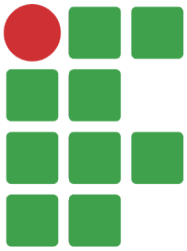
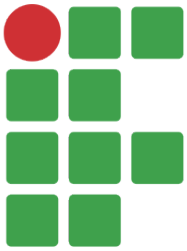


Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Curso Superior Bacharelado em Engenharia de Computação

Geração automática de Testes de Unidade com LLMs em projetos de larga escala

Maria Eduarda Pereira de Souza Melo

Orientador: Prof. Henrique do Nascimento Cunha



Instituto Federal de Educação, Ciência e Tecnologia da Paraíba
Campus Campina Grande
Coordenação do Curso Superior Bacharelado em Engenharia de Computação

Geração automática de Testes de Unidade com LLMs em projetos de larga escala

Maria Eduarda Pereira de Souza Melo

Trabalho de conclusão de curso apresentado ao Curso de Bacharelado em Engenharia de Computação, do Instituto Federal de Educação, Ciência e Tecnologia da Paraíba - *Campus* Campina Grande, em cumprimento às exigências parciais para a obtenção do título de Bacharel em Engenharia de Computação.

Orientador: Prof. Henrique do Nascimento Cunha.

Campina Grande, Dezembro de 2025

Catalogação na fonte:
Ficha catalográfica elaborada por Gustavo César Nogueira da Costa – CRB 15/479

M528g Melo, Maria Eduarda Pereira de Souza.
Geração automática de testes de unidade com LLMs em projetos de larga escala / Maria Eduarda Pereira de Souza Melo. - Campina Grande, 2025.
40 f.: il.

Trabalho de Conclusão de Curso (Graduação em Bacharelado em Engenharia de Computação) - Instituto Federal da Paraíba, 2025.
Orientador: Prof. Henrique do Nascimento Cunha

1. Engenharia de computação. 2. Engenharia de software. 3. Inteligência artificial. 4. Modelos de linguagem. I. Cunha, Henrique do Nascimento. II Título.

CDU 004.45

Maria Eduarda Pereira de Souza Melo

Prof. Henrique do Nascimento Cunha

Prof. Igor Barbosa da Costa
Membro da Banca

Prof. Paulo Ribeiro Lins Junior
Membro da Banca

Campina Grande, Paraíba, Brasil
Dezembro/2025

Agradecimentos

Agradeço primeiramente à minha família, pelo amor, incentivo e apoio incondicional ao longo de toda a minha trajetória acadêmica.

Ao meu namorado, pela paciência, compreensão e presença constante nos momentos de maior desafio.

Registro também minha gratidão ao Laboratório do IDEIA, ambiente no qual tive a oportunidade de aplicar na prática os conhecimentos adquiridos durante a graduação e amadurecer profissionalmente.

A todos que, de alguma forma, contribuíram para a realização deste trabalho, deixo aqui minha sincera gratidão.

Resumo

Este trabalho investiga a viabilidade e a eficácia da geração automática de testes de unidade por Large Language Models (LLMs) aplicada a projetos de software de larga escala. A pesquisa se apoia em um estudo empírico conduzido com três modelos de código aberto (Qwen 2.5 Coder 14B, Qwen 2.5 Coder 32B e DeepSeek Coder V2), executados em múltiplos ambientes computacionais, e avaliados sobre cinco sistemas reais escritos em Java. A metodologia combinou métricas quantitativas, como taxa de compilação, cobertura de código e análise estrutural, e métricas qualitativas, como legibilidade e relevância semântica dos testes gerados.

Os resultados mostram que, apesar do potencial dos LLMs, seu desempenho é altamente sensível à complexidade dos projetos, apresentando taxas de sucesso consistentes apenas em sistemas de menor acoplamento. O principal gargalo encontrado foi a baixa cobertura de branch, que impossibilitou a aplicação da Análise de Mutação com confiabilidade. A análise qualitativa revelou recorrência de problemas como alucinação de métodos, incompatibilidade de tipos genéricos, uso inadequado de anotações e compreensão limitada do contexto global do projeto.

Conclui-se que os LLMs atuam hoje como assistentes de programação com desempenho semelhante ao de desenvolvedores iniciantes, exigindo supervisão humana e retrabalho substancial. O estudo também demonstra que modelos maiores não são universalmente superiores, apresentando complementaridade e pontos cegos distintos. Como trabalhos futuros, sugere-se aprimorar a engenharia de prompts, adotar técnicas de aumento de contexto e explorar a orquestração de múltiplos modelos.

