

FACULDADE SENAC BLUMENAU
Superior de Tecnologia em Análise e Desenvolvimento de Sistemas

Cairo Augusto de Andrade
Djonathan Mariano D'Ávila
Edson Eurides
Eduardo Roberto Schulle
Gabriel Michelmann
Jonathan Bauer

**PROTÓTIPO DE UM SISTEMA DE REVISÕES DE CÓDIGO PARA DETECÇÃO E
SUGESTÃO DE CORREÇÃO DE *CODE SMELLS* COM AUXÍLIO DE
INTELIGÊNCIA ARTIFICIAL**

Blumenau
2025

Cairo Augusto de Andrade

Djonathan Mariano D'Ávila

Edson Eurides

Eduardo Roberto Schulle

Gabriel Michelmann

Jonathan Bauer

**PROTÓTIPO DE UM SISTEMA DE REVISÕES DE CÓDIGO PARA DETECÇÃO E
SUGESTÃO DE CORREÇÃO DE *CODE SMELLS* COM AUXÍLIO DE
INTELIGÊNCIA ARTIFICIAL**

Trabalho apresentado ao Senac Blumenau como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador(es): Me. Cláudio Ratke
Esp. Fabiano Oss
Me. Pedro Floriani

Blumenau

2025

Cairo Augusto de Andrade
Djonathan Mariano D'Ávila
Edson Eurides
Eduardo Roberto Schulle
Gabriel Michelmann
Jonathan Bauer

**PROTÓTIPO DE UM SISTEMA DE REVISÕES DE CÓDIGO PARA DETECÇÃO E
SUGESTÃO DE CORREÇÃO DE *CODE SMELLS* COM AUXÍLIO DE
INTELIGÊNCIA ARTIFICIAL**

Trabalho apresentado ao Senac Blumenau como requisito parcial para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas.

Orientador: Me. Cláudio Ratke
Esp. Fabiano Oss
Me. Pedro Floriani

Avaliadores:

Me. Cláudio Ratke. – Senac Blumenau

Esp. Fabiano Oss. – Senac Blumenau

Me. Pedro Floriani. – Senac Blumenau

Blumenau, 05 de julho de 2025

Ficha de identificação da obra

A554p Andrade, Cairo Augusto de

Protótipo de um sistema de revisão de código para detecção e sugestão de correção de *code smells* com auxílio de Inteligência Artificial / Cairo Augusto de Andrade ; Djonathan Mariano D'Ávila ; Edson Eurides ; Eduardo Roberto Schulle ; Gabriel Michelmann ; Jonathan Bauer. – Blumenau (SC): Faculdade Senac Blumenau, 2025.

106 f.: il. color.

Orientador: Me. Cláudio Ratke.

Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) – Faculdade Senac Blumenau, 2025.

1. Code Review. 2. Code Smells. 3. Inteligência Artificial. I. Andrade, Cairo Augusto de. II. D'Ávila, Djonathan Mariano. III. Eurides, Edson. IV. Schulle, Eduardo Roberto. V. Michelmann, Gabriel. V. Bauer, Jonathan. VI. Título.

CDD 22 ed.: 006.3

Ficha catalográfica elaborada pela Bibliotecária Aline Ferreira – CRB 14/1721



Creative Commons - Atribuição - Não Comercial CC BY-NC

AGRADECIMENTOS

Gostaríamos de expressar nossa sincera gratidão aos professores Cláudio Ratke, Fabiano Oss e Pedro Floriani pelo acompanhamento e orientação durante a realização deste trabalho. Suas contribuições foram fundamentais para a consolidação do conhecimento técnico e metodológico ao longo da pesquisa.

Ao professor Cláudio Ratke, agradecemos especialmente pela escuta atenta, pelos direcionamentos assertivos e pela confiança em nossas decisões ao longo do desenvolvimento do projeto. Sua orientação foi indispensável para a estruturação e amadurecimento da proposta.

Ao professor Fabiano Oss, somos gratos pelas valiosas sugestões e pelo incentivo constante à qualidade técnica do sistema, o que contribuiu diretamente para o aprimoramento do sistema.

Ao professor Pedro Floriani, agradecemos pela disponibilidade, pela visão crítica e pelas contribuições pontuais que enriqueceram a fundamentação teórica e os aspectos técnicos deste trabalho.

Agradecemos também aos nossos familiares, que, com paciência e apoio incondicional, nos proporcionaram o suporte necessário para que pudéssemos nos dedicar integralmente a esta etapa da vida acadêmica.

Por fim, registramos nosso reconhecimento aos colegas de grupo, pelo esforço conjunto, comprometimento e colaboração ao longo de todo o processo. O sucesso deste projeto é resultado direto da construção coletiva e da dedicação de cada integrante.

RESUMO

O processo de revisão de código (*code review*) é fundamental para garantir a qualidade, a manutenibilidade e a evolução sustentável de projetos de software. Contudo, a alta demanda por revisões, aliada à complexidade crescente dos sistemas, pode levar a avaliações superficiais ou inconsistentes. Nesse sentido, desenvolveu-se a ferramenta BloodHound, que utiliza a inteligência artificial para analisar as últimas atualizações dos arquivos de código-fonte em repositórios (*pull requests*), identificando potenciais *code smells* e sugerindo melhorias automaticamente. O projeto consiste na criação de uma aplicação capaz de avaliar o código adicionado e alterado no *pull request*, fornecendo relatórios sobre *code smells* encontrados, como métodos longos, duplicações e classes com múltiplas responsabilidades. Por meio de técnicas de processamento de linguagem natural e engenharia de prompt, o sistema reconhece padrões de más práticas e propõe refatorações alinhadas a recomendações reconhecidas pela comunidade de engenharia de software. Além disso, a ferramenta padroniza o feedback oferecido aos desenvolvedores, reduzindo a subjetividade típica de revisões manuais. A colaboração da inteligência artificial nesse processo ocorre porque a tecnologia oferece recursos avançados de análise contextual e geração de sugestões, assegurando uma abordagem mais completa e ágil para a detecção de problemas. Em síntese, o BloodHound representa uma solução que busca aprimorar o processo de revisão de código, contribuindo para a elevação da qualidade do software e a otimização do tempo gasto em refatorações.

Palavras-chave: Code Review; Code Smells; Inteligência Artificial.

ABSTRACT

The code review process is essential to ensure the quality, maintainability, and sustainable evolution of software projects. However, the high demand for reviews, coupled with the increasing complexity of systems, can lead to superficial or inconsistent evaluations. In this regard, BloodHound was developed, using artificial intelligence to analyze the latest updates of source code files in repositories (pull requests), identifying potential code smells and suggesting improvements. The project consists of creating an application capable of evaluating the code added and modified in the pull request, providing reports on detected code smells such as long methods, code duplications, and classes with multiple responsibilities. By applying natural language processing and prompt engineering techniques, the system recognizes bad practice patterns and proposes refactoring aligned with widely accepted software engineering guidelines. Furthermore, the tool standardizes the feedback offered to developers, reducing the subjectivity typically found in manual reviews. Artificial Intelligence plays a key role in this process by offering advanced contextual analysis and suggestion generation, ensuring a more comprehensive and agile approach to problem detection. In summary, BloodHound represents a solution aimed at enhancing the code review process, contributing to improved software quality and optimization of the time spent on refactoring.

Keywords: Code Review; Code Smells; Artificial Intelligence.

LISTAS DE ILUSTRAÇÕES

Figura 1 - A distribuição da quantidade de pontos exploráveis nas superfícies de ataque.	17
Figura 2 - Impacto da remoção das ocorrências de code smells na qualidade.....	18
Figura 3 - Linha do tempo: Do desenvolvimento de software ao surgimento do code review e sua perpetuação até os dias atuais.	21
Figura 4 - Fluxo de Trabalho de Pull Request no GitHub	23
Figura 5 - Exemplo de Code Smell: Método Longo.....	31
Figura 6 - Exemplo de Code Smell: Inveja de Recursos	32
Figura 7 - Exemplo de <i>Code Smell</i> : Classe Deus.....	33
Figura 8 - Exemplo de <i>Code Smell</i> : Agrupamentos de Dados.....	35
Figura 9 - Exemplo de <i>Code Smell</i> : Código Duplicado	35
Figura 10 - Linha do tempo: Da IA ao ChatGPT	38
Figura 11 - Gráfico de Pesquisa: Principal ocupação	45
Figura 12 - Gráfico de Pesquisa: Nível de experiência	46
Figura 13 - Gráfico de Pesquisa: Frequência de Code Reviews	46
Figura 14 - Gráfico de Pesquisa: Plataforma de hospedagem mais utilizada	47
Figura 15 - Gráfico de Pesquisa: Funcionalidades essenciais em uma ferramenta de IA.....	48
Figura 16- Gráfico de Pesquisa: Média de importância do top 10 code smells.....	49
Figura 17 - Arquitetura do Sistema.....	53
Figura 18 - Modelo Entidade Relacionamento	56
Figura 19 - Fragmento de código referente à autenticação com OAuth 2.0	58
Figura 20 - Fragmento de código referente à comunicação com o Webhook	59
Figura 21 - Fluxo de Integração com GitHub	60
Figura 22 - Fragmento de código referente à comunicação com a OpenAI API	61
Figura 23 - Fluxo de Integração com a OpenAI API.....	62
Figura 24 - Fragmento Inicial do Prompt enviado à API da OpenAI	64
Figura 25 - Fragmento Final do Prompt enviado à API da OpenAI	65
Figura 26 - Sistema proposto: Página de login	66
Figura 27 - Sistema proposto: Página inicial - Dashboard.....	66

Figura 28 - Sistema proposto: Rules.....	67
Figura 29 - Sistema proposto: Repositórios	68
Figura 30 - Sistema proposto: Pull requests	68
Figura 31 - Provisão de recursos com Terraform	70
Figura 32 - Trecho da refatoração do <i>Code Smell God Class</i>	75
Figura 33 - Trecho da refatoração do Code Smell Long Method	76
Figura 34 - Gráfico de Pesquisa: IA no processo de <i>Code review</i>	81

LISTA DE ABELAS

Tabela 1 - Princípios do Código Limpo	24
Tabela 2 - <i>Code Smells</i> sugeridos por Martin Fowler	28
Tabela 3 - SPIKES	51
Tabela 4 - História de usuário	54
Tabela 5 - Resultados dos Testes de Detecção e Sugestão de Code Smells com a Ferramenta BloodHound	73

LISTA DE SIGLAS

GPT - *Generative Pre-Trained Transformer*

IA - Inteligência Artificial

ICAA - *Intelligent Code Analysis Agents*

IDEs - *Integrated Development Environment*

LLMs - *Large Language Models*

MER - Modelo Entidade-Relacionamento

NIST - *National Institute of Standards and Technology*

PLN - Processamento de Linguagem Natural

PMD - *Programming Mistake Detector*

RUP - *Rational Unified Process*

SUMÁRIO

1 INTRODUÇÃO	13
1.1 OBJETIVOS	15
1.1.1 Objetivo geral.....	15
1.1.2 Objetivos específicos.....	15
1.2 JUSTIFICATIVA.....	15
2 FUNDAMENTAÇÃO TEÓRICA	20
2.1 DESENVOLVIMENTO DE SOFTWARE	20
2.1.1 Qualidade de Software.....	21
2.1.1 Code Review	22
2.1.3 Boas Práticas de Programação.....	24
2.1.4 Code Smell.....	25
2.1.4.1 HISTÓRICO	27
2.1.4.2 EXEMPLOS DE <i>CODE SMELLS</i> COMUNS.....	28
2.1.4.4 <i>CODE SMELLS</i> RELEVANTES.....	31
2.1.4.5 <i>DETECÇÃO AUTOMATIZADA DE CODE SMELLS</i>	36
2.2 INTELIGÊNCIA ARTIFICIAL.....	37
2.2.1 Fundamentos da Inteligência Artificial.....	37
2.2.2 Modelos de Linguagem em Larga Escala (LLMs).....	39
2.2.3 ChatGPT	40
2.2.2 Inteligência Artificial e a análise de código	40
3 METODOLOGIA	42
3.1 CARACTERIZAÇÃO DA PESQUISA.....	42
3.2 TÉCNICAS DE COLETA DE DADOS	42
3.3 FORMA DE ANÁLISE DOS DADOS	43
4. RESULTADOS E DISCUSSÃO	44
4.1 CONTEXTUALIZAÇÃO DOS DADOS	44
4.2 APRESENTAÇÃO DOS RESULTADOS	44
4.3 ANÁLISE CRÍTICA DOS RESULTADOS	49

5 PROPOSIÇÃO DA UTILIZAÇÃO DE UM SISTEMA DE REVISÕES DE CÓDIGO PARA DETECÇÃO E SUGESTÃO DE CORREÇÃO DE <i>CODE SMELLS</i>	51
5.1 PROPOSIÇÃO	51
5.1.1 Spikes.....	51
5.1.2 Arquitetura	52
5.1.3 Histórias de Usuário.....	54
5.2 SOFTWARE DESENVOLVIDO	55
5.2.1 Modelo Entidade Relacionamento	56
5.2.2 Integração com GitHub	57
5.2.3 Integração com a API da OpenAI	60
5.2.4 <i>Prompt</i> utilizado	63
5.2.5 Operacionalização do Sistema.....	65
5.2.5 Infraestrutura do Sistema	69
5.3 TESTES E ANÁLISE DOS RESULTADOS	71
5.3.1 Escolha da Linguagem para os Testes	71
5.3.2 Estruturação dos Testes.....	72
5.3.2 Análise Quantitativa dos Testes	73
5.3.2 Análise Qualitativa das Sugestões	74
6 CONCLUSÃO	79
REFERÊNCIAS.....	83
APÊNDICE A - DICIONÁRIO DE DADOS DAS TABELAS DO MER	91
APÊNDICE B - TABELA COM CASOS DE TESTE COM CORREÇÃO E SUGESTÃO DE CORREÇÃO	94
APÊNDICE C - FORMULÁRIO <i>ONLINE</i> PARA PESQUISA COM PROFISSIONAIS DA ÁREA DE DESENVOLVIMENTO DE SISTEMAS.....	98

1 INTRODUÇÃO

O desenvolvimento de software, especialmente em ambientes modernos, costuma adotar abordagens ágeis, caracterizadas por ciclos iterativos e incrementais de entrega. Os times ágeis trabalham em ciclos curtos de tempo chamados *sprints*, nos quais pequenas funcionalidades são planejadas, implementadas, testadas e entregues continuamente ao longo do projeto (AJIT SINGH, 2019).

Esse processo se inicia com a identificação e priorização de necessidades do cliente, traduzidas em “histórias de usuário”. A cada *sprint*, uma seleção desses requisitos é transformada em funcionalidades por meio da colaboração entre desenvolvedores, testadores e outros membros da equipe. Durante a *sprint*, ocorrem diversas atividades essenciais, como codificação, testes, revisões e refinamentos, sempre com foco em entregar valor contínuo ao cliente (SCHWABER; SUTHERLAND, 2020).

A natureza incremental e colaborativa do modelo ágil exige mecanismos eficazes para garantir a qualidade do software à medida que ele evolui. Uma dessas práticas fundamentais no contexto ágil é o processo de revisão de código ou *code review*, que atua de forma contínua e integrada ao fluxo de desenvolvimento.

O *code review* consiste na análise do código-fonte por outros desenvolvedores da equipe. O objetivo é verificar a aderência às boas práticas de programação, a clareza da implementação e possíveis erros (BECK, 2001).

Esse processo contribui para aumentar a qualidade do software, pois permite detectar inconsistências, melhorar a legibilidade e garantir que as normas de codificação sejam seguidas (FOWLER, 1999).

Além dos benefícios técnicos, a revisão de código também facilita a troca de conhecimento entre os membros da equipe. De acordo com McIntosh et al. (2015), esse processo estimula a disseminação de boas práticas e o aprendizado de princípios de design, contribuindo para a formação de profissionais mais preparados.

Em ambientes de desenvolvimento ágil, membros menos experientes podem aprender diretamente com colegas mais experientes. Essa dinâmica fortalece a capacidade coletiva de resolver problemas e aprimora a qualidade do software na totalidade.

No entanto, conforme os projetos de software crescem e complexidade, a revisão manual de código se torna desafiadora. Com um grande volume de alterações

sendo feitas simultaneamente, muitos *pull requests* — solicitações para integrar novos trechos de código a um repositório — podem se acumular em curtos períodos, sobrecarregando os revisores. Além disso, fatores humanos, como a falta de tempo ou de atenção, podem fazer com que problemas importantes sejam ignorados ou subestimados (BACCHELLI; BIRD, 2013).

Entre os problemas nesse processo, estão os chamados *code smells*. O termo *code smell* foi introduzido por Kent Beck e popularizado por Martin Fowler como uma metáfora para indicar sinais de que o código pode estar mal projetado ou se tornar problemático com o tempo, mesmo que ainda funcione corretamente.

Esses "cheiros" não representam *bugs* diretos, mas sim indícios de que a estrutura do código pode estar violando princípios de design, como coesão, acoplamento ou legibilidade (FOWLER; BECK, 1999).

Exemplos comuns de *code smells* incluem métodos excessivamente longos, classes com múltiplas responsabilidades e trechos de código duplicados. Essas características podem dificultar a manutenção e a evolução do sistema.

Segundo Beck (2005), esses sinais frequentemente revelam problemas estruturais mais profundos, exigindo a refatoração — uma reescrita do código visando melhorar sua organização e facilitar a manutenção ao longo do tempo.

Para mitigar esses desafios, a Inteligência Artificial Generativa surge como uma solução alternativa. Nos últimos anos, a aplicação de LLMs (modelos de linguagem de grande escala) tem se destacado como uma abordagem promissora para automatizar partes do processo de geração e revisão de código (HUYNH, N.; LIN, B., 2025).

Nessa perspectiva, Rasheed et al. (2024) demonstraram que agentes baseados em LLMs podem detectar potenciais falhas e propor otimizações, contribuindo para a melhoria da qualidade do software e para a educação dos desenvolvedores.

É nesse cenário que se insere o BloodHound *Code Reviewer*: uma ferramenta projetada para atuar como um assistente inteligente no processo de revisão de código. Integrando-se a repositórios de software, o BloodHound analisa as solicitações para integrar novos trechos de código ao código-fonte buscando detectar *code smells* e, quando aplicável, fornece sugestões de refatoração fundamentadas em boas práticas de desenvolvimento.

Além de otimizar a tarefa dos revisores, essa abordagem promove uma cultura de qualidade contínua, onde cada alteração de código é avaliada de forma objetiva e padronizada. Isso resulta em benefícios como a redução de erros e retrabalho, além de contribuir para a formação de equipes mais alinhadas com as melhores práticas de desenvolvimento.

1.1 OBJETIVOS

A seguir serão apresentados o objetivo geral e os objetivos específicos que sustentarão o presente estudo.

1.1.1 Objetivo geral

Investigar o potencial da Inteligência Artificial Generativa para auxiliar na detecção e sugestão de correção de *code smells* durante o processo de *code review*, por meio do desenvolvimento de um protótipo denominado BloodHound Code Reviewer.

1.1.2 Objetivos específicos

Para alcançar o desenvolvimento do objetivo geral, foram definidos os seguintes objetivos específicos:

- a) Estudar o processo de revisão de código (*code review*).
- b) Pesquisar sobre os *code smells*, seus tipos e impactos.
- c) Investigar o uso de Inteligência Artificial Generativa para detecção de *code smells*.
- d) Desenvolver um protótipo (BloodHound) que realize a detecção de *code smells*.

1.2 JUSTIFICATIVA

Como destaca Sommerville (2015), o desenvolvimento de software é um processo contínuo de manutenção e evolução, pois o software precisa se ajustar a requisitos, tecnologias e contextos em constante mudança.

Nesse cenário, a revisão manual de código é importante, mas pode se tornar ineficiente em equipes grandes e projetos de alta complexidade, resultando em verificações superficiais dos *pull requests* (RIGBY; BIRD, 2013).

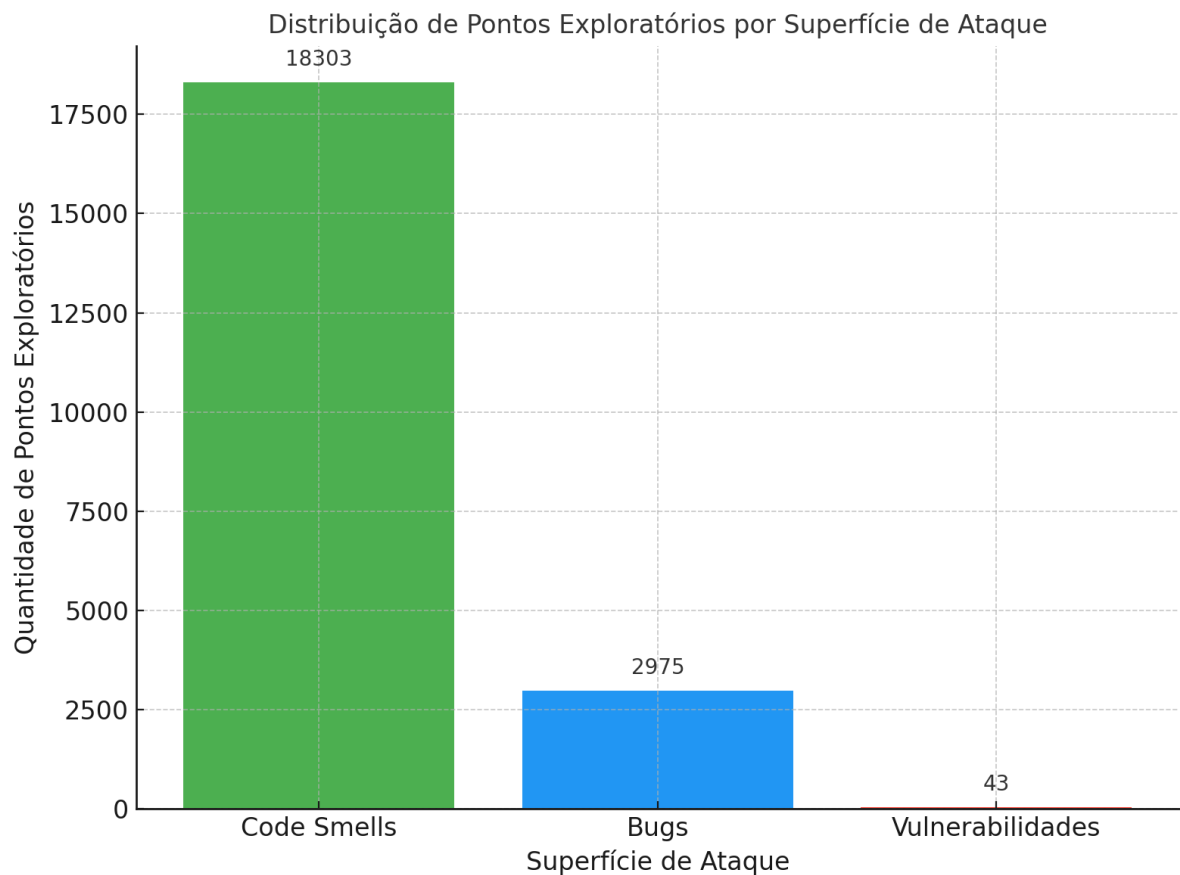
Essa revisão superficial, entretanto, pode permitir que problemas estruturais passem despercebidos, favorecendo o acúmulo de *code smells* no projeto. Tais falhas, quando não corrigidas, geram problemas de manutenção, aumentam a incidência de *bugs* e até elevam o risco de vulnerabilidades de segurança.

Sob essa perspectiva, um estudo recente sobre vulnerabilidades em projetos de código aberto (EDWARD et al., 2024) destacou a importância de identificar problemas estruturais no código, conhecidos como *code smells*, para a segurança do software.

Nesse estudo, foram analisadas 30 aplicações, e os pesquisadores classificaram três tipos principais de “superfícies de ataque” — conjunto de pontos de entrada ou componentes de um sistema que podem ser explorados por atacantes para comprometer sua segurança — que poderiam ser utilizados: *code smells*, *bugs* e vulnerabilidades conhecidas.

Como mostra a Figura 1 (A distribuição da quantidade de pontos exploráveis nas superfícies de ataque) a seguir, 85,8% dos pontos potencialmente exploráveis estavam associados a *code smells*, superando significativamente o percentual de *bugs* (14%) e vulnerabilidades propriamente ditas (0,2%).

Figura 1 - A distribuição da quantidade de pontos exploráveis nas superfícies de ataque.



Fonte: Adaptado de Edward et al. (2024)

Os números acima evidenciam que a presença de *code smells* amplia a quantidade de pontos exploráveis no software, abrindo brechas que podem facilitar a introdução de falhas de segurança (EDWARD et al., 2024).

Segundo Anwar et al. (2018), as brechas de segurança em um software podem gerar impactos financeiros significativos, afetando tanto as empresas desenvolvedoras quanto a economia em geral. Além dos custos diretos com a correção de falhas, como o tempo de desenvolvedores dedicado à criação e redistribuição de patches, há também despesas associadas à reimplantação de sistemas e à perda de produtividade.

Ademais, vulnerabilidades podem prejudicar a confiança dos usuários, comprometer a reputação da marca e, em última instância, resultar na perda de clientes. Esses fatores contribuem para uma desvantagem competitiva no mercado global, especialmente em comparação com empresas menos suscetíveis a falhas. Um

estudo do NIST (National Institute of Standards and Technology) estima que a economia dos Estados Unidos perde aproximadamente 60 bilhões de dólares anualmente devido a esses impactos (ANWAR et al., 2018).

Além de brechas de segurança, o acúmulo de *code smells* pode ocasionar a ocorrência de *bugs*. Sob essa perspectiva, uma revisão sistemática da literatura realizada por (CAIRO, A. S. et al., 2018), analisou 18 estudos e evidenciou que a presença de *code smells* está diretamente relacionada ao aumento da incidência de *bugs* no desenvolvimento de software. Dos estudos analisados, 16 apontaram essa correlação de forma significativa. Esses *bugs* acarretam diversos prejuízos para as organizações, incluindo custos elevados de manutenção, redução da produtividade das equipes de desenvolvimento e potenciais perdas de receita.

