UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

LUÍS ESTEVAM ROSA CHAVES

DETECÇÃO AUTOMATIZADA DE *CODE SMELLS* EM PYTHON UTILIZANDO LLMS MULTIAGENTES

QUIXADÁ

2026

LUÍS ESTEVAM ROSA CHAVES

DETECÇÃO AUTOMATIZADA DE *CODE SMELLS* EM PYTHON UTILIZANDO LLMS MULTIAGENTES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientadora: Profa. Dra. Carla Ilane Moreira Bezerra.

QUIXADÁ

2026

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C439d Chaves, Luís Estevam Rosa.
Detecção automatizada de code smells em Python utilizando LLMs multiagentes / Luís Estevam Rosa Chaves. – 2026.
99 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2026.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. Code Smells. 2. Sistema Multiagente. 3. Engenharia de Prompts. 4. Qualidade de código. I. Título.
CDD 005.1

LUÍS ESTEVAM ROSA CHAVES

DETECÇÃO AUTOMATIZADA DE *CODE SMELLS* EM PYTHON UTILIZANDO LLMS MULTIAGENTES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em: 22 de Janeiro de 2026.

BANCA EXAMINADORA

Profa. Dra. Carla Ilane Moreira Bezerra (Orientadora)
Universidade Federal do Ceará (UFC)

Prof. Dr. Rohit Gheyi
Universidade Federal de Campina Grande (UFCG)

Prof. Dr. Regis Pires Magalhães
Universidade Federal do Ceará (UFC)

A todos que lutaram, que choraram, que pensaram em desistir, mas encontraram forças para se levantar mais uma vez. A todos que me abraçaram e me confortaram nos momentos de alegria e de dificuldade.

AGRADECIMENTOS

Agradeço primeiramente a Deus, que escutou as minhas orações, cuidou de mim, me ensinou a persistir e a não fraquejar, estando comigo em todos os momentos.

Aos meus pais, Manoel Maria da Silva Chaves (*in memoriam*) e Antonia Carvalho Rosa Chaves, que cuidaram de mim e correram antes de mim para que eu pudesse andar, derramando cada gota de suor para que eu tivesse sombra.

Aos meus avós (*in memoriam*), que foram presentes na minha vida, me protegeram e me ensinaram o que é carinho e paz, e sempre faziam doce de leite para mim.

À minha família, representada por minha tia Raimunda Rosa (*in memoriam*), que simboliza o que significa ser a nossa família: cuidava, brigava, ria, protegia. Assim somos nós, unidos (nem sempre), mas é isso que é ser família.

Aos meus amigos, que estiveram ao meu lado todo esse tempo. Rimos, lutamos, choramos, sofremos e, no final, vencemos. O fardo foi mais leve com vocês. Muito obrigado, de verdade.

À minha igreja, que orou por mim.

À minha orientadora, Prof.ª Dr.ª Carla Ilane Moreira Bezerra, pelos ensinamentos, pela orientação e por me motivar a melhorar sempre.

À Universidade Federal do Ceará por todos esses anos e a todos os professores que contribuíram para o meu desenvolvimento acadêmico.

"O perigo real não é que computadores
comecem a pensar como homens, mas que
homens comecem a pensar como computadores."
(Sydney J. Harris (1917–1986))

RESUMO

A detecção de *code smells* é uma atividade essencial para garantir a qualidade e a manutenibilidade de sistemas de software. Ferramentas tradicionais de análise estática, embora amplamente utilizadas, apresentam limitações como alta taxa de falsos positivos e incapacidade de considerar o contexto específico do projeto. Este trabalho investiga a aplicação de sistemas multiagentes baseados em Grandes Modelos de Linguagem (LLMs) para detecção automatizada de *code smells* de implementação em projetos *Python*. A proposta utiliza uma arquitetura com um agente *Supervisor* coordenando onze agentes especializados, cada um dedicado à detecção de um tipo específico de *code smell*: *Long Method*, *Complex Method*, *Complex Conditional*, *Long Parameter List*, *Long Statement*, *Long Identifier*, *Magic Number*, *Empty Catch Block*, *Missing Default*, *Long Lambda Function* e *Long Message Chain*. Foram avaliados três modelos de linguagem (*Claude Sonnet 4.5*, *GPT-4o-mini* e *DeepSeek V3.2*) e comparados dois tipos de *prompts*: elaborados e simples. Os experimentos foram conduzidos em um conjunto de 20 arquivos *Python* totalizando 8.463 linhas de código, validados contra um padrão de referência com 411 instâncias de *smells*. Os resultados demonstram que o sistema alcançou *F1-Score* entre 61,62% e 65,12%, com o *GPT-4o-mini* apresentando o melhor equilíbrio entre qualidade e custo. A análise revelou que *prompts* elaborados proporcionam ganho de 4,33 pontos percentuais no *F1-Score*, embora a diferença não seja estatisticamente significativa. Como contribuição adicional, foi desenvolvida uma extensão para *Visual Studio Code* que integra a detecção de *smells* ao fluxo de trabalho do desenvolvedor.

Palavras-chave: *code smells*; modelos de linguagem; sistemas multiagente; análise estática; qualidade de código; engenharia de *prompts*.

ABSTRACT

Code smell detection is an essential activity for ensuring software quality and maintainability. Traditional static analysis tools, although widely used, have limitations such as high false positive rates and inability to consider project-specific context. This work investigates the application of multi-agent systems based on Large Language Models (LLMs) for automated detection of implementation code smells in Python projects. The proposed system uses an architecture with a Supervisor agent coordinating eleven specialized agents, each dedicated to detecting a specific type of smell: Long Method, Complex Method, Complex Conditional, Long Parameter List, Long Statement, Long Identifier, Magic Number, Empty Catch Block, Missing Default, Long Lambda Function, and Long Message Chain. Three language models were evaluated (Claude Sonnet 4.5, GPT-4o-mini, and DeepSeek V3.2) and two types of prompts were compared: elaborate and simple. Experiments were conducted on a dataset of 20 Python files totaling 8,463 lines of code, validated against a reference standard with 411 smell instances. Results demonstrate that the system achieved F1-Score between 61.62% and 65.12%, with GPT-4o-mini presenting the best balance between quality and cost. The analysis revealed that elaborate prompts provide a 4.33 percentage point gain in F1-Score, although the difference is not statistically significant. As an additional contribution, a Visual Studio Code extension was developed that integrates smell detection into the developer workflow.

Keywords: code smells; language models; multi-agent systems; static analysis; code quality; prompt engineering.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

1 INTRODUÇÃO

A revisão de código constitui etapa fundamental para garantir a qualidade e a manutenibilidade de sistemas de *software*. Seu propósito é identificar falhas, inconsistências e más práticas de programação que, se não tratadas, comprometem o ciclo de vida do projeto (Rasheed *et al.*, 2024; Tandon *et al.*, 2024; Almeida *et al.*, 2024). Entre os principais alvos dessas análises estão os *code smells*, sintomas de deficiências estruturais no código. Métodos extensos, classes sobrecarregadas de responsabilidades e listas de parâmetros excessivas são exemplos recorrentes. Tais padrões não impedem a execução do software, mas deterioram sua legibilidade, extensibilidade e testabilidade ao longo do tempo (Tandon *et al.*, 2024; Zhang *et al.*, 2023; Almeida *et al.*, 2024).

Ferramentas de análise estática como *Pylint*, *Designite* e *SonarQube* automatizam parte desse trabalho, oferecendo verificações rápidas e sistemáticas de padrões problemáticos. Contudo, a prática revela limitações consideráveis: a alta incidência de falsos positivos frequentemente sobrecarrega desenvolvedores com alertas irrelevantes, enquanto as sugestões de correção tendem a ignorar nuances do domínio ou convenções específicas do projeto (Sadik; Govind, 2025; Saavedra; Ferreira, 2022). Em casos extremos, essas limitações induzem refatorações desnecessárias ou, pior, a introdução de novos defeitos.

Nesse contexto, os *Large Language Models* (Modelos de Linguagem de Grande Escala)s (LLMs) (*Large Language Models* — Modelos de Linguagem de Grande Escala) apresentam-se como alternativa promissora. Sua capacidade de interpretação semântica permite analisar o código-fonte de forma diferente das abordagens baseadas em regras fixas: em vez de comparar métricas com limiares predefinidos, os LLMs consideram o contexto mais amplo, incluindo padrões de desenvolvimento, requisitos do projeto e convenções da equipe. Essa flexibilidade viabiliza diagnósticos mais precisos e sugestões de refatoração alinhadas com boas práticas (Silva *et al.*, 2024; Zhang *et al.*, 2023). A capacidade de processar linguagem natural constitui vantagem adicional, pois torna as recomendações mais compreensíveis para os desenvolvedores.

Soluções baseadas em LLMs monolíticos, no entanto, enfrentam dificuldade em equilibrar cobertura e precisão. Estudos como o de (Silva *et al.*, 2024) já tentaram mitigar isso através de *prompts* mais longos e detalhados, mas os resultados indicam que o modelo tende a "se perder"em contextos extensos (fenômeno *lost-in-the-middle*). Particularmente em sistemas complexos onde múltiplos tipos de *code smells* coexistem (Ishibashi; Nishimura, 2024; Tran *et*

al., 2025). Uma abordagem para contornar essa limitação consiste em adotar arquiteturas de sistemas multiagente. A divisão de responsabilidades entre agentes especializados, cada um dedicado a detectar uma categoria específica de *smell*, possibilita o uso de *prompts* elaborados, com definições precisas, exemplos demonstrativos e critérios de validação direcionados. Essa especialização tende a reduzir ruído nas detecções e melhorar a precisão (Chen *et al.*, 2023; Nunez *et al.*, 2024; Pei *et al.*, 2024).

Diante desse cenário, este trabalho investiga como agentes especializados, coordenados em uma arquitetura multiagente, podem aprimorar a detecção de *code smells* de implementação em projetos *Python*. A abordagem proposta combina técnicas de engenharia de *prompts* com a especialização de agentes, visando superar as limitações tanto das ferramentas tradicionais quanto das soluções baseadas em LLMs monolíticos.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolver e avaliar uma arquitetura multiagente com agentes especializados para aprimorar a detecção automatizada de *code smells* de implementação em projetos *Python*, comparando sua eficácia entre diferentes LLMs e analisando o impacto da engenharia de *prompts* no desempenho do sistema.

1.1.2 Objetivos Específicos

a) Avaliar a eficácia geral do sistema multiagente na detecção de *code smells* de implementação, mensurando precisão, *recall* e *F1-Score* contra um padrão de referência.
b) Analisar o impacto da qualidade dos *prompts* na precisão das detecções, comparando o desempenho entre *prompts* elaborados (com definições, exemplos e critérios de validação) e *prompts* simples.
c) Comparar o desempenho de diferentes modelos de linguagem (*Claude Sonnet 4.5*, *GPT-4o-mini* e *DeepSeek V3.2*) na tarefa de detecção de *code smells*, identificando forças e limitações de cada modelo.
d) Avaliar a contribuição individual de cada agente especializado, identificando quais categorias de *smells* são detectadas com maior precisão e quais apresentam desafios para o sistema.

e) Mensurar e comparar a eficiência operacional entre os diferentes modelos, considerando o tempo de execução, consumo de *tokens* e custos computacionais, estabelecendo uma análise de custo-benefício.

1.2 Questões de Pesquisa

Com base nos objetivos propostos, este trabalho busca responder às seguintes questões de pesquisa:

1. **QP1:** Qual a eficácia do sistema multiagente na detecção de *code smells* de implementação em projetos *Python*, em termos de precisão, *recall* e *F1-Score*?
2. **QP2:** Qual o impacto da qualidade dos *prompts* (elaborados vs. simples) na precisão das detecções realizadas pelo sistema multiagente?
3. **QP3:** Como diferentes modelos de linguagem (*Claude Sonnet 4.5*, *GPT-4o-mini*, *DeepSeek V3.2*) se comparam em termos de desempenho na detecção de *code smells*?
4. **QP4:** Qual a contribuição individual de cada agente especializado para o desempenho geral do sistema?
5. **QP5:**Qual a relação custo-benefício entre os diferentes modelos, considerando tempo de execução, consumo de *tokens* e custos computacionais?

1.3 Estrutura do Trabalho

Este trabalho está organizado em sete capítulos, além dos elementos pré-textuais e pós-textuais. O Capítulo 1 apresenta o contexto e a motivação do trabalho, os objetivos e as questões de pesquisa que orientam a investigação.

O Capítulo 2 aborda os conceitos fundamentais que sustentam a pesquisa, incluindo *code smells*, sistemas multiagente com LLMs e técnicas de engenharia de *prompts*.

O Capítulo 3 apresenta uma revisão de trabalhos que investigam a detecção de *code smells* utilizando LLMs, estabelecendo o posicionamento deste trabalho no contexto da literatura existente.

O Capítulo 4 (Sistema Multiagente para detecção de code smells) descreve a arquitetura desenvolvida, detalhando o escopo da análise, a coordenação dos agentes especializados e as técnicas de engenharia de *prompts* aplicadas.

O Capítulo 5 apresenta os procedimentos metodológicos adotados, incluindo a

seleção da amostra, o desenho experimental e as métricas utilizadas para avaliar o sistema.

O Capítulo 6 apresenta os resultados experimentais, respondendo às cinco questões de pesquisa propostas e analisando o desempenho do sistema sob diferentes perspectivas.

Por fim, o Capítulo 7 sintetiza os principais achados, discute as contribuições do trabalho, apresenta as limitações identificadas e sugere direções para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta os conceitos que sustentam a pesquisa. A seção 2.1 introduz os *code smells* (Fowler *et al.*, 1999), discutindo suas implicações para a qualidade do *software* e as abordagens existentes para sua detecção. Na seção 2.2, são explorados os sistemas multiagente com LLMs (Chen *et al.*, 2023; Wu *et al.*, 2023), abordando a arquitetura, os benefícios da especialização de agentes e as tecnologias de orquestração como *LangChain* e *LangGraph* (LangChain, 2024). Por fim, a seção 2.3 trata dos fundamentos da engenharia de *prompts* (White *et al.*, 2023), com ênfase em técnicas como *few-shot learning* (Brown *et al.*, 2020), *chain-of-thought* (Wei *et al.*, 2022) e *structured output*.

2.1 *Code Smells*

O termo *code smell* surgiu na literatura de engenharia de software no final da década de 1990 e ganhou notoriedade com a publicação de *Refactoring: Improving the Design of Existing Code* (Fowler *et al.*, 1999). Trata-se de sintomas de problemas de projeto no código-fonte: não impedem a execução do software, mas sinalizam trechos com baixa legibilidade e alta complexidade (Tandon *et al.*, 2024; Abdou; Ramadan, 2022). Com o tempo, esses indicadores contribuem para o acúmulo de dívida técnica, dificultando tanto a manutenção quanto a evolução do sistema.

Os detectores tradicionais de *smells* baseiam-se em regras heurísticas que comparam métricas estáticas, como complexidade ciclomática, acoplamento e número de métodos, com limiares predefinidos. Essa abordagem, embora direta, frequentemente resulta em altas taxas de falsos positivos e exige ajustes manuais caso a caso (Alazba; Aljamaan, 2021; Silva *et al.*, 2024). Para contornar tais limitações, pesquisadores têm proposto técnicas de *machine learning* e *deep learning*. Essas abordagens buscam eliminar a dependência de limiares fixos e reduzir falsos positivos por meio de *ensembles*, modelos Bi-LSTM e GRU, além de estratégias de balanceamento de dados (Azeem *et al.*, 2019; Fontana *et al.*, 2016).

2.1.1 Taxonomia de Code Smells

A literatura organiza os *code smells* em categorias distintas (Mäntylä *et al.*, 2003). Entre elas, os *implementation smells* merecem destaque por se manifestarem no nível de métodos e funções individuais, a granularidade mais fina de análise. Identificar com precisão esses

smells é relevante para orientar esforços de refatoração e preservar a qualidade arquitetural do sistema (Lacerda *et al.*, 2020; Palomba *et al.*, 2018). A Tabela 1 sumariza os principais *smells* de implementação documentados na literatura.

Tabela 1 – Taxonomia dos *implementation smells*

Code Smell	**Descrição**	**Referência**
Long Method	Métodos com número extenso de linhas de código ou declarações, prejudicando compreensão e teste.	Fowler *et al.* (1999), Martin (2008)
Long Parameter List	Funções com número excessivo de parâmetros, tornando chamadas complexas e propensas a erros.	Fowler *et al.* (1999), Martin (2008)
Complex Method	Métodos com alta complexidade ciclomática e múltiplas estruturas condicionais aninhadas.	Fowler *et al.* (1999)
Complex Conditional	Expressões condicionais com mais de dois operadores lógicos, dificultando compreensão e manutenção.	Fowler (2018)
Long Statement	Linhas de código longas (acima de 120 caracteres) que dificultam leitura e compreensão.	Martin (2008)
Long Identifier	Identificadores longos que prejudicam a legibilidade quando ultrapassam limites razoáveis.	Sharma (2025)
Magic Number	Valores literais numéricos sem explicação ou atribuição a constantes nomeadas.	Martin (2008)
Long Message Chain	Cadeia de chamadas de métodos que viola a Lei de Demeter.	Fowler *et al.* (1999), Lieberherr e Holland (1989)
Empty Catch Block	Blocos de exceção vazios que suprimem erros silenciosamente.	Sharma (2025)
Missing Default	Ausência de cláusula *default* em estruturas condicionais.	Sharma (2025)
Long Lambda Function	Funções lambda que excedem o tamanho recomendado para expressões *inline*.	Chen *et al.* (2016)

Fonte: Elaborado pelo autor com base na literatura.

2.1.2 Exemplo de Code Smell

Código-fonte 1 – Exemplo de função com múltiplos *code smells*.

```
1 def processar_pedido(cliente_id, itens, desconto, taxa_entrega,
2                       endereco, cidade, estado, cep, telefone):
3     # Long Parameter List: 9 parametros
```

```
4
5       total = 0
6       for item in itens:
7           preco = item['preco'] * item['quantidade']
8           if item['categoria'] == 1:  # Magic Number
9               preco = preco * 0.9  # Magic Number
10          elif item['categoria'] == 2:
11              preco = preco * 0.85
12          total = total + preco
13
14      # Long Message Chain
15      taxa = cliente.conta.configuracoes.preferencias.get_taxa()
16
17      if desconto > 0:
18          total = total - desconto
19      total = total + taxa_entrega + 5.99  # Magic Number
20
21      # ... mais 50 linhas de codigo ...
22      return total
```

A Código-fonte 1 exemplifica múltiplos *code smells* em uma única função: (1) *long parameter list* com 9 parâmetros; (2) *magic numbers* como `0.9`, `0.85` e `5.99` sem constantes nomeadas; e (3) *long message chain* violando a Lei de Demeter. Essa concentração de problemas compromete a manutenibilidade, tornando a função difícil de testar e modificar.

2.2 Sistemas Multiagente com LLMs

Sistemas multiagentes baseados em LLMs representam uma abordagem promissora para o desenvolvimento de sistemas autônomos inteligentes, ao combinar raciocínio, planejamento e interação cooperativa em tarefas complexas (Li *et al.*, 2024). A combinação é interessante porque une o processamento paralelo e especializado de múltiplos agentes ao entendimento contextual dos modelos de linguagem. Na prática, essas arquiteturas costumam empregar *frameworks* como *LangGraph* para orquestrar os fluxos de execução, com implementações em Python para gerenciar a comunicação entre agentes (Pelluru, 2025).

Nesse paradigma, cada agente assume um papel funcional bem definido, voltado à execução de uma subtarefa específica dentro do sistema. Essa divisão de responsabilidades favorece o uso de *prompts* especializados, formulados com instruções claras, exemplos contextualizados e critérios explícitos de validação, o que contribui para maior precisão e previsibilidade

dos resultados (Wu *et al.*, 2023). Além disso, a presença de múltiplos agentes introduz um mecanismo natural de verificação cruzada, no qual inconsistências, falhas ou vieses individuais podem ser identificados e mitigados por meio da interação entre agentes especializados (Tran *et al.*, 2025).

