Detecção de *Code Smells* em JavaScript utilizando LLMs em Ambiente Local

Tarcisio S. Pieroni[1], **Carlos Alberto S. Junior**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia de Minas Gerais (IFMG)
Rodovia MGC 262 - Sobradinho - Sabará - MG - CEP: 34515-640

0073276@academico.ifmg.edu.br, carlos.junior@ifmg.edu.br

Abstract. *Code smells negatively impact software maintainability and evolution. Although LLMs demonstrate potential in static analysis, the reliance on proprietary cloud-based models raises data privacy and intellectual property concerns. This paper investigates the effectiveness of locally executing an LLM (gpt-oss-20b) to detect code smells in JavaScript projects. A study was conducted using a dataset of 840 instances, comparing three prompt engineering strategies: Zero-Shot, Role-Based, and Chain-of-Thought. The results indicate that while the model detects Long Method with high performance, God Class detection is significantly more challenging due to context limitations, and the Chain-of-Thought strategy is crucial for reducing false positives. The findings suggest that local LLMs are a viable and secure alternative for code review, provided that appropriate prompting techniques are applied.*

Resumo. *Code smells impactam negativamente a manutenibilidade e evolução do software. Embora LLMs demonstrem potencial na análise estática, a dependência de modelos proprietários em nuvem levanta preocupações sobre privacidade de dados e propriedade intelectual. Este artigo investiga a eficácia da execução local de um LLM (gpt-oss-20b) na detecção de code smells em projetos JavaScript. Realizou-se um estudo com um dataset de 840 instâncias, comparando três estratégias de engenharia de prompt: Zero-Shot, Role-Based e Chain-of-Thought. Os resultados indicam que, embora o modelo detecte Long Method com alto desempenho, a detecção de God Class é significativamente mais desafiadora devido a limitações de contexto e que a estratégia Chain-of-Thought é crucial para reduzir falsos positivos. As descobertas sugerem que LLMs locais são uma alternativa viável e segura para revisão de código, desde que aplicadas técnicas adequadas de prompt.*

1. Introdução

A manutenção e a evolução de software são desafios constantes no ciclo de vida de desenvolvimento, frequentemente dificultados pela presença de code smells. Definidos como sintomas no código-fonte que indicam problemas profundos de design [Fowler 2018], esses padrões, embora não sejam bugs, prejudicam a legibilidade e aumentam a dívida técnica.

Historicamente, a detecção de *code smells* tem dependido de inspeção manual ou ferramentas de análise estática baseadas em métricas rígidas. Com o avanço

da Inteligência Artificial, abordagens baseadas em Aprendizado de Máquina (do inglês, *Machine Learning* ML) ganharam destaque, porém muitas vezes limitadas a ecossistemas específicos, como Java, e dependentes da extração manual de características. Recentemente, as LLMs demonstraram capacidade promissora na compreensão semântica de código, superando algumas limitações das ferramentas estáticas. No entanto, o uso de LLMs proprietários baseados em nuvem (como GPT-4) impõe barreiras significativas relacionadas à privacidade de dados e propriedade intelectual. Enviar código corporativo sensível para APIs de terceiros representa um risco de segurança que algumas organizações não podem assumir. Além disso, existe uma escassez de estudos focados na aplicação dessas técnicas em linguagens dinâmicas como JavaScript, que possui paradigmas distintos de linguagens estáticas. Neste contexto, este trabalho propõe avaliar a viabilidade técnica do uso de LLMs executados localmente para a detecção de code smells. O objetivo é validar se modelos abertos, rodando em infraestrutura própria, conseguem oferecer um desempenho comparável a soluções proprietárias, garantindo a soberania dos dados.

Para isso, este estudo, utilizando o modelo *gpt-oss-20b* sobre um *dataset* de projetos JavaScript reais, investiga o impacto de diferentes estratégias de Engenharia de Prompt *Zero-Shot*, *Role-Based* e *Chain-of-Thought* (CoT) analisando métricas de acurácia, precisão, revocação e F1-score. Os resultados contribuem para a literatura ao demonstrar como o raciocínio passo a passo (CoT) pode mitigar alucinações em modelos locais e ao evidenciar as limitações de contexto na análise de classes extensas.

2. Trabalhos Relacionados

A detecção automatizada de *code smells* tem sido investigada na literatura de Engenharia de Software, medindo a eficiência das técnicas e buscando a comparação dos métodos mais utilizados, como as linguagens mais populares. Esta seção apresenta estudos que fundamentam esta pesquisa, explicitando três vertentes: abordagens baseadas em Aprendizado de Máquina, a construção de *datasets* específicos para JavaScript e o uso de LLMs na detecção de *code smells*.

