UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM SISTEMAS DE INFORMAÇÃO

THAÍS DE KÁSSIA PEREIRA MÁXIMO

AVALIAÇÃO DE UMA PIPELINE AUTOMATIZADA COM LLMS PARA GERAÇÃO DE TESTES EM UM SISTEMA FINANCEIRO

Recife,
2025
THAÍS DE KÁSSIA PEREIRA MÁXIMO

CURSO DE BACHARELADO EM SISTEMAS DE INFORMAÇÃO

AVALIAÇÃO DE UMA PIPELINE AUTOMATIZADA COM LLMS PARA GERAÇÃO DE TESTES EM UM SISTEMA FINANCEIRO

Trabalho apresentado ao curso de Graduação em Sistemas de Informação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Orientador(a): Paola Rodrigues de Godoy Accioly

Recife

2025

THAÍS DE KÁSSIA PEREIRA MÁXIMO

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Máximo, Thaís de Kássia Pereira.
Avaliação de uma pipeline automatizada com LLMS para geração de testes em um sistema financeiro / Thaís de Kássia Pereira Máximo. - Recife, 2025.
42 : il., tab.

Orientador(a): Paola Rodrigues de Godoy Accioly
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Sistemas de Informação - Bacharelado, 2025.
Inclui referências, anexos.

1. Geração de testes automatizados. 2. LLMs. 3. Automação. 4. Sistemas Financeiros. I. Accioly, Paola Rodrigues de Godoy. (Orientação). II. Título.

000 CDD (22.ed.)

AVALIAÇÃO DE UMA PIPELINE AUTOMATIZADA COM LLMS PARA GERAÇÃO DE TESTES EM UM SISTEMA FINANCEIRO

Trabalho apresentado ao programa de Graduação em Sistemas de Informação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do grau de Bacharel em Sistemas de Informação.

Aprovado em: 13/08/2025

BANCA EXAMINADORA

_____

Prof. Paola Rodrigues de Godoy Accioly (Orientadora)
Universidade Federal de Pernambuco

_____

Prof. Filipe Calegario (Examinador Interno)
Universidade Federal de Pernambuco

RESUMO

A validação de software constitui uma etapa essencial no desenvolvimento de sistemas, especialmente em domínios críticos como o financeiro, onde falhas podem acarretar prejuízos econômicos significativos e sanções regulatórias. Este trabalho propõe e avalia uma pipeline automatizada baseada em Modelos de Linguagem de Grande Porte (LLMs) para geração de testes em um sistema financeiro simplificado. A metodologia adotada compreende o desenvolvimento de um sistema com funcionalidades básicas, como cadastro de usuários e movimentações financeiras entre indivíduos, instituições ou governos, e a implementação de um fluxo de trabalho capaz de identificar trechos de código, gerar prompts, interagir com modelos de linguagem e validar os testes gerados. Foram avaliados os modelos *GPT-4o-mini, Claude Sonnet-4, Qwen2.5-Coder:7b* e *GPT-4.1* quanto ao tempo de execução, cobertura de código, qualidade dos testes, custo computacional e clareza dos casos gerados. Os resultados indicam que, embora os LLMs apresentem potencial para automatizar a validação com qualidade comparável à de testes manuais, sua eficácia depende de fatores como engenharia de prompts, contexto fornecido e modelo adotado. Uma contribuição importante foi a identificação de testes "*dummies*" que inflam artificialmente métricas de cobertura sem agregar valor funcional. Dessa forma, conclui-se que a integração de LLMs em pipelines de teste é viável e promissora, desde que acompanhada de estratégias adequadas de controle, validação e auditoria.

Palavras-chave: validação de software; LLMs; automação de testes; sistema financeiro; pipeline; testes *dummy*.

ABSTRACT

Software validation is a fundamental phase in system development, especially in critical domains such as finance, where failures may lead to significant financial losses and regulatory penalties. This study proposes and evaluates an automated pipeline based on Large Language Models (LLMs) for generating tests in a simplified financial system. The methodology includes the development of a system with core functionalities, such as user registration and financial transactions between individuals, institutions, or governments, and the implementation of a workflow capable of identifying code segments, generating prompts, interacting with language models, and validating the generated tests. The models GPT-4o-mini, Claude Sonnet-4, Qwen2.5-Coder:7b, and GPT-4.1 were assessed in terms of execution time, code coverage, test quality, computational cost, and case clarity. The results show that although LLMs demonstrate potential for automating validation with quality comparable to manual testing, their effectiveness depends on factors such as prompt engineering, provided context, and the model used. An important contribution was the identification of "dummies" tests that artificially inflate coverage metrics without adding functional value. It is concluded that the integration of LLMs into test pipelines is feasible and promising, provided proper strategies for control, validation, and auditing are employed.

Keywords: software validation; LLMs; test automation; financial system; pipeline; dummy tests.

LISTA DE FIGURAS

Figura 1 – Principais casos de uso de IA Generativa..... 14
Figura 2 – Cobertura em cenário de regras de negócio.....20
Figura 3 – Fluxo do sistema..... 24
Figura 4 – Passos de execução do fluxo de trabalho..... 25
Figura 5 – Instruções do fluxo em um Pull Request.....25
Figura 6 – Cobertura de código por modelo.....31

LISTA DE TABELAS

Tabela 1 – Tempo total de execução da pipeline..... 30
Tabela 2 – Resultado das execuções dos testes..... 31
Tabela 2 – Tokens vs Custo Estimado por Execução..... 32

LISTA DE ABREVIAÇÕES

LLM	Large Language Models
ATG	Automated Test Generation
APIs	Application Programming Interface
BSC	Business Scenario Coverage

SUMÁRIO

1. INTRODUÇÃO **11**
1.1. Contexto 11
1.2. Objetivos 13
2. REFERENCIAL TEÓRICO **13**
2.1. Sistemas Financeiros 13
2.1.1 Visão tecnológica 13
2.1.1 IA Generativa em Sistemas Financeiros 15
2.2. Validação de Software 15
2.3. Geração Automática de Testes (ATG) 16
2.4. Modelos de Linguagem de Grande Porte (LLMs) 16
2.5. Tokens e Custo Computacional 17
3. TRABALHOS RELACIONADOS **18**
3.1. Evaluating Large Language Models Trained on Code 19
3.2. ASTER 19
3.3. LLM4Fin 19
3.4. Desafios Comuns 21
4. METODOLOGIA **21**
4.1. Planejamento 22
4.1.1. Limitações assumidas do estudo 22
4.1.2. Seleção dos modelos 22
4.2. Desenvolvimento do sistema financeiro 23
4.3. Engenharia de prompt 25
4.3.1. Variações testadas 27
4.4. Experimentação Controlada 27
4.4.1. Protocolo experimental 27
4.5. Métricas de Avaliação 27
4.6. Validação e Controle de Qualidade 28
5. RESULTADOS **29**
5.1. Análise Quantitativa 29
5.1.1. Desempenho Temporal e Eficiência 29
5.1.2. Eficácia da Geração de Testes (Válidos vs Falhos) 30
5.1.3. Cobertura de Código: Qualidade vs. Quantidade 31
5.1.4. Consumo de Tokens e Custo Computacional 32
5.2. Análise Subjetiva 33
5.2.1. Clareza e legibilidade 33
5.2.2. Relevância dos testes 33
6. DISCUSSÕES **35**
6.1. Eficiência Operacional e Implicações Práticas 35
6.2. Qualidade vs Volume 35
6.3. Economia de Recursos: Análise Custo-Benefício Aprofundada 36
6.4. Relação entre Prompt e Resultado 36
6.5. Cobertura Funcional vs Cobertura Sintática 36
6.6. Aplicabilidade em Sistemas Financeiros Reais 37

6.7. Limitações Identificadas e Estratégias de Mitigação 37
6.8. Impacto dos testes dummies 38
7. CONCLUSÃO E TRABALHOS FUTUROS 38
7.1. Principais Contribuições 38
7.2. Implicações Práticas 39
7.3. Limitações 39
7.4. Trabalhos Futuros 40
7.5. Considerações Finais 40
REFERÊNCIAS 41

1. INTRODUÇÃO

A crescente complexidade dos sistemas computacionais, somada à necessidade de entregas rápidas e confiáveis, impulsiona o avanço de técnicas de automação no desenvolvimento de software. Entre essas técnicas, destaca-se a automação de testes, prática essencial para assegurar que um sistema opere conforme o esperado e com qualidade suficiente para atender às demandas de usuários e organizações [1]. Recentemente, a introdução dos Modelos de Linguagem de Grande Porte (LLMs, *Large Language Models*) tem ampliado significativamente as possibilidades de automação, permitindo que tarefas tradicionalmente executadas por humanos, como a geração de casos de teste, possam ser realizadas com base em instruções textuais, regras de negócio ou diretamente a partir do código-fonte [8]. Esses modelos têm se mostrado promissores ao gerar artefatos de teste de forma rápida e contextualizada, ainda que sua eficácia esteja diretamente associada à qualidade das instruções fornecidas, ao modelo adotado e ao domínio da aplicação.

1.1. Contexto

A validação de software constitui uma etapa essencial para garantir que os sistemas se comportem conforme os requisitos especificados e estejam livres de falhas que possam comprometer seu funcionamento ou a confiança de seus usuários. Segundo Ammann e Offutt, o teste é a técnica mais amplamente adotada para detectar falhas e avaliar a qualidade de sistemas, sendo particularmente relevante em domínios onde a confiabilidade, precisão e robustez são exigências inegociáveis [1]. De forma complementar, Pezzè e Young reforçam que o processo de teste deve ser sistemático e orientado por critérios formais, especialmente em domínios críticos nos quais falhas podem ter consequências severas [2].