Outrossim, estudos recentes indicam que o custo de corrigir defeitos aumenta significativamente à medida que o software avança pelas fases do ciclo de vida. Por exemplo, a IBM Systems Sciences Institute (2024) relatou que corrigir um erro encontrado após o lançamento do produto pode ser até 100 vezes mais caro do que se fosse identificado durante a fase de design.

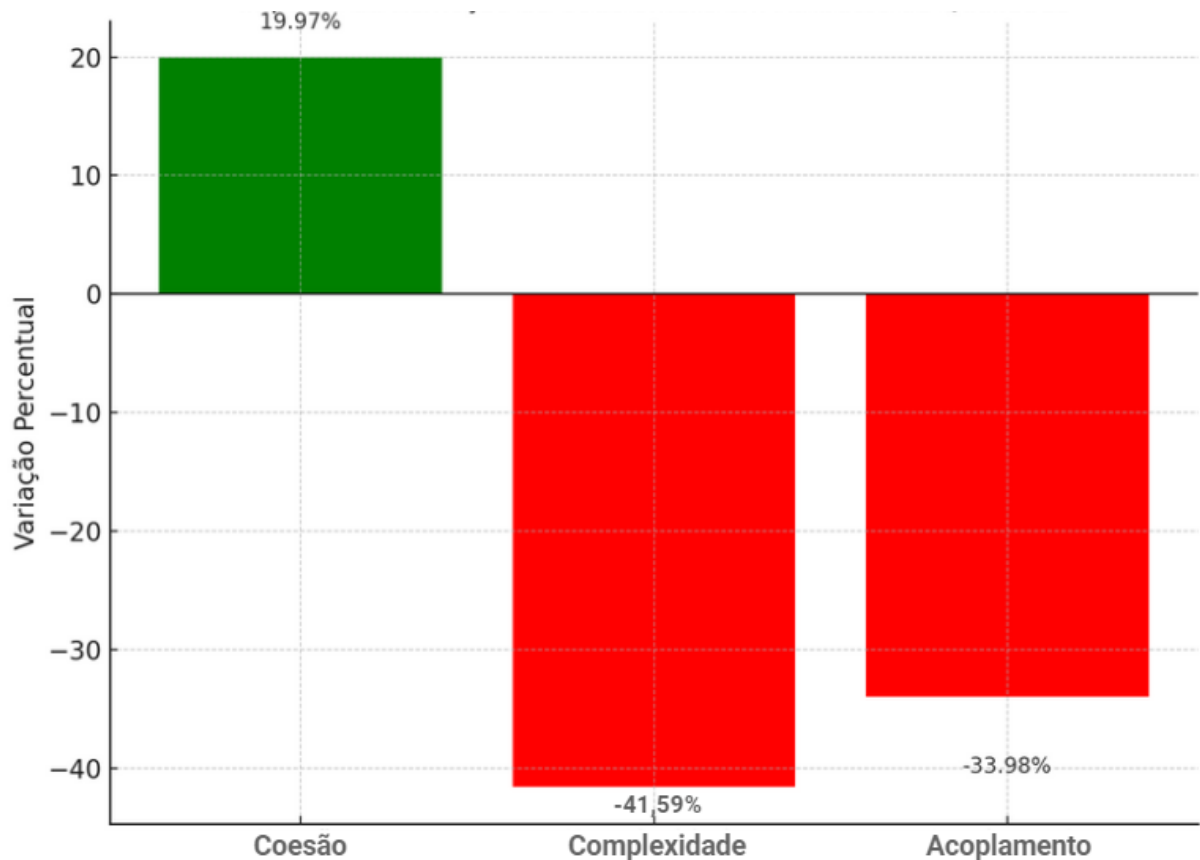
Além disso, um estudo feito por Black (2002) observou que o esforço necessário para identificar e corrigir defeitos aumenta à medida que o software progride pelas fases de desenvolvimento.

Além de possíveis vulnerabilidades e *bugs*, de acordo com Fowler (2019), projetos que acumulam *code smells* tornam-se progressivamente mais difíceis de manter, aumentando a complexidade do código e dificultando futuras evoluções.

Esses problemas podem comprometer a escalabilidade da aplicação, tornando a adaptação a novas tecnologias ou requisitos do mercado um desafio caro e demorado (BROWN et al., 1998).

Em vista disso, segundo um estudo realizado por Martins (2021), a remoção das ocorrências de *code smells* em sistemas industriais pode melhorar significativamente os atributos internos de qualidade do software. Em seu estudo, que analisou diversos projetos, a eliminação dessas anomalias mostrou ganhos notáveis em métricas como coesão, redução de acoplamento e diminuição da complexidade do código, como pode ser visto na Figura 2 (Impacto da remoção das ocorrências de *code smells* na qualidade).

Figura 2 - Impacto da remoção das ocorrências de code smells na qualidade.



Fonte: Adaptado de Martins (2021)

Os resultados apresentados na Figura 2 demonstram que a remoção de ocorrências específicas de *code smells* pode melhorar significativamente os atributos internos de qualidade do software. Em seus melhores casos, a eliminação dos *code smells* aumentou a coesão em 19,97%, reduziu a complexidade em 41,59% e baixou o acoplamento em 33,98% (MARTINS, 2021).

Nesse contexto, o protótipo *BloodHound Code Reviewer* busca oferecer suporte às equipes de desenvolvimento na detecção de *code smells*, auxiliando na manutenção e evolução contínua do código. Com a incorporação de inteligência artificial, a ferramenta propõe-se a apoiar o processo de *code review*, fornecendo informações que acompanham a qualidade do código ao longo do tempo e contribuindo para a adoção de processos mais ágeis e sustentáveis.

“2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo será apresentada a fundamentação teórica que embasa o desenvolvimento do presente trabalho. Serão abordados conceitos sobre qualidade de software, revisão de código, *code smells* e sua classificação, bem como técnicas de detecção. Também serão exploradas as capacidades de modelos de linguagem (LLMs) no contexto de engenharia de software, os fundamentos da análise automatizada de código em *pull requests*, além de tópicos relacionados à integração de serviços em nuvem e boas práticas em ferramentas de apoio ao desenvolvimento.

2.1 DESENVOLVIMENTO DE SOFTWARE

O desenvolvimento de software tem passado por uma evolução contínua desde as primeiras décadas da computação moderna, acompanhando os avanços tecnológicos e as crescentes demandas do mercado e da sociedade. Ao longo do tempo, diferentes abordagens foram propostas para organizar as etapas envolvidas na construção de sistemas, como levantamento de requisitos, design, codificação, testes e manutenção. Embora essas fases sejam comuns à maioria dos projetos, a forma como são organizadas variou conforme o contexto e os objetivos das equipes.

Inicialmente, prevaleciam modelos sequenciais e rígidos, como o modelo cascata (*waterfall*), proposto por Royce (1970). Nesse formato, cada etapa dependia da finalização completa da anterior, o que dificultava adaptações no meio do caminho. À medida que os projetos se tornaram mais complexos e sujeitos a mudanças constantes, surgiram abordagens mais flexíveis, como o Modelo Espiral e o RUP (*Rational Unified Process*), que introduziram ciclos de retroalimentação e permitiram revisões periódicas ao longo do desenvolvimento (NILSSON; WILSON, 2012).

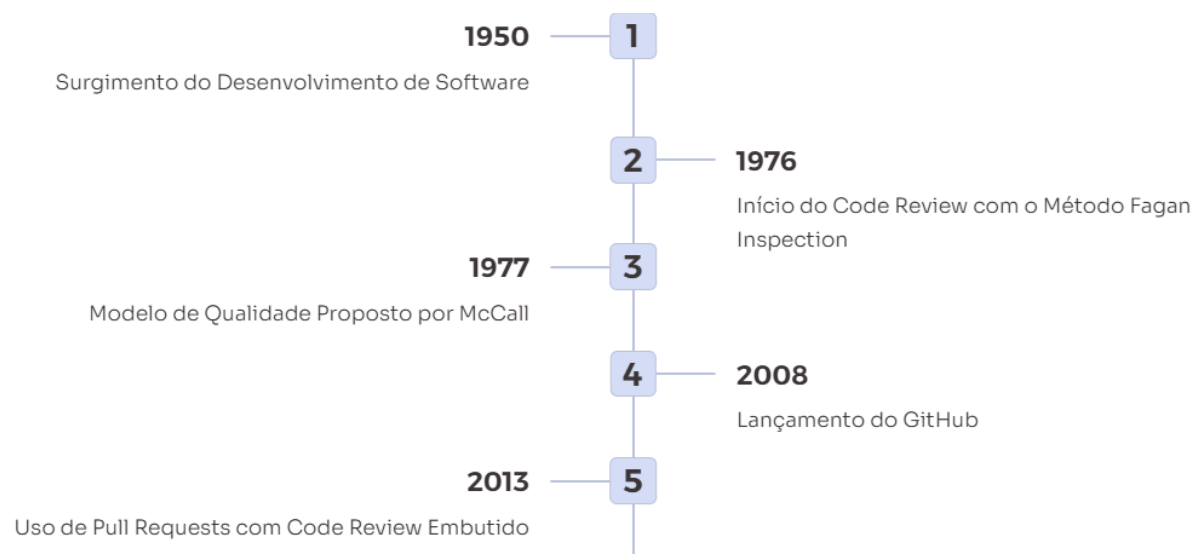
A partir dos anos 2000, métodos ágeis ganharam destaque por sua capacidade de lidar com incertezas e promover entregas contínuas. Com base em ciclos curtos de desenvolvimento — chamados de *sprints* — essas abordagens incentivam o planejamento incremental, testes frequentes e a colaboração entre equipe e cliente, resultando em produtos mais alinhados às necessidades reais (BECK et al., 2001).

Independentemente da metodologia adotada, a busca pela qualidade do software sempre esteve no centro das discussões. Garantir que um sistema seja

funcional, seguro, eficiente e sustentável é um desafio presente em todos os modelos de desenvolvimento. Práticas como testes automatizados, refatorações constantes e revisões de código se consolidaram como estratégias fundamentais para alcançar esse objetivo (DR. D. I. DE SILVA et al., 2023).

Para ilustrar a evolução histórica dos conceitos que envolvem o desenvolvimento de software — desde sua estruturação inicial até o surgimento de práticas modernas como o *code review* — a Figura 3 apresenta uma linha do tempo com os principais marcos. A imagem permite visualizar como, ao longo das décadas, o foco em qualidade e colaboração moldou as práticas utilizadas até os dias atuais, apoio à qualidade e sua consolidação como parte integrante dos processos modernos de engenharia de software.

Figura 3 - Linha do tempo: Do desenvolvimento de software ao surgimento do *code review* e sua perpetuação até os dias atuais.



Fonte: Elaborado pelos autores (2025).

2.1.1 Qualidade de Software

Conforme o desenvolvimento de sistemas cresceu em escala e complexidade, tornou-se essencial garantir não apenas que o software funcionasse, mas que também apresentasse atributos desejáveis como manutenibilidade, desempenho, segurança e confiabilidade. Foi nesse contexto que emergiu o conceito de qualidade

de software, inicialmente sistematizado por McCall et al. (1977), que propuseram um dos primeiros modelos para avaliar dimensões internas e externas do software.

Nesse sentido, a qualidade de software pode ser entendida como a capacidade de um produto atender aos requisitos funcionais e não funcionais, mantendo um desempenho satisfatório ao longo do tempo (CHAZETTE; BRUNOTTE; SPEITH, 2022). Esse conceito não se limita ao funcionamento imediato do sistema, mas envolve também sua adaptabilidade, facilidade de manutenção, reutilização e compreensão por parte da equipe técnica.

Modelos posteriores ampliaram esse entendimento ao incluir características como segurança, usabilidade, compatibilidade e eficiência no uso de recursos. Em ambientes ágeis, garantir a qualidade exige a adoção de práticas contínuas de inspeção, monitoramento e adaptação ao longo do ciclo de vida do software, visto que entregas frequentes demandam validações constantes do que está sendo construído (CHAZETTE; BRUNOTTE; SPEITH, 2022).

Dentro desse conjunto de práticas voltadas à preservação da qualidade, destaca-se o *code review* — ou revisão de código — como um mecanismo preventivo de controle técnico. Sua função é justamente apoiar a detecção precoce de falhas, a padronização de estilos de codificação e a disseminação de boas práticas entre os desenvolvedores. Por meio da análise colaborativa do código-fonte por outros membros da equipe, o *code review* contribui de forma significativa para a manutenção da qualidade estrutural do software e a redução de débitos técnicos (BACCHELLI; BIRD, 2013).

2.1.1 Code Review

Nesse sentido, o processo de revisão de código consiste na análise do código-fonte produzido por um desenvolvedor por outro membro da equipe, visando identificar erros, sugerir melhorias e assegurar a aderência aos padrões técnicos e de estilo do projeto. Essa prática desempenha um papel fundamental na garantia da qualidade do software, contribuindo para a detecção precoce de falhas e o fortalecimento da colaboração entre os membros da equipe (BACCHELLI; BIRD, 2013).

A origem do *code review* remonta à década de 1970, com os trabalhos de Michael Fagan na IBM. Fagan introduziu formalmente o conceito de software *inspections* como parte de um esforço para melhorar a qualidade dos produtos e

reduzir custos com defeitos detectados tardiamente (FAGAN, 1976). Com o tempo, essa prática evoluiu para formatos menos formais, integrando-se a metodologias ágeis e ferramentas modernas de versionamento como GitHub e GitLab.

No desenvolvimento de software contemporâneo, o *code review* é frequentemente realizado por meio de *pull requests*, uma funcionalidade presente em plataformas de hospedagem de código como GitHub, GitLab e Bitbucket. Esse processo inicia-se quando um desenvolvedor cria uma ramificação (*branch*) do repositório principal para desenvolver uma nova funcionalidade ou corrigir um erro (JURECZKO, 2020).

Após concluir as alterações, ele submete um *pull request* — uma solicitação para que esse novo código seja integrado à versão principal do sistema (geralmente chamada de *main* ou *master*).

Para facilitar a compreensão do funcionamento da revisão de código no desenvolvimento colaborativo moderno, a Figura 4 (Fluxo de Trabalho de *Pull Request* no GitHub) apresenta um diagrama simplificado do fluxo de trabalho de *pull request*, conforme ilustrado pela documentação oficial do GitHub (2023).

Esse fluxo demonstra como um desenvolvedor cria uma ramificação (*feature1*), realiza alterações no código e, posteriormente, solicita a revisão antes da integração dessas mudanças à base principal do projeto (*main*).

Figura 4 - Fluxo de Trabalho de *Pull Request* no GitHub



Fonte: Adaptado de GitHub Docs (2023).

Antes que o código seja de fato mesclado (realizado o *merge*) ao código-fonte principal, um ou mais revisores analisam as modificações propostas. Eles verificam aspectos como legibilidade, aderência às boas práticas, ausência de falhas ou vulnerabilidades e alinhamento com os padrões definidos pelo time.

Essa revisão pode incluir sugestões, questionamentos e até solicitações de mudanças adicionais. Somente após a aprovação da revisão é que o código é incorporado ao projeto principal. Esse processo garante uma segunda camada de

controle de qualidade, promovendo maior robustez e colaboração dentro da equipe (RIGBY; STOREY, 2011).

Estudos como o de Bacchelli e Bird (2013) demonstram que *code reviews* são eficazes não apenas para detectar defeitos, mas também para fomentar discussões técnicas e aprendizado coletivo dentro das equipes de desenvolvimento.

Nesse contexto, o *code review* emerge como um mecanismo de reforço dos padrões técnicos e de estilo adotados pelas equipes, servindo como uma barreira contra práticas inadequadas e sustentando a qualidade do produto final (O'CONNOR, 2024). A seguir, discute-se com mais detalhes a importância dessas boas práticas de programação e seu papel fundamental na construção de softwares sustentáveis e de alta qualidade.

2.1.3 Boas Práticas de Programação

A adoção de boas práticas de programação é essencial para o desenvolvimento de software de qualidade. Essas práticas visam tornar o código mais legível, manutenível e eficiente, facilitando a colaboração entre desenvolvedores e a evolução contínua dos sistemas. Robert C. Martin, em *The Clean Coder* (2011), destaca que a responsabilidade profissional do desenvolvedor inclui a escrita de código limpo. Ele argumenta que um código claro e bem estruturado facilita a comunicação entre os membros da equipe e reduz a incidência de erros.

Tabela 1 - Princípios do Código Limpo

Princípio	Descrição
Nomes Significativos	Utilizar nomes claros e descritivos para variáveis, funções e classes.
Funções Pequenas	Manter funções curtas que realizam uma única tarefa.

Evitar Duplicação	Reutilizar código para prevenir redundâncias.
Tratamento de Erros	Gerenciar exceções de forma clara e consistente.
Comentário Justificado	Comentar apenas quando necessário para esclarecer o código.

Fonte: Adaptado de Robert C. Martin, *The Clean Coder* (2011).

Tom Hombergs, por sua vez, aborda a aplicação prática dos princípios de arquitetura limpa. Ele demonstra como a separação de responsabilidades e a organização do código em camadas bem definidas contribuem para a manutenibilidade e escalabilidade do software. (HOMBERGS, 2019) reforça que a adoção de uma arquitetura limpa não é apenas uma questão técnica, mas também estratégica para o sucesso a longo prazo dos projetos.

A implementação dessas práticas impacta diretamente na produtividade das equipes de desenvolvimento. Um código bem estruturado reduz o tempo necessário para compreensão e modificação, permitindo que os desenvolvedores se concentrem em tarefas mais complexas e inovadoras. Consequentemente, isso resulta em entregas mais rápidas e com menor probabilidade de introdução de erros.

Em síntese, a prática do código limpo é essencial para o desenvolvimento de software robusto e sustentável. Ela promove a qualidade, a eficiência e a colaboração, sendo um investimento que traz benefícios significativos tanto para os desenvolvedores quanto para os usuários finais. A adoção dessas práticas, conforme discutido por autores como Martin (2011) e Hombergs (2019), é um passo fundamental para a excelência na engenharia de software.

2.1.4 Code Smell

Em oposição às boas práticas de desenvolvimento de software, surgem os chamados “code smells”, ou “maus cheiros de código”. O termo “*code smell*” foi

inicialmente introduzido por Kent Beck e posteriormente popularizado por Martin Fowler como uma metáfora poderosa para indicar sinais sutis de que um trecho de código pode estar mal projetado ou que, com o passar do tempo, poderá se tornar fonte de problemas mais sérios, mesmo que aparentemente ainda funcione de forma correta.

Esses “cheiros” não se tratam necessariamente de falhas funcionais ou bugs explícitos, mas sim de indícios que sugerem uma possível deterioração da estrutura interna do sistema. Eles costumam violar princípios fundamentais do design de software, como a coesão entre os componentes, o baixo acoplamento entre módulos e a legibilidade e clareza do código-fonte. Assim, os *code smells* funcionam como alertas que os desenvolvedores devem considerar seriamente durante o processo de manutenção e evolução contínua de sistemas de software (FOWLER; BECK, 1999).

Existem diversos tipos de *code smells* amplamente discutidos e documentados na literatura especializada em engenharia de software, cada um com características específicas e impactos distintos sobre a manutenibilidade do sistema. Entre os exemplos mais frequentemente encontrados em projetos reais estão os métodos excessivamente longos (*Long Method*), que concentram muitas responsabilidades em um único bloco de código e dificultam sua compreensão e reutilização.

Outro exemplo notório é a presença de classes que assumem múltiplas funções e responsabilidades dentro do sistema, conhecidas como *God Class*, o que viola diretamente o princípio da responsabilidade única. Além disso, o uso excessivo de comentários para explicar o funcionamento do código pode indicar que o mesmo não está suficientemente claro ou bem estruturado — situação que configura o *code smell* conhecido como *Excessive Comments*. Em um código bem escrito, idealmente, os próprios nomes de métodos e variáveis já deveriam ser suficientemente descritivos. A duplicação de lógica em diversos trechos da aplicação (*Duplicated Code*) também é um problema recorrente, pois aumenta a complexidade e o esforço necessário para aplicar mudanças consistentes ao sistema (FOWLER; BECK, 1999).

A presença desses elementos indesejáveis afeta diretamente a qualidade interna do software, mesmo quando não causam falhas imediatas em sua execução. Estudos acadêmicos e análises empíricas apontam que códigos com diversos *code smells* tendem a ser significativamente mais difíceis de entender, testar e modificar. Essa dificuldade impacta diretamente o tempo e o custo das atividades de

manutenção, além de aumentar o risco de introdução de novos erros ao se realizar qualquer tipo de alteração (MIN ZHANG et al., 2011).

Em ambientes corporativos, onde o tempo de resposta às mudanças nos requisitos é crucial, essa fragilidade estrutural pode comprometer seriamente a competitividade e a sustentabilidade do sistema. Sistemas que acumulam más práticas de codificação e não passam por refatorações periódicas enfrentam barreiras consideráveis para se adaptar a novas tecnologias, padrões de mercado e exigências dos usuários, o que pode comprometer sua evolução e longevidade no médio e longo prazo (MIN ZHANG et al., 2011).

2.1.4.1 Histórico

A ideia de identificar sinais de má qualidade no código antecede a formalização do conceito de *code smells*. Em 1968, Edsger Dijkstra já criticava o uso do comando goto, alertando para sua influência negativa na estrutura e legibilidade dos programas (KOKOL et al., 2021). Essa preocupação inicial reflete a intuição de que certos padrões indicam problemas potenciais, mesmo sem necessariamente provocarem falhas diretas no funcionamento do sistema.

O termo *code smell* foi introduzido por Kent Beck e popularizado por Martin Fowler na obra *Refactoring: Improving the Design of Existing Code* (1999). Na definição de Fowler e Beck, *code smells* são indícios superficiais de problemas mais profundos no design do software. Embora não configurem erros funcionais, esses "cheiros" alertam para práticas que podem comprometer atributos como manutenibilidade, legibilidade e extensibilidade do código.

Diversos estudos contemporâneos reforçam essa visão, tratando os *code smells* como anomalias estruturais que, embora não causem bugs diretos, impactam negativamente a qualidade do software ao longo do tempo. Essas anomalias podem se manifestar de várias formas, como dependências inadequadas entre módulos, alocação incorreta de métodos ou duplicação de código, afetando tanto o desempenho quanto a evolução dos sistemas (MACIA et al., 2012, SACA et al., 2017).

Além disso, evidências mostram que a presença de *code smells* está fortemente associada à degradação arquitetural e violações de modularidade, prejudicando a manutenção de aplicações críticas (LACERDA et al., 2020). A detecção desses sinais nem sempre é trivial ao exigir análise criteriosa e,

frequentemente, o apoio de ferramentas automatizadas para identificar padrões sutis no código.

Ao longo dos anos, a catalogação de *code smells* se expandiu. Inicialmente, Fowler (1999) propôs um catálogo com 22 tipos, mas novas categorias foram introduzidas posteriormente para diversas linguagens além de Java, como Matlab e Python (PEREIRA et al., 2020).

Dentre as abordagens mais recentes, destaca-se a taxonomia proposta por Tandon, Kumar e Singh (2024), que agrupa os *code smells* nas seguintes categorias:

- *Bloaters* (Inchaços): Componentes que cresceram excessivamente em tamanho ou complexidade.
- *Object-Orientation Abusers* (Abusadores de orientação a objetos): Violações dos princípios fundamentais da programação orientada a objetos.
- *Change Preventers* (Preventores de mudanças): Estruturas que dificultam a modificação segura do software.
- *Dispensables* (Dispensáveis): Elementos desnecessários que comprometem a clareza do código.

Couplers (Acopladores): Interdependências excessivas entre classes ou módulos.

Compreender e classificar os *code smells* é essencial para aplicar práticas de refatoração eficazes, promovendo a sustentabilidade e a evolução contínua de sistemas de software.

2.1.4.2 Exemplos de *code smells* comuns

Após a compreensão dos conceitos e impactos dos *code smells*, é pertinente apresentar os tipos mais recorrentes e suas implicações na qualidade do software. A seguir, uma seleção abrangente desses *smells*, suas descrições e categorias associadas, baseada no catálogo de Martin Fowler em *Refactoring: Improving the Design of Existing Code* e em estudos recentes. (FOWLER, 1999):

Tabela 2 - *Code Smells* sugeridos por Martin Fowler

	Nome do <i>Code Smell</i>	Descrição
--	----------------------------------	------------------

1	<i>Divergent Change</i> (Mudança Divergente)	Uma única classe precisa ser modificada por diferentes tipos de mudanças, indicando que ela tem múltiplas responsabilidades.
2	<i>Shotgun Surgery</i> (Cirurgia de Escopeta)	Uma única alteração exige modificações em várias classes diferentes, mostrando que a lógica está espalhada.
3	<i>Long Parameter List</i> (Lista Longa de Parâmetros)	Métodos ou construtores com muitos parâmetros, tornando o código difícil de ler e propenso a erros.
4	<i>Feature Envy</i> (Inveja de Função)	Um método que acessa dados de outra classe com frequência, sugerindo que ele pode estar na classe errada.
5	<i>Data Clumps</i> (Aglomerados de Dados)	Grupos de dados que aparecem juntos em vários lugares, sugerindo que deveriam ser encapsulados em uma classe própria.
6	<i>Large Class/God Class</i> (Classe Grande/Classe Deus)	Classes que acumulam muitas responsabilidades, atributos e métodos. O excesso de funcionalidades dificulta a compreensão e manutenção.
7	<i>Long Method</i> (Método Longo)	Métodos muito longos tornam o código difícil de entender e manter. O ideal é dividir em métodos menores e mais específicos..
8	<i>Message Chains</i> (Cadeias de Mensagens)	Sequências longas de chamadas de métodos em cascata, dificultando a compreensão e a manutenção.
9	<i>Middle Man</i> (Intermediário)	Classes que apenas repassam chamadas para outros objetos, sem agregar valor, podendo ser eliminadas.
10	<i>Speculative Generality</i> (Generalidade Especulativa)	Código genérico ou abstrato demais sem necessidade real, criado por antecipação de futuras demandas.