2.1. Detecção via aprendizado de máquina

A detecção de *code smells*, com abordagens baseadas em Aprendizado de Máquina (ML), é uma evolução aos métodos de rotulagem manual. Moreira et al. [Moreira et al. 2024] realizaram um estudo avaliando cinco algoritmos de ML (MLP, Árvore de Decisão, Floresta Aleatória, Gradient Boost e SVM) para detectar quatro tipos de smells: *God Class*, *Data Class*, *Feature Envy* e *Long Method*. Utilizando um dataset para linguagem Java e métricas de software como atributos de entrada, os autores reportaram uma acurácia variando entre 93,7% e 99,2%, com o algoritmo Floresta Aleatória (*Random Forest*) apresentando o melhor desempenho geral. Este trabalho evidencia a eficácia de classificadores tradicionais quando alimentados com métricas de software bem definidas, embora limitado ao ecossistema Java. Adicionalmente, Arcelli Fontana et al. [Arcelli Fontana et al. 2016] realizaram um estudo comparativo abrangente de técnicas de aprendizado de máquina para detecção de *code smells*, avaliando diferentes conjuntos de dados e reforçando o potencial dessas técnicas. No cenário de datasets de múltiplos rótulos, Alomari et al. [Alomari et al. 2025] propuseram o *SmellyCode++*, ampliando as referências e bases de dados disponíveis para o treinamento de modelos.

2.2. Construção do dataset

Apesar da larga bibliografia de pesquisas em Java, fornecendo diversos datasets, linguagens dinâmicas como JavaScript carecem de recursos padronizados. Sarafim et al. [Sarafim et al. 2024] abordaram essa limitação ao introduzir um *dataset* rotulado publicamente disponível focado especificamente na detecção de *code smells* em JavaScript. Os autores destacam que as diferenças de paradigma entre Java e JavaScript impedem a transferência direta de modelos e ferramentas de uma linguagem para outra. O estudo de Sarafim construiu um *ground truth* através de um processo de anotação, com múltiplos avaliadores analisando instâncias de projetos modernos hospedados no GitHub. O *dataset* resultante foca nos *code smells God Class* (classe Deus) e *Long Method* (método longo), fornecendo as métricas estruturais e o código-fonte bruto correspondente. Esses *smells* possuem definições operacionais consolidadas na literatura [Fowler 2018]. Uma *God Class* representa uma classe inflada que concentra muitas responsabilidades, violando o Princípio da Responsabilidade Única e dificultando a manutenção. Um *Long Method* é um método excessivamente extenso e complexo, com muitas linhas de código, o que prejudica a legibilidade e a testabilidade. No contexto de JavaScript, a detecção automática desses desvios apresenta desafios singulares devido ao dinamismo da linguagem (como o uso intenso de funções de primeira classe e protótipos). Para mitigar essas dificuldades, ferramentas específicas como o *JSNose* [Fard and Mesbah 2013] foram desenvolvidas, integrando análises estáticas e dinâmicas para capturar o comportamento em tempo de execução e identificar anomalias estruturais de forma mais precisa.

2.3. LLMs na Detecção de Smells

Com a evolução recente das *LLMs*, a capacidade dos modelos de compreender código sem a necessidade de extração explícita de métricas, gera uma possível abordagem nova para de identificação de code smells por LLMs. Blaauwendraad [Blaauwendraad 2025] investigou sistematicamente o potencial do modelo GPT-4o mini para detectar smells e classificar sua severidade. O estudo comparou diferentes estratégias de *prompting*, incluindo *Zero-Shot*, *Few-Shot*, *Role-Based* e *Chain-of-Thought* (CoT). Os resultados indicaram que, embora o *Few-Shot* tenha obtido métricas numéricas ligeiramente superiores, a estratégia *Chain-of-Thought* ofereceu o desempenho mais equilibrado e generalizável, reduzindo falsos positivos ao forçar o modelo a raciocinar passo a passo sobre a estrutura do código. O autor concluiu que, embora promissores para detecção binária, os LLMs ainda enfrentam dificuldades na classificação precisa da severidade dos smells e na detecção de problemas que exigem contexto de acoplamento complexo, como *Feature Envy*. Além disso, estudos como o de Sadik et al. [Sadik et al. 2023] investigaram o uso de modelos baseados em chat sobre código-fonte de maneira geral, indicando que, embora os modelos consigam identificar trechos problemáticos, a formulação e calibração das instruções são cruciais para a consistência e redução de alucinações semânticas.