Para que as decisões sejam confiáveis, faz-se necessário adotar estratégias de coordenação robustas. Protocolos colaborativos podem selecionar automaticamente a interpretação mais estável ou acionar mecanismos de arbitragem quando houver impasse (Tran *et al.*, 2025). O diferencial dessa abordagem reside justamente no comportamento colaborativo: o desempenho do conjunto tende a superar a soma das partes individuais, distinguindo-se tanto de abordagens monolíticas quanto do uso isolado de LLMs (Chen *et al.*, 2023; Wu *et al.*, 2023).

2.2.1 LangChain e LangGraph

O desenvolvimento de aplicações complexas baseadas em LLMs foi simplificado pelo surgimento de *frameworks* como o *LangChain*. O *LangChain* oferece abstrações e componentes modulares que permitem a construção de *pipelines*, facilitando a integração de LLMs com fontes de dados externas e a criação de cadeias de execução sequenciais (*chains*) (Jeong *et al.*, 2024).

Sistemas multiagente que exigem fluxos de controle mais sofisticados (ciclos, ramificações condicionais e operações paralelas) encontram limitações nas *chains* tradicionais. O *LangGraph*, extensão do *LangChain*, permite modelar a lógica de aplicações multiagente como um grafo de estados (LangChain, 2024). Nessa abordagem, os agentes são representados como nós (*nodes*) e as transições como arestas (*edges*). O estado do grafo é passado entre os nós, permitindo que cada agente atue com base no trabalho realizado pelos outros.

2.2.2 Padrão Supervisor

Uma arquitetura comum em sistemas multiagente é o padrão *supervisor*, que opera com estrutura hierárquica e coordenada (Microsoft, 2024). Nesse modelo, um agente orquestrador (o Supervisor) recebe as requisições, delega aos agentes especialistas apropriados e consolida os resultados em uma resposta final estruturada.

A função do Supervisor não é executar as tarefas diretamente, mas coordenar o fluxo de trabalho entre os agentes especialistas. Cada agente especialista é dotado de um *prompt* elaborado contendo definições, exemplos e critérios de validação específicos para sua área. Essa arquitetura permite escalabilidade, pois novos agentes podem ser adicionados sem modificar a

lógica central do sistema.

2.3 Engenharia de *Prompts*

A engenharia de *prompts* consiste em projetar e otimizar as instruções fornecidas aos LLMs para obter respostas mais precisas e relevantes (Liu *et al.*, 2023). Trata-se de um conjunto de procedimentos para a formulação e programação de instruções destinadas a personalizar as saídas e interações de LLMs, configurando-se como uma habilidade essencial para garantir a eficácia da comunicação e da utilização dessas ferramentas (Marvin *et al.*, 2024).

Ao contrário do treinamento tradicional de modelos de *machine learning*, essa técnica envolve projetar estrategicamente instruções específicas para tarefas, orientando a saída do modelo sem alterar os parâmetros (Sahoo *et al.*, 2024). Desse modo, a engenharia de *prompt* emerge como um mecanismo fundamental para explorar de forma eficaz as capacidades dos modelos de linguagem, influenciando diretamente tanto o processo de aprendizagem quanto a qualidade e a coerência das respostas geradas (Cain, 2024).

A qualidade do *prompt* impacta diretamente o desempenho obtido. Estudos indicam que um *prompt* eficaz deve definir claramente a tarefa, estabelecer critérios objetivos, fornecer exemplos do comportamento esperado e especificar o formato de saída desejado (White *et al.*, 2023; Reynolds; McDonell, 2021).

2.3.1 Few-Shot Learning

Few-shot learning consiste em fornecer ao modelo alguns exemplos demonstrativos dentro do próprio *prompt*, de modo que ele aprenda o padrão desejado a partir de poucos casos (Brown *et al.*, 2020). A técnica tipicamente inclui exemplos positivos, que demonstram o comportamento esperado, e exemplos negativos, que ilustram o que não deve ser considerado.

A inclusão de exemplos negativos mostra-se particularmente útil para reduzir falsos positivos. Ao apresentar casos que parecem similares mas têm naturezas distintas, auxilia-se o modelo a calibrar melhor suas detecções.

2.3.2 Chain-of-Thought

A técnica de *chain-of-thought* (CoT) encoraja o modelo a explicitar seu processo de raciocínio passo a passo antes de apresentar a resposta final (Wei *et al.*, 2022). Em vez de

solicitar uma resposta direta, o *prompt* instrui o modelo a seguir um processo estruturado de análise.

Essa abordagem tende a melhorar a precisão porque força o modelo a realizar análises intermediárias verificáveis. Com isso, reduzem-se erros de raciocínio e torna-se mais fácil identificar inconsistências nas respostas.

2.3.3 Structured Output

Structured output refere-se à capacidade dos LLMs de gerar respostas em formatos estruturados predefinidos, como JSON, seguindo esquemas específicos (OpenAI, 2024). Essa capacidade é fundamental para integrar LLMs em *pipelines* automatizados, pois garante que as respostas possam ser parseadas e processadas programaticamente.

Na prática, utilizam-se *schemas* de validação, como *Pydantic* em Python, para definir a estrutura esperada das respostas. O modo `json_mode` disponível nos LLMs modernos força o modelo a retornar exclusivamente JSON válido, eliminando problemas de parsing e assegurando consistência.

Vale ressaltar que essas técnicas não são mutuamente exclusivas. A integração de exemplos demonstrativos (*few-shot learning*) com instruções de raciocínio estruturado (*chain-of-thought*) e especificação de formato de saída (*structured output*) resulta em *prompts* mais robustos. Essa combinação é particularmente relevante em tarefas de análise de código, onde tanto a precisão na identificação de padrões quanto a consistência das respostas constituem requisitos essenciais.

3 TRABALHOS RELACIONADOS

Este capítulo revisa trabalhos que abordam a detecção automatizada de *code smells* utilizando LLMs. A análise está organizada em quatro eixos: (i) o paradigma da solução, contrastando ferramentas estáticas, LLMs monolíticos e sistemas *multiagente*; (ii) o impacto da engenharia de *prompts* na qualidade das detecções; (iii) a comparação de desempenho entre diferentes modelos; e (iv) a análise de custo-efetividade das abordagens propostas.

3.1 *Detecting Code Smells using ChatGPT: Initial Insights*

O estudo de Silva *et al.* (2024) investiga a eficácia do *ChatGPT* na detecção de *code smells* em projetos *Java*. Os autores utilizaram um conjunto de dados industrial com 14.739 instâncias de quatro *smells* (*Blob*, *Data Class*, *Feature Envy* e *Long Method*), categorizados em três níveis de severidade.

A metodologia avaliou o *ChatGPT* 3.5-Turbo sob duas estratégias de *prompt*: uma genérica, que solicitava identificação de *smells* sem especificá-los, e outra detalhada, que listava explicitamente os tipos de interesse. Os resultados demonstraram que a especificidade do *prompt* aumentou em 2,54 vezes a probabilidade de uma resposta correta. O modelo também apresentou maior eficiência na detecção de *smells* de severidade crítica.

O trabalho de Silva *et al.* (2024) é relevante para esta pesquisa por demonstrar empiricamente que instruções detalhadas melhoram a precisão das detecções. Essa descoberta fundamenta a premissa de que a especialização, implementada neste trabalho por meio de agentes distintos com *prompts* elaborados, é fator determinante para análises mais eficazes.

3.2 *Benchmarking LLM for Code Smells Detection: OpenAI GPT-4.0 vs DeepSeek-V3*

Sadik e Govind (2025) propõem uma metodologia para comparar LLMs na detecção de *code smells* em contexto multilíngue. Os autores criaram um conjunto de dados customizado com *smells* conhecidos em quatro linguagens (*Java*, *Python*, *JavaScript* e C++), avaliando *GPT-4.0* e *DeepSeek-V3* por meio de *prompts* padronizados.

Os resultados demonstraram superioridade do *GPT-4.0* em precisão (0.79 contra 0.42), gerando menos falsos positivos. Ambos os modelos apresentaram baixo *recall*, falhando em detectar parte considerável dos *smells* reais. A análise de custo-benefício indicou que, apesar do custo superior, o *GPT-4.0* justifica o investimento pela qualidade das respostas.

Este trabalho compartilha com Sadik e Govind (2025) o interesse na comparação entre modelos e na análise de custo-eficiência. A diferença reside na abordagem: enquanto o *benchmark* citado avalia LLMs monolíticos, esta pesquisa investiga se uma arquitetura *multiagente* pode otimizar a capacidade de detecção por meio da especialização e colaboração entre agentes.

3.3 *Large Language Models for Multilingual Vulnerability Detection*

Shu *et al.* (2025) realizaram estudo empírico sobre detecção de vulnerabilidades em contexto multilíngue, avaliando *Pretrained Language Models* (PLMs) e LLMs. A pesquisa utilizou o *corpus REEF*, que abrange sete linguagens de programação, com avaliação em dois níveis de granularidade: função e linha.

Os resultados indicaram que o *GPT-4o*, combinado com técnicas de *instruction tuning* e *few-shot learning*, alcançou *F1-score* de 0.6641 no nível de linha. O estudo concluiu que LLMs são promissores para a tarefa, ressaltando que o tamanho do modelo não é fator decisivo para o desempenho.

Embora o foco de Shu *et al.* (2025) seja vulnerabilidades e não *code smells*, o trabalho oferece evidências sobre a eficácia de técnicas de engenharia de *prompts*, especialmente *few-shot learning*, para tarefas de análise de código. Essas técnicas são incorporadas na arquitetura *multiagente* proposta nesta pesquisa.

3.4 *Boosting Vulnerability Detection via Curriculum Preference Optimization*

Wen *et al.* (2025) propõem o *ReVD*, uma estrutura que utiliza síntese de dados de raciocínio e otimização de preferência para detecção de vulnerabilidades. O trabalho aborda duas limitações dos LLMs: a ausência de dados de raciocínio detalhados sobre vulnerabilidades e o foco em representações semânticas em detrimento do raciocínio explícito.

Nos conjuntos *PrimeVul* e *SVEN*, o *ReVD* alcançou melhoria de 12,24% a 22,77% na acurácia. Integrado ao *Qwen2.5-Coder-7B-Instruct*, superou o *GPT-4* em todas as faixas de comprimento de *token*.

O trabalho de Wen *et al.* (2025) demonstra a importância de estruturas de raciocínio para análise de código. Essa perspectiva é incorporada nesta pesquisa por meio da técnica *Chain-of-Thought* (Cadeia de Pensamento) (CoT), que orienta os agentes a explicitar seu processo de análise antes de apresentar conclusões.

3.5 Análise Comparativa

O Quadro 1 sintetiza as características dos trabalhos revisados, permitindo identificar as contribuições específicas de cada estudo e posicionar a presente pesquisa no contexto da literatura.

Quadro 1 – Comparativo entre trabalhos relacionados

Característica	**Silva *et al.* (2024)**	**Sadik e Govind (2025)**	**Shu *et al.* (2025)**	**Wen *et al.* (2025)**	**Este Trabalho**
Objeto de Estudo	*Code smells* em *Java*	*Code smells* multilíngue	Vulnerabilidades multilíngue	Vulnerabilidades em *C/C++*	*Code smells* em *Python*
Arquitetura	*LLM* monolítico	*LLM* monolítico	*LLM* monolítico	*LLM* com *fine-tuning*	Sistema *multiagente*
Modelos Avaliados	*ChatGPT 3.5-Turbo*	*GPT-4.0*, *DeepSeek-V3*	*GPT-4o*, PLMs diversos	*Qwen2.5-Coder*, *GPT-4*	*Claude Sonnet 4.5*, *GPT-4o-mini*, *DeepSeek V3.2*
Técnicas de *Prompt*	Genérico vs. detalhado	Padronizado	*Few-shot*, *instruction tuning*	Síntese de raciocínio	*Few-shot*, *CoT*, saída estruturada
Métricas	Precisão, severidade	Precisão, *Recall*, *F1*	*F1-score* por granularidade	Acurácia	Precisão, *Recall*, *F1*, custo/*token*
Análise de Custo	Não	Sim	Parcial	Não	Sim

Fonte: Elaborado pelo autor.

A análise do Quadro 1 revela que os trabalhos existentes concentram-se em avaliar LLMs de forma isolada, seja comparando modelos diferentes (Sadik; Govind, 2025; Shu *et al.*, 2025) ou investigando o impacto de estratégias de *prompt* (Silva *et al.*, 2024). Nenhum dos estudos revisados explora arquiteturas *multiagente* para detecção de *code smells*.

Este trabalho diferencia-se ao propor uma arquitetura onde agentes especializados colaboram sob coordenação de um padrão *supervisor*. Além disso, combina múltiplas técnicas de engenharia de *prompts* (*few-shot learning*, CoT e saída estruturada) e realiza análise comparativa entre três modelos de diferentes faixas de custo, permitindo avaliar a relação entre investimento e qualidade das detecções.

4 SISTEMA MULTIAGENTE PARA DETECÇÃO DE CODE SMELLS

Este capítulo explora a arquitetura multiagente desenvolvida para este estudo. O sistema opera sob o padrão *supervisor*, uma arquitetura hierárquica onde um agente coordenador central, o *CodeSmellSupervisor*, recebe o código-fonte e gerencia o fluxo de trabalho. Este *supervisor* é responsável por validar o tamanho do arquivo, numerar as linhas para facilitar a localização e distribuir o código para onze agentes especializados, cada um dedicado exclusivamente à detecção de um tipo específico de implementation *smell* por meio de *prompts* otimizados e técnicas de saída estruturada via *Pydantic*. O processo ocorre preferencialmente em execução paralela para reduzir a latência, permitindo que cada agente consulte o LLM de forma focada, evitando a degradação de desempenho. Ao final, o *supervisor* coleta as detecções, aplica filtros pós-processamento para reduzir falsos positivos e consolida os resultados em um arquivo *JSON* que integra as métricas de detecção e consumo de *tokens*.

Para detalhar a arquitetura e a funcionalidade do sistema desenvolvido, as seções seguintes descrevem os pilares fundamentais da solução: a seção 4.1 define o escopo da análise, estabelecendo os critérios de seleção e os *thresholds* dos *code smells*; a seção 4.2 descreve a arquitetura multiagente e o fluxo de execução, detalhando a coordenação centralizada e a especialização dos agentes; e a seção 4.3 explana a integração com os LLMs, demonstrando a aplicação de técnicas de engenharia de *prompts* para otimizar a precisão das detecções.

4.1 Escopo da Análise

4.1.1 Definição Critérios de Seleção dos Code Smells

A delimitação do escopo aos onze *implementation smells*s apresentados na taxonomia do Tabela 1 baseou-se em três critérios. O primeiro diz respeito à disponibilidade de padrão de referência validado: os *smells* selecionados (*Long Method*, *Complex Method*, *Complex Conditional*, *Long Parameter List*, *Long Statement*, *Long Identifier*, *Magic Number*, *Empty Catch Block*, *Missing Default*, *Long Lambda Function* e *Long Message Chain*) correspondem aos tipos anotados no conjunto de dados *DPy* (Sharma, 2025), cujas anotações manuais foram validadas por especialistas, condição necessária para uma avaliação quantitativa rigorosa. O segundo critério refere-se à granularidade de análise: *implementation smells*s representam a categoria mais específica na taxonomia de *code smells*, manifestando-se no nível de código-fonte

individual, contexto em que LLMs tendem a demonstrar melhor capacidade analítica. O terceiro critério é a objetividade: todos os *smells* selecionados possuem *thresholds* quantitativos bem definidos na literatura de engenharia de software.

4.1.2 Seleção dos Thresholds

Os *thresholds* selecionados e adotados para este estudo seguem as definições da ferramenta *DPy* (Sharma, 2025), que foram derivadas da literatura clássica de engenharia de software. O Quadro 2 apresenta cada *smell* com seu respectivo *threshold* e referência bibliográfica.

Quadro 2 – *Implementation smells* detectados pelo sistema e seus *thresholds*.

Code Smell	***Threshold***	**Referência**
Long Method	> 67 linhas	Fowler *et al.* (1999)
Complex Method	CC > 7	Sharma (2025)
Complex Conditional	> 2 operadores lógicos	Fowler (2018)
Long Parameter List	> 4 parâmetros	Fowler *et al.* (1999)
Long Statement	> 120 caracteres	PEP 8*
Long Identifier	> 20 caracteres	Martin (2008)
Magic Number	Literais (exceto 0, 1, -1)	Fowler *et al.* (1999), Martin (2008)
Empty Catch Block	Bloco vazio ou só `pass`	Martin (2008)
Missing Default	Sem `case _`	CWE-478**
Long Lambda Function	> 80 caracteres	Chen *et al.* (2016)
Long Message Chain	> 2 métodos encadeados	Fowler *et al.* (1999)

* PEP 8 é o guia de estilo oficial para código *Python*.
** CWE (*Common Weakness Enumeration*) é um catálogo de vulnerabilidades de software.
Fonte: Elaborado pelo autor com base em Sharma (2025).

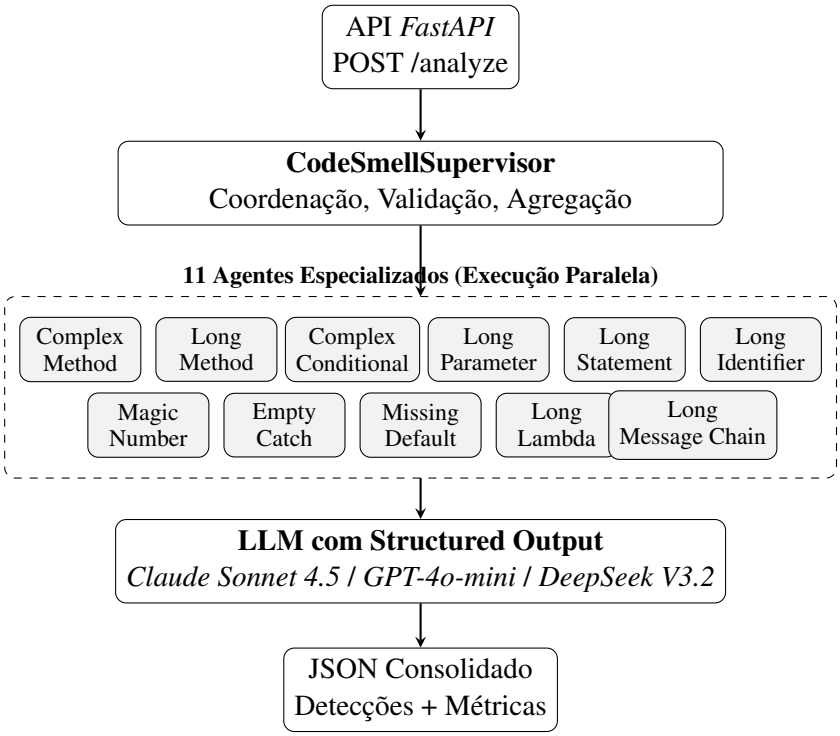
4.2 Arquitetura do Sistema Multiagente

A arquitetura do sistema segue o padrão *supervisor* descrito na subseção 2.2.2. Nesse modelo, um agente coordenador distribui tarefas a agentes especialistas e consolida os resultados em uma resposta estruturada. A escolha por essa arquitetura fundamenta-se em evidências da literatura: Silva *et al.* (2024) demonstraram que *prompts* detalhados aumentam a precisão das detecções, enquanto Sadik e Govind (2025) evidenciaram que diferentes modelos apresentam desempenhos distintos conforme a tarefa, o que sugere benefícios em combinar especialização com coordenação centralizada.

O sistema desenvolvido é composto por três elementos principais: um coordena-

dor central (*CodeSmellSupervisor*), onze agentes especializados e um modelo de linguagem configurado para saída estruturada. A Figura 1 ilustra a organização desses componentes.

Figura 1 – Arquitetura do sistema multiagente para detecção de *code smells*.



Fonte: Elaborado pelo autor.

4.2.1 Implementação do CodeSmellSupervisor

O *CodeSmellSupervisor* foi implementado como a classe central do sistema, responsável por coordenar todo o processo de análise. Suas funcionalidades compreendem: (i) validar o tamanho do código de entrada (limitado a 3.000 linhas e 500KB para não exceder os limites de contexto do LLM);(ii) formatar o código com numeração de linhas, facilitando a identificação de ocorrências;(iii) distribuir o código aos onze agentes especializados;(iv) coletar e validar as detecções, filtrando falsos positivos; e, por fim,(v) agregar os resultados em uma resposta consolidada que inclui métricas de consumo de *tokens*.