Palavras-chave: testes de unidade; Large Language Models; automação de testes; cobertura de código; engenharia de software.

Abstract

This work investigates the feasibility and effectiveness of automatic unit test generation using Large Language Models (LLMs) applied to large-scale software projects. The research is supported by an empirical study conducted with three open-source models (Qwen 2.5 Coder 14B, Qwen 2.5 Coder 32B, and DeepSeek Coder V2), executed in multiple computational environments, and evaluated on five real-world systems written in Java. The methodology combined quantitative metrics, such as compilation rate, code coverage, and structural analysis, with qualitative metrics, such as readability and semantic relevance of the generated tests.

The results show that, despite the potential of LLMs, their performance is highly sensitive to project complexity, presenting consistent success rates only in systems with lower coupling. The main bottleneck found was the low branch coverage, which prevented the reliable application of Mutation Testing. The qualitative analysis revealed recurrent issues such as method hallucination, generic type incompatibility, improper use of annotations, and limited understanding of the global project context.

It is concluded that LLMs currently act as programming assistants with performance similar to novice developers, requiring human supervision and substantial rework. The study also demonstrates that larger models are not universally superior, presenting complementarity and distinct blind spots. Future work suggests improving prompt engineering, adopting context augmentation techniques, and exploring multi-model orchestration.

Keywords: unit tests; Large Language Models; test automation; code coverage; software engineering.

Sumário

Lista de Siglas e Abreviaturas

API	Application Programming Interface (Interface de Programação de Aplicação)
AST	Abstract Syntax Tree (Árvore de Sintaxe Abstrata)
CPU	Central Processing Unit (Unidade Central de Processamento)
GPU	Graphics Processing Unit (Unidade de Processamento Gráfico)
IA	Inteligência Artificial
ICSE	International Conference on Software Engineering
IDE	Integrated Development Environment (Ambiente de Desenvolvimento Integrado)
ISO/IEC	International Organization for Standardization / International Electrotechnical Commission
LLM	Large Language Models (Modelo de linguagem geral)
RAM	Random Access Memory (Memória de Acesso Aleatório)
RAG	Retrieval-Augmented Generation (Geração Aumentada por Recuperação)
SBST	Search-Based Software Testing (Teste de Software Baseado em Busca)
SBES	Simpósio Brasileiro de Engenharia de Software
SSD	Solid State Drive (Unidade de Estado Sólido)
ULT	UnLeakedTestbench (Banco de Testes Não Vazado)
VM	Virtual Machine (Máquina Virtual)
VRAM	Video Random Access Memory (Memória de Acesso Aleatório de Vídeo)

Lista de Figuras

Capítulo 1

Introdução

1.1 Contextualização

A evolução da engenharia de *software* tem sido marcada pela busca contínua por maior qualidade e confiabilidade dos sistemas, especialmente em ambientes críticos. Nesse contexto, os testes de software desempenham um papel essencial. Segundo (PRESSMAN, 2010), o teste de software é a atividade que consiste em executar um programa ou sistema com o objetivo de encontrar erros. Trata-se de um processo de verificação e validação fundamental para assegurar que as funcionalidades se comportem conforme o esperado e que alterações no código não introduzam falhas. No entanto, o desafio se intensifica quando se busca atender às métricas de qualidade da indústria, que frequentemente exigem uma cobertura de testes acima de 80% para componentes críticos. Alcançar e manter esses patamares através da escrita manual de testes de unidade representa um esforço substancial, consumindo entre 20% a 40% do tempo total de desenvolvimento e tornando-se uma atividade repetitiva e frequentemente negligenciada em cronogramas apertados. Assim, a introdução de novas técnicas de automação, impulsionadas por LLMs, representa uma oportunidade de reduzir custos e aumentar a robustez dos processos de validação.

Embora a comunidade científica tenha desenvolvido diversas técnicas de geração automática de casos de teste, como geração baseada em modelos, *combinatorial testing* ou técnicas de *search-based testing*,o desafio de reduzir custos, aumentar a escalabilidade/praticabilidade industrial e manter a eficácia/qualidade dos testes gerados ainda persiste, conforme amplamente documentado na literatura (PAPADAKIS et al., 2019; ANAND et al., 2013).

Nesse cenário, os LLMs emergem como uma abordagem promissora, pois oferecem o potencial de gerar testes a partir da compreensão semântica do código-fonte, um paradigma distinto das técnicas algorítmicas convencionais. Apesar desse potencial, seu uso ainda carece de estudos sistemáticos em ambientes industriais. Grande parte dos trabalhos existentes se concentra em *benchmarks* acadêmicos, frequentemente limitados a pequenos repositórios de código aberto. Essa distância entre pesquisa e prática cria uma lacuna que este estudo busca reduzir, explorando o impacto real da aplicação de LLMs em projetos extensos e de alta complexidade.