3. Metodologia

Esta seção detalha os procedimentos adotados no estudo para a avaliação da capacidade de LLMs, executadas em ambiente local, na detecção de *code smells* em códigos JavaScript. O objetivo principal deste estudo é avaliar a capacidade do modelo em verificar a presença de *code smells*, priorizando a privacidade dos dados, bem como comparar

sistematicamente diferentes técnicas de engenharia de prompt. As etapas seguidas e detalhadas nas subseções foram a seleção do *dataset*, sua validação e higienização, a construção do ambiente experimental, a escolha das estratégias de *prompt* a serem utilizadas, a execução de 15 repetições dos experimentos para análise de variabilidade, e a etapa de análise estatística dos dados obtidos.

3.1. Seleção do Dataset

Conforme apontado por Sarafim et al. [Sarafim et al. 2024], existe uma escassez crítica de estudos e, consequentemente, uma ausência de datasets padronizados para JavaScript. Essa lacuna não apenas dificulta a criação de benchmarks comparativos, mas também ignora as particularidades estruturais e dinâmicas do JavaScript que diferem substancialmente do paradigma estático, como, por exemplo, a linguagem Java. Por essa razão foi realizada a escolha do *dataset* proposto por Sarafim et al. [Sarafim et al. 2024], curado e rotulado especificamente para a linguagem JavaScript, permitindo uma amostragem estatística suficiente para a validação dos modelos.

O *dataset* é composto por 840 instâncias, sendo 420 do tipo *God Class* (nível de classe) e 420 do tipo *Long Method* (nível de método) extraídas de oito projetos JavaScript *open-source* populares e hospedados no GitHub, incluindo bibliotecas e ferramentas amplamente utilizadas.

3.2. Validação do dataset

O *dataset* foi particionado de forma automatizada utilizando a biblioteca *scikit-learn* em um script Python. Para preservar a proporção das classes, a amostragem foi estratificada pelo tipo de *code smell*, e fixou-se uma semente aleatória (*random seed*) igual a 42 para garantir a reprodutibilidade. O conjunto de dados original foi dividido em dois subconjuntos: Desenvolvimento (20%) e Teste (80%). O conjunto de desenvolvimento foi empregado para a calibração de formato e testes de prompt (garantindo que o modelo gerasse saídas no formato JSON esperado), enquanto o conjunto de teste foi estritamente reservado para a avaliação de desempenho final das estratégias.

Para garantir a confiabilidade dos experimentos, foi implementada uma etapa de higienização do *dataset*. Antes de submeter as instâncias ao modelo, cada arquivo referenciado foi baixado localmente e processado por um algoritmo de validação automática. O procedimento aplicou os seguintes critérios de exclusão, invalidando as instâncias comprometidas:

1. **Inconsistência de Intervalos e Tipos:** Casos onde os índices de linha inicial (L_{start}) ou final (L_{end}) fossem nulos, não numéricos, ou matematicamente incoerentes (por exemplo, não-positivos ($L \leq 0$) ou com início posterior ao fim ($L_{start} > L_{end}$)).
2. **Violação de Limites Físicos:** Instâncias onde a linha final indicada excedia o número total de linhas do arquivo físico ($L_{end} > L_{total}$), impossibilitando a extração correta do contexto.
3. **Integridade do Arquivo:** Instâncias que, embora listadas no *dataset*, não tiveram o download do arquivo referenciado completado com sucesso.

A Tabela 1 detalha a estatísticas de instâncias descartadas durante a fase de higienização. De um total de 840 instâncias processadas, 244 foram removidas, sendo as

exclusões devidas a falhas de download (50,82%) e violação de limites físicos de linhas (49,18%).

Tabela 1. Distribuição das Instâncias removidas

Critério de Exclusão	God Class	Long Method	Total
Inconsistência de Intervalos e Tipos	0	0	0
Violação de Limites Físicos	92	28	120
Integridade de Download	64	60	124
Total Desabilitado	**156**	**88**	**244**

Após a validação, o conjunto de teste correspondente a 80% das instâncias válidas separadas na etapa anterior apresentou a distribuição detalhada na Tabela 2:

Tabela 2. Distribuição das Instâncias para teste pós validação

Tipo de smell	Presença smells	Sem presença de smell	Total
God Class	54	156	210
Long method	96	173	269
Total smells	**150**	**329**	**479**

3.3. Ambiente Experimental e Modelo

Diferentemente de abordagens fundamentadas em APIs proprietárias baseadas em nuvem, este estudo optou pela execução local dos modelos de linguagem. Esta decisão metodológica visa mitigar riscos de vazamento de dados, garantindo a soberania da informação e assegurando que trechos de código sensíveis, que frequentemente contêm propriedade intelectual, não sejam transmitidos a servidores externos.