11	<i>Refused Bequest</i> (Herança Recusada)	Código criado para possíveis necessidades futuras que nunca se concretizam, tornando o sistema mais complexo sem necessidade.
12	<i>Parallel Inheritance Hierarchies</i> (Hierarquias de Herança Paralelas)	Quando para cada classe de uma hierarquia é necessário criar uma correspondente em outra hierarquia, levando a duplicação de estruturas.
13	<i>Alternative Classes with Different Interfaces</i> (Classes Alternativas com Interfaces Diferentes)	Classes que fazem coisas semelhantes, mas têm interfaces diferentes, dificultando o uso intercambiável.
14	<i>Inappropriate Intimacy</i> (Intimidade Inapropriada)	Classes que acessam detalhes internos de outras classes, violando o encapsulamento.
15	<i>Comments</i> (Comentários Excessivos)	Uso excessivo de comentários para explicar código confuso, em vez de melhorar a clareza do próprio código.
16	<i>Lazy Class</i> (Classe Preguiçosa)	Classes que não justificam sua existência porque têm pouca funcionalidade; podem ser removidas ou fundidas.
17	<i>Duplicated Code</i> (Código Duplicado)	Trechos de código idênticos ou muito semelhantes aparecem em mais de um lugar. Isso dificulta a manutenção, pois uma alteração precisa ser replicada em vários pontos.
18	<i>Temporary Field</i> (Campo Temporário)	Campos em uma classe que só são usados em situações específicas, indicando que poderiam ser movidos para outra estrutura.
19	<i>Data Class</i> (Classe de Dados)	Classes que apenas armazenam dados, sem comportamento associado, geralmente apenas com getters e setters.
20	<i>Switch Statements</i> (Estruturas de Seleção)	Uso frequente de estruturas switch/case ou if/ else aninhados, que podem indicar a necessidade de polimorfismo.

21	<i>Primitive Obsession</i> (Obsessão por Primitivos)	Uso excessivo de tipos primitivos em vez de criar tipos ou classes mais expressivas para o domínio.
22	<i>Incomplete Library Class</i> (Classe de Biblioteca Incompleta)	Classes de bibliotecas que não oferecem toda a funcionalidade necessária, levando a implementações alternativas ou subclasses.

Fonte: Adaptado de FOWLER; BECK (1999).

A presença de *code smells* não indica erros imediatos, mas aponta fragilidades no design do software que podem dificultar sua evolução. Como destaca Fowler (1999), identificar esses sinais é essencial para evitar problemas futuros e manter a qualidade do código. Refatorar para eliminar *code smells* torna o sistema mais claro, coeso e fácil de manter, contribuindo para sua sustentabilidade a longo prazo.

2.1.4.4 Code Smells relevantes

Estudos recentes identificaram os seguintes *code smells* como os mais frequentemente analisados na literatura:

O *code smell* conhecido como *Long Method* refere-se a métodos que possuem um número excessivo de linhas de código, tornando-se longos e complexos para compreensão e manutenção. Métodos extensos dificultam o entendimento do fluxo lógico, aumentam a probabilidade de erros e tornam a refatoração mais trabalhosa, já que pequenas alterações podem afetar diversas partes do método. Segundo (FOWLER, 1999), métodos mais curtos promovem melhor legibilidade e facilitam testes e reuso.

Estudos mostram que a identificação manual de *Long Methods* é custosa, levando ao desenvolvimento de ferramentas automatizadas baseadas em métricas como complexidade ciclomática e quantidade de instruções. Além disso, métodos longos frequentemente acumulam múltiplas responsabilidades, contrariando princípios de design orientado a objetos (VASCONCELOS, 2023). A presença desse *smell* está associada à degradação da qualidade do software e à maior dificuldade de evolução do código. Por isso, recomenda-se dividir métodos longos em funções menores e mais coesas, facilitando a manutenção e a evolução do sistema.

Figura 5 - Exemplo de *Code Smell*: Método Longo

```
java

public class Pedido {
    public void processarPedido() {
        System.out.println("Validando pedido...");
        // validação do pedido
        System.out.println("Calculando frete...");
        // cálculo do frete
        System.out.println("Processando pagamento...");
        // processamento do pagamento
        System.out.println("Enviando confirmação...");
        // envio de confirmação
    }
}
```

Fonte: Elaborado pelos autores (2025).

O método (`processarPedido`) realiza múltiplas tarefas: validação, cálculo de frete, processamento de pagamento e envio de confirmação. Cada uma dessas tarefas poderia ser encapsulada em métodos separados, promovendo o princípio da responsabilidade única.

O *Feature Envy* ocorre quando um método de uma classe utiliza mais dados ou métodos de outra classe do que da própria classe em que está definido, indicando uma quebra de encapsulamento. Segundo Fowler (1999), esse *smell* surge quando um método "deseja" funcionalidades de outra classe, centralizando operações que deveriam pertencer ao objeto dono dos dados.

A refatoração recomendada é o *Move Method* (método de movimento), realocando o método para a classe cujos dados ele mais utiliza, melhorando a coesão e reduzindo o acoplamento. Pesquisas recentes exploram técnicas de *deep learning* para aprimorar a detecção, mas destacam a necessidade de validação manual para evitar falsos positivos (AL-FRAIHAT, 2024). A presença de *Feature Envy* está associada às dificuldades de manutenção e degradação progressiva da qualidade do código

Figura 6 - Exemplo de *Code Smell*: Inveja de Recursos

```
java

public class RelatorioFuncionario {
    public void imprimirDetalhes(Funcionario funcionario) {
        System.out.println("Nome: " + funcionario.getNome());
        System.out.println("Cargo: " + funcionario.getCargo());
        System.out.println("Salário: " + funcionario.getSalario());
    }
}
```

Fonte: Elaborado pelos autores (2025).

O método (`imprimirDetalhes`) acessa diretamente os dados do objeto (`Funcionario`), sugerindo que essa responsabilidade poderia pertencer à própria classe (`Funcionario`).

O *God Class* ou *Large Class* refere-se a classes que concentram um número excessivo de responsabilidades, funcionalidades e dados, tornando-se centrais e dominantes na arquitetura do sistema. Essas classes apresentam alta complexidade e baixa coesão, dificultando a compreensão, manutenção e evolução do software, além de aumentar o esforço de desenvolvimento e os custos associados (MARTINS,2021).

God Classes violam princípios fundamentais de design, como o princípio da responsabilidade única, e tendem a ser difíceis de modificar sem introduzir erros em outras partes do sistema. Estudos mostram que a presença de *God Classes* está fortemente associada a problemas de qualidade interna, como alto acoplamento e complexidade ciclomática elevada, prejudicando a manutenibilidade do código (TURKISTANI,2019).

A detecção automática desse *smell* pode ser feita por meio de métricas de código e técnicas de aprendizado de máquina, que analisam características como número de métodos, atributos e interações com outras classes. Estratégias de refatoração, como a aplicação de padrões de projeto e a divisão em subclasses, são recomendadas para mitigar esse problema e melhorar a qualidade do software.

Figura 7 - Exemplo de *Code Smell*: Classe Deus

```
java
public class Sistema {
    private List<Usuario> usuarios;
    private List<Produto> produtos;

    public void adicionarUsuario(Usuario usuario) {
        usuarios.add(usuario);
    }

    public void removerUsuario(Usuario usuario) {
        usuarios.remove(usuario);
    }

    public void adicionarProduto(Produto produto) {
        produtos.add(produto);
    }

    public void removerProduto(Produto produto) {
        produtos.remove(produto);
    }

    public void gerarRelatorio() {
        // Lógica para gerar relatório
    }

    public void enviarEmail() {
        // Lógica para enviar email
    }

    // outros métodos diversos
}
```

Fonte: Elaborado pelos autores (2025).

A classe (Sistema) está sobrecarregada, gerenciando usuários, produtos, geração de relatórios e envio de e-mails. Essa concentração de responsabilidades torna a classe difícil de entender e manter.

O *Data Clumps* refere-se à ocorrência recorrente de grupos de variáveis que aparecem juntos em diferentes partes do código, seja como parâmetros de métodos ou como atributos em várias classes, indicando uma possível abstração ausente. Esses agrupamentos dificultam a manutenção, pois qualquer alteração exige modificações em múltiplos pontos, aumentando o risco de inconsistências e erros. Estudos apontam que *data clumps* são um dos *smells* mais comuns em projetos web e tendem a formar clusters que se espalham pelo sistema ao longo do ciclo de vida do software, tornando sua detecção e refatoração desafiadoras (BAUMGARTNER, 2024).

Ferramentas automatizadas, como plugins para IDEs e analisadores baseados em árvores sintáticas abstratas, têm sido desenvolvidas para identificar e sugerir a extração desses agrupamentos em novas classes ou objetos, promovendo maior coesão e encapsulamento. Pesquisas recentes também destacam a importância de

datasets específicos para avaliar o impacto dos *data clumps* na qualidade do software e apoiar estratégias de refatoração (FOWLER, 1999). A eliminação desse *smell* contribui para a redução do acoplamento e para a melhoria da clareza e da manutenibilidade do código.

Figura 8 - Exemplo de *Code Smell*: Agrupamentos de Dados

```
public class Pedido {  
    public void processarPedido(String nomeCliente, String enderecoCliente, String telefoneClient  
        // lógica para processar o pedido  
    }  
}
```

Fonte: Elaborado pelos autores (2025).

Os parâmetros (*nomeCliente*), (*enderecoCliente*) e (*telefoneCliente*) estão sempre juntos, sugerindo que representam um conceito único: o cliente.

O *Duplicate Code* refere-se à presença de trechos de código idênticos ou muito semelhantes espalhados por diferentes partes de um sistema, frequentemente resultado de cópia e colagem durante o desenvolvimento. Esse fenômeno é considerado prejudicial porque aumenta o esforço de manutenção: alterações ou correções precisam ser replicadas manualmente em todos os locais duplicados, o que eleva o risco de inconsistências e falhas. Além disso, a duplicação contribui para o crescimento desnecessário da base de código, dificultando sua compreensão e gerenciamento ao longo do tempo (ALLAMANIS, 2018).

Estudos indicam que, embora o código duplicado possa, em alguns casos, facilitar a leitura por reduzir a necessidade de navegação entre métodos, ele compromete a qualidade a longo prazo, especialmente em projetos de grande porte. Ferramentas automatizadas, como detectores de clones, são recomendadas para identificar e apoiar a refatoração desses trechos, promovendo maior reutilização e modularidade. A eliminação do código duplicado é fundamental para reduzir o débito técnico e melhorar a manutenibilidade do software.

Figura 9 - Exemplo de *Code Smell*: Código Duplicado

```

public class EmailService {

    public void enviarEmailBoasVindas(Usuario usuario) {
        Email email = new Email();
        email.setDestinatario(usuario.getEmail());
        email.setAssunto("Bem-vindo!");
        email.setCorpo("Olá " + usuario.getNome() + ", seja bem-vindo!");
        email.enviar();
    }

    public void enviarEmailRecuperacaoSenha(Usuario usuario) {
        Email email = new Email();
        email.setDestinatario(usuario.getEmail());
        email.setAssunto("Recuperação de Senha");
        email.setCorpo("Olá " + usuario.getNome() + ", clique no link para redefinir sua senha.")
        email.enviar();
    }
}

```

Fonte: Elaborado pelos autores (2025).

Os métodos (`enviarEmailBoasVindas`) e (`enviarEmailRecuperacaoSenha`) possuem código duplicado na criação e envio do e-mail. A única diferença está no assunto e corpo do e-mail

A identificação e correção desses *code smells* são essenciais para melhorar a qualidade do software, facilitando sua manutenção e evolução. Ferramentas como JDeodorant, PMD e iPlasma são frequentemente utilizadas para detectar essas anomalias.

2.1.4.5 Detecção Automatizada de Code Smells

A identificação e correção de *code smells* são etapas essenciais para garantir a qualidade de sistemas de software. Ferramentas tradicionais como o SonarQube, amplamente utilizadas na indústria, são baseadas em análise estática de código — ou seja, analisam o código-fonte sem executá-lo, procurando por padrões que indicam problemas estruturais.

Essas ferramentas são eficazes na identificação de uma variedade de *code smells*, como classes muito grandes, métodos complexos ou repetições de código (SONARSOURCE, 2018). Elas ajudam os desenvolvedores a manter padrões técnicos, reduzindo o retrabalho futuro.

Apesar de sua relevância, o uso do SonarQube é predominantemente estático, o que impõe algumas limitações. Como opera com regras fixas e genéricas, a ferramenta nem sempre consegue interpretar o contexto específico do sistema ou compreender a intenção do desenvolvedor. Por exemplo, duas funções com lógicas semelhantes, mas implementações distintas, podem passar despercebidas pela análise mesmo quando poderiam ser unificadas para evitar duplicação. Além disso, o SonarQube raramente sugere soluções detalhadas — ele aponta o problema, mas não orienta sobre como resolvê-lo, exigindo do desenvolvedor conhecimento prévio ou investigação adicional para aplicar uma refatoração adequada (ZHANG et al., 2024).

2.2 INTELIGÊNCIA ARTIFICIAL

Neste capítulo será apresentada a fundamentação teórica que embasa o desenvolvimento do presente trabalho. Serão abordados conceitos sobre qualidade de software, com foco em *code smells* e sua classificação, bem como técnicas de detecção.

Também serão exploradas as capacidades de modelos de linguagem (LLMs) no contexto de engenharia de software, os fundamentos da análise automatizada de código em *pull requests*, além de tópicos relacionados à integração de serviços em nuvem e boas práticas em ferramentas de apoio ao desenvolvimento.

2.2.1 Fundamentos da Inteligência Artificial

A Inteligência Artificial (IA) é um campo da ciência da computação que visa desenvolver sistemas capazes de realizar tarefas que normalmente requerem inteligência humana, como reconhecimento de padrões, tomada de decisões e aprendizado a partir de dados. Desde sua origem na década de 1950, a IA tem evoluído significativamente, incorporando diversas técnicas e abordagens para simular aspectos do comportamento humano (MIAN, 2022).

Entre as principais áreas da IA, destacam-se o Aprendizado de Máquina (Machine Learning), que permite que sistemas aprendem com dados; o Processamento de Linguagem Natural (PLN), que capacita as máquinas a

compreenderem e gerarem linguagem humana; e as Redes Neurais Artificiais, inspiradas na estrutura do cérebro humano, utilizadas para reconhecer padrões complexos em dados. (SARKER, 2021).

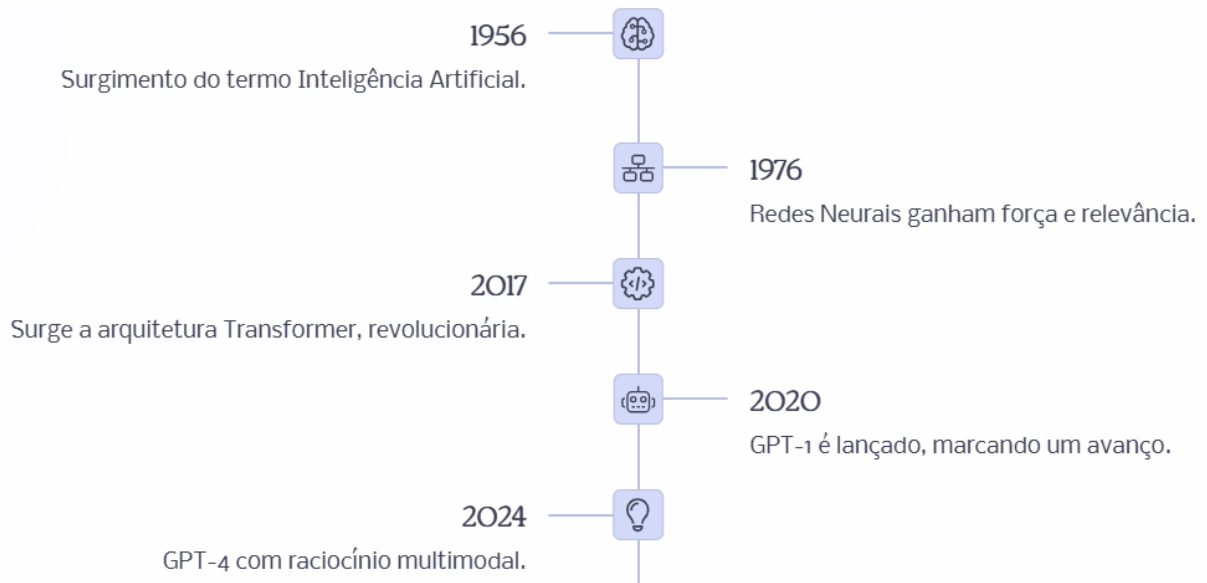
A IA pode ser categorizada em duas abordagens principais: a IA simbólica e a IA conexionista (baseada em aprendizado). A IA simbólica utiliza regras e lógica explícitas para representar o conhecimento, sendo eficaz em domínios bem estruturados. Por outro lado, a IA conexionista aprende a partir de grandes volumes de dados, identificando padrões e realizando inferências sem depender de regras pré-definidas. Recentemente, tem-se observado uma convergência entre essas abordagens, buscando combinar suas respectivas vantagens. (GOEL, 2022; XIONG et al., 2024).

A IA está cada vez mais presente em soluções práticas do dia a dia, desde assistentes virtuais e sistemas de recomendação até o diagnóstico médico e a automação de tarefas na engenharia de software. Esses avanços têm sido impulsionados, principalmente, pelo crescimento de abordagens conexionistas e pela utilização de grandes volumes de dados para treinamento.

Para complementar a compreensão sobre a evolução da Inteligência Artificial ao longo do tempo, a Figura 10 (Linha do tempo: Da IA ao ChatGPT) ilustra os principais marcos históricos desse campo. A linha do tempo inicia-se na década de 1950, com o surgimento do termo "Inteligência Artificial" e os primeiros estudos sobre máquinas inteligentes. Em seguida, destaca-se a consolidação do aprendizado de máquina e das redes neurais, que permitiram avanços significativos na capacidade das máquinas de reconhecer padrões e aprender com dados.

Na década de 2020, são evidenciados os avanços no Processamento de Linguagem Natural (PLN), impulsionados pelo surgimento dos Modelos de Linguagem em Larga Escala (LLMs), como o ChatGPT. Esses modelos marcaram uma nova era na forma como a IA interage com humanos, permitindo interpretações mais profundas e respostas contextuais mais precisas, com potencial de aplicação em diversas áreas, incluindo o desenvolvimento e a revisão de código-fonte.

Figura 10 - Linha do tempo: Da IA ao ChatGPT



Fonte: Elaborado pelos autores (2025).

Dentre essas abordagens, destacam-se os LLMs (Modelos de Linguagem em Larga Escala), que revolucionaram o processamento de linguagem natural ao permitir que máquinas compreendam e gerem texto com fluência, coerência e contextualização impressionantes.

2.2.2 Modelos de Linguagem em Larga Escala (LLMs)

Entre os avanços mais expressivos na área de IA nas últimas décadas estão os Modelos de Linguagem em Larga Escala, conhecidos como LLMs (Modelos de Linguagem em Larga Escala). Esses modelos são algoritmos treinados com enormes volumes de texto para aprender padrões linguísticos, realizar inferências e gerar conteúdo textual coerente. Utilizando arquiteturas como o Transformer (VASWANI et al., 2017), os LLMs conseguem capturar relações complexas entre palavras, frases e contextos, oferecendo resultados impressionantes em tarefas como tradução, sumarização, geração de código e até compreensão de intenções.

O diferencial dos LLMs está na sua capacidade de generalização. Eles são pré-treinados em vastos conjuntos de dados e posteriormente ajustados (*fine-tuned*) para tarefas específicas. Esse processo permite que um único modelo seja reaproveitado em múltiplos domínios com ajustes mínimos, o que potencializa sua aplicabilidade em diversos cenários (JEREMY; SEBASTIAN, 2018).

2.2.3 ChatGPT

Dentre os LLMs mais conhecidos, destaca-se o ChatGPT, desenvolvido pela OpenAI. Sua primeira versão pública foi lançada em 2018 com o GPT-1, evoluindo rapidamente até chegar ao atual GPT-4o, que incorpora capacidades multimodais e maior capacidade de raciocínio contextual. Com bilhões de parâmetros, o ChatGPT é capaz de gerar textos coerentes, responder perguntas, traduzir, resumir conteúdos e até escrever código, consolidando-se como uma ferramenta versátil para apoio em diferentes tarefas cognitivas (OPENAI, 2024).

A escolha do ChatGPT para este projeto se deve à sua arquitetura consolidada, à robustez de seu treinamento e à sua capacidade de lidar com linguagem técnica e natural de forma integrada. Sua evolução contínua, tanto em capacidade quanto em aplicabilidade, indica um potencial crescente para aplicação na engenharia de software, incluindo revisão de código e detecção de padrões problemáticos em sistemas computacionais.

2.2.2 Inteligência Artificial e a análise de código

Com o avanço da tecnologia, a Inteligência Artificial (IA) passou a desempenhar um papel relevante na engenharia de software, principalmente na automação de tarefas que antes eram feitas exclusivamente por humanos.

Sob essa perspectiva, uma pesquisa de Fan et al. (2023) explora como a Inteligência Artificial tem sido incorporada às ferramentas de análise de código por meio dos chamados Agentes Inteligentes de Análise de Código (ICAA). O estudo parte da observação de que as abordagens tradicionais de análise se baseiam fortemente em regras fixas e heurísticas manuais, o que limita sua capacidade de adaptação e generalização frente à diversidade de estilos e estruturas de código encontradas em ambientes reais.

Esses agentes inteligentes propostos pelos autores são concebidos como sistemas híbridos que integram modelos de linguagem e aprendizado de máquina com técnicas tradicionais da engenharia de software. Eles são capazes de interpretar trechos de código não apenas sob uma ótica sintática, mas também semântica, identificando padrões recorrentes, inconsistências e possíveis falhas lógicas (FAN et al., 2023).

Nesse sentido, essas ferramentas apresentam um ganho significativo em precisão na detecção de *bugs* e na identificação de vulnerabilidades quando comparadas a ferramentas convencionais (FAN et al., 2023).

Ademais, Fan et al. defendem que os ICAAs representam um novo paradigma para a análise de software, com potencial para transformar práticas de revisão de código, detecção de falhas e garantia de qualidade. O estudo também sugere que pesquisas futuras devem focar em melhorar a aplicabilidade desses agentes, bem como em reduzir os custos de operação, tornando essas soluções mais acessíveis para ambientes corporativos.

3 METODOLOGIA

Neste capítulo, serão descritos os procedimentos metodológicos adotados durante a execução do estudo, incluindo os instrumentos utilizados para a coleta de dados. O objetivo é fornecer uma explicação detalhada sobre o processo que orientou a obtenção dos resultados e o alcance dos objetivos estabelecidos.

3.1 CARACTERIZAÇÃO DA PESQUISA

A pesquisa desenvolvida é de caráter bibliográfico e exploratório, com a utilização de um questionário para coleta de dado, com vistas à análise e interpretação dos resultados. A pesquisa bibliográfica é essencial para fornecer a base teórica necessária, além de auxiliar na definição e na escolha dos métodos adotados, bem como colaborar na construção da revisão de literatura e na redação do trabalho (LAKATOS; MARCONI, 2010).

A pesquisa exploratória, conforme definido por Vergara (2015), visa proporcionar uma melhor compreensão sobre um tema pouco abordado, com o intuito de explorar novas possibilidades e esclarecimentos acerca do assunto. Esse tipo de pesquisa é frequentemente empregado quando o tema é recente ou não está amplamente desenvolvido na literatura existente, sendo, portanto, uma etapa preliminar importante para o aprofundamento de estudos futuros.

No presente estudo, em virtude da aplicação de um questionário, a pesquisa assume uma abordagem quantitativa, que utiliza a quantificação de dados para garantir a precisão dos resultados, minimizando as chances de interpretações distorcidas e oferecendo uma base segura para as conclusões (CRESWELL, 2014).

3.2 TÉCNICAS DE COLETA DE DADOS

A coleta de dados foi realizada por meio da aplicação de um questionário elaborado visando obter informações sobre a utilização e importância de uma ferramenta de detecção de *Code Smells*. O questionário foi respondido de forma anônima pelos participantes e contém 15 questões, sendo 2 de múltipla escolha e 13 com respostas de escolha única.

A elaboração do questionário foi realizada por meio da plataforma Google Forms, e o link de acesso foi distribuído em grupos de WhatsApp e em outras redes sociais, como LinkedIn, visando alcançar um público diversificado e representativo. O período de coleta ocorreu entre os dias 01/05/2025 e 27/06/2025, totalizando 57 dias para a coleta dos dados. Ao final desse período, foram obtidas 57 respostas válidas, que serão utilizadas para a análise dos resultados e para a formulação das conclusões do estudo.

3.3 FORMA DE ANÁLISE DOS DADOS

Os dados coletados foram analisados com o auxílio de ferramentas computacionais, como o Microsoft Excel, que possibilitou a tabulação, organização e visualização dos dados de maneira clara por meio de gráficos e tabelas. A análise seguiu uma abordagem quantitativa, onde as respostas dos participantes foram classificadas e organizadas em categorias pertinentes para facilitar a interpretação.

Realizou-se uma análise qualitativa das respostas abertas, a fim de explorar as percepções e opiniões dos respondentes de maneira mais profunda. A combinação das duas abordagens, quantitativa e qualitativa, proporcionou uma análise abrangente e robusta, permitindo que as conclusões e recomendações do estudo fossem sustentadas por uma base sólida de dados.

4. RESULTADOS E DISCUSSÃO

Esta seção apresenta os dados coletados de forma objetiva, nesta ordem: contexto da amostra (quem respondeu), principais indicadores e percepções sobre *code smells*. Cada figura será comentada de forma concisa, destacando os resultados relevantes.

