UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”
FACULDADE DE CIÊNCIAS – CAMPUS DE BAURU
BACHARELADO EM SISTEMAS DE INFORMAÇÃO

CAUET CORRÊA DE MENEZES
GUSTAVO CARVALHO ALVES DOS SANTOS

RefactorScore: Um Sistema de Monitoramento e Avaliação de Qualidade de Código em Commits

BAURU
2025

CAUET CORRÊA DE MENEZES
GUSTAVO CARVALHO ALVES DOS SANTOS

RefactorScore: Um Sistema de Monitoramento e Avaliação de Qualidade de Código em Commits

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus de Bauru, como parte dos requisitos para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Higor Amario de Souza.

BAURU
2025

Menezes, Cauet Corrêa de,
A importância da biblioteca na formação acadêmica : não dá pra viver sem ela / Cauet Corrêa de Menezes, Gustavo Carvalho Alves dos Santos. – Bauru, 2025
46 f. : il.

Trabalho de conclusão de curso (Bacharelado – Sistemas de Informação)–Universidade Estadual Paulista (Unesp), Faculdade de Ciências, Bauru
Orientador: Higor Amario de Souza

1. Análise de código. 2. Clean code. 3. Engenharia de software. 4. Modelo de linguagem (LLM). 5. Refatoração. I. Gustavo Carvalho Alves dos Santos. II. Universidade Estadual Paulista. Faculdade de Ciências. III. Título.

CAUET CORRÊA DE MENEZES
GUSTAVO CARVALHO ALVES DOS SANTOS

RefactorScore: Um Sistema de Monitoramento e Avaliação de Qualidade de Código em Commits por Inteligência Artificial

Trabalho de Conclusão de Curso apresentado ao Curso de Bacharelado em Sistemas de Informação da Universidade Estadual Paulista "Júlio de Mesquita Filho", Faculdade de Ciências, Campus de Bauru, como parte dos requisitos para obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Prof. Dr. Higor Amario de Souza.

Banca Examinadora

Prof. Dr. Higor Amario de Souza
Orientador

Bauru, 01 de Dezembro de 2025.

RESUMO

A aprendizagem de boas práticas de programação é essencial para estudantes iniciantes, pois influencia diretamente a qualidade, manutenção e evolução de seus projetos. Entretanto, a identificação de erros e a compreensão de recomendações de melhoria ainda representam desafios para esse público, devido à falta de experiência e feedback personalizado. Este trabalho apresenta o desenvolvimento de um sistema analisador de código em .NET voltado para iniciantes, capaz de avaliar repositórios locais de exercícios e fornecer, por meio de um dashboard, notas de qualidade e recomendações de estudo personalizadas. Para a análise do código, o sistema utiliza técnicas de análise estática combinadas com um modelo de linguagem avançado (LLM), especificamente o Qwen2.5-coder, que permite identificar pontos de melhoria relacionados a padrões de codificação, complexidade e legibilidade, apresentando justificativas baseadas em boas práticas de desenvolvimento. Os resultados obtidos incluem o fornecimento de feedback imediato ao programador iniciante, a padronização do aprendizado e a redução de vícios comuns em programação, contribuindo para a evolução técnica dos estudantes.

Palavras-chave: Análise de código; .NET; educação em programação; iniciantes; feedback automatizado; Qwen2.5-coder.

ABSTRACT

Learning good programming practices is essential for beginner students as it directly influences the quality, maintainability, and evolution of their projects. However, identifying mistakes and understanding improvement recommendations remain challenges for this audience due to the lack of experience and personalized feedback. This work presents the development of a .NET-based code analysis system designed for beginners, capable of evaluating local exercise repositories and providing quality scores and personalized study recommendations through a dashboard. For code analysis, the system employs static analysis techniques combined with an advanced language model (LLM), specifically the Qwen2.5-coder, which identifies improvement points related to coding standards, complexity, and readability, offering justifications grounded in software development best practices. The results include providing immediate feedback to novice programmers, standardizing learning, and reducing common programming pitfalls, thereby contributing to the technical development of students.

Keywords: Code analysis; .NET; programming education; beginners; automated feedback; Qwen2.5-coder.

Lista de Figuras

Lista de Tabelas

Lista de Siglas e Abreviaturas

GPU	Graphics Processing Unit: Unidade de Processamento Gráfico utilizada para acelerar cálculos paralelos, especialmente em renderização de imagens e aplicações de IA.
GTX	Giga Texel Shader eXtreme: Linha de placas de vídeo da Nvidia focada em alto desempenho gráfico, sem núcleos dedicados a ray tracing.
RTX	Ray Tracing TeXel: Linha de GPUs da Nvidia equipada com núcleos dedicados a ray tracing e IA (RT Cores e Tensor Cores), oferecendo iluminação e efeitos mais realistas.
API	Interface de Programação de Aplicações
Qwen2.5-coder	Modelo especializado em tarefas de programação
Git	Sistema de Controle de Versão Distribuído
IA	Inteligência Artificial
LLM	Modelo de Linguagem de Grande Escala
Ollama	Ferramenta de gerenciamento de LLM
.NET	Framework de desenvolvimento da Microsoft

Sumário

1 INTRODUÇÃO

O desenvolvimento de software é uma atividade dinâmica e contínua, na qual o código-fonte passa constantemente por modificações, seja para correção de defeitos, inclusão de novas funcionalidades ou melhorias técnicas. Para estudantes iniciantes, a aprendizagem de boas práticas de programação é fundamental, pois influencia diretamente a qualidade, manutenção e evolução de seus projetos. No entanto, nem sempre essas alterações são realizadas de maneira que garantam a qualidade e a manutenção do código, especialmente devido à falta de experiência e feedback personalizado. Nesse contexto, práticas de engenharia de software como refatoração e revisão de código se tornam fundamentais para assegurar a saúde do projeto ao longo do tempo (Fowler 2018).

A refatoração, definida como a melhoria contínua da estrutura interna do código sem alteração de seu comportamento externo, é uma das práticas mais relevantes para manter a qualidade técnica de um software (Martin 2008). Entretanto, para iniciantes, identificar erros e compreender recomendações de melhoria ainda representa um desafio, tornando a medição e o acompanhamento da qualidade das refatorações realizadas ao longo do tempo uma tarefa complexa.

A qualidade do código-fonte é um dos principais fatores que influenciam a manutenção, extensibilidade e confiabilidade de sistemas de software. Em ambientes de aprendizagem e desenvolvimento ágil, o acompanhamento sistemático da qualidade das mudanças realizadas no código, especialmente no nível dos exercícios e *commits*, torna-se uma tarefa desafiadora (Martin 2008).

Muitas organizações e ambientes educacionais dependem de revisões manuais de código (*code review*) para garantir a qualidade das alterações. Apesar de ser uma prática recomendada, o *code review* possui limitações, como a subjetividade da análise, dependência da experiência dos revisores e o tempo necessário para execução (Fowler 2018). Além disso, com a crescente adoção de práticas *DevOps* e integração contínua (CI), o número de *commits* ou exercícios realizados diariamente pode ser muito elevado, dificultando uma revisão detalhada de cada um.

A dificuldade em identificar padrões ou tendências de qualidade a partir dos exercícios ou *commits* realizados é outro problema comum. É comum que equipes técnicas e estudantes tenham métricas relacionadas ao número de *commits* ou linhas de código alteradas, porém é raro que possuam indicadores objetivos sobre a qualidade dessas alterações (Brown et al. 2010). A falta de ferramentas para automatizar a análise qualitativa do código contribui para o surgimento de problemas estruturais, elevando a complexidade do sistema e tornando-o mais suscetível a falhas. Isso afeta diretamente os custos de manutenção e a produtividade ao longo do tempo. MacCormack e Sturtevant 2016 discutem essa questão em sua análise sobre como o acoplamento e outras formas de dívida técnica degradam a arquitetura e aumentam o esforço necessário para a evolução de sistemas de software.

Com o crescimento de abordagens baseadas em dados e inteligência artificial, torna-se

viável o desenvolvimento de soluções automatizadas capazes de monitorar e avaliar aspectos qualitativos das mudanças realizadas no código-fonte. O uso de grandes modelos de linguagem (LLMs), como o Qwen2.5-coder, permite interpretar e analisar código de maneira contextual, gerando *insights* e recomendações personalizadas que antes dependiam exclusivamente de revisores humanos (Chen et al. 2021).

Entretanto, apesar da existência dessas tecnologias, ainda são escassas as soluções que integrem de forma automatizada a análise de qualidade de código em ambientes educacionais ou *pipelines* de desenvolvimento, gerando indicadores quantitativos e qualitativos de fácil visualização para estudantes, líderes e gestores.

Diante desse cenário, o problema central que este trabalho busca solucionar é: Como automatizar o monitoramento da qualidade de código produzido por iniciantes, a partir da análise de exercícios e *commits*, atribuindo notas e recomendações baseadas em critérios técnicos, de forma a fornecer métricas visuais e feedback personalizado que auxiliem na evolução técnica dos estudantes?

Para responder a essa questão, propõe-se o desenvolvimento de um sistema automatizado de monitoramento de qualidade de código, que executa análises periódicas dos exercícios ou *commits* realizados em repositórios locais. O sistema atribui uma nota para cada exercício, avaliando a qualidade da implementação e classificando-a em categorias como problemático, precisa melhorar, aceitável, bom, muito bom e excelente, além de fornecer recomendações de estudo personalizadas. Esses dados são armazenados e disponibilizados em *dashboards*, fornecendo uma visão clara sobre a evolução técnica dos estudantes.

A arquitetura sugerida se baseia em um *Worker Service*, operando de maneira cíclica de acordo com as demandas do usuário e com acesso direto ao repositório do projeto. A avaliação do código é executada pelo Qwen2.5-coder, funcionando localmente e guiado por *prompts* personalizados, que orientam a análise das respostas dos exercícios. Finalmente, os resultados produzidos são salvos em um banco de dados não relacional e, posteriormente, acessados por meio de um *dashboard* personalizado, executado localmente no dispositivo.

Assim, este trabalho busca contribuir com a comunidade de desenvolvimento de software ao propor uma ferramenta de apoio à aprendizagem de boas práticas de programação, alinhando refatoração, inteligência artificial e análise de dados para atender às necessidades de estudantes iniciantes e ambientes educacionais modernos.

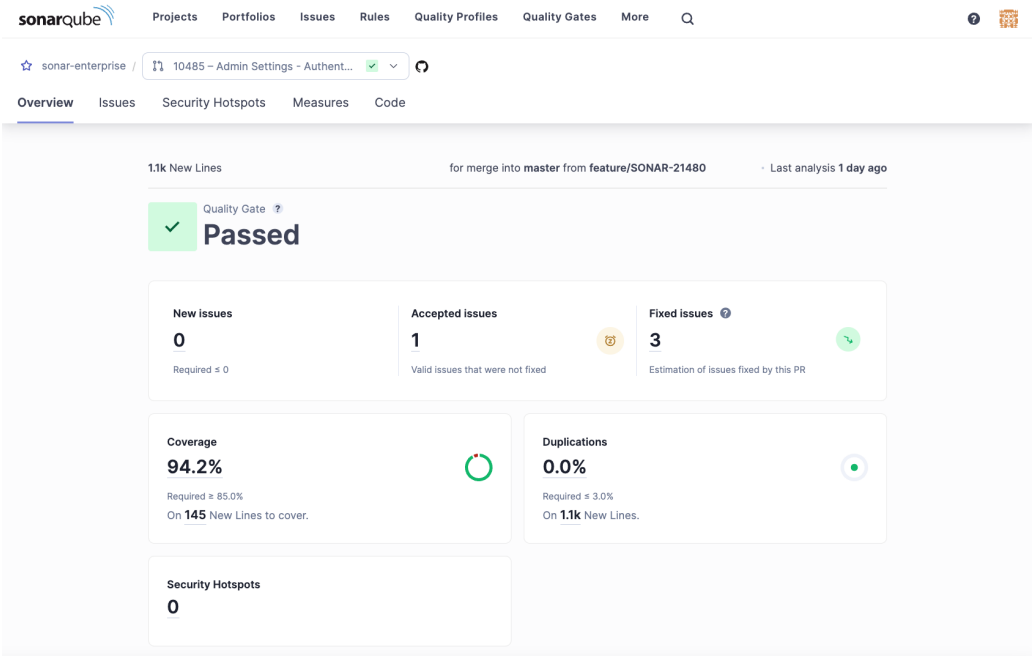
2 SOLUÇÕES EXISTENTES

Existem diversas ferramentas e práticas de mercado voltadas para a análise de código e suporte à qualidade de software. Entretanto, poucas delas possuem foco específico em realizar avaliações cíclicas e automatizadas sobre *commits* recentes, com objetivo de gerar indicadores estratégicos para gestores de software. A seguir, destacam-se algumas soluções relevantes e semelhantes à ideia central proposta neste trabalho.

