Avaliação de Pagamento de Dívida técnica Automatizado com LLM em Projetos Python

Bernardo Oliveira Pires[1]
[1]Pontifícia Universidade Católica de Minas Gerais (PUC Minas) – Praça da Liberdade
Caixa Postal 14248 – CEP 30535-901 – Belo Horizonte – MG – Brasil
bernardoopires14@gmail.com

1

Abstract. *This study is part of the software engineering field, focusing on code quality and technical debt management. The addressed problem is the uncertainty surrounding the effectiveness of automated technical debt payment using large language models (LLMs), such as GPT-4o, in real-world scenarios of technical debt repayment. The main objective is to evaluate whether GPT-4o can produce technical debt payment that approximates the improvements made by human developers when removing self-admitted technical debt, typically marked by comments such as TODO, FIXME, or HACK. The approach is empirical: popular Python repositories from GitHub are mined, and commits that remove self-admitted technical debts comments are identified using PyDriller. For each case, the code before the commit has its technical debt paid by GPT-4o with predefined zero-shot and few-shot prompts. The resulting code is then compared to the human-written commit using structural metrics (e.g., cyclomatic complexity) and diff-based similarity measures (BLEU-diff, CrystalBLEU-diff, LEMOD). The expected results include practical evidence of GPT-4o's usefulness in software maintenance, as well as a critical assessment of its limitations regarding context awareness, consistency, and alignment with human refactoring practices.*

Resumo. *Este trabalho está inserido na área de Engenharia de Software, com foco na qualidade de código e na gestão da dívida técnica. O problema abordado é a incerteza sobre a efetividade do pagamento de dívida técnica automatizado com modelos de linguagem de grande porte (LLMs), como o GPT-4o, em cenários reais de pagamento de dívida técnica. O objetivo geral é avaliar se o GPT-4o é capaz de produzir pagamentos de dívida técnica que se aproximem das melhorias realizadas por desenvolvedores humanos ao remover comentários de dívida técnica auto-admitida, como TODO, FIXME ou HACK. A abordagem é empírica: repositórios Python populares do GitHub são minerados, e commits que removem as dividas tecnicas auto-admitidas são identificados por meio do PyDriller. Para cada caso, o código anterior ao commit é submetido ao GPT-4o com prompts pré-definidos nos modos zero-shot e few-shot. O código gerado é então comparado ao commit humano utilizando métricas estruturais (complexidade ciclomática) e medidas de similaridade baseadas em diffs (BLEU-diff, CrystalBLEU-diff, LEMOD). Os resultados esperados incluem evidências práticas da utilidade do GPT-4o na manutenção de software, bem como uma análise crítica de suas limitações em termos de compreensão de*

contexto, consistência e alinhamento com práticas humanas de pagamentos de dívida técnica.

Bacharelado em Engenharia de *Software* - PUC Minas
Trabalho de Conclusão de Curso (TCC)

Orientador de conteúdo (TCC I): João Pedro Oliveira Batisteli - 1019080@sga.pucminas.br
Orientador de conteúdo (TCC I): Joana Souza - joanasouza@pucminas.br
Orientador de conteúdo (TCC I): Leonardo Vilela - leonardocardoso@pucminas.br
Orientador acadêmico (TCC I): Cleiton Tavares - cleitontavares@pucminas.br
Orientador do TCC II: (A ser definido no próximo semestre)

Belo Horizonte, 10 de 05 de 2025.

1. Introdução

O uso crescente da inteligência artificial (IA) tem transformado o desenvolvimento de software, especialmente na automação de tarefas como a geração e pagamento de dívida técnica de código [Midolo and Di Penta 2025]. Embora essa automação traga benefícios como agilidade e redução de custos, estudos recentes alertam para um possível efeito colateral da sua adoção: o aumento da dívida técnica, especialmente quando utilizada sem critérios adequados ou ferramentas apropriadas de avaliação contínua da qualidade [Zhang et al. 2022]. Nesse cenário, o pagamento de dívida técnica automatizado com modelos de linguagem como o GPT-4o (um LLM, *Large Language Model*) vêm sendo investigadas como alternativas promissoras para mitigar a dívida técnica ao longo do ciclo de vida do software [Shirafuji et al. 2023].

Apesar do avanço das ferramentas de IA no suporte à engenharia de software, ainda persistem incertezas sobre sua efetividade prática na melhoria da qualidade. Estudos como [Guo et al. 2024] destacam que, embora modelos como o ChatGPT consigam propor alterações estruturais plausíveis, muitas dessas sugestões não resultam em benefícios claros em termos de dívida técnica. Além disso, pesquisas apontam que o pagamento de dívida técnica automatizado pode introduzir inconsistências ou padrões não idiomáticos, dificultando a manutenção [Zhang et al. 2022, Midolo and Di Penta 2025].

Permanece, portanto, incerta a eficácia prática de modelos como o GPT-4o na redução da dívida técnica em códigos escritos em Python quando avaliados por critérios objetivos, como complexidade ciclomática (CC, *Cyclomatic Complexity*). Mesmo quando as modificações propostas são plausíveis, não há garantia de que se aproximem das melhorias de fato aplicadas por desenvolvedores humanos em seus commits. A verificação dessa eficácia é essencial para compreender benefícios e limitações do uso de IA na manutenção e na mitigação da dívida técnica ao longo do ciclo de vida dos projetos.

A dívida técnica impacta diretamente a manutenibilidade e o custo operacional de sistemas de software. Embora ferramentas de IA venham sendo adotadas para apoiar o desenvolvimento, ainda há incertezas sobre sua contribuição real para redução de complexidade e melhoria de sustentabilidade [Guo et al. 2024]. O acúmulo de dívida técnica decorre, muitas vezes, de decisões rápidas e tende a comprometer a evolução do sistema, exigindo soluções proativas para sua gestão [Zhang et al. 2022]. Nesse contexto, a automação apoiada por IA é promissora; entretanto, sem critérios claros pode introduzir

comportamentos inesperados e falhas sutis [Midolo and Di Penta 2025]. Avaliar o pagamento de dívida técnica com GPT-4o constitui, assim, questão de qualidade de software e de gestão eficiente de recursos.