O coordenador suporta dois modos de execução: paralelo, onde todos os agentes são acionados simultaneamente, e sequencial, onde os agentes são executados um após o outro com intervalo de 300 milissegundos entre chamadas. O modo paralelo oferece menor latência, enquanto o modo sequencial permite maior controle sobre o consumo de recursos.

4.2.2 Integração do sistema com Agentes Especializados

Conforme ilustrado na Figura 1, o sistema integra-se com onze agentes especializados, cada um responsável pela detecção de um único tipo de *implementation smells*. A quantidade de agentes corresponde aos onze tipos de *smells* presentes no padrão de referência. Alternativas como agrupamento de *smells* similares ou uso de *prompt* único consolidado foram descartadas por duas razões.

A primeira razão refere-se ao fenômeno *lost in the middle* (Liu *et al.*, 2024). Estudos demonstram que LLMs apresentam dificuldade em utilizar adequadamente informações posicionadas no meio de contextos longos: o desempenho é significativamente melhor quando as informações relevantes estão no início ou no final do *prompt*. Um *prompt* consolidado contendo definições, exemplos e critérios para todos os onze *smells* resultaria em texto extenso, onde as instruções intermediárias seriam processadas com menor atenção pelo modelo. A divisão em agentes especializados mantém cada *prompt* conciso e focado, evitando essa degradação.

A segunda razão é a rastreabilidade: a granularidade permite identificar quais agentes apresentam melhor ou pior desempenho, facilitando otimizações incrementais e diagnóstico de falhas. O *overhead* de múltiplas chamadas é mitigado pela execução paralela, resultando em latência total inferior à soma das latências individuais. Esta especialização permite que cada agente utilize um *prompt* otimizado para sua tarefa específica, conforme recomendado por Silva *et al.* (2024).

Cada agente recebe dois componentes: um *prompt* que instrui o LLM sobre o que detectar e um *schema Pydantic* que define a estrutura da resposta esperada. A combinação de *prompt* e *schema* implementa a técnica de *structured output* descrita na subseção 2.3.3. Os *thresholds* utilizados para cada tipo de *smell* seguem os valores definidos no Quadro 2.

4.2.3 Configuração do Modelo de Linguagem com Saída Estruturada

O LLM foi configurado no sistema com temperatura zero para reduzir a variabilidade e aumentar a reprodutibilidade das respostas. A temperatura zero força o modelo a escolher sempre o *token* de maior probabilidade, reduzindo a aleatoriedade do processo de geração. Contudo, mesmo com temperatura zero, nem todos os LLMs são estritamente determinísticos devido à natureza probabilística dos modelos e a diferenças na implementação entre fornecedores. Para maximizar a reprodutibilidade dentro dessas limitações, cada configuração experimental

foi executada uma única vez, decisão justificada pelo custo das chamadas às APIs. Múltiplas execuções permitiriam calcular intervalos de confiança, sendo recomendadas em trabalhos futuros com maior orçamento. A integração utiliza o modo `json_mode`, que força o modelo a retornar exclusivamente JSON válido conforme o *schema Pydantic* definido. Esta configuração elimina problemas de *parsing* e garante que todas as detecções sigam o formato esperado para processamento automatizado.

Três modelos foram avaliados: *Claude Sonnet 4.5* da *Anthropic*, *GPT-4o-mini* da *OpenAI* e *DeepSeek V3.2* da *DeepSeek*. A seleção considerou três critérios: representatividade de diferentes fornecedores, diversidade de faixas de custo, e disponibilidade de modo de saída estruturada (JSON). O *GPT-4o-mini* foi escolhido em detrimento do *GPT-4o* por representar a categoria de modelos otimizados para custo-benefício, cada vez mais relevante em aplicações práticas. Reconhece-se que a comparação envolve modelos de diferentes capacidades, porém essa heterogeneidade reflete o cenário real de decisão que desenvolvedores enfrentam ao escolher um LLM.

4.2.4 Validação das Detecções

A etapa de validação das detecções é realizada nas duas últimas etapas do *CodeSmellSupervisor*, conforme listado na Subseção 4.2.1. Neste passo, é conduzido um processo de verificação pós-processamento com o objetivo de reduzir a ocorrência de falsos positivos gerados pelos agentes. Esse procedimento é conduzido pelo componente *DetectionValidator*, responsável por aplicar regras específicas e critérios de validação adequados a cada tipo de *code smell* identificado.

O processo verifica se os valores reportados pelos agentes efetivamente violam os *thresholds* estabelecidos. Uma detecção de *Long Identifier*, por exemplo, só é aceita se o comprimento do identificador for estritamente maior que 20 caracteres. Essa verificação mostrou-se necessária porque os LLMs ocasionalmente reportam casos limítrofes ou valores incorretos.

O sistema também aplica filtros específicos para cada categoria de *smell*, além da validação de *thresholds*. No caso de *Magic Number*, são filtrados valores triviais como 0, 1, -1, 2, 10 e 100, que frequentemente representam constantes de inicialização ou índices legítimos. Para *Long Identifier*, ignoram-se identificadores de bibliotecas conhecidas (*jax*, *numpy*, *torch*) e métodos especiais do *Python* (*dunder methods*), que seguem convenções de nomenclatura da

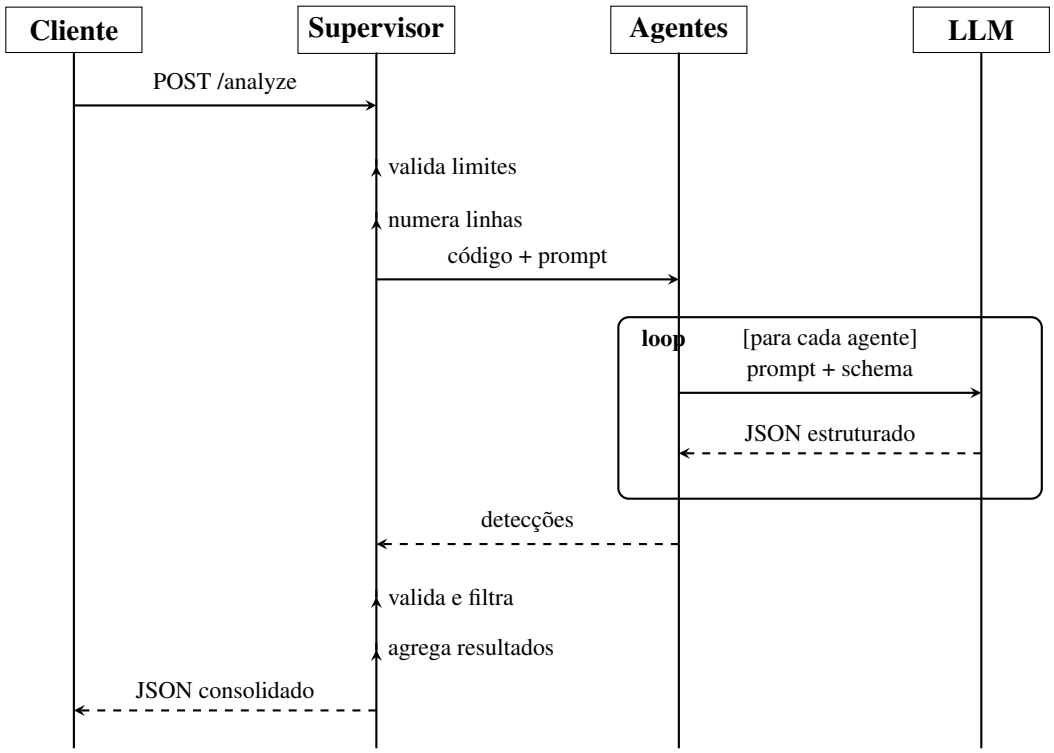
linguagem. Para *Long Message Chain*, excluem-se cadeias de chamadas iniciadas com módulos de biblioteca, como `os.path.join()` ou `jax.numpy.array()`.

Essa abordagem de validação em múltiplas camadas combina a capacidade de análise contextual do LLM com regras determinísticas. O objetivo é aumentar a precisão do sistema, reduzindo falsos positivos sem comprometer os verdadeiros positivos.

4.2.5 Fluxo de Execução da Arquitetura

O processo de análise segue um fluxo estruturado de interações entre os componentes do sistema, ilustrado no diagrama de sequência da Figura 2.

Figura 2 – Diagrama de sequência do fluxo de execução.



Fonte: Elaborado pelo autor.

O fluxo inicia quando o cliente envia uma requisição POST com o código *Python* e metadados. O Supervisor valida os limites de tamanho ($\leq$ 3.000 linhas, $\leq$ 500KB) e adiciona numeração às linhas do código. Em seguida, distribui o código aos onze agentes, que podem executar em paralelo ou sequencialmente. Cada agente constrói uma mensagem composta pelo *prompt* específico seguido do código numerado e a envia ao LLM, que retorna um objeto JSON automaticamente validado pelo *schema Pydantic*. O Supervisor coleta as detecções, aplica regras

adicionais para filtrar falsos positivos e agrega os resultados em uma resposta consolidada com métricas de consumo de *tokens*.

4.3 Engenharia de *Prompts*

Conforme discutido na seção 2.3, a qualidade do *prompt* impacta diretamente o desempenho do modelo. Para investigar essa relação e responder à segunda questão de pesquisa, desenvolvemos dois conjuntos de *prompts*: um simples e outro elaborado.

4.3.1 Definição dos Prompts Simples

Os *prompts* simples foram construídos apenas com a definição básica do *smell* e um exemplo de detecção. Essa abordagem representa o cenário de uso mínimo: instruções diretas, sem contexto adicional. O Código-fonte 2 ilustra um exemplo para o agente *Long Method*.

Código-fonte 2 – Exemplo de *prompt* simples para detecção de *Long Method*.

```
"""Detecte Long Method: métodos com mais de 67 linhas.

Exemplo:
def process_order(order):  # linha 1, 70 linhas
    # 70 linhas de código aqui...
    validate()
    calculate()
    save()

Saída esperada:
{
  "detected": true,
  "detections": [{
    "Smell": "Long method",
    "Method": "process_order",
    "total_lines": 70,
    "threshold": 67
  }]
}
```

```
20 """
```

Fonte: Elaborado pelo autor.

4.3.2 Definição dos Prompts Elaborados

Os *prompts* elaborados foram definidos mediante a aplicação de técnicas de *few-shot learning* e *chain-of-thought* descritas nas Subseções 2.3.1 e 2.3.2. A estrutura de cada *prompt* elaborado inicia com uma referência bibliográfica que fundamenta a definição do *smell*, seguida por uma descrição detalhada que diferencia o que deve e o que não deve ser detectado. O *prompt* também inclui um processo de análise passo a passo que guia o raciocínio do modelo, exemplos de casos positivos e negativos para calibrar as detecções, e regras de validação que estabelecem critérios para evitar falsos positivos.

Como ilustração, o *prompt* para detecção de *Long Method* referencia Fowler *et al.* (1999) e define que apenas funções declaradas com a palavra-chave `def` devem ser analisadas, excluindo scripts de módulo e classes. O processo de análise orienta o modelo a identificar cada função, calcular o número total de linhas entre a declaração e o fim do bloco, comparar com o *threshold* de 67 linhas e reportar no máximo dez ocorrências. Os exemplos contrastam uma função de 70 linhas que deve ser detectada com uma função de 5 linhas que deve ser ignorada. As regras reforçam que funções importadas ou apenas referenciadas não devem ser reportadas.

Os *prompts* completos de todos os agentes estão disponíveis no Apêndice A (elaborados) e no Apêndice B (simples).

4.4 Disponibilização do Sistema

O sistema multiagente desenvolvido está disponível como código aberto no repositório *GitHub*[1], incluindo a implementação completa da API *FastAPI*, os onze agentes especializados, os *prompts* elaborados e simples, além de scripts para análise em lote e geração de figuras acadêmicas.

Para aproximar a detecção de *code smells* do fluxo de trabalho do desenvolvedor, foi desenvolvida uma extensão para o *Visual Studio Code*[2] que se integra à API do sistema multiagente. A extensão permite que desenvolvedores recebam *feedback* sobre *code smells* em

[1] Disponível em: https://github.com/Estevam1to/multi-agent-smell-detector
[2] Disponível em: https://github.com/Estevam1to/extension-smell-detector

tempo real, diretamente no ambiente de desenvolvimento integrado (IDE).

A interface da extensão oferece duas visualizações complementares. A Figura 3 apresenta o painel informativo, que exibe uma tabela com todos os onze tipos de *smells* detectáveis, suas categorias, *thresholds*, severidades, descrições e referências bibliográficas. Esse painel serve como guia de consulta rápida para o desenvolvedor compreender os critérios utilizados nas detecções.

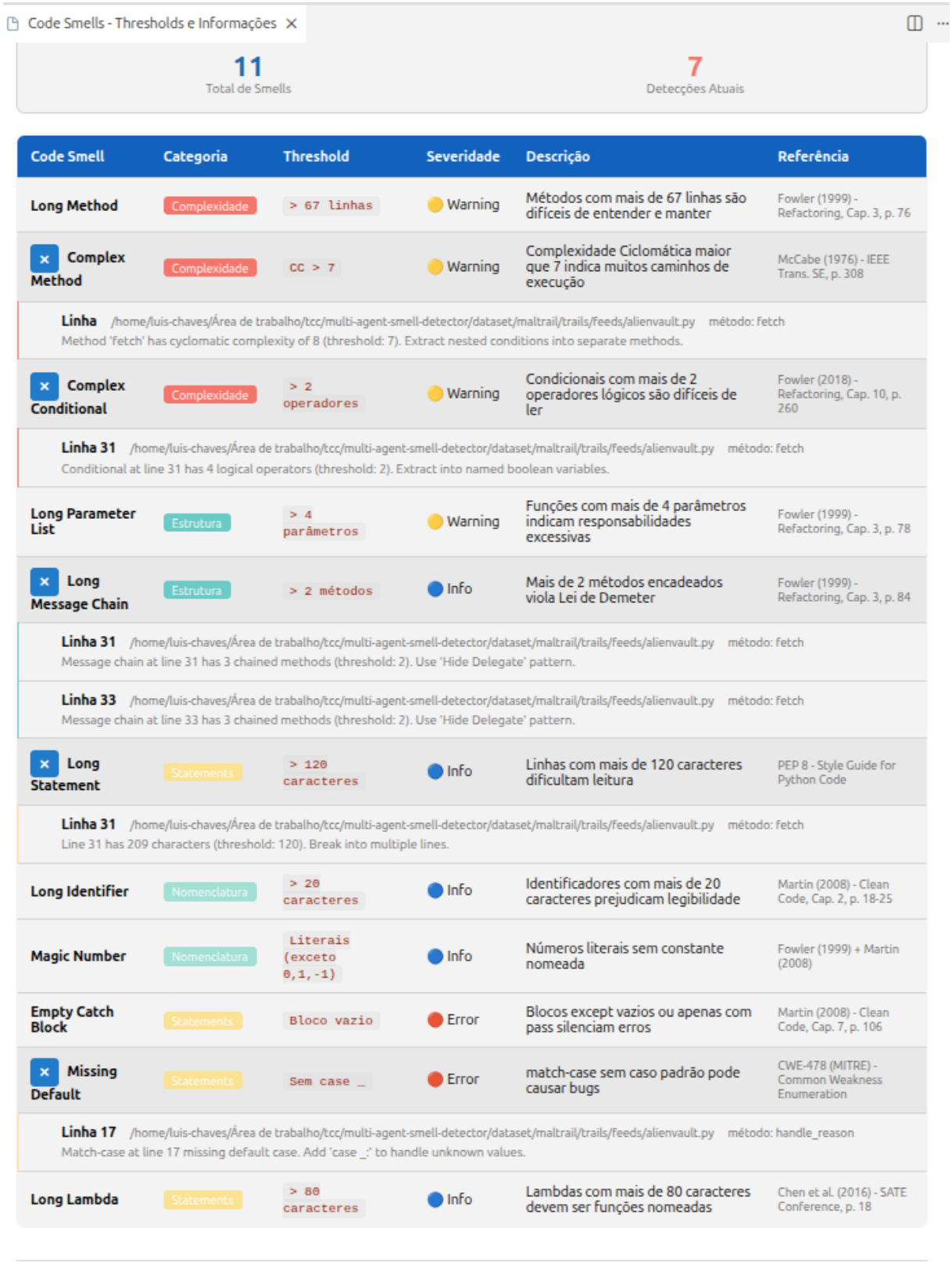
A Figura 4 ilustra a extensão em uso durante o desenvolvimento. As detecções são exibidas diretamente no editor, com sublinhados coloridos nas linhas problemáticas. Ao posicionar o cursor sobre uma detecção, o desenvolvedor visualiza detalhes como o tipo de *smell*, o valor detectado, o *threshold* violado e uma sugestão de correção. No exemplo apresentado, a extensão identificou sete ocorrências de *smells* no arquivo `alienvault.py`, incluindo *Complex Method*, *Complex Conditional*, *Long Message Chain*, *Long Statement* e *Missing Default*.

A extensão foi implementada em *TypeScript* utilizando a API de extensões do *VSCode*. Suas principais funcionalidades incluem: detecção automática ao salvar arquivos *Python*, exibição de diagnósticos no painel de problemas do editor, indicadores visuais com diferentes severidades (erro, aviso e informação) e barra de status com contador de *smells* detectados.

O fluxo de funcionamento, ilustrado na Figura 5, segue uma arquitetura cliente-servidor: quando o desenvolvedor salva um arquivo *Python*, a extensão envia o código para a API *FastAPI* via requisição HTTP, recebe o JSON com as detecções e converte os resultados em diagnósticos nativos do *VSCode*. Esta abordagem permite que qualquer *smell* detectado pelo sistema multiagente seja imediatamente visível no editor, com destaque na linha correspondente e descrição detalhada ao passar o cursor.

As severidades foram categorizadas conforme o impacto do *smell*: *Empty Catch Block* e *Missing Default* são classificados como erros por representarem potenciais falhas de execução; *Complex Method*, *Complex Conditional*, *Long Method* e *Long Parameter List* são avisos por afetarem a manutenibilidade; os demais são informações por serem questões de estilo.

Figura 3: Painel informativo da extensão com *thresholds* e descrições dos *code smells*.

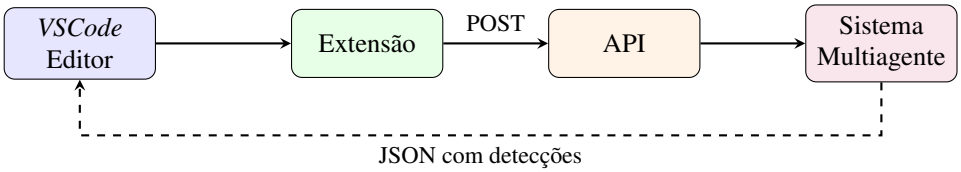


Fonte: Elaborado pelo autor.

Figura 4: Extensão em uso: detecções de *code smells* exibidas no editor durante o desenvolvimento.

Fonte: Elaborado pelo autor.

Figura 5: Fluxo de comunicação entre a extensão *VSCode* e a API.



Fonte: Elaborado pelo autor.

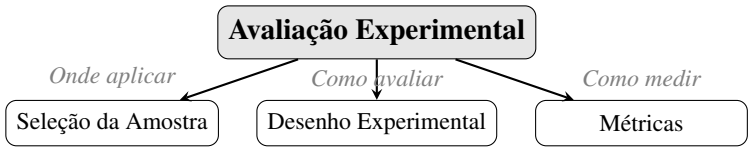
5 AVALIAÇÃO EXPERIMENTAL

Este capítulo descreve os procedimentos metodológicos adotados para avaliar experimentalmente a eficácia do sistema multiagente proposto. Com a solução tecnicamente consolidada, busca-se fundamentar as respostas aos objetivos e questões de pesquisa delineados na introdução por meio de uma abordagem experimental quantitativa e aplicada. Para tanto, detalham-se a seguir as etapas que compreendem desde a definição do escopo e seleção da amostra até o desenho experimental e as métricas de avaliação utilizadas.

5.1 Visão Geral dos Procedimentos Metodológicos

A metodologia caracteriza-se como uma pesquisa experimental de natureza aplicada, fundamentada em uma avaliação comparativa para mensurar o desempenho dos agentes especializados na detecção de *code smells*. O processo foi estruturado para confrontar diferentes modelos de linguagem e estratégias de *prompts* contra um padrão de referência validado. A organização lógica das etapas metodológicas e suas respectivas inter-relações estão ilustradas na Figura 6.