2.1 SonarQube

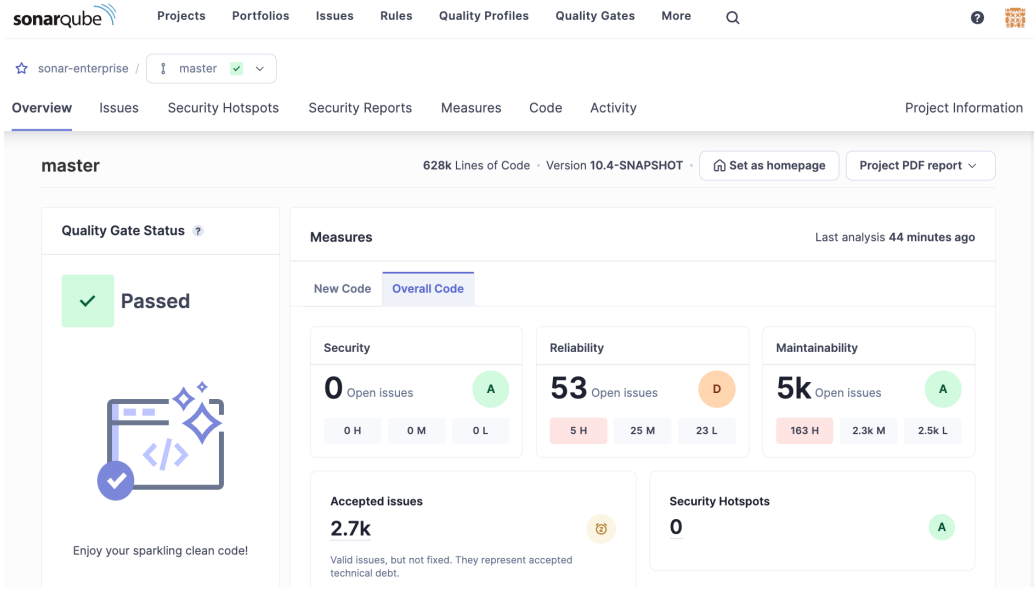
O SonarQube é uma das ferramentas mais consolidadas do mercado para análise estática de código. Ele permite avaliar qualidade técnica por meio de regras pré-definidas, buscando detectar vulnerabilidades, *code smells*, duplicações e complexidade ciclomática. Apesar de eficiente, o SonarQube atua fortemente sobre o código como um todo, e não exclusivamente sobre os *commits* diários e seu impacto evolutivo.

Figura 1: Resumo do pull request no servidor SonarQube mostrando as categorias de problemas aceitos e problemas corrigidos.



Fonte: <https://www.sonarsource.com/products/sonarqube/whats-new/sonarqube-10-4/>

Figura 2: Interface do SonarQube exibindo indicadores de qualidade do código, como *bugs*, vulnerabilidades e duplicações.

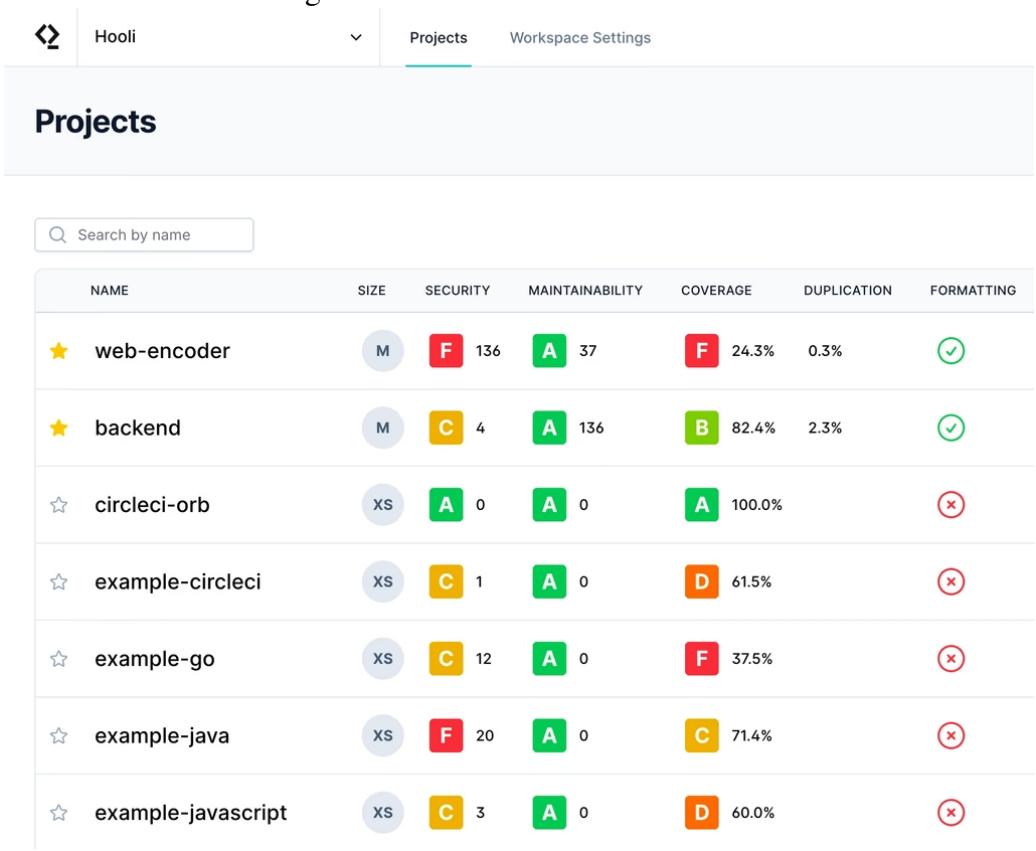


Fonte: <https://www.sonarsource.com/products/sonarqube/whats-new/sonarqube-10-4/>

2.2 CodeClimate

O CodeClimate é uma plataforma que oferece *insights* de qualidade de código e métricas relacionadas à manutenção. Assim como o SonarQube, atua com análise estática, mas com foco em equipes ágeis e *DevOps*. Porém, não há uma análise detalhada de refatorações ou uma pontuação de *commits* individuais.

Figura 3: Dashboard do CodeClimate.

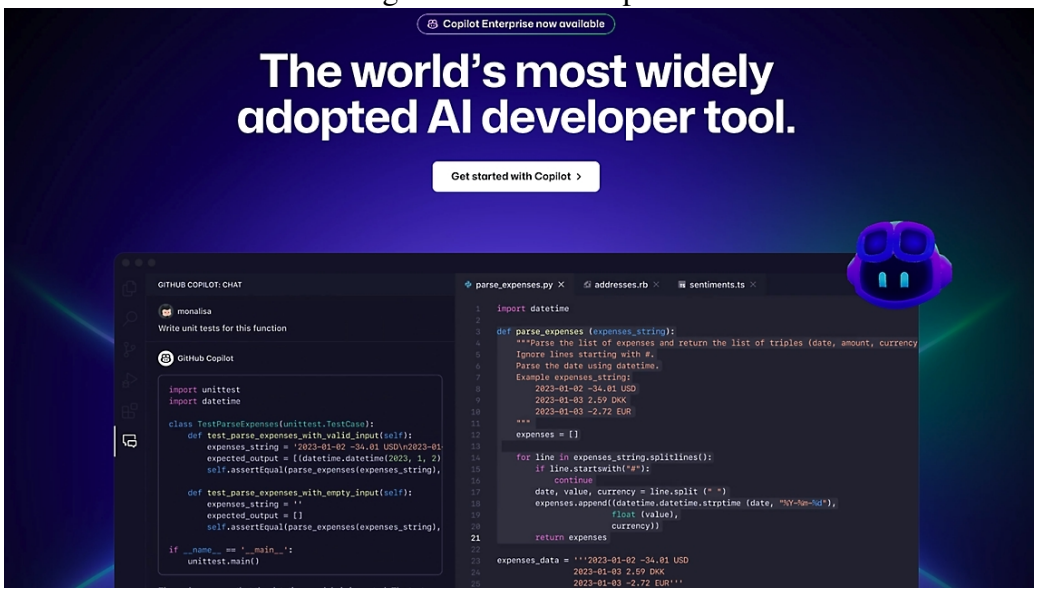


Fonte: <https://qlty.sh/>

2.3 GitHub Copilot + Code Review

O GitHub Copilot, aliado a ferramentas de *code review*, pode auxiliar desenvolvedores com sugestões de código, mas não possui um recurso de análise histórica ou monitoramento contínuo da evolução do repositório.

Figura 4: GitHub Copilot.



Fonte: <https://azure.microsoft.com/pt-br/products/github/copilot>

3 OBJETIVOS

O objetivo geral deste trabalho é desenvolver um sistema automatizado denominado *RefactorScore*, capaz de analisar e avaliar a qualidade de código com base em *commits* realizados em repositórios *Git*, fornecendo orientações educativas e métricas de acompanhamento direcionadas ao desenvolvimento de habilidades de programação para desenvolvedores iniciantes e autodidatas.

3.1 Objetivos Específicos

a) Investigar e revisar literatura e ferramentas existentes voltadas à análise da qualidade de código e sua relação com *commits*;

b) Projetar uma arquitetura de sistema baseada nos princípios de arquitetura limpa, capaz de integrar-se com repositórios *Git* locais, coletar informações de *commits* e realizar análises automatizadas utilizando modelos de linguagem;

c) Implementar mecanismos de avaliação de qualidade baseados em critérios definidos pelo *Clean Code: A Handbook of Agile Software Craftsmanship*;

d) Desenvolver funcionalidades de geração de recomendações personalizadas, incluindo sugestões de melhoria e referências a capítulos do livro *Clean Code: A Handbook of Agile Software Craftsmanship*;

e) Implementar recursos de visualização temporal que permitam aos desenvolvedores acompanhar sua evolução e identificar tendências na qualidade de seu código ao longo do tempo.

4 DESCRIÇÃO DO PROJETO PROPOSTO

Esta seção apresenta o sistema *RefactorScore*, uma solução automatizada para análise da qualidade de código em repositórios *Git* locais, especialmente voltada para estudantes iniciantes. O sistema realiza inspeções periódicas dos *commits* realizados, utilizando técnicas de análise estática combinadas a grandes modelos de linguagem para avaliar aspectos técnicos e qualitativos do código. O código-fonte da ferramenta encontra-se disponível para consulta em repositório público[1].

Desenvolvido para operar em ambiente *Docker*, o *RefactorScore* garante portabilidade, isolamento e facilidade de implantação, permitindo que a análise seja executada de forma confiável em diferentes máquinas e contextos. A seguir, são detalhadas as principais funcionalidades do sistema, sua arquitetura modular e os componentes que viabilizam a geração de métricas, notas e recomendações personalizadas apresentadas em *dashboards* interativos, promovendo a evolução técnica e a padronização do aprendizado em programação.

4.1 Principais funcionalidades

O sistema proposto tem como objetivo a análise da qualidade de código de um repositório *Git* local, por meio da inspeção diária dos *commits* realizados. Esta análise é efetuada por um grande modelo de linguagem (*Large Language Model* - LLM), que aplica critérios técnicos e qualitativos definidos por meio de *prompt engineering*, com base nos princípios apresentados na obra *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin 2008). As principais funcionalidades do sistema são detalhadas a seguir:

a) **Execução automatizada e configurável:** o sistema é concebido como um serviço automatizado, executado em um ambiente conteinerizado com *Docker* para assegurar portabilidade, facilidade de implantação e isolamento do ambiente de execução. As análises periódicas dos *commits* em repositórios locais são realizadas em intervalos configuráveis (padrão de 1 hora), atribuindo notas de qualidade e fornecendo recomendações personalizadas. Para tal, são empregadas técnicas de análise estática combinadas com o modelo de linguagem *qwen2.5-coder:7b*, capaz de identificar pontos de melhoria relacionados a padrões de codificação, complexidade e legibilidade;

b) **Leitura de repositório local:** por meio da biblioteca `LibGit2Sharp`, o sistema acessa diretamente repositórios *Git* locais, extraindo informações de *commits*, diferenças entre versões (*diffs*) e metadados associados, sem a necessidade de conexão com servidores remotos;

c) **Coleta e mapeamento de *commits*:** o serviço lê os objetos do *Git* via `LibGit2Sharp` e mapeia internamente as alterações para entidades de domínio. São aplicados filtros

[1] Disponível em: <https://github.com/GustavoCarvalho25/RefactorScore>. Acesso em: 5 nov. 2025.