Nos últimos anos, a inteligência artificial generativa, impulsionada por modelos de linguagem

de larga escala, tem se consolidado como uma tecnologia transversal na engenharia de software. Além da geração de testes, LLMs já vêm sendo aplicados em atividades como autocompletar código, refatoração, documentação automática, geração de cenários de integração e suporte ao debugging. Essas aplicações demonstram que os LLMs não apenas reduzem tarefas repetitivas, mas também atuam como assistentes cognitivos. Estudos de larga escala, como a análise de produtividade do *GitHub Copilot*, evidenciam ganhos significativos na velocidade de desenvolvimento (PENG et al., 2023), ao mesmo tempo que revelam novas formas de interação entre desenvolvedores e sistemas. A incorporação dessas ferramentas ao ciclo de desenvolvimento representa uma mudança de paradigma, como apontado por recentes revisões da literatura (ZHANG et al., 2024): de um processo centrado na codificação manual para um modelo mais colaborativo, onde humanos e IA trabalham em sinergia para acelerar a entrega de software com maior qualidade.

Diante disso, o problema central desta pesquisa é a carência de estudos empíricos que avaliem sistematicamente a eficácia e o custo-benefício de diferentes LLMs para a geração de testes de unidade em sistemas industriais brasileiros de alta complexidade, que operam sob pressão de mudanças regulatórias. A contribuição deste trabalho se dá em dois níveis complementares: acadêmico e industrial. Do ponto de vista acadêmico, busca-se oferecer um estudo empírico robusto sobre a eficácia de LLMs na geração de testes de unidade, trazendo evidências que dialogam com a literatura existente e apontando novas direções de pesquisa. Do ponto de vista industrial, a análise fornece subsídios práticos para empresas que avaliam a adoção de LLMs em seus fluxos de desenvolvimento, identificando tanto os ganhos possíveis quanto os desafios técnicos e organizacionais envolvidos.

1.2 Objetivos

1.2.1 Objetivo geral

O objetivo geral deste trabalho é analisar a viabilidade e a eficácia da geração automática de testes de unidade utilizando Modelos de Linguagem de Grande Escala (LLMs) aplicada a projetos de software de larga escala, identificando seus benefícios, desafios e limitações práticas.

1.2.2 Objetivos específicos

- Investigar o estado da arte da geração automática de testes de unidade com LLMs;
- Executar experimentos de geração de testes com múltiplos modelos de LLM e múltiplos projetos;
- Avaliar quantitativamente os testes gerados;
- Analisar qualitativamente a utilidade e a complexidade dos testes;
- Identificar os desafios e as limitações para a adoção em larga escala;

1.3 Organização do trabalho

Este trabalho está estruturado da seguinte forma:

1.3.1 Fundamentação teórica

O capítulo de fundamentação teórica apresenta os conceitos essenciais para a compreensão desta pesquisa. São discutidos os testes de unidade, o funcionamento dos *Large Language Models*, bem como as abordagens existentes que aplicam LLMs à geração automática de testes. Também são incluídos trabalhos relacionados que contextualizam o estado da arte.

1.3.2 Metodologia

Neste capítulo são descritos os procedimentos adotados para o desenvolvimento da pesquisa. São detalhados o funcionamento da ferramenta *ChatUniTest*, a metodologia de avaliação dos modelos e os critérios utilizados nos experimentos para garantir a reprodutibilidade e a comparabilidade dos resultados.

1.3.3 Resultados e Discussão

Este capítulo apresenta os resultados obtidos nos experimentos e discute suas implicações à luz dos objetivos definidos. São comparados os diferentes modelos utilizados, com análise crítica sobre pontos fortes, limitações e oportunidades de melhoria.

1.3.4 Conclusões e Trabalhos Futuros

Por fim, este capítulo apresenta as conclusões da pesquisa, destacando suas contribuições, limitações e possíveis desdobramentos futuros, como a inclusão de novos modelos, aplicação em outros contextos e ampliação da ferramenta.

Capítulo 2

Fundamentação Teórica

2.1 A Importância e os Desafios dos Testes de Unidade

A qualidade de um *software* está intrinsecamente ligada a atributos como testabilidade, manutenibilidade e a capacidade de resistir a regressões ao longo de seu ciclo de vida. Nesse contexto, os testes de unidade desempenham um papel fundamental, pois validam o comportamento de componentes de forma isolada, facilitam a detecção precoce de erros e servem como uma documentação executável, garantindo que cada parte do código funcione conforme o esperado (SCHÄFER et al., 2023).