Em sistemas de natureza financeira, essas exigências são amplificadas devido à presença de operações críticas como manipulação de dados sensíveis e controle de movimentações monetárias, que operam sob forte regulamentação. Essa criticidade molda a importância da eficácia do processo de validação, uma vez que falhas no sistema podem resultar em perdas financeiras, interrupções operacionais, violações legais e danos à reputação institucional. Dessa forma, a validação torna-se não apenas desejável, mas obrigatória, uma vez que a complexidade técnica e a

criticidade operacional dessas aplicações impõem altos riscos em caso de falhas, exigindo conformidade com padrões de qualidade, regulamentações legais e expectativas de confiabilidade por parte das instituições envolvidas.

Tradicionalmente, a construção de casos de teste é realizada manualmente, exigindo esforço significativo, domínio do código, atenção a múltiplos detalhes técnicos e semânticos, além de tempo considerável dos desenvolvedores e analistas de qualidade (QA, *Quality Assurance*). Esse processo é propenso a erros, especialmente quando a cobertura de testes é limitada [1]. Como resposta a esses desafios, métodos de Geração Automática de Testes (*Automated Test Generation* – ATG) vêm sendo explorados para aumentar a eficácia da validação e reduzir a carga de trabalho envolvida [6]. Recentemente, a introdução de Modelos de Linguagem de Grande Porte (*Large Language Models* – LLMs) tem ampliado essas possibilidades, viabilizando abordagens que utilizam linguagem natural para gerar testes de maneira mais adaptativa e semântica [8].

Nos últimos anos, o surgimento dos LLMs abriu novas possibilidades na engenharia de software, incluindo a geração automatizada de testes. Tais modelos, treinados em grandes volumes de texto e códigos, têm demonstrado capacidade para gerar casos de teste compreensíveis, coerentes e alinhados com a lógica do sistema-alvo [14].

Nesse contexto, o presente trabalho se propõe a desenvolver e avaliar os resultados gerados por uma pipeline automatizada para geração de testes alimentada por modelos LLMs. Como base para a avaliação, será desenvolvido um sistema financeiro que servirá de código-fonte para geração de testes. Além disso, busca-se investigar em que medida a adoção de LLMs pode contribuir para o aumento da eficiência do processo de validação, avaliando métricas como cobertura de código, relevância dos testes gerados, custo computacional e clareza dos artefatos produzidos. Com isso, acredita-se que os resultados obtidos permitirão a análise crítica do potencial e das limitações desta abordagem, fornecendo subsídios para o avanço de práticas mais robustas e automatizadas de validação de software em sistemas financeiros.

1.2. Objetivos

O objetivo geral deste trabalho é propor e avaliar os resultados de uma pipeline automatizada alimentada com LLMs para o desenvolvimento de testes em um sistema financeiro. Através da análise dos testes gerados utilizando quatro modelos distintos — *GPT-4o-mini*, *Qwen2.5-Coder*, *Claude-Sonnet-4* e o *GPT-4.1* — será possível reunir métricas importantes para a validação desta ferramenta.

Como objetivos específicos, busca-se: (i) entender como e quais modelos, dentre os escolhidos, possuem resultados mais assertivos e com melhor qualidade; (ii) avaliar o custo computacional da produção de testes e sua relação com a qualidade obtida; (iii) evidenciar a criação e o efeito de testes *dummy* em métricas de validação como cobertura de linhas; e (iv) apoiar o uso de LLMs como parte do processo de validação de software, avaliando os resultados obtidos por modelos amplamente utilizados na indústria.

2. REFERENCIAL TEÓRICO

2.1. Sistemas Financeiros

Sistemas financeiros representam um dos pilares da infraestrutura econômica moderna, sendo responsáveis pelo registro, processamento e análise de transações monetárias entre indivíduos, instituições e governos. Tais sistemas desempenham papel central na alocação de recursos, controle de risco, e monitoramento de atividades econômicas, exigindo elevados padrões de segurança, precisão e disponibilidade.

2.1.1 Visão tecnológica

Com o avanço das tecnologias de software e arquitetura distribuída, muitos desses sistemas migraram para plataformas digitais baseadas em APIs (*Application Programming Interface*), com bancos de dados em tempo real e integração contínua com diversos serviços, externos e internos, como autenticação, sistemas de crédito e redes bancárias. Nesse cenário, essa categoria de sistemas envolve manipulação de dados sensíveis, como saldos, transações, históricos e perfis de clientes — dados que demandam confiabilidade e disponibilidade.

Segundo o relatório da NVIDIA, divulgado em 2024, a demanda por soluções de alta confiabilidade no setor financeiro impulsiona o uso de tecnologias de inteligência artificial, principalmente para tarefas como análise de risco, detecção de fraudes e automação de processos [3]. A natureza crítica e regulada desses processos, também impõe desafios específicos de validação de software, ao exigirem verificações funcionais, legais, éticas e operacionais. Tais exigências tornam o uso de modelos de inteligência artificial como alternativas promissoras para acelerar a produção e validação do código-fonte.

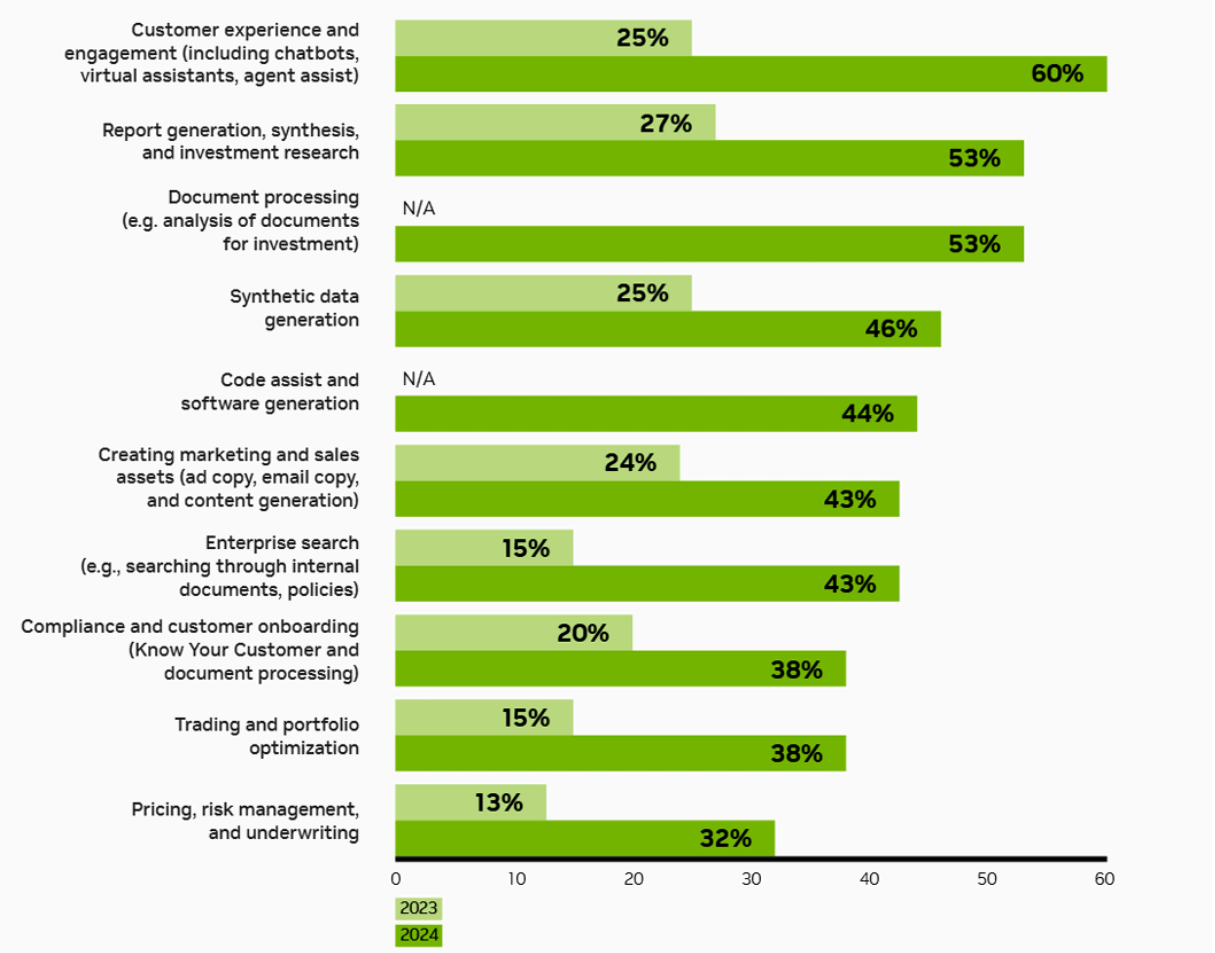

Figura 1 – Principais casos de uso de IA Generativa
Fonte: NVIDIA, 2024.

Como apresentado na figura 1, o caso de uso, *"code assist and software generation"*, indicado no eixo Y, mostra a evolução e *trending* do uso de IA generativa como agente auxiliar de código, tornando essa tecnologia como parte do ecossistema de software.