por data e projeto para processar alterações recentes e relevantes, preparando os dados (arquivos, *diffs* e metadados) para a etapa de análise pelo modelo de linguagem;

d) **Análise por modelo de linguagem local:** os arquivos de cada *commit* são submetidos ao modelo `qwen2.5-coder:7b`, hospedado localmente via *Ollama*. O sistema instrui o modelo a avaliar cinco métricas fundamentadas nos princípios de *Clean Code*, atribuindo notas de 1 a 10 para cada uma: nomenclatura de variáveis (clareza e consistência dos nomes), tamanho de funções (foco e simplicidade), código autoexplicativo (minimização de comentários desnecessários), coesão de métodos (responsabilidades bem definidas em classes e métodos) e código morto (ausência de código redundante ou não utilizado). Além das notas, o modelo fornece justificativas textuais específicas para cada métrica, citando elementos concretos do código analisado;

e) **Pontuação e classificação de qualidade:** a nota final de cada arquivo é calculada como a média aritmética das cinco métricas avaliadas. Com base nessa nota, o sistema classifica a qualidade do código em seis níveis: excelente (9,0-10,0), muito bom (7,5-8,9), bom (6,0-7,4), aceitável (5,0-5,9), precisa melhorar (3,5-4,9) e problemático (abaixo de 3,5). A nota geral do *commit* é obtida pela média das notas de todos os arquivos analisados;

f) **Geração de sugestões de melhoria:** com base nas métricas avaliadas, o sistema gera até três sugestões priorizadas de melhoria para cada arquivo. As sugestões focam nas áreas com notas mais baixas, citando elementos específicos do código e referenciando capítulos relevantes da obra *Clean Code*. Cada sugestão inclui título, descrição detalhada, prioridade (baixa, média ou alta), tipo (nomenclatura, estrutura, documentação, coesão ou código morto, entre outros), dificuldade de implementação (fácil, média ou difícil) e recursos de estudo recomendados;

g) **Armazenamento local dos resultados:** todas as análises realizadas são armazenadas em um banco de dados local para garantir a rastreabilidade e a integridade dos dados. Informações como o identificador do *commit*, data, autor, projeto, notas por arquivo e gerais, justificativas detalhadas, sugestões de melhoria e metadados de alterações (linhas adicionadas e removidas) são preservadas. O banco utiliza índices otimizados para consultas por projeto, data e identificador de *commit*;

h) **Interface de acesso aos dados:** um serviço de aplicação expõe uma interface de comunicação para consumo pelos painéis de visualização. Entre os recursos disponibilizados, estão a obtenção de estatísticas gerais (total de análises, nota média, arquivos únicos analisados e total de sugestões), a listagem de análises com paginação e filtros por projeto e período, a consulta detalhada de um *commit* específico por seu identificador e a listagem de projetos únicos disponíveis;

i) **Visualização dos dados:** os resultados das análises são disponibilizados por meio de painéis interativos (*dashboards*) que facilitam o acompanhamento da evolução técnica. A interface apresenta estatísticas consolidadas, gráficos de evolução temporal das notas, distribuição de arquivos por linguagem de programação e detalhamento individual de cada *commit* com suas métricas, justificativas e sugestões personalizadas. O sistema permite filtros por projeto e período, navegação interativa entre gráficos e análises, e visualização detalhada de cada arquivo analisado. A apresentação visual dos dados visa facilitar o acompanhamento da evolução técnica dos estudantes, promover a padronização do aprendizado e contribuir para a redução de práticas de programação inadequadas.

4.1.1 Diagrama de blocos

A Figura 5 apresenta o diagrama de blocos da arquitetura do sistema *RefactorScore*, organizada em camadas, de modo a garantir a separação de responsabilidades, escalabilidade e facilidade de manutenção, seguindo os princípios de *Clean Architecture* e *Domain-Driven Design*.

- **Camada de domínio (núcleo):** Contém as entidades centrais e as regras de negócio do sistema, como `CommitAnalysis` (representação de uma análise de *commit*), `CommitFile` (arquivo analisado), `CleanCodeRating` (avaliação de qualidade com as cinco métricas) e `Suggestion` (sugestões de melhoria). Esta camada define também as interfaces de serviços essenciais, como análise de código, acesso a repositórios *Git* e comunicação com modelos de linguagem, garantindo independência de tecnologias externas;

- **Camada de aplicação:** Orquestra os casos de uso do sistema, coordenando a lógica de negócio por meio de serviços como análise automatizada de *commits*, avaliação de qualidade por modelo de linguagem e geração de sugestões personalizadas. Esta camada conecta o domínio à infraestrutura, executando as regras de negócio definidas no núcleo;

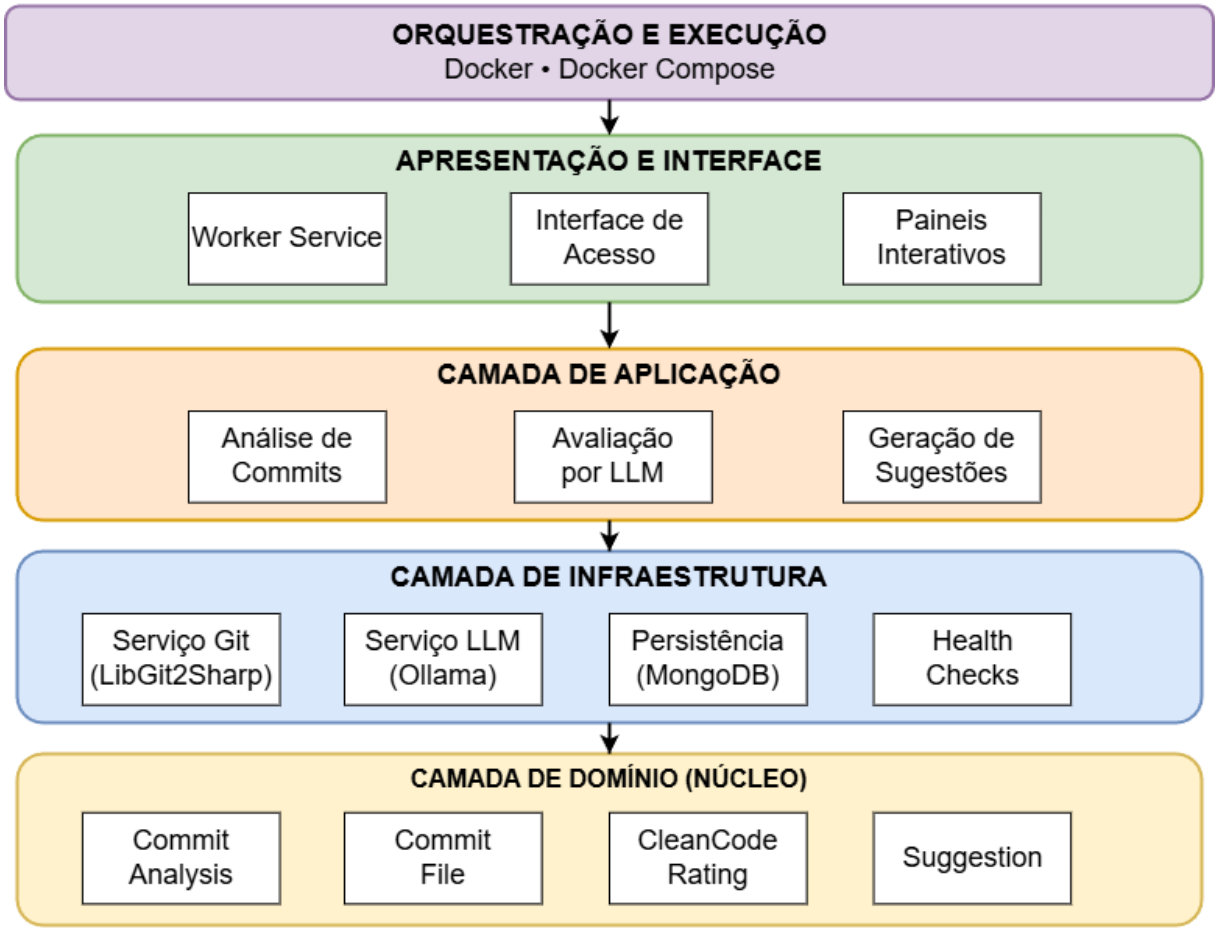
- **Camada de infraestrutura:** Fornece implementações concretas dos serviços definidos pelo domínio, incluindo acesso a repositórios *Git* (via `LibGit2Sharp`), comunicação com modelo de linguagem (*Ollama*), persistência em banco de dados (*MongoDB*), mapeamento de entidades, verificações de saúde (*health checks*) e configurações do sistema. Esta camada isola detalhes tecnológicos, permitindo substituição de componentes sem impactar o núcleo;

- **Camada de apresentação e interface:** Composta por dois componentes principais: um serviço automatizado (*worker*) responsável pela execução periódica das análises, e uma interface de acesso aos dados que disponibiliza os resultados para consumo externo. A visualização é realizada por meio de painéis interativos que apresentam estatísticas, gráficos e detalhes das análises;

- **Orquestração e execução:** Utiliza *Docker* e *Docker Compose* para gerenciar a execução isolada e integrada de todos os componentes do sistema (banco de dados, modelo de linguagem, serviço de análise e interface), garantindo portabilidade, facilidade de implantação e consistência entre diferentes ambientes de execução.

Esta arquitetura modular favorece a evolução futura do sistema, facilitando a introdução de novos serviços e tecnologias sem comprometer a estabilidade das funcionalidades existentes.

Figura 5: Diagrama de Blocos



Fonte: Criado pelos autores.

4.1.2 Diagrama de classe

O diagrama de classes, apresentado na Figura 6, ilustra a estrutura fundamental do sistema *RefactorScore*, detalhando as principais entidades, suas propriedades e os relacionamentos entre elas. O sistema é composto por classes que representam tanto os elementos de domínio (como análises de *commits* e sugestões de melhoria) quanto interfaces de serviços e persistência, seguindo os princípios de *Domain-Driven Design* (Evans 2003) e *Clean Architecture* (Martin 2017).

- `CommitAnalysis`: representa a análise completa de um *commit*, armazenando informações como identificador do *commit*, projeto, autor, e-mail, datas de *commit* e análise, linguagem predominante, linhas adicionadas e removidas, lista de arquivos analisados, avaliação geral de qualidade (`CleanCodeRating`) e nota geral calculada (`OverallNote`). Esta classe é o agregado raiz do domínio (Evans 2003), coordenando a análise de todos os arquivos de um *commit*;

- `CommitFile`: detalha as informações e análise de um arquivo individual dentro de um *commit*, incluindo caminho do arquivo, linguagem de programação, linhas adicionadas e removidas, conteúdo completo do arquivo, avaliação de qualidade específica (`CleanCodeRating`) e lista de sugestões de melhoria. A propriedade `HasAnalysis` indica se o arquivo já foi submetido à análise pelo modelo de linguagem;

- `CleanCodeRating`: objeto de valor (Evans 2003) responsável por armazenar avaliações quantitativas e qualitativas relacionadas aos princípios de código limpo, incluindo cinco métricas numéricas (nomenclatura de variáveis, tamanho de funções, código autoexplicativo, coesão de métodos e código morto), nota calculada, classificação de qualidade e dicionário de justificativas textuais detalhadas para cada métrica avaliada;

- `Suggestion`: objeto de valor que representa sugestões de melhoria geradas a partir da análise, contendo título, descrição detalhada, prioridade (baixa, média ou alta), tipo (nomenclatura, estrutura, documentação, coesão, código morto, entre outros), dificuldade de implementação (fácil, média ou difícil), referência ao arquivo analisado, data de última atualização e lista de recursos de estudo recomendados (capítulos de *Clean Code*);

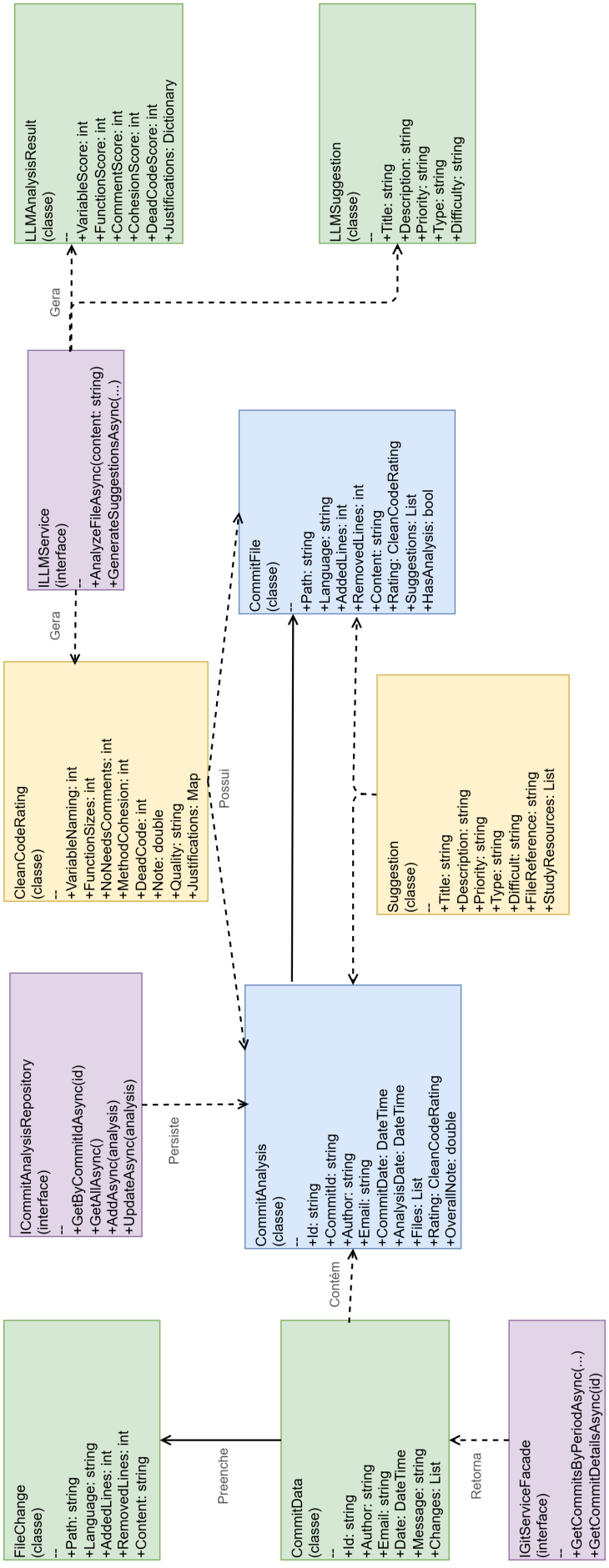
- `ILLMService`: interface de serviço de domínio responsável pela comunicação com modelos de linguagem para análise de código, definindo métodos para análise de arquivos individuais e geração de sugestões personalizadas com base nas avaliações de qualidade;

- `IGitServiceFacade`: interface de serviço de domínio que abstrai operações de acesso a repositórios *Git* locais, incluindo métodos para obtenção de *commits* por período, recuperação de informações detalhadas de *commits* específicos e extração de conteúdo de arquivos modificados;

- `ICommitAnalysisService`: interface de serviço de aplicação responsável por orquestrar o processo completo de análise de *commits*, coordenando a leitura do repositório, submissão ao modelo de linguagem e persistência dos resultados;
- `ICommitAnalysisRepository`: interface de repositório (Evans 2003) responsável pela persistência das análises de *commits*, definindo operações de inclusão, consulta por identificador único ou por identificador de *commit*, listagem completa, atualização e exclusão de registros no banco de dados.