4.1 Contextualização dos Dados

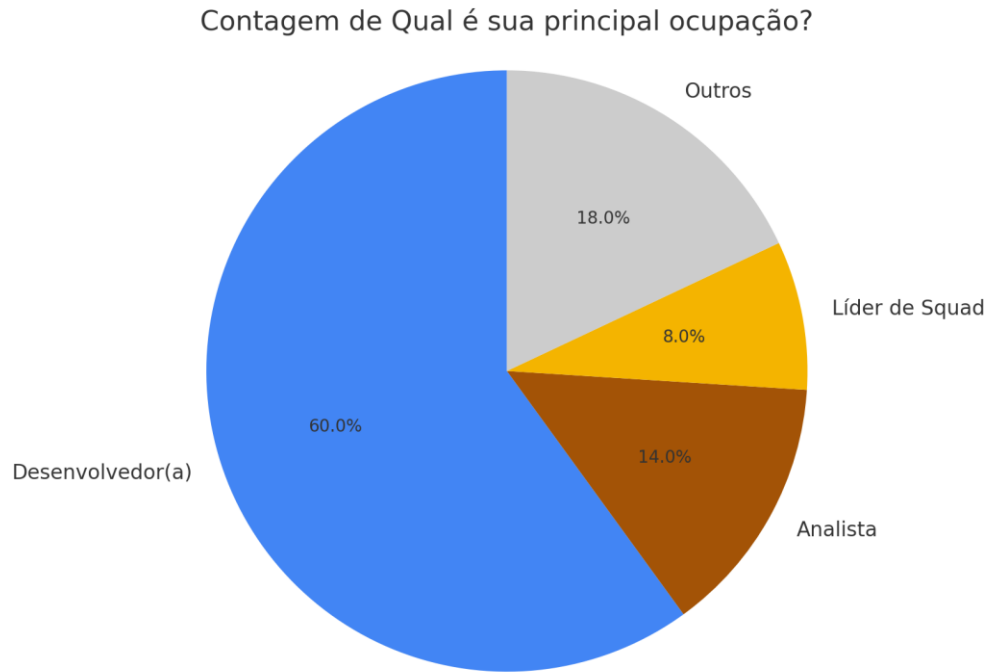
A amostra analisada foi composta por profissionais da área de tecnologia que participam ativamente de atividades relacionadas ao desenvolvimento de software. A seleção incluiu desenvolvedores, analistas, líderes de *squad* e outros cargos, com experiência variada e consolidação da prática de *code review* em seus respectivos ambientes. Também foi considerada a distribuição dos profissionais em diferentes plataformas de hospedagem de código, com destaque para aquelas com maior adesão.

Adicionalmente, foram identificadas as preferências dos participantes em relação às funcionalidades essenciais de ferramentas baseadas em inteligência artificial voltadas à revisão de código, visando mapear exatamente quais recursos são priorizados pelos usuários. Esse cenário preliminar cria o pano de fundo necessário para compreender os resultados obtidos, os quais serão apresentados com mais detalhes na seção seguinte.

4.2 Apresentação dos Resultados

Como ilustra a Figura 11 (Gráfico de Pesquisa: Principal ocupação), a pesquisa foi respondida por 57 profissionais da área de tecnologia, sendo a maioria composta por Desenvolvedores(as) (60%), seguidos por Analistas (14%), Líderes de Squad (8%) e outros cargos.

Figura 11 - Gráfico de Pesquisa: Principal ocupação

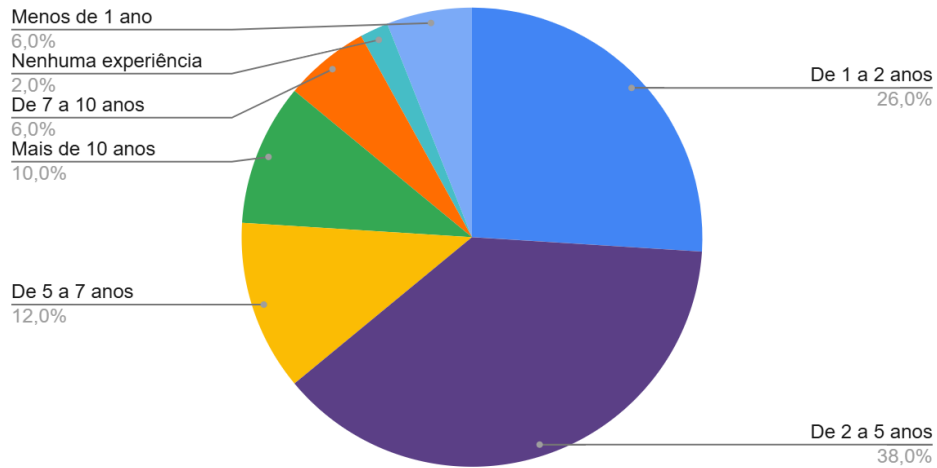


Fonte: Elaborado pelos autores (2025).

Quanto ao nível de experiência, mostrado na Figura 12 (Gráfico de Pesquisa: Nível de experiência) abaixo, na área de desenvolvimento, destacam-se os profissionais com 2 a 5 anos de experiência (35%) e 1 a 2 anos (27%). Isso indica que boa parte dos respondentes já possui familiaridade prática com processos de desenvolvimento e revisão de código.

Figura 12 - Gráfico de Pesquisa: Nível de experiência

Contagem de Qual é seu nível de experiência na área de Desenvolvimento?

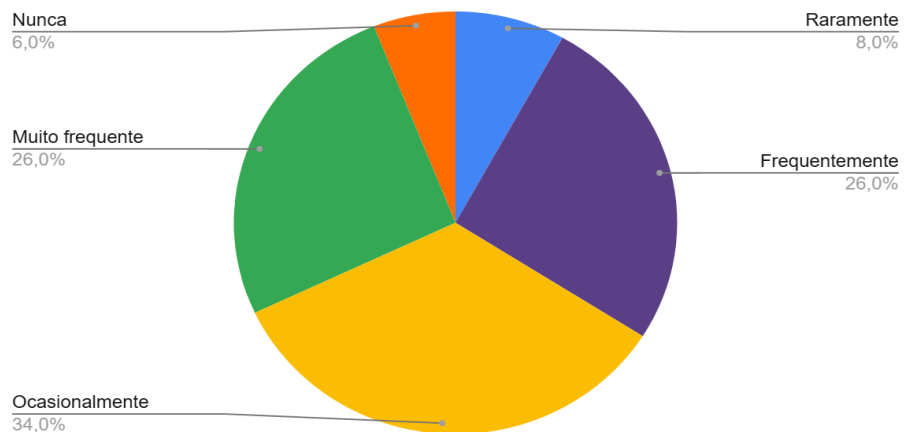


Fonte: Elaborado pelos autores (2025).

No que se refere à frequência com que realizam *Code Reviews*, 34% afirmaram fazê-los *ocasionalmente*, enquanto 26% o fazem *frequentemente* e outros 26% de forma *muito frequente*, demonstrando que a prática está consolidada em muitos ambientes de desenvolvimento, conforme Figura 13 - (Gráfico de Pesquisa: Frequência de *Code Reviews*).

Figura 13 - Gráfico de Pesquisa: Frequência de *Code Reviews*

Contagem de Com qual frequência você realiza *Code Reviews* em seu trabalho?

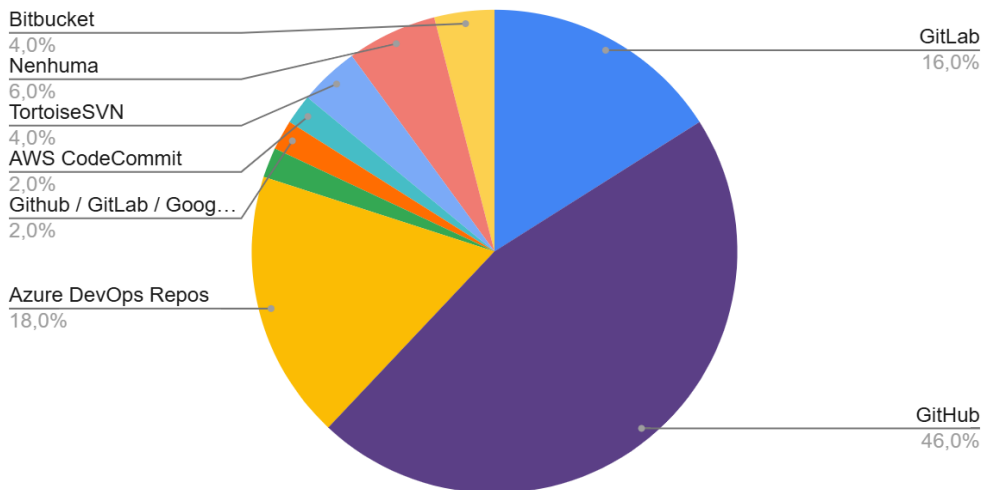


Fonte: Elaborado pelos autores (2025).

No que se refere o uso entre diferentes plataformas de hospedagem de código-fonte. O GitHub é amplamente dominante, sendo utilizado por 46% dos respondentes, seguido pelo Azure DevOps Repos (18%) e GitLab (16%). As demais opções, como Bitbucket, TortoiseSVN, AWS CodeCommit e outras, somam individualmente até 6%, representando uma adoção mais dispersa e menos significativa. A Figura 14 Gráfico de Pesquisa: Plataforma de hospedagem mais utilizada) reforça a concentração de uso em poucas plataformas, com destaque claro para GitHub.

Figura 14 - Gráfico de Pesquisa: Plataforma de hospedagem mais utilizada

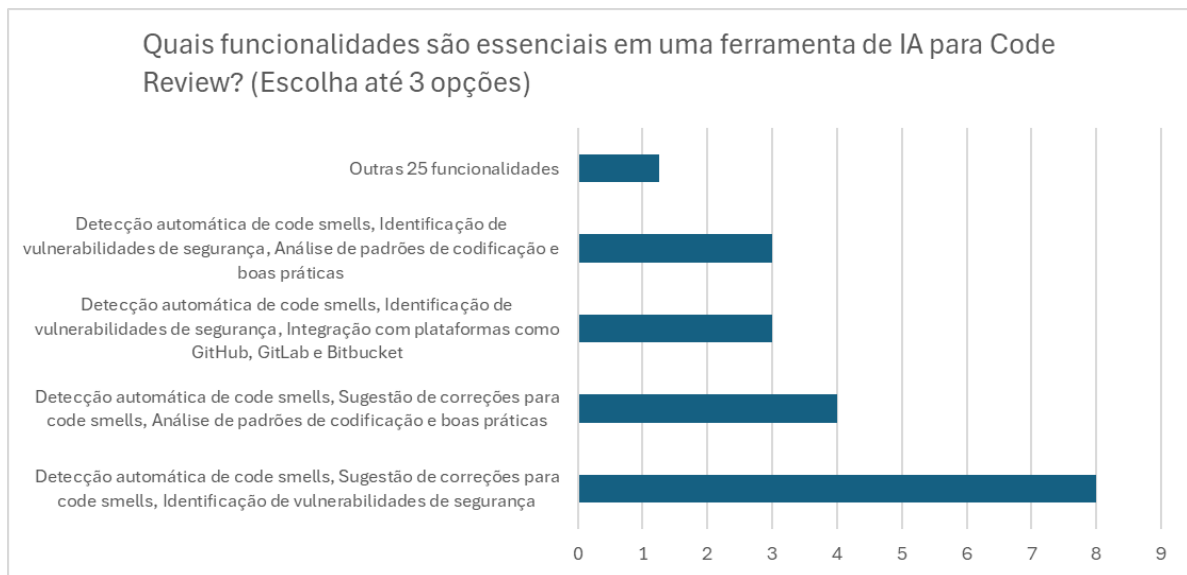
Contagem de Qual plataforma de hospedagem de código-fonte você mais utiliza?



Fonte: Elaborado pelos autores (2025).

Nas funcionalidades que os usuários consideram essenciais em uma ferramenta de IA para revisão de código. A principal preferência (com 8 votos) é por soluções que detectem *code smells*, sugiram correções e identifiquem vulnerabilidades de segurança. Outras combinações que incluem análise de padrões de codificação, integração com plataformas e sugestões de boas práticas também aparecem, mas com menos destaque. Um grupo genérico de “outras 25 funcionalidades” obteve apenas 2 ou menos menções, indicando que há um conjunto de prioridades bem definido entre os participantes, conforme Figura 15 (Gráfico de Pesquisa: Funcionalidades essenciais em uma ferramenta de IA).

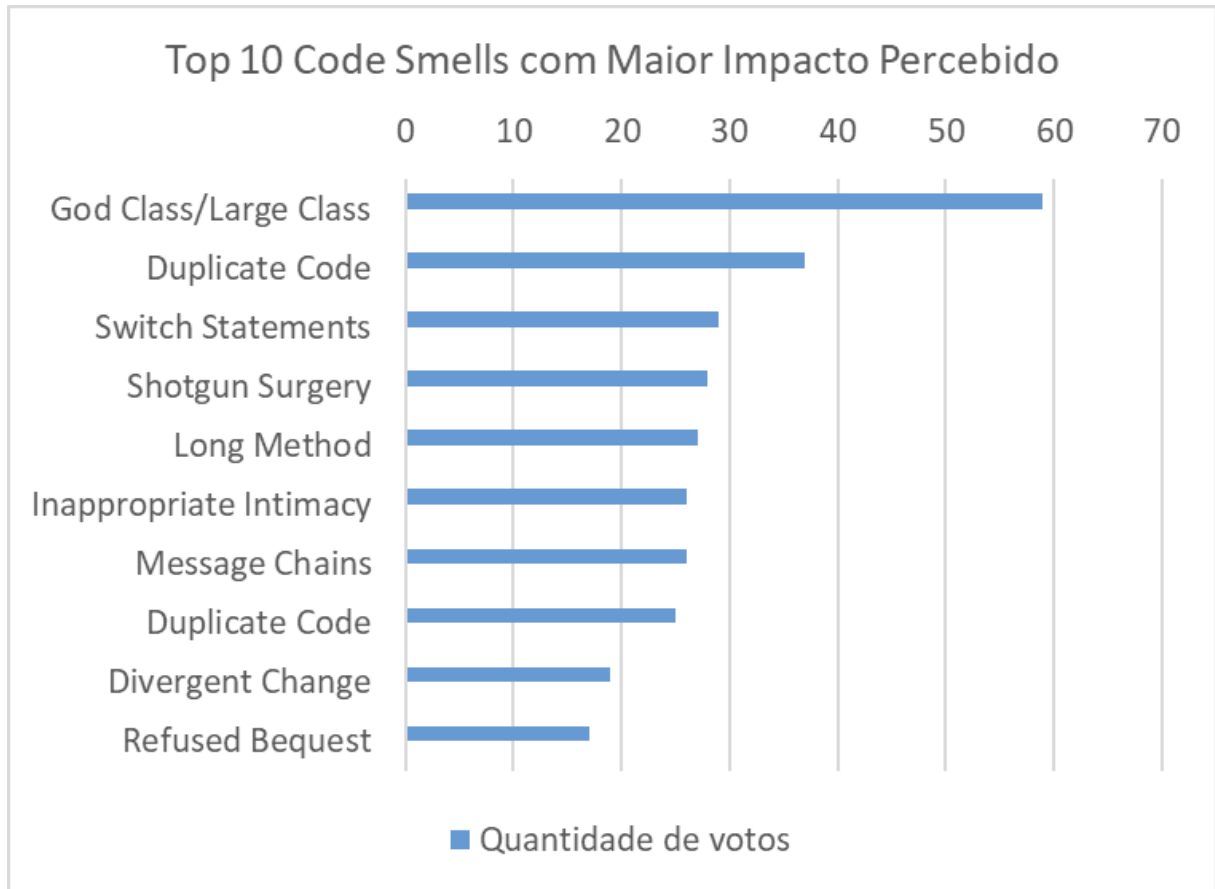
Figura 15 - Gráfico de Pesquisa: Funcionalidades essenciais em uma ferramenta de IA



Fonte: Elaborado pelos autores (2025).

Com o objetivo de identificar os *code smells* mais relevantes na percepção de desenvolvedores, foi realizada uma pesquisa empírica com profissionais da área. Os participantes avaliaram o impacto de diversos *smells* em termos de comprometimento da legibilidade, manutenção e evolução do código. A Figura 16 (Gráfico de Pesquisa: Média de importância do top 10 *code smells*) apresenta os dez *smells* considerados mais críticos. Destacam-se, entre eles, a repetição de blocos de código, classes com múltiplas responsabilidades e a necessidade de alterações espalhadas, todos com média de importância superior a 2,4 em uma escala de 1 (baixo) a 3 (alto). Esses resultados evidenciam a ênfase prática em aspectos ligados à duplicação, coesão e acoplamento, corroborando estudos como os de FOWLER (1999) e YAMASHITA; MOONEN (2013) sobre os *smells* que mais afetam a manutenibilidade.

Figura 16- Gráfico de Pesquisa: Média de importância do top 10 *code smells*



Fonte: Elaborado pelos autores (2025).

4.3 Análise Crítica dos Resultados

A análise dos dados revelou que grande parte dos participantes são desenvolvedores com até cinco anos de experiência, atuando em contextos nos quais a prática de code review já está consolidada, especialmente em plataformas como GitHub. Os code smells apontados como mais relevantes pelos entrevistados duplicação de código, God Class e Shotgun Surgery corroboram as preocupações já registradas em estudos de Fowler (1999) e Yamashita & Moonen (2013), enfatizando desafios tradicionais de manutenção ligados à coesão e acoplamento.

A preferência por funcionalidades como detecção automática de code smells e sugestões de refatoração reflete uma expectativa por ferramentas que integrem inteligência no apoio à revisão de código, sem, entretanto, associar diretamente esses resultados à promoção de qualquer solução específica. Esses achados permitem

identificar tendências emergentes na percepção profissional sobre qualidade estrutural do software.

Observou-se que, embora a amostra tenha incluído apenas 57 profissionais, abrangendo diferentes níveis de experiência e funções técnicas, ela oferece um retrato relevante das necessidades e percepções atuais do mercado. No entanto, é importante reconhecer que esse perfil pode refletir particularidades de determinados ambientes de trabalho, o que deve ser considerado ao generalizar os resultados para outras realidades técnicas.

A estrutura da pesquisa e os instrumentos utilizados (escalas de avaliação, perfil dos respondentes e perfil de ferramentas utilizadas) parecem adequados ao escopo proposto. Ainda assim, a amplitude de temas explorado desde práticas de revisão até percepções sobre funcionalidades de IA poderia beneficiar-se de abordagens metodológicas complementares ou de amostras mais diversificadas.

Finalmente, os dados indicam caminhos úteis para assistência futura ao processo de *code review* em ambiente corporativo. O estudo sugere que pesquisas subsequentes poderiam explorar a relação entre percepção e efetividade de ferramentas auxiliadas por IA, bem como aprofundar o impacto dessa tecnologia em projetos de grande escala ou em equipes com estruturas diferentes da predominante.

5 PROPOSIÇÃO DA UTILIZAÇÃO DE UM SISTEMA DE REVISÕES DE CÓDIGO PARA DETECÇÃO E SUGESTÃO DE CORREÇÃO DE *CODE SMELLS*

Com base nos dados apresentados na pesquisa e fundamentação teórica, esta seção propõe o desenvolvimento de um sistema de revisão de código automatizado que integra a análise de *pull requests* com a utilização de Inteligência Artificial para detectar *code smells* e sugerir possíveis refatorações.

5.1 PROPOSIÇÃO

A partir das evidências observadas na literatura e dos requisitos identificados durante a análise do problema, propõe-se o desenvolvimento do protótipo BloodHound. A ferramenta tem como objetivo analisar automaticamente *pull requests*, identificar *code smells* com apoio de IA e fornecer sugestões contextuais de melhoria baseadas em boas práticas de engenharia de software.

5.1.1 Spikes

Spikes são tarefas exploratórias utilizadas durante o desenvolvimento ágil com o objetivo de reduzir incertezas técnicas e investigar soluções para problemas complexos. Essas atividades não resultam diretamente em funcionalidades prontas, mas fornecem subsídios importantes para tomada de decisão e implementação de futuras funcionalidades. A seguir, são apresentados os principais *spikes* definidos para o desenvolvimento do protótipo BloodHound.

Tabela 3 - SPIKES

Estudo da API da OpenAI	Explorar o funcionamento da API da OpenAI, com foco nos modelos GPT para geração de respostas baseadas em análise de código. Avaliar limites, formatos de requisição e boas práticas de uso.
Engenharia de <i>Prompt</i>	Investigar técnicas para melhorar a performance das respostas do modelo por meio de engenharia de <i>prompts</i> , adaptando-os para detecção e sugestão de refatorações de <i>code smells</i> .

Análise da estrutura de <i>pull requests</i>	Estudar como os repositórios Git estruturam <i>pull requests</i> em plataformas como GitHub, identificando os dados essenciais para análise automatizada.
Armazenamento de análises	Explorar formas de persistir os resultados das análises feitas pela IA em um banco de dados, garantindo rastreabilidade e histórico por <i>pull request</i> .
Classificação de <i>code smells</i> relevantes	Avaliar os <i>code smells</i> mais recorrentes e impactantes, com base na literatura, para priorização na fase inicial da ferramenta.

Fonte: Elaborado pelos autores (2025)

5.1.2 Arquitetura

A arquitetura do BloodHound foi estruturada de forma modular, utilizando tecnologias modernas e serviços em nuvem para garantir integração eficiente com plataformas de versionamento e facilidade de manutenção.

Conforme ilustrado na Figura 17 (Arquitetura do Sistema), o sistema é composto por cinco principais componentes:

Frontend (Azure Static Web Apps): Interface web acessada pelo usuário, onde é possível autenticar-se via GitHub, acompanhar análises realizadas e visualizar sugestões de melhorias no código.

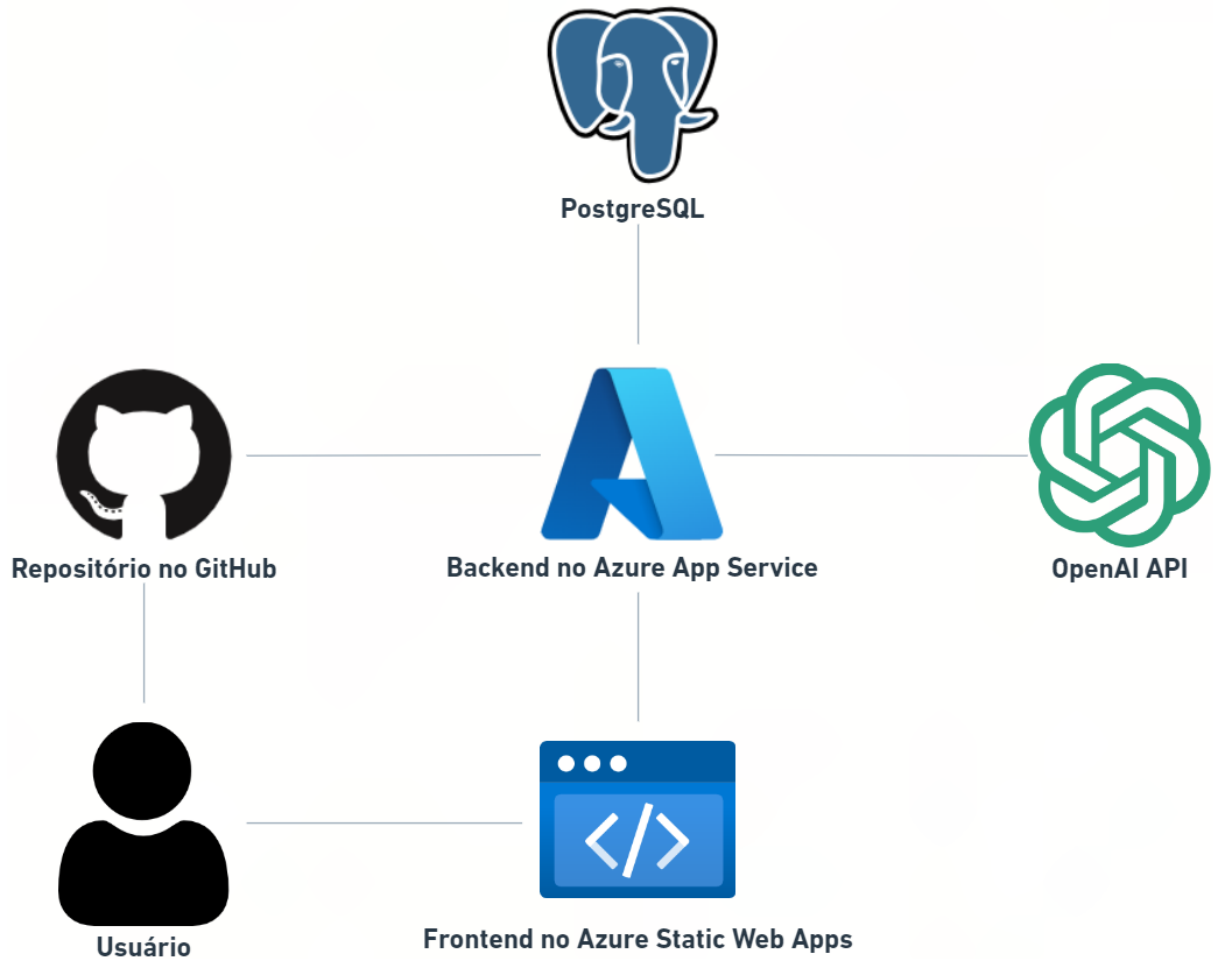
Backend (Azure App Service): Responsável por orquestrar toda a lógica do sistema, como o recebimento de eventos do GitHub, envio de código para análise via IA e armazenamento das informações processadas.

Banco de Dados (PostgreSQL): Utilizado para persistir dados dos usuários, repositórios conectados, análises realizadas e histórico de sugestões emitidas.

GitHub: Plataforma de versionamento onde ocorre a instalação do bot do BloodHound, que monitora os repositórios dos usuários e envia eventos como *pull requests* para análise.