2.1.1 IA Generativa em Sistemas Financeiros

O uso de Modelos de Linguagem de Grande Porte (LLMs) em sistemas financeiros vem crescendo rapidamente, impulsionado por sua capacidade de gerar códigos, interpretar regras de negócio, redigir relatórios e auxiliar na validação de funcionalidades de sistemas. A integração desses modelos às rotinas de desenvolvimento permite acelerar a criação de testes, sugerir melhorias em código e documentar comportamentos inesperados automaticamente.

Conforme destacado pela ESMA (*European Securities and Markets Authority*) em conjunto do ILB (*Institut Louis Bachelier*) e *The Alan Turing Institute* em 2025, LLMs já vem sendo aplicados em tarefas como análise contratual, avaliação de risco e suporte a processos de KYC (*Know Your Customer*, Conheça seu cliente). Ainda que existam riscos associados, como falta de clareza ou comportamento imprevisível, o potencial de automação e ganho de produtividade justifica sua crescente adoção [4].

2.2. **Validação de Software**

A validação de software consiste no processo de garantir que um sistema esteja em conformidade com seus requisitos específicos, operando de maneira correta e segura em diferentes condições de uso. Esse processo é essencial em sistemas críticos, sendo particularmente relevante no contexto financeiro devido às suas características de alta criticidade.

Segundo Cadar e Sen, ferramentas e práticas de validação são fundamentais para reduzir riscos operacionais, aumentar a confiabilidade e garantir a rastreabilidade das decisões de sistemas [5]. A complexidade do setor financeiro, marcado por regras de negócios dinâmicas e alta interdependência entre componentes, torna sua validação um desafio constante.

Com a adoção de inteligência artificial no desenvolvimento de software, surgem novos cenários para validação, incluindo a necessidade de verificar se os comportamentos gerados por modelos de LLMs estão em conciliação com os objetivos estabelecidos. Nesse contexto, a integração de validação contínua em fluxos de trabalhos automatizados, alimentados por LLMs, para geração de testes,

torna-se uma abordagem estratégica que conecta diretamente com as técnicas de Geração Automática de Testes.

2.3. **Geração Automática de Testes (ATG)**

A Geração Automática de Testes (*Automated Test Generation*) refere-se ao conjunto de técnicas e ferramentas destinadas a automatizar a criação de casos de teste a partir do código-fonte de um sistema. Seu principal objetivo é reduzir o esforço humano necessário na elaboração de testes, ao mesmo tempo, em que busca ampliar a cobertura e a qualidade da verificação do programa.

Dentre as abordagens tradicionais de ATG, destaca-se o *EvoSuite*, ferramenta automatizada de criação para testes para Java, que utiliza algoritmos evolucionários para gerar testes estruturais com base em critérios como cobertura de instruções, ramos e mutações. No entanto, embora eficaz quando se fala de cobertura sintática, o *EvoSuite* apresenta limitações significativas na geração de oráculos, ou seja, na verificação de que a saída produzida por uma execução corresponde ao comportamento esperado do sistema [6].

Com a introdução de Modelos de Linguagem, novas abordagens de ATG vem sendo investigadas, especialmente aquelas que envolvem geração de testes baseados em linguagem natural, raciocínio semântico e compreensão contextual. Essas estratégias possibilitam uma transição de métodos estruturais para soluções mais adaptativas, que visam alcançar cobertura, mas também coerência lógica e validade dos testes.

2.4. **Modelos de Linguagem de Grande Porte (LLMs)**

Os Modelos de Linguagem de Grande Porte (LLMs, *Large Language Models*) representam uma das mais recentes inovações aplicadas à engenharia de software. Trata-se de modelos baseados em arquiteturas de aprendizado profundo, treinados com extensivos conjuntos de dados textuais, visuais e de códigos, capazes de compreender e gerar linguagem natural e código-fonte de maneira autônoma e contextualizada [11].

Na prática de engenharia de software, os LLMs vêm sendo abordados em atividades como sugestão de código (*code completion*), refatoração, explicação de

trechos complexos, documentação automática, através dos chamados agentes de código como *GitHub Copilot*[1], *Cider*[2] e outros, e mais recentemente, na geração de testes. Essa aplicação está diretamente relacionada à sua capacidade de compreender instruções em linguagem natural e transformá-las em artefatos de software.

Contudo, estudos recentes apontam limitações importantes. Segundo Konstantinou et al., embora os LLMs possam gerar trechos de código válidos, frequentemente cometem erros de lógica ou não compreendem completamente os requisitos da tarefa proposta [7]. Já para Azanza et al., tais modelos possuem dificuldades em manter a consistência entre múltiplos arquivos e em contextos maiores, comprometendo sua eficácia em tarefas complexas [8].

Outro fator central no uso de LLMs é a engenharia de *prompts*. A qualidade das saídas está diretamente relacionada à clareza, estrutura e completude das instruções fornecidas. Nesse sentido, Park, D. et al., apresenta que variações sutis na construção de prompts resultam em diferenças substanciais na acurácia, eficiência e relevância das respostas [9].

Adicionalmente, o uso de LLMs impõe custos computacionais relacionados à quantidade de tokens processados, afetando o tempo de execução, o custo e a qualidade por requisição. Isso se torna especialmente relevante quando os modelos são integrados a pipeline automatizadas com fluxos de CI/CD. Portanto, embora os LLMs representam um avanço significativo para a automação de *tasks*, sua aplicação prática exige planejamento, escolha cuidadosa do modelo, construção eficaz de *prompts* e validação supervisionada dos resultados.

2.5. **Tokens e Custo Computacional**

Modelos de Linguagem de Grande Porte (LLMs) operam por meio da *tokenização* de texto, processo no qual entradas e saídas são convertidas em unidades menores chamadas *tokens*. Cada token pode representar uma palavra completa, parte de uma palavra ou até mesmo um símbolo de pontuação, dependendo do idioma e modelo utilizado. Essa granularidade permite aos LLMs

[1] Disponível em: https://github.com/features/copilot
[2] Disponível em: https://www.ciderapp.ai/

lidar com instruções textuais complexas, mas também introduz uma métrica central no custo de utilização dos modelos.

Em serviços comerciais como OpenAI e Claude, o custo de cada requisição é calculado com base na quantidade total de tokens processados, somando entradas e saídas. Essa soma impacta diretamente no custo financeiro de execuções em pipelines alimentadas com LLMs, especialmente em projetos que envolvem múltiplos arquivos ou prompts longos. Para evitar um grande gasto de *tokens*, cada modelo possui um limite máximo de *tokens* que podem ser utilizados em chamadas, restringindo a quantidade de contexto que pode ser fornecido em uma única interação.

Esse fator é particularmente relevante no contexto de geração de testes, visto que *prompts* mal otimizados podem consumir mais tokens do que o necessário, elevando os custos e reduzindo a eficiência e qualidade. Segundo Sherifi et al., há uma relação direta entre a estrutura dos prompts, o número de *tokens* gerados e a eficácia dos testes produzidos. Modelos como *Claude* tendem a consumir mais tokens em tarefas complexas, enquanto modelos menores, como o *GPT-4o-mini*, demonstraram gerar respostas mais rápidas com menor custo computacional, ainda que por vezes menos completas [10].

A compreensão e o controle do uso de *tokens* são, portanto, elementos essenciais na construção de pipelines eficientes com LLMs. A escolha cuidadosa do modelo, a compactação do contexto fornecido e a formulação concisa dos *prompts* são estratégias fundamentais para otimizar tanto a qualidade dos resultados quanto os recursos computacionais utilizados.

3. TRABALHOS RELACIONADOS

A aplicação de LLMs para geração automatizada de testes tem despertado interesse crescente na comunidade científica, especialmente em cenários que demandam alta cobertura e adaptabilidade ao contexto. Diversas abordagens vêm sendo propostas para avaliar e expandir as capacidades desses modelos em diferentes níveis de teste, desde testes unitários até testes de aceitação.

3.1. **Evaluating Large Language Models Trained on Code**

O trabalho de Chen et al., explora extensivamente o desempenho de LLMs treinados com dados de código, como Codex e GPT, em tarefas relacionadas à programação. Os autores avaliam os modelos em *benchmarks* que envolvem a completude de código, resolução de problemas e escrita de funções completas. Uma das principais conclusões é que, apesar do desempenho promissor, os LLMs apresentam limitações em relação à generalização sem exemplos ou ao lidar com instruções mal especificadas. Isso sugere que, embora úteis, os LLMs ainda requerem contexto claro e direcionamento adequado para gerar códigos confiáveis, incluindo testes [11].

3.2. **ASTER**

Por sua vez, em ASTER, PAN, R. et al., propõe um sistema de geração automática de testes unitários a partir de descrições naturais, explorando tanto a capacidade de interpretação semântica dos LLMs quanto sua adaptação a múltiplas linguagens de programação. O diferencial do ASTER está na integração de análise estática com o uso de LLMs para geração de entradas e assertivas. As avaliações do sistema demonstram que ele supera abordagens baseadas somente em heurísticas ou *templates*, mas ainda enfrenta desafios ao lidar com comportamentos esperados complexos ou *APIs* com baixa documentação. O artigo reforça a necessidade de mecanismos adicionais de validação e a importância da integração com ferramentas tradicionais de teste [12].