Figura 6: Fases dos procedimentos metodológicos do estudo.



Fonte: Elaborado pelo autor.

5.2 Seleção da Amostra

A etapa inicial dos procedimentos metodológicos consiste na seleção criteriosa de uma amostra de códigos-fonte que servirá como base para a execução dos experimentos e a validação do sistema. Esta amostra é composta por 20 arquivos Python, extraídos de repositórios de código aberto com diferentes níveis de maturidade, totalizando 8.463 linhas de código submetidas à análise. A escolha fundamenta-se na necessidade de confrontar as detecções do sistema contra um padrão de referência pré-validado por especialistas, garantindo o rigor necessário para responder às questões de pesquisa propostas. As subseções a seguir detalham

como foi realizado esse procedimento.

5.2.1 Definição do Padrão de Referência

Para este experimento, foi utilizado como referência o conjunto de dados da ferramenta *DPy*, desenvolvida por Sharma (2025). Esse conjunto de dados contém anotações manuais de *code smells* em projetos *Python* de código aberto[1]. Cabe ressaltar que o padrão de referência não foi criado pelo autor deste trabalho, mas reutilizado do conjunto de dados *DPy*, o que garante independência na avaliação.

O conjunto de dados *DPy* foi construído por meio de anotações manuais realizadas por especialistas em engenharia de software. O coeficiente *Kappa* de Cohen de 0,87 indica concordância substancial entre os avaliadores, conferindo confiabilidade às anotações. Da amostra selecionada para este estudo, o padrão de referência totaliza 411 instâncias de *smells* distribuídas em 20 arquivos. A Tabela 2 apresenta a distribuição das instâncias por tipo de *smell* no padrão de referência.

Tabela 2: Distribuição de instâncias de *code smells* no padrão de referência.

Code Smell	**Instâncias**
Long Statement	205
Magic Number	90
Long Method	28
Empty Catch Block	26
Long Identifier	18
Long Lambda Function	16
Long Parameter List	14
Complex Method	6
Complex Conditional	5
Long Message Chain	2
Missing Default	1
Total	**411**

Fonte: Elaborado pelo autor com dados do conjunto de dados *DPy*.

A distribuição é desbalanceada, com predominância de *Long Statement* (49,9%) e *Magic Number* (21,9%), enquanto *Missing Default* apresenta apenas uma ocorrência. Esse desbalanceamento reflete a distribuição natural de *smells* em código real e representa um desafio para os modelos de detecção, sobretudo para categorias com poucas amostras.

[1] Disponível em: https://zenodo.org/records/14283946. Acesso em: 25 dez. 2024.

5.2.2 Seleção dos Repositórios

A seleção dos repositórios seguiu cinco critérios principais:

1. **Disponibilidade de anotações validadas**: os repositórios deveriam fazer parte do conjunto de dados *DPy* (Sharma, 2025), que contém anotações manuais de *code smells* validadas por especialistas, garantindo um padrão de referência confiável para avaliação;
2. **Diversidade de domínios**: buscou-se representar diferentes áreas de aplicação (ferramentas de desenvolvimento, segurança e inteligência artificial) para avaliar a generalização do sistema;
3. **Maturidade dos projetos**: priorizaram-se projetos com histórico consolidado de desenvolvimento, variando de 3 a 13 anos de existência, indicando estabilidade e práticas de codificação estabelecidas;
4. **Relevância na comunidade**: considerou-se o engajamento da comunidade por meio de métricas como *stars* e *forks* no *GitHub*, totalizando mais de 10.000 estrelas combinadas;
5. **Variedade de tamanhos**: a amostra deveria incluir arquivos de diferentes dimensões, desde módulos compactos (26 linhas) até arquivos extensos (1.409 linhas), permitindo avaliar o comportamento do sistema em cenários variados de complexidade.

O Quadro 3 apresenta as características de cada repositório selecionado.

Quadro 3: Repositórios *Python* selecionados para análise.

Repositório	**Descrição**	***Stars***	***Forks***	**Desde**	***Commit***
codespell[2]	Verificador ortográfico para código-fonte, utilizado em projetos como *Linux Kernel* e *CPython*.	2.286	506	2011	`061e2fa`
maltrail[3]	Sistema de detecção de tráfego malicioso baseado em listas de ameaças conhecidas.	7.830	1.211	2014	`cd0128c`
Mava[4]	*Framework* para aprendizado por reforço multiagente em *JAX*[5], desenvolvido pela *InstaDeep*.	866	118	2021	`c4e40ce`

Fonte: Elaborado pelo autor com dados do *GitHub* em dezembro de 2024.

[2] Disponível em: https://github.com/codespell-project/codespell
[3] Disponível em: https://github.com/stamparm/maltrail
[4] Disponível em: https://github.com/instadeepai/Mava
[5] *JAX* é uma biblioteca de computação numérica de alto desempenho do *Google*.

Dos três repositórios selecionados, foram extraídos 20 arquivos *Python* totalizando 8.463 linhas de código, com 411 instâncias de *code smells* documentadas no padrão de referência.

5.2.3 Análise dos Arquivos Python

Dos repositórios selecionados, foram extraídos 20 arquivos *Python* que compõem a amostra de análise. O Quadro 4 detalha cada arquivo com sua respectiva quantidade de linhas de código.

Quadro 4: Arquivos *Python* selecionados para análise.

Repositório	**Arquivo**	**Linhas**
codespell	`_codespell.py`	1.351
	`_spellchecker.py`	75
	`test_basic.py`	1.409
	`test_dictionary.py`	374
maltrail	`addr.py`	73
	`colorized.py`	50
	`common.py`	324
	`log.py`	302
	`settings.py`	526
	`update.py`	426
	`alienvault.py`	35
	`badips.py`	30
	`dataplane.py`	26
	`torproject.py`	26
Mava	`ff_ippo_store_experience.py`	687
	`ff_ippo.py`	591
	`ff_mappo.py`	574
	`rec_ippo.py`	742
	`ff_isac.py`	625
	`checkpointing.py`	217

Fonte: Elaborado pelo autor.

A amostra totaliza 20 arquivos e 8.463 linhas de código, variando desde módulos pequenos como `torproject.py` (26 linhas) até arquivos extensos como `test_basic.py` (1.409 linhas). Esta diversidade permite avaliar o comportamento do sistema em diferentes cenários de complexidade.

5.3 Desenho Experimental

O desenho experimental foi estruturado com o propósito de fundamentar as respostas às cinco questões de pesquisa delineadas na introdução. Para assegurar a validade dos resultados, as subseções a seguir detalham as variáveis independentes e dependentes manipuladas, bem

como os tratamentos experimentais aplicados ao sistema.

5.3.1 Variáveis Independentes

O experimento manipulou duas variáveis independentes. A primeira é o modelo de linguagem, com três níveis: *Claude Sonnet 4.5*, *GPT-4o-mini* e *DeepSeek V3.2*. A segunda é o tipo de *prompt*, com dois níveis: simples e elaborado. A comparação entre tipos de *prompt* foi realizada exclusivamente com o modelo *Claude Sonnet 4.5*, enquanto a comparação entre modelos utilizou apenas *prompts* elaborados.

5.3.2 Variáveis Dependentes

As métricas coletadas para cada execução incluem verdadeiros positivos (detecções corretas confirmadas pelo padrão de referência), falsos positivos (detecções incorretas), falsos negativos (*smells* não detectados), consumo de *tokens* de entrada e saída, custo em dólares e tempo de execução em segundos.

5.3.3 Tratamentos Experimentais

O experimento compreendeu quatro tratamentos, conforme as questões de pesquisa definidas na seção 1.2. Para avaliar o impacto do tipo de *prompt* na precisão das detecções (QP2), o modelo *Claude Sonnet 4.5* foi executado com *prompts* elaborados e simples. Para avaliar a eficácia geral do sistema (QP1) e comparar o desempenho entre modelos de linguagem (QP3), os modelos *GPT-4o-mini* e *DeepSeek V3.2* foram executados apenas com *prompts* elaborados.

5.3.4 Conexão com as Questões de Pesquisa

O Quadro 5 relaciona cada questão de pesquisa, definida na seção 1.2, com as métricas e comparações que permitem respondê-la.

Quadro 5: Relação entre questões de pesquisa e métricas de avaliação.

QP	Questão de Pesquisa	Métricas/Comparação
1	Eficácia do sistema multiagente	Precision, Recall, F1-Score
2	Impacto dos *prompts*	F1 (elaborado) vs F1 (simples)
3	Comparação entre LLMs	F1 por modelo
4	Contribuição individual dos agentes	F1 por tipo de *smell*
5	Relação custo-benefício	Custo, *tokens*, tempo

Fonte: Elaborado pelo autor.

5.4 Métricas de Avaliação

A avaliação do desempenho do sistema utiliza métricas de classificação binária, onde cada *smell* do padrão de referência representa uma instância a ser detectada.

5.4.1 Processo de Validação

A validação das detecções foi realizada por meio de comparação binária por presença, considerando cada combinação única de módulo (arquivo) e tipo de *smell*. Para cada par (módulo, *smell*), verifica-se se existe ao menos uma ocorrência no padrão de referência e se o sistema reportou ao menos uma detecção.

Os critérios de classificação seguem a lógica apresentada no Quadro 6. Esta abordagem avalia a capacidade do sistema em identificar a presença de cada tipo de *smell* em cada arquivo, independentemente da contagem exata de ocorrências.

Quadro 6: Critérios de classificação para validação das detecções.

Classificação	No Padrão de Referência	Detectado	Interpretação
Verdadeiro Positivo (VP)	Presente	Sim	Detecção correta
Falso Positivo (FP)	Ausente	Sim	Detecção incorreta
Falso Negativo (FN)	Presente	Não	*Smell* não detectado
Verdadeiro Negativo (VN)	Ausente	Não	Ausência correta

Fonte: Elaborado pelo autor.

A escolha pela validação por presença binária, em vez de contagem exata de instâncias, fundamenta-se em dois aspectos. Primeiro, o objetivo primário do sistema é alertar o desenvolvedor sobre a existência de problemas em determinado arquivo, funcionando como triagem inicial - identificar *onde* há problemas é mais relevante que contar *quantos* existem. Segundo, a contagem exata introduziria complexidade adicional na correspondência entre de-

tecções (que podem usar diferentes granularidades de localização) e anotações do padrão de referência. Reconhece-se, contudo, que essa abordagem pode superestimar o desempenho em arquivos com múltiplas ocorrências do mesmo *smell*, limitação documentada na seção 7.3.

O processo de validação foi implementado em *Python* utilizando a biblioteca *Pandas* para manipulação dos dados. Para cada modelo e configuração de *prompt*, o *script* carrega os resultados da detecção e o padrão de referência, normaliza os nomes dos *smells* para garantir correspondência, agrupa as detecções por módulo e tipo de *smell*, e compara com as anotações do padrão de referência para classificar cada par como VP, FP, FN ou VN. As métricas agregadas são então calculadas a partir dessas classificações.

5.4.2 Métricas de Detecção

Nas fórmulas a seguir, VP representa Verdadeiros Positivos (detecções corretas), FP representa Falsos Positivos (detecções incorretas) e FN representa Falsos Negativos (*smells* não detectados).

Precisão: mede a proporção de detecções corretas entre todas as detecções realizadas. Indica a confiabilidade das detecções reportadas.

$$\text{Precision} = \frac{VP}{VP + FP} \tag{5.1}$$

Recall: mede a proporção de *smells* reais que foram corretamente identificados. Indica a cobertura da análise.

$$\text{Recall} = \frac{VP}{VP + FN} \tag{5.2}$$

F1-Score: média harmônica entre precisão e *recall*, fornecendo uma métrica balanceada do desempenho geral.

$$\text{F1-Score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.3}$$

5.4.3 Métricas de Eficiência

Consumo de *tokens*: registrado automaticamente para cada chamada ao LLM, discriminando *tokens* de entrada (*prompt*) e saída (resposta).

Custo: calculado com base nas tabelas de preços dos provedores, multiplicando o consumo de *tokens* pelo custo unitário de cada modelo.

Custo por Verdadeiro Positivo: métrica de eficiência que relaciona o investimento financeiro com o resultado obtido.

$$\text{Custo por VP} = \frac{\text{Custo Total}}{VP} \tag{5.4}$$

6 ESTUDO PARA INVESTIGAR A DETECÇÃO DE *CODE SMELLS* COM SISTEMAS MULTIAGENTES

Este capítulo apresenta os resultados experimentais obtidos com a avaliação do sistema multiagente proposto. A base de avaliação compreendeu 411 instâncias de *code smells* em 20 arquivos *Python*, conforme descrito na metodologia. Nas seções seguintes, analisam-se a eficácia geral do sistema, o impacto das técnicas de engenharia de *prompts*, a comparação entre diferentes modelos de linguagem, o desempenho por tipo de *smell* e a relação custo-benefício.

6.1 Introdução

A detecção automatizada de *code smells* representa um desafio relevante na engenharia de software, uma vez que ferramentas tradicionais de análise estática frequentemente apresentam limitações como alta taxa de falsos positivos e incapacidade de considerar o contexto específico do código analisado. Nesse cenário, a aplicação de LLMs em arquiteturas multiagentes surge como alternativa promissora, combinando a capacidade de compreensão contextual dos modelos de linguagem com a especialização proporcionada por agentes dedicados.

O objetivo deste estudo é avaliar a eficácia de um sistema multiagente baseado em LLMs para detecção de *code smells* de implementação em projetos *Python*. Especificamente, busca-se investigar o desempenho de diferentes modelos de linguagem, o impacto da engenharia de *prompts* na qualidade das detecções e a relação custo-benefício entre as alternativas avaliadas. Para orientar esta investigação, foram definidas as seguintes questões de pesquisa:

1. **QP1:** Qual a eficácia do sistema multiagente na detecção de *code smells* de implementação em projetos *Python*, em termos de precisão, *recall* e *F1-Score*?
2. **QP2:** Qual o impacto da qualidade dos *prompts* (elaborados vs. simples) na precisão das detecções realizadas pelo sistema multiagente?
3. **QP3:** Como diferentes modelos de linguagem (*Claude Sonnet 4.5*, *GPT-4o-mini*, *DeepSeek V3.2*) se comparam em termos de desempenho na detecção de *code smells*?
4. **QP4:** Qual a contribuição individual de cada agente especializado para o desempenho geral do sistema?
5. **QP5:** Qual a relação custo-benefício entre os diferentes modelos, considerando tempo de execução, consumo de *tokens* e custos computacionais?

6.2 QP1: Qual a eficácia do sistema multiagente na detecção de *code smells* de implementação em projetos *Python*, em termos de precisão, *recall* e *F1-Score*?

O sistema multiagente foi avaliado quanto à sua capacidade de identificar corretamente os *code smells* presentes no padrão de referência. A Tabela 3 apresenta as métricas de desempenho obtidas com o modelo *Claude Sonnet 4.5* utilizando *prompts* elaborados.

Tabela 3: Métricas de eficácia do sistema multiagente (*Claude Sonnet 4.5*).

Métrica	**Valor**
Verdadeiros Positivos (VP)	57
Falsos Positivos (FP)	59
Falsos Negativos (FN)	12
Verdadeiros Negativos (VN)	81
Precisão	49,14%
Recall	82,61%
F1-Score	61,62%
Acurácia	66,03%

Fonte: Elaborado pelo autor.

O sistema demonstrou alta sensibilidade na detecção: o *recall* de 82,61% indica que a maioria dos *smells* presentes no padrão de referência foi identificada corretamente. Por outro lado, a precisão de 49,14% revela tendência a gerar falsos positivos. O *F1-Score* resultante de 61,62% representa um equilíbrio moderado entre essas duas métricas.

A precisão de 49,14% indica que aproximadamente metade das detecções são falsos positivos, o que poderia gerar *alert fatigue* em uso contínuo. Contudo, o sistema foi projetado como ferramenta de triagem, não como substituto da revisão humana. O alto *recall* (82,61%) garante que a maioria dos problemas reais seja sinalizada, cabendo ao desenvolvedor a decisão final sobre cada alerta. A extensão para *VSCode* permite descartar rapidamente falsos positivos por meio de interação direta, minimizando o impacto na produtividade. A análise dos erros revelou padrões interessantes. Os falsos positivos concentram-se em *smells* onde os LLMs apresentam dificuldade em calcular corretamente a métrica ou aplicar adequadamente o *threshold*, como *Long Identifier* e *Complex Method*. No caso de *Complex Method*, embora o *threshold* seja bem definido (CC > 7), os LLMs podem calcular incorretamente a complexidade ciclomática, especialmente em métodos com estruturas aninhadas complexas. Para *Long Identifier*, mesmo com *threshold* claro (> 20 caracteres), os modelos tendem a reportar identificadores que tecnicamente

violam o limite mas são legítimos (como identificadores de bibliotecas ou métodos especiais do Python), gerando falsos positivos. Já os falsos negativos, em menor quantidade, ocorreram predominantemente em casos limítrofes, próximos aos *thresholds* estabelecidos.

A Tabela 4 apresenta os resultados das métricas de eficácia obtidas pelos demais LLMs avaliados neste estudo (*GPT-4o-mini* e *DeepSeek V3.2*), evidenciando variações nos percentuais de desempenho entre os modelos.

Tabela 4: Comparação de métricas entre modelos LLM.

Modelo	**Precisão**	***Recall***	***F1-Score***	**Acurácia**
Claude Sonnet 4.5	49,14%	82,61%	61,62%	66,03%
GPT-4o-mini	54,37%	81,16%	65,12%	72,73%
DeepSeek V3.2	71,70%	55,07%	62,30%	77,99%

Fonte: Elaborado pelo autor.

Em cenários que demandam maior precisão e menor tolerância a falsos positivos, o *DeepSeek V3.2* destaca-se como uma alternativa adequada, alcançando 71,70% de precisão e a maior acurácia entre os avaliados. Por outro lado, o *GPT-4o-mini* obteve desempenho superior em termos de *recall* e apresentou o maior *F1-Score*, indicando melhor equilíbrio entre precisão e cobertura. De modo geral, observa-se que ambos os modelos apresentaram resultados consistentemente acima da média nas métricas analisadas, principalmente *recall* e *F1-Score*.

Resposta da Questão de Pesquisa 1 (Eficácia): Os resultados obtidos indicam que o sistema apresenta eficácia consistente na detecção de *code smells*. Embora a métrica de precisão revele uma tendência de alguns modelos à geração de falsos positivos, observa-se que esses modelos alcançaram elevados níveis de cobertura e um bom equilíbrio entre as métricas de *recall*, *F1-Score* e acurácia. Esses resultados evidenciam a adequação do sistema para apoiar atividades de análise e avaliação da qualidade de código.

6.3 QP2: Qual o impacto da qualidade dos *prompts* (elaborados vs. simples) na precisão das detecções realizadas pelo sistema multiagente?

Para avaliar se a qualidade das instruções fornecidas aos agentes influencia o desempenho, comparamos o modelo *Claude Sonnet 4.5* com dois conjuntos de *prompts*: os elaborados, que incluem definições detalhadas, exemplos e raciocínio estruturado; e os simples, que contêm apenas a definição básica do *smell*. A Tabela 5 exibe o resultado das três métricas analisadas de

acordo com o tipo de *prompt* utilizado.