Este trabalho tem como objetivo geral avaliar a eficácia do modelo GPT-4o na automação do pagamento de Dívida Técnica Auto-Admitida (SATD, *Self-Admitted Technical Debt*) em projetos Python, e sua proximidade em relação ao pagamento de dívida técnica humano, observadas em commits. O estudo é conduzido de forma empírica: são identificados commits em repositórios populares que removem comentários SATD (`TODO`, `FIXME`, `HACK`); para cada caso, o código anterior é submetido ao GPT-4o com *prompts* pré-definidos nos modos *zero-shot* e *few-shot*, e o código gerado é comparado ao commit real. Os objetivos específicos são: (i) identificar e aplicar métricas objetivas para mensurar a qualidade estrutural do código antes e depois do reparo de dívida técnica, focando na **Complexidade Ciclomática (CC)** e nas **Linhas de Código (LoC, SLoC, e LLoC)**. Para comparar a aderência e o impacto do reparo em nível de *diff*, são utilizadas **BLEU-diff**, **CrystalBLEU-diff** (com atenuação de *n-grams* sintáticos) e **LEMOD** (similaridade estrutural em linha). (ii) avaliar a capacidade do GPT-4o em propor pagamentos de dívida técnica eficazes e próximos dos realizados por humanos a partir de trechos com SATD; e (iii) comparar, de forma quantitativa e qualitativa, pagamentos de dívida técnica automatizados e pagamentos de dívida técnica humanos observados nos commits.

Como resultados esperados, pretende-se verificar se o pagamento de dívida técnica automatizado com GPT-4o se aproxima dos pagamentos de dívida técnica aplicados por desenvolvedores humanos em cenários reais. A avaliação é realizada por meio de métricas estruturais (complexidade ciclomática) e métricas baseadas em *diff* (BLEU-diff, CrystalBLEU-diff e LEMOD), buscando identificar tanto ganhos quanto limitações do modelo em termos de compreensão de contexto e alinhamento com práticas humanas. Para garantir reprodutibilidade, os *prompts* utilizados são padronizados e apresentados no trabalho.

Em relação ao restante do conteúdo, a Seção 2 apresenta o referencial teórico; a Seção 3 discute os trabalhos relacionados; e a Seção 4 descreve os materiais e métodos utilizados no estudo.

2. Fundamentação Teórica

2.1. Dívida Técnica

O conceito de dívida técnica foi introduzido por Ward Cunningham em 1992 como uma metáfora para descrever as consequências de decisões apressadas no desenvolvimento de software, comparando-as a uma dívida financeira: soluções rápidas podem acelerar a entrega, mas geram custos futuros de manutenção e retrabalho [Kruchten et al. 2012].

Martin Fowler (2009) propôs uma categorização da dívida técnica em quatro quadrantes, com base em dois eixos: intencionalidade (deliberada ou inadvertida) e prudência (prudente ou imprudente). Essa classificação tem o objetivo de auxiliar na compreensão das motivações por trás da introdução da dívida técnica e orientar estratégias adequadas para sua gestão [Fowler 2009]. Os quadrantes são descritos da seguinte forma:

- **Deliberada e prudente**: ocorre quando a equipe técnica, ciente das implicações, decide assumir a dívida temporariamente, planejando corrigi-la mais adiante. Por

exemplo, optar por uma solução mais rápida para cumprir um prazo de entrega, sabendo que o código será revisado depois.
- **Deliberada e imprudente**: representa dívidas assumidas conscientemente, mas sem avaliar adequadamente os riscos ou sem intenção real de pagamento. Um exemplo seria ignorar testes automatizados para acelerar a entrega, mesmo sabendo do risco de bugs futuros.
- **Inadvertida e prudente**: resulta de decisões tomadas com boas intenções, mas que levam à dívida por limitações de conhecimento técnico ou contexto incompleto, com disposição posterior para corrigir.
- **Inadvertida e imprudente**: dívidas inseridas por desconhecimento, descuido ou falta de experiência, geralmente sem percepção clara dos impactos negativos.

Essa tipologia foi amplamente adotada por pesquisadores como uma base conceitual para entender as origens da dívida técnica e é referenciada por trabalhos como o de Kruchten et al. (2012), que defendem a importância de classificar e tornar visíveis esses diferentes tipos de dívida para facilitar sua gestão ao longo do ciclo de vida do software [Kruchten et al. 2012]. Essa taxonomia foi posteriormente reconhecida e utilizada em estudos acadêmicos, como o mapeamento sistemático de Li et al. (2015), que relaciona essas categorias com abordagens de gestão de dívida técnica.

Um subtipo importante é a *dívida técnica auto-admitida* (SATD, do inglês *Self-Admitted Technical Debt*), em que os próprios desenvolvedores registram a existência de dívida por meio de comentários como *TODO*, *FIXME* e *HACK*. Esses marcadores oferecem uma oportunidade única para estudos empíricos, pois permitem identificar de forma explícita pontos do código em que a equipe reconheceu a necessidade de melhoria. O pagamento dessa dívida pode ser observado em commits que removem tais comentários, servindo como fonte de dados para pesquisas que investigam práticas reais de manutenção.

As práticas recomendadas na gestão da dívida técnica incluem:

- Identificação, monitoramento e prevenção da dívida nas etapas de desenvolvimento [Li et al. 2014];
- Uso de ferramentas de análise estática e métricas de qualidade para identificar *code smells* e violações de padrões de design, com destaque para as mais de 29 ferramentas mapeadas na revisão de Li et al. (2014);
- Inclusão de itens relacionados à dívida técnica (como refatoração e revisão de código) no backlog e no planejamento de releases, conforme proposto por Kruchten et al. (2012) [Kruchten et al. 2012];
- Aplicação contínua de refatorações com apoio de ferramentas ou práticas sistemáticas de desenvolvimento, destacada por Halepmollasi e Tosun (2022), que identificam a refatoração como a estratégia mais usada para "pagar" a dívida técnica por meio da remoção de code smells [Halepmollasi and Tosun 2022].