OpenAI API: Serviço de inteligência artificial utilizado para analisar trechos de código e sugerir melhorias com base em boas práticas de programação.

Figura 17 - Arquitetura do Sistema



Fonte: Elaborado pelos autores (2025)

Essa composição arquitetural procura favorecer a modularidade do sistema, permitindo que cada componente desempenhe sua função de forma independente, mas coordenada. Além disso, a adoção de recursos em nuvem visa garantir maior escalabilidade, manutenibilidade e capacidade de integração com ferramentas amplamente utilizadas na engenharia de software contemporânea.

5.1.3 Histórias de Usuário

As histórias de usuário são uma técnica utilizada em metodologias ágeis para representar, de forma simples e centrada no usuário, as funcionalidades esperadas de um sistema. Elas descrevem, sob a perspectiva do usuário, o que deve ser desenvolvido e qual o valor gerado pela funcionalidade. No contexto do protótipo BloodHound, essas histórias servem como base para orientar o desenvolvimento das principais interações do sistema com seus usuários. A seguir, apresenta-se o conjunto de histórias de usuário definidas para este projeto.

Tabela 4 - História de usuário

Login via GitHub	Como um usuário, quero fazer login com minha conta do GitHub, para acessar rapidamente o sistema sem precisar criar uma nova conta.
Visualização de repositórios	Como um usuário autenticado, quero visualizar todos os meus repositórios conectados, para escolher com quais deles quero trabalhar.
Criação e edição de regras por repositório	Como um usuário, quero cadastrar e editar regras específicas para cada repositório, para configurar critérios personalizados de validação de código.
Visualização de pull requests por repositório	Como um usuário, quero visualizar todos os pull requests de um repositório, para acompanhar o andamento das contribuições e revisões.
Visualizar informações completas de um <i>pull request</i>	Como um usuário, quero visualizar o nome da branch, status, e acessar o link do GitHub de cada <i>pull request</i> , para poder abrir o PR diretamente no GitHub com um clique.
Carregamento com <i>spinner</i> durante o login	Como um usuário, quero ver um indicador de carregamento enquanto sou redirecionado pelo GitHub, para entender que a autenticação está em andamento.
Exibição dinâmica das regras aplicadas	Como um usuário, quero ver quais regras estão sendo aplicadas a cada repositório, para garantir que os critérios definidos estão sendo utilizados corretamente.

Integração com a API do GitHub para detalhes	Como um desenvolvedor, quero que o sistema busque automaticamente detalhes dos <i>pull requests</i> via API do GitHub, para exibir o link direto ao PR e demais informações que não estão salvas no banco.
Acesso rápido aos detalhes do PR no GitHub	Como um usuário, quero clicar no número do PR e ser redirecionado para o GitHub, para revisar, comentar ou aprovar o <i>pull request</i> .
Análise automatizada de <i>code smells</i> ao abrir um PR	Como um usuário, quero que o sistema analise automaticamente o código de um <i>pull request</i> ao ser aberto, para identificar possíveis <i>code smells</i> sem intervenção manual.
Visualização dos <i>code smells</i> identificados em um PR	Como um usuário, quero visualizar os <i>code smells</i> detectados em cada <i>pull request</i> , para entender onde estão os problemas no código antes da aprovação.
Sugestões de melhoria para cada <i>code smell</i>	Como um usuário, quero ver sugestões de correções ou refatorações para cada <i>code smell</i> encontrado, para aplicar melhorias com base em boas práticas de desenvolvimento.
Contextualização da análise com base nas regras configuradas	Como um usuário, quero que a análise de <i>code smells</i> respeite as regras personalizadas do repositório, para garantir que os critérios específicos da minha equipe sejam considerados.
Envio automático de comentários com sugestões no PR	Como um usuário, quero que as sugestões de correção de <i>code smells</i> sejam postadas automaticamente nos comentários do <i>pull request</i> no GitHub, para facilitar a revisão e a comunicação com a equipe.

Fonte: Elaborado pelos autores (2025)

5.2 SOFTWARE DESENVOLVIDO

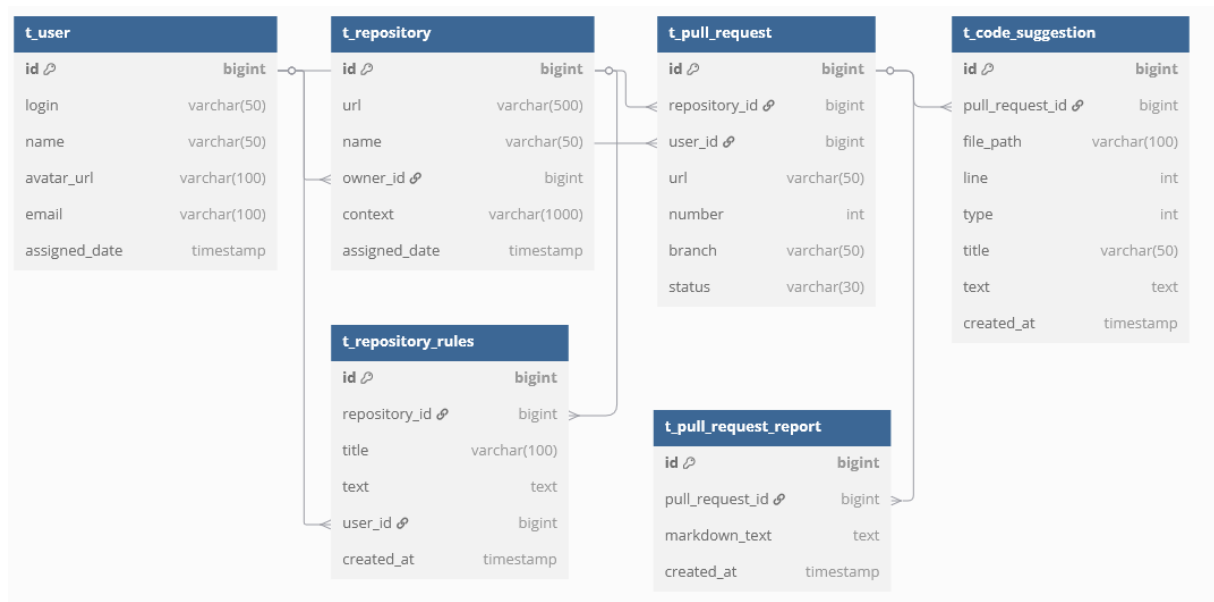
A partir das funcionalidades levantadas na etapa de proposição e dos requisitos definidos com base na fundamentação teórica, foi desenvolvido o protótipo da ferramenta BloodHound.

Nesta seção, serão apresentados o Modelo Entidade-Relacionamento (MER), responsável por estruturar a base de dados do sistema, bem como as principais interfaces gráficas desenvolvidas, que demonstram a interação do usuário com a aplicação.

5.2.1 Modelo Entidade Relacionamento

O modelo Entidade-Relacionamento (MER) da aplicação BloodHound, representado na Figura 18 (Modelo Entidade Relacionamento), foi projetado para suportar integralmente o fluxo de revisão automatizada de código, especialmente voltado à análise de *pull requests* e detecção de *code smells*. O sistema contempla entidades que representam usuários, repositórios, regras de análise, *pull requests*, sugestões de código e relatórios gerados pela inteligência artificial.

Figura 18 - Modelo Entidade Relacionamento



Fonte: Elaborado pelos autores (2025)

A seguir, descreve-se a estrutura e os relacionamentos principais do MER:

- A tabela `t_user` armazena os dados dos usuários que interagem com o sistema, como login do GitHub, nome, avatar e e-mail.
- A tabela `t_repository` representa os repositórios vinculados à conta do usuário. Cada repositório pertence a um usuário específico (relação muitos-para-um).
- A tabela `t_pull_request` registra os *pull requests* associados a um repositório e ao usuário que realizou a submissão.
- A tabela `t_code_suggestion` armazena sugestões de melhorias geradas pela IA para cada trecho analisado em um *pull request*.

- A tabela `t_pull_request_report` contém os relatórios em formato *Markdown* gerados a partir da análise do código.
- A tabela `t_repository_rules` permite o registro de regras de análise personalizadas definidas por cada usuário para seus repositórios.

O MER reflete uma estrutura relacional coesa, capaz de garantir rastreabilidade das análises e flexibilidade na configuração de regras, integrando os dados capturados com o GitHub ao processo de detecção automatizada de *smells*. No Anexo A encontra-se o Dicionário de Dados das Tabelas do MER para mais detalhes.

5.2.2 Integração com GitHub

Para viabilizar a análise automática de *pull requests*, o protótipo BloodHound estabelece uma integração direta com o GitHub. Essa integração ocorre por meio de diferentes recursos disponibilizados pela plataforma, como autenticação via OAuth 2.0, configuração de *webhooks* e instalação de *bots* personalizados, ou seja, agentes automatizados desenvolvidos para executar tarefas específicas dentro de repositórios.

O primeiro passo da integração se dá no momento em que o usuário realiza login no sistema BloodHound. Utilizando o protocolo OAuth 2.0, o sistema redireciona o usuário para a autenticação via GitHub, permitindo que, mediante consentimento, sejam concedidas permissões específicas de acesso aos repositórios associados à sua conta.

Esse processo garante segurança, controle de escopo e praticidade para os usuários. Um exemplo de código envolvido nessa integração é o visto na Figura 19 (Fragmento de código referente à autenticação com OAuth 2.0)

Figura 19 - Fragmento de código referente à autenticação com OAuth 2.0

```

[HttpGet("callback")]
[ApiVersion("1.0")]
public async Task<IActionResult> Callback([FromQuery] string code)
{
    var accessToken = await _githubService.GetUserAccessToken(code);

    var userDTO = await _githubService.GetUserInfo(accessToken);

    var user = await _userService.AssignUser(userDTO);

    var installationId = await _githubService.GetUserInstallation(accessToken);

    if (installationId is null)
    {
        var appSlug = _config["GitHub:AppSlug"];
        var installUrl = $"https://github.com/apps/{appSlug}/installations/new";
        return Redirect(installUrl);
    }

    var userToken = GenerateToken(user.Login);
    var redirectUrl = $"http://localhost:4200/auth/callback?token={userToken.Token}&expiration={userToken.Expiration:0}";
    return Redirect(redirectUrl);
}

```

Fonte: Elaborado pelos autores (2025)

Após a autenticação, o usuário permite a instalação o *bot* do BloodHound em sua organização ou repositório pessoal. Essa instalação define o nível de acesso do *bot*, como leitura de metadados do repositório, permissão para escutar eventos de *pull requests* e leitura do conteúdo dos arquivos alterados. Com isso, o *bot* passa a atuar como um agente observador, sem alterar o conteúdo do repositório diretamente.

Uma vez instalado, o *bot* é integrado aos *webhooks* do GitHub. Os *webhooks* são responsáveis por notificar o sistema BloodHound sempre que uma nova *pull request* é aberta, atualizada ou reaberta. Essas notificações são enviadas por meio de requisições HTTP contendo os dados necessários para que o sistema realize a análise automatizada do código. Um exemplo de código envolvido nessa integração é o visto na Figura 20 (Fragmento de código referente à comunicação com o Webhook).

Figura 20 - Fragmento de código referente à comunicação com o Webhook

```

[HttpPost(template: "webhook")]
[ApiVersion("1.0")]
Edson Adriano +1
public async Task<IActionResult> HandleWebhook([FromBody] JsonElement payload)
{
    var action :string? = payload.GetProperty("action").GetString();
    var installationId :long = payload.GetProperty("installation").GetProperty("id").GetInt64();

    string? installationUserLogin = null;

    if (payload.TryGetProperty("installation", out var installation :JsonElement) &&
        installation.TryGetProperty("account", out var account :JsonElement) &&
        account.TryGetProperty("login", out var loginProp :JsonElement))
    {
        installationUserLogin = loginProp.GetString();
    }

    User installationUser = new User();

    if (installationUserLogin is not null)
        installationUser = await _userService.GetUserByLogin(installationUserLogin);

    if (payload.TryGetProperty("pull_request", out var pr :JsonElement) && action == "opened"){...}
    else if (payload.TryGetProperty("repositories", out var CreatedRepositories :JsonElement) && action == "created"){...}
    else if (payload.TryGetProperty("repositories", out var DeletedRepositories :JsonElement) && action == "deleted"){...}
    else if (payload.TryGetProperty("repositories_added", out var AddedRepositories :JsonElement) &&
        action == "added"){...}
    else if (payload.TryGetProperty("repositories_removed", out var RemovedRepositories :JsonElement) &&
        action == "removed"){...}

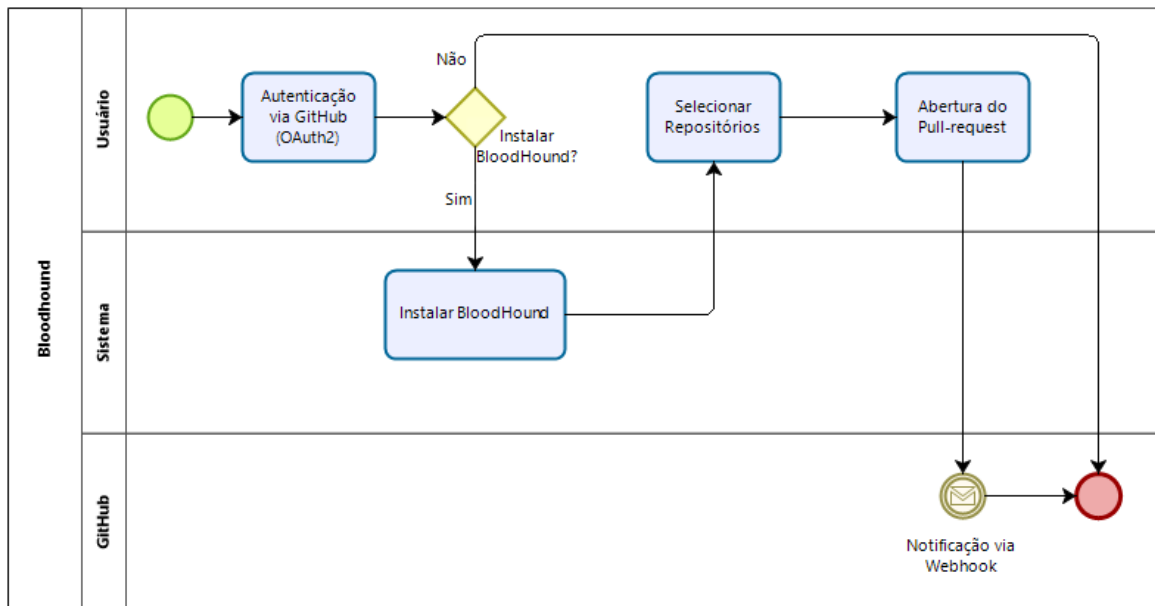
    return Ok();
}

```

Fonte: Elaborado pelos autores (2025)

Para fins de clareza quanto à comunicação entre os componentes, a Figura 21 (Fluxo de Integração com o GitHub) ilustra as principais etapas envolvidas no processo de autenticação via OAuth2, instalação do aplicativo do GitHub e ativação dos *webhooks* responsáveis por acionar a análise automatizada a cada novo *pull request*.

Figura 21 - Fluxo de Integração com GitHub



Powered by
bizagi
Modeler

Fonte: Elaborado pelos autores (2025)

Esse processo torna o fluxo contínuo e transparente para o usuário: ao abrir uma nova *pull request* no GitHub, o *bot* do BloodHound que está instalado é automaticamente acionado via *Webhook*, realiza a chamada para a análise do código com apoio de inteligência artificial. Dessa forma, a integração com o GitHub permite ao BloodHound atuar de maneira reativa e não invasiva, incorporando-se ao ecossistema de desenvolvimento sem alterar a rotina das equipes.

5.2.3 Integração com a API da OpenAI

A análise dos trechos de código no BloodHound é potencializada pela integração com a API da OpenAI, que fornece acesso a modelos de linguagem de última geração. Essa integração permite que o sistema utilize inteligência artificial para identificar *code smells* e sugerir refatorações de forma contextualizada e alinhada às boas práticas de engenharia de software.

A comunicação com a API ocorre por meio de requisições HTTP enviadas do backend do BloodHound para o endpoint da OpenAI. O conteúdo do *pull request* — incluindo arquivos modificados, descrições e o contexto do repositório — é utilizado

para compor um *prompt* detalhado, enviado ao modelo por meio de uma chamada POST.

Durante essa requisição, são especificados parâmetros como o modelo a ser utilizado (neste caso, gpt-4o), o *prompt* com instruções específicas sobre o tipo de análise esperada e, opcionalmente, configurações como temperatura (para controlar a criatividade da resposta) e limites de *tokens*, como ilustrado na Figura 21 (Fragmento de código referente à comunicação com a OpenAI API) abaixo.

Figura 22 - Fragmento de código referente à comunicação com a OpenAI API

```
public async Task<List<PullRequestComment>> GetPullRequestCommentByFiles(long repositoryId, List<PullRequestFile> files)
{
    var comments = new List<PullRequestComment>();

    foreach (var file in files)
    {
        var prompt = await GetPrompt(file.Path, file.Content, repositoryId);

        var requestBody = new
        {
            model = "gpt-4o",
            messages = new[]
            {
                new { role = "system", content = "Você é um revisor de código especializado em detectar code smells e sugerir melhorias." },
                new { role = "user", content = prompt }
            },
            temperature = 0.3
        };

        var requestJson = JsonSerializer.Serialize(requestBody);
        var content = new StringContent(requestJson, Encoding.UTF8, "application/json");

        var client = _httpClientFactory.CreateClient();

        client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Bearer", _openAIKey);
        var response = await client.PostAsync("https://api.openai.com/v1/chat/completions", content);

        if (!response.IsSuccessStatusCode)
            continue; // ou log

        var responseJson = await response.Content.ReadAsStringAsync();
        using var doc = JsonDocument.Parse(responseJson);

        var message = doc.RootElement
            .GetProperty("choices")[0]
            .GetProperty("message")
            .GetProperty("content")
            .GetString();

        if (!string.IsNullOrWhiteSpace(message) && !message.Contains("NO CODE SMELLS"))
        {
            comments.Add(new PullRequestComment
            {
                Path = file.Path,
                Position = 1,
                Body = message
            });
        }
    }

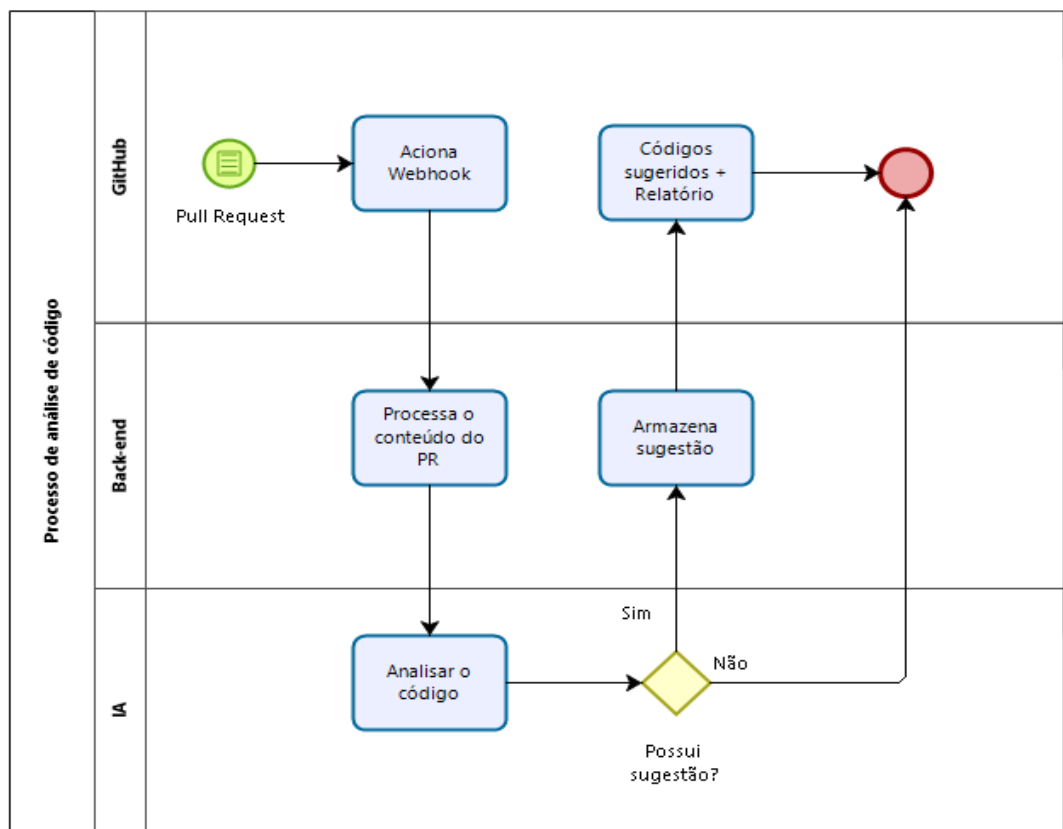
    return comments;
}
```

Fonte: Elaborado pelos autores (2025)

Após o envio, a resposta da API contém um texto gerado pelo modelo, que é processado e armazenado pelo *backend* do BloodHound. Essa resposta pode conter a identificação de *code smells*, explicações técnicas, sugestões de refatoração e até referências a boas práticas de design de software, dependendo da instrução incluída no *prompt*.

Abaixo, representado na Figura 23 (Fluxo de Integração com API da OpenAI) segue-se o processo de integração do sistema com o GitHub.

Figura 23 - Fluxo de Integração com a OpenAI API



Fonte: Elaborado pelos autores (2025)

Esse fluxo visa tornar possível a geração de análises inteligentes sem a necessidade de que desenvolvedores compreendam todos os detalhes internos da IA, buscando promover acessibilidade, eficiência e qualidade contínua no processo de revisão de código.

5.2.4 Prompt utilizado

O sistema BloodHound utiliza um prompt estruturado com base em técnicas de engenharia de prompt aplicadas à interação com LLMs, a fim de conduzir o modelo a operar como um revisor de código especializado. A formulação do prompt foi orientada por abordagens sistematizadas descritas na literatura recente sobre prompt engineering.

A construção do prompt adota a técnica de *Chain-of-Thought Prompting* (CoT), que incentiva o modelo a realizar raciocínios intermediários para a análise de smells, permitindo que o modelo divida decisões complexas em etapas sucessivas e interprete melhor os padrões de design inadequados em código-fonte, como Feature Envy. Tal abordagem foi validada por WEI et al. (2022) e demonstrou ganhos na acurácia de tarefas que exigem raciocínio estruturado.

A técnica de *Self-Consistency*, proposta por WANG et al. (2022), também está presente implicitamente ao estruturar o prompt de modo a exigir explicações técnicas precedendo as sugestões de código. Isso induz o modelo a validar sua própria inferência antes de apresentar uma sugestão concreta, promovendo coerência entre diagnóstico e recomendação.

A utilização de *Prompt Chaining*, conforme discutido por AMATRIAIN (2024), é observada na decomposição do fluxo de entrada do prompt, que incorpora: (i) a especificação do papel do modelo (revisor de código); (ii) a definição das *code smells* a serem identificadas; (iii) a apresentação das regras do repositório; (iv) o fornecimento do conteúdo do código a ser analisado; e (v) a delimitação do formato de resposta. Essa sequência estruturada permite que o modelo construa contexto progressivamente, facilitando a tomada de decisão baseada em múltiplas camadas de instruções.

A técnica *ReAct* (*Reasoning and Acting*), conforme descrita por YAO et al. (2022), também é parcialmente utilizada. O prompt demanda que o modelo avalie logicamente os problemas no código antes de “agir”, isto é, propor refatorações. Essa separação entre inferência e ação é reforçada pela obrigatoriedade de incluir uma explicação técnica antes de qualquer bloco de sugestão, contribuindo para maior transparência e rastreabilidade da recomendação.

O modelo ainda se beneficia de *In-Context Learning*, principalmente na variante *zero-shot*, conforme discutido por SAHOO et al. (2024). Ao fornecer exemplos estruturais da tarefa (como o formato esperado de resposta e os tipos possíveis de code smells), o *prompt* permite que o modelo generalize comportamentos desejáveis mesmo na ausência de demonstrações explícitas para cada cenário.

Adicionalmente, foi incorporada a técnica de *prompt patterns* catalogados, como proposto por WHITE et al. (2023), com o uso de estruturas padronizadas de formatação (ex: uso de Markdown, enumeração estruturada, *fallback* explícito) que aumentam a previsibilidade da saída e reduzem ambiguidades de interpretação.

Essas técnicas foram selecionadas com base em estudos sistemáticos como o de SCHULHOFF et al. (2024), que avaliam a efetividade de padrões de engenharia de prompt para tarefas especializadas em LLMs.