3.3. **LLM4Fin**

O trabalho LLM4Fin, apresentado por XUE, Z. et al, introduz uma pipeline completa para geração automática de testes de aceitação em sistemas financeiros, com ênfase na extração de regras de negócios a partir de documentos de linguagem natural. O sistema combina LLMs afinados com algoritmos determinísticos para geração de cenários de teste. Uma das principais contribuições do LLM4Fin é sua capacidade de produzir testes com alta cobertura de regras e de código, superando inclusive engenheiros especializados em diversos cenários e tempo [13].

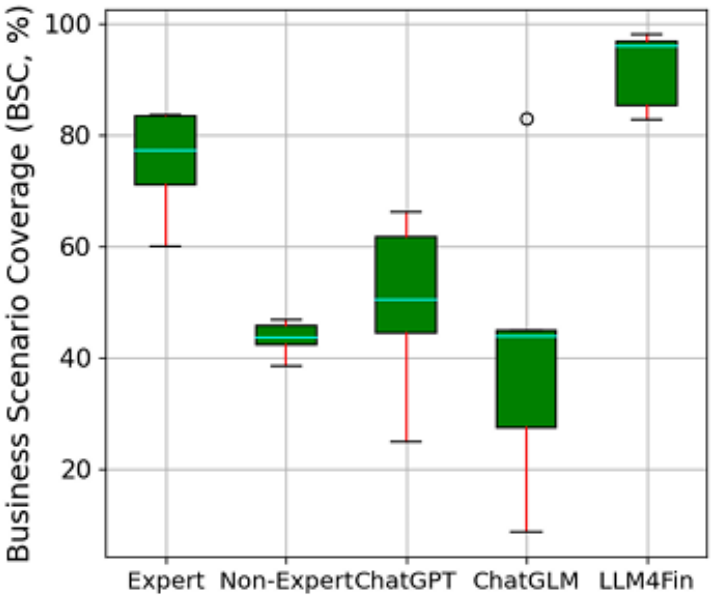

Figura 2 – Cobertura em cenário de regras de negócio

Fonte: XUE, Z. et al, 2024.

A Figura 2 ilustra a avaliação do sistema em *Business Scenario Coverage (BSC)*, métrica que representa o percentual de regras de negócio cobertas pelos testes gerados, considerando os comportamentos esperados do sistema a partir da documentação e requisitos funcionais. Diferentemente da cobertura de código tradicional, que mede apenas a execução de linhas ou instruções, o BSC avalia a efetividade da validação em termos de lógica de negócio. No trabalho do LLM4Fin, esse índice alcançou até 98,18%, indicando uma elevada capacidade do modelo em reproduzir cenários críticos e relevantes do domínio financeiro. Logo abaixo dele encontra-se o ChatGPT, demonstrando uma eficiência moderada, embora com algumas inconsistências e instabilidades nos testes gerados.

Em contraste com as abordagens anteriores, este trabalho apresenta uma investigação empírica da geração automatizada de testes no contexto de sistemas financeiros genéricos. A metodologia proposta adota um processo mais leve que utiliza modelos de linguagem pré-treinados sem *fine-tuning*, aplicando técnicas de engenharia de prompt para avaliar a efetividade e qualidade dos testes gerados.

A pesquisa explora critérios essenciais como adaptabilidade, custo-benefício, consumo de tokens, cobertura de linhas e eficácia de *LLMs* genéricos em cenários reais de desenvolvimento de software. O estudo busca fornecer contribuições

relevantes para a aplicação prática dessa tecnologia sem a necessidade de *fine-tuning* ou infraestrutura especializada, implementando a solução por meio de *scripts* locais e *GitHub Actions*[3].

3.4. Desafios Comuns

Segundo XUE, Z. et al, LLMs genéricos enfrentam desafios, como a dependência de *prompts* bem formulados e o desconhecimento de regras específicas do domínio, onde uma abordagem híbrida proposta, combinando LLMs com métodos algorítmicos, revela-se eficaz para mitigar essas limitações e aumentar a reprodutibilidade e confiabilidade dos testes [13].

Esses trabalhos, em conjunto, demonstram que os LLMs têm potencial significativo para transformar como casos de teste são gerados. Entretanto, também evidenciam limitações em termos de controle, precisão e dependência de conhecimento de domínio. A presente pesquisa se inspira nesses estudos ao propor e avaliar a execução de uma pipeline explorando quatro diferentes LLMs para auxiliar na geração de testes automatizados, investigando sua efetividade no contexto de sistemas financeiros com base em métricas de validação de software.

4. METODOLOGIA

Esta seção descreve a metodologia utilizada para a obtenção e análise dos dados da avaliação em questão. A metodologia proposta combina técnicas de engenharia de *prompt*, avaliação empírica dos testes gerados e análise quantitativa e subjetiva dos resultados obtidos, com foco na identificação de limitações e potencialidades dos LLMs na geração automatizada de testes.

O desenvolvimento desta avaliação está dividido em cinco etapas principais: (1) Planejamento e definição do escopo experimental; (2) Implementação do sistema financeiro e pipeline; (3) Desenvolvimento e refinamento de prompts; (4) Experimentação controlada; e (5) Avaliação multidimensional dos resultados.

[3] Disponivel em: https://github.com/features/actions

4.1. **Planejamento**

Na etapa inicial, foi realizada uma revisão bibliográfica sobre técnicas de Geração Automática de Testes (ATG), com foco específico nas abordagens que utilizam LLMs. Foram estudadas métricas e ferramentas para avaliar a qualidade de testes, incluindo a cobertura de código utilizando *pytest-cov*, um *plugin* responsável pela geração de relatórios de cobertura de linhas e *branches*.

4.1.1. **Limitações assumidas do estudo**

- Sistema financeiro simplificado, não representando a complexidade total de sistemas bancários reais
- Ausência de comparação direta com ferramentas tradicionais de ATG (*EvoSuite*, *Randoop*)
- Número limitado de execuções devido a restrições de tempo e recursos computacionais
- Foco em testes unitários, não abordando testes de integração ou aceitação

4.1.2. **Seleção dos modelos**

Para a composição do trabalho, foram selecionados quatro modelos representativos do estado da arte em LLMs, considerando critérios de disponibilidade, custo, capacidade técnica e diversidade arquitetural. A combinação desses modelos permitiu a análise comparativa de diferentes abordagens de geração de testes, considerando variáveis como consumo de tokens, custo estimado por execução, tempo de resposta, qualidade dos testes gerados e sua aplicabilidade em ambientes reais de desenvolvimento.

- GPT-4o-mini (OpenAI):

O modelo possui um baixo custo por requisição (aproximadamente US$ 0,015) e agilidade nas respostas, apresentando-se como uma alternativa eficiente para cenários com restrição de recursos ou necessidade de respostas rápidas.

- *GPT-4.1* (OpenAI):

O modelo apresenta desempenho em tarefas complexas de linguagem e geração de código, sendo amplamente utilizado na indústria. Seu custo aproximado por execução é de US$ 0,03, oferecendo um bom equilíbrio entre qualidade e custo.

- *Claude-Sonnet-4 (Anthropic*):

O modelo foi selecionado por sua capacidade de produzir respostas mais completas e coerentes, sobretudo em tarefas com múltiplas instruções ou trechos de código mais longos. Apesar de seu custo mais elevado (cerca de US$ 0,15), sua inclusão permitiu avaliar os benefícios e limitações de um modelo com maior janela de contexto.

- *Qwen2.5-coder:7b* (via *Ollama*):

O modelo possui código aberto e gratuito para execução, embora demande infraestrutura local e maior tempo de processamento. A escolha da versão 7b se deu pela compatibilidade com o cache e desempenho do *GitHub Actions*, viabilizando a integração a um fluxo de integração contínua (CI) com menor impacto em custo e maior controle sobre o ambiente de execução.

4.2. **Desenvolvimento do sistema financeiro**

Para o desenvolvimento do sistema financeiro, foram definidos os requisitos do sistema a ser desenvolvido, composto por funcionalidades essenciais como:

- Criação e gerenciamento de usuários
- Abertura e administração de contas financeiras
- Realização de transações entre entidades (indivíduos, instituições, governos)
- Validação de regras de negócio básicas (saldo suficiente, contas válidas)
- Histórico de movimentações

Essa etapa contou com o desenvolvimento do sistema, *utils* para leitura e extração de código-fonte a fim de serem inseridos no *prompt* e a pipeline.

O sistema foi estruturado com uma arquitetura modular, visando facilitar a geração de testes por parte dos LLMs. A arquitetura do sistema é constituída de

models, *database*, *schemas, enums* e um arquivo principal, *main*, responsável pelas rotas da *API*. Cada módulo é responsável por uma camada ou validação da aplicação, sendo *utils* dedicado a funções auxiliares que apoiam tarefas como leitura de arquivos, extração de código e formatação.