Esse conjunto de estratégias contribui para manter o código sustentável ao longo do tempo e reduzir o impacto negativo da dívida técnica nos projetos de software.

2.2. Limitações da Inteligência Artificial na Engenharia de Software

Modelos de linguagem como o GPT-4o são poderosos aliados na automação de tarefas de engenharia de software, mas apresentam limitações importantes. Em primeiro lugar, o

modelo funciona como uma *black box*, ou seja, o raciocínio por trás das respostas geradas não é transparente [Guo et al. 2024]. Isso dificulta a validação das refatorações geradas.

Além disso, esses modelos podem cometer erros sutis, como sugerir mudanças que alteram o comportamento funcional do código, mesmo que mantenham a sintaxe correta. Guo et al. (2024) destacam que esse tipo de falha pode comprometer seriamente a confiabilidade da refatoração gerada por LLMs. Outra limitação recorrente é a falta de contexto: como o modelo responde a partir de trechos isolados, ele não compreende o projeto como um todo, o que pode levar à perda de dependências importantes ou à violação de regras arquiteturais [Midolo and Di Penta 2025].

A qualidade dos prompts utilizados também é um fator crítico. Estudos mostram que a estrutura e a clareza do prompt influenciam diretamente a qualidade do código gerado [Shirafuji et al. 2023]. Estratégias como *zero-shot prompting* (sem exemplos) e *few-shot prompting* (com exemplos) impactam diretamente a legibilidade, a eficiência e a adesão às boas práticas no código gerado por modelos de linguagem. Shirafuji et al. (2023) mostram que a inclusão de exemplos melhora significativamente a qualidade estrutural do código refatorado, reduzindo complexidade e aumentando a manutenibilidade [Shirafuji et al. 2023].

Por fim, modelos como o GPT-4o podem gerar código com padrões não idiomáticos, ou seja, que funcionam, mas não seguem o estilo convencional da linguagem, o que prejudica a manutenção a longo prazo [Midolo and Di Penta 2025]. Esses fatores reforçam a importância de avaliar os resultados com métricas objetivas, como complexidade ciclomática e índice de manutenibilidade, bem como com medidas de similaridade em nível de diff, como BLEU-diff, CrystalBLEU-diff e LEMOD, para comparar refatorações humanas e automáticas.

2.3. LLMs e manutenção

Modelos de linguagem de grande porte (LLMs) vêm sendo empregados em tarefas de manutenção e evolução de software, incluindo migração de bibliotecas, modernização de APIs e ajustes de compatibilidade. Resultados iniciais sugerem que LLMs são capazes de propor transformações estruturais úteis em cenários reais, desde que haja um desenho de *prompts* adequado e validação posterior das sugestões [Almeida et al. 2024]. Em particular, a migração de bibliotecas é um caso de uso relevante por envolver mudanças concisas, porém sensíveis ao contexto: atualizações de chamadas, adaptação de assinaturas de métodos e ajustes de dependências exigem preservação de comportamento e aderência a padrões idiomáticos. Esses achados reforçam o potencial de LLMs como apoio à engenharia de manutenção, ao mesmo tempo em que evidenciam a necessidade de avaliação sistemática dos artefatos gerados.

No âmbito específico da dívida técnica auto-admitida (SATD), estudos recentes investigam a efetividade de LLMs na "quitação" dessa dívida, isto é, na geração de modificações que removem *TODO*/*FIXME*/*HACK* acompanhadas de melhorias reais no código. A evidência atual aponta que os modelos podem, em diversos casos, produzir refatorações plausíveis; contudo, os ganhos nem sempre são consistentes, e há risco de introdução de padrões não idiomáticos ou mudanças que afetam o comportamento [Sheikhaei et al. 2025]. Tais resultados indicam a importância de avaliações quantitativas (por métricas) e qualitativas (por inspeção) para aferir o impacto das refatorações auto-

matizadas sobre a manutenibilidade.

2.4. Métricas de qualidade e de similaridade para avaliar pagamento de divída tecnica

s

A avaliação de refatorações humanas ou automatizadas demanda métricas complementares que capturem tanto aspectos estruturais do código quanto a proximidade entre as mudanças propostas e as efetivamente aplicadas em projetos reais.

2.4.1. Métricas Estruturais

A *complexidade ciclomática* (CC) estima o número de caminhos lógicos independentes em funções e métodos, sendo usada como indicador de esforço de teste e dificuldade de compreensão. O *índice de manutenibilidade* (MI) sintetiza sinais de legibilidade e complexidade, servindo como medida resumida da facilidade de modificação e evolução do código. Além disso, detecções de *code smells* ajudam a identificar padrões recorrentes de desenho que sugerem pontos de refatoração (por exemplo, funções longas, muitos parâmetros, ou alta complexidade local). Esse conjunto provê uma visão objetiva do efeito "antes e depois" sobre a qualidade interna do sistema.

2.4.2. Métricas de Similaridade Baseadas em *Diff*

Medidas de similaridade orientadas a *diff* comparam a refatoração produzida por um LLM com a refatoração humana observada no commit. Esta abordagem é fundamental no contexto de Dívida Técnica Auto-Admitida (SATD), pois foca a avaliação estritamente nas linhas de código que foram alteradas, mitigando o ruído de métodos *whole-code* [Sheikhaei et al. 2025].

Exact Match (EM) O EM é a métrica mais estrita, medindo a porcentagem de casos em que o código gerado pelo LLM corresponde **exatamente** ao código de verdade fundamental (*ground truth*). Para garantir a confiabilidade no contexto de código, o cálculo do EM é precedido por um rigoroso pré-processamento que remove *imports*, comentários e docstrings (**ICD**), padronizando a formatação.