Este diagrama evidencia a separação de responsabilidades entre as camadas (Martin 2017), a modularidade do sistema, a independência de *frameworks* e tecnologias externas por meio da inversão de dependências, e a integração entre análise automatizada por modelo de linguagem, avaliação qualitativa estruturada e geração de recomendações personalizadas.

Figura 6: Diagrama de Classe



Fonte: Criado pelos autores.

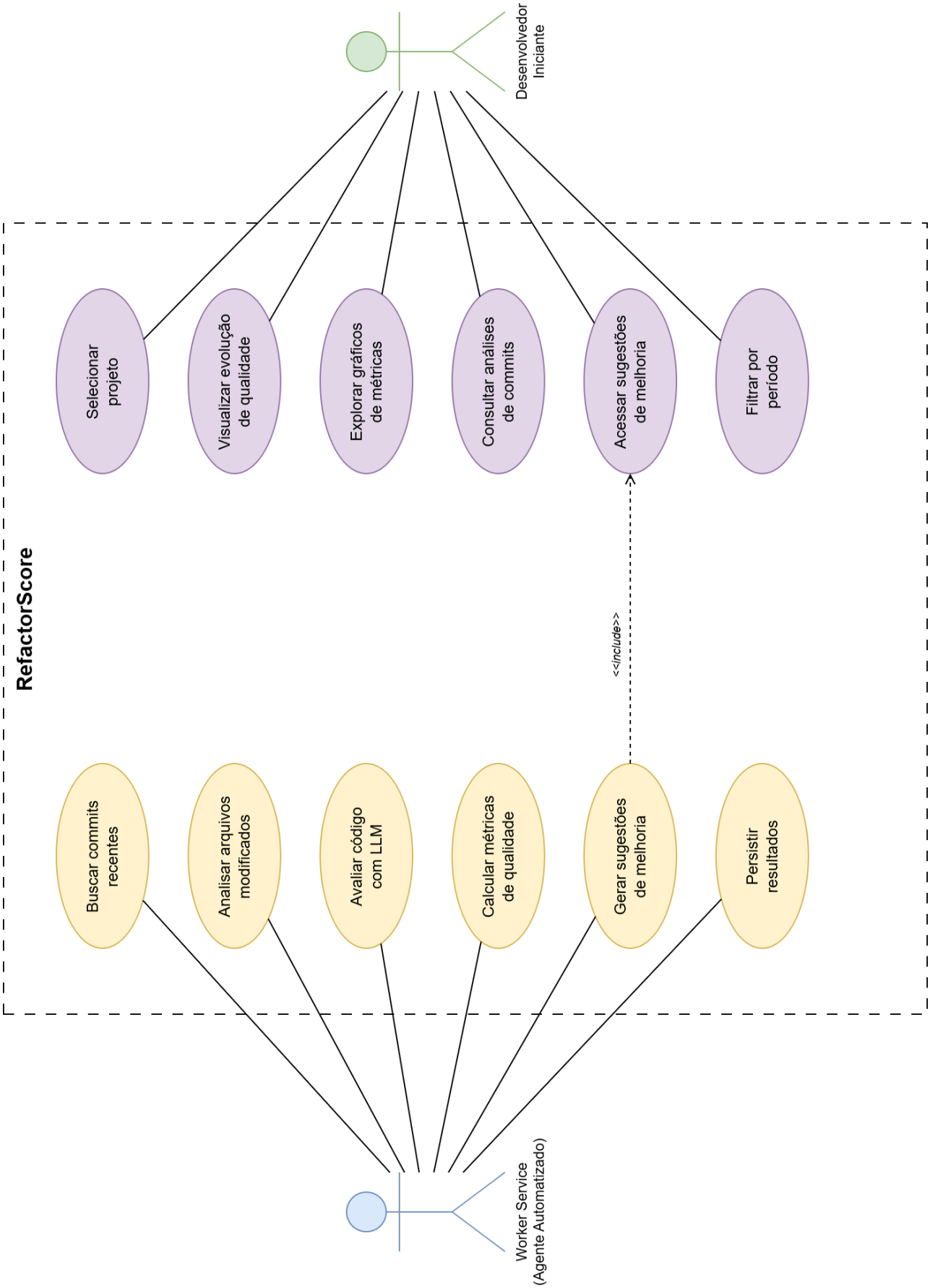
4.1.3 Diagrama de casos de uso

O diagrama de casos de uso, apresentado na Figura 7, descreve as principais interações entre os atores e o sistema *RefactorScore*, evidenciando os fluxos funcionais essenciais para análise automatizada de código e acompanhamento educacional.

- **Worker Service (Agente Automatizado):** representa o serviço de análise contínua responsável por executar periodicamente as seguintes operações:
  - buscar *commits* recentes do repositório *Git* local;
  - analisar arquivos modificados em cada *commit*;
  - avaliar código utilizando modelo de linguagem (LLM);
  - calcular métricas de qualidade segundo princípios de *Clean Code*;
  - gerar sugestões de melhoria personalizadas;
  - persistir resultados no banco de dados.
- **Desenvolvedor Iniciante:** interage com a interface web do *RefactorScore* para:
  - selecionar projeto a ser visualizado;
  - explorar o histórico de evolução de qualidade do código;
  - visualizar gráficos de distribuição de métricas e evolução temporal;
  - acessar análises detalhadas de *commits* específicos;
  - consultar sugestões de melhoria com recursos de estudo (capítulos de *Clean Code*);
  - filtrar análises por período e pesquisar por identificador de *commit*.

Este diagrama evidencia como o *RefactorScore* integra automação de análise de código e acompanhamento educacional, oferecendo *feedback* contínuo, personalizado e fundamentado em boas práticas de engenharia de software para apoiar o desenvolvimento de habilidades em programação limpa.

Figura 7: Diagrama de Casos de Uso



Fonte: Criado pelos autores.

4.2 Metodologia

A metodologia adotada para o desenvolvimento e validação do sistema proposto foi estruturada em etapas sequenciais e iterativas, com base em práticas ágeis e princípios de engenharia de software. O processo buscou garantir a robustez funcional e a fidedignidade das análises geradas. As etapas metodológicas são detalhadas a seguir:

1. **Levantamento e Definição dos Critérios de Avaliação:** A primeira etapa consistiu na fundamentação teórica do processo de análise. Foram definidos critérios de avaliação de qualidade de código com base nos princípios e práticas propostos na obra *Clean Code: A Handbook of Agile Software Craftsmanship*, de Robert C. Martin (Martin 2008). Tais critérios serviram como diretriz para a construção dos modelos de instrução (*prompts*) do sistema.

2. **Desenvolvimento Iterativo do Serviço de Análise (*Worker Service*):** O serviço de análise foi desenvolvido de forma modular e incremental. As funcionalidades de leitura de repositório, extração de *commits*, interação com o modelo de linguagem e persistência de dados foram implementadas e testadas em ciclos curtos, permitindo o refinamento contínuo de cada componente.

3. **Aplicação de Arquitetura Limpa e *Domain-Driven Design*:** O *Worker Service* foi estruturado seguindo os princípios de *Clean Architecture* (Martin 2017), organizando o código em camadas bem definidas (*Domain, Application, Infrastructure* e *Presentation*), promovendo a separação de responsabilidades e independência de *frameworks*. Adicionalmente, foram aplicados conceitos de *Domain-Driven Design* (Evans 2003) para modelagem do domínio do problema, utilizando padrões táticos como Entidades, Objetos de Valor, Agregados e Repositórios, garantindo que o código reflita as regras de negócio relacionadas à análise de qualidade de código.

4. **Elaboração e Refinamento de *Prompts* (Engenharia de *Prompt*):** A construção de *prompts* eficazes foi uma etapa fundamental do projeto. Seguindo as diretrizes de Berryman e Ziegler (Berryman e Ziegler 2024), foram desenvolvidas e testadas múltiplas estruturas de instrução para o modelo de linguagem, aplicando técnicas de engenharia de *prompt* como contextualização detalhada, decomposição de tarefas e especificação clara de formato de saída. O objetivo foi assegurar que o modelo produzisse análises coerentes, consistentes e alinhadas aos critérios predefinidos de *Clean Code*, mesmo diante de entradas de código extensas e complexas. Instruções explícitas foram incluídas para mitigar alucinações e aumentar a confiabilidade das respostas geradas.

5. **Limitação de Recursos e Integração de Modelos de Linguagem Locais:** O principal desafio técnico residiu na elevada demanda computacional exigida para a execução de grandes modelos de linguagem (LLMs) em ambiente local (via *Ollama*). Tentativas

iniciais de integração com modelos como o *DeepSeek* apresentaram dificuldades de compatibilidade e desempenho. Após uma fase de testes com diferentes modelos de código aberto, como *Deepseek-coder:6.7b* e *Mistral:7b-instruct-v0.2-q4-0*, o modelo *Qwen 2.5* foi selecionado, pois apresentou o melhor equilíbrio entre desempenho, qualidade das respostas e facilidade de integração na arquitetura proposta. Ainda assim, a demanda de processamento impôs uma readequação do escopo do projeto, direcionando o foco da ferramenta para a análise de fragmentos de código menores, característicos de repositórios de estudantes iniciantes.

6. **Validação da Coerência do Modelo:** Para aferir a precisão das análises automáticas, foi realizado um processo de validação cruzada. Uma amostra representativa de *commits* foi avaliada manualmente, e os resultados foram comparados com as avaliações geradas pelo modelo de linguagem. As divergências identificadas permitiram a calibração e o ajuste fino dos *prompts* e dos parâmetros de avaliação.

7. **Implementação de Testes Automatizados:** Em consonância com as práticas recomendadas por Martin (Martin 2008), foram implementados testes automatizados para validar o comportamento dos componentes críticos do sistema. Os testes cobriram regras de negócio do domínio, cálculos de métricas de qualidade e integrações entre camadas, assegurando a confiabilidade e facilitando a manutenção evolutiva do código.

8. **Testes de Desempenho e Portabilidade:** A fim de avaliar a viabilidade do sistema em diferentes contextos de *hardware*, foram realizados testes de desempenho em ambientes locais com configurações distintas. O principal indicador de desempenho mensurado foi o tempo de processamento por análise de *commit*. Este procedimento visou identificar potenciais limitações e validar a portabilidade da solução conteinerizada.

4.3 Tecnologias Utilizadas

A seleção das tecnologias que compõem o sistema proposto foi realizada considerando critérios de desempenho, compatibilidade, robustez e a capacidade de atender aos requisitos funcionais e não funcionais do projeto. Cada componente foi escolhido para cumprir um papel específico na arquitetura da solução, garantindo a eficiência e a escalabilidade do sistema. A Tabela 2 detalha as principais tecnologias e ferramentas empregadas no desenvolvimento do *RefactorScore*, apresentando suas respectivas finalidades.

Tabela 2: Tecnologias utilizadas e suas finalidades no projeto RefactorScore.