O modelo ISO/IEC 25010:2011 define oito características principais de qualidade: adequação funcional, eficiência de desempenho, compatibilidade, usabilidade, confiabilidade, segurança, manutenibilidade e portabilidade. Essas características servem como referência internacional (International Organization for Standardization, 2011). Entre esses atributos, a confiabilidade e a manutenibilidade, que diz respeito à facilidade de modificação e evolução do código, possuem relação direta com práticas de teste consolidadas na área (SOMMERVILLE, 2019)

A importância do teste é reforçada por estudos que demonstram as consequências de sua ausência. Uma análise em larga escala, por exemplo, revelou que a cobertura de código é um preditor significativo de defeitos pós-lançamento, o que implica em maiores custos de manutenção e menor estabilidade (KOCHHAR et al., 2017). Nesse sentido, a automação da geração de testes, especialmente com o uso de LLMs, surge como uma estratégia promissora para apoiar a manutenção contínua e a confiabilidade de sistemas em larga escala.

Apesar de sua relevância, a criação e manutenção de suítes de testes de unidade robustas é uma tarefa intensiva em recursos, exigindo tempo e um profundo conhecimento da base de código. Em sistemas de grande porte ou legados, a tarefa torna-se ainda mais desafiadora, levando a uma cobertura de testes inadequada e ao acúmulo de débito técnico. Essa lacuna entre a necessidade de testes e a capacidade de produzi-los manualmente tem impulsionado a busca por abordagens automatizadas (GU et al., 2024).

2.2 Limitações da Cobertura e a Análise de Mutação

Embora a cobertura de código seja a métrica mais difundida para avaliar suítes de testes, ela possui limitações intrínsecas. Estudos em larga escala, como KOCHHAR et al. (2017), apontam que, apesar de a cobertura ser um preditor de defeitos pós-lançamento, ela avalia apenas quais linhas foram executadas, e não necessariamente se o comportamento do software foi verificado corretamente. É possível, por exemplo, ter 100% de cobertura de linha com asserções triviais ou inexistentes, o que gera uma falsa sensação de segurança.

Para mitigar essa deficiência, a Análise de Mutação (*Mutation Testing*) surge como um critério mais robusto para avaliar a qualidade dos testes ("testar os testes"). A técnica consiste na injeção deliberada de pequenos defeitos sintáticos no código original, chamados de "mutantes", para verificar se a suíte de testes é capaz de detectá-los.

O funcionamento baseia-se em dois conceitos principais:

- **Mutante Morto (*Killed*)**: Quando um teste falha ao executar a versão alterada do código. Isso é o desejável, pois indica que o teste foi capaz de perceber a mudança de comportamento.
- **Mutante Sobrevivente (*Survived*)**: Quando os testes continuam passando mesmo com a alteração no código. Isso indica uma lacuna na qualidade do teste (asserção fraca ou cenário não coberto).

No contexto de geração automática por inteligência artificial, ABDULLIN; DERAKHSHANFAR; PANICHELLA (2025) destacam que ferramentas baseadas em LLMs tendem a superar abordagens tradicionais (como SBST) na pontuação de mutação, sugerindo que os modelos de linguagem geram testes semanticamente mais relevantes e capazes de detectar falhas lógicas, e não apenas percorrer caminhos de execução.

Portanto, a pontuação de mutação é frequentemente considerada um "padrão-ouro"na pesquisa de testes de software, pois avalia não a quantidade de código exercitado, mas a capacidade efetiva da suíte de testes em revelar falhas (*Fault Detection Capability*). Ferramentas como o *PITEST* são comumente utilizadas em ambiente Java para automatizar a geração desses mutantes e o cálculo do escore de mutação.

2.3 Geração Automática de Testes com *Large Language Models*

Com o avanço dos Large Language Models (LLMs), emergiu uma nova fronteira para a automação de tarefas de engenharia de software. Até então, o estado da arte era dominado por abordagens algorítmicas clássicas, consolidado pelo trabalho de FRASER; ARCURI (2014) com a ferramenta *EvoSuite*. Essa abordagem, conhecida como *Search-Based Software Testing* (SBST), utiliza algoritmos genéticos para explorar heuristicamente o espaço de entradas e maximizar métricas objetivas, como a cobertura de código. No entanto, embora eficazes na cobertura estrutural, esses métodos frequentemente geram testes de difícil compreensão humana. Em contraste, os

LLMs operam com base em um vasto conhecimento contextual extraído de dados de código-fonte. Essa capacidade permite que eles interpretem a semântica do código, compreendam convenções de nomenclatura e gerem testes com uma naturalidade que se assemelha à de um desenvolvedor humano.