BLEU-diff e CrystalBLEU-diff Estas métricas adaptam os algoritmos de similaridade textual *n-gram* para focar apenas no texto das diferenças entre o código de entrada e as versões pós-reparo (candidata e referência). *BLEU-diff* aplica a lógica de *n-gramas* ao texto de mudanças. *CrystalBLEU-diff* é uma variação que atenua a influência de *n-grams* sintaticamente verbosos (como pontuações) na pontuação, focando em termos mais informativos sobre a modificação.

Line-Level Exact Match on Diff (LEMOD) O LEMOD é uma métrica proposta para o reparo de SATD [Sheikhaei et al. 2025] que avalia a qualidade do reparo em termos

de **Precisão**, **Recall** e **F1** em nível de linha. Foca na sobreposição dos conjuntos de linhas modificadas (*diff*) entre a verdade fundamental e o código gerado pelo modelo, oferecendo maior interpretabilidade.

As componentes do LEMOD são definidas como:

- $reference\_diff$: Conjunto de linhas alteradas na verdade fundamental.
- $candidate\_diff$: Conjunto de linhas alteradas no código gerado.

$$\text{LineP} = \frac{\text{count}(\text{intersection}(reference\_diff, candidate\_diff))}{\text{count}(candidate\_diff)} \quad \text{(Precisão em Linha)} \tag{1}$$

$$\text{LineR} = \frac{\text{count}(\text{intersection}(reference\_diff, candidate\_diff))}{\text{count}(reference\_diff)} \quad \text{(Recall em Linha)} \tag{2}$$

$$\text{LineF1} = \frac{2 \cdot \text{LineP} \cdot \text{LineR}}{(\text{LineP} + \text{LineR})} \quad \text{(F1 em Linha)} \tag{3}$$

Em conjunto, tais métricas fornecem sinais complementares: enquanto as estruturais (CC, MI, *smells*) avaliam o impacto na qualidade do código resultante, as métricas de *diff* (EM, BLEU-diff, CrystalBLEU-diff e LEMOD) medem o quão próxima a refatoração automatizada está daquela praticada pelos desenvolvedores no histórico do projeto.

3. Trabalhos Relacionados

Nesta seção, são discutidos trabalhos relacionados ao tema deste estudo, com foco na aplicação de inteligência artificial e modelos de linguagem para refatoração automática de código e redução da dívida técnica em projetos de software, especialmente na linguagem Python. Esses trabalhos tratam de problemas correlatos aos definidos na introdução deste trabalho, e servem de base para comparação e aprofundamento da proposta aqui apresentada.

Uma revisão sistemática conduzida por Binta et al. (2023) realiza uma revisão sistemática da literatura para investigar como técnicas de inteligência artificial podem ser aplicadas na identificação e gestão da dívida técnica em projetos de software. O estudo analisou quinze artigos relevantes e categorizou as principais abordagens de uso da IA, incluindo análise de código, testes automatizados, refatoração, manutenção preditiva e documentação automática. Os autores destacam que ferramentas como SonarQube e CAST apresentam potencial significativo, mas ainda enfrentam desafios relacionados à necessidade de dados de alta qualidade e às implicações éticas do uso de algoritmos automáticos. Este artigo oferece uma base teórica valiosa sobre as possibilidades e limitações do uso de IA na redução da dívida técnica, servindo como apoio para o presente estudo, que avança ao aplicar experimentalmente um modelo de linguagem natural (GPT-4o) na refatoração de código Python.

Em uma abordagem voltada à melhoria da legibilidade e aderência a boas práticas em Python, Midolo e Di Penta (2025) apresentam uma abordagem empírica de refatoração

automática de código Python não idiomático, utilizando heurísticas baseadas na árvore sintática abstrata (AST) para avaliar a eficácia do modelo *GPT-4o* nesse processo. O estudo tem como objetivo melhorar a legibilidade e aderência a boas práticas da linguagem Python por meio da substituição automatizada de construções não idiomáticas. Os resultados obtidos pelos autores, demonstraram que o *GPT-4o* foi capaz de propor 28,8% mais refatorações que o benchmark manual e obteve uma taxa de correção de 90,7%, superando os 76,8% do método anterior. Os resultados do estudo demonstram o potencial dos modelos de linguagem na melhoria da estrutura do código de forma automática. O presente trabalho se relaciona a essa pesquisa ao também utilizar o *GPT-4o* como ferramenta de refatoração, mas com foco adicional na análise da redução da dívida técnica, utilizando métricas quantitativas para mensurar os efeitos das modificações aplicadas ao código.

Zhang et al. (2022) propuseram uma ferramenta automática para refatorar código Python com base em nove padrões idiomáticos da linguagem, definidos a partir da comparação entre a gramática de Python e a de Java. A metodologia consiste na identificação de construções não idiomáticas por meio de padrões sintáticos e na aplicação de operações de reescrita sobre a árvore sintática abstrata (AST) para transformá-las em versões idiomáticas. A avaliação foi conduzida em mais de 10600 repositórios do GitHub, apresentando alta acurácia na aplicação dos padrões e alcançando uma taxa de aceitação de 60% por parte dos desenvolvedores nas *pull requests* enviadas. Em termos de desempenho por tipo de refatoração, a ferramenta obteve melhor resultado no padrão de list comprehension, com recall de 0,66, demonstrando boa precisão, mas deixando de identificar parte dos trechos não idiomáticos. Este trabalho se relaciona ao presente estudo por também tratar da refatoração automática em Python; no entanto, enquanto Zhang et al. focam na padronização sintática e idiomática, este trabalho avalia o impacto das refatorações geradas por modelos de linguagem na redução da dívida técnica, especificamente com o uso do *GPT-4o*.