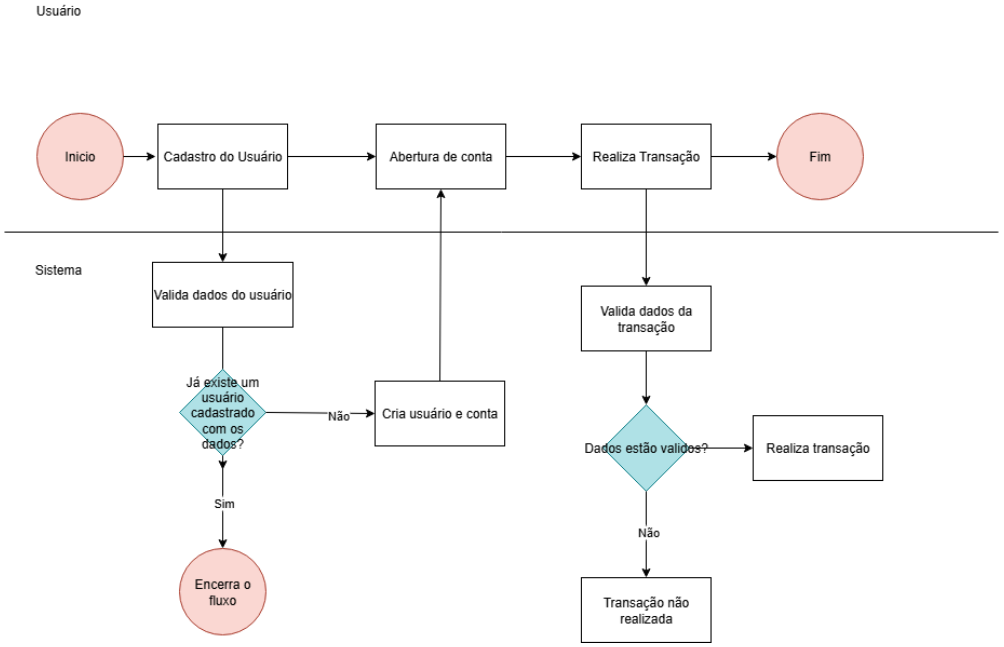

Figura 3 – Fluxo do sistema

Conforme apresentado na figura 3, o fluxo de operações disponíveis inicia-se com a criação de um usuário que apresenta dados como nome e e-mail, onde cada usuário possui uma conta, representando o funcionamento de agências, que possuem informações do seu saldo e histórico de movimentações financeiras. Em seguida, o usuário está liberado para realizar transações em suas contas, caso sejam realizadas entre contas válidas e existentes e com valores acima de zero. Para garantir precisão dos dados, cada adição ou leitura no banco de dados possui validações presentes no arquivo de *schemas* com suporte do *pydanticV2*[4].

Em seguida, deu-se início a implementação de *utils* responsáveis por extrair linhas sem coberturas no reporte ou arquivos inteiros. As principais responsabilidades dos métodos de auxílio são:

- Ler e identificar os códigos-fonte relevantes para o *prompt*.

[4] Disponível em: https://docs.pydantic.dev/latest/

- Gerar *prompts* adequados com base na estrutura e contexto do código.
- Inicializar *clients* para conexão com os modelos.
- Enviar os *prompts* ao modelo escolhido.
- Armazenar os testes gerados.

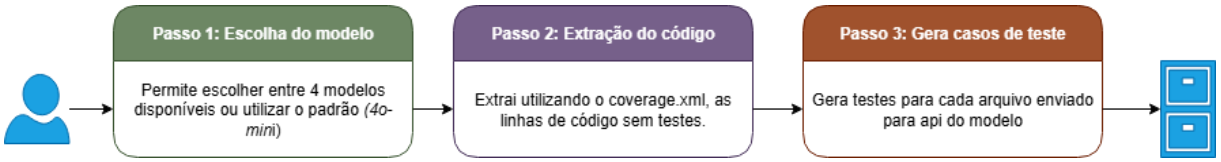

Figura 4 – Passos de execução do fluxo de trabalho

Opcionalmente, a pipeline foi integrada a um fluxo de integração contínua (CI), utilizando o *GitHub Actions*, de modo a viabilizar a execução automática dos testes sempre que o usuário solicitar a partir de comentários no Pull Request.

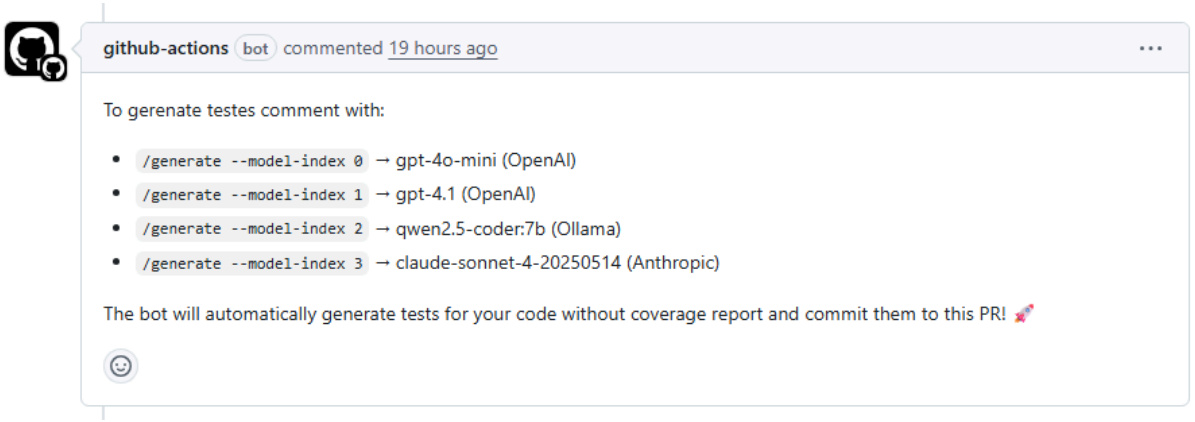

Figura 5 – Instruções do fluxo em um *Pull Request*

4.3. Engenharia de *prompt*

Uma contribuição específica deste trabalho foi o desenvolvimento de um *prompt* estruturado para geração de testes. O processo de engenharia de *prompts* seguiu uma abordagem iterativa, utilizando as boas práticas previstas pela OpenAI[5] e Anthropic[6] e com refinamentos baseados na qualidade dos resultados obtidos.

[5] Disponível em: https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api
[6] Disponível em: https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/claude-4-best-practices

````
You are an SDET specializing in test automation. Your responsibility is to
write unit and integration tests for python projects.

You are an expert in all testing techniques and understand when each is
appropriate. Your tests always follow the best quality assurance practices,
especially for Pytest.

You will receive three inputs: a code file, a list of imports and some
contexts files. Please perform the following tasks:

All the provided code is in Python, using FastAPI, pydanticV2 and
sqlalchemy and are on the main folder: app.
    Example: app/{filename}.py

1. Read and analyze the provided code file. This code has no tests, and you
will write tests for it.

```python
{code}
```

 2. Identify all the test cases and scenarios to cover. Consider including
happy-paths, sad-paths and edge cases in your test sample. Your goal is to
reach at least 90% of line coverage.

3. Analyze the context files to better write the tests and understand the
code dependencies.

4. Avoid redundant tests and duplications in your test cases, covering the
same line multiple times unnecessarily.

5. Generate an output just with the tests ready to be executed, without any
additional text or markdown formatting.

6. Generate pytest fixtures, IF NEEDED, using scope='function'.

7. Ensure data uniqueness within tests

8. Perform explicit rollback after exceptions

9. Ensure tests use float or str (which can be automatically converted to
json) instead of directly using Decimal

### File context

List of imports used on the main code file:

```python
{imports}
```
Context files that can be used to understand the code and its dependencies:
{format_related_files(related_files)}

Your output must be a complete pytest file, ready to be executed. Write
tests that increase the code coverage.  Give clear, relevant, and easily
maintainable tests that cover happy and sad paths. The user must have the
least work possible.
````

4.3.1. **Variações testadas**

- **Prompt básico:** Instruções mínimas para geração de testes
- **Prompt estruturado:** Incluindo diretrizes específicas e formato de resposta

4.4. **Experimentação Controlada**

Com a conclusão do desenvolvimento e integração com *GitHub Actions*, iniciou-se a etapa de experimentação. Para garantir confiabilidade e reprodutibilidade dos resultados, foram estabelecidos protocolos rigorosos de execução.

4.4.1. **Protocolo experimental**

- **Ambiente padronizado:** Todas as execuções em ambiente controlado
- **Configuração consistente:** Parâmetros fixos (temperature=0.1, max_tokens=4000)
- **Isolamento de execuções:** Limpeza completa do ambiente entre testes
- **Coleta sistemática:** Métricas padronizadas em todas as execuções

Devido a restrições de tempo e recursos computacionais, foram realizadas 2 execuções por modelo. Embora este número seja inferior ao ideal para análise estatística robusta, os resultados fornecem indícios relevantes sobre o comportamento dos modelos.

4.5. **Métricas de Avaliação**

A avaliação foi estruturada em duas dimensões complementares:

Análise Quantitativa:

- **Tempo de execução:** Medição desde o envio do *prompt* até a conclusão da geração
- **Taxa de sucesso:** Proporção de testes válidos vs. testes com falhas sintáticas
- **Cobertura de código:** Porcentagem de linhas executadas (*pytest-cov*)

- **Consumo computacional:** Número de tokens processados e custo monetário estimado
- **Produtividade:** Número de testes gerados por minuto

Análise Subjetiva:

Todas as análises subjetivas foram realizadas integralmente pela autora deste trabalho, com base na observação direta dos resultados e nas métricas obtidas. As métricas utilizadas foram:

- **Relevância funcional:** Avaliação da aderência às regras de negócio
- **Detecção de testes *dummy*:** Identificação de testes que não agregam valor
- **Clareza e manutenibilidade:** Qualidade da nomenclatura e documentação
- **Cobertura de cenários:** Diversidade de casos testados

Critérios para identificação de testes dummy:

1. Verificação apenas da existência de atributos
2. Testes de representação *string* sem lógica associada
3. Validações triviais que não detectam falhas reais
4. Redundância excessiva sem valor adicional

A combinação das análises possibilitou uma comparação mais pontual entre os modelos, destacando vantagens e limitações de cada na execução da mesma tarefa: gerar testes. Esses dados fundamentam a discussão presente na seção seguinte.