2.3.1 Estudos Empíricos e o Estado da Arte

O marco fundamental para a validação de LLMs na escrita de código foi estabelecido por CHEN et al. (2021), com o desenvolvimento do *Codex* e a proposição do benchmark *HumanEval*. Este trabalho introduziu a métrica *pass@k*, que avalia a probabilidade de pelo menos uma solução correta ser gerada dentro de k tentativas, reconhecendo formalmente a natureza estocástica (não determinística) dos modelos generativos. Essa métrica tornou-se o padrão para medir a capacidade funcional de modelos de código, demonstrando que o pré-treinamento em grandes volumes de código-fonte aberto permite que a IA compreenda sintaxe e lógica de programação em um nível muito superior ao de modelos estatísticos anteriores.

Estudos empíricos recentes apresentam um cenário de resultados mistos, revelando um claro *trade-off* entre as abordagens. Em um estudo comparativo em larga escala, ABDULLIN; DE-RAKHSHANFAR; PANICHELLA (2025) demonstraram que ferramentas baseadas em LLMs, embora fiquem atrás de SBST e execução simbólica em métricas tradicionais como cobertura de linha e de ramo, superam significativamente essas abordagens na pontuação de mutação. Este resultado sugere que os LLMs geram testes semanticamente mais relevantes, capazes de detectar uma gama mais ampla de falhas lógicas. Por outro lado, para a detecção de falhas reais, definidas como defeitos genuínos introduzidos por desenvolvedores durante o ciclo de vida do software (em oposição às falhas artificiais dos mutantes) (JUST et al., 2014), as ferramentas tradicionais ainda se mostraram mais eficazes.

No contexto acadêmico brasileiro, essa variabilidade de desempenho também foi objeto de investigação recente. SILVA E. C. O. (2025), em estudo apresentado no Simpósio Brasileiro de Engenharia de Software (SBES), conduziram um *benchmarking* comparativo que reforça a heterogeneidade dos resultados entre diferentes LLMs. A pesquisa destaca que a eficácia da geração não depende apenas do tamanho do modelo, mas de como sua arquitetura se adapta às especificidades da linguagem de programação e dos *frameworks* de teste alvo, validando a necessidade de estudos empíricos que confrontem múltiplas famílias de modelos em cenários controlados.

Expandindo a análise para além da arquitetura dos modelos, a influência da linguagem de programação alvo também se mostrou um fator determinante. PAN et al. (2025), em trabalho apresentado na International Conference on Software Engineering (ICSE), investigaram a capacidade multi-linguagem dos LLMs e identificaram que a eficácia da geração de testes não é uniforme. Os resultados indicaram que linguagens verbosas e estaticamente tipadas, como Java, impõem desafios superiores de contexto e sintaxe em comparação com linguagens dinâmicas, como Python. Essa constatação é fundamental para esta pesquisa, pois o ecossistema Java exige uma gestão rigorosa de tipos, importações e configurações de *mock*, aumentando a complexidade da tarefa de

geração e a probabilidade de erros de compilação.

Aprofundando a análise no ecossistema Java, SIDDIQ et al. (2024), em estudo empírico apresentado na conferência EASE, focaram exclusivamente na viabilidade da geração de testes com o *framework* JUnit. Os autores constataram que, embora os LLMs demonstrem alta proficiência na construção da estrutura básica dos casos de teste (setup, execução e asserção), a taxa de testes que compilam e executam com sucesso na primeira tentativa (*zero-shot*) enfrenta barreiras significativas. As principais causas de falha identificadas, como alucinação de métodos auxiliares e configuração incorreta de *mocks*, alinham-se diretamente aos obstáculos observados em projetos industriais, sugerindo que a eficácia da ferramenta depende menos da capacidade criativa do modelo e mais da sua precisão em inferir o contexto de dependências do projeto.

Além das métricas de desempenho e desafios linguísticos, a natureza comportamental dos testes gerados é um ponto de atenção crítica. KONSTANTINOU; DEGIOVANNI; PAPADAKIS (2024) investigaram se os oráculos de teste (asserções) produzidos por LLMs capturam o comportamento esperado (correto) do software ou apenas espelham o comportamento atual (potencialmente incorreto) do código fornecido no prompt. O estudo concluiu que existe um forte viés para que os modelos gerem oráculos baseados na implementação observada, atuando efetivamente como ferramentas de geração de testes de regressão. Isso implica que, embora os LLMs sejam excelentes para garantir que alterações futuras não quebrem funcionalidades existentes, eles podem falhar em identificar bugs lógicos pré-existentes, pois tendem a validar o código "como ele é", exigindo do desenvolvedor uma revisão minuciosa para garantir a correção funcional.