Outra abordagem relevante foi apresentada por Shirafuji et al. (2023), que propõem o uso do *GPT-3.5* para refatorar automaticamente códigos Python, com foco na redução da complexidade ciclomática. A abordagem utiliza *few-shot prompting*, com exemplos cuidadosamente selecionados para orientar o modelo a gerar versões menos complexas dos programas. A avaliação foi conduzida com 880 programas reais extraídos do Aizu Online Judge. Os resultados mostram que o modelo produziu refatorações corretas em 95,68% dos casos, considerando como base de comparação a versão original dos programas. Em média, as versões geradas apresentaram uma redução de 17,35% na complexidade ciclomática e de 25,84% no número de linhas de código, evidenciando o potencial do modelo em gerar versões estruturalmente mais simples sem comprometer a funcionalidade.

Por fim, Guo et al. (2024) conduzem um estudo empírico para avaliar a eficácia do ChatGPT no refinamento automatizado de código com base em revisões humanas, utilizando métricas como similaridade sintática, qualidade percebida e número de modificações realizadas. O modelo foi testado em novos conjuntos de dados e comparado à ferramenta CodeReviewer como benchmark. Os resultados indicam que o ChatGPT superou o CodeReviewer, alcançando um EM-trim de 22,78 e um BLEU-trim de 76,44. Este trabalho se relaciona ao presente estudo por também empregar modelos de linguagem natural na refatoração de código, enquanto o presente trabalho avança ao focar especifica-

mente na redução da dívida técnica em Python, utilizando métricas quantitativas clássicas da engenharia de software.

4. Materiais e Métodos

Tipo de pesquisa e ambiente. O presente estudo caracteriza-se como pesquisa aplicada, de natureza experimental e quantitativa. A eficácia de pagemtnos de dívida técnica automatizados com LLM (GPT-4o) na redução de dívida técnica auto-admitida (SATD) em projetos Python de código aberto é investigada em ambiente local com Python 3.11, *PyDriller* (mineração), *Radon* (métricas estáticas), scripts próprios para métricas em nível de *diff* (BLEU-diff, CrystalBLEU-diff e LEMOD), Windows 11, VS Code e acesso a LLM via API. Para reprodutibilidade e eficiência, utilizou-se **cache local de repositórios** (evitando re-download) e registro de *hashes* de commits e versões das ferramentas.

Procedimento metodológico. A metodologia é organizada em seis etapas, seguindo coleta *all-in* (sem filtros na mineração) e curadoria pós-coleta, com amostragem determinística.

1. **Seleção dos repositórios.** Repositórios públicos com predominância de Python foram obtidos via API GraphQL do GitHub e armazenados em CSV (nome, URL, estrelas, distribuição de linguagens). A seleção visou amplitude de projetos ativos em Python, preservando a diversidade do ecossistema.
2. **Identificação de SATD (coleta all-in, sem filtros).** A árvore de commits foi percorrida com *PyDriller*, extraindo-se *todos* os *diffs* que **removem** tokens de SATD (`TODO`/`FIXME`/`HACK`).
3. **Extração do código *antes/depois*.** Para cada *diff* com remoção de SATD, foram salvos os arquivos completos correspondentes à revisão anterior (`before.py`) e posterior (`after.py`) ao commit, bem como as coordenadas do *hunk* em que a SATD foi removida. A partir dessas coordenadas, foram extraídos recortes locais de código (*hunk* acrescido de um pequeno contexto de linhas anteriores e posteriores), tanto para o pagamento humano quanto para as variantes geradas pela LLM.
4. **Amostragem e curadoria manual dos pares.** A partir do conjunto total de pares coletados (6968), foi sorteada uma amostra de **365** pares (`pair_id`) para inspeção manual. Cada par foi analisado considerando três critérios: (i) presença efetiva de um comentário de SATD na versão *before*; (ii) o commit humano de fato paga a dívida técnica (isto é, a alteração é relevante para a SATD); e (iii) o pagamento da dívida está restrito ao próprio arquivo, sem depender de modificações em múltiplos arquivos. Após essa curadoria, **151** pares permaneceram como válidos para as etapas seguintes de geração com LLM e cálculo de métricas.
5. **Pagamento da dívida técnica com LLM (duas variantes).** Para cada *antes*, foram geradas duas versões *depois-LLM*: (i) **zero-shot** (instruções objetivas) e (ii) **few-shot** (com exemplo curto) padronizadas.
6. **Avaliação das refatorações.** As variantes, *human_after* (arquivo após o pagamento da dívida técnica feito pelo humano no commit), *after_ai_zero* e *after_ai_few* (arquivos com o pagamento gerado pela LLM) são avaliadas por meio de métricas estruturais e de similaridade em relação ao humano: com o *Radon* são calculadas, para cada recorte local de código, a complexidade ciclomática

média (avg_cc), a distribuição A–F por função/método (cc_A–cc_F) e as contagens de loc, sloc e lloc, a partir das quais se obtêm as diferenças estruturais entre o pagamento humano e cada variante automatizada (d_avg_cc, d_cc_A–d_cc_F–, d_loc, d_sloc, d_lloc); em nível de *diff* e de linha, a similaridade das mudanças em relação ao humano é medida para cada par de variantes (variant_pair) por meio de exact_match, **BLEU-diff**, **CrystalBLEU-diff** (BLEU com *stoplist* de tokens frequentes), **LEMOD** (Levenshtein normalizado sobre *diffs*); por fim, em um subconjunto de pares, essas evidências quantitativas são complementadas por uma análise qualitativa das edições, verificando se o pagamento da dívida técnica pela LLM é de fato coerente com o pagamento humano ou apenas superficial.

Métricas utilizadas. foram adotados quatro grupos: (i) **Complexidade Ciclomática** (média por arquivo e distribuição A–F por função/método, via Radon); (ii) **Tamanho** do código (**SLOC/LLOC**, Radon *raw*); (iii) **Similaridade de *diff*** entre LLM e humano (**BLEU-diff**, **CrystalBLEU-diff** com *stoplist*, e **LEMOD** — Levenshtein normalizado em *diffs*). Todas as métricas são calculadas sobre **arquivos coletados nas etapas anteriores** por variante.