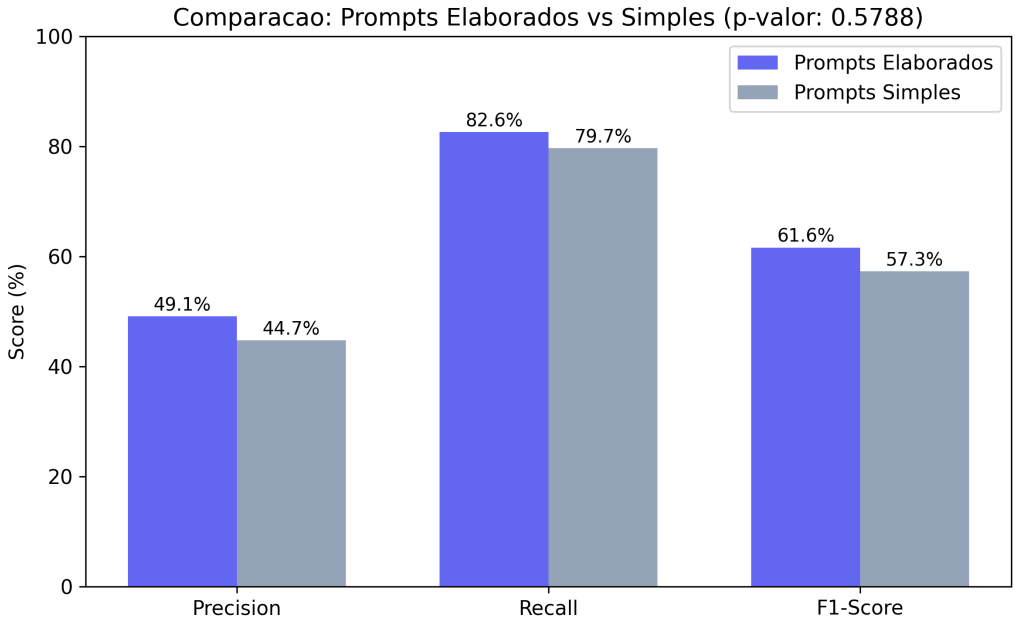
Tabela 5: Comparação de desempenho entre *prompts* elaborados e simples.

Métrica	**Elaborados**	**Simples**	**Diferença**
Precisão	49,14%	44,72%	+4,42 pp
Recall	82,61%	79,71%	+2,90 pp
F1-Score	61,62%	57,29%	+4,33 pp

Fonte: Elaborado pelo autor.

Os *prompts* elaborados apresentaram desempenho superior em todas as métricas, com ganho de 4,33 pontos percentuais no *F1-Score*. A Figura 7 ilustra essa comparação.

Figura 7: Comparação de desempenho entre *prompts* elaborados e simples.



Fonte: Elaborado pelo autor.

Embora os *prompts* elaborados apresentem vantagem numérica em todas as métricas avaliadas, essa diferença pode não se justificar em todos os cenários, dependendo do perfil dos *smells* prioritários para cada projeto.

A ausência de significância estatística não invalida, contudo, a utilidade de *prompts* elaborados em contextos específicos. Observamos que os ganhos concentram-se em *smells* que demandam análise contextual mais profunda. Em casos como *Complex Conditional* e *Magic Number*, as instruções detalhadas e os exemplos auxiliam o modelo a compreender nuances

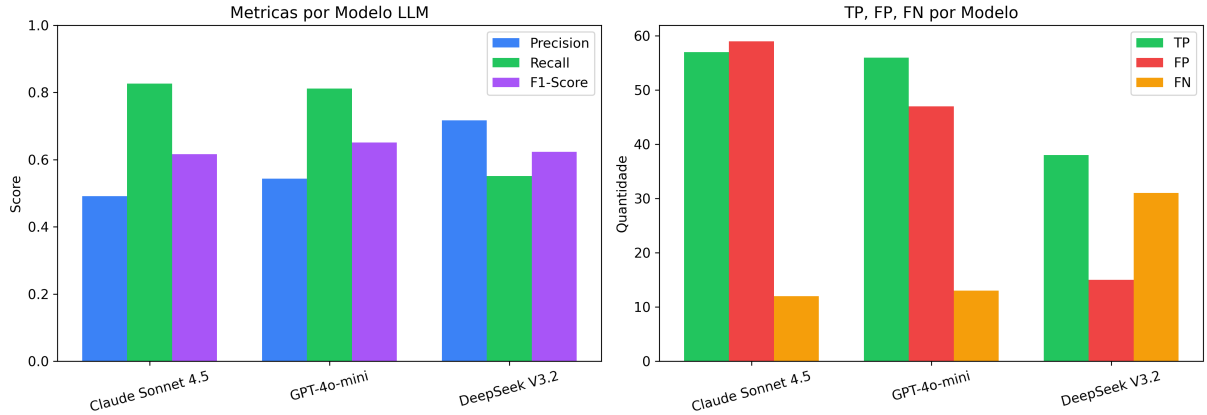
específicas do domínio. Para *smells* com critérios puramente quantitativos (como *Long Statement* ou *Long Method*), *prompts* simples podem ser suficientes. A decisão sobre qual tipo de *prompt* utilizar deve ponderar o esforço de elaboração contra o perfil dos *smells* prioritários para cada projeto.

Resposta da Questão de Pesquisa 2 (Impacto do tipo de *prompt*): A utilização de *prompts* elaborados resultou em aumentos relevantes nas métricas de Precisão, *Recall* e *F1-Score* em comparação aos *prompts* simples, ainda que parte desses ganhos não tenha apresentado significância estatística (para o modelo *Claude Sonnet 4.5*). Esses resultados indicam que o investimento em engenharia de *prompts* tende a ser mais vantajoso em cenários específicos, especialmente na detecção de *code smells* que demandam análises mais profundas e contextualizadas.

6.4 QP3: Como diferentes modelos de linguagem (*Claude Sonnet 4.5*, *GPT-4o-mini*, *DeepSeek V3.2*) se comparam em termos de desempenho na detecção de *code smells*?

Para essa comparação, o sistema foi avaliado com três modelos de linguagem de diferentes fornecedores: *Claude Sonnet 4.5* (*Anthropic*), *GPT-4o-mini* (*OpenAI*) e *DeepSeek V3.2*. Todos utilizaram os *prompts* elaborados, permitindo uma comparação justa entre as capacidades intrínsecas de cada modelo. A Figura 8 apresenta a visualização comparativa das métricas e da distribuição de verdadeiros positivos, falsos positivos e falsos negativos para cada modelo.

Figura 8: Comparação de desempenho entre modelos LLM: métricas e distribuição de detecções (VP, FP, FN).



Fonte: Elaborado pelo autor.

Ao analisar as análises exibidas na Tabela 4 e na Figura 8, os resultados revelam

perfis distintos de detecção. O *Claude Sonnet 4.5* alcançou o maior *recall* (82,61%), identificando a maioria dos *smells* presentes, embora com mais falsos positivos. O *GPT-4o-mini* obteve o melhor *F1-Score* (65,12%), equilibrando precisão e *recall*. Já o *DeepSeek V3.2* destacou-se pela maior precisão (71,70%) e acurácia (77,99%), adotando postura mais conservadora nas detecções, o que resultou em menor *recall* (55,07%).

A análise do número de detecções corrobora esses perfis: *Claude* identificou 500 instâncias, *GPT* detectou 283 e *DeepSeek* reportou 211, em comparação às 411 instâncias no padrão de referência. Essa variação evidencia que os modelos possuem calibrações internas distintas quanto à sensibilidade de detecção.

Resposta da Questão de Pesquisa 3 (Comparação entre LLMs): O *GPT-4o-mini* apresentou o desempenho mais equilibrado entre as métricas avaliadas, alcançando um F1-Score de 65,12%, o que indica uma boa relação entre precisão e *recall*. Em contraste, o *Claude Sonnet 4.5* destacou-se por priorizar a sensibilidade do sistema, obtendo os maiores valores de *recall* e demonstrando maior capacidade de identificar a maioria dos casos relevantes, ainda que com maior incidência de falsos positivos. Já o *DeepSeek V3.2* evidenciou um foco maior na confiabilidade das detecções, alcançando a maior precisão (71,70%), o que o torna mais adequado para cenários em que a redução de falsos positivos é um requisito prioritário.

6.5 QP4: Qual a contribuição individual de cada agente especializado para o desempenho geral do sistema?

Com o objetivo de analisar o desempenho individual na detecção de cada tipo de *smell*, os agentes especializados foram avaliados separadamente, permitindo identificar quais categorias de *code smells* são reconhecidas com maior ou menor eficácia. A Tabela 6 apresenta o *F1-Score* obtido para cada combinação de *smell* e modelo.

A Figura 9 apresenta a visualização comparativa do *F1-Score* por tipo de *smell*, facilitando a identificação dos pontos fortes de cada modelo.

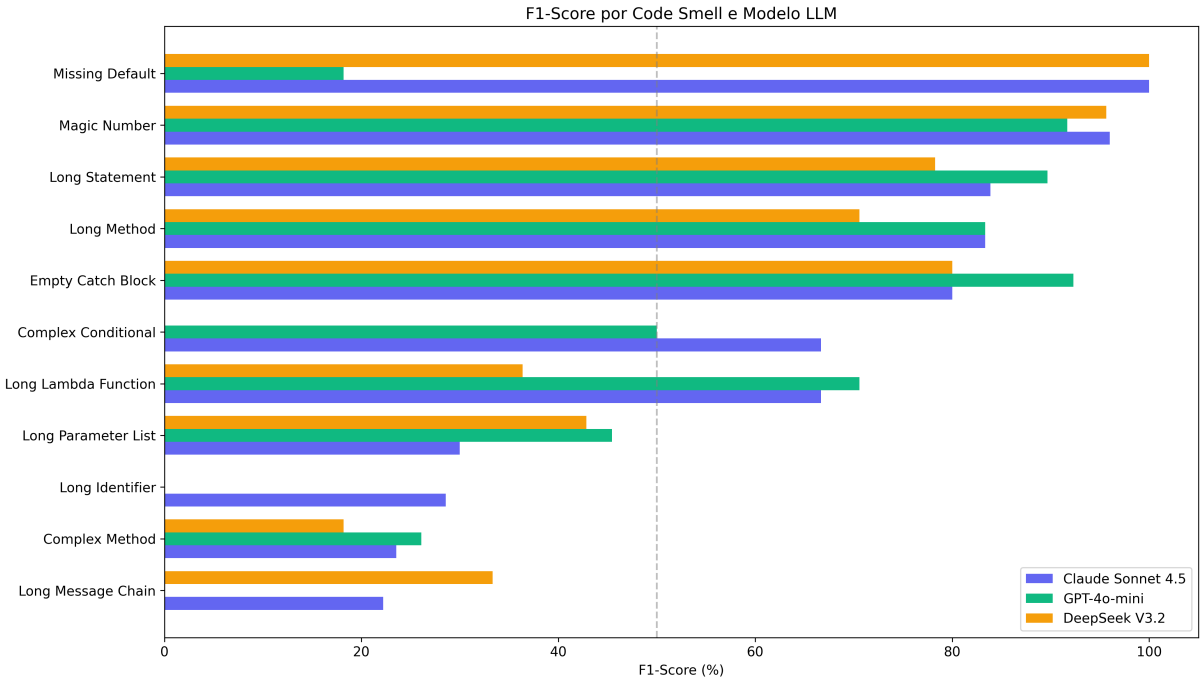
Quanto à distribuição de melhor desempenho por modelo, *Claude* e *GPT* empataram com 5 agentes cada como melhor *performer*, enquanto *DeepSeek* liderou em apenas 1 agente. Os *smells* detectados com maior eficácia foram *Missing Default* (100%), *Magic Number* (96%) e *Long Statement* (89,7%). Por outro lado, *Complex Method* (26,1%) e *Long Message Chain* (33,3%) representaram os maiores desafios.

Tabela 6: *F1-Score* por tipo de *code smell* e modelo LLM.

Code Smell	***Claude***	***GPT***	***DeepSeek***	**Melhor**
Complex Conditional	66,7%	50,0%	0,0%	*Claude*
Complex Method	23,5%	26,1%	18,2%	*GPT*
Empty Catch Block	80,0%	92,3%	80,0%	*GPT*
Long Identifier	28,6%	0,0%	0,0%	*Claude*
Long Lambda Function	66,7%	70,6%	36,4%	*GPT*
Long Message Chain	22,2%	0,0%	33,3%	*DeepSeek*
Long Method	83,3%	83,3%	70,6%	*Claude/GPT*
Long Parameter List	30,0%	45,5%	42,9%	*GPT*
Long Statement	83,9%	89,7%	78,3%	*GPT*
Magic Number	96,0%	91,7%	95,7%	*Claude*
Missing Default	100,0%	18,2%	100,0%	*Claude/DeepSeek*

Fonte: Elaborado pelo autor.

Figura 9: *F1-Score* por tipo de *code smell* e modelo LLM.



Fonte: Elaborado pelo autor.

Cabe ressaltar que o padrão de referência apresenta distribuição desbalanceada de instâncias por tipo de *smell*, conforme apresentado no Tabela 2. Enquanto *Long Statement* possui 205 ocorrências e *Magic Number* conta com 90, categorias como *Missing Default* (1 ocorrência) e *Long Message Chain* (2 ocorrências) são sub-representadas. Esse desbalanceamento afeta a interpretação das métricas: um *F1-Score* de 100% em *Missing Default* baseia-se em uma única instância, sendo estatisticamente menos robusto que os 89,7% obtidos em *Long Statement*, que refletem desempenho consistente em centenas de casos. Portanto, os resultados para categorias com poucas amostras devem ser interpretados com cautela.

Uma análise qualitativa indica que *smells* com critérios simples de verificação, como contagem de linhas e presença de estruturas específicas, são detectados com maior acurácia. Já *smells* que requerem cálculo de métricas mais complexas, como a complexidade ciclomática (mesmo com *threshold* bem definido), apresentam maior variabilidade entre modelos e taxa de discordância mais elevada com o padrão de referência, pois os LLMs podem calcular incorretamente essas métricas.

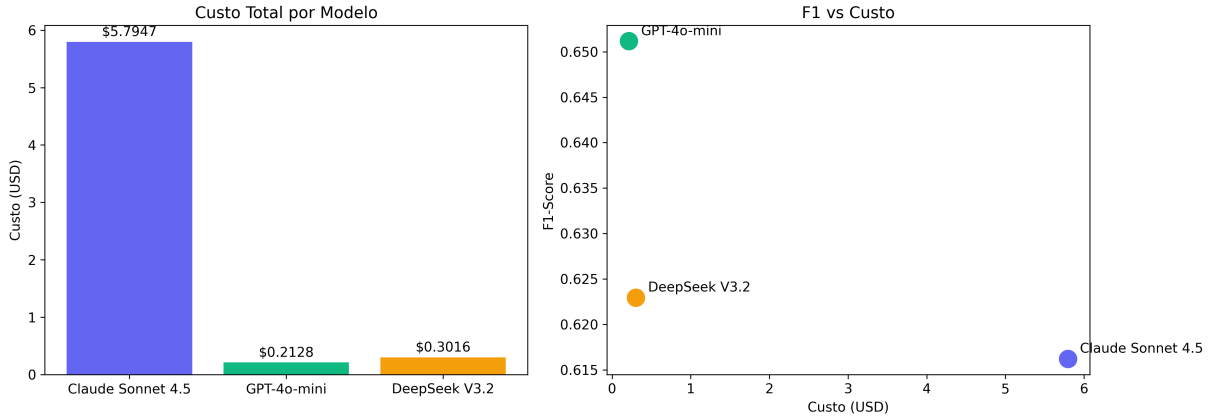
Resposta da Questão de Pesquisa 4 (Contribuição dos Agentes): Agentes voltados à identificação de code smells com critérios simples de verificação, como *Missing Default* e *Magic Number*, alcançaram níveis de eficácia próximos a 100%. Em contraste, agentes responsáveis por smells onde os LLMs apresentam dificuldade em calcular corretamente a métrica, mesmo com *thresholds* bem definidos, como *Complex Method* (que requer cálculo preciso de complexidade ciclomática) e *Long Identifier* (onde o modelo tende a reportar falsos positivos mesmo quando o *threshold* é tecnicamente violado), apresentaram maiores dificuldades de detecção, registrando eficácia inferior (abaixo de 30%).

6.6 QP5: Qual a relação custo-benefício entre os diferentes modelos, considerando tempo de execução, consumo de *tokens* e custos computacionais?

Além da qualidade das detecções, avaliamos a eficiência operacional dos modelos considerando o custo monetário das chamadas às APIs e o tempo total de processamento. A Figura 10 apresenta a distribuição de custos entre os modelos, bem como a relação entre investimento e qualidade obtida.

O gráfico de dispersão evidencia que *GPT-4o-mini* ocupa a posição mais favorável: maior *F1-Score* combinado com menor custo. O *Claude Sonnet 4.5*, posicionado no extremo

Figura 10: Análise de custo: custo total por modelo e relação *F1-Score* vs custo.



Fonte: Elaborado pelo autor.

direito, representa investimento significativamente maior sem retorno proporcional em qualidade. As métricas de eficiência são detalhadas na Tabela 7.

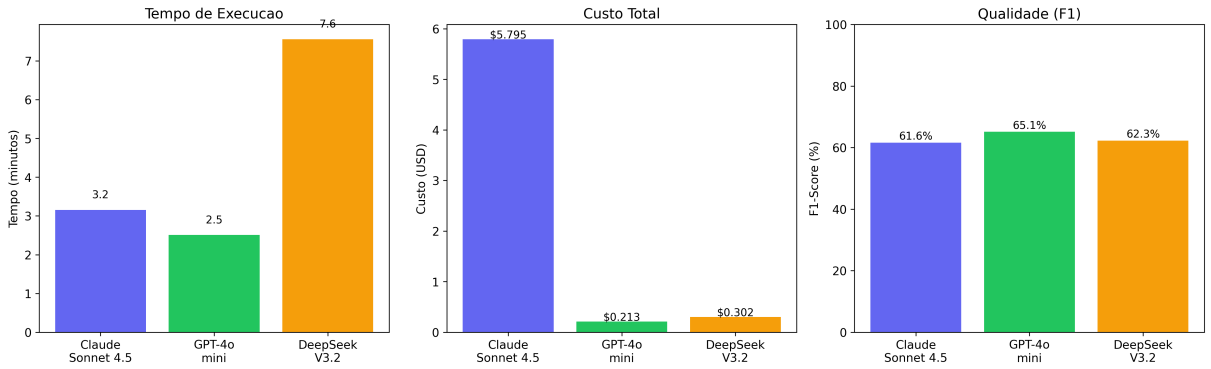
Tabela 7: Análise de custo-benefício por modelo LLM.

Modelo	***F1***	**Custo (USD)**	**Tempo**	***F1*/USD**	***F1*/min**
Claude Sonnet 4.5	61,6%	$5,79	3,2 min	10,6	19,5
GPT-4o-mini	65,1%	$0,21	2,5 min	306,0	25,9
DeepSeek V3.2	62,3%	$0,30	7,6 min	206,5	8,2

Fonte: Elaborado pelo autor.

A Figura 11 ilustra as três dimensões de eficiência: tempo de execução, custo total e qualidade (*F1-Score*) para cada modelo.

Figura 11: Análise de eficiência operacional: tempo, custo e qualidade por modelo.



Fonte: Elaborado pelo autor.

O *GPT-4o-mini* destacou-se como a opção mais eficiente: obteve o melhor *F1-Score* (65,1%) com o menor custo ($0,21) e menor tempo de execução (2,5 minutos). A métrica *F1*/USD de 306,0 indica que esse modelo oferece aproximadamente 29 vezes mais valor por dólar investido em comparação ao *Claude Sonnet 4.5* (10,6).

O *Claude Sonnet 4.5*, apesar do custo significativamente superior ($5,79), não demonstrou ganho proporcional em qualidade. Seu uso pode se justificar em cenários que demandem maior *recall*, onde a identificação exaustiva de *smells* seja prioritária. O *DeepSeek V3.2* posicionou-se como alternativa intermediária em custo, porém com tempo de execução cerca de três vezes superior aos concorrentes.

Resposta da Questão de Pesquisa 5 (Custo-Benefício): O *GPT-4o-mini* revelou-se o modelo mais eficiente globalmente, sendo 29 vezes mais econômico que o *Claude Sonnet 4.5* e 1,5 vezes mais que o *DeepSeek V3.2* em termos de valor por dólar *(F1/USD)*; o *DeepSeek*, por sua vez, posicionou-se como uma alternativa intermediária de baixo custo, superando o *Claude* em eficiência financeira, mas apresentando o maior tempo de execução entre todos os modelos testados.

6.7 Síntese dos Resultados

A Tabela 8 consolida os principais achados deste estudo.

Os resultados evidenciam que sistemas multiagentes baseados em LLMs constituem alternativa viável para detecção automatizada de *code smells*. A escolha do modelo deve considerar o contexto de aplicação: *GPT-4o-mini* para uso geral com restrições orçamentárias, *Claude* para cenários que priorizem cobertura máxima, e *DeepSeek* quando a precisão for crítica e falsos positivos devam ser minimizados.