Na indústria, a viabilidade da aplicação em larga escala foi comprovada por ferramentas como o *TestGen-LLM*, da Meta, que gerou ganhos concretos em cobertura e qualidade em plataformas como Facebook e Instagram (AI, 2024). Paralelamente, o TestPilot (SCHÄFER et al., 2023) demonstrou que modelos generalistas, mesmo sem *fine-tuning*, podem produzir testes funcionais através de uma engenharia de *prompt* sofisticada, que inclui não apenas o código-alvo, mas também exemplos de seu uso.

2.3.2 Técnicas de Gestão de Contexto

A janela de contexto (*context window*) é um parâmetro técnico fundamental dos LLMs que define a quantidade máxima de informação medida em *tokens* (fragmentos de texto) que o modelo pode processar em uma única interação. Na prática, essa janela funciona como a memória de curto prazo do modelo. Essa capacidade limitada representa um desafio crítico na engenharia de software, pois bases de código extensas, com suas múltiplas dependências e arquivos interrelacionados, frequentemente extrapolam o tamanho da janela, impedindo que o modelo tenha uma visão completa do projeto para realizar a tarefa (SCHÄFER et al., 2023).

Para mitigar essa restrição arquitetural, a literatura recente aponta para o uso de Retrieval-Augmented Generation (RAG) aplicado especificamente à engenharia de software. Conforme detalhado em um amplo levantamento realizado por HONG; BAIK (2025), o RAG para código difere da geração tradicional ao introduzir um componente de recuperação que busca dinamicamente in-

formações externas relevantes, como definições de classes dependentes, documentação técnica ou exemplos de testes similares, em uma base indexada antes da geração. Essa abordagem permite que o LLM acesse conhecimentos específicos do projeto que não estavam presentes em seu treinamento original, reduzindo alucinações e permitindo a manipulação de bases de código que excedem largamente o limite da janela de contexto, sem os custos proibitivos de um *fine-tuning* contínuo.

Nesse cenário, para que o ciclo seja eficaz em projetos de grande escala, é necessário superar a limitação da janela de contexto dos LLMs. Essa necessidade de curadoria estratégica de informações consolidou uma nova disciplina denominada Engenharia de Contexto (*Context Engineering*). Diferentemente da Engenharia de Prompt, que foca na otimização das instruções, OSMANI (2024) define a Engenharia de Contexto como a prática de selecionar, organizar e formatar os dados de entrada (como *snippets* de código, definições de tipos e diagramas de arquitetura) para maximizar o raciocínio do modelo dentro de suas restrições computacionais. O objetivo é reduzir o ruído e garantir que o LLM tenha acesso apenas às dependências essenciais para a tarefa. A técnica de contexto focal adaptativo é crucial nesse processo, pois atua como um mecanismo de seleção inteligente, identificando e extraindo apenas os trechos de código estritamente necessários para a tarefa (como a assinatura do método, suas dependências diretas e exemplos de uso). Ao otimizar o prompt com informações de alta relevância, a técnica garante que o LLM compreenda a semântica do código e gere testes mais precisos.

Nessa linha de aprimoramento da engenharia de contexto, GU; NASHID; MESBAH (2025) propuseram o Panta, uma abordagem que supera as limitações da recuperação puramente textual (como o RAG tradicional ou a busca por palavras-chave). O Panta introduz a Análise de Programa Híbrida e Iterativa, combinando análise estática com traços de execução dinâmica para construir *slices* (fatias) de programa precisos. Ao rastrear as dependências de dados e controle durante a execução real do código, essa abordagem garante que o LLM receba no prompt não apenas o método focal, mas todas as definições externas que influenciam seu comportamento em tempo de execução, reduzindo drasticamente os erros de compilação causados por alucinação de tipos ou dependências desconhecidas.

2.3.3 Técnicas Avançadas de Geração e Reparo

Para superar as limitações da geração de testes em uma única etapa (*one-shot*), onde os LLMs frequentemente produzem código que não compila ou contém erros lógicos, a comunidade de pesquisa desenvolveu estratégias mais sofisticadas e interativas. Uma das abordagens mais proeminentes é o ciclo iterativo de **geração–validação–reparo**, implementado por ferramentas como o *ChatUniTest* (ZJU-ACES-ISE, 2023). Este ciclo transforma a geração de testes em um processo dinâmico e corretivo, composto por três fases fundamentais:

1. **Geração:** Na primeira fase, o LLM recebe uma instrução (*prompt*) contendo o código-alvo e gera uma primeira versão do teste de unidade.