A Figura 24 (Fragmento Inicial do *Prompt* enviado à API da OpenAI) apresenta a estrutura inicial do *prompt* enviada à API da OpenAI durante o processo de análise:

Figura 24 - Fragmento Inicial do Prompt enviado à API da OpenAI

```
You are a code reviewer specialized in software engineering and best practices. Carefully analyze the following code and identify possible *code smells*.

Your analysis should consider the following criteria:

---

🔗 Code Smells to Identify

During the analysis, identify and suggest corrections for any existing *code smells*, focusing especially with special attention and rigor in identifying Feature Envy, Large Class / God Class and Long Method:

Feature Envy
Happens when a method in one class is overly interested in the data or methods of another class.
- Methods that access multiple fields or methods from another class more than their own;
- Patterns like: `external.getSomething().doAnotherThing()` or heavy use of getters from other objects;
- Cases where logic could be moved to the data-owning class for better cohesion;
- Use of foreign data manipulation instead of internal responsibility handling.

Large Class / God Class
A class that takes on too many responsibilities and grows beyond its intended purpose.
- Becomes hard to read, test, and maintain;
- Often manipulates unrelated concerns like persistence, presentation, and domain logic;
- Breaks encapsulation and the SRP.

Long Method
This smell exists when a method tries to do too much, even if it's short in lines.
- Often mixes concerns like business logic, validation, and I/O;
- Increases complexity and reduces reusability;
- Difficult to read and test in isolation.
```

Fonte: Elaborado pelos autores (2025)

A Figura 25 (Fragmento Final do *Prompt* enviado à API da OpenAI) apresenta a estrutura final do *prompt* enviada à API da OpenAI durante o processo de análise:

Figura 25 - Fragmento Final do Prompt enviado à API da OpenAI

```

**Expected Response Format (Function Calling returning `PullRequestComment[]`)**

For each identified *code smell*, generate an object of the `PullRequestComment` class in the following format:

```json
{
 "path": "<path of the analyzed file>",
 "type": "<Enum of Code Smell Type>",
 "start_line": <number of the first affected line>,
 "end_line": <number of the last affected line>,
 "body": "🔍 Code Smell Detected: <Name of the Code Smell>\n\n💡 Improvement suggestion: <objective, technical, and clear description of the problem and how to refactor>\n\n"
}
```

⚠️ Important Constraints and Validation Rules:
- Do not return suggestions that contain only comments or vague remarks.
- Do not return placeholders like `// improve this` or `// extract method`.
- The code inside the `suggestion` block must be executable and reflect a real refactoring (e.g., extracted method, class split, delegation moved).
- Only submit the final version of the refactored code as it would appear in the actual file, not just a description of what to do.
- The response must be a list (array) of PullRequestComment objects, as a single file may contain multiple code smells.
- These are the possible enums for the type: DivergentChange, ShotgunSurgery, LongParameterList, FeatureEnvy, DataClumps, LargeClass, GodClass, LongMethod, MessageChains, Middle
- The start line must always be 1, so the suggestion should begin from the first line of the file.
- The explanation must always come before the suggestion block and follow GitHub's Pull Request Review style.
- The explanation should be modern, technical, and direct, avoiding outdated terms or vague jargon.
- The explanation and suggestions must be in Portuguese.
- Code suggestions should not be just comments. The proposed snippet must show real modifications or a refactored version with executable code.
- If no code smells are identified, return exclusively: ```json []```

❌ Bad Example (do not return this, with just a line of code comment):

```json
[
 {
 "body": "Code Smell Detectado: Método Longo\n\n💡 Refatorar este método. Está muito longo. ```suggestion\n//Separar responsabilidades\n```"
 }
]
```

```

Fonte: Elaborado pelos autores (2025)

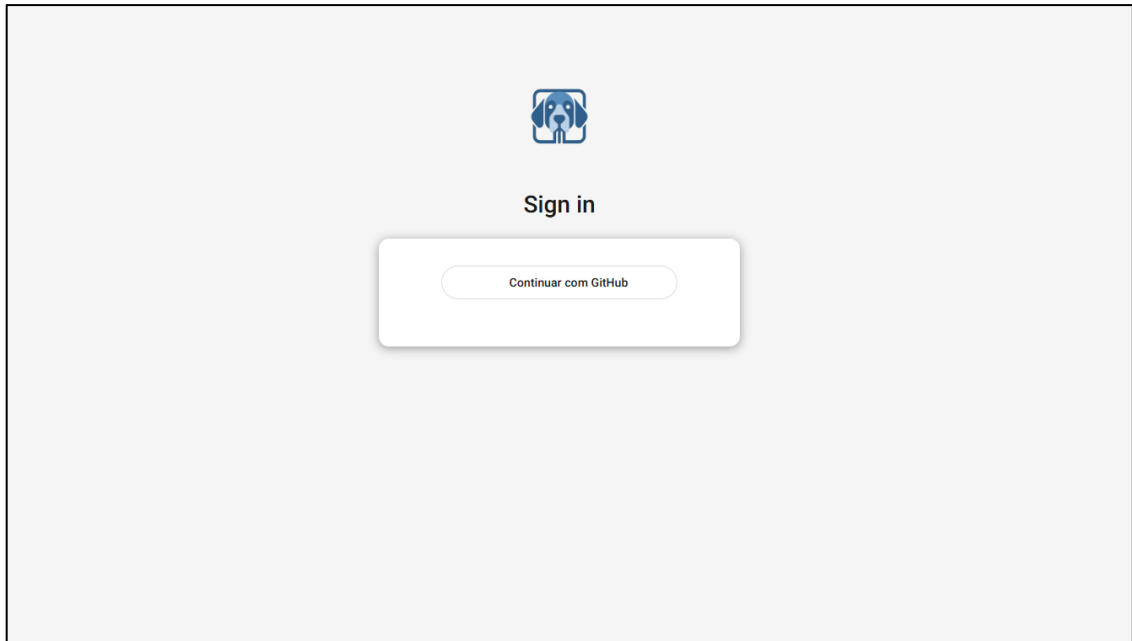
Essas técnicas foram selecionadas com base em estudos sistemáticos como o de SCHULHOFF et al. (2024), que avaliam a efetividade de padrões de engenharia de prompt para tarefas especializadas em LLMs e implementadas na estruturação do *prompt* enviado à OpenAI API.

5.2.5 Operacionalização do Sistema

A interface web do BloodHound foi desenvolvida com foco na praticidade e integração direta com o fluxo de trabalho dos desenvolvedores.

A Figura 26 (Sistema proposto: Página de login) apresenta a autenticação do usuário é realizada por meio do protocolo OAuth2 do GitHub, permitindo acesso seguro e direto à conta e repositórios do usuário.

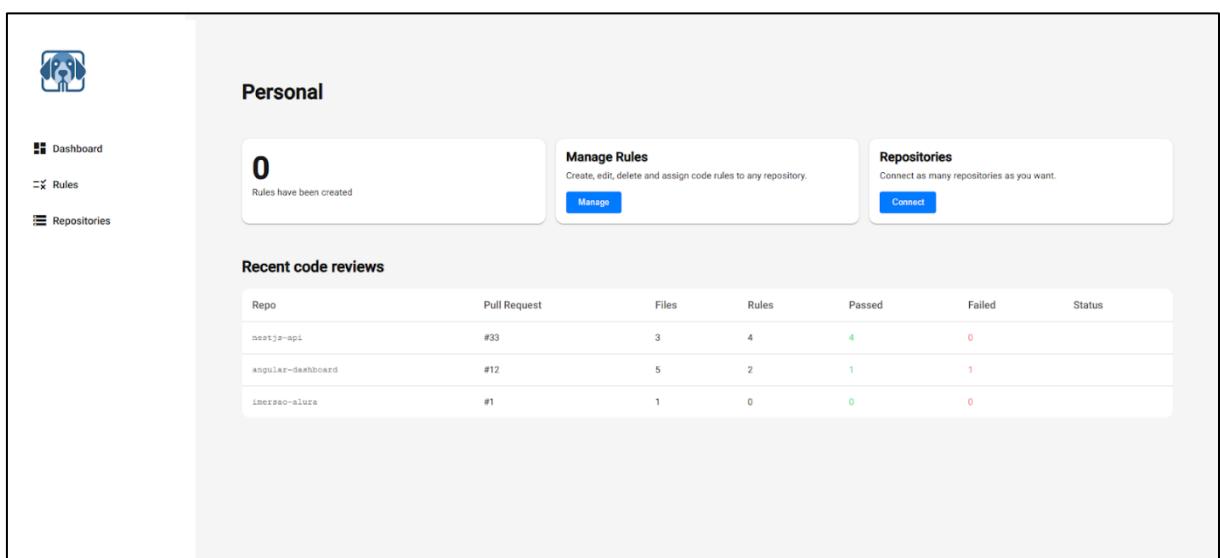
Figura 26 - Sistema proposto: Página de login



Fonte: Elaborado pelos autores (2025)

Após o login, o usuário é direcionado ao *Dashboard*, apresentado na Figura 27 (Sistema proposto: Página inicial - Dashboard), onde são exibidas informações dos *pull requests* recentes analisados, com indicadores visuais de status e *feedbacks* gerados.

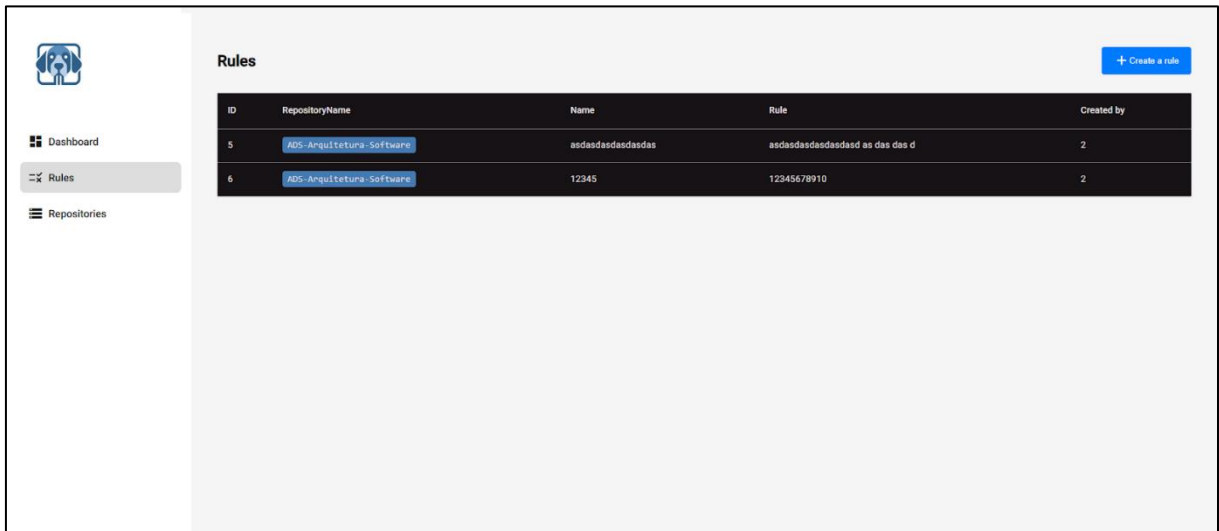
Figura 27 - Sistema proposto: Página inicial - Dashboard



Fonte: Elaborado pelos autores (2025)

Partindo do *Dashboard*, o usuário pode personalizar regras de análise para cada repositório utilizando uma estrutura de chave-valor, permitindo um controle mais refinado dos critérios utilizados na detecção de *code smells*, como pode ser observado na Figura 28 (Sistema proposto: *Rules*).

Figura 28 - Sistema proposto: Rules

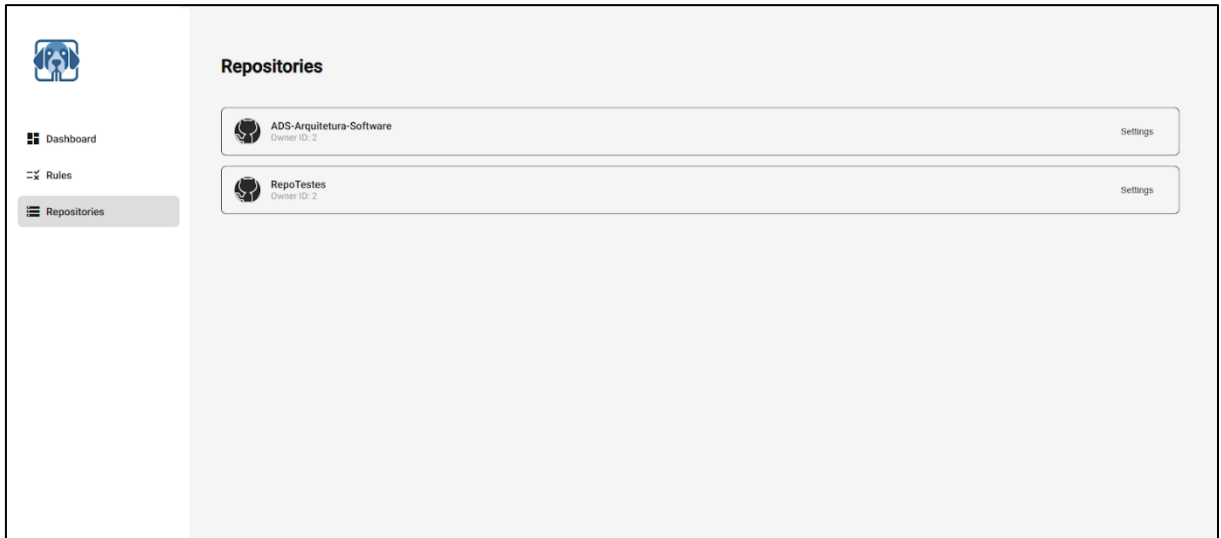


| ID | RepositoryName | Name | Rule | Created by |
|----|--------------------------|-----------------|------------------------------|------------|
| 5 | ADS-Arquitetura-Software | asdasdasdasdasd | asdasdasdasdasd as das das d | 2 |
| 6 | ADS-Arquitetura-Software | 12345 | 12345678910 | 2 |

Fonte: Elaborado pelos autores (2025)

Além de regras personalizadas, o usuário pode acessar todos seus repositórios do GitHub que foram conectados ao BloodHound, conforme Figura 29 (Sistema proposto: Repositórios).

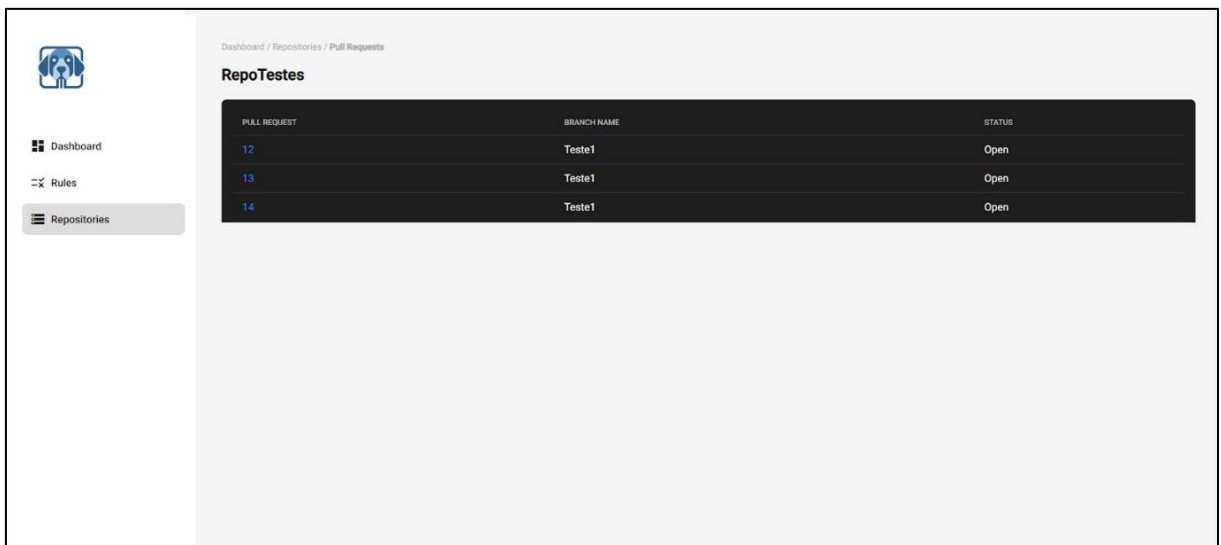
Figura 29 - Sistema proposto: Repositórios



Fonte: Elaborado pelos autores (2025)

Ao clicar em um repositório, o usuário é redirecionado para a tela de *pull requests* correspondentes, conforme Figura 30 (Sistema proposto: *Pull requests*). Ao selecionar um *pull request* específico, o usuário é redirecionado para a interface do GitHub, diretamente na página do *pull request*.

Figura 30 - Sistema proposto: Pull requests



Fonte: Elaborado pelos autores (2025)

No ambiente do GitHub, o sistema destaca automaticamente os trechos de código onde foram identificados *code smells*, adicionando comentários com sugestões de melhoria logo abaixo das linhas afetadas, promovendo uma revisão ágil e contextualizada.

5.2.5 Infraestrutura do Sistema

A infraestrutura do sistema BloodHound foi construída adotando o conceito de Infraestrutura como Código (IaC), utilizando a ferramenta Terraform para provisionamento automatizado dos recursos na plataforma de nuvem Microsoft Azure. Essa abordagem permite que toda a infraestrutura seja versionada, reproduzível e auditável, favorecendo escalabilidade e controle de mudanças (HASHICORP, 2024).

O Terraform é uma ferramenta de código aberto que permite definir a infraestrutura por meio de arquivos de configuração declarativos, nos quais descreve-se o estado desejado dos recursos, como servidores, bancos de dados e serviços em nuvem. Após definidas, essas configurações são aplicadas automaticamente com comandos simples, reduzindo a chance de erros manuais e aumentando a eficiência da entrega de ambientes (HASHICORP, 2024).

Para o backend da aplicação, foi utilizado o Azure App Service, um serviço gerenciado de hospedagem de aplicações web. Esse serviço permite o *deploy* de aplicações sem a necessidade de gerenciamento direto da infraestrutura subjacente. O plano de serviço utilizado foi o F1 (Free Tier), uma opção gratuita recomendada para projetos de pequeno porte e testes iniciais, que oferece recursos básicos de execução, como suporte a HTTPS e variáveis de ambiente, mas com limitações de escalabilidade e tempo de atividade contínua (MICROSOFT, 2024a).

A aplicação backend foi configurada por meio de código Terraform, incluindo recursos como o *Resource Group*, o plano de serviço e o Web App em si. As variáveis de ambiente necessárias para a execução do sistema — como chaves da API da OpenAI, credenciais OAuth do GitHub, dados de conexão com o banco de dados e configurações de autenticação JWT — foram devidamente armazenadas na configuração do App Service, garantindo segurança e portabilidade da aplicação.

A seguir, a Figura 31 (Provisão de Recursos com Terraform) apresenta um trecho do código Terraform utilizado para provisionar os recursos do backend na Azure:

Figura 31 - Provisão de recursos com Terraform

```

1  resource "azurerms_resource_group" "rg" {
2      name      = var.resource_group_name
3      location = var.location
4  }
5
6  resource "azurerms_service_plan" "asp" {
7      name          = "${var.project_name}-asp"
8      location      = azurerms_resource_group.rg.location
9      resource_group_name = azurerms_resource_group.rg.name
10     os_type       = "Windows"
11     sku_name      = "F1"
12 }
13
14 resource "azurerms_windows_web_app" "app" {
15     name          = "${var.project_name}-app"
16     location      = azurerms_resource_group.rg.location
17     resource_group_name = azurerms_resource_group.rg.name
18     service_plan_id = azurerms_service_plan.asp.id
19
20     site_config {
21         always_on      = false # F1 não suporta AlwaysOn
22     }
23
24     app_settings = {
25         "ConnectionStrings__DefaultConnection" = var.db_connection_string
26
27         "OpenAI__Key"          = var.openai_key
28
29         "GitHub__ClientId"     = var.github_client_id
30         "GitHub__ClientSecret" = var.github_client_secret
31         "GitHub__RedirectUri"  = var.github_redirect_uri
32         "GitHub__AppId"       = var.github_app_id
33         "GitHub__AppSlug"     = var.github_app_slug
34         "GitHub__PrivateKey"  = var.github_private_key
35
36         "Jwt__SecretKey"       = var.jwt_secret_key
37         "Jwt__Issuer"         = var.jwt_issuer
38         "Jwt__Audience"      = var.jwt_audience
39
40         "Logging__LogLevel__Default" = "Debug"
41         "Logging__LogLevel__System"  = "Information"
42         "Logging__LogLevel__Microsoft" = "Information"

```

Fonte: Elaborado pelos autores (2025)

O *frontend* da aplicação, desenvolvido com Angular, foi gerado e implantado por meio do serviço Azure Static Web Apps, uma solução que oferece hospedagem de aplicações estáticas com integração simplificada de pipelines CI/CD e suporte nativo ao GitHub. Essa escolha se mostrou adequada ao tipo de aplicação proposta, por permitir alta performance com baixo custo operacional e facilidade de integração com o fluxo de desenvolvimento (MICROSOFT, 2024b).

Essa combinação entre *backend* gerenciado no Azure App Service e *frontend* hospedado no Static Web Apps busca garantir uma arquitetura leve, escalável e de fácil manutenção. Com o uso do Terraform, toda a infraestrutura pode ser replicada ou atualizada com segurança e agilidade, o que atende bem às necessidades de sistemas modernos baseados em nuvem.

5.3 TESTES E ANÁLISE DOS RESULTADOS

A validação de ferramentas que envolvem Inteligência Artificial requer experimentação controlada com conjuntos de dados representativos e análise rigorosa dos resultados obtidos. Neste tópico, são descritas as estratégias adotadas para testar o protótipo BloodHound, que tem como objetivo detectar e sugerir correções para *code smells* em *pull requests* de projetos de software.

A seção aborda desde a escolha do *dataset* utilizado, passando pela justificativa da linguagem de programação adotada nos testes, até a organização dos cenários experimentais e a interpretação dos dados gerados. Dessa forma, busca-se avaliar a eficácia da ferramenta frente a problemas reais e mensuráveis relacionados à qualidade de código.

5.3.1 Escolha da Linguagem para os Testes

A escolha da linguagem Java como foco dos testes da ferramenta BloodHound se fundamenta em sua ampla adoção tanto na indústria quanto na comunidade acadêmica. Sob essa perspectiva, a linguagem tem sido historicamente preferida em estudos voltados à análise de código, detecção de *code smells* e refatoração automática. Um levantamento conduzido por Palomba et al. (2015), por exemplo, aponta que a maioria dos *datasets* públicos disponíveis sobre *code smells* — como o Landfill e o iPlasma — são baseados em projetos escritos em Java.

Essa predominância ocorre porque o modelo orientado a objetos da linguagem facilita a modelagem de estruturas de código ricas, permitindo a extração de métricas como complexidade ciclomática, acoplamento e coesão, essenciais para a análise de qualidade de software (ARCELLI FONTANA et al., 2015).

5.3.2 Estruturação dos Testes

Com o objetivo de fundamentar os testes da ferramenta em fontes confiáveis e amplamente utilizadas na literatura, foram utilizados dois conjuntos de dados: o *Landfill Dataset* e o *Trusted Code Smells Dataset*.

O *Landfill* é uma base de dados aberta que reúne diversas instâncias de *code smells* identificadas em 20 projetos de software livre, como Apache Tomcat, Apache Cassandra, Eclipse, Google Guava, entre outros (PALOMBA et al., 2015).

Essa base fornece arquivos-fonte em linguagem Java extraídos de sistemas reais, sem anotações explícitas nos códigos, o que evita vieses de interpretação durante os testes. As ocorrências dos *code smells* estão documentadas separadamente por meio de scripts SQL, os quais associam os nomes dos arquivos ao tipo de *smell* identificado (PALOMBA et al., 2015).

Como fonte complementar, o *Trusted Code Smells Dataset* foi utilizado especificamente para testes adicionais com o *code smell* Long Method. Esse dataset apresenta instâncias confiáveis de *smells* validadas manualmente, permitindo verificar a capacidade da ferramenta em identificar casos reais e desafiadores (KACHANOV; MARKOV, 2023).

A análise combinada dos *datasets* revelou que os *code smells* mais recorrentes são: *Long Method*, *God/Large Class*, *Eager Test* e *Feature Envy*. Considerando que o escopo deste trabalho não inclui análise de *smells* relacionados a testes automatizados, como o *Eager Test*, foram selecionados os três tipos mais frequentes e relevantes do ponto de vista estrutural para a avaliação: *Long Method*, *God Class* e *Feature Envy*.

Para a realização dos testes, os arquivos-fonte presentes no *dataset* foram submetidos a *pull requests* em um repositório GitHub controlado. A ferramenta BloodHound foi integrada ao fluxo de trabalho desses *pull requests* por meio de *Webhooks*, realizando automaticamente a análise de cada arquivo de forma

contextual. O processo avaliou tanto a capacidade de detecção dos *smells* quanto a sugestão de correções estruturadas via IA.

5.3.2 Análise Quantitativa dos Testes

O protocolo experimental foi dividido em etapas. Primeiramente, os arquivos contendo *code smells* foram separados por categoria. Em seguida, foram enviados baseados na sua categoria por *pull requests* para garantir a rastreabilidade dos resultados. Para cada caso de teste, foi registrado se a ferramenta detectou o *code smell* esperado e, quando aplicável, se também foi sugerida uma ação de refatoração.

A Tabela 5 (Resultados dos Testes de Detecção e Sugestão de *Code Smells* com a Ferramenta BloodHound) apresenta um resumo quantitativo dos testes realizados para cada tipo de *code smell*. São exibidos o número total de arquivos avaliados, quantos desses foram corretamente identificados pela ferramenta como contendo *smells*, e, dentro desses, quantos receberam sugestões de correção por parte do modelo.

Tabela 5 - Resultados dos Testes de Detecção e Sugestão de Code Smells com a Ferramenta BloodHound

| Tipo de Code Smell | Casos Testados | Detectados pela IA | Taxa de Detecção (%) | Com Sugestão de Correção | Taxa de Correção (%) |
|--------------------------|----------------|--------------------|----------------------|--------------------------|----------------------|
| <i>Long Method</i> | 10 | 10 | 100% | 10 | 100% |
| <i>God/Large e Class</i> | 10 | 9 | 90% | 8 | 80% |
| <i>Feature Envy</i> | 10 | 10 | 100% | 6 | 60% |

Fonte: Elaborado pelos autores (2025)

Conforme observado, o tipo *Long Method* apresentou desempenho máximo em ambas as métricas, com 100% de detecção e 100% de sugestões de correção nos casos analisados. Esse resultado pode estar relacionado à natureza desse smell, que

costuma apresentar características estruturais facilmente reconhecíveis por modelos de linguagem treinados para interpretar blocos extensos de código.

No caso do *God/Large Class*, a ferramenta detectou 9 dos 10 casos avaliados, o que representa uma taxa de detecção de 90%. Desses, 8 foram acompanhados de sugestões de correção, totalizando uma taxa de 80%. A leve queda na taxa de detecção pode ser explicada pela subjetividade envolvida na definição do limiar em que uma classe é considerada excessivamente grande ou com responsabilidades múltiplas. Essa subjetividade também pode ter impacto na geração de sugestões específicas, dada a necessidade de interpretação semântica mais refinada por parte do modelo.