Tecnologia	**Finalidade**
.NET 8 Worker Service	Execução automatizada e periódica do serviço de análise de código em segundo plano (*background service*).
LibGit2Sharp	Biblioteca .NET para interação programática com repositórios *Git* locais, permitindo extração de *commits*, metadados e diferenças de código.
Qwen2.5-Coder:7b (Ollama)	Modelo de linguagem local com 7 bilhões de parâmetros, especializado em análise de código segundo princípios de *Clean Code*.
MongoDB	Banco de dados NoSQL orientado a documentos para armazenamento de histórico de análises, métricas de qualidade e sugestões.
xUnit	Framework de testes unitários para validação automatizada de componentes e regras de negócio do sistema .NET.
Node.js e TypeScript	Plataforma e linguagem para desenvolvimento da API REST com tipagem estática e melhor manutenibilidade.
Express.js	Framework minimalista para construção de rotas, *middlewares* e *endpoints* da API HTTP.
Vue 3 (Composition API)	Framework *JavaScript* progressivo para construção da interface *web* reativa utilizando padrão de composição.
Pinia	Biblioteca de gerenciamento de estado global centralizado com suporte nativo ao TypeScript.
Chart.js	Biblioteca *JavaScript* para renderização de gráficos interativos (linha, radar, barras) na visualização de métricas.
Vite	Ferramenta de *build* e servidor de desenvolvimento com *Hot Module Replacement* otimizado.
Axios	Cliente HTTP baseado em *Promises* para comunicação entre *frontend* e API REST.
SCSS (Sass)	Pré-processador CSS com suporte a variáveis, aninhamento e reutilização de estilos.
Docker e Docker Compose	Conteinerização de serviços (*Worker*, API, MongoDB, Ollama) e orquestração de ambientes multi-container.

Fonte: Criado pelos autores.

4.4 Desafios e Soluções Adotadas

O desenvolvimento do sistema de análise de código implicou a superação de diversos desafios de ordem técnica e metodológica. A seguir, detalham-se os principais obstáculos encontrados e as respectivas soluções implementadas para assegurar a viabilidade e a eficácia do projeto.

1. **Gerenciamento do Limite de Contexto do LLM (*Tokens*):** O modelo de linguagem possui um limite máximo de *tokens* (unidades de texto) que pode processar em uma única requisição. Em *commits* que envolviam alterações extensas, esse limite era frequentemente extrapolado, resultando em análises incompletas ou incoerentes. Para mitigar este problema, foi desenvolvida uma estratégia de pré-processamento que segmenta o código de *commits* volumosos. Consecutivamente, foram implementados *prompts* dinâmicos, capazes de enviar apenas as seções mais relevantes do código para a análise, otimizando o uso da janela de contexto.

2. **Validação e Calibração das Análises do Modelo:** Assegurar que os *feedbacks* gerados pelo LLM fossem precisos e consistentes representou um desafio metodológico signifi-

cativo. Durante a fase de validação, identificou-se que o modelo, com sua configuração padrão, tendia a reproduzir análises padronizadas e repetitivas para contextos de código distintos, comprometendo a especificidade da avaliação. Este comportamento foi atribuído a um baixo valor no hiperparâmetro de **temperatura**, que controla a aleatoriedade das respostas. Para corrigir essa deficiência e promover uma maior variabilidade e sensibilidade ao contexto, o valor da temperatura foi ajustado de 0.1 para **0.5**. Esta calibração, somada a um ciclo contínuo de comparação entre as análises automáticas e avaliações manuais, foi fundamental para garantir que os *feedbacks* fossem genuinamente personalizados para cada *commit* analisado.

3. **Desempenho da Análise em Repositórios Volumosos:** Mesmo com o foco em códigos menores, a extração e processamento de dados em repositórios com um longo histórico de *commits* apresentaram gargalos de desempenho. Para garantir a robustez do sistema, foram implementados mecanismos de otimização na leitura dos dados do *Git* e um sistema de *logs* detalhado, que permite o diagnóstico rápido de falhas e a monitorização do tempo de execução de cada etapa da análise.

Essas adaptações foram fundamentais para contornar as limitações de *hardware* e as complexidades inerentes aos LLMs, permitindo a entrega de uma ferramenta funcional e útil para o público-alvo, mesmo em ambientes com recursos computacionais restritos.

4.4.1 Resultados de Desempenho

Dando sequência à caracterização dos ambientes de teste, esta seção apresenta os resultados de desempenho obtidos. A métrica principal avaliada foi o tempo de processamento necessário para que o sistema realizasse uma análise completa de um único *commit*. Os dados coletados, representando a faixa de tempo observada em cada *hardware*, estão consolidados na Tabela 3.

Tabela 3: Tempo de Análise por Commit em Diferentes Configurações de Hardware

Configuração de Hardware	**Tempo de Análise por Arquivo**
Hardware A (Desktop - RTX 3070)	30 segundos a 1 minuto
Hardware B (Notebook - RTX 2060)	30 segundos a 1 minuto
Hardware C (Notebook - GTX 1650)	1,5 a 2 minutos

Fonte: Criado pelos autores.

4.5 Resultados obtidos pelo RefactorScore

A interação do desenvolvedor com o sistema é estruturada em etapas sequenciais que facilitam a navegação e a compreensão dos dados. A seguir, apresenta-se o fluxo completo de utilização da ferramenta, desde a seleção inicial do projeto até a análise detalhada de *commits* específicos.

4.5.1 Seleção de Projeto

Ao acessar o sistema pela primeira vez, o usuário é direcionado à tela inicial de seleção de projeto, ilustrada na Figura 8. Nessa interface, são apresentados todos os projetos que possuem análises disponíveis no sistema, acompanhados de métricas resumidas como o número total de *commits* analisados, a nota média de qualidade e a quantidade de arquivos avaliados. Essa visão panorâmica permite que o desenvolvedor identifique rapidamente qual projeto deseja explorar, facilitando o acesso aos dados históricos de qualidade de código.

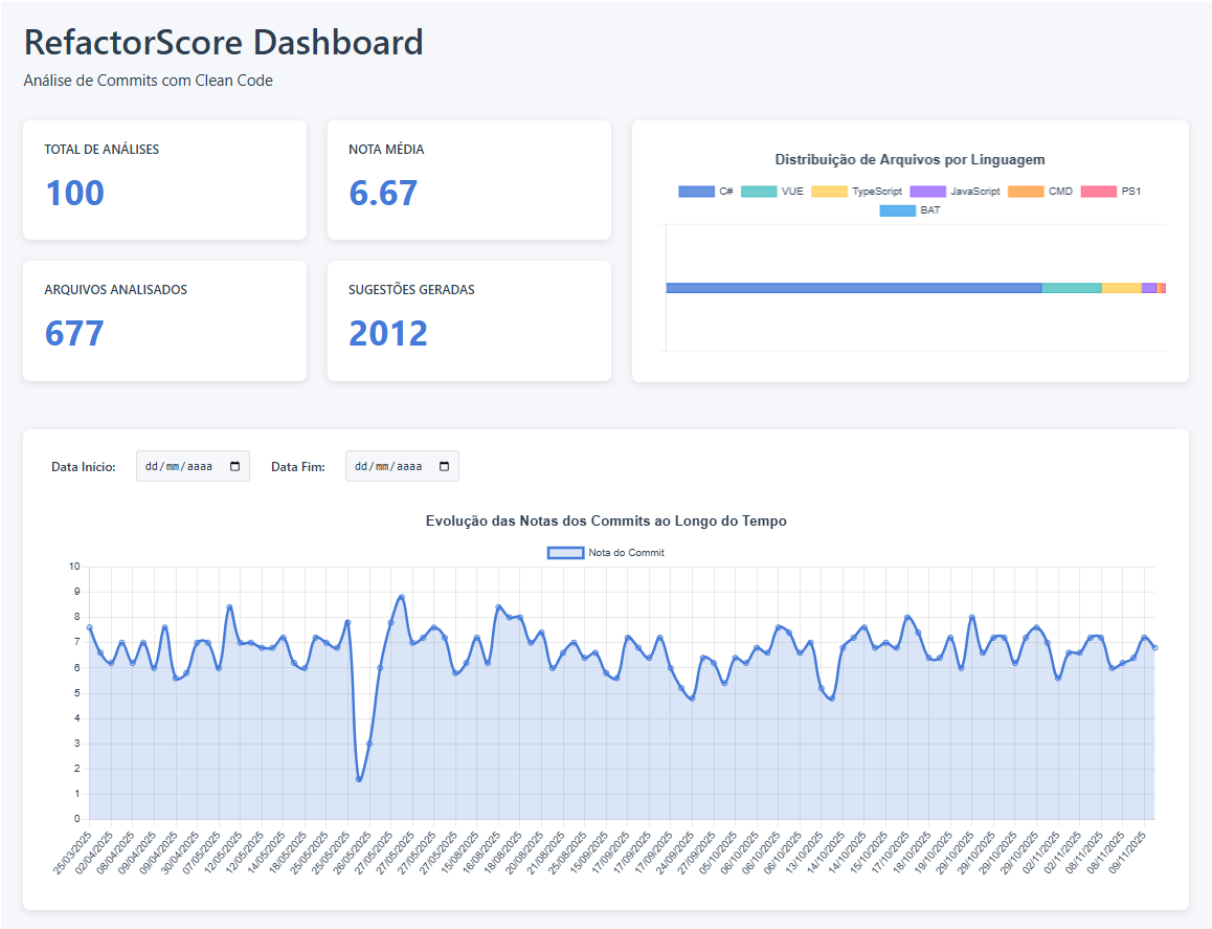
Figura 8: Tela de seleção de projeto

Fonte: Criado pelos autores.

4.5.2 Visão Geral no Dashboard

Após a seleção do projeto, o usuário é direcionado ao painel principal (*dashboard*), apresentado na Figura 9. Essa interface concentra indicadores agregados que oferecem uma perspectiva temporal da evolução da qualidade do código ao longo do tempo. São exibidos gráficos de evolução de notas, distribuição de arquivos por linguagem e métricas consolidadas do projeto selecionado. O *dashboard* funciona como ponto de partida para a navegação mais aprofundada, permitindo que o desenvolvedor identifique tendências, períodos de melhoria ou degradação da qualidade e *commits* específicos que merecem investigação detalhada.

Figura 9: Dashboard do Sistema RefactorScore



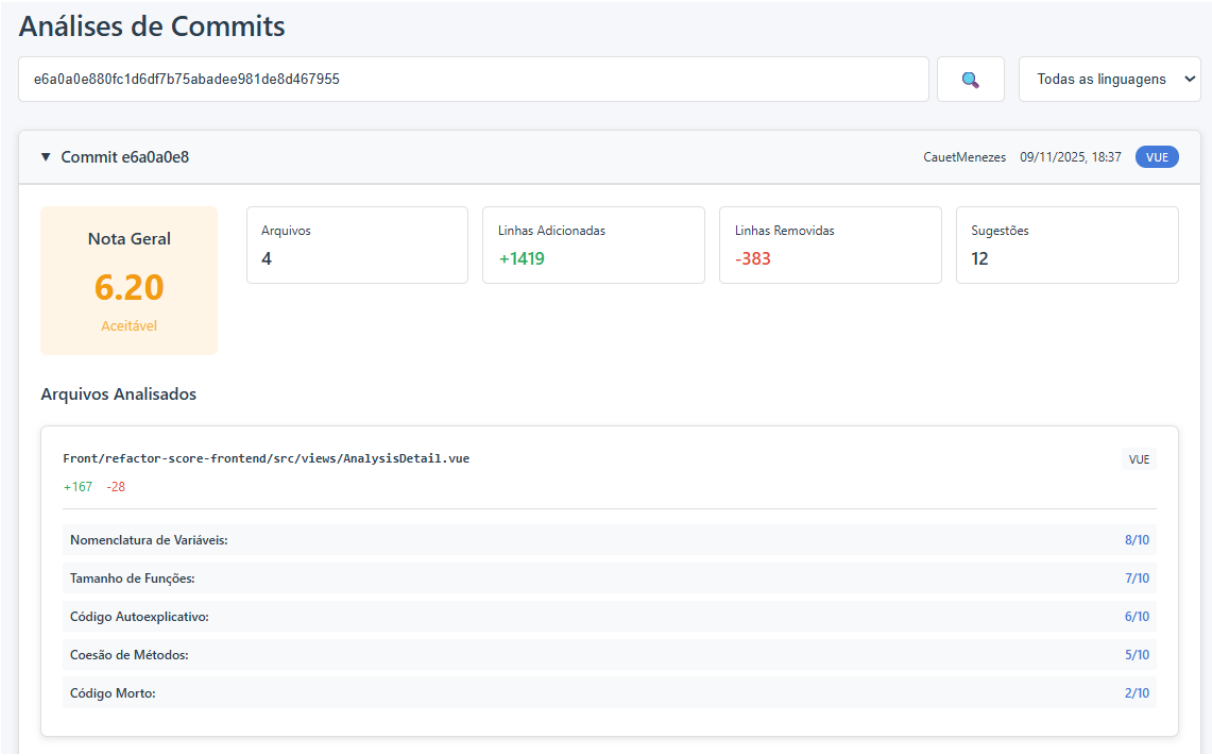
Fonte: Criado pelos autores.

4.5.3 Análise Detalhada de Commit Específico

A partir do *dashboard*, o desenvolvedor pode selecionar *commits* individuais para análise aprofundada ao interagir com o mesmo no gráfico temporal. Para ilustrar o funcionamento dessa funcionalidade, um *commit* realizado no dia 29 de outubro de 2025 foi escolhido como objeto de estudo. Ao acessar a análise detalhada, o sistema apresenta um resumo com as informações-chave do *commit*, oferecendo uma perspectiva geral da qualidade do código modificado.