Instrumentos e reprodutibilidade. Os scripts de mineração, geração e avaliação, bem como os *prompts* padronizados e parâmetros, foram organizados em repositório público. Cada linha dos CSVs referencia pair_id, repo@commit_hash (link reproduzível ao GitHub) e caminhos em disco (before.py, after.py, after_ai_zero.py, after_ai_few.py); para o humano, disponibiliza-se índice do **AFTER completo do commit**. Versões de ferramentas e *hashes* são registrados para auditoria. Toda curadoria *pós-coleta* (remoções cosméticas e deduplicações/substituições) é acompanhada de regras, contagens e exemplos.

Fluxo metodológico.

5. Resultados

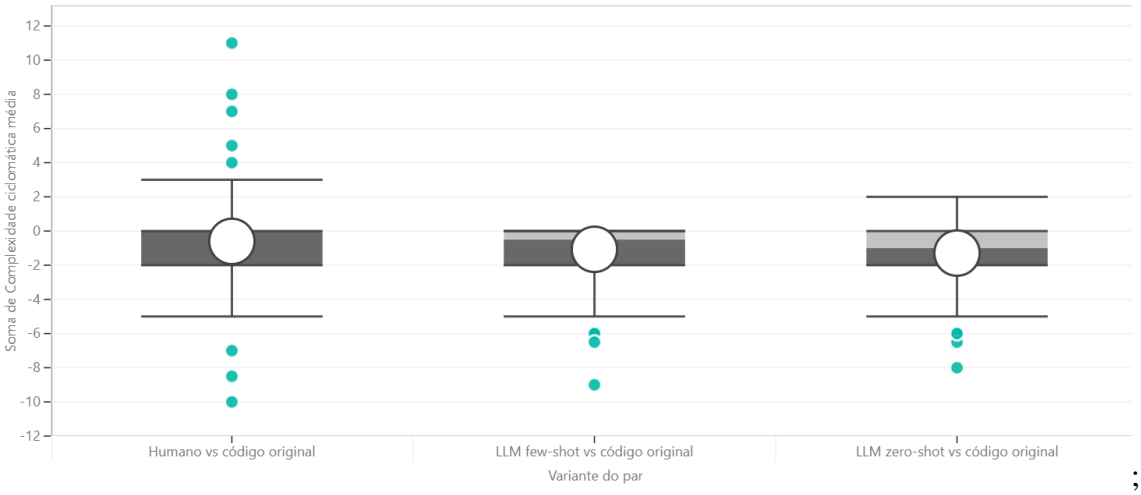
Esta seção apresenta os resultados obtidos a partir das métricas de qualidade estrutural e de similaridade de diff para os 151 hunks de código, comparando o desempenho do GPT-4o (com prompt zero-shot e few-shot) com o ground truth humano.

5.1. Caracterização do Conjunto de Dados

O conjunto de dados utilizado para este estudo é composto por 151 hunks (pedaços de código) de pagamento de Dívida Técnica Auto-Admitida (SATD) extraídos de repositórios Python populares do GitHub.

A coleta de dados foi realizada pela mineração de commits que removeram marcadores de SATD (e.g., TODO, FIXME, HACK), utilizando a ferramenta PyDriller. Para cada caso, foram geradas quatro variantes de código para análise e comparação:

- **Original (Before)**: O código contendo a dívida técnica, antes do pagamento.
- **Humano (Human)**: O código após o pagamento da dívida pelo desenvolvedor humano (ground truth).
- **LLM Zero-Shot**: O código após o pagamento da dívida pelo GPT-4o.
- **LLM Few-Shot**: O código após o pagamento da dívida pelo GPT-4o.



;

indent:

Figura 5.1 - Mediana de F1 LEMOD

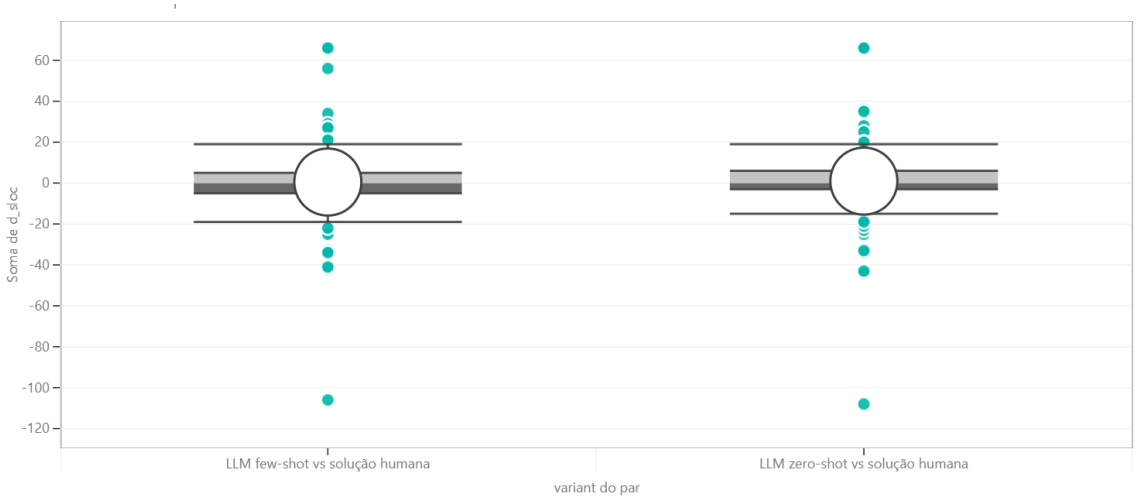
5.2. Similaridade e Alinhamento com o Pagamento de Dívida Técnica Humano

;

Figura 5.2 - Similaridade CrystalBLEU-diff

A similaridade de edição avalia o quanto o processo de refatoração automatizado converge com a prática humana.