Já o *Feature Envy* obteve taxa de detecção de 100%, com 6 dos 10 casos recebendo sugestões de refatoração. Apesar da identificação completa dos casos, a taxa de correção foi inferior (60%). Isso sugere que, embora o modelo seja capaz de reconhecer indícios de acesso excessivo a dados de outras classes — principal característica do *smell* —, a geração de sugestões contextualmente adequadas pode ser mais desafiadora, exigindo maior compreensão da lógica de relacionamento entre objetos.

De modo geral, os resultados demonstram que a ferramenta apresenta alta efetividade na tarefa de identificação de *code smells*, com taxas de detecção iguais ou superiores a 90% para todos os tipos analisados. A taxa de geração de sugestões de correção foi mais variável, indicando que, embora a IA seja eficaz na detecção, ainda existem limitações na proposição de melhorias específicas em cenários mais complexos ou ambíguos.

Esses dados reforçam o potencial de aplicação da ferramenta como um apoio automatizado ao processo de revisão de código, especialmente na triagem inicial de *code smells*. Contudo, os resultados também indicam que a supervisão humana continua sendo um fator relevante na etapa de refatoração, especialmente nos casos em que a estrutura do código apresenta maior complexidade ou ambiguidade semântica.

5.3.2 Análise Qualitativa das Sugestões

Para além dos resultados quantitativos obtidos nos testes, é importante considerar também a qualidade e a utilidade prática das sugestões geradas pela

ferramenta BloodHound. A seguir, são apresentados dois exemplos qualitativos representativos: um relacionado ao *code smell* do tipo *God Class* e outro ao *Long Method*. Os exemplos analisam a profundidade da sugestão oferecida e sua viabilidade prática para auxiliar no processo de refatoração.

O primeiro caso envolve a detecção de uma *God Class* — um *code smell* caracterizado pela concentração excessiva de responsabilidades em uma única classe, contrariando o princípio da responsabilidade única. Nesse caso em específico, embora a ferramenta tenha corretamente identificado o *smell*, a sugestão gerada apresentou limitações. A recomendação foi textual e genérica, indicando apenas que a classe poderia ser dividida em múltiplas classes especializadas, conforme ilustrado na Figura 32 (Trecho da refatoração do *Code Smell God Class*) abaixo:

Figura 32 - Trecho da refatoração do *Code Smell God Class*

God Class/CdmaDataConnectionTracker.java

... @@ -0,0 +1,975 @@

1 + /*

bot-bloodhound (bot) 2 days ago

Code Smell Detected: Large Class / God Class

Sugestão de melhoria: A classe `CdmaDataConnectionTracker` parece estar assumindo muitas responsabilidades, o que a torna difícil de ler, testar e manter. Considere dividir essa classe em várias classes menores, cada uma responsável por uma parte específica da lógica de conexão de dados CDMA. Isso pode incluir a separação das responsabilidades de gerenciamento de estado, manipulação de eventos e configuração de conexão.

Suggested change

```

1 - /*
1 + // Exemplo de refatoração:
2 + // Crie classes separadas para gerenciar diferentes responsabilidades, como:
3 + // - CdmaConnectionManager
4 + // - CdmaEventHandler
5 + // - CdmaStateHandler
6 + // Cada uma dessas classes pode ser responsável por uma parte específica da lógica atual.
```

Commit suggestion

Fonte: Elaborado pelos autores (2025)

Apesar da orientação conceitualmente coerente com as boas práticas de design orientado a objetos, a sugestão não incluiu exemplos concretos de como essa divisão poderia ser aplicada no código-fonte original. A ausência de blocos de código

dificultou a aplicabilidade direta da recomendação, tornando-a menos útil no contexto prático da revisão automática. Esse tipo de sugestão evidencia uma limitação no modelo em traduzir a detecção conceitual em uma proposta de ação tangível.

Por outro lado, o segundo exemplo refere-se a um caso de *Long Method*, um *code smell* que ocorre quando um único método realiza múltiplas operações, tornando sua leitura e manutenção mais difíceis. Nesse teste, a ferramenta não apenas identificou corretamente o *smell* bem como propôs uma refatoração detalhada e contextualizada. A Figura 33 (Trecho da refatoração do *Code Smell Long Method*) apresenta o trecho de código original analisado e a sugestão de refatoração gerada:

Figura 33 - Trecho da refatoração do Code Smell Long Method



bot-bloodhound (bot) yesterday

...

Code Smell Detected: Long Method

💡 Improvement suggestion: O método `onCreate` é muito longo e mistura diferentes responsabilidades, como inicialização de fragmentos e configuração de layout. Considere dividir o método em métodos menores e mais específicos para melhorar a legibilidade e a manutenção.

```
Suggested change
1 - /*
1 + @Override
2 + protected void onCreate(Bundle savedInstanceState) {
3 +     super.onCreate(savedInstanceState);
4 +     initializeFragments(savedInstanceState);
5 +     setupLayoutInflater();
6 + }
7 +
8 + private void initializeFragments(Bundle savedInstanceState) {
9 +     mFragments.attachActivity(this, mContainer, null);
10 +     NonConfigurationInstances nc = (NonConfigurationInstances) getLastNonConfigurationInstance();
11 +     if (nc != null) {
12 +         mAllLoaderManagers = nc.loaders;
13 +     }
14 +     if (savedInstanceState != null) {
15 +         Parcelable p = savedInstanceState.getParcelable(FRAGMENTS_TAG);
16 +         mFragments.restoreAllState(p, nc != null ? nc.fragments : null);
17 +     }
18 +     mFragments.dispatchCreate();
19 + }
20 +
21 + private void setupLayoutInflater() {
22 +     if (getLayoutInflater().getFactory() == null) {
23 +         getLayoutInflater().setFactory(this);
24 +     }
25 + }
```

Fonte: Elaborado pelos autores (2025)

A proposta dividiu o método `onCreate()` em dois métodos auxiliares: `initializeFragments()` e `setupLayoutInflater()`, separando responsabilidades distintas — configuração de layout e inicialização de componentes — em unidades menores e mais coesas. Essa abordagem reflete uma aplicação prática dos princípios de código limpo e favorece a manutenção futura do sistema, além de reforçar a utilidade do modelo como ferramenta de apoio à engenharia de software.

Esses dois casos ilustram diferentes níveis de profundidade nas sugestões oferecidas pela IA. Enquanto o primeiro exemplo apresentou uma sugestão conceitual, sem detalhamento técnico, o segundo demonstrou a capacidade do modelo em aplicar conhecimento contextualizado e gerar alternativas estruturadas para refatoração. Essa variação está alinhada com a literatura, que reconhece que *code smells* mais estruturais, como *Long Method*, são mais facilmente detectáveis e refatoráveis por modelos automatizados, enquanto *smells* mais semânticos ou de design exigem interpretação mais complexa do contexto (PEREIRA DOS REIS et al., 2022).

6 CONCLUSÃO

Durante o desenvolvimento deste trabalho, constatou-se que, durante o processo de revisão de código, podem surgir problemas recorrentes relacionados à má estruturação do código, conhecidos como *code smells*. Esses problemas, se não identificados e corrigidos, podem comprometer a qualidade do software e aumentar o débito técnico ao longo do tempo. Com o intuito de minimizar esses impactos e promover maior qualidade no desenvolvimento, propôs-se a criação de um protótipo baseado em Inteligência Artificial Generativa, o BloodHound *Code Reviewer*, capaz de detectar automaticamente *code smells* e sugerir melhorias durante o processo de *code review*.

Nesse sentido, o propósito central da pesquisa consistiu em investigar o uso de IA generativa, por meio de modelos como o GPT-4, para auxiliar na identificação e correção de *code smells* de forma automatizada. Com base na revisão teórica e na análise prática dos resultados obtidos, foi possível propor uma solução concreta para esse desafio. Verificou-se que os objetivos traçados foram cumpridos com êxito, reforçando a viabilidade e relevância da proposta no cenário atual da engenharia de software.

Em relação ao primeiro objetivo geral, buscou-se estudar o processo de revisão de código, compreendendo suas etapas, desafios e importância no ciclo de desenvolvimento. Por meio de uma investigação bibliográfica aprofundada, foi possível reconhecer as limitações das abordagens manuais, o que sustentou a necessidade de ferramentas automatizadas como o protótipo proposto.

O segundo objetivo geral consistiu em pesquisar os *code smells*, seus principais tipos e os impactos negativos que causam na manutenção e legibilidade do código. A análise realizada permitiu identificar que essas falhas são comuns no cotidiano de desenvolvedores e frequentemente negligenciadas, tornando clara a relevância de uma ferramenta que atue na sua detecção precoce.

O terceiro objetivo geral visou investigar o uso da Inteligência Artificial Generativa na detecção de *code smells*. Essa etapa foi alcançada com êxito, ao se analisar as capacidades do GPT-4 em interpretar trechos de código-fonte e sugerir refatorações baseadas em boas práticas. Essa abordagem demonstrou ser funcional e promissora dentro do contexto proposto.

Por fim, o desenvolvimento do protótipo *BloodHound Code Reviewer* representou o último estágio da pesquisa. O sistema foi projetado para se integrar a plataformas como GitHub e GitLab, analisando automaticamente os *pull requests* e emitindo sugestões técnicas. Concluiu-se que a necessidade identificada pela pesquisa foi devidamente demonstrada, uma vez que a maioria dos participantes indicou ser favorável a uma ferramenta de automatização do processo de *code review*, por meio da detecção e sugestão de correção de *code smells*.

Durante o estudo, também foi possível identificar limitações em ferramentas tradicionais de análise estática, como o SonarQube. Embora amplamente utilizadas, tais ferramentas operam com regras fixas e genéricas, o que restringe sua capacidade de compreender o contexto do sistema e a intenção do desenvolvedor. Em muitos casos, não conseguem identificar oportunidades de generalização entre funções similares nem oferecem sugestões concretas de melhoria, limitando-se a apontar os problemas detectados (ZHANG et al., 2024). Esse cenário reforça ainda mais o valor das abordagens baseadas em IA, que trazem uma camada adicional de interpretação e contexto à análise de código.

Modelos baseados em IA, ao contrário da análise estática tradicional, conseguem realizar uma análise dinâmica e contextualizada. Por exemplo, são capazes de identificar que uma classe aparentemente pequena está concentrando diversas responsabilidades — como acesso ao banco de dados, lógica de negócio e controle da interface — caracterizando-se como uma *God Class*, mesmo sem ultrapassar limites rígidos de linhas ou complexidade. Situações como essa são difíceis de capturar por regras estáticas, mas podem ser percebidas por sistemas inteligentes treinados em grandes volumes de código (LEWOWSKI; MADEYSKI, 2021).

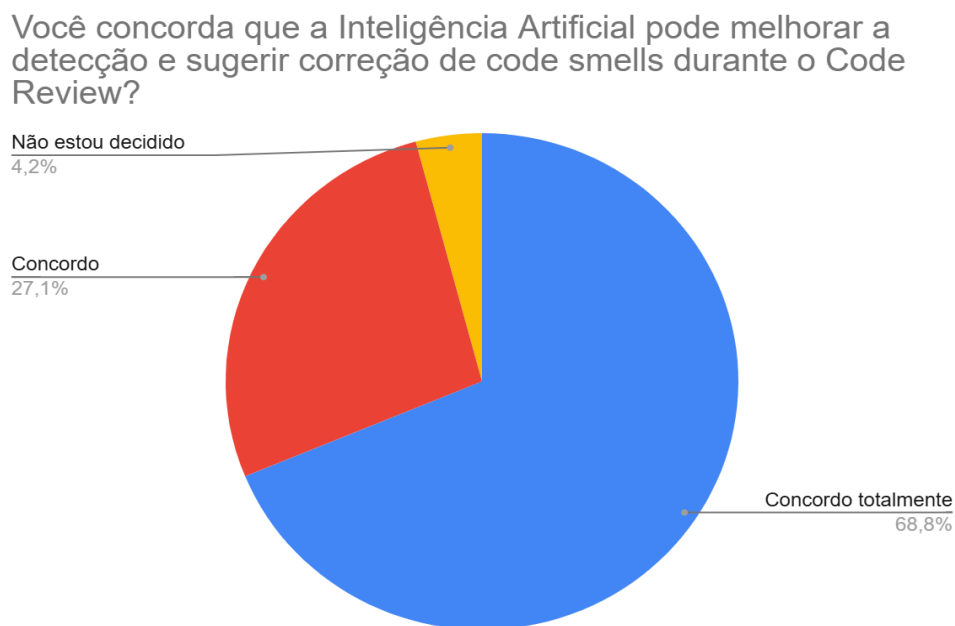
Entretanto, esses modelos ainda podem gerar falsos positivos, sugerir refatorações inadequadas e requerem ajustes frequentes, além de uma base robusta de dados. Dessa forma, a recomendação é utilizá-los como apoio ao trabalho humano, combinando-os com revisões manuais e ferramentas tradicionais para potencializar a qualidade do software (CIHAN et al., 2024).

Assim, a detecção automatizada de *code smells* por meio da inteligência artificial representa uma abordagem híbrida, que não substitui o olhar crítico do desenvolvedor, mas o complementa, fortalecendo as práticas de engenharia de

software e contribuindo para a construção de sistemas mais limpos, organizados e sustentáveis (CIHAN et al., 2024).

Conforme ilustrado na Figura 34 (Aceitação da IA na detecção e sugestão de correção de code smells durante o *Code Review*), mais de 95% dos respondentes afirmaram concordar (27,1%) ou concordar totalmente (68,8%) que a IA pode melhorar a detecção e sugerir correções para *code smells* nesse contexto.

Figura 34 - Gráfico de Pesquisa: IA no processo de *Code review*



Fonte: Elaborado pelos autores (2025).

Esse resultado reforça a viabilidade e a relevância prática da proposta do sistema BloodHound, validando sua aderência ao ambiente real de desenvolvimento de software e ao interesse da comunidade técnica na automatização inteligente desse processo.

Como recomendações para trabalhos futuros, sugere-se:

1. Ampliar a base de treinamento com repositórios públicos rotulados;

2. Estender o suporte da ferramenta para múltiplas linguagens de programação;
3. Incorporar métricas quantitativas de qualidade, como complexidade ciclomática e cobertura de testes;
4. Realizar experimentos controlados para avaliar o impacto da ferramenta em ambientes de desenvolvimento reais.

REFERÊNCIAS

- AL-FRAIHAT, D. *et al.* Detecting and resolving feature envy through automated machine learning and move method refactoring. **International Journal of Electrical & Computer Engineering (2088-8708)**, [s. l.], v. 14, n. 2, p. 2330–2343, 2024. DOI 10.11591/ijece.v14i2.pp2330-2343. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=8a0f6109-764e-3e00-a592-c4ef94c29245>. Acesso em: 8 maio. 2025.
- ALLAMANIS, M. *et al.* A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, [s. l.], v. 51, n. 4, p. 1–37, 2019. DOI 10.1145/3212695. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=eafa5e72-9aa1-3a3a-a729-5565a580a730>. Acesso em: 24 abr. 2025.
- ALLAMANIS, M. The Adverse Effects of Code Duplication in Machine Learning Models of Code. [s. l.], 2018. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=0c8f7c6d-4dac-356a-b60a-f16ed0ba17d9>. Acesso em: 8 maio. 2025.
- AMATRIAIN, X. Prompt Design and Engineering: Introduction and Advanced Methods. [s. l.], 2024. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=1a1ff151-90b3-3488-bd1f-db010f306460>. Acesso em: 24 jun. 2025.
- ANWAR, A. *et al.* Understanding the Hidden Cost of Software Vulnerabilities: Measurements and Predictions. **United States, North America: Springer International Publishing, 2018**. DOI 10.1007/978-3-030-01701-9_21. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=9a26c1b5-cf2c-34c3-a762-2cb29bfb9d73>. Acesso em: 29 Mar. 2025.
- AHMED, S. (Saad). Integrating AI-Driven Automated Code Review in Agile Development: Benefits, Challenges, and Best Practices. **International Journal of Advanced Engineering, Management and Science**, [s. l.], 2025. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=8b1eb77b-ee12-33fc-8f46-8e3849441d46>. Acesso em: 25 abr. 2025.
- AJIT SINGH. Agile & Scrum. Babelcube Inc, 2019. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=38593f78-34a0-3c02-9993-a1f943a0052c>. Acesso em: 13 abr. 2025.
- BACCHELLI, A.; BIRD, C. Expectations, Outcomes, and Challenges of Modern Code Review. *ICSE: International Conference on Software Engineering*, [s. l.], p. 712–721, 2013. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=51105490-ba34-3e67-92e9-f8d87e891ef9>. Acesso em: 6 mar. 2025.
- BAVOTA, G. *et al.* When Does a Refactoring Induce Bugs? An Empirical Study. 2012 IEEE 12th International Working Conference on Source Code Analysis & Manipulation, [s. l.], p. 104–113, 2012. DOI 10.1109/SCAM.2012.20. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=09df0b19-617f-38d2-959f-5798863d4fcf>. Acesso em: 4 abr. 2025.

BAUMGARTNER, N.; ADLEH, F.; PULVERMÜLLER, E. Live Code Smell Detection of Data Clumps in an Integrated Development Environment. [s. l.], 2023. DOI 10.5220/0011727500003464. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=7cede749-4ae8-3e01-9424-63693e8b5261>. Acesso em: 8 maio. 2025.

BECK, K.; ANDRES, C. Extreme Programming Explained: Embrace Change. [s. l.], 2005. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=f36223c3-d066-3f53-860f-d87be2c06dab>. Acesso em: 16 maio. 2025.

BECK, K. et al. **Manifesto for Agile Software Development, 2001**. Disponível em: <https://agilemanifesto.org>. Acesso em: 21 abr. 2025.

BLACK, P.E. (2011). Counting bugs is harder than you think. **IEEE 11th International Working Conference on Source Code Analysis and Manipulation**. Disponível em: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=908871. Acesso em 14 abr. 2025.

BOEHM, B. (1981). **Software Engineering Economics**. Prentice Hall.

BROWN, W. H.; MALVEAU, R. C.; MCCORMICK III, H. W.; MOWBRAY, T. J. (1998). **AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis**. John Wiley & Sons.

CAIRO, A. S.; CARNEIRO, G. de F.; MONTEIRO, M. P. The impact of code smells on software bugs: a systematic literature review. [s. l.], 2018. DOI 10.3390/info9110273. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=cf9f299a-465f-3c76-b312-244dca017764>. Acesso em: 28 Mar. 2025.

CHAZETTE, L.; BRUNOTTE, W.; SPEITH, T. Explainable software systems: from requirements analysis to system evaluation. **Requirements Engineering**, [s. l.], v. 27, n. 4, p. 457–487, 2022. DOI 10.1007/s00766-022-00393-5. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=c003c8a0-49f5-3395-accd-01c854942812>. Acesso em: 8 maio. 2025.

CIHAN, U. et al. Automated Code Review In Practice. [s. l.], 2024. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=fb2c6c0b-af41-3ed5-9fe4-948a3f8a45cf>. Acesso em: 25 abr. 2025.

CRESWELL, J. W. (2014). **Research Design: Qualitative, Quantitative, and Mixed Methods Approaches, 5th Edition**. Disponível em : <https://journals.econsciences.com/index.php/JSAS/article/view/1313/1341> Acesso em: 24 abr. 2025.

DIJKSTRA, E. W. Go To Statement Considered Harmful. United States, North America: ACM, 2022. DOI 10.1145/3544585.3544604. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=42099365-b3f6-360d-9c4f-5ec2d07b4a86>. Acesso em: 18 abr. 2025.

DR. D. I. DE SILVA et al. The Effectiveness of Code Reviews on Improving Software Quality: An Empirical Study. **International Journal of Recent Technology and**

Engineering (IJRTE), 12(2), 1-10, (2023-07-30, [s. l.], 2023. DOI 10.35940/ijrte.B7666.0712223. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=b3c1cdcc-808f-3664-a359-56695a172a59>. Acesso em: 16 maio. 2025.

DU, X. *et al.* Evaluating Large Language Models in Class-Level Code Generation. Proceedings of the **IEEE/ACM** 46th International Conference on Software Engineering ; page 1-13, [s. l.], 2024. DOI 10.1145/3597503.3639219. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=71236940-4ac3-35e8-8288-ec2e67c85a54>. Acesso em: 1 maio. 2025.

ESTOMII EDWARD; ALLY S. NYAMAWE; NOE ELISA. On the Impact of Refactorings on Software Attack Surface. **IEEE Access**, [s. l.], v. 12, p. 128570–128584, 2024. DOI 10.1109/ACCESS.2024.3404058. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=4c4b7642-8efd-3d31-94b6-a1432d723171>. Acesso em: 29 mar. 2025.

FAGAN, M. E. Design and code inspections to reduce errors in program development. **IBM Systems Journal**, [s. l.], v. 38, n. 2–3, p. 258–287, 1999. DOI 10.1147/sj.382.0258. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=ba4f25b0-df39-3475-9a60-507fa7afe096>. Acesso em: 4 abr. 2025.

FAN, G. *et al.* Static Code Analysis in the AI Era: An In-depth Exploration of the Concept, Function, and Potential of Intelligent Code Analysis Agents. [s. l.], 2023. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=421d36ab-8a15-3c70-ac4a-78c3f274d933>. Acesso em: 26 abr. 2025.

ARCELLI FONTANA, F. *et al.* On experimenting refactoring tools to remove code smells. [S. l.], 2015. DOI: 10.1145/2764979.2764986. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=1c6c5375-ff42-3272-ae45-45f30ba337db>. Acesso em: 22 jun. 2025.

FOWLER, M. (1999). **Refactoring: Improving the Design of Existing Code.** Addison-Wesley.

FOWLER, Martin. **Refactoring: improving the design of existing code.** Boston: Addison-Wesley, 1999.

FOWLER, M. (2019). **Refactoring: Improving the Design of Existing Code (2nd ed.)**. Addison-Wesley.

GITHUB DOCS. Pull request review process. **GitHub**, 2023. Disponível em: <https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches>. Acesso em: 27 abr. 2025.

GOEL, A. Looking Back, Looking Ahead: Symbolic versus Connectionist AI. **AI Magazine** ; volume 42, issue 4, page 83-85 ; ISSN 2371-9621 0738-4602, [s. l.], 2022. DOI 10.1609/aimag.v42i4.15111. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=d117be92-7fa6-3737-9246-a11bff1084bd>. Acesso em: 24 abr. 2025.

GUO, Y.; SPÍNOLA, R.; SEAMAN, C. Exploring the costs of technical debt management - a case study. **Empirical Software Engineering**, [s. l.], v. 21, n. 1, p. 159–182, 2016. DOI 10.1007/s10664-014-9351-7. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=bb7bbb9a-4429-343f-92ad-a010bcb2ab48>. Acesso em: 21 abr. 2025.

HASHICORP. Terraform: Infrastructure as Code. **HashiCorp**, 2024. Disponível em: <https://developer.hashicorp.com/terraform/docs>. Acesso em: 10 jun. 2025.

HIGHSMITH, J. **Agile Project Management: Creating Innovative Products**. 2. ed. Addison-Wesley, 2009.

HOLVITIE, J. et al. Technical debt and agile software development practices and processes: An industry practitioner survey. [s. l.], 2021. DOI 10.1016/j.infsof.2017.11.015. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=79ed1ed8-c795-3456-96ac-b4e636cec02f>. Acesso em: 4 abr. 2025.

HOMBERGS, T.; HOMBERGS, T. Get Your Hands Dirty on Clean Architecture. [s. l.], 2019. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=3d7eec8d-3697-301b-9e1d-915e0c6a470e>. Acesso em: 3 maio. 2025.

HUYNH, N.; LIN, B. A Survey On Large Language Models For Code Generation. [s. l.], 2025. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=de7a4548-ab79-3e85-aa40-d08ba6277e92>. Acesso em: 27 abr. 2025.

IBM Systems Sciences Institute. *Defect Prevention: Reducing Costs and Enhancing Quality*. ISIXSIGMA. Disponível em: <https://www.isixsigma.com/software/defect-prevention-reducing-costs-and-enhancing-quality/>. Acesso em: 14 abr. 2025.

JURECZKO, M.; KAJDA, Ł.; GÓRECKI, P. Code review effectiveness: an empirical study on selected factors influence. **IET Software (Wiley-Blackwell)**, [s. l.], v. 14, n. 7, p. 794–805, 2020. DOI 10.1049/iet-sen.2020.0134. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=d97da159-6250-3a19-ae34-d3c5fcb0ba19>. Acesso em: 21 abr. 2025.

KACHANOV VLADIMIR; MARKOV SERGEY. Trusted Code Smells Dataset. [s. l.], 2023. DOI 10.5281/zenodo.7612725. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=8a14ded8-42f5-3acd-b5ff-05c3f09c1394>. Acesso em: 4 jun. 2025.

LACERDA, G. et al. Code Smells and Refactoring: A Tertiary Systematic Review of Challenges and Observations. [s. l.], 2020. DOI 10.1016/j.jss.2020.110610. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=62c59f42-2320-3726-b938-276018ed880c>. Acesso em: 26 abr. 2025.

LAKATOS, E. M.; MARCONI, M. de A. **Fundamentos de metodologia científica / Eva Maria Lakatos, Marina de Andrade Marconi**. [s. l.]: Atlas, 2017. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=ac2aef72-9c22-3e7f-9d20-41b499042e59>. Acesso em: 25 abr. 2025.

LEFEVER, J. et al. On the Lack of Consensus Among Technical Debt Detection Tools. [s. l.], 2021. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=ca898436-1758-3a06-820c-ed9a74afea51>. Acesso em: 4 abr. 2025.

LEWOWSKI, T.; MADEYSKI, L. **Code Smells Detection Using Artificial Intelligence Techniques: A Business-Driven Systematic Review**. United States, North America: Springer International Publishing, 2021. DOI 10.1007/978-3-030-77916-0_12. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=9bda45d1-3f77-354a-9376-eea637ce68d2>. Acesso em: 9 maio. 2025.