Esses dados englobam a pontuação geral atribuída pelo modelo de linguagem, o número de arquivos alterados, a soma de linhas inseridas e removidas, e o volume de sugestões de melhoria identificadas durante a análise. Tais indicadores formam um diagnóstico inicial, ajudando o desenvolvedor a compreender de forma rápida a extensão das modificações e seu possível impacto na qualidade do projeto.

Figura 10: Dados gerais e arquivo analisado

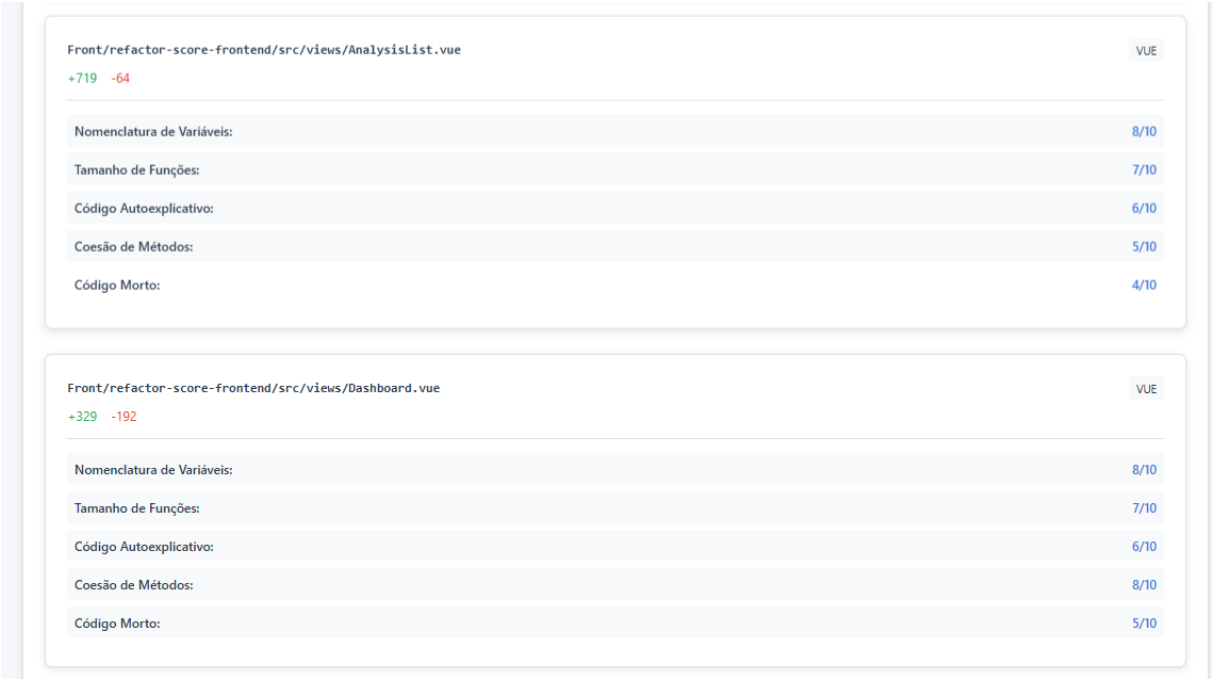


Fonte: Criado pelos autores.

Em seguida, cada arquivo modificado no *commit* é apresentado e examinado em detalhes, como exemplificado nas Figuras 10, 11 e 12. Para cada arquivo, o sistema exibe métricas aprofundadas, evidenciando o número de linhas adicionadas e removidas, o que permite avaliar a complexidade das alterações e identificar tendências como expansões desnecessárias, remoção de código obsoleto ou refatorações estruturais.

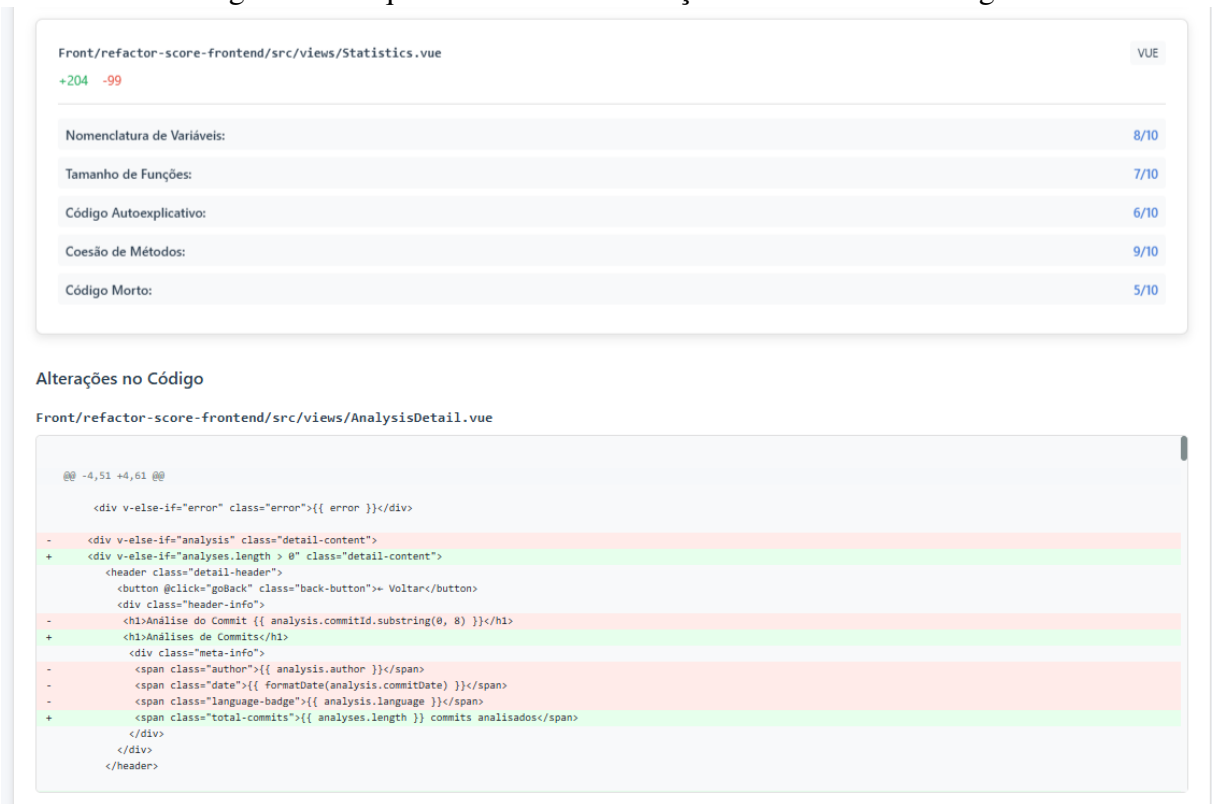
Adicionalmente, o *RefactorScore* atribui notas específicas para cada uma das métricas de *Clean Code* analisadas, incluindo nomenclatura de variáveis, tamanho de funções, clareza e autoexplicabilidade do código, coesão dos métodos e presença de código morto. Tais avaliações auxiliam o desenvolvedor a identificar, de forma objetiva, quais aspectos da implementação necessitam de maior atenção para aprimoramento.

Figura 11: Arquivo analisado



Fonte: Criado pelos autores.

Figura 12: Arquivo analisado e alterações detectadas no código



Fonte: Criado pelos autores.

4.5.4 Visualização de Diferenças de Código (*Diff*)

Ao explorar as modificações incluídas no *commit*, o sistema apresenta as diferenças de código (*diff*), organizando de forma clara quais trechos foram removidos e quais foram adicionados. Essa funcionalidade é ilustrada nas figuras 13 e 14, que demonstram a apresentação visual das alterações no primeiro arquivo associado ao *commit* em análise, permitindo examinar as mudanças de forma minuciosa.

No caso específico do arquivo `AnalysisDetail.vue`, o código corresponde a um componente desenvolvido em *Vue 3*, utilizando *TypeScript*. Esse componente desempenha a função de apresentar as análises dos *commits* processados pelo sistema. A versão anterior permitia visualizar apenas um *commit* por vez. A nova implementação, por sua vez, suporta a apresentação de múltiplos *commits*, incluindo suas respectivas métricas, arquivos modificados, trechos alterados e sugestões geradas pelo modelo de linguagem. Essa evolução estrutural ampliou a capacidade de inspeção e aprofundamento das informações, tornando o processo de análise mais completo e eficiente.

A figura 13 apresenta a modificação realizada na estrutura condicional superior do componente Vue (AnalysisDetail.vue). A visualização foi ajustada para operar com uma lista de análises, em vez de um único objeto. A diretiva v-else-if passa a verificar se existe ao menos um item na lista (analyses.length > 0). O título foi generalizado de "Análise do Commit X"para "Análises de Commits", e as informações específicas de um único commit (como autor e data) foram removidas do cabeçalho principal, sendo substituídas por um contador que indica o total de commits analisados. Além disso, são exibidas as linhas deletadas e inseridas em cada análise.

Figura 13: Exemplo de alteração no código – Alteração no Cabeçalho Principal

Front/refactor-score-frontend/src/views/AnalysisDetail.vue

```
@@ -4,51 +4,61 @@

      <div v-else-if="error" class="error">{{ error }}</div>

-     <div v-else-if="analysis" class="detail-content">
+     <div v-else-if="analyses.length > 0" class="detail-content">
        <header class="detail-header">
          <button @click="goBack" class="back-button">← Voltar</button>
          <div class="header-info">
-           <h1>Análise do Commit {{ analysis.commitId.substring(0, 8) }}</h1>
+           <h1>Análises de Commits</h1>
            <div class="meta-info">
-             <span class="author">{{ analysis.author }}</span>
-             <span class="date">{{ formatDate(analysis.commitDate) }}</span>
-             <span class="language-badge">{{ analysis.language }}</span>
+             <span class="total-commits">{{ analyses.length }} commits analisados</span>
            </div>
          </div>
        </header>
```

Fonte: Criado pelos autores.

Segue uma versão mais clara, técnica e fluida do texto:

Dando continuidade à análise do mesmo arquivo, a figura 14 apresenta a inclusão de um laço de repetição ‘v-for‘ para iterar sobre o array de análises, gerando um contêiner de detalhes individual para cada commit. Além disso, na seção de pontuação (*score-section*), a exibição

da nota geral e da qualidade foi refatorada: em vez de acessar diretamente as propriedades do objeto (por exemplo, ‘analysis.overallNote‘), o código passou a utilizar funções auxiliares — como ‘getOverallNote(analysis)‘ e ‘getFirstFileRating(analysis)‘ — para recuperar esses valores de forma mais estruturada e coerente com a nova arquitetura.

Figura 14: Exemplo de alteração no código – Implementação de Lista e Ajuste de Pontuação

Fonte: Criado pelos autores.

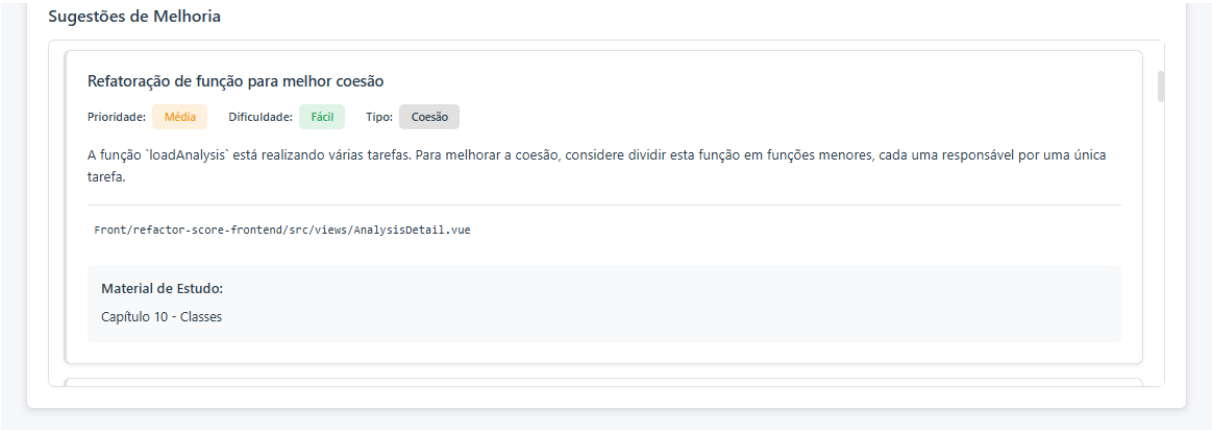
4.5.5 Sugestões de Melhoria e Recursos Educacionais

As figuras 15 e 16 apresentam a seção de Sugestões de Melhoria, que exibe de forma estruturada as recomendações geradas automaticamente pelo modelo de linguagem durante a análise do commit. Cada sugestão é categorizada conforme critérios de prioridade, dificuldade de implementação e tipo de problema identificado, oferecendo uma visão clara, objetiva e orientada para ação sobre os pontos passíveis de aprimoramento.

Além da descrição detalhada de cada recomendação, o sistema indica o arquivo e o contexto específico em que a melhoria se aplica, fundamentando tecnicamente a razão pela qual a sugestão foi identificada. Complementarmente, são fornecidas referências educacionais diretas, incluindo indicações de capítulos específicos da obra *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin 2008), abordando tópicos como *code smells*, princípios de design, coesão e organização de código.