A infraestrutura experimental foi orquestrada utilizando o **Ollama** para o gerenciamento e inferência local do modelo, expondo uma API para comunicação. A estruturação e o envio dos *prompts* foram mediados pelo framework **LangChain**, garantindo a reprodutibilidade na interação com o modelo. Para os experimentos, optou-se pelo modelo **gpt-oss-20b**, uma arquitetura *Mixture-of-Experts* (MoE) de pesos abertos desenvolvida pela OpenAI [OpenAI et al. 2025]. Embora modelos abertos altamente consolidados como *DeepSeek-Coder* e *Qwen* fossem alternativas viáveis na literatura, optou-se pela arquitetura MoE do *gpt-oss-20b* devido ao seu balanço favorável entre capacidade de raciocínio profundo e eficiência de memória na execução em hardware local de médio porte. Testes preliminares com modelos densos de menor escala (como 7B e 8B parâmetros) indicaram taxas de erro substancialmente maiores no preenchimento do esquema de saída JSON exigido e menor estabilidade nas etapas de raciocínio da estratégia CoT, o que justificou a seleção de uma arquitetura de maior escala (20B parâmetros). A escolha desse modelo fundamenta-se em três características críticas para a análise de código:

1. **Capacidade de Raciocínio (*Reasoning*) Local:** O *gpt-oss-20b* foi treinado utilizando técnicas para replicar as capacidades de raciocínio profundo de modelos

maiores, permitindo uma análise estrutural de *code smells* com alta precisão sem a necessidade de envio de dados para APIs externas.

2. **Eficiência via *Mixture-of-Experts*:** Com 21 bilhões de parâmetros totais, ele oferece um equilíbrio superior entre latência e desempenho. Isso viabiliza a execução com um custo computacional significativamente menor que modelos densos de tamanho similar.
3. **Alinhamento para Instruções Complexas:** O modelo apresenta otimizações específicas para seguimento de instruções, características essenciais para interpretar as definições rigorosas dos *code smells* e retornar saídas estritamente formatadas em JSON.

Para a execução dos experimentos, a temperatura de inferência foi configurada como $T = 0$, visando o maior determinismo possível por meio de decodificação gananciosa (*greedy decoding*). Os demais parâmetros de amostragem foram mantidos nos valores padrão do Ollama ($top\_p = 0.9$ e $top\_k = 40$), e o tamanho máximo de saída foi limitado em $max\_tokens = 128$, visto que o modelo precisava apenas retornar uma chave JSON simples. Embora a temperatura zero teoricamente anule a variabilidade, reportou-se desvio padrão nas 15 repetições. Essa flutuação decorre de fontes de não-determinismo inerentes à execução local em GPU, tais como o uso de algoritmos de redução paralela não-determinísticos nas bibliotecas CUDA (ex: cuBLAS/cuDNN) e efeitos de ruído de quantização (típicos de modelos executados sob quantização de 4 ou 8 bits no Ollama), que acumulam pequenas variações de ponto flutuante e alteram marginalmente as probabilidades dos tokens em escolhas limiares.

3.4. Estratégias de Engenharia de Prompt

A seleção das estratégias de *Prompt Engineering* segue uma abordagem incremental de complexidade baseada na análise feita por Blaauwendraad [Blaauwendraad 2025]. Utilizou-se Zero-Shot como linha de base, para aferir o conhecimento intrínseco do modelo, Chain-of-Thought (CoT) por ter sido identificado como a técnica mais robusta para generalização e redução de falsos positivos, superando abordagens baseadas apenas em exemplos (Few-Shot). Por fim, Role-Based para investigar se a personificação de um especialista em qualidade de software refina a precisão da detecção ou se induz a uma criticidade excessiva, conforme observado nos experimentos de Blaauwendraad. Os prompts utilizados nesta pesquisa seguem como o padrão abaixo:

- **Zero-Shot:** O modelo é questionado diretamente sobre a presença do *code smell* sem exemplos prévios, conforme a estrutura detalhada na Fig. 2.

Instrução de Formato de Saída (Comum a todas as estratégias)

Tendo analisado o código, forneça sua resposta apenas no formato JSON. O JSON deve conter uma única chave: "presence", com o valor 0 (ausente) ou 1 (presente).

Exemplo de saída (Presente):

```
{
    "presence": 1
}
```

Exemplo de saída (Ausente):

```
{
    "presence": 0
}
```

Figura 1. Instrução de formatação JSON injetada em todos os prompts.