2. **Validação:** Em vez de assumir que o teste é correto, a ferramenta o submete a uma validação automatizada, compilando e executando o código. O resultado desta fase é um *feedback* de sucesso ou falha e, neste último caso, informações de diagnóstico como erros de compilação ou *stack traces*.

3. **Reparo (Autocorreção):** Se a validação falhar, o sistema utiliza o *feedback* do erro para iniciar um novo ciclo. O erro retornado é incorporado a um novo prompt, instruindo o LLM a corrigir o problema específico. Este processo de autocorreção pode ser repetido, permitindo que o modelo refine iterativamente sua solução.

Além das melhorias de processo, surgem estratégias neuro-simbólicas focadas em maximizar a cobertura estrutural. LEMIEUX et al. (2023) apresentaram na ICSE o CodaMosa, uma abordagem híbrida projetada para escapar dos 'platôs de cobertura', momentos em que algoritmos de busca tradicionais (SBST) estagnam por não conseguirem satisfazer condições complexas de desvio. O método opera alternando entre a busca evolucionária e a geração por LLM: quando a busca trava, o modelo de linguagem é acionado para sintetizar casos de teste que exercitem os caminhos inexplorados (*uncovered branches*), reinjetando diversidade na população de testes. Essa sinergia sugere que a solução para a baixa cobertura de *branch*, um dos principais gargalos identificados na literatura, reside na orquestração das capacidades complementares da exploração algorítmica e da inferência semântica.

Indo além, o TestART (GU et al., 2024) implementa um mecanismo de **coevolução entre geração e reparo**, onde um módulo gerador e um módulo reparador evoluem em conjunto, utilizando o *feedback* de cobertura e *templates* de reparo para otimizar a taxa de sucesso e a qualidade geral dos testes. Diferentemente do ciclo reativo do *ChatUniTest*, a abordagem do *TestART* especializa as tarefas: o módulo gerador foca em criar novos testes, enquanto o módulo reparador aprende a corrigir falhas de forma mais eficiente, muitas vezes a partir de padrões de erro comuns. A utilização do *feedback* de cobertura também representa um avanço, pois permite que a ferramenta otimize não apenas testes que falham, mas também aqueles que passam, mas que possuem baixa qualidade (baixa cobertura de código), incentivando a criação de uma suíte de testes mais robusta.

Embora o feedback baseado em cobertura, utilizado por ferramentas como o TestART, garanta que o código seja exercitado, ele não assegura a qualidade das verificações (asserções). Para mitigar essa limitação, abordagens mais recentes propõem a Geração de Testes Guiada por Mutação (Mutation-Guided Unit Test Generation). Segundo WANG et al. (2024), integrar a análise de mutação diretamente no ciclo de geração permite que o modelo refine os testes com base na capacidade de detectar falhas, e não apenas na execução de linhas. Nessa abordagem, os mutantes sobreviventes servem como sinais de feedback para o LLM, indicando que o teste gerado, embora sintaticamente correto, falhou em capturar a semântica do código ou possui asserções triviais. Isso representa uma evolução em relação à geração guiada por cobertura, pois força o modelo a criar cenários de teste mais rigorosos e assertivos desde a etapa de construção.

2.4 Desafios e Barreiras para Adoção Industrial

A transição dessas ferramentas do ambiente acadêmico para o fluxo de trabalho industrial enfrenta barreiras significativas. Os principais desafios incluem a geração de testes que não compilam ou falham na execução devido à "alucinação"de dependências ou à falta de compreensão do contexto global do projeto. Além disso, a produção de *asserts* genéricos ou triviais, que não validam o comportamento funcional crítico, continua sendo um problema.

