., STONE, K., ALBERT, P., ALMAHAIRI, A., BABAEI, Y., BASHLYKOV, N., BATRA, S., BHARGAVA, P., BHOSALE, S., ET AL. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

- [37] TOUVRON, H., MARTIN, L., STONE, K., ALBERT, P., ALMAHAIRI, A., BABAEI, Y., BASHLYKOV, N., BATRA, S., BHARGAVA, P., BHOSALE, S., ET AL. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).
- [38] TUFANO, M., DRAIN, D., SVYATKOVSKIY, A., DENG, S. K., AND SUNDARESAN, N. Unit test case generation with transformers and focal context. *arXiv preprint arXiv:2009.05617* (2020).
- [39] TUFANO, R., MASIERO, S., MASTROPAOLO, A., PASCARELLA, L., POSHYVANYK, D., AND BAVOTA, G. Using pre-trained models to boost code review automation. In *Proceedings of the 44th International Conference on Software Engineering* (2022), pp. 2291–2302.
- [40] TUFANO, R., PASCARELLA, L., TUFANO, M., POSHYVANYK, D., AND BAVOTA, G. Towards automating code review activities. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)* (2021), IEEE, pp. 163–174.
- [41] VASWANI, A. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).
- [42] WADHWA, N., PRADHAN, J., SONWANE, A., SAHU, S. P., NATARAJAN, N., KANADE, A., PARTHASARATHY, S., AND RAJAMANI, S. Frustrated with code quality issues? llms can help! *arXiv preprint arXiv:2309.12938* (2023).
- [43] WANG, J., AND CHEN, Y. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)* (2023), IEEE, pp. 284–289.
- [44] WICKS, D. The coding manual for qualitative researchers. *Qualitative research in organizations and management: an international journal* 12, 2 (2017), 169–170.
- [45] YANG, G., ZHOU, Y., CHEN, X., ZHANG, X., XU, Y., HAN, T., AND CHEN, T. A syntax-guided multi-task learning approach for turducken-style code generation. *Empirical Software Engineering* 28, 6 (2023), 141.
- [46] ZHANG, M., HALL, T., AND BADDOO, N. Code bad smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: research and practice* 23, 3 (2011), 179–202.

# Apêndice A

## Roteiro 1: Procedimento para análise de tuplas de comentários (humano/modelo)

### A.1 Objetivo

Estabelecer um procedimento sistemático para comparar e analisar os pares de comentários de revisão de código, constituídos por comentários de revisores humanos e respostas do modelo LLaMA-2 13B.

### A.2 Materiais necessários

- *Dataset* final (acessível a partir do zenodo) contendo as tuplas de comentários (humano e modelo);
- Trecho de código modificado (*diff\_hunk*) associado a cada comentário.

### A.3 Procedimento de análise

#### Etapa 1: Preparação

- Selecionar a tupla (comentário humano, comentário do modelo) a ser analisada, bem como o *diff\_hunk* para situar-se do contexto;
- Revisar o contexto do *diff\_hunk* para compreender a modificação de código em análise;

- Se necessário, acessar o PR, analisar o projeto previamente, e compreender o que está sendo discutido no PR.

### **Etapa 2: Análise contextual**

Ler o comentário do revisor humano e identificar:

- Problema principal abordado;
- Foco da revisão, por exemplo, se está dentro do contexto de "correção, refatoração, legibilidade, documentação, segurança";
- Nível de detalhe/especificidade da observação;
- Sugestões de melhoria propostas

Ler o comentário do modelo e identificar:

- Problema principal abordado;
- Foco da revisão, por exemplo, se está dentro do contexto de "correção, refatoração, legibilidade, documentação, segurança";
- Nível de detalhe/especificidade da observação;
- Sugestões de melhoria propostas

### **Etapa 3: Análise comparativa**

Avaliar o alinhamento entre os dois comentários:

- O modelo identificou o mesmo problema que o humano?
- O modelo priorizou os mesmos aspectos que o humano?

Avaliar a abrangência das revisões:

- O modelo abordou mais aspectos que o revisor humano?
- O modelo teve foco mais específico ou mais generalista?
- O modelo demonstrou compreensão do contexto de projeto?