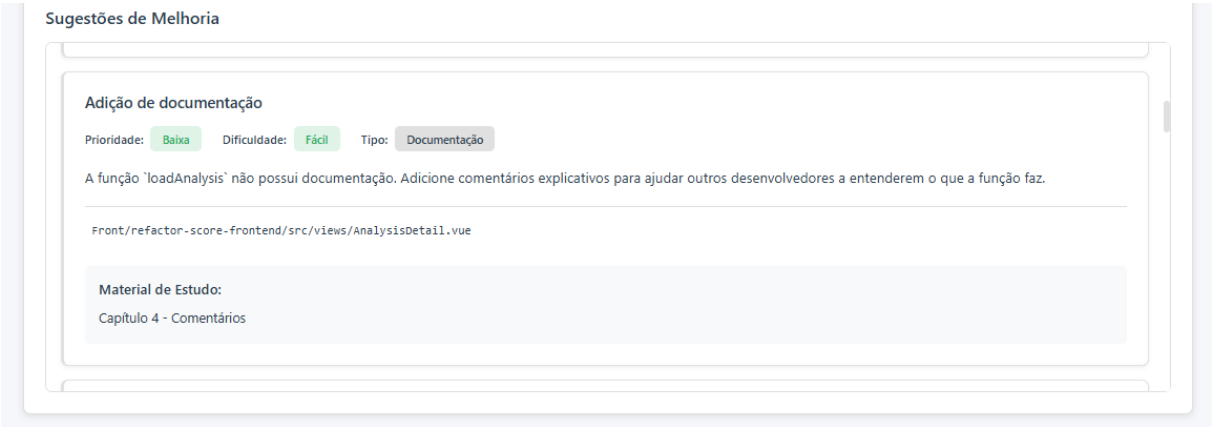
Dessa forma, o desenvolvedor não apenas recebe orientações corretivas pontuais, mas também é estimulado ao aprendizado contínuo, consolidando boas práticas de programação e elevando progressivamente a qualidade técnica de suas implementações.

Figura 15: Sugestões de melhoria — Refatoração de função para melhor coesão



Fonte: Criado pelos autores.

Figura 16: Sugestões de melhoria — Adição de comentários na função



Fonte: Criado pelos autores.

4.5.6 Discussão dos Resultados

A análise dos dados apresentados na Tabela 3 permite extrair conclusões relevantes sobre a viabilidade e os requisitos de hardware do sistema. Os Hardwares A e B, equipados com unidades de processamento gráfico (GPUs) mais robustas da série NVIDIA RTX, apresentaram desempenho similar e significativamente superior, com análises concluídas em aproximadamente um minuto por arquivo. Em contrapartida, o Hardware C, dotado de uma GPU de entrada (GTX 1650), demandou tempo de processamento superior, variando entre 1 minuto e 30 segundos a 2 minutos por arquivo analisado.

Essa diferença de desempenho evidencia a relação entre capacidade de processamento paralelo da GPU e velocidade de inferência do modelo de linguagem. Contudo, mesmo na configuração de hardware mais modesta, o tempo de análise permaneceu em patamar aceitável para a proposta da ferramenta. Considerando que o serviço opera de forma assíncrona e em segundo plano, esses tempos de processamento não interferem no fluxo de trabalho do

desenvolvedor, validando a viabilidade da solução para amplo espectro de usuários, incluindo aqueles com equipamentos de menor capacidade.

Quanto às características gerais dos *commits* examinados, os resultados estão consolidados na Tabela 4. A amostra compreendeu 40 *commits* e 285 arquivos, representando volume significativo de dados para avaliação da qualidade do código. As métricas de qualidade atribuídas pelo modelo de linguagem revelaram consistência notável: a média geral por arquivo foi de 7,65, enquanto a média por *commit* atingiu 7,21, indicando estabilidade nas avaliações independentemente do tamanho ou complexidade dos elementos analisados.

A distribuição das notas revela que 11 *commits* (27,5%) foram classificados como de alta qualidade (nota ≥ 8), enquanto a maioria das contribuições (25 *commits*, ou 62,5%) situou-se na faixa de qualidade média (entre 5 e 8), padrão coerente com ambientes de desenvolvimento real nos quais se espera aprimoramento contínuo. Apenas 4 *commits* (10%) foram categorizados como de baixa qualidade (nota < 5), evidenciando baixa incidência de casos críticos.

Tabela 4: Métricas Gerais dos Commits Analisados

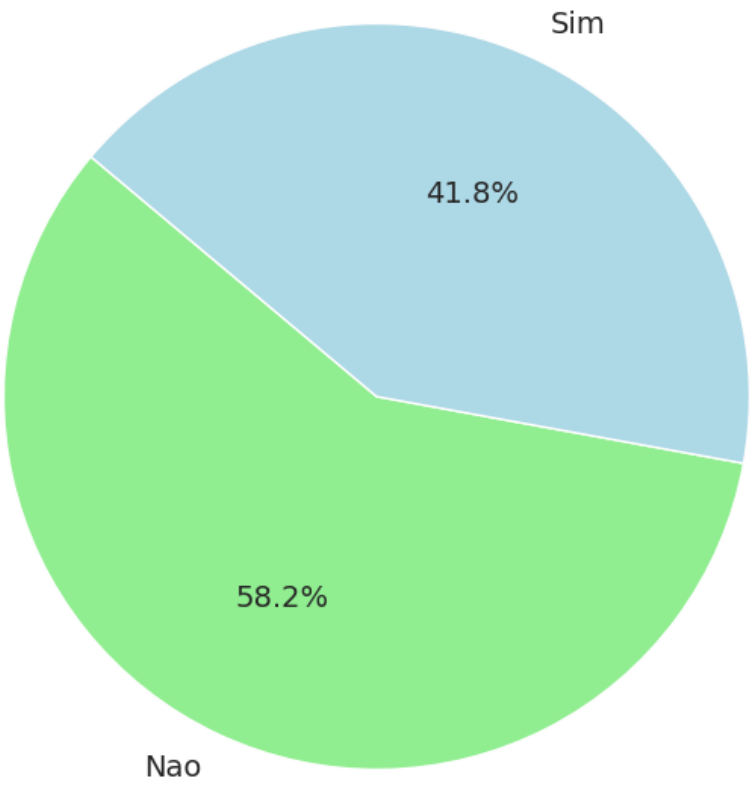
Métrica	**Valor**
Total de Commits	40
Total de Arquivos	285
Média da Nota LLM (por Arquivo)	7,65
Média da Nota LLM (por Commit)	7,21
Commits de Alta Qualidade (≥ 8)	11
Commits de Média Qualidade (5–8)	25
Commits de Baixa Qualidade (< 5)	4
Projetos Únicos	2
Linguagens Únicas	3

Fonte: Criado pelos autores.

A diversidade do contexto analisado, abrangendo dois projetos distintos e três linguagens de programação, reforça a robustez da ferramenta, demonstrando capacidade de avaliação consistente em diferentes cenários tecnológicos.

No tocante à confiabilidade das análises, investigou-se a frequência de alucinações geradas pelo modelo de linguagem. Para tanto, realizou-se validação manual exaustiva, inspecionando individualmente cada análise produzida para a amostra de 40 *commits* (285 arquivos). Cada arquivo foi classificado quanto à presença ou ausência de alucinações, independentemente da quantidade ou tipo. Constatou-se que a ferramenta demonstrou assertividade em 58,2% dos arquivos analisados, com ocorrência de alguma forma de alucinação nos 41,8% restantes, conforme ilustrado na Figura 17.

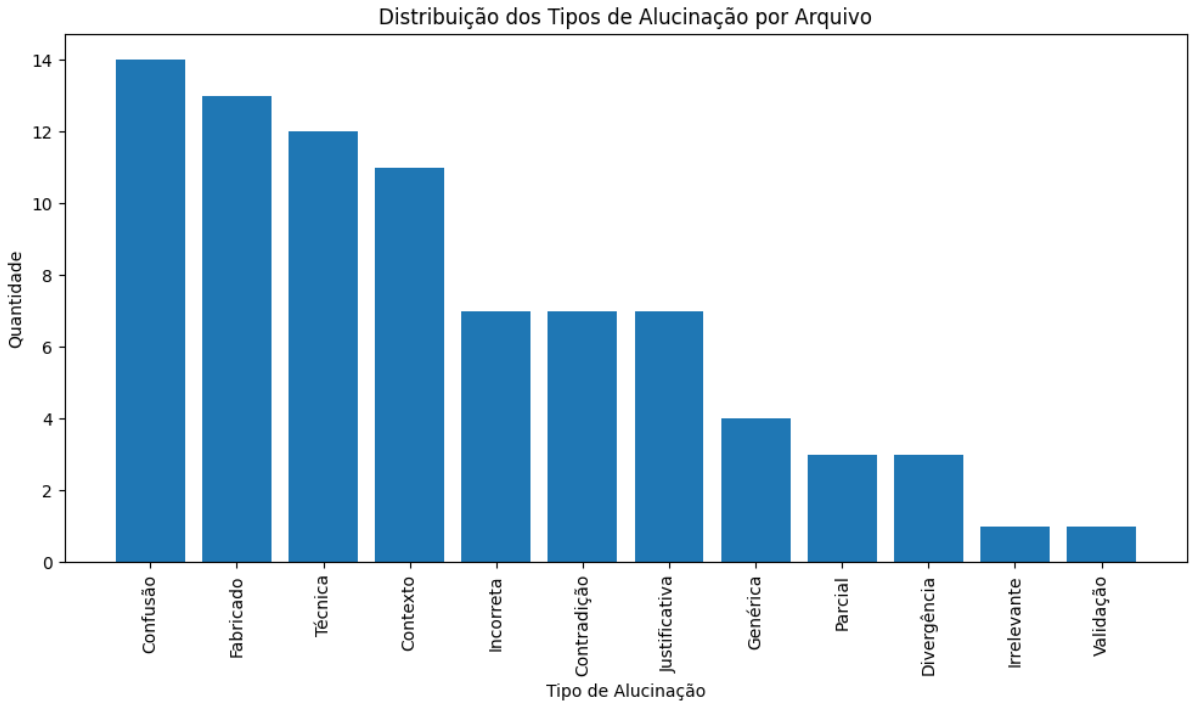
Figura 17: Frequência de Alucinações por Arquivo



Fonte: Criado pelos autores.

É importante ressaltar que a presença de alucinação em um arquivo não implica, necessariamente, na invalidação total da análise. As inconsistências podem manifestar-se em diferentes aspectos da avaliação (notas específicas, justificativas ou sugestões) e em variados graus de severidade. A análise qualitativa permitiu classificar essas inconsistências em categorias distintas, conforme apresentado na Figura 18, revelando os padrões mais frequentes de erros apresentados pelo modelo.

Figura 18: Distribuição dos Tipos de Alucinação por Arquivo



Fonte: Criado pelos autores.

As categorias identificadas foram:

Confusão (14 ocorrências) A análise demonstra confusão sobre o conteúdo, estrutura ou propósito do arquivo. Exemplos incluem confundir variáveis com funções, trocar nomes de entidades, ou interpretar erroneamente a natureza do componente analisado.

Fabricado (13 ocorrências) O modelo cria fatos completos sem base no código real, inventando partes inteiras de código, fluxos de execução, responsabilidades, variáveis ou decisões arquiteturais inexistentes. Representa o tipo mais grave de alucinação.

Técnica (12 ocorrências) O erro envolve conceitos técnicos incorretos, incluindo suposições equivocadas sobre APIs, padrões de projeto, sintaxe ou comportamento esperado do código.

Contexto (11 ocorrências) O modelo introduz contexto inexistente, citando funções, classes, métodos ou relações que não aparecem nas modificações analisadas. Diferencia-se de "Fabricado"por adicionar elementos plausíveis mas ausentes.

Genérica (7 ocorrências) O modelo forneceu justificativas amplas e vagas, não fundamentadas especificamente nas modificações. Embora não sejam tecnicamente falsas, carecem de ancoragem nos elementos concretos do código.

Incorreta (7 ocorrências) A justificativa está objetivamente incorreta em relação ao código, descrevendo algo que não existe ou não ocorreu nas modificações.

Contradição (7 ocorrências) A análise apresenta inconsistência lógica interna, tipicamente entre a nota atribuída e sua justificativa. Por exemplo, atribuir nota alta com justificativa predominantemente negativa.

Justificativa (7 ocorrências) O modelo inventou ou extrapolou a justificativa para um critério específico, criando motivos não presentes no código real.

Parcial (4 ocorrências) A justificativa é parcialmente correta, mas contém pequenos equívocos, interpretações imprecisas ou detalhes irrelevantes. Não compromete totalmente a utilidade da avaliação.

Divergência (3 ocorrências) A justificativa ou nota diverge do conteúdo entre diferentes arquivos do mesmo *commit*, como se o modelo "misturasse"análises distintas.