LI, Z. et al. Technical Debt Management in OSS Projects: An Empirical Study on GitHub. [s. l.], 2022. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=25fd0e73-8c44-3ade-a8a8-a58d74ea74f3>. Acesso em: 4 abr. 2025.

LU, J. et al. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. [s. l.], 2023. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=65f44b96-d5ee-3820-a174-63098201e790>. Acesso em: 14 abr. 2025.

MACIA, I. et al. Enhancing the Detection of Code Anomalies with Architecture-Sensitive Strategies. **2013 17th European Conference on Software Maintenance & Reengineering**, [s. l.], p. 177–186, 2013. DOI 10.1109/CSMR.2013.27. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=add79618-e25c-31d8-81f1-4dbd7efa20ac>. Acesso em: 26 abr. 2025.

MARTINS, J. et al. How do Code Smell Co-occurrences Removal Impact Internal Quality Attributes? A Developers' Perspective. **Brazilian Symposium on Software Engineering** ; page 54-63, [s. l.], 2021. DOI 10.1145/3474624.3474642. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=c366c2be-91ff-3634-8234-874c0e673bbc>. Acesso em: 8 maio. 2025.

MARTINS, J. S. Investigando o Impacto das Coocorrências de Code Smells nos Atributos Internos de Qualidade. Dissertação de Mestrado – Universidade Federal do Ceará, Campus de Quixadá, 2021. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=e5b70160-9d66-3d5b-bf6a-1368b4f1c4c4>. Acesso em: 28 Mar. 2025.

MARTIN, R. C. The clean coder: a code of conduct for professional programmers. [s. l.], 2011. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=938c30ca-6323-390e-b53d-a369a69ca311>. Acesso em: 3 maio. 2025.

MCINTOSH, Shane et al. An empirical study of the impact of modern code review practices on software quality. [S. l.], 2015. DOI: [10.1007/s10664-015-9381-9](https://doi.org/10.1007/s10664-015-9381-9). Disponível em: <https://doi.org/10.1007/s10664-015-9381-9>. Acesso em: 29 mar. 2025.

MIN ZHANG; HALL, T.; BADD00, N. Code Bad Smells: a review of current knowledge. *Journal of Software Maintenance & Evolution: Research & Practice*, [s. l.], v. 23, n. 3, p. 179–202, 2011. DOI 10.1002/smr.521. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=933f5052-f0a9-3788-a783-fd92f2be4f0b>. Acesso em: 5 abr. 2025.

MICROSOFT. What is Azure App Service?. **Microsoft, 2024**. Disponível em: <https://learn.microsoft.com/en-us/azure/app-service/overview>. Acesso em: 10 jun. 2025.

MICROSOFT. Overview of Azure Static Web Apps. **Microsoft, 2024**. Disponível em: <https://learn.microsoft.com/en-us/azure/static-web-apps/overview>. Acesso em: 10 jun. 2025.

MURILLO, M. I. *et al.* Identification and Management of Technical Debt: A Systematic Mapping Study Update. **Journal of Software Engineering Research & Development**, [s. l.], v. 11, n. 1, p. 1–20, 2023. DOI 10.5753/jserd.2023.2671. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=80f700fe-6c57-3a3f-bf73-5aa6cc698e4f>. Acesso em: 21 abr. 2025.

NILSSON, A.; WILSON, T. L. Reflections on Barry W. Boehm’s “A spiral model of software development and enhancement”. [s. l.], 2012. DOI 10.1108.17538371211269031. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=3d9328d7-09fa-3996-8718-f32bb78824e4>. Acesso em: 2 maio. 2025.

O’CONNOR, D. L. Code Review Practices - Guidelines and Benefits: Investigating code review practices, guidelines, and the benefits of peer code reviews in improving code quality and knowledge sharing. **Journal of Artificial Intelligence Research**, v. 4, n. 1, p. 128–136, maio 2024. Disponível em: <https://nucleuscorp.org/JAIR/article/view/190>. Acesso em: 25 abr. 2025.

OPENAI et al. GPT-4 Technical Report. [s. l.], 2023. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=6391f319-5fe2-3a50-8c78-dd4c92821989>. Acesso em: 1 maio. 2025.

OPENAI et al. GPT-4o System Card. [s. l.], 2024. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=ba422d6b-706d-3798-9a80-8b4f356cf51b>. Acesso em: 1 maio. 2025.

PALOMBA, F. *et al.* Landfill: An Open Dataset of Code Smells with Public Evaluation. **2015 IEEE/ACM 12th Working Conference on Mining Software Repositories**, [s. l.], p. 482–485, 2015. DOI 10.1109/MSR.2015.69. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=5f8f3e36-4815-3dd0-8bc2-41cf31c0c7fd>. Acesso em: 8 jun. 2025.

PALOMBA, F. *et al.* Mining Version Histories for Detecting Code Smells. **IEEE Transactions on Software Engineering**, [s. l.], v. 41, n. 5, p. 462–489, 2015. DOI 10.1109/TSE.2014.2372760. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=69f41bfb-1af9-3e7f-b3ca-291bb8027603>. Acesso em: 26 abr. 2025.

PEREIRA DOS REIS, J. *et al.* Code Smells Detection and Visualization: A Systematic Literature Review. *Archives of Computational Methods in Engineering*, [s. l.], v. 29, n. 1, p. 47–94, 2022. DOI 10.1007/s11831-021-09566-x. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=1c21832e-cac3-3f17-8f12-db44d2c0d51b>. Acesso em: 11 abr. 2025.

PETER C. RIGBY; CHRISTIAN BIRD. Convergent Contemporary Software Peer Review Practices. <http://www.cabird.com/papers/rigby2013convergent.pdf>, [s. l.], [s. d.]. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=c68cc183-69ce-3ce6-96a3-426ab7feca8f>. Acesso em: 14 Mar. 2025.

PETER C. RIGBY; MARGARET-ANNE STOREY. Understanding broadcast based peer review on open source software projects. <http://flosshub.org/sites/flosshub.org/files/Rigby2011ICSE.pdf>, [s. l.], 2011. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=4e1b0931-fdd1-31a0-889b-a47fe5ae292f>. Acesso em: 21 abr. 2025.

RASHEED, Z. *et al.* AI-powered Code Review with LLMs: Early Results. [s. l.], 2024. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=2b816b41-7157-3ac7-91c9-9c219eadd2f9>. Acesso em: 14 abr. 2025.

ROYCE, W. W. Managing the development of large software systems: concepts and techniques. **ICSE: International Conference on Software Engineering**, [s. l.], p. 328–338, 1987. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=e3a13357-42a6-31a4-a3c4-69e6686bbc8f>. Acesso em: 2 maio. 2025.

SACA, M. A. Refactoring improving the design of existing code. **2017 IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII)**, [s. l.], 2017. DOI 10.1109/concapan.2017.8278488. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=ba7e1a71-d825-3885-a543-c20074430277>. Acesso em: 26 abr. 2025.

SAHOO, P. *et al.* A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications. [s. l.], 2024. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=dde12460-a157-3a20-99a7-da4a7f54b3bf>. Acesso em: 24 jun. 2025.

SCHULHOFF, S. *et al.* The Prompt Report: A Systematic Survey of Prompt Engineering Techniques .. [s. l.], 2024. DOI 10.48550/arxiv.2406.06608. Disponível em: <https://research.ebsco.com/linkprocessor/plink?id=9f19bde1-1176-3825-b325-e264733d1ecf>. Acesso em: 24 jun. 2025.

SCHWABER, K.; SUTHERLAND, J. (2020). *The Scrum Guide: The Definitive Guide to Scrum: The Rules of the Game*. Disponível em: <https://billlewisstraining.com/wp-content/uploads/2017/02/PMP-Agile-Study-Materials.pdf>. Acesso em: 13 abr. 2025

SOMMERVILLE, I. (2015). **Software Engineering (10th ed.)**. Addison-Wesley.

SONARSOURCE. **Static Code Analysis Using SonarQube: A Step-by-Step Guide. 2018**. Disponível em: <https://www.sonarsource.com/learn/static-code-analysis-using-sonarqube/>. Acesso em: 09 maio 2025.

STEFANOVIĆ, D. *et al.* Identification of strategies over tools for static code analysis. **IOP Conference Series: Materials Science and Engineering**; volume **1163**, issue **1**, page **012012**; ISSN **1757-8981 1757-899X**, [s. l.], 2021. DOI 10.1088/1757-899x/1163/1/012012. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=b219748c-9868-3fc4-8d32-2a3fbb8fea4f>. Acesso em: 21 abr. 2025.

KOKOL, P.; KOKOL, M.; ZAGORANSKI, S. Code smells: A Synthetic Narrative Review. [s. l.], 2021. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=3b7478d5-13cb-351a-840f-374d27a53366>. Acesso em: 5 abr. 2025.

STUTI TANDON; VIJAY KUMAR; V. B. SINGH. Study of Code Smells: A Review and Research Agenda. *International Journal of Mathematical, Engineering and Management Sciences*, [s. l.], v. 9, n. 3, p. 472–498, 2024. DOI 10.33889/IJMEMS.2024.9.3.025. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=41a26749-3117-33d4-891b-cf0e2e8fb0ac>. Acesso em: 18 abr. 2025.

TURHAN, B. On the dataset shift problem in software engineering prediction models. **Empirical Software Engineering**, [s. l.], v. 17, n. 1/2, p. 62–74, 2012. DOI 10.1007/s10664-011-9182-8. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=7724f2c3-f38b-32bc-86db-d232d1aaf97c>. Acesso em: 23 abr. 2025.

TURKISTANI, B.; LIU, Y. Reducing the Large Class Code Smell by Applying Design Patterns. **2019 IEEE International Conference on Electro Information Technology (EIT)**, [s. l.], 2019. DOI 10.1109/eit.2019.8833851. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=7f6fa777-cbfb-305a-bc92-d50d4e030303>. Acesso em: 8 maio. 2025.

VANHANEN, J. *et al.* Bad Smells in Software - a Taxonomy and an Empirical Study. [s. l.], 2003. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=76df26dd-d4c9-3b80-a2c5-fb50f3b881ee>. Acesso em: 5 abr. 2025.

VASCONCELOS, Audrey Emmely Rodrigues. *Explorando métricas de código para a detecção de Long Envious Methods*. 2023. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Instituto de Computação, Universidade Federal de Alagoas, Maceió, 2023. Disponível em: <http://www.repositorio.ufal.br/jspui/handle/123456789/10684>. Acesso em: 08 maio 2025.

VERGARA, S. C. **Métodos de pesquisa em administração / Sylvia Constant Vergara**. [s. l.]: Atlas, 2015. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=7b4c7288-735d-316f-859b-0a37c2040cf0>. Acesso em: 25 abr. 2025.

WANG, X. *et al.* Self-Consistency Improves Chain of Thought Reasoning in Language Models. [s. l.], 2022. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=3e8aa491-9d0a-34b9-91f4-2fe2f7a572b2>. Acesso em: 24 jun. 2025.

WEI, J. *et al.* Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. [s. l.], 2022. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=4ffda9e6-0d25-3f71-b0d0-3b67739db99d>. Acesso em: 24 jun. 2025.

WHITE, J. *et al.* A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. [s. l.], 2023. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=ab93a981-ae15-3496-85e8-02e4b9100c66>. Acesso em: 24 jun. 2025.

XIONG, H. *et al.* Converging Paradigms: The Synergy of Symbolic and Connectionist AI in LLM-Empowered Autonomous Agents. [s. l.], 2024. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=c4649c15-a07a-3629-afed-b228569a6ebe>. Acesso em: 24 abr. 2025.

YADAV, P. S. *et al.* Machine Learning-Based Methods for Code Smell Detection: A Survey. **Applied Sciences (2076-3417)**, [s. l.], v. 14, n. 14, p. 6149, 2024. DOI 10.3390/app14146149. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=886af53d-5162-37f6-9d86-5b239acad806>. Acesso em: 24 abr. 2025.

YAO, S. *et al.* ReAct: Synergizing Reasoning and Acting in Language Models. [s. l.], 2022. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=cd1e8020-44e9-318a-95e6-bbcd23d6b853>. Acesso em: 24 jun. 2025.

ZHANG, Y. *et al.* Code smell detection based on supervised learning models: A survey. **Neurocomputing**, [s. l.], v. 565, p. N.PAG, 2024. DOI 10.1016/j.neucom.2023.127014. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=ac45c1f4-556c-359a-bafc-50f74fa9b5b7>. Acesso em: 9 maio. 2025.

ZHANG, F. *et al.* Data Preparation for Deep Learning based Code Smell Detection: A Systematic Literature Review. [s. l.], 2024. Disponível em:

<https://research.ebsco.com/linkprocessor/plink?id=a175ed9f-f1e0-3e4a-ab1a-787a488499db>. Acesso em: 24 abr. 2025.

A seguir, apresenta-se o dicionário de dados das principais tabelas que compõem o Modelo Entidade-Relacionamento (MER) do sistema. Este dicionário descreve os campos, tipos e funções de cada elemento nas tabelas.

Tabela: t_user

| Campo | Tipo | Descrição |
|---------------|--------------|-----------------------------------|
| id | bigint (PK) | Identificador único do usuário |
| login | varchar(50) | Nome de usuário (login do GitHub) |
| name | varchar(50) | Nome completo do usuário |
| avatar_url | varchar(100) | URL do avatar do GitHub |
| email | varchar(100) | E-mail do usuário |
| assigned_date | timestamp | Data de cadastro no sistema |

Tabela: t_repository

| Campo | Tipo | Descrição |
|---------------|---------------|--|
| id | bigint (PK) | Identificador único do repositório |
| url | varchar(500) | URL completa do repositório no GitHub |
| name | varchar(50) | Nome do repositório |
| owner_id | bigint (FK) | Referência ao usuário proprietário (t_user.id) |
| context | varchar(1000) | Contexto ou descrição do repositório |
| assigned_date | timestamp | Data de registro do repositório no sistema |

Tabela: t_pull_request

| Campo | Tipo | Descrição |
|---------------|-------------|---|
| id | bigint (PK) | Identificador único do pull request |
| repository_id | bigint (FK) | Repositório associado (t_repository.id) |
| user_id | bigint (FK) | Usuário que criou o PR (t_user.id) |
| url | varchar(50) | URL do pull request no GitHub |
| number | int | Número do pull request |

| | | |
|--------|-------------|---|
| branch | varchar(50) | Nome da branch associada ao PR |
| status | varchar(30) | Status atual do PR (ex: open, closed, merged) |

Tabela: t_code_suggestion

| Campo | Tipo | Descrição |
|-----------------|--------------|--|
| id | bigint (PK) | Identificador único da sugestão de código |
| pull_request_id | bigint (FK) | Pull request analisado (t_pull_request.id) |
| file_path | varchar(100) | Caminho do arquivo onde a sugestão foi feita |
| line | int | Linha do arquivo onde a sugestão se aplica |
| type | int | Tipo de sugestão ou code smell detectado |
| title | varchar(50) | Título ou tipo da sugestão |
| text | text | Descrição detalhada ou proposta de refatoração |
| created_at | timestamp | Data de criação da sugestão |

Tabela: t_pull_request_report

| Campo | Tipo | Descrição |
|-----------------|-------------|--|
| id | bigint (PK) | Identificador único do relatório |
| pull_request_id | bigint (FK) | Pull request associado (t_pull_request.id) |
| markdown_text | text | Texto do relatório gerado, em formato Markdown |
| created_at | timestamp | Data de geração do relatório |

Tabela: t_repository_rules

| Campo | Tipo | Descrição |
|--------------|-------------|------------------|
|--------------|-------------|------------------|

| | | |
|---------------|--------------|---|
| id | bigint (PK) | Identificador único da regra |
| repository_id | bigint (FK) | Repositório associado (t_repository.id) |
| title | varchar(100) | Título ou nome da regra |
| text | text | Conteúdo ou descrição da regra |
| user_id | bigint (FK) | Usuário que definiu a regra (t_user.id) |
| created_at | timestamp | Data de criação da regra |

APÊNDICE B - Tabela com casos de teste com correção e sugestão de correção

| ID de Teste | Objetivo do teste | Arquivo Analisado | Smell Detectado | Sugestão de Refatoração | Data da execução |
|--------------------|--------------------------|--------------------------------|------------------------|--------------------------------|-------------------------|
| T01 | Detectar "God Class" | AndroidDebugBridge.java | Sim | Sim | 25/06/2025 |
| T02 | Detectar "God Class" | Buffer.java | Sim | Sim | 25/06/2025 |
| T03 | Detectar "God Class" | CdmaDataConnectionTracker.java | Sim | Não, apenas comentários | 25/06/2025 |
| T04 | Detectar "God Class" | DataConnection.java | Sim | Sim | 25/06/2025 |
| T05 | Detectar "God Class" | IOUtils.java | Sim | Sim | 25/06/2025 |
| T06 | Detectar "God Class" | FragmentActivity.java | Sim | Sim | 26/06/2025 |
| T07 | Detectar "God Class" | LogPanel.java | Sim | Não, apenas comentários | 26/06/2025 |
| T08 | Detectar "God Class" | PhoneBase.java | Sim | Sim | 26/06/2025 |
| T09 | Detectar "God Class" | SourceMapper.java | Sim | Não, apenas comentários | 26/06/2025 |
| T10 | Detectar "God Class" | Workspace.java | Não | Não | 26/06/2025 |

| | | | | | |
|-----|-------------------------|-------------------------|-----|------------------------|------------|
| T11 | Detector "Long Method" | CliClient.java | Sim | Sim | 26/06/2025 |
| T12 | Detector "Long Method" | ELParser.java | Sim | Sim | 26/06/2025 |
| T13 | Detector "Long Method" | NodeCmd.java | Sim | Sim | 26/06/2025 |
| T14 | Detector "Long Method" | PluginJAR.java | Sim | Sim | 26/06/2025 |
| T15 | Detector "Long Method" | SourceMapper.java | Sim | Sim | 26/06/2025 |
| T16 | Detector "Long Method" | 1/before.java | Sim | Sim | 26/06/2025 |
| T17 | Detector "Long Method" | 2/before.java | Sim | Sim | 26/06/2025 |
| T18 | Detector "Long Method" | 3/before.java | Sim | Sim | 26/06/2025 |
| T19 | Detector "Long Method" | 4/before.java | Sim | Sim | 26/06/2025 |
| T20 | Detector "Long Method" | 5/before.java | Sim | Sim | 26/06/2025 |
| T21 | Detector "Feature Envy" | Http11NioProcessor.java | Sim | Não, apenas comentário | 26/06/2025 |

| | | | | | |
|-----|-------------------------|----------------------|-----|------------------------|------------|
| T22 | Detectar "Feature Envy" | NodeProbe.java | Sim | Não, apenas comentário | 26/06/2025 |
| T23 | Detectar "Feature Envy" | Path.java | Sim | Sim | 26/06/2025 |
| T24 | Detectar "Feature Envy" | SSTableWriter.java | Sim | Sim | 26/06/2025 |
| T25 | Detectar "Feature Envy" | TextAreaPainter.java | Sim | Sim | 26/06/2025 |
| T26 | Detectar "Feature Envy" | CacheService.java | Sim | Não, apenas comentário | 26/06/2025 |
| T27 | Detectar "Feature Envy" | JUnitTask.java | Sim | Não, apenas comentário | 26/06/2025 |
| T28 | Detectar "Feature Envy" | MouseHandler.java | Sim | Sim | 26/06/2025 |
| T29 | Detectar "Feature Envy" | TextBody.java | Sim | Sim | 26/06/2025 |
| T30 | Detectar "Feature Envy" | TokenMetadata.java | Sim | Sim | 26/06/2025 |

Testes realizados pelo aluno Djonathan Mariano D'Ávila

APÊNDICE C - Formulário *online* para pesquisa com profissionais da área de desenvolvimento de sistemas.

Pesquisa sobre o Uso de Inteligência Artificial em Code Review

Olá! 🙌

Esta pesquisa faz parte do meu Trabalho de Conclusão de Curso (TCC) e tem como objetivo entender como desenvolvedores utilizam **Code Review** no dia a dia e qual a percepção sobre o uso de **Inteligência Artificial (IA) para detectar e sugerir correções de code smells**.

🔍 O que são Code Smells?

Code smells são indícios de que um código pode ter problemas de manutenção, legibilidade ou estrutura. Eles não são necessariamente bugs, mas podem dificultar futuras alterações e aumentar a complexidade do sistema. Exemplos incluem código duplicado, funções muito grandes ou classes que fazem mais do que deveriam.

Os insights coletados ajudarão a direcionar o desenvolvimento de uma ferramenta que torna o **Code Review mais eficiente e automatizado**, reduzindo retrabalho e facilitando a manutenção do código.

Sua participação é muito importante! A pesquisa é rápida e suas respostas contribuirão para a evolução do uso de IA na revisão de código.

Obrigado por participar! 🚀

* Indica uma pergunta obrigatória

Qual é sua principal ocupação? *

- Analista
- Desenvolvedor(a)
- Estudante
- Gerente de projetos
- Líder de Squad
- Outro:

Qual é seu nível de experiência na área de Desenvolvimento? *

- Nenhuma experiência
- Menos de 1 ano
- De 1 a 2 anos
- De 2 a 5 anos
- De 5 a 7 anos
- De 7 a 10 anos
- Mais de 10 anos

Qual linguagem de programação você utiliza atualmente? *

- Java
- Python
- C#
- JavaScript
- Delphi
- ABAP
- C++
- Outro: _____

Qual seu nível de conhecimento no desenvolvimento de sistema? *

- Nenhum conhecimento
- Muito básico
- Básico
- Intermediário
- Avançado
- Muito avançado
- Especialista

Com qual frequência você realiza Code Reviews em seu trabalho? *

- Muito frequente
- Frequentemente
- Ocasionalmente
- Raramente
- Nunca

Você utiliza alguma ferramenta automatizada para auxiliar no Code Review? *

- SonarQube
- Codacy
- DeepSource
- Manual
- Nenhum
- Outro: _____

Os **Code Smells** são indícios de problemas no código que podem afetar sua qualidade e manutenção. Com base na sua experiência, avalie o impacto de cada um dos seguintes code smells: *

| | Baixo | Médio | Alto |
|--|-----------------------|-----------------------|-----------------------|
| Múltiplas alterações em um mesmo arquivo para diferentes razões | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Alteração em várias partes do código para modificar uma funcionalidade | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Classe com muitas responsabilidades | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Método que usa mais atributos de outra classe do que da sua própria | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Conjuntos de variáveis que frequentemente aparecem juntas e deveriam ser encapsulados. | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Uma classe possui muitos métodos e atributos, tornando-se difícil de entender e modificar. | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Função extensa e difícil de entender | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |

| | | | |
|---|-----------------------|-----------------------|-----------------------|
| Múltiplas chamadas encadeadas dificultando a leitura | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Classe ou método que apenas repassa chamadas | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Uso desnecessário de herança ou abstrações | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Uma subclasse herda métodos ou atributos que não usa. | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Estruturas de classes duplicadas para diferentes propósitos | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Classes similares que poderiam ser unificadas | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Classes acessando detalhes internos umas das outras | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Comentários Excessivos | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Classe que não é utilizada no projeto | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Blocos de código repetidos em diferentes partes do sistema | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Variável usada apenas em determinados cenários | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Classe que apenas armazena dados sem comportamento | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Muitos IFs aninhados dificultando a compreensão | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |

Com que frequência você identifica code smells no seu código durante o desenvolvimento ou no processo de Code Review? *

- Sempre
- Frequentemente
- Às vezes
- Raramente
- Nunca

Como você classificaria a importância das seguintes funcionalidades em uma ferramenta de IA para Code Review? *

| | Baixa | Média | Alta |
|--|-----------------------|-----------------------|-----------------------|
| Detecção automática de code smells | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Sugestão de correções para code smells | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Identificação de vulnerabilidades de segurança | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Análise de padrões de codificação e boas práticas | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Explicação do problema encontrado e sugestão de melhoria | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Integração com plataformas como GitHub, GitLab e Bitbucket | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Geração de relatórios detalhados sobre qualidade do código | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Prioritização dos problemas com base no impacto no sistema | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |
| Revisão automática de pull/merge requests | <input type="radio"/> | <input type="radio"/> | <input type="radio"/> |

Quais ferramentas de Integração Contínua (CI) você utiliza? (Selecione todas as que se aplicam) *

GitHub Actions

GitLab CI/CD

Jenkins

CircleCI

Travis CI

Bitbucket Pipelines

Azure DevOps

TeamCity

Bamboo

Buildkite

Outro: _____

Quais ferramentas você utiliza para gerenciar o ciclo de vida do desenvolvimento do seu software (ALM)? *

Azure

ClickUp

DevOps

Jira

Trello

ServiceNow

Ferramenta interna

Nenhum

Outro: _____

Qual plataforma de hospedagem de código-fonte você mais utiliza? *

- Azure DevOps Repos
- AWS CodeCommit
- Bitbucket
- GitHub
- GitLab
- Google Cloud Source Repositories
- TortoiseSVN
- Nenhuma
- Outro: _____

Qual é a sua satisfação geral com as ferramentas de Code Review atualmente disponíveis? *

- Muito satisfeito
- Satisfeito
- Indiferente
- Insatisfeito
- Muito insatisfeito

Quais funcionalidades são essenciais em uma ferramenta de IA para Code Review? (Escolha até 3 opções) *

- Detecção automática de code smells
- Sugestão de correções para code smells
- Identificação de vulnerabilidades de segurança
- Análise de padrões de codificação e boas práticas
- Explicação do problema encontrado e sugestão de melhoria
- Integração com plataformas como GitHub, GitLab e Bitbucket
- Geração de relatórios detalhados sobre qualidade do código
- Prioritização dos problemas com base no impacto no sistema
- Revisão automática de pull/merge requests
- Outro:

Você concorda que a Inteligência Artificial pode melhorar a detecção e sugerir correção de code smells durante o Code Review? *

- Concordo totalmente
- Concordo
- Não estou decidido
- Discordo
- Discordo totalmente