Prompt: Zero-Shot

SYSTEM:
Analise o código-fonte fornecido em busca de algum code smell do tipo {smell_name}. Instruções de Saída: *[Inserir Instrução JSON]*

USER:
Código-Fonte para Análise:

```
```javascript
{code_snippet}
```
```

Figura 2. Estrutura do prompt para a estratégia Zero-Shot.

- **Role-Based:** Atribui ao modelo a persona de um "Especialista em Qualidade de Código", cuja estrutura de prompt correspondente é apresentada na Fig. 3.

Prompt: Role-Based

SYSTEM:
Você é um engenheiro de software sênior e um especialista em qualidade de código. Sua tarefa é analisar o código-fonte fornecido em busca de algum code smell do tipo {`smell_name`}. Não use regras de limiares rígidos, mas sim o contexto semântico e estrutural.

Definição do Code Smell: {`smell_definition`}
Instruções de Saída: *[Inserir Instrução JSON]*

USER:
Código-Fonte para Análise:

```
```javascript
{code_snippet}
```
```

Figura 3. Estrutura do prompt para a estratégia Role-Based.

- **Chain-of-Thought:** O modelo é instruído a raciocinar passo a passo sobre o design do código antes da classificação, seguindo as diretrizes estruturadas na Fig. 4.

Prompt: Chain-of-Thought (CoT)

SYSTEM:
Você é um engenheiro de software sênior e um especialista em qualidade de código. Analise o código-fonte fornecido em busca de algum code smell do tipo {`smell_name`}. Siga estes passos de raciocínio antes de dar sua resposta final:

1. Revise a definição do code smell: {`smell_definition`}
2. Examine o código-fonte linha por linha.
3. Compare as características do código com os sinais do code smell.
4. Formule uma conclusão interna (presente ou ausente).
5. Gere a saída JSON final conforme as instruções, sem incluir seu raciocínio.

Instruções de Saída: *[Inserir Instrução JSON da Fig. 1]*

USER:
Código-Fonte para Análise:

```
```javascript
{code_snippet}
```
```

Figura 4. Estrutura do prompt para a estratégia Chain-of-Thought.

3.5. Métricas de Avaliação

O problema foi modelado como uma tarefa de classificação binária (presença ou ausência do *code smell*). Para garantir a confiabilidade dos resultados, dado o caráter não-determinístico de modelos de linguagem, cada experimento foi repetido 15 vezes.

Os resultados obtidos foram confrontados com o *ground truth* do dataset, sendo reportados em termos de **Média (μ) e Desvio Padrão (σ)**. A avaliação baseou-se nas seguintes métricas:

- **Acurácia:** A proporção global de predições corretas.
- **Precisão:** A capacidade do modelo de não gerar falsos positivos.
- **Revocação (Recall):** A capacidade do modelo de detectar todas as instâncias reais do problema (minimizar falsos negativos).
- **F1-Score:** A média harmônica entre Precisão e Revocação, utilizada como métrica principal para comparar o desempenho equilibrado entre as estratégias.

Além das métricas escalares, a análise incluiu matrizes de confusão geradas normalizadas e matrizes de contagem média para visualizar a distribuição de Verdadeiros Positivos (VP), Verdadeiros Negativos (VN), Falsos Positivos (FP) e Falsos Negativos (FN) por estratégia.

As métricas foram calculadas utilizando as bibliotecas *scikit-learn* e *pandas*, conforme as equações padrão da literatura.

4. Resultados e Discussão

Nesta seção, são apresentados os resultados obtidos a partir da aplicação do modelo *gpt-oss* sobre o dataset. A análise detalha o desempenho quantitativo das métricas (Média e Desvio Padrão) e discute o impacto das diferentes estratégias de engenharia de prompt.

4.1. Análise Quantitativa Geral

A Tabela 3 apresenta os resultados consolidados para as métricas de acurácia, precisão, revocação e F1-score. Os valores representam a média das 15 execuções seguidos pelo desvio padrão ($\mu \pm \sigma$).

Tabela 3. Desempenho dos Prompts na Detecção de Code Smells (Média $\pm$ Desvio Padrão)

Smell	Estratégia	Acurácia	Precisão	Recall	F1-Score
God Class	Zero-Shot	0.685 ± 0.013	0.450 ± 0.010	$\mathbf{0.996 \pm 0.008}$	0.619 ± 0.009
	Role-Based	0.781 ± 0.034	0.547 ± 0.041	0.927 ± 0.027	0.686 ± 0.030
	CoT	0.780 ± 0.027	0.544 ± 0.033	0.938 ± 0.029	$\mathbf{0.688 \pm 0.025}$
Long Method	Zero-Shot	0.861 ± 0.007	0.738 ± 0.009	$\mathbf{0.949 \pm 0.011}$	0.830 ± 0.009
	Role-Based	0.873 ± 0.006	0.776 ± 0.015	0.908 ± 0.020	0.836 ± 0.007
	CoT	$\mathbf{0.879 \pm 0.010}$	$\mathbf{0.782 \pm 0.015}$	0.915 ± 0.012	$\mathbf{0.844 \pm 0.012}$

4.2. Análise de God Class

A detecção de *God Class* demonstrou ser mais complexa para o modelo. A estratégia **Zero-Shot** apresentou um comportamento extremo: obteve um *Recall* muito alto (0.996), indicando que o modelo identificou praticamente todas as instâncias reais do problema.