Tabela 8: Síntese dos principais achados experimentais.

Aspecto Avaliado	**Principal Achado**
Eficácia geral	O sistema alcançou *F1-Score* de 61,62% com alto *recall* (82,61%), demonstrando capacidade de identificar a maioria dos *code smells* de implementação.
Engenharia de *prompts*	*Prompts* elaborados proporcionaram ganho de 4,33 pontos percentuais no *F1-Score*, porém a diferença não é estatisticamente significativa (p = 0,58).
Comparação de modelos	*GPT-4o-mini* obteve o melhor *F1-Score* (65,12%), seguido por *DeepSeek* (62,30%) e *Claude* (61,62%). Cada modelo apresenta perfil distinto de detecção.
Desempenho por *smell*	*Claude* e *GPT* empataram como melhores em 5 tipos de *smells* cada. *Magic Number* e *Missing Default* alcançaram detecção próxima a 100%.
Custo-benefício	*GPT-4o-mini* apresentou melhor relação custo-benefício (306 *F1*/USD), sendo 29× mais eficiente que *Claude* em termos monetários.

Fonte: Elaborado pelo autor.

7 CONCLUSÃO

Este capítulo apresenta as considerações finais do trabalho. Retomam-se os objetivos propostos, sintetizam-se as principais contribuições, discutem-se as limitações identificadas e sugerem-se direções para trabalhos futuros.

7.1 Síntese do Trabalho

Este trabalho investigou a aplicação de sistemas multiagentes baseados em LLMs para detecção automatizada de *code smells* em código *Python*. O sistema proposto adota uma arquitetura com um agente Supervisor que coordena onze agentes especializados, cada qual responsável pela detecção de um tipo específico de *smell* de implementação.

O objetivo geral de desenvolver e avaliar um sistema multiagente para detecção de *code smells* foi alcançado. Os experimentos conduzidos com três modelos de linguagem (*Claude Sonnet 4.5*, *GPT-4o-mini* e *DeepSeek V3.2*) demonstraram que a abordagem é viável e apresenta resultados promissores, com *F1-Score* variando entre 61,62% e 65,12% conforme o modelo utilizado.

Os objetivos específicos também foram atingidos. Implementamos com sucesso a arquitetura multiagente utilizando *prompts* estruturados e saídas em formato *JSON* validadas por esquemas *Pydantic*. A comparação entre modelos revelou perfis distintos de detecção. A análise do impacto da engenharia de *prompts* mostrou ganhos moderados, embora não estatisticamente significativos. Por fim, a extensão para *Visual Studio Code* aproximou a ferramenta do fluxo de trabalho cotidiano do desenvolvedor.

7.2 Principais Contribuições

Este trabalho oferece contribuições para a área de qualidade de código e aplicações de LLMs em engenharia de software.

A primeira diz respeito à arquitetura multiagente especializada. O sistema demonstrou que a divisão de responsabilidades entre agentes, cada um focado em um tipo de *smell*, viabiliza análises mais precisas do que abordagens monolíticas. A coordenação pelo agente Supervisor, aliada à execução paralela dos agentes especializados, mostrou-se eficiente tanto em tempo quanto na organização dos resultados.

A segunda contribuição é a análise comparativa de modelos. Os experimentos

revelaram perfis distintos de detecção: *Claude* com maior *recall*, *DeepSeek* com maior precisão e *GPT-4o-mini* com melhor equilíbrio e custo-benefício. Essa caracterização pode auxiliar profissionais na escolha do modelo mais adequado a cada contexto.

A terceira contribuição refere-se à avaliação de técnicas de engenharia de *prompts*. A comparação entre *prompts* elaborados e simples fornece evidências empíricas sobre o retorno do investimento em engenharia de *prompts*: os ganhos, embora perceptíveis, podem não justificar o esforço adicional em todos os cenários.

A quarta contribuição é a ferramenta prática para desenvolvedores. A extensão para *Visual Studio Code* demonstra a aplicabilidade do sistema no dia a dia, permitindo detecção de *smells* em tempo real durante o desenvolvimento.

7.3 Limitações e Ameaças à Validade

Algumas limitações devem ser consideradas na interpretação dos resultados.

7.3.1 Ameaças à Validade Interna

A validação por presença binária constitui uma simplificação do problema real. O método adotado verifica se o sistema detectou ao menos uma ocorrência de cada tipo de *smell* em cada arquivo, sem considerar a contagem exata de instâncias. Na prática, isso significa que um arquivo com dez ocorrências de *Magic Number* é tratado da mesma forma que um arquivo com apenas uma ocorrência, desde que o sistema detecte ao menos uma. Essa escolha metodológica pode superestimar o desempenho em cenários com múltiplas ocorrências do mesmo *smell*.

A dependência de um único padrão de referência representa outra limitação. Embora o conjunto de dados *DPy* tenha sido validado por especialistas com métricas de concordância satisfatórias, a utilização de uma única fonte de referência pode introduzir vieses específicos da metodologia de anotação original. Diferentes anotadores ou critérios poderiam resultar em avaliações distintas do mesmo sistema.

Os *prompts* foram desenvolvidos pelo próprio autor, o que pode introduzir viés na formulação das instruções. Outros pesquisadores poderiam elaborar *prompts* com estruturas ou ênfases diferentes, potencialmente obtendo resultados distintos.

7.3.2 Ameaças à Validade Externa

O tamanho da amostra, embora adequado para um estudo exploratório, limita a generalização dos resultados. Com 20 arquivos e 411 instâncias de *smells*, os resultados podem não representar adequadamente a diversidade de projetos *Python* existentes. A limitação foi condicionada pelo custo das chamadas às APIs e pelo tempo disponível para análise manual dos resultados. O conjunto de dados *DPy* contém anotações para projetos adicionais que poderiam ampliar a amostra em trabalhos futuros. Recomenda-se que pesquisas subsequentes, com maior orçamento, expandam a avaliação para verificar a estabilidade dos resultados em amostras maiores e mais diversificadas.

A restrição a uma única linguagem de programação (*Python*) impede generalizações para outras linguagens. A sintaxe, os padrões de código e a manifestação de *smells* variam significativamente entre linguagens, e o desempenho do sistema pode diferir em contextos como *Java*, *JavaScript* ou *C++*.

A seleção de repositórios de código aberto pode não representar código proprietário ou de outros domínios. Projetos corporativos podem apresentar padrões de código, convenções e tipos de *smells* diferentes dos encontrados em projetos *open source*.

7.3.3 Ameaças à Validade de Construto

As métricas de classificação binária (precisão, *recall*, *F1-Score*) capturam apenas parte da qualidade das detecções. Aspectos como a utilidade das explicações fornecidas, a localização precisa no código e a relevância das sugestões de correção não foram avaliados quantitativamente.

A análise de custo considera apenas os valores das APIs no momento da execução. Os preços dos serviços de LLMs variam ao longo do tempo, e novos modelos com melhor relação custo-desempenho são lançados frequentemente, o que pode alterar as conclusões sobre viabilidade econômica.

7.4 Trabalhos Futuros

Uma evolução natural do trabalho seria a expansão do escopo de análise. A inclusão de *design smells* e *architecture smells* permitiria uma análise mais abrangente da qualidade do código. Do mesmo modo, a extensão para outras linguagens de programação ampliaria a

aplicabilidade da ferramenta.

O refinamento da metodologia de validação também pode aprimorar a precisão das avaliações. A adoção de validação por contagem exata de instâncias, em vez de presença binária, forneceria métricas mais rigorosas. Criar um padrão de referência próprio, com múltiplos anotadores, permitiria calcular métricas de concordância e reduzir vieses.

Outra frente promissora é a investigação de técnicas avançadas de *prompting*. Estratégias como *self-consistency*, onde múltiplas respostas são geradas e agregadas, ou *chain-of-verification*, em que o modelo verifica suas próprias respostas, podem reduzir falsos positivos e aumentar a confiabilidade das detecções.

A integração com *pipelines* de integração contínua (CI/CD) permitiria análise automatizada a cada *commit* ou *pull request*, tornando a detecção de *smells* parte do fluxo de desenvolvimento. Essa integração demandaria otimizações de desempenho e custo para viabilizar execuções frequentes.

Mecanismos de *feedback* do usuário possibilitariam aprendizado incremental. Ao permitir que desenvolvedores confirmem ou rejeitem detecções, o sistema poderia ajustar seus critérios ao longo do tempo, adaptando-se às convenções de cada projeto ou equipe.

Por fim, a comparação com ferramentas tradicionais de análise estática, como *Pylint*, *SonarQube* e *Designite*, forneceria perspectiva mais ampla sobre o posicionamento da abordagem baseada em LLMs no ecossistema de ferramentas de qualidade de código.

7.5 Considerações Finais

Este trabalho demonstrou que sistemas multiagentes baseados em LLMs constituem alternativa viável para detecção automatizada de *code smells*. Um *F1-Score* de aproximadamente 65% significa que, em média, uma a cada três detecções é incorreta. Esse desempenho posiciona o sistema como ferramenta de apoio, não de substituição à revisão humana. Os cenários ideais de aplicação incluem: (i) projetos sem cobertura de ferramentas tradicionais de análise estática; (ii) triagem inicial em bases de código legado, onde a identificação rápida de áreas problemáticas é prioritária; (iii) uso educacional, auxiliando desenvolvedores menos experientes a reconhecer padrões problemáticos; e (iv) complemento à revisão de código, fornecendo uma segunda perspectiva automatizada. A revisão manual continua indispensável para decisões críticas de refatoração.

A escolha do modelo deve considerar o contexto de aplicação. O *GPT-4o-mini*

mostra-se adequado para uso geral com restrições orçamentárias, oferecendo o melhor equilíbrio entre qualidade e custo. O *Claude* pode ser preferido em cenários que priorizem cobertura máxima e tolerem mais falsos positivos. O *DeepSeek* destaca-se quando a precisão for crítica e falsos positivos precisem ser minimizados.

O campo de aplicação de LLMs em engenharia de software evolui rapidamente. Novos modelos com capacidades aprimoradas são lançados com frequência, e técnicas de *prompting* continuam sendo refinadas pela comunidade. Este trabalho contribui com uma base metodológica e resultados empíricos que podem servir de referência para futuras investigações nessa área promissora.

REFERÊNCIAS

ABDOU, A.; RAMADAN, N. Severity classification of software code smells using machine learning techniques: A comparative study. **Journal of Software: Evolution and Process**, v. 34, p. 37, 04 2022.

ALAZBA, A.; ALJAMAAN, H. Code smell detection using feature selection and stacking ensemble: An empirical investigation. **Information and Software Technology**, v. 138, p. 106648, 2021. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584921001129. Acesso em: 29 jan. 2026.

ALMEIDA, Y.; ALBUQUERQUE, D.; FILHO, E. D.; MUNIZ, F.; de Farias Santos, K.; PERKUSICH, M.; ALMEIDA, H.; PERKUSICH, A. Aicodereview: Advancing code quality with ai-enhanced reviews. **SoftwareX**, v. 26, p. 101677, 2024. ISSN 2352-7110. Disponível em: https://www.sciencedirect.com/science/article/pii/S2352711024000487. Acesso em: 29 jan. 2026.

AZEEM, M. I.; PALOMBA, F.; SHI, L.; WANG, Q. Machine learning techniques for code smell detection: A systematic literature review and meta-analysis. **Information and Software Technology**, Elsevier, v. 108, p. 115–138, 2019.

BROWN, T. B.; MANN, B.; RYDER, N.; SUBBIAH, M.; KAPLAN, J.; DHARIWAL, P.; NEELAKANTAN, A.; SHYAM, P.; SASTRY, G.; ASKELL, A.; AGARWAL, S.; HERBERT-VOSS, A.; KRUEGER, G.; HENIGHAN, T.; CHILD, R.; RAMESH, A.; ZIEGLER, D. M.; WU, J.; WINTER, C.; HESSE, C.; CHEN, M.; SIGLER, E.; LITWIN, M.; GRAY, S.; CHESS, B.; CLARK, J.; BERNER, C.; MCCANDLISH, S.; RADFORD, A.; SUTSKEVER, I.; AMODEI, D. Language models are few-shot learners. In: **Proceedings of the 34th International Conference on Neural Information Processing Systems (NIPS'20)**. Red Hook, NY, USA: Curran Associates Inc., 2020. p. 1877–1901. Introduz o conceito de Few-Shot Learning com GPT-3.

CAIN, W. Prompting change: Exploring prompt engineering in large language model ai and its potential to transform education. **TechTrends**, v. 68, p. 47–57, 2024. Disponível em: https://doi.org/10.1007/s11528-023-00896-0.

CHEN, W.; SU, Y.; ZUO, J.; YANG, C.; YUAN, C.; CHAN, C.-M.; YU, H.; LU, Y.; HUNG, Y.-H.; QIAN, C.; QIN, Y.; CONG, X.; XIE, R.; LIU, Z.; SUN, M.; ZHOU, J. **AgentVerse: Facilitating Multi-Agent Collaboration and Exploring Emergent Behaviors**. 2023. Disponível em: https://arxiv.org/abs/2308.10848. Acesso em: 29 jan. 2026.

CHEN, Z.; CHEN, L.; MA, W.; XU, B. Detecting code smells in python programs. In: IEEE. **2016 International Conference on Software Analysis, Testing and Evolution (SATE)**. [*S. l.*], 2016. p. 18–23.

FONTANA, F. A.; MäNTYLä, M. V.; ZANONI, M.; MARINO, A. Comparing and experimenting machine learning techniques for code smell detection. In: **Empirical Software Engineering**. [*S. l.*]: Springer, 2016. v. 21, n. 3, p. 1143–1191.

FOWLER, M. **Refactoring: Improving the Design of Existing Code**. 2. ed. Boston, MA: Addison-Wesley Professional, 2018. Segunda edição atualizada com exemplos em JavaScript. ISBN 978-0134757599.

FOWLER, M.; BECK, K.; BRANT, J.; OPDYKE, W.; ROBERTS, D. **Refactoring: Improving the Design of Existing Code**. Boston, MA: Addison-Wesley Professional, 1999. Primeira documentação sistemática de code smells como Long Method, Long Parameter List, Feature Envy, Data Clumps, entre outros. ISBN 978-0201485677.

ISHIBASHI, Y.; NISHIMURA, Y. **Self-Organized Agents: A LLM Multi-Agent Framework toward Ultra Large-Scale Code Generation and Optimization**. 2024. Disponível em: https://arxiv.org/abs/2404.02183. Acesso em: 29 jan. 2026.

JEONG, J.; GIL, D.; KIM, D.; JEONG, J. Current research and future directions for off-site construction through langchain with a large language model. **Buildings**, v. 14, n. 8, 2024. ISSN 2075-5309. Disponível em: https://www.mdpi.com/2075-5309/14/8/2374. Acesso em: 29 jan. 2026.

LACERDA, G.; PETRILLO, F.; PIMENTA, M.; GUéHéNEUC, Y.-G. Code smells and refactoring: A tertiary systematic review of challenges and observations. **Journal of Systems and Software**, Elsevier, v. 167, p. 110610, 2020.

LangChain. **LangGraph: Build Stateful Multi-Actor Applications with LLMs**. 2024. Documentação oficial. Disponível em: https://langchain-ai.github.io/langgraph/. Acesso em: 29 jan. 2026.

LI, X.; WANG, S.; ZENG, S.; WU, Y.; YANG, Y. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. **Vicinagearth**, v. 1, n. 1, p. 9, 2024. ISSN 3005-060X. Disponível em: https://doi.org/10.1007/s44336-024-00009-2.

LIEBERHERR, K. J.; HOLLAND, I. M. Object-oriented programming: An objective sense of style. **ACM SIGPLAN Notices**, ACM, v. 24, n. 10, p. 323–334, 1989. Introduz a Lei de Demeter, base para identificação do smell Long Message Chain.

LIU, N. F.; LIN, K.; HEWITT, J.; PARANJAPE, A.; BEVILACQUA, M.; PETRONI, F.; LIANG, P. Lost in the middle: How language models use long contexts. In: **Transactions of the Association for Computational Linguistics**. [*S. l.*]: MIT Press, 2024. v. 12, p. 157–173.

LIU, P.; YUAN, W.; FU, J.; JIANG, Z.; HAYASHI, H.; NEUBIG, G. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. **ACM Computing Surveys**, ACM, v. 55, n. 9, p. 1–35, 2023.

MARTIN, R. C. **Clean Code: A Handbook of Agile Software Craftsmanship**. Upper Saddle River, NJ: Prentice Hall, 2008. Documenta smells como Long Method, Too Many Arguments, Magic Numbers, e estabelece princípios de código limpo. ISBN 978-0132350884.

MARVIN, G.; HELLEN, N.; JJINGO, D.; NAKATUMBA-NABENDE, J. Prompt engineering in large language models. In: JACOB, I. J.; PIRAMUTHU, S.; FALKOWSKI-GILSKI, P. (Ed.). **Data Intelligence and Cognitive Informatics**. Singapore: Springer Nature Singapore, 2024. p. 387–402. ISBN 978-981-99-7962-2.

Microsoft. **Scheduler Agent Supervisor pattern**. 2024. Azure Architecture Center. Disponível em: https://learn.microsoft.com/en-us/azure/architecture/patterns/scheduler-agent-supervisor. Acesso em: 29 jan. 2026.

MäNTYLä, M. V.; VANHANEN, J.; LASSENIUS, C. A taxonomy and an initial empirical study of bad smells in code. **International Conference on Software Maintenance**, IEEE, p. 381–384, 2003. Taxonomia de code smells incluindo Long Method, Long Parameter List, e Magic Numbers.

NUNEZ, A.; ISLAM, N. T.; JHA, S. K.; NAJAFIRAD, P. Autosafecoder: A multi-agent framework for securing llm code generation through static analysis and fuzz testing. **arXiv preprint arXiv:2409.10737**, 2024. Disponível em: https://arxiv.org/abs/2409.10737.

OpenAI. **Structured Outputs**. 2024. Disponível em: https://platform.openai.com/docs/guides/structured-outputs. Acesso em: 29 jan. 2026.

PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; FASANO, F.; OLIVETO, R.; LUCIA, A. D. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. **Empirical Software Engineering**, Springer, v. 23, n. 3, p. 1188–1221, 2018.

PEI, C.; WANG, Z.; LIU, F.; LI, Z.; LIU, Y.; HE, X.; KANG, R.; ZHANG, T.; LI, J.; CHEN, J.; XIE, G.; PEI, D. Flow-of-action: Sop enhanced llm-based multi-agent system for root cause analysis. **OpenReview**, 2024. ICLR 2025 Conference Submission. Disponível em: https://openreview.net/forum?id=X7dQuJqs8c.

PELLURU, K. Langchain & langgraph in production: Architectures for multi-agent llm systems. **Journal of Data and Digital Innovation**, v. 2, n. 3, p. 1–9, 2025.

RASHEED, Z.; SAMI, M. A.; WASEEM, M.; KEMELL, K.-K.; WANG, X.; NGUYEN, A.; SYSTÄ, K.; ABRAHAMSSON, P. Ai-powered code review with llms: Early results. **arXiv preprint arXiv:2404.18496**, 2024. Disponível em: https://doi.org/10.48550/arXiv.2404.18496. Acesso em: 29 jan. 2026.

REYNOLDS, L.; MCDONELL, K. Prompt programming for large language models: Beyond the few-shot paradigm. **Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems**, ACM, p. 1–7, 2021.

SAAVEDRA, N.; FERREIRA, J. F. Glitch: Automated polyglot security smell detection in infrastructure as code. **arXiv preprint arXiv:2205.14371**, 2022. Disponível em: https://arxiv.org/abs/2205.14371.

SADIK, A. R.; GOVIND, S. **Benchmarking LLM for Code Smells Detection: OpenAI GPT-4.0 vs DeepSeek-V3**. 2025. Disponível em: https://arxiv.org/abs/2504.16027. Acesso em: 29 jan. 2026.

SAHOO, P.; SINGH, A. K.; SAHA, S.; JAIN, V.; MONDAL, S.; CHADHA, A. A systematic survey of prompt engineering in large language models: Techniques and applications. **arXiv preprint arXiv:2402.07927**, 2024.