4.6. Validação e Controle de Qualidade

Para garantir a confiabilidade dos resultados, foram implementados mecanismos de validação:

Validação técnica:

- Execução de todos os testes gerados
- Verificação de sintaxe e importações

- Análise de cobertura através de ferramentas padronizadas

Validação semântica:

- Revisão manual dos testes para identificação de *dummy tests*
- Avaliação da relevância para o domínio financeiro
- Verificação de aderência às melhores práticas de teste

Esta metodologia, embora apresente limitações em termos de escala experimental, sendo realizada somente duas execuções por modelo, fornece uma base sólida para avaliar o potencial e as limitações dos LLMs na geração automatizada de testes para sistemas financeiros.

5. RESULTADOS

Nesta seção são apresentados e analisados os resultados obtidos a partir da execução da pipeline proposta, que utilizou quatro modelos de LLMs para geração de testes: *GPT-4o-mini, GPT-4.1, Qwen2.5-Coder:7b e Claude-Sonnet-4-20250514*. Os experimentos foram conduzidos em ambiente controlado, com parâmetros fixos e condições replicáveis, de modo a garantir a reprodutibilidade dos ensaios e a confiabilidade dos dados gerados.

5.1. Análise Quantitativa

5.1.1. Desempenho Temporal e Eficiência

O tempo médio de execução do fluxo de geração de testes apresentou variação significativa entre os modelos avaliados. O *GPT-4o-mini* demonstrou maior eficiência com média de 86 segundos, seguido pelo *GPT-4.1* com 90 segundos. Os modelos *Qwen2.5-Coder:7b* (128 segundos) e *Claude-Sonnet-4* (131 segundos) apresentaram tempos de execução superiores.

Modelo	Execução 1 (seg)	Execução 2 (seg)	Desvio (seg)
GPT-4o-mini	84	88	±2
GPT-4.1	89	91	±1
Claude-Sonnet-4	129	133	±3
Qwen-2.5-coder:7b	125	131	±2

Tabela 1 – Tempo total de execução da pipeline

Esta variação pode ser atribuída a fatores distintos para cada modelo: no caso do *Qwen2.5-Coder:7b*, o maior tempo deve-se à execução local via *Ollama*, incluindo *overhead* de cache e processamento. O *Claude-Sonnet-4*, apesar do tempo elevado, justifica a latência adicional pela complexidade do modelo e qualidade superior dos resultados, conforme será detalhado nas próximas seções.

5.1.2. **Eficácia da Geração de Testes (Válidos vs Falhos)**

A análise da proporção entre testes válidos (executados com sucesso) e testes falhos (com erros de sintaxe ou lógica) revela diferenças substanciais na capacidade dos modelos de gerar testes funcionais. O *Claude-Sonnet-4* destacou-se com o melhor desempenho: 78 testes válidos contra 11 falhos na primeira execução (87,6% de taxa de sucesso) e 75 válidos contra 13 falhos na segunda execução (85,2% de taxa de sucesso).

O segundo melhor desempenho foi apresentado pelo *Qwen2.5-Coder:7b*, com 36 válidos e 12 falhos na primeira execução (75% de taxa de sucesso) e 32 válidos e 10 falhos na segunda execução (76,2% de taxa de sucesso). O *GPT-4o-mini* manteve desempenho consistente com 35 válidos e 5 falhos na primeira execução (87,5% de taxa de sucesso) e 32 válidos e 4 falhos na segunda execução (88,9% de taxa de sucesso).

O *GPT-4.1* apresentou o pior desempenho, com 38 válidos e 62 falhos na primeira execução (38% de taxa de sucesso) e 36 válidos contra 40 falhos na segunda execução (47,4% de taxa de sucesso). Essa baixa taxa de sucesso do *GTP-4.1* está diretamente relacionada à geração massiva de testes *dummy*, conforme será detalhado na análise subjetiva.

Modelo	Execução 1	Execução 2
GPT-4o-mini	35 válidos / 5 falhos	32 válidos / 4 falhos
GPT-4.1	38 válidos / 62 falhos	36 válidos / 40 falhos
Claude-Sonnet-4	78 válidos / 11 falhos	75 válidos / 13 falhos
Qwen-2.5-coder:7b	36 válidos / 12 falhos	32 válidos / 10 falhos

Tabela 2 – Resultado das execuções dos testes

5.1.3. **Cobertura de Código: Qualidade vs. Quantidade**

A cobertura de código, aferida pelo *pytest-cov*, revelou que o volume de testes não se traduz necessariamente em melhor cobertura funcional. O *GTP-4o-mini* alcançou a maior cobertura efetiva, com 97% de cobertura na primeira execução e 96% na segunda, seguido pelos *Claude-Sonnet-4* com 89% na primeira execução e 90% na segunda execução, registrando +2%. O *GPT-4.1* também obteve resultado significativo, 85% na primeira execução e 87% na segunda, também registrando um aumento de +2%. Enquanto isso, *Qwen2.5-coder:7b* teve o menor resultado de cobertura entre os modelos, com 83% de cobertura na primeira execução e 82% na segunda.

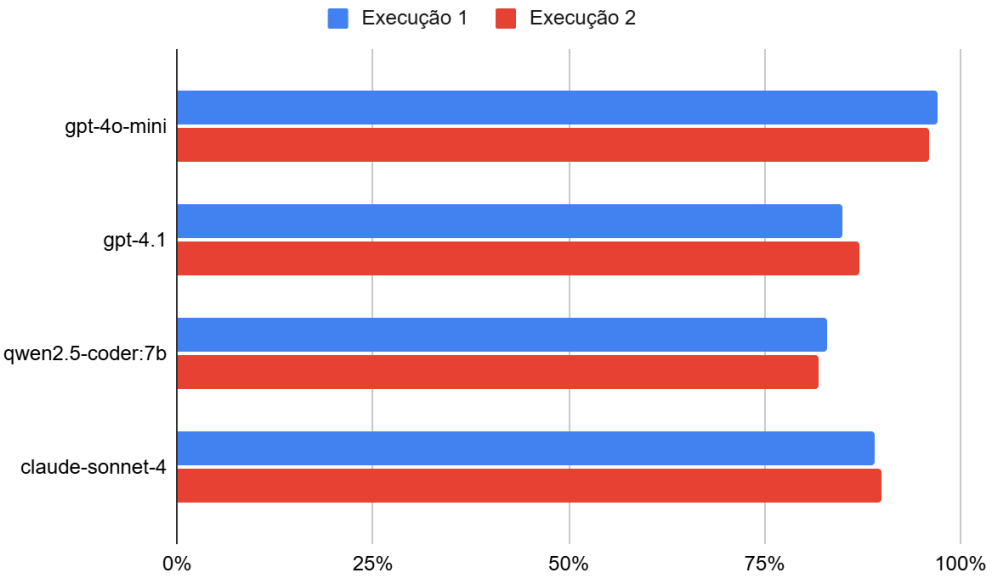

Figura 6 – Cobertura de código por modelo

Um ponto crítico identificado foi que apesar do GPT-4.1 possuir a terceira melhor cobertura, ele gerou um grande volume de testes *dummy* que inflacionaram artificialmente a cobertura de linhas sem validar requisitos relevantes do domínio. Esta observação evidencia a necessidade de análise subjetiva complementar às métricas quantitativas de cobertura.

5.1.4. **Consumo de Tokens e Custo Computacional**

O custo operacional de cada modelo foi calculado com base no consumo médio de tokens (somando *input* e *output*) e nos valores praticados pelas plataformas até o momento da pesquisa. Optou-se pelo uso da média em vez do valor absoluto de todas as execuções, pois houve variações pontuais no tamanho dos prompts e nas respostas retornadas, influenciadas pelo código analisado e pelo contexto fornecido. Dessa forma, a média representa melhor o comportamento geral do modelo e evita que valores atípicos distorçam a análise comparativa.

Todos os testes foram gerados com *temperature=0.1*, parâmetro que favorece maior consistência e menor aleatoriedade nas respostas. Esse valor foi escolhido com base na natureza do problema, onde a reprodutibilidade e a precisão dos casos de teste eram mais relevantes do que a diversidade criativa nas respostas.

Os resultados revelaram diferenças significativas no custo por execução:

Modelo	Tokens Médios	Custo/Execução (US$)	Eficiência*
GPT-4o-mini	~7.340	0,015	22,33
GPT-4.1	~7.500	0,03	12,33
Claude-Sonnet-4	~30.000	0,15	5,1
Qwen2.5-Coder:7b	~8.000	0,000**	0,000**

Tabela 2 – Tokens vs Custo Estimado por Execução

*Eficiência = Média dos testes validos / custo de execução.

**Gratuito, mas requer infraestrutura própria

Desse modo, o modelo utilizado da *Anthropic*, o *Claude-Sonnet-4*, apesar do maior consumo de *tokens* e custo, justifica o investimento pela qualidade superior dos testes gerados. O *GTP-4o-mini* apresenta a melhor relação custo-benefício para cenários com restrições orçamentárias, enquanto o *Qwen2.5-coder:7b* oferece alternativa gratuita com qualidade intermediária, porém com custos indiretos de infraestrutura.