No entanto, isso veio ao custo de uma precisão baixa (0.450), sugerindo um alto volume de falsos positivos. Essencialmente, sem contexto adicional, o modelo tende a classificar muitas classes normais como problemáticas.

A introdução das estratégias **Role-Based** e **Chain-of-Thought (CoT)** equilibrou significativamente o desempenho:

- A acurácia saltou de ≈ 0.68 (Zero-Shot) para ≈ 0.78 (Role/CoT).
- O F1-score, que balanceia os erros, subiu de 0.619 para 0.688 com o uso de CoT.

Essa melhora de comportamento e a consequente redução de falsos positivos são evidenciadas pelas matrizes de confusão comparativas ilustradas na Fig. 5.

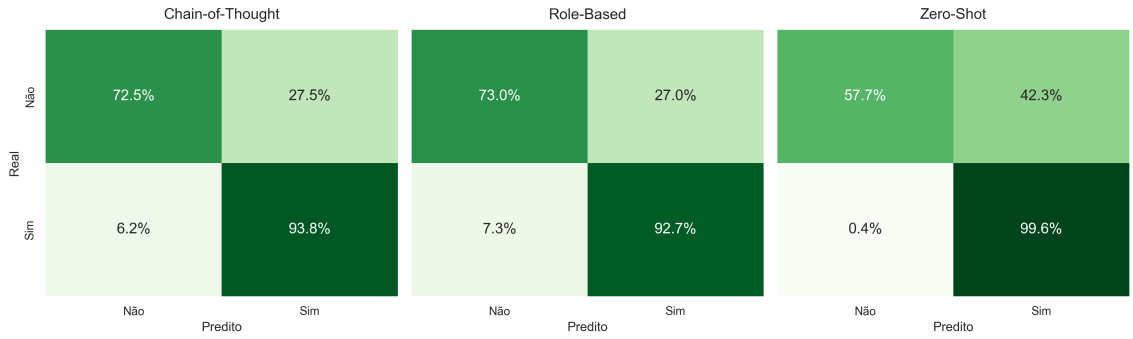

Figura 5. Matrizes de confusão comparativa entre estratégias God Class

Isso demonstra que fornecer instruções de papel ou passos de raciocínio ajuda o LLM a filtrar falsos positivos, diminuindo a tendência do modelo de marcar tudo como smells.

4.3. Análise de Long Method

Para o *Long Method*, um problema de natureza mais estrutural e menos subjetiva, o modelo demonstrou um desempenho superior e mais estável, com F1-scores consistentemente acima de 0.83. Aqui, a estratégia **Chain-of-Thought (CoT)** obteve os melhores resultados globais (F1-score de 0.844 e acurácia de 0.879). Diferente da *God Class*, onde a diferença entre Role-Based e CoT foi marginal, no *Long Method* o raciocínio passo a passo auxiliou levemente na Precisão (0.782 contra 0.738 do Zero-Shot), mantendo um Recall robusto (0.915). Essas dinâmicas de classificação e distribuição de erros e acertos estão detalhadas nas matrizes de confusão comparativas da Fig. 6.

4.4. Comparativo Geral e Significância Estatística

Os baixos valores de desvio padrão (DP) em todas as métricas (geralmente abaixo de 0.03) indicam que o modelo *gpt-oss* é consistente e estável nas suas avaliações. Fica evidente que a complexidade do *code smell* dita a necessidade de engenharia de prompt.