F1 LEMOD (Similaridade de Diff): A métrica de similaridade F1 LEMOD, apresentadas na Figura 5.1, que quantifica a sobreposição das linhas de código modificadas, apresentou medianas consistentemente baixas para ambas as variantes LLM (aproximadamente entre $0{,}20$ e $0{,}30$). Este resultado indica que o processo de edição do LLM diverge radicalmente do caminho de alteração adotado pelo desenvolvedor humano, resultando em um baixo alinhamento processual.



indent:
Figura 5.3 - Diferença de Linhas de Código (Δ SLOC)

Similaridade CrystalBLEU-diff (Gráfico 5.3): A métrica CrystalBLEU-diff, detalhada na Figura 5.2, que avalia a similaridade em nível de *n-grams* com atenuação de tokens sintaticamente verbosos, reforça a divergência. As medianas para ambas as variantes LLM (ponto central do círculo) estão na faixa de $0{,}20$ a $0{,}25$. Isso confirma que o conteúdo lexical das alterações feitas pelo LLM é significativamente diferente do conteúdo do commit humano.

Diferença de Linhas de Código (SLOC Diff): A diferença no número de linhas de código alteradas entre o LLM e a solução humana ($\Delta\mathrm{SLOC}$), ilustrada na Figura 5.3, apresentou uma mediana próxima a zero. Contudo, a distribuição é ampla e exibe outliers significativos (acima de $+60$ e abaixo de -100), sugerindo que, em casos específicos, o LLM introduz soluções com uma diferença substancial no tamanho do código modificado em comparação ao humano.

5.3. Impacto na Qualidade Estrutural do Código

Esta subseção foca na variação da Complexidade Ciclomática Média ($\Delta\mathrm{CC}$), presente na Figura 5.4, após a intervenção, em relação à versão original do código.

Variação da Complexidade Ciclomática Média ($\Delta\mathrm{CC}$):

Ação Humana: A mediana da $\Delta\mathrm{CC}$ para a solução humana é próxima de $0{,}0$, indicando que a intervenção humana tendeu a ser conservadora, removendo o marcador SATD sem alterar significativamente a complexidade estrutural subjacente do código.

Ação do LLM: As medianas da $\Delta\mathrm{CC}$ para as variantes LLM (few-shot e zero-shot) são consistentemente negativas (aproximadamente entre $-1{,}0$ e $-2{,}0$). Este achado demonstra que o LLM introduziu alterações que reduziram ativamente a Complexidade Ciclomática Média, indicando uma melhoria estrutural agressiva do código.

Comparação Direta $\Delta\mathrm{CC}$ (LLM vs. Humano): A comparação direta do $\Delta\mathrm{CC}$ entre o LLM e a solução humana, apresentada na Figura 5.5, confirma que a mediana do LLM se localiza abaixo de zero, reforçando que o modelo produziu códigos com menor

indent:

Figura 5.4 - Variação da Complexidade Ciclomática Média (ΔCC) vs Original

Complexidade Ciclomática do que o *ground truth* humano.

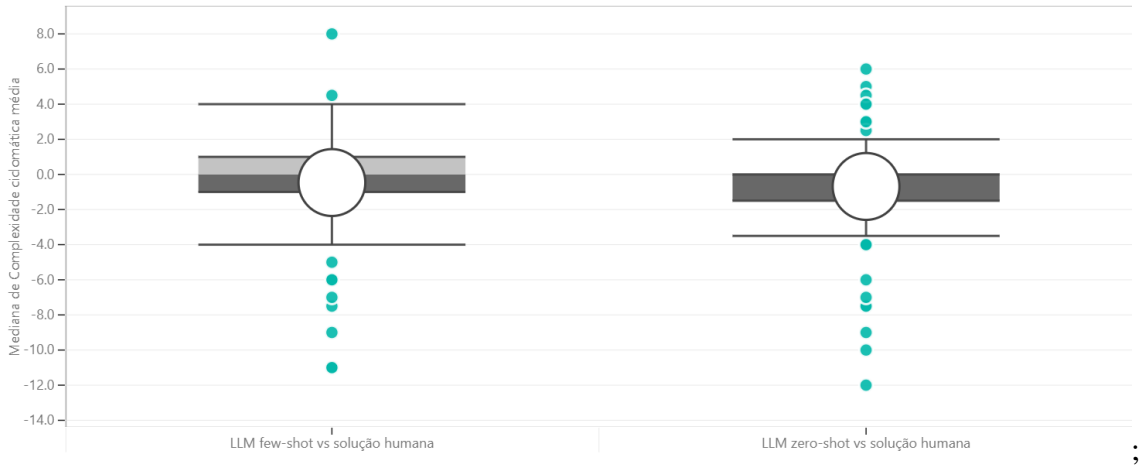
6. Discussão dos Resultados

A discussão dos resultados revela um dilema fundamental na adoção do GPT-4o para o pagamento de dívida técnica: a diferença entre o processo de edição e a qualidade estrutural do código resultante. As métricas de similaridade de *diff*, evidenciadas pela baixa mediana do F1 LEMOD ($0{,}20$ a $0{,}30$) e do CrystalBLEU-diff ($0{,}20$ a $0{,}25$), demonstram um baixo alinhamento processual com o desenvolvedor humano. Isso sugere que o LLM atua como um gerador de soluções alternativas e não como um replicador do commit esperado. Embora a mediana da diferença no número de linhas alteradas ($\Delta\mathrm{SLOC}$) seja próxima de zero, indicando que, em média, o *diff* não é substancialmente maior, a baixa similaridade (F1 LEMOD e CrystalBLEU-diff) significa que as alterações são fundamentalmente diferentes, o que pode aumentar o custo de revisão e introduzir *diffs* inesperados no fluxo de trabalho.