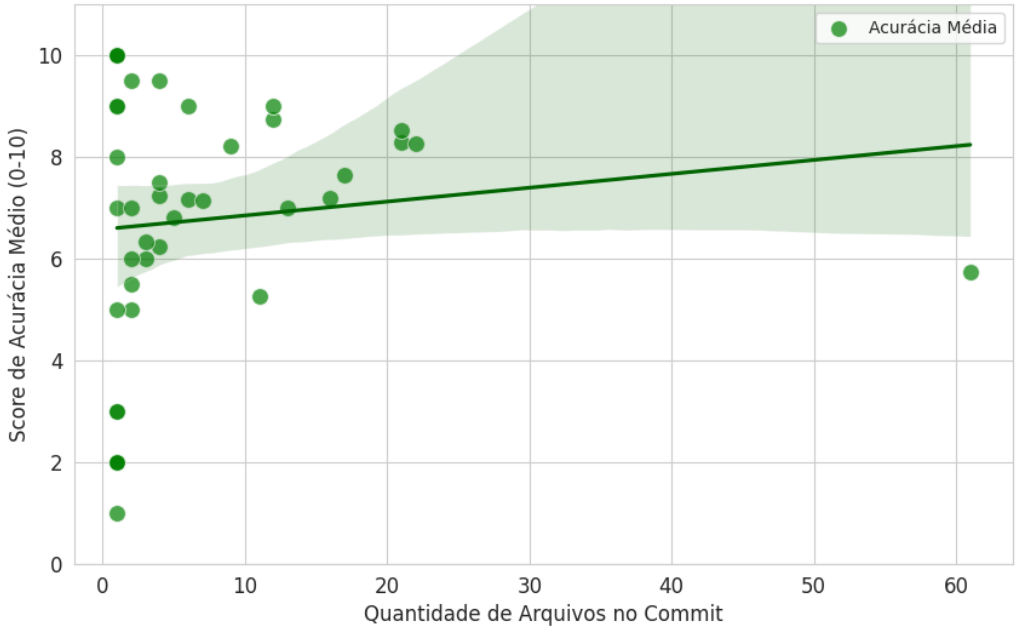
Irrelevante (1 ocorrência) A resposta comenta aspectos que não se aplicam ao tipo de arquivo ou à modificação realizada. Por exemplo, avaliar lógica de negócio em arquivo de configuração.

Validação (1 ocorrência) Erros na interpretação dos próprios critérios de validação. O sistema aplica um critério de *Clean Code* de maneira equivocada ao contexto específico.

A distribuição dessas categorias revela que as alucinações mais frequentes (**Confusão**, **Fabricado** e **Técnica**) correspondem a desafios característicos de modelos de linguagem de médio porte na interpretação de contextos técnicos especializados. Nem todos os tipos comprometem criticamente a utilidade educacional do *feedback* fornecido. Esses resultados estão alinhados com as características esperadas do modelo *Qwen2.5-Coder:7b*, considerando as escolhas arquiteturais voltadas à acessibilidade e execução local.

Complementarmente, investigou-se a relação entre o tamanho do *commit* e a precisão das análises. A Figura 19 apresenta a distribuição da acurácia em função da quantidade de arquivos modificados. Observa-se que *commits* com 1 a 10 arquivos tendem a apresentar acurácia mais próxima da média esperada, com menor dispersão em torno da linha de tendência. Contudo, *commits* com mais de 10 arquivos apresentam maior variabilidade nos resultados, sugerindo que a análise de *commits* extensos pode ser influenciada por fatores adicionais, como a complexidade das modificações, a heterogeneidade das linguagens envolvidas ou a natureza das alterações realizadas.

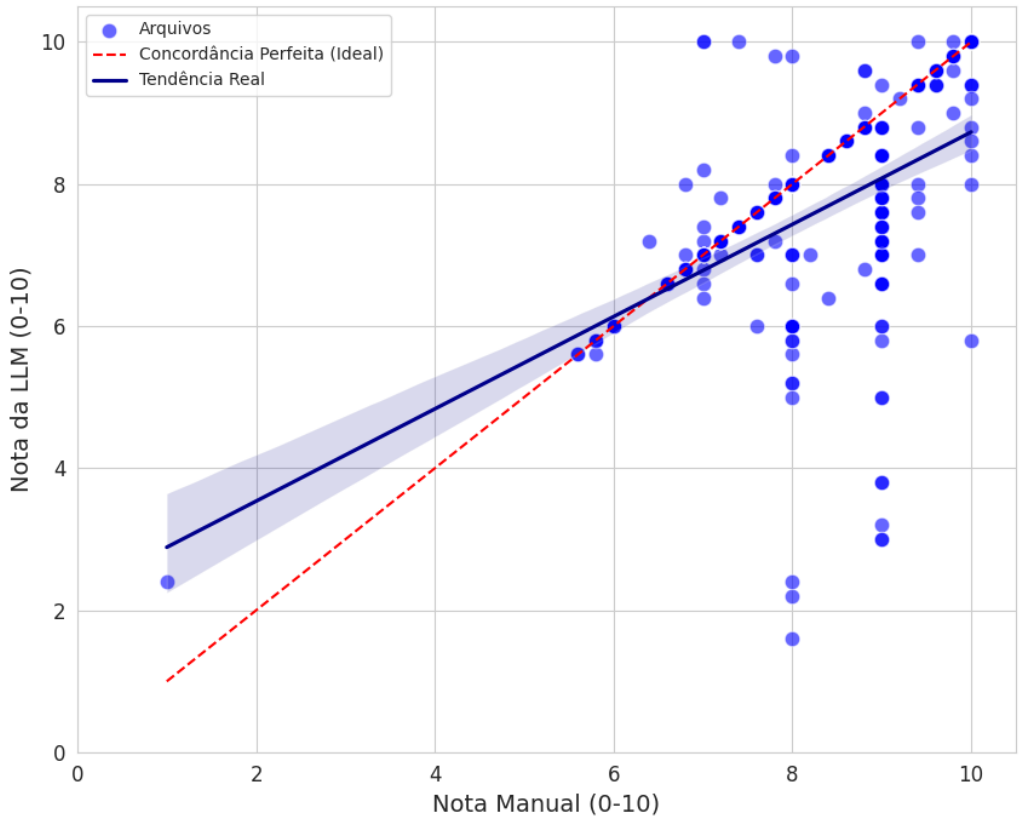
Figura 19: Relação entre Acurácia da Análise e Tamanho do Commit



Fonte: Criado pelos autores.

Por fim, realizou-se análise comparativa entre as notas atribuídas manualmente, fundamentadas nos princípios de *Clean Code*, e as avaliações geradas pelo modelo. A Figura 20 ilustra essa correlação, onde a linha tracejada vermelha representa o cenário de concordância perfeita (100%). Pontos situados acima desta linha indicam que o modelo foi menos rigoroso que o avaliador humano, enquanto pontos abaixo denotam critério mais exigente por parte do modelo. A linha de tendência azul aproxima-se satisfatoriamente da concordância ideal, especialmente na faixa de notas médias a altas (6-10), indicando calibração adequada para o propósito educacional da ferramenta.

Figura 20: Comparação: Nota Manual vs Nota LLM



Fonte: Criado pelos autores.

Em síntese, os resultados quantitativos e de desempenho demonstram que o RefactorScore atinge seus objetivos como ferramenta educacional de análise de qualidade de código, apresentando comportamento estável e oferecendo avaliações consistentes dentro das expectativas estabelecidas pelas escolhas arquiteturais do projeto.

5 CONCLUSÃO

O desenvolvimento de um sistema de monitoramento da qualidade de código, baseado na análise automatizada de *commits*, representa uma contribuição relevante para a prática da engenharia de software, contexto no qual a busca por qualidade e produtividade no desenvolvimento de sistemas é cada vez mais exigida. O RefactorScore foi criado especificamente para auxiliar desenvolvedores iniciantes em sua jornada de aprendizado, capacitando-os a compreender e aplicar os princípios de *Clean Code* de forma prática e contextualizada em seus próprios projetos.

A proposta de utilizar grandes modelos de linguagem (LLMs) como ferramenta de análise representa um diferencial importante do projeto. Ao contrário de abordagens tradicionais baseadas exclusivamente em análise estática de código, a utilização de um LLM permite interpretações mais próximas do raciocínio humano, possibilitando a avaliação qualitativa de aspectos como clareza, legibilidade, modularidade e aderência a boas práticas. No entanto, essa abordagem também apresenta desafios, como a construção de *prompts* eficientes, o gerenciamento da quantidade de *tokens* em entradas extensas e a mitigação de alucinações do modelo, desafios esses que foram abordados ao longo da construção do projeto com estratégias específicas documentadas.

A decisão de manter todo o ecossistema de análise em ambiente local constitui outro aspecto estratégico do projeto. Ao evitar a dependência de servidores remotos ou APIs externas, o sistema garante maior privacidade dos dados analisados, além de aumentar a confiabilidade do processo em contextos de baixa conectividade. A arquitetura proposta também remove barreiras de custo para o uso da ferramenta, uma vez que os modelos são executados localmente via Ollama sem despesas operacionais recorrentes, ficando a critério do usuário definir qual modelo utilizar. Essa decisão favorece, ainda, a democratização do acesso à ferramenta, tornando-a viável para estudantes e profissionais iniciantes com recursos limitados.

A implementação seguindo princípios de *Clean Architecture* e *Domain-Driven Design* proporcionou modularidade e separação clara de responsabilidades, facilitando manutenção e evolução futura do sistema. A execução como serviço de *Worker* do .NET garante robustez, escalabilidade e integração nativa com sistemas operacionais modernos. Seu funcionamento automatizado e assíncrono permite análise contínua sem interrupção do fluxo de trabalho do desenvolvedor, enquanto o *dashboard* implementado amplia o valor da solução ao oferecer visualizações intuitivas da evolução da qualidade do código ao longo do tempo.

A validação experimental conduzida demonstrou que o sistema atinge seus objetivos educacionais propostos. Com uma amostra de 40 *commits* compreendendo 285 arquivos analisados, observou-se taxa de assertividade de 58,2% das análises, valor que se mostrou adequado considerando o público-alvo e o propósito formativo da ferramenta. As inconsistências identificadas nos demais casos, categorizadas em 15 tipos distintos, apresentaram severidades variadas. Importante destacar que nem todas comprometeram criticamente a utilidade educacional do *feedback* fornecido. Esses resultados validam a viabilidade da abordagem adotada, demonstrando que modelos locais e acessíveis, mesmo com limitações inerentes quando comparados a soluções de

grande escala, podem oferecer valor significativo em contextos educacionais.

A arquitetura modular do sistema permite, futuramente, sua adaptação para contextos profissionais mais exigentes. A camada de abstração implementada para o serviço de LLM possibilita substituição do modelo *Qwen2.5-Coder:7b* por alternativas mais robustas, sejam modelos locais de maior porte executados em *hardware* especializado, sejam soluções baseadas em APIs comerciais. Essa flexibilidade arquitetural posiciona o RefactorScore não apenas como ferramenta educacional, mas como base sólida para soluções de análise de qualidade em ambientes empresariais, mediante ajustes adequados no modelo de inteligência artificial utilizado.

Embora o projeto esteja inicialmente limitado à análise de repositórios locais, essa decisão foi tomada estrategicamente para garantir a viabilidade técnica e a estabilidade do sistema em seu primeiro ciclo de vida. Como trabalho futuro, destaca-se a possibilidade de integração com plataformas de versionamento remoto, o que ampliaria significativamente o alcance e a aplicabilidade da ferramenta em ambientes colaborativos de desenvolvimento.

Em síntese, este projeto não apenas propõe uma ferramenta funcional e aplicável no contexto da engenharia de software educacional, como também lança as bases para discussões mais amplas sobre o uso responsável de inteligência artificial no apoio à formação de desenvolvedores. Os resultados obtidos demonstram que é possível conciliar acessibilidade, privacidade e utilidade educacional em ferramentas de análise de código assistidas por inteligência artificial. Ao unir conceitos de engenharia de software, automação e inteligência artificial, a solução apresentada reafirma o potencial transformador da tecnologia na construção de uma base sólida de conhecimento em práticas de desenvolvimento de software de qualidade, especialmente para profissionais em início de carreira.

Referências

Berryman e Ziegler 2024 BERRYMAN, J.; ZIEGLER, A. *Prompt Engineering for LLMs: The Art and Science of Building Large Language Model-Based Applications*. Edição em inglês. Sebastopol: O'Reilly Media, 2024.

Brown et al. 2010 BROWN, N. et al. Managing technical debt in software-reliant systems. *FSE/SDP Workshop*, 2010.

Chen et al. 2021 CHEN, M. et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Evans 2003 EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [S.l.]: Addison-Wesley, 2003.

Fowler 2018 FOWLER, M. *Refactoring: Improving the Design of Existing Code*. [S.l.]: Addison-Wesley, 2018.

MacCormack e Sturtevant 2016 MACCORMACK, A.; STURTEVANT, D. J. Technical debt and system architecture: The impact of coupling on defect-related activity. *Journal of Systems and Software*, Elsevier, v. 117, p. 1–13, 2016.

Martin 2008 MARTIN, R. C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st. ed. Upper Saddle River: Prentice Hall, 2008. (Robert C. Martin Series).

Martin 2017 MARTIN, R. C. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. [S.l.]: Prentice Hall, 2017.