Para validar se as diferenças de desempenho observadas entre as estratégias de prompt são estatisticamente significativas, aplicou-se o teste de postos sinalizados de Wilcoxon (*Wilcoxon signed-rank test*) sobre as F1-scores obtidas nas 15 execuções. Na detecção de *God Class*, a estratégia *Chain-of-Thought* (CoT) apresentou superioridade

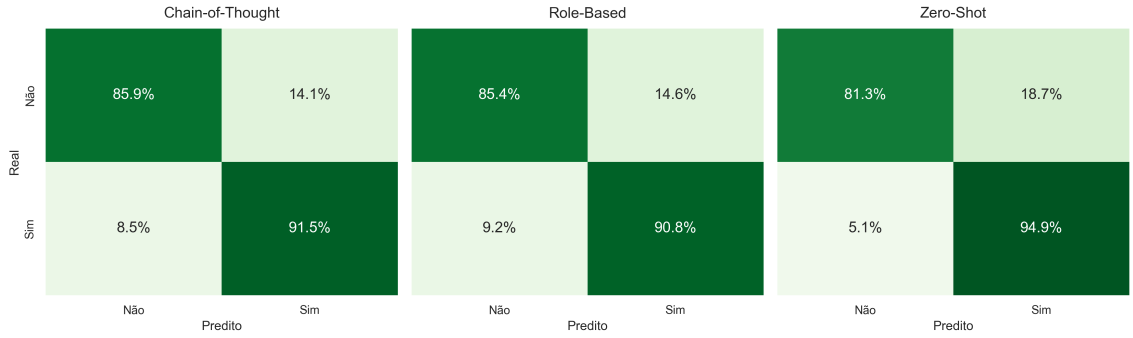

Figura 6. Matrizes de confusão comparativa entre estratégias Long Method

estatisticamente significativa em relação à estratégia básica *Zero-Shot* ($p \approx 0.00006$, com ganho médio de 0.068 no F1-score) e a estratégia *Role-Based* também superou a *Zero-Shot* de forma significativa ($p \approx 0.00006$, ganho médio de 0.067). Entretanto, a diferença entre CoT e *Role-Based* não se mostrou estatisticamente significativa para esse smell ($p \approx 0.679$). Por outro lado, na detecção de *Long Method*, todas as comparações pareadas foram estatisticamente significativas: a estratégia CoT superou significativamente a *Zero-Shot* ($p \approx 0.0067$) e a *Role-Based* ($p \approx 0.0353$), enquanto a *Role-Based* também apresentou superioridade estatística frente à *Zero-Shot* ($p \approx 0.0479$). Esses resultados comprovam empiricamente que o uso de raciocínio passo a passo (CoT) fornece melhorias robustas e mensuráveis para problemas estruturais em código.

A complexidade do *code smell* dita a necessidade de engenharia de prompt. Enquanto o *Long Method* é bem detectado até com *Zero-Shot*, a *God Class* exige estratégias mais elaboradas para evitar a fadiga de alertas causada por falsos positivos, resultados que podem ser atribuídos à natureza do problema:

1. **Long Method:** É uma métrica mais visual e linear (tamanho da função), facilitando a identificação pelo LLM mesmo com janelas de contexto limitadas.
2. **God Class:** Exige uma compreensão semântica profunda das responsabilidades da classe, e a característica dinâmica do JavaScript torna mais complexa essa análise. O modelo local pode ter enfrentado dificuldades em manter o contexto de classes muito extensas, como demonstrado na Figura 7, onde o aumento do tamanho do prompt degrada a performance do modelo local.

5. Conclusão e Trabalhos Futuros

As evidências quantitativas permitem concluir que modelos locais de código aberto são ferramentas promissoras para a análise estática de código, embora apresentem comportamentos distintos dependendo da complexidade semântica do problema:

- **Avaliação das estratégias:** A estratégia *Zero-Shot* demonstrou ser extremamente sensível, alcançando um Recall próximo a 100% na detecção de *God Class*, porém com baixa Precisão (0.45). Isso indica que, sem orientação específica, o modelo tende a ser excessivamente alarmista, classificando classes complexas (mas legítimas) como problemáticas.
- **Eficácia da Engenharia de Prompt:** A aplicação de *Chain-of-Thought* (CoT) e *Role-Based* provou-se essencial para mitigar alucinações e falsos positivos. O CoT, em particular, elevou o F1-score e a precisão em ambos os *smells*,