5.2. **Análise Subjetiva**

A análise subjetiva foi realizada integralmente pela autora, a partir da observação direta e detalhada dos testes gerados por cada modelo. Para isso, foram considerados critérios como clareza da nomenclatura, organização estrutural do código, uso de *docstrings* e comentários, coerência das asserções e adequação dos cenários testados às regras de negócio. Os testes foram inspecionados individualmente, sem apoio de ferramentas automatizadas de avaliação, permitindo identificar padrões, pontos fortes e limitações de cada modelo. Essa abordagem buscou avaliar não apenas a correção sintática, mas também a legibilidade, manutenibilidade e relevância prática dos artefatos produzidos.

5.2.1. **Clareza e legibilidade**

A análise da clareza e legibilidade dos testes gerados revelou diferenças marcantes entre os modelos. O *Claude-Sonnet-4* e *GPT-4o-mini* demonstraram superior aderência a boas práticas de codificação, produzindo testes com nomenclatura descritiva, estrutura organizada e documentação adequada por meio de *docstrings* e comentários explicativos.

O *Qwen2.5-Coder:7b* apresentou desempenho intermediário, gerando código sintaticamente correto, mas com menor consistência na nomenclatura e documentação. Em contrapartida, o *GPT-4.1* evidenciou limitações significativas, produzindo frequentemente testes com nomes genéricos e ausência de comentários explicativos, comprometendo a manutenibilidade dos artefatos gerados. Embora o código tenha se mantido, em sua maioria, sintaticamente correto, a falta de clareza nas asserções e na descrição dos cenários testados pode dificultar o entendimento e a reutilização dos testes, sobretudo em projetos de maior escala ou com equipes distribuídas. Ademais, foi identificada uma certa tendência dos modelos a reproduzir estruturas repetitivas e a gerar funções com escopo restrito a operações triviais, como verificação de valores fixos ou existência de atributos.

5.2.2. **Relevância dos testes**

A avaliação de relevância dos testes constitui aspecto fundamental para determinar a utilidade prática da geração automatizada. O *Claude-Sonnet-4*

destacou-se pela capacidade de gerar que exercitam cenários críticos do domínio financeiro, incluindo validação de saldo, consistência dos dados, verificação de moedas válidas e controle de limites operacionais. Apesar de terem sido identificados testes de caráter superficial (como a verificação literal de valores de *enums*), a incidência desses testes foi relativamente baixa.

Em contrapartida, o modelo *GPT-4.1* apresentou a maior incidência de testes *dummy*, caracterizados por validações triviais que não agregam valor ao processo de validação. Exemplos incluem verificação da existência de atributos simples e manipulação de representações *string* de *enums*, conforme demonstrado nos trechos de código apresentados anteriormente.

Categoria 1: Verificação de Existência

```
1  def test_base_is_declarative_base():
2      from app import database
3      assert hasattr(database.Base, "metadata")
```

Categoria 2: Testes de Representação *String*

```
1  def test_enum_str_representation():
2      assert str(TransactionStatus.PENDING) == "pending"
3      assert str(TransactionType.DEPOSIT) == "deposit"
4      assert str(AccountType.BUSINESS) == "business"
```

Categoria 3: Validações Triviais

```
1  def test_user_has_id():
2      user = User(name="Test", email="test@test.com")
3      assert hasattr(user, 'id')
```

Esses testes, embora contribuam para aumentar artificialmente a métrica de cobertura de código, não agregam valor real à validação do sistema, tampouco são eficazes para detectar falhas relevantes no contexto do domínio financeiro.

Os modelos *GPT-4o-mini* e *Qwen-2.5-coder:7b* apresentaram desempenho intermediário, conseguindo gerar testes válidos para fluxos importantes, mas também produzindo uma quantidade moderada de testes *dummy*, especialmente nos arquivos relacionados a configurações e *enums*.

6. DISCUSSÕES

Os resultados obtidos fornecem considerações importantes sobre a viabilidade, limitações e potencial da aplicação práticas dos LLMs na geração automatizada de testes para sistemas financeiros.

6.1. Eficiência Operacional e Implicações Práticas

A variação no tempo de execução entre os modelos (86-131 segundos) tem implicações diretas para integração em pipelines de *CI/CD*. O *GPT-4o-mini*, com o menor tempo de execução, mostra-se adequado para ambientes que priorizam rapidez na retroalimentação de desenvolvimento. Contudo, a análise isolada do tempo de execução pode ser enganosa: embora o *Claude-Sonnet-4* demande 52% mais tempo que o GPT-4o-mini, sua taxa de sucesso superior (87,6% vs 87,5%) e qualidade dos testes justificam o investimento adicional em tempo.

Para organizações que implementam desenvolvimento contínuo, a escolha do modelo deve considerar o *trade-off* entre velocidade e qualidade, sendo o tempo adicional do *Claude-Sonnet-4* compensado pela redução de retrabalho e maior confiabilidade dos testes gerados.

6.2. Qualidade vs Volume

A disparidade entre volume de testes e qualidade, especialmente evidente no *GPT-4.1*, revela limitação crítica das métricas tradicionais de avaliação. Embora este modelo tenha gerado grande volume de testes, sua baixa taxa de sucesso (38-47%) e alta incidência de testes *dummy* comprometem sua aplicabilidade prática.

A geração massiva de testes *dummy* pelo *GPT-4.1* não somente infla artificialmente métricas de cobertura, mas pode criar falsa sensação de segurança na validação do sistema. Testes que verificam apenas a existência de atributos ou representações *string* não contribuem para detectar falhas funcionais ou violações de regras de negócio.

Esta observação sugere a necessidade de desenvolvimento de métricas subjetivas complementares, focadas na relevância semântica dos testes gerados, especialmente em domínios críticos como o financeiro.

6.3. **Economia de Recursos: Análise Custo-Benefício Aprofundada**

A análise econômica revela cenários distintos de aplicação para cada modelo:

- **Claude-Sonnet-4** (US$ 0,15/execução): Justifica-se em contextos onde a qualidade dos testes é prioritária e os custos de falhas são elevados. Para um sistema financeiro em produção, o custo adicional é insignificante comparado aos riscos de falhas não detectadas.
- **GPT-4o-mini** (US$ 0,015/execução): Oferece solução econômica para validação contínua em ambientes de desenvolvimento, onde a geração frequente de testes compensa a menor sofisticação individual dos casos gerados.
- **Qwen2.5-Coder:7b** (gratuito): Apresenta alternativa viável para organizações com restrições orçamentárias, quando dispuserem de infraestrutura computacional adequada. O custo indireto de infraestrutura deve ser considerado na análise econômica.

6.4. **Relação entre *Prompt* e Resultado**

A variação na qualidade dos resultados entre execuções evidencia a dependência crítica da engenharia de prompts. Modelos como *GPT-4.1* e *Qwen2.5-Coder:7b* mostraram-se particularmente sensíveis à estrutura das instruções, resultando em comportamentos inconsistentes.

A adoção bem-sucedida de LLMs para geração de testes requer investimento significativo em: (i) desenvolvimento de prompts mais robustos, introduzindo regras de negócio e padrões do sistema; (ii) padronização de instruções; (iii) validação sistemática dos resultados; e (iv) mecanismos de filtragem automática de testes *dummy*.

6.5. **Cobertura Funcional vs Cobertura Sintática**

Os resultados confirmam que cobertura de código, isoladamente, constitui métrica inadequada para avaliar a eficácia da validação automatizada. A capacidade do GPT-4.1 de atingir alta cobertura mediante testes *dummy* demonstra a necessidade de métricas complementares que avaliem:

- **Cobertura de regras de negócio**: Proporção de regras específicas do domínio validadas pelos testes
- **Cobertura de cenários críticos**: Validação de fluxos de exceção e condições de contorno
- **Densidade semântica**: Relação entre linhas de código de teste e validações significativas realizadas

6.6. Aplicabilidade em Sistemas Financeiros Reais

A integração bem-sucedida com *GitHub Actions* demonstra a viabilidade técnica da abordagem em ambientes reais de desenvolvimento. Contudo, a aplicação em sistemas financeiros de produção requer considerações adicionais:

- **Conformidade regulatória**: Sistemas financeiros operam sob frameworks regulatórios rigorosos que podem exigir auditoria e documentação dos processos de validação automatizada.
- **Rastreabilidade**: A capacidade de explicar e justificar cada teste gerado torna-se crítica em ambientes auditados, favorecendo modelos como *Claude-Sonnet-4* que produzem testes mais claros e bem documentados.
- **Integração gradual**: A implementação prática deve considerar adoção incremental, iniciando com componentes menos críticos e expandindo gradualmente conforme a maturidade da solução.

6.7. Limitações Identificadas e Estratégias de Mitigação

Os experimentos revelaram limitações importantes que devem ser endereçadas para aplicação prática:

- **Inconsistência entre execuções**: Variação na qualidade dos testes entre execuções sucessivas, mitigável por meio de múltiplas execuções e seleção dos melhores resultados.
- **Contexto limitado**: Dificuldade dos modelos em manter coerência em projetos com múltiplos arquivos, endereçável através de estratégias de divisão de contexto e geração incremental.

- **Validação humana necessária**: Dependência de revisão manual para garantir relevância e correção dos testes, exigindo processo híbrido humano-máquina.