Em contrapartida, o LLM demonstrou ser um agente de melhoria estrutural agressiva, com medianas de $\Delta\mathrm{CC}$ Média consistentemente negativas (entre $-1{,}0$ e $-2{,}0$), superando a ação humana, que foi mais conservadora ($\Delta\mathrm{CC}$ Média $\approx 0{,}0$). Essa diferença implica que o GPT-4o executa um pagamento de dívida técnica oportunista que ativamente simplifica a Complexidade Ciclomática do código. Assim, o GPT-4o é altamente eficaz na otimização da qualidade estrutural, mas o faz através de um processo de edição que é distinto e potencialmente invasivo. Conclui-se que o LLM é valioso como um gerador de soluções de alta qualidade estrutural, mas seu uso exige uma validação humana robusta, principalmente pela ausência de testes funcionais que comprovem a preservação do comportamento do código após a pagamento de dívida técnica agressivo.

7. Conclusões

O objetivo deste trabalho foi avaliar se o Large Language Model (LLM) GPT-4o é capaz de produzir o pagamento de Dívida Técnica Auto-Admitida (SATD) que se aproxima das



;

indent:

Figura 5.5 Comparação Direta da ΔCC entre LLM e Humano

melhorias realizadas por desenvolvedores humanos em projetos Python. Os resultados estabelecem um dilema de eficácia: o LLM falhou em replicar o processo de edição humano (baixa mediana de F1 LEMOD), atuando como um gerador de soluções alternativas. No entanto, o modelo demonstrou ser um agente de melhoria mais agressivo, ultrapassando o escopo mínimo do pagamento da dívida ao reduzir a Complexidade Ciclomática Média em maior grau que o humano (mediana ΔCC entre $-1{,}0$ e $-2{,}0$ para o LLM vs. $\approx 0{,}0$ para o humano). Em síntese, o GPT-4o é uma ferramenta promissora para a melhoria da qualidade estrutural, mas sua baixa aderência ao processo humano exige que seu uso seja complementado por um ciclo de validação rigoroso. A principal contribuição do trabalho é fornecer evidências empíricas sobre o *trade-off* entre a eficácia na otimização de métricas e o risco processual no uso de LLMs para automação de pagamento de dívida técnica.

- **Divergência Processual (Risco)**: As métricas de similaridade de diff (F1 LEMOD) foram baixas, indicando que o LLM gera uma solução alternativa para o diff humano.
- **Superioridade Estrutural (Benefício)**: O LLM demonstrou ser um agente de melhoria mais agressivo, superando a intervenção humana na redução da Complexidade Ciclomática Média (mediana ΔCC entre $-1{,}0$ e $-2{,}0$ para o LLM vs. $\approx 0{,}0$ para o humano).

Em síntese, o GPT-4o é uma ferramenta valiosa para a melhoria da qualidade estrutural, mas sua baixa aderência ao processo humano exige um ciclo de validação rigoroso. A principal contribuição do trabalho é fornecer evidências empíricas sobre o trade-off entre a eficácia na otimização de métricas e o risco processual (divergência de diff) no uso de LLMs para automação de pagamento de dívida técnica.

7.1. Trabalhos Futuros

Com base nas limitações metodológicas e nos resultados obtidos, propomos as seguintes direções para trabalhos futuros:

1. **Verificação Semântica e Testes Funcionais**: É essencial incorporar testes de unidade para os hunks analisados, garantindo que a redução agressiva da Complexidade Ciclomática promovida pelo LLM não tenha introduzido bugs e que o comportamento funcional do código seja preservado.
2. **Engenharia de Prompt Focada no Processo**: Investigar prompts que instruam o LLM a produzir alterações minimamente invasivas e alinhadas com padrões de diff específicos, buscando aumentar a similaridade de diff e a previsibilidade do commit gerado.
3. **Expansão e Validação Externa**: Replicar o experimento em outras linguagens de programação (como Java ou C++) e em diferentes domínios de aplicação para aumentar a validade externa das conclusões.
4. **Avaliação da Aceitação Humana**: Realizar um estudo com desenvolvedores (e.g., survey ou estudo de caso) para avaliar o custo-benefício de revisar um diff de baixa similaridade, mas com alta melhoria de qualidade estrutural (LLM).

8. Pacote de replicaãoo

https://github.com/ICEI-PUC-Minas-PPLES-TI/plf-es-2025-1-tcci-0393100-pes-bernardo-pires/tree/main/Instrumentos/OutrosInstrumentos

Referências

Almeida, A., Xavier, L., and Valente, M. T. (2024). Automatic library migration using large language models: First results. In *Proceedings of Conference acronym 'XX'*, pages 1–7. ACM.

Binta, N., Adnan, M., Salah, K., and Jayaraman, R. (2023). Artificial intelligence for technical debt management in software development. *arXiv preprint arXiv:2306.10194*.

Fowler, M. (2009). Technical debt quadrant. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`. Acessado em 25 jun. 2025.

Guo, Q., Zhang, B., Liu, Y., Chen, X., and Wang, Q. (2024). Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th International Conference on Software Engineering (ICSE 2024)*, pages 1400–1411. IEEE/ACM.

Halepmollasi, R. and Tosun, A. (2022). Exploring the relationship between refactoring and code debt indicators. *Journal of Software: Evolution and Process*.

Kruchten, P., Nord, R. L., and Ozkaya, I. (2012). Technical debt: from metaphor to theory and practice. *IEEE Software*, 29(6):18–21.

Li, Z., Avgeriou, P., and Liang, P. (2014). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220.

Midolo, G. and Di Penta, M. (2025). Automated refactoring of non-idiomatic python code: A differentiated replication with llms. *arXiv preprint arXiv:2501.17024*.

Sheikhaei, M. S., Tian, Y., Wang, S., and Xu, B. (2025). Understanding the effectiveness of llms in automated self-admitted technical debt repayment. *arXiv preprint arXiv:2501.09888*.

Shirafuji, T., Hido, S., Iio, R., and Matsumoto, K.-i. (2023). Refactoring programs using large language models with few-shot examples. *arXiv preprint arXiv:2311.11690*.

Zhang, Z., Xing, Z., Xia, X., Xu, X., and Zhu, L. (2022). Making python code idiomatic by automatic refactoring: Non-idiomatic python code with pythonic idioms. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*, pages 696–708. ACM.