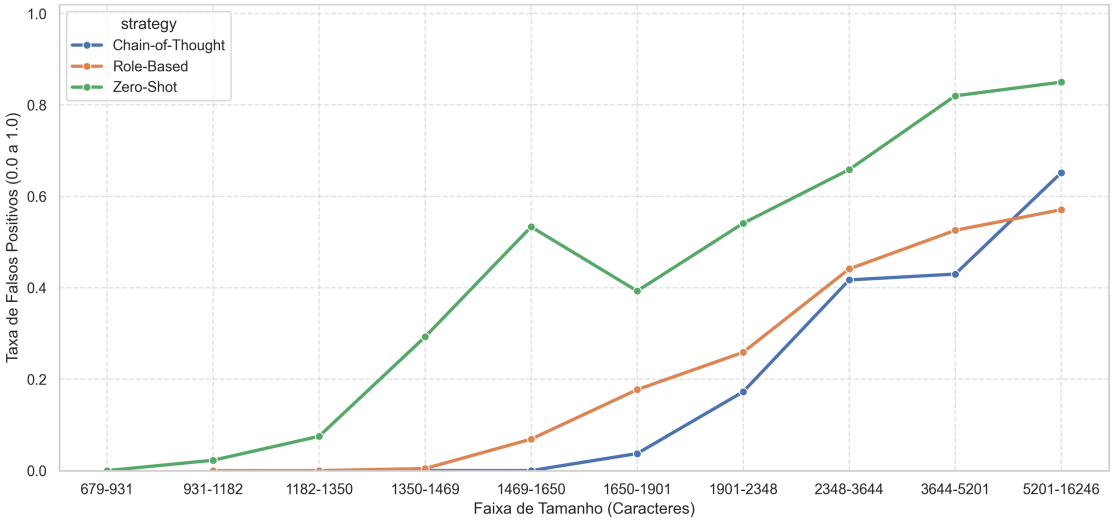

Figura 7. Tendencia de falso positivos com aumento de characters do prompt

confirmando que obrigar o modelo a "raciocinar" passo a passo melhora sua capacidade de discernimento, viabilizando seu uso como ferramenta de triagem preliminar em um assistente de revisão de código mais confiável.

- **Natureza do Smell:** O modelo obteve desempenho consistentemente superior na detecção de *Long Method* em comparação à *God Class*. Isso sugere que a janela de contexto e a capacidade de abstração do modelo local ainda enfrentam desafios para compreender o acoplamento de classes extensas.

Do ponto de vista prático, os resultados validam o uso de LLMs locais como uma alternativa viável para contextos que necessitam de privacidade de dados. Embora modelos proprietários (como GPT-4) possam oferecer maior capacidade de raciocínio, o *gpt-oss* demonstrou que, com a engenharia de prompt adequada, é possível atingir níveis úteis de detecção sem custos de API e sem expor propriedade intelectual a terceiros.

5.1. Limitações e Trabalhos Futuros

A principal limitação observada reside na dificuldade de modelo em manter a coerência em contextos muito longos, o que impactou a precisão na *God Class*.

Como direcionamento para trabalhos futuros, sugere-se:

1. Uma análise das métricas apresentadas no *dataset*, como complexidade, quantidade de linhas etc., sobre os *code smells*, correlacionando com as instâncias que geraram falsos positivos, com o objetivo de identificar quais fatores podem levar o modelo a gerar uma porcentagem maior de falsos positivos ou falsos negativos, possibilitando uma compreensão maior do fluxo de identificação do modelo e uma proposta de uso mais objetiva para o uso de LLMs, neste contexto.
2. Avaliar com uma temperatura maior que 0, possibilitar o modelo a ser mais "criativo"analisando se os resultados serão melhores.
3. Expandir a validação para outros tipos de anomalias, como *Feature Envy* e *Duplicated Code* e análise de severidade.

Referências

Alomari, N., Alazba, A., Aljamaan, H., and Alshayeb, M. (2025). Smellycode++: Multi-label dataset for code smell detection. *Scientific Data*, 12(1):1207.

Arcelli Fontana, F., Mäntylä, M. V., Zanoni, M., and Marino, A. (2016). Comparing and experimenting machine learning techniques for code smell detection. *Empirical Software Engineering*, 21(3):1143–1191.

Blaauwendraad, R. (2025). *Evaluating Large Language Models for Code Smell Detection*. PhD thesis, Radboud University. Bachelor's thesis.

Fard, A. M. and Mesbah, A. (2013). Jsnose: Detecting code smells in javascript applications. In *IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125. IEEE.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Moreira, R. A. F., Braz, L. J. L., Ferreira, F. J., and Amora, M. A. B. (2024). Estudo empírico: detecção de code smells com aprendizado de máquinas. In *Congresso Ibero-Americano em Engenharia de Software (CIbSE)*, pages 301–312. SBC.

OpenAI, Agarwal, S., Ahmad, L., Ai, J., Altman, S., et al. (2025). gpt-oss-120b & gpt-oss-20b model card.

Sadik, A. R., Ceravola, A., Joublin, F., and Patra, J. (2023). Analysis of chatgpt on source code. *arXiv preprint arXiv:2306.00597*.

Sarafim, D. S., Delgado, K. V., and Cordeiro, D. (2024). Detecting code smells in javascript: An annotated dataset for software quality analysis. In *Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 313–322. SBC.