Esta análise crítica dos resultados fundamenta as conclusões e recomendações apresentadas na seção seguinte, oferecendo direcionamento prático para pesquisadores e profissionais interessados na aplicação de LLMs para validação de software.

6.8. Impacto dos testes dummies

A análise dos resultados obtidos nesta pesquisa revelou uma limitação fundamental das métricas quantitativas tradicionais quando aplicadas à avaliação de testes gerados por *LLMs*. A observação sistemática dos testes produzidos pelos quatro modelos avaliados demonstrou que a cobertura de código, isoladamente, constitui métrica inadequada para validar a eficácia da geração automatizada, especialmente devido à prevalência de testes *dummies* que inflam artificialmente os indicadores sem agregar valor funcional ao processo de validação.

Os resultados experimentais confirmaram a geração significativa de testes *dummies*, particularmente pelo GPT-4.1, que apresentou comportamento exemplar deste fenômeno. Conforme documentado na análise quantitativa, este modelo alcançou cobertura de código de 85-87%, mas registrou simultaneamente a pior taxa de sucesso (38-47%) e maior incidência de testes sem relevância funcional

7. CONCLUSÃO E TRABALHOS FUTUROS

A presente pesquisa investigou a viabilidade e eficácia de uma pipeline automatizada baseada em LLMs para geração de testes em sistemas financeiros. Através da análise comparativa de quatro modelos distintos — *GPT-4o-mini, GPT-4.1, Claude-Sonnet-4 e Qwen2.5-Coder:7b* — foi possível obter percepções relevantes sobre as potencialidades e limitações desta abordagem

7.1. Principais Contribuições

Os resultados demonstram que a integração de LLMs em pipelines de validação é não apenas viável, mas promissora, desde que acompanhada de estratégias adequadas de seleção de modelos, engenharia de prompts e validação

dos resultados. O *Claude-Sonnet-4* emergiu como a solução mais robusta, apresentando a melhor combinação de qualidade, relevância e consistência dos testes gerados, justificando seu custo computacional superior em contextos críticos.

A pesquisa evidenciou que métricas quantitativas tradicionais, como cobertura de código, podem ser enganosas quando aplicadas isoladamente à validação automatizada por LLMs. A capacidade de modelos como *GPT-4.1* de gerar testes dummy que inflam artificialmente a cobertura sem agregar valor real destaca a necessidade de abordagens de avaliação mais sofisticadas.

7.2. **Implicações Práticas**

Para organizações considerando a adoção desta tecnologia, os resultados indicam que, apesar do investimento financeiro mais elevado, Claude Sonnet 4 emerge como a solução mais recomendada devido à sua superior qualidade de análise e confiabilidade. Sugere-se uma estratégia de implementação que priorize *Claude Sonnet 4* para contextos críticos de produção, complementada por modelos como *GPT-4o-mini* em cenários de desenvolvimento e validação onde o custo-benefício seja prioritário. O diferencial técnico do *Claude Sonnet 4* justifica o investimento adicional, especialmente considerando que a integração bem-sucedida com *GitHub Actions* demonstra a maturidade da tecnologia para aplicação imediata em ambientes corporativos exigentes

7.3. **Limitações**

Este estudo apresenta limitações importantes que devem ser consideradas na interpretação dos resultados. Primeiramente, a utilização de um sistema financeiro simplificado como base experimental pode restringir a generalização dos achados para sistemas de maior complexidade e com regras de negócio mais intrincadas.

Adicionalmente, os modelos empregados não passaram por processos de *fine-tuning* específicos para o domínio financeiro, representando cenário de uso genérico que pode não refletir o potencial máximo da tecnologia quando adequadamente especializada.

A dependência de métricas quantitativas, embora complementada por análise subjetiva, ainda carece de *frameworks* mais robustos para avaliação semântica e

relevância contextual dos testes gerados. Por fim, a análise foi limitada pelo escopo temporal do trabalho, não abordando aspectos de evolução e manutenção dos testes em cenários de longo prazo.

7.4. **Trabalhos Futuros**

As limitações identificadas e as percepções obtidas abrem diversas oportunidades para pesquisas futuras:

- **Expansão de domínio**: Investigação da aplicabilidade da abordagem em sistemas financeiros de maior porte e complexidade, incluindo sistemas bancários core e plataformas de alta frequência.
- **Especialização de modelos**: Desenvolvimento e avaliação de modelos com *fine-tuning* específico para o domínio financeiro, melhorando potencialmente a relevância e precisão dos testes gerados.
- **Métricas avançadas**: Criação de frameworks de avaliação que incorporem métricas semânticas, cobertura de regras de negócio e densidade de validação, superando as limitações das métricas tradicionais.
- **Abordagens híbridas**: Investigação de soluções que combinem LLMs com técnicas tradicionais de ATG, maximizando potencialmente as vantagens de ambas as abordagens.

7.5. **Considerações Finais**

Este trabalho representa um passo importante na exploração do potencial dos LLMs para automação de validação de software em domínios críticos. Embora desafios permaneçam, especialmente relacionados à consistência e relevância dos testes gerados, os resultados indicam que esta tecnologia está suficientemente madura para aplicação prática, desde que implementada com as devidas precauções e estratégias de validação.

A crescente sofisticação dos modelos de linguagem e o desenvolvimento de técnicas de engenharia de prompts mais avançadas sugerem que muitas das limitações atuais serão superadas em futuras iterações desta tecnologia. Para a comunidade acadêmica e profissional, a pesquisa oferece base sólida para exploração continuada desta fronteira promissora da engenharia de software.

REFERÊNCIAS

[1] AMMANN, P.; OFFUTT, J. Introduction to software testing. Cambridge, United Kingdom ; New York, Ny, Usa: Cambridge University Press, 2008. Disponível em: https://ebooks.allfree-stuff.com/eBooks_down/Software%20Testing/Introduction%20to%20Software%20Testing.pdf Acesso em: 10 jun. 2025

[2] PEZZÈM.; YOUNG, M. Software testing and analysis : process, principles, and techniques. Hoboken, N.J.: Wiley, 2008. Disponível em: https://ix.cs.uoregon.edu/~michal/book/Samples/book.pdf Acesso em: 17 jul. 2025

[3] Accelerating Financial Services with NVIDIA AI. Disponível em: https://www.nvidia.com/en-eu/industries/finance/ai-financial-services-report/ Acesso em: 17 jul. 2025.

[4] BAGATTINI, G. et al. LEVERAGING LARGE LANGUAGE MODELS IN FINANCE: Pathways to Responsible Adoption. Disponível em: https://www.esma.europa.eu/document/working-paper-leveraging-large-language-models-finance-pathways-responsible-adoption Acesso em: 12 jul. 2025.

[5] CADAR, C.; SEN, K. Symbolic execution for software testing. Communications of the ACM, v. 56, n. 2, p. 82, 1 fev. 2013. Disponível em: https://people.eecs.berkeley.edu/~ksen/papers/cacm13.pdf Acesso em: 14 jul. 2025.

[6] FRASER, G.; ARCURI, A. EvoSuite. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11, 2011. p. 209-210. Disponível em: https://www.evosuite.org/wp-content/papercite-data/pdf/esecfse11.pdf

[7] KONSTANTINOU, M.; DEGIOVANNI, R.; PAPADAKIS, M. Do LLMs generate test oracles that capture the actual or the expected program behaviour?, 28 out. 2024. Disponível em: https://arxiv.org/html/2410.21136v1 Acesso em: 18 jun. 2025

[8] AZANZA, M.; PÉREZ LAMANCHA, B.; PIZARRO, E. Tracking the Moving Target: A Framework for Continuous Evaluation of LLM Test Generation in Industry., 26 abr. 2025. Disponível em: https://arxiv.org/html/2504.18985v1 Acesso em: 25 jun. 2025

[9] PARK, D. et al. A Study on Performance Improvement of Prompt Engineering for Generative AI with a Large Language Model. Journal of web engineering, p. 1187–1206, 22 fev. 2024. Disponível em: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10452390 Acesso em: 17 jul. 2025

[10] SHERIFI, B.; SLHOUB, K.; NEMBHARD, F. The Potential of LLMs in Automating Software Testing: From Generation to Reporting. 31 dez. 2024. Disponível em: https://arxiv.org/html/2501.00217v1 Acesso em: 10 jul. 2025.

[11] CHEN, M. et al. Evaluating Large Language Models Trained on Code. **arXiv:2107.03374 [cs**], 14 jul. 2021. Disponível em: https://arxiv.org/pdf/2107.03374 Acesso em: 02 jul. 2025.

[12] PAN, R. et al. ASTER: Natural and Multi-language Unit Test Generation with LLMs. IBM Research, [s.d.]. Disponível em: https://arxiv.org/pdf/2409.03093 Acesso em: 17 jul. 2025

[13] XUE, Z. et al. LLM4Fin: Fully Automating LLM-Powered Test Case Generation for FinTech Software Acceptance Testing. v. 12, p. 1643–1655, 11 set. 2024. Disponivel em: https://dl.acm.org/doi/10.1145/3650212.3680388 Acesso em: 17 jul. 2025.

[14] SIDDIQ, M. L. et al. An Empirical Study of Unit Test Generation with Large Language Models. **arXiv:2406.18181**, 26 jun. 2024. Disponível em: https://arxiv.org/html/2406.18181v1 Acesso em:13 ago. 2025.