SHARMA, T. Dpy: Code smells detection tool for python. In: **Proceedings of the 22nd International Conference on Mining Software Repositories (MSR)**. [*S. l.*]: IEEE/ACM, 2025. Ferramenta que detecta 8 design smells e 11 implementation smells em Python, incluindo Long Method, Complex Method, Long Parameter List, Long Statement, Long Identifier, Magic Number, Long Message Chain, Empty Catch Block, Missing Default, Long Lambda Function.

SHU, H.; FU, M.; YU, J.; WANG, D.; TANTITHAMTHAVORN, C.; CHEN, J.; KAMEI, Y. **Large Language Models for Multilingual Vulnerability Detection: How Far Are We?** 2025. Disponível em: https://arxiv.org/abs/2506.07503. Acesso em: 29 jan. 2026.

SILVA, L. L.; SILVA, J. R. d.; MONTANDON, J. E.; ANDRADE, M.; 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**. New York, NY, USA: Association for Computing Machinery, 2024. (ESEM '24), p. 400–406. ISBN 9798400710476. Disponível em: https://doi.org/10.1145/3674805.3690742. Acesso em: 29 jan. 2026.

TANDON, S.; KUMAR, V.; SINGH, V. B. Study of code smells: A review and research agenda. **International Journal of Mathematical, Engineering and Management Sciences**, Ram Arti Publishers, v. 9, n. 3, p. 472–498, 2024.

TRAN, K.-T.; DAO, D.; NGUYEN, M.-D.; PHAM, Q.-V.; O'SULLIVAN, B.; NGUYEN, H. D. **Multi-Agent Collaboration Mechanisms: A Survey of LLMs**. 2025. Disponível em: https://arxiv.org/abs/2501.06322. Acesso em: 29 jan. 2026.

WEI, J.; WANG, X.; SCHUURMANS, D.; BOSMA, M.; ICHTER, B.; XIA, F.; CHI, E. H.; LE, Q. V.; ZHOU, D. Chain-of-thought prompting elicits reasoning in large language models. In: **Proceedings of the 36th International Conference on Neural Information Processing Systems (NIPS'22)**. Red Hook, NY, USA: Curran Associates Inc., 2022. p. 24824–24837. Introduz a técnica Chain-of-Thought para raciocínio em LLMs.

WEN, X.-C.; YANG, Y.; GAO, C.; XIAO, Y.; YE, D. **Boosting Vulnerability Detection of LLMs via Curriculum Preference Optimization with Synthetic Reasoning Data**. 2025. Disponível em: https://arxiv.org/abs/2506.07390. Acesso em: 29 jan. 2026.

WHITE, J.; FU, Q.; HAYS, S.; SANDBORN, M.; OLEA, C.; GILBERT, H.; ELNASHAR, A.; SPENCER-SMITH, J.; SCHMIDT, D. C. **A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT**. 2023. Disponível em: https://arxiv.org/abs/2302.11382. Acesso em: 29 jan. 2026.

WU, Q.; BANSAL, G.; ZHANG, J.; WU, Y.; LI, B.; ZHU, E.; JIANG, L.; ZHANG, X.; ZHANG, S.; LIU, J.; AWADALLAH, A. H.; WHITE, R. W.; BURGER, D.; WANG, C. **AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation**. 2023. Disponível em: https://arxiv.org/abs/2308.08155. Acesso em: 29 jan. 2026.

ZHANG, Q.; FANG, C.; XIE, Y.; ZHANG, Y.; YANG, Y.; SUN, W.; YU, S.; CHEN, Z. A survey on large language models for software engineering. **arXiv preprint arXiv:2312.15223**, 2023. Disponível em: https://arxiv.org/abs/2312.15223.

glossario-uece

GLOSSÁRIO

Architecture Smell Categoria de *code smells* que representa problemas estruturais no nível arquitetural do sistema, como dependências cíclicas entre componentes ou violações de camadas

Chain-of-Thought Técnica que encoraja o modelo a explicitar seu processo de raciocínio passo a passo antes de chegar à resposta final, melhorando a precisão em tarefas que requerem raciocínio complexo

Code Smell Sintoma de problema de projeto no código-fonte que, embora não represente um erro que impeça a execução do software, sinaliza trechos de código com baixa legibilidade e alta complexidade, dificultando sua manutenção e extensão

DPy Ferramenta de avaliação de qualidade de código para *Python*, derivada do *Designite*. Oferece detecção de *architecture smells*, *design smells*, *implementation smells* e métricas orientadas a objetos. Também inclui detecção de *smells* específicos de *machine learning*. O nome é pronunciado como /di:pai/

Design Smell Categoria de *code smells* que indica problemas no projeto de classes e suas relações, como acoplamento excessivo ou violações de princípios de design orientado a objetos

F1-Score Média harmônica entre precisão e *recall*, fornecendo uma métrica única que equilibra ambas as medidas. Calculada como $2 \times \frac{Preciso \times Recall}{Preciso + Recall}$

Few-Shot Learning Técnica de engenharia de *prompts* que consiste em fornecer ao modelo alguns exemplos demonstrativos dentro do próprio *prompt*, permitindo que ele aprenda o padrão desejado a partir de poucos casos

Implementation Smells Categoria de *code smells* que engloba problemas localizados no nível de métodos e funções individuais, em contraste com *smells* de design ou arquitetura

LangChain *Framework* de código aberto que oferece abstrações e componentes modulares para construção de aplicações baseadas em LLMs, facilitando a integração com fontes de dados externas

LangGraph Extensão do *LangChain* que permite modelar aplicações multiagente como grafos de estados, onde agentes são representados como nós e as transições como arestas

Long Message Chain *Code smell* que ocorre quando um objeto solicita outro objeto, que solicita outro, e assim sucessivamente, violando a Lei de Demeter

Long Parameter List *Code smell* que descreve funções que recebem número excessivo de

parâmetros, tornando sua chamada complexa e propensa a erros

Magic Number *Code smell* que se refere ao uso de valores literais numéricos no código sem explicação ou atribuição a constantes nomeadas

Prompt Instrução ou conjunto de instruções fornecidas a um modelo de linguagem para guiar sua resposta. Pode incluir contexto, exemplos e especificações de formato

Pydantic Biblioteca Python para validação de dados usando anotações de tipo, frequentemente utilizada para definir *schemas* de entrada e saída em aplicações

Recall Métrica que mede a proporção de casos positivos corretamente identificados em relação ao total de casos positivos existentes. Também conhecida como sensibilidade ou taxa de verdadeiros positivos. Calculada como $\frac{VP}{VP+FN}$

Structured Output Capacidade de LLMs de gerar respostas em formatos estruturados pre-definidos (como JSON), seguindo esquemas específicos que permitem processamento programático

Token Unidade básica de processamento de texto em modelos de linguagem. Um *token* pode representar uma palavra, parte de uma palavra ou caractere, dependendo do tokenizador utilizado

Análise Estática Técnica de verificação de código-fonte realizada sem executar o programa, utilizando ferramentas automatizadas para identificar padrões problemáticos, vulnerabilidades e violações de estilo

Complexidade Ciclomática Métrica de software que quantifica a complexidade de um programa medindo o número de caminhos linearmente independentes através do código-fonte

CoT *Chain-of-Thought* (Cadeia de Pensamento)

Dívida Técnica Metáfora que representa o custo implícito de retrabalho futuro causado por escolhas de implementação subótimas feitas para ganhar velocidade no curto prazo

Engenharia de *Prompts* Processo de projetar e otimizar as instruções fornecidas aos LLMs para obter respostas mais precisas e relevantes, sem modificar os parâmetros internos do modelo

Falso Positivo Detecção incorreta onde o sistema identifica um problema que na realidade não existe. No contexto de *code smells*, ocorre quando o sistema reporta um *smell* em um código que não apresenta o problema

JSON *JavaScript Object Notation*

Lei de Demeter Princípio de design de software que estabelece que um objeto deve ter conheci-

mento limitado sobre outros objetos, comunicando-se apenas com seus vizinhos imediatos. Também conhecida como *Law of Demeter* ou Princípio do Menor Conhecimento

LLM *Large Language Models* (Modelos de Linguagem de Grande Escala)

Padrão *Supervisor* Arquitetura de sistemas multiagente onde um agente orquestrador coordena o fluxo de trabalho, delegando tarefas aos agentes especialistas e consolidando os resultados

Padrão de Referência Conjunto de dados de referência considerado como verdade absoluta para fins de avaliação e validação de sistemas automatizados. Também conhecido como *ground truth* na literatura

Precisão Métrica que mede a proporção de detecções corretas (verdadeiros positivos) em relação ao total de detecções realizadas pelo sistema. Calculada como $\frac{VP}{VP+FP}$

Refatoração Processo de reestruturação do código existente sem alterar seu comportamento externo, visando melhorar sua estrutura interna, legibilidade e manutenibilidade

Sistema Multiagente Arquitetura de software composta por múltiplos agentes autônomos que colaboram para resolver problemas complexos, onde cada agente pode ser especializado em uma tarefa específica

APÊNDICE A – *PROMPTS* ELABORADOS DOS AGENTES

Este apêndice apresenta os *prompts* elaborados utilizados pelos onze agentes especializados do sistema. Cada *prompt* incorpora técnicas de *few-shot learning* (exemplos positivos e negativos) e *chain-of-thought* (processo de raciocínio estruturado).

A.1 *Complex Method Agent*

Código-fonte 3: *Prompt* elaborado do agente *Complex Method*.

```
"""Você detecta Complex Method (Complexidade Ciclomática > 7).
Referência: McCabe (1976).

## DEFINIÇÃO:
Função excessivamente complexa em fluxo de controle. CC = 1 +
   pontos de decisão.
Pontos de decisão: if, elif, for, while, except, and, or

O QUE É: Funções/métodos definidos com 'def nome():'
O QUE NÃO É: Scripts de módulo, atribuições simples, chamadas de
    função

## PROCESSO:
1. Para cada função 'def', identifique start_line e end_line
2. Conte pontos de decisão (if, elif, for, while, except, and,
   or) dentro do método
3. CC = 1 + pontos de decisão
4. Se CC > 7: adicione à lista
5. Retorne no máximo 10 detecções
6. SEMPRE retorne start_line e end_line; Line_no deve ser vazio
   ("")

## EXEMPLO:
def validate(data):  # linha 10, CC=9
    if data:  # +1
```

```
        if data.valid:  # +1
            if data.type == "A":  # +1
                if data.status:  # +1
                    for item in data.items:  # +1
                        if item.checked:  # +1
                            if item.value > 0:  # +1
                                return True
    return False
# CC = 1 + 8 = 9

## EXEMPLO NEGATIVO (NÃO É SMELL):
def process_item(item):  # linha 5, CC=7
    if item:  # +1
        if item.valid:  # +1
            for i in range(5):  # +1
                if i > 0:  # +1
                    if item.value:  # +1
                        if item.status:  # +1
                            return True
    return False
# CC = 1 + 6 = 7
Análise: CC = 7. Como threshold é > 7 (não >= 7), NÃO é smell.

## REGRAS:
1. APENAS funções DEFINIDAS com 'def' no código fornecido
2. NÃO detecte funções apenas chamadas/importadas
3. cyclomatic_complexity = número inteiro (nunca "unknown")
4. IMPORTANTE: Métodos com CC = 7 NÃO são smells (threshold: >
   7, não >= 7)
5. Se não conseguir calcular CC, NÃO inclua a detecção
6. Line_no = "" (vazio) para smells de método
7. SEMPRE retorne start_line e end_line
"""
```

Fonte: Elaborado pelo autor.

A.2 *Long Method Agent*

Código-fonte 4: *Prompt* elaborado do agente *Long Method*.

```
"""Você detecta Long Method (métodos com > 67 linhas).
Referência: Fowler (1999) - Refactoring.

## DEFINIÇÃO:
Método longo = muitas linhas de código, dificulta compreensão e
   manutenção.

O QUE É: Funções/métodos definidos com 'def nome():'
O QUE NÃO É: Scripts de módulo, variáveis, classes

## PROCESSO:
1. Para cada função 'def', identifique start_line (linha do 'def
   ') e end_line (última linha)
2. Calcule: end_line - start_line + 1
3. Se > 67 linhas: adicione à lista
4. Retorne no máximo 10 detecções
5. SEMPRE retorne start_line e end_line; Line_no deve ser vazio
   ("")

## EXEMPLO:
def process_order(order):  # linha 1, 70 linhas
    # 70 linhas de código...
    validate()
    calculate()
    save()

## REGRAS:
1. APENAS funções DEFINIDAS com 'def' no código fornecido
2. NÃO detecte funções apenas chamadas/importadas
```

```
3. total_lines = número inteiro (nunca "Unknown")
4. Line_no = "" (vazio) para smells de método
5. SEMPRE retorne start_line e end_line
"""
```

Fonte: Elaborado pelo autor.

A.3 *Complex Conditional Agent*

Código-fonte 5: *Prompt* elaborado do agente *Complex Conditional*.

```
"""Você detecta Complex Conditional (condicionais com > 2
   operadores lógicos and/or).
Referência: Fowler (2018).

## DEFINIÇÃO:
Condicional (if, elif, while) com mais de 2 operadores lógicos (
   and, or).

O QUE É: Expressões em if/elif/while com > 2 operadores and/or
   na MESMA condição
O QUE NÃO É:
- Condicionais com 1 ou 2 operadores (são aceitáveis)
- Atribuições (x = a and b)
- Operadores dentro de list comprehensions
- Múltiplos ifs separados

## PROCESSO:
1. Identifique linhas com if, elif, while
2. Conte APENAS operadores 'and' e 'or' na condição
3. Se > 2 operadores (3 ou mais): adicione à lista
4. Retorne no máximo 10 detecções

## EXEMPLO POSITIVO (É SMELL):
```

```
def check_eligibility(user):
    # 3 operadores and - É SMELL (> 2)
    if user.age > 18 and user.verified and user.balance > 100
       and user.active:
        return True

## EXEMPLOS NEGATIVOS (NÃO SÃO SMELLS):

### Exemplo 1: Apenas 1 operador (aceitável)
if user.active and user.verified:  # 1 operador - NÃO É SMELL
    process(user)

### Exemplo 2: Exatamente 2 operadores (no limite, aceitável)
if a > 0 and b > 0 and c > 0:  # 2 operadores - NÃO É SMELL
    return True

### Exemplo 3: Atribuições (NUNCA são smells)
result = a and b and c and d and e  # NÃO É SMELL - é atribuição

## REGRAS CRÍTICAS:
1. APENAS conte operadores em if, elif, while - NÃO em atribuiçõ
   es
2. Threshold é > 2 (mais que 2 operadores = 3 ou mais)
3. 1 ou 2 operadores são ACEITÁVEIS - não detecte
4. Máximo 10 detecções
"""
```

Fonte: Elaborado pelo autor.

A.4 *Long Parameter List Agent*

Código-fonte 6: *Prompt* elaborado do agente *Long Parameter List*.

```
"""Você detecta Long Parameter List (funções/métodos com > 4 parâmetros).
Referência: Fowler (1999).

## DEFINIÇÃO:
Função ou método com número excessivo de parâmetros (> 4).

O QUE É: Funções/métodos com 'def' que têm estritamente > 4 parâmetros
O QUE NÃO É:
- Funções com exatamente 4 parâmetros (NÃO é smell, threshold é > 4, não >= 4)
- Funções com <= 4 parâmetros
- *args/**kwargs (não contam como parâmetros)
- self/cls em métodos (não contam)
- Valores default contam normalmente

## PROCESSO:
1. Conte parâmetros de cada função (exceto self, cls, *args, **kwargs)
2. Se > 4: adicione à lista
3. Retorne no máximo 10 detecções

## EXEMPLO:
def create_user(name, email, age, address, phone, country):  # linha 5, 6 params
    pass

## EXEMPLO NEGATIVO (NÃO É SMELL):
def create_user(name, email, age, address):  # linha 5, exatamente 4 parâmetros
    pass
Análise: Função tem exatamente 4 parâmetros. Como threshold é > 4 (não >= 4), NÃO é smell.
```

```
28
29 ## REGRAS:
30 1. Conte parâmetros (exceto self, cls, *args, **kwargs)
31 2. Parâmetros com default contam normalmente
32 3. IMPORTANTE: Funções com exatamente 4 parâmetros NÃO são
      smells
33 4. Só detecte se estritamente > 4 parâmetros
34 """
```

Fonte: Elaborado pelo autor.

A.5 *Long Statement Agent*

Código-fonte 7: *Prompt* elaborado do agente *Long Statement*.

```
1 """Você detecta Long Statement (linhas com > 120 caracteres).
2 Referência: PEP 8.
3
4 ## DEFINIÇÃO:
5 Linha de código que excede 120 caracteres (inclui código, coment
      ários, strings).
6
7 O QUE É: Qualquer linha com estritamente > 120 caracteres
8 O QUE NÃO É:
9 - Linhas com exatamente 120 caracteres (NÃO é smell, threshold é
       > 120)
10 - Linhas <= 120 caracteres
11
12 ## PROCESSO:
13 1. Para cada linha, conte caracteres visíveis (letras, números,
      símbolos, espaços)
14 2. NÃO conte \n no final (use len(line.rstrip()))
15 3. Se > 120 caracteres: adicione à lista
16 4. Retorne no máximo 10 detecções (priorize as mais longas)
```

```
17
18 ## FORMATO:
19 Código vem com numeração: "  7 | código aqui"
20 - Line_no = número à esquerda do "|" (ex: "7")
21 - Conte caracteres APENAS do código (à direita do "|"), sem
      numeração nem "|"
22 - line_length = comprimento real do código (sem \n)
23
24 ## EXEMPLO NEGATIVO (NÃO É SMELL):
25   5 | result = some_function(arg1, arg2, arg3)  # exatamente 120
         chars
26 Análise: Linha tem exatamente 120 caracteres. NÃO é smell.
27
28 ## REGRAS:
29 1. Use número à esquerda do "|" como Line_no
30 2. Conte apenas código (direita do "|")
31 3. Não conte numeração, "|" nem \n
32 4. IMPORTANTE: Linhas com exatamente 120 caracteres NÃO são
      smells
33 5. Só detecte se estritamente > 120 caracteres
34 """
```

Fonte: Elaborado pelo autor.

A.6 *Long Identifier Agent*

Código-fonte 8: *Prompt* elaborado do agente *Long Identifier*.

```
1 """Você detecta Long Identifier (nomes com > 20 caracteres).
2 Referência: Martin (2008).
3
4 ## DEFINIÇÃO:
5 Identificador (função, classe, variável, constante) com nome >
     20 caracteres.
```

```
6
7  ## PROCESSO:
8  1. Encontre nomes de variáveis, funções, classes, constantes
       DEFINIDOS no código
9  2. Conte os caracteres de cada nome
10 3. Se > 20 caracteres: adicione à lista
11 4. Retorne no máximo 10 detecções
12
13 ## EXEMPLO POSITIVO (É SMELL):
14 DESKTOP_ENVIRONMENT_CONFIG_NAME = "config"  # 31 chars - É SMELL
15 very_long_variable_name_here = 42  # 28 chars - É SMELL
16 calculate_total_price_with_tax = lambda x: x  # 31 chars - É
       SMELL
17
18 ## EXEMPLO NEGATIVO (NÃO É SMELL):
19 user_name = "John"  # 9 chars - NÃO É SMELL
20 total_count = 0  # 11 chars - NÃO É SMELL
21
22 ## REGRAS:
23 1. Conte caracteres do nome (sem espaços)
24 2. Threshold: > 20 caracteres
25 3. IGNORE: imports, strings literais, dunder methods (__init__)
26 4. Máximo 10 detecções
27 """
```

Fonte: Elaborado pelo autor.