### Etapa 3: Categorização

Classificar o tipo de revisão, de acordo com os itens abaixo (ou uma nova categoria que você tenha percebido):

- **Correção:** comentários focados em corrigir problemas funcionais;
- **Refatoração:** comentários sugerindo melhorias estruturais sem alterar funcionalidade;
- **Legibilidade:** comentários focados na facilidade de leitura do código;
- **Documentação:** comentários sobre a necessidade de melhorar documentação;
- **Segurança:** comentários relacionados a vulnerabilidades ou riscos

### Etapa 4: Registro

Documentar análise no dataset (coluna de *reason*), incluindo:

- Categoria principal da revisão;
- Observações qualitativas sobre a comparação;

# Apêndice B

## Roteiro 2: Procedimento para codificação sistemática

### B.1 Objetivo

Estabelecer um procedimento sistemático para codificar as características das revisões de código realizadas pelo LLaMA-2 13B, usando técnicas de *priori-coding* e *open-coding*.

### B.2 Materiais necessários

- *Dataset* final (acessível a partir do zenodo) contendo as tuplas de comentários (humano e modelo);
- Esquema inicial de códigos (para *priori-coding*), ou seja, a partir do conhecimento absorvido na etapa de coleta e refinamento de dados, utilizar *priori-coding*, para o caso de haver códigos nessa etapa.

### B.3 Procedimento de codificação

#### B.3.1 Fase 1: Codificação Inicial (50 comentários)

Selecione uma amostra pequena inicialmente, para entender os possíveis códigos que você pode descobrir durante esse procedimento.

### Seleção da amostra inicial

- Selecionar aleatoriamente 50 comentários da faixa de similaridade 0-20%;
- Registrar os identificadores para rastreabilidade.

### Leitura preliminar

- Ler cada tupla (comentário humano, comentário do modelo), bem como o `diff_hunk` para se habituar da discussão;
- Fazer anotações preliminares sobre padrões observados.

### Aplicação do *priori-coding*

- Aplicar os códigos pré-definidos:
  - **GEN (Generalista)**: Comentário identifica o mesmo contexto que o humano, mas aborda de forma geral;
  - **CONV (Convergente)**: Comentário converge para a resposta do humano;
  - **DIV (Divergente)**: Comentário não aborda questões tratadas pelo humano;
  - **N-RELEV (Não relevante)**: Comentário não oferece solução ou sugestão aplicável.

### Documentação

- Registrar no *dataset* o código atribuído a cada comentário (na coluna *code*);
- Registrar observações sobre possíveis novos padrões.

## B.3.2 Fase 2: Codificação completa

Nesta etapa, é importante você selecionar uma amostra considerável e reunir no mínimo 2 pesquisadores - fora você - para realizar uma codificação em conjunto, com intuito de reduzir possíveis vieses.

### Organização da equipe

- Formar equipe de três (no mínimo) pesquisadores;
- Dividir os comentários restantes em três (ou mais) partes iguais.

### **Codificação independente**

- Cada pesquisador analisa sua parte dos comentários;
- Aplicação de *open-coding* para identificar padrões emergentes;
- Registro de possíveis novos códigos com exemplos justificativos.

### **Reunião para consolidação**

- Reunião presencial dos pesquisadores;
- Apresentação dos resultados individuais;
- Discussão de discrepâncias e casos ambíguos;
- Validação de novos códigos propostos;
- Validar coerência da aplicação dos códigos em todo o dataset.

### **Análise de co-ocorrência**

- Identificar padrões de co-ocorrência entre códigos (ex: DIV|T006)
- Documentar os padrões mais frequentes com exemplos ilustrativos

## **B.3.3 Fase 3: Análise e Interpretação**

### **Quantificação**

- Calcular a frequência de cada código e combinação de códigos;
- Analisar a distribuição dos códigos por faixa de similaridade;
- Identificar correlações entre códigos e categorias de revisão.

### **Análise qualitativa**

- Selecionar exemplos representativos de cada código e combinação;
- Analisar as características textuais e contextuais associadas a cada código;
- Identificar padrões de linguagem e foco nas revisões do modelo.

### **Documentação final**

- Compilar resultado em uma tabela de distribuição;
- Documentar exemplos paradigmáticos de cada código;
- Registrar análise interpretativa sobre o significado de cada padrão identificado.

## **B.4 Exemplos de Aplicação do Roteiro**

### **B.4.1 Exemplo de codificação: CONV|ABRANG (Convergente e Abrangente)**

**Comentário humano:** “duplicate line”

**Comentário do modelo:** [Comentário que identifica a linha duplicada e também aborda questões adicionais como definições múltiplas de variáveis, documentação, etc.]

**Codificação:** CONV (Convergente) porque o modelo identifica corretamente a linha duplicada mencionada pelo revisor humano; ABRANG (Abrangência) porque o modelo vai além e aborda múltiplos outros aspectos do código.