A.7 *Magic Number Agent*

Código-fonte 9: *Prompt* elaborado do agente *Magic Number*.

```
1  """Você detecta Magic Number (literais numéricos sem constante
       nomeada).
2  Referência: Fowler (1999) + Martin (2008).
```

```
3
4  ## DEFINIÇÃO:
5  Literal numérico usado diretamente sem constante que explique
       seu significado.
6
7  O QUE É: Números literais (exceto valores triviais) sem
       constante nomeada
8  O QUE NÃO É:
9  - Valores triviais: 0, 1, -1, 2, -2, 10, 100, 0,0, 1,0, -1,0,
       2,0, -2,0
10 - Constantes já definidas (PI, E, MAX_SIZE, etc.)
11 - Índices de array/lista (arr[0], list[1])
12 - Versões (version = "1,0,0")
13 - Contadores simples em loops (for i in range(10))
14 - Valores em testes unitários
15 - Contextos matemáticos comuns (pi ~ 3,14, e ~ 2,71)
16
17 ## PROCESSO:
18 1. Encontre literais numéricos no código
19 2. Ignore valores triviais (0, 1, -1, 2, -2, 10, 100, 0,0, 1,0,
       -1,0, 2,0, -2,0)
20 3. Ignore constantes já definidas e contextos apropriados
21 4. Verifique contexto (não detecte em testes, versões, índices)
22 5. Retorne no máximo 10 detecções
23
24 ## EXEMPLO:
25 def calculate_energy(mass, height):  # linha 10
26     return mass * 9,81 * height  # linha 11
27
28 ## EXEMPLO NEGATIVO (NÃO É SMELL):
29 def process_items(items):  # linha 10
30     for i in range(10):  # linha 11 - 10 é contador, NÃO é magic
           number
```

```
31          if items[i] > 0:  # linha 12 - 0 é trivial, NÃO é magic
               number
32              result = items[i] * 2  # linha 13 - 2 é trivial, NÃO
                   é magic number
33      return items[0]  # linha 14 - 0 é índice, NÃO é magic number
34
35  ## REGRAS:
36  1. Ignore valores triviais: 0, 1, -1, 2, -2, 10, 100
37  2. Ignore constantes já definidas
38  3. Ignore índices de array/lista
39  4. Ignore valores em testes unitários
40  5. Ignore versões e contadores simples em loops
41  """
```

Fonte: Elaborado pelo autor.

A.8 *Empty Catch Block Agent*

Código-fonte 10: *Prompt* elaborado do agente *Empty Catch Block*.

```
1  """Você detecta Empty Catch Block (except vazio ou só com pass).
2  Referência: Martin (2008).
3
4  ## DEFINIÇÃO:
5  Bloco except que não trata a exceção (vazio ou apenas 'pass'),
      silenciando erros.
6
7  O QUE É: except: pass, except Exception: pass, except ValueError
      : (vazio)
8  O QUE NÃO É: except com logging, re-raise, ou tratamento (return
       default_value)
9
10 ## PROCESSO:
11 1. Encontre blocos try-except
```

```
2. Verifique se except está vazio ou só tem 'pass'
3. Retorne no máximo 10 detecções

## EXEMPLO:
def load_data():  # linha 10
    try:
        risky_operation()
    except Exception:  # linha 13
        pass

## REGRAS:
1. Detecte apenas except vazio ou com 'pass'
2. Ignore except com tratamento (logging, re-raise, return)
"""
```

Fonte: Elaborado pelo autor.

A.9 *Missing Default Agent*

Código-fonte 11: *Prompt* elaborado do agente *Missing Default*.

```
"""Você detecta Missing Default (match-case sem case _).
Referência: CWE-478 (MITRE).

## DEFINIÇÃO:
Bloco match-case sem caso padrão (case _), podendo causar
   comportamento indefinido.

O QUE É: match sem case _ (default)
O QUE NÃO É: match com case _, if-elif com else

## PROCESSO:
1. Encontre blocos match-case (Python 3.10+)
2. Verifique se tem 'case _:' (default)
```

```
3. Se não tem: adicione à lista
4. Retorne no máximo 10 detecções

## EXEMPLO:
def handle_status(status):  # linha 10
    match status:  # linha 11
        case "active":
            activate()
        case "inactive":
            deactivate()

## REGRAS:
1. Detecte apenas match-case sem case _
2. Ignore match-case com case _ ou if-elif com else
"""
```

Fonte: Elaborado pelo autor.

A.10 *Long Lambda Function Agent*

Código-fonte 12: *Prompt* elaborado do agente *Long Lambda Function*.

```
"""Você detecta Long Lambda Function (lambdas com > 80
   caracteres).
Referência: Chen et al. (2016).

## DEFINIÇÃO:
Função lambda com expressão > 80 caracteres, prejudicando
   legibilidade.

O QUE É: lambda com estritamente > 80 caracteres
O QUE NÃO É:
- lambda com exatamente 80 caracteres (NÃO é smell, threshold é
   > 80, não >= 80)
```

```
- lambda <= 80 caracteres
- funções nomeadas (def)

## PROCESSO:
1. Encontre expressões lambda no código
2. Conte caracteres da expressão completa
3. Se > 80: adicione à lista
4. Retorne no máximo 10 detecções

## EXEMPLO:
result = map(lambda x: x * 2 if x > 0 else x * -1 if x < 0 else
    0 if x == 0 else x + 10, numbers)  # 95 chars

## EXEMPLO NEGATIVO (NÃO É SMELL):
result = map(lambda x: x * 2 if x > 0 else x * -1, numbers)  #
    exatamente 80 chars
Análise: Lambda tem exatamente 80 caracteres. Como threshold é >
     80, NÃO é smell.

## REGRAS:
1. Conte caracteres da expressão lambda completa
2. IMPORTANTE: Lambdas com exatamente 80 caracteres NÃO são
    smells
3. Ignore funções nomeadas (def)
4. Só detecte se estritamente > 80 caracteres
"""
```

Fonte: Elaborado pelo autor.

A.11 *Long Message Chain Agent*

Código-fonte 13: *Prompt* elaborado do agente *Long Message Chain*.

```
"""Você detecta Long Message Chain (> 2 métodos encadeados).
```

```
Referência: Fowler (1999).

## DEFINIÇÃO:
Cadeia de chamadas de métodos encadeados (> 2), violando Lei de
   Demeter.

## PROCESSO:
1. Encontre padrões: obj.method1().method2().method3()
2. Conte métodos com parênteses encadeados
3. Se > 2 métodos encadeados: adicione à lista
4. Retorne no máximo 10 detecções

## EXEMPLO POSITIVO (É SMELL):
# 3 métodos encadeados
zip_code = customer.get_address().get_city().get_zip_code()
data = response.json().get('data').values()
result = df.filter().sort().head()

## EXEMPLO NEGATIVO (NÃO É SMELL):
# Apenas 2 métodos (aceitável)
result = data.filter().sort()
text = string.strip().lower()

# Acesso a atributos (sem parênteses)
value = config.settings.database.host

## REGRAS:
1. Conte métodos com parênteses: .method()
2. Atributos sem parênteses não contam
3. Threshold: > 2 métodos encadeados
4. Máximo 10 detecções
"""
```

Fonte: Elaborado pelo autor.

APÊNDICE B – *PROMPTS* SIMPLES DOS AGENTES

Este apêndice apresenta os *prompts* simples utilizados pelos onze agentes especializados do sistema. Diferentemente dos *prompts* elaborados (Apêndice A), estes contêm apenas a definição básica do *smell* e um exemplo de detecção, sem técnicas de *few-shot learning* ou *chain-of-thought*.

B.1 *Complex Method Agent*

Código-fonte 14: *Prompt* simples do agente *Complex Method*.

```
"""Detecte Complex Method: funções com Complexidade Ciclomática > 7.
Complexidade Ciclomática = 1 + número de pontos de decisão (if, elif, for, while, except, and, or).

Exemplo:
def validate(data):  # linha 10
    if data:
        if data.valid:
            if data.type == "A":
                if data.status:
                    for item in data.items:
                        if item.checked:
                            if item.value > 0:
                                return True
    return False

Saída esperada:
{
  "detected": true,
  "detections": [{
    "Smell": "Complex method",
    "Method": "validate",
    "Line_no": "",
```

```
24        "cyclomatic_complexity": 9,
25        "threshold": 7,
26        "start_line": 10,
27        "end_line": 17
28      }]
29    }
30
31    IMPORTANTE: Para smells de método, sempre retorne:
32    - Line_no vazio ("") - pois o smell se refere ao método
         inteiro
33    - start_line: linha onde o método começa (linha do 'def')
34    - end_line: última linha do método
35    """
```

Fonte: Elaborado pelo autor.

B.2 *Long Method Agent*

Código-fonte 15: *Prompt* simples do agente *Long Method*.

```
1   """Detecte Long Method: métodos com mais de 67 linhas.
2
3   Exemplo:
4   def process_order(order):  # linha 1, 70 linhas
5       # 70 linhas de código aqui...
6       validate()
7       calculate()
8       save()
9
10  Saída esperada:
11  {
12    "detected": true,
13    "detections": [{
14      "Smell": "Long method",
```

```
15      "Method": "process_order",
16      "Line_no": "",
17      "total_lines": 70,
18      "threshold": 67,
19      "start_line": 1,
20      "end_line": 70
21    }]
22  }
23
24  IMPORTANTE: Para smells de método, sempre retorne:
25  - Line_no vazio ("") - pois o smell se refere ao método
       inteiro
26  - start_line: linha onde o método começa (linha do 'def')
27  - end_line: última linha do método
28  """
```

Fonte: Elaborado pelo autor.

B.3 *Complex Conditional Agent*

Código-fonte 16: *Prompt* simples do agente *Complex Conditional*.

```
1   """Detecte Complex Conditional: condicionais (if, elif, while)
        com mais de
2   2 operadores lógicos (and/or).
3
4   Exemplo:
5   def check_eligibility(user):  # linha 10
6       if user.age > 18 and user.verified and user.balance > 100
           and user.active:  # linha 11
7           return True
8
9   Saída esperada:
10  {
```

```
11    "detected": true,
12    "detections": [{
13      "Smell": "Complex conditional",
14      "Method": "check_eligibility",
15      "Line_no": "11",
16      "logical_operators": 3,
17      "threshold": 2
18    }]
19  }
20  """
```

Fonte: Elaborado pelo autor.

B.4 *Long Parameter List Agent*

Código-fonte 17: *Prompt* simples do agente *Long Parameter List*.

```
1   """Detecte Long Parameter List: funções/métodos com mais de 4
       parâmetros
2   (exceto self, cls, *args, **kwargs).
3
4   Exemplo:
5   def create_user(name, email, age, address, phone, country):  #
        linha 5, 6 params
6       pass
7
8   Saída esperada:
9   {
10    "detected": true,
11    "detections": [{
12      "Smell": "Long parameter list",
13      "Method": "create_user",
14      "Line_no": "5",
15      "parameter_count": 6,
```

```
16        "threshold": 4
17      }]
18    }
19    """
```

Fonte: Elaborado pelo autor.

B.5 *Long Statement Agent*

Código-fonte 18: *Prompt* simples do agente *Long Statement*.

```
1   """Detecte Long Statement: linhas com mais de 120 caracteres.
2   O código será fornecido com numeração de linhas. Use o número
       à esquerda
3   do "|" como Line_no.
4
5   Exemplo:
6     1 | x = 1
7     2 | result = some_function(arg1, arg2, arg3, arg4, arg5,
         arg6, arg7)  # 150 chars
8
9   Saída esperada:
10  {
11    "detected": true,
12    "detections": [{
13      "Smell": "Long statement",
14      "Method": "",
15      "Line_no": "2",
16      "line_length": 150,
17      "threshold": 120
18    }]
19  }
20  """
```

Fonte: Elaborado pelo autor.

B.6 *Long Identifier Agent*

Código-fonte 19: *Prompt* simples do agente *Long Identifier*.

```
"""Detecte Long Identifier: nomes de funções, classes, variáveis ou
constantes com mais de 20 caracteres.

Exemplo:
DESKTOP_ENVIRONMENT_CONFIG_NAME = "config"  # linha 5, 31 chars

Saída esperada:
{
  "detected": true,
  "detections": [{
    "Smell": "Long identifier",
    "Method": "",
    "Line_no": "5",
    "identifier_name": "DESKTOP_ENVIRONMENT_CONFIG_NAME",
    "length": 31,
    "threshold": 20
  }]
}
"""
```

Fonte: Elaborado pelo autor.

B.7 *Magic Number Agent*

Código-fonte 20: *Prompt* simples do agente *Magic Number*.

```
"""Detecte Magic Number: literais numéricos (exceto 0, 1, -1) usados
diretamente no código sem constante nomeada.

Exemplo:
def calculate_energy(mass, height):  # linha 10
    return mass * 9,81 * height  # linha 11

Saída esperada:
{
  "detected": true,
  "detections": [{
    "Smell": "Magic number",
    "Method": "calculate_energy",
    "Line_no": "11",
    "Description": "Magic number 9,81 found. Define as GRAVITY_CONSTANT = 9,81."
  }]
}
"""
```

Fonte: Elaborado pelo autor.

B.8 *Empty Catch Block Agent*

Código-fonte 21: *Prompt* simples do agente *Empty Catch Block*.

```
"""Detecte Empty Catch Block: blocos except vazios ou apenas com 'pass',
que silenciam exceções.

Exemplo:
def load_data():  # linha 10
    try:
```

```
7           risky_operation()
8       except Exception:  # linha 13
9           pass
10
11  Saída esperada:
12  {
13    "detected": true,
14    "detections": [{
15      "Smell": "Empty catch block",
16      "Method": "load_data",
17      "Line_no": "13",
18      "Description": "Empty catch block at line 13. Add logging
            or error handling."
19    }]
20  }
21  """
```

Fonte: Elaborado pelo autor.

B.9 *Missing Default Agent*

Código-fonte 22: *Prompt* simples do agente *Missing Default*.

```
1   """Detecte Missing Default: blocos match-case sem caso padrão
        (case _).
2
3   Exemplo:
4   def handle_status(status):  # linha 10
5       match status:  # linha 11
6           case "active":
7               activate()
8           case "inactive":
9               deactivate()
10
```

```
Saída esperada:
{
  "detected": true,
  "detections": [{
    "Smell": "Missing default",
    "Method": "handle_status",
    "Line_no": "11",
    "Description": "Match-case at line 11 missing default case
       . Add 'case _:'."
  }]
}
"""
```

Fonte: Elaborado pelo autor.

B.10 *Long Lambda Function Agent*

Código-fonte 23: *Prompt* simples do agente *Long Lambda Function*.

```
"""Detecte Long Lambda Function: funções lambda com mais de 80
    caracteres.

Exemplo:
result = map(lambda x: x * 2 if x > 0 else x * -1 if x < 0
   else 0 if x == 0 else x + 10, numbers)  # 95 chars

Saída esperada:
{
  "detected": true,
  "detections": [{
    "Smell": "Long lambda function",
    "Method": "",
    "Line_no": "5",
    "lambda_length": 95,
```

```
14      "threshold": 80
15    }]
16  }
17  """
```

Fonte: Elaborado pelo autor.

B.11 *Long Message Chain Agent*

Código-fonte 24: *Prompt* simples do agente *Long Message Chain*.

```
1   """Detecte Long Message Chain: cadeias de mais de 2 métodos
       encadeados.
2
3   Exemplo:
4   def get_zip(customer):  # linha 10
5       return customer.get_address().get_city().get_zip_code()  #
            linha 11, 3 métodos
6
7   Saída esperada:
8   {
9     "detected": true,
10    "detections": [{
11      "Smell": "Long message chain",
12      "Method": "get_zip",
13      "Line_no": "11",
14      "chain_length": 3,
15      "threshold": 2
16    }]
17  }
18  """
```

Fonte: Elaborado pelo autor.

GLOSSÁRIO

Architecture Smell Categoria de *code smells* que representa problemas estruturais no nível arquitetural do sistema, como dependências cíclicas entre componentes ou violações de camadas

Chain-of-Thought Técnica que encoraja o modelo a explicitar seu processo de raciocínio passo a passo antes de chegar à resposta final, melhorando a precisão em tarefas que requerem raciocínio complexo

Code Smell Sintoma de problema de projeto no código-fonte que, embora não represente um erro que impeça a execução do software, sinaliza trechos de código com baixa legibilidade e alta complexidade, dificultando sua manutenção e extensão

DPy Ferramenta de avaliação de qualidade de código para *Python*, derivada do *Designite*. Oferece detecção de *architecture smells*, *design smells*, *implementation smells* e métricas orientadas a objetos. Também inclui detecção de *smells* específicos de *machine learning*. O nome é pronunciado como /di:pai/

Design Smell Categoria de *code smells* que indica problemas no projeto de classes e suas relações, como acoplamento excessivo ou violações de princípios de design orientado a objetos

F1-Score Média harmônica entre precisão e *recall*, fornecendo uma métrica única que equilibra ambas as medidas. Calculada como $2 \times \frac{Preciso \times Recall}{Preciso + Recall}$

Few-Shot Learning Técnica de engenharia de *prompts* que consiste em fornecer ao modelo alguns exemplos demonstrativos dentro do próprio *prompt*, permitindo que ele aprenda o padrão desejado a partir de poucos casos

Implementation Smells Categoria de *code smells* que engloba problemas localizados no nível de métodos e funções individuais, em contraste com *smells* de design ou arquitetura

LangChain *Framework* de código aberto que oferece abstrações e componentes modulares para construção de aplicações baseadas em LLMs, facilitando a integração com fontes de dados externas

LangGraph Extensão do *LangChain* que permite modelar aplicações multiagente como grafos de estados, onde agentes são representados como nós e as transições como arestas

Long Message Chain *Code smell* que ocorre quando um objeto solicita outro objeto, que solicita outro, e assim sucessivamente, violando a Lei de Demeter

Long Parameter List *Code smell* que descreve funções que recebem número excessivo de

parâmetros, tornando sua chamada complexa e propensa a erros

Magic Number *Code smell* que se refere ao uso de valores literais numéricos no código sem explicação ou atribuição a constantes nomeadas

Prompt Instrução ou conjunto de instruções fornecidas a um modelo de linguagem para guiar sua resposta. Pode incluir contexto, exemplos e especificações de formato

Pydantic Biblioteca Python para validação de dados usando anotações de tipo, frequentemente utilizada para definir *schemas* de entrada e saída em aplicações

Recall Métrica que mede a proporção de casos positivos corretamente identificados em relação ao total de casos positivos existentes. Também conhecida como sensibilidade ou taxa de verdadeiros positivos. Calculada como $\frac{VP}{VP+FN}$

Structured Output Capacidade de LLMs de gerar respostas em formatos estruturados pre-definidos (como JSON), seguindo esquemas específicos que permitem processamento programático

Token Unidade básica de processamento de texto em modelos de linguagem. Um *token* pode representar uma palavra, parte de uma palavra ou caractere, dependendo do tokenizador utilizado

Análise Estática Técnica de verificação de código-fonte realizada sem executar o programa, utilizando ferramentas automatizadas para identificar padrões problemáticos, vulnerabilidades e violações de estilo

Complexidade Ciclomática Métrica de software que quantifica a complexidade de um programa medindo o número de caminhos linearmente independentes através do código-fonte

CoT *Chain-of-Thought* (Cadeia de Pensamento)

Dívida Técnica Metáfora que representa o custo implícito de retrabalho futuro causado por escolhas de implementação subótimas feitas para ganhar velocidade no curto prazo

Engenharia de *Prompts* Processo de projetar e otimizar as instruções fornecidas aos LLMs para obter respostas mais precisas e relevantes, sem modificar os parâmetros internos do modelo

Falso Positivo Detecção incorreta onde o sistema identifica um problema que na realidade não existe. No contexto de *code smells*, ocorre quando o sistema reporta um *smell* em um código que não apresenta o problema

JSON *JavaScript Object Notation*

Lei de Demeter Princípio de design de software que estabelece que um objeto deve ter conheci-

mento limitado sobre outros objetos, comunicando-se apenas com seus vizinhos imediatos. Também conhecida como *Law of Demeter* ou Princípio do Menor Conhecimento

LLM *Large Language Models* (Modelos de Linguagem de Grande Escala)

Padrão *Supervisor* Arquitetura de sistemas multiagente onde um agente orquestrador coordena o fluxo de trabalho, delegando tarefas aos agentes especialistas e consolidando os resultados

Padrão de Referência Conjunto de dados de referência considerado como verdade absoluta para fins de avaliação e validação de sistemas automatizados. Também conhecido como *ground truth* na literatura

Precisão Métrica que mede a proporção de detecções corretas (verdadeiros positivos) em relação ao total de detecções realizadas pelo sistema. Calculada como $\frac{VP}{VP+FP}$

Refatoração Processo de reestruturação do código existente sem alterar seu comportamento externo, visando melhorar sua estrutura interna, legibilidade e manutenibilidade

Sistema Multiagente Arquitetura de software composta por múltiplos agentes autônomos que colaboram para resolver problemas complexos, onde cada agente pode ser especializado em uma tarefa específica