Consequentemente, o alto custo de retrabalho para revisar, depurar e corrigir os testes gerados é uma das maiores barreiras à adoção, podendo, em alguns cenários, anular os ganhos de produtividade. Essa limitação qualitativa é corroborada por estudos recentes sobre Test Smells (maus cheiros em testes) em códigos gerados por IA. ALVES; BEZERRA; MACHADO (2024) conduziram uma análise empírica sobre testes produzidos pelo GitHub Copilot e constataram que, embora as ferramentas automatizem a escrita, elas frequentemente introduzem padrões de código inadequados que comprometem a manutenibilidade e a confiabilidade da suíte. A presença recorrente desses smells, como lógica condicional complexa dentro do teste ou verificações redundantes, evidencia que a alta produtividade na geração pode vir acompanhada de um aumento no débito técnico, exigindo revisão e refatoração humana substancial para garantir que os testes sejam não apenas executáveis, mas também limpos e sustentáveis a longo prazo

Para agravar, a falta de metodologias padronizadas para avaliação dificulta a comparação objetiva entre as ferramentas. Em resposta, a comunidade de pesquisa está se movendo para além da cobertura de código, consolidando a pontuação de mutação como uma métrica de qualidade superior e propondo *benchmarks* mais realistas, como o ULT (*UnLeakedTestbench*), para avaliar a verdadeira capacidade de generalização dos modelos.

Além dos desafios relacionados à qualidade e ao retrabalho, a viabilidade econômica da integração de LLMs depende de um importante *trade-off* de infraestrutura: o uso de modelos via API de terceiros versus a execução de modelos localmente.

- **Modelos via API (GPT-4, Claude):** Esta abordagem oferece acesso aos modelos mais avançados sem a necessidade de um investimento inicial em *hardware*. No entanto, ela introduz custos operacionais variáveis (pagamento por *token*) que podem se tornar proibitivos em larga escala, especialmente em processos de integração contínua que executam testes massivamente. Ademais, surgem preocupações com a privacidade e a segurança, uma vez que o código-fonte proprietário da empresa precisa ser enviado a servidores externos.

- **Modelos Locais (CodeLlama, DeepSeek Coder):** A execução local, abordagem adotada nesta pesquisa, oferece controle total sobre os dados, eliminando questões de privacidade. Os custos se tornam mais previsíveis, concentrados no investimento inicial em *hardware* robusto (como GPUs com alta capacidade de VRAM e na manutenção do ambiente. Contudo, essa modalidade exige um maior custo computacional e de complexidade técnica para configurar e gerenciar a infraestrutura de inferência, além de potencialmente utilizar modelos com performance inferior aos maiores modelos comerciais.

A escolha da abordagem (API vs. local) também se reflete nas configurações experimentais encontradas na literatura. Trabalhos seminais como o TestPilot (SCHÄFER et al., 2023) e estudos comparativos como o "Test Wars"(ABDULLIN; DERAKHSHANFAR; PANICHELLA, 2025) focaram primariamente no uso de modelos proprietários de ponta via API, como o GPT-4 e a família Codex, abstraindo a necessidade de hardware local. Por outro lado, pesquisas que avaliam modelos abertos, como o TestART (GU et al., 2024), reportam o uso de infraestrutura de alto desempenho, como servidores equipados com GPUs de datacenter (NVIDIA A100), para executar modelos como o CodeLlama. Gigantes da indústria, como a Meta, utilizam seus próprios modelos Llama especializados e uma vasta infraestrutura interna para sua ferramenta TestGen-LLM (AI, 2024).

Essa análise evidencia que a maioria dos resultados de ponta publicados depende de recursos computacionais ou financeiros significativos. Este cenário reforça a relevância desta monografia, que busca avaliar a eficácia de modelos de código aberto executados em uma configuração de hardware robusta, porém mais acessível e representativa de um ambiente de desenvolvimento corporativo que não dispõe de recursos de hiperescala.

2.5 Síntese e Justificativa da Pesquisa

Diante do exposto, evidencia-se uma dicotomia: de um lado, o imenso potencial dos LLMs para gerar testes semanticamente ricos e, de outro, as barreiras pragmáticas que ainda limitam sua utilidade e confiabilidade no desenvolvimento de software em larga escala. Torna-se, portanto, imperativo não apenas explorar as capacidades dessas tecnologias, mas também investigar criticamente suas limitações e o custo real de sua integração. Esta pesquisa se insere nesse contexto, buscando identificar os cenários onde a geração automática de testes é viável e onde ela ainda carece de maturidade para gerar valor imediato na prática profissional.

Capítulo 3

Metodologia

Neste capítulo, é detalhado o plano de pesquisa adotado para atingir os objetivos propostos. A metodologia segue uma abordagem experimental de natureza mista, combinando análises quantitativas e qualitativas para avaliar a geração automática de testes de unidade por LLMs em um ambiente industrial.

3.1 Descrição do ambiente

Para garantir a validade e a fidedignidade dos resultados, o estudo foi conduzido em um ambiente controlado. A seguir, são descritos os componentes de hardware, software, os objetos de estudo e os procedimentos executados.

3.1.1 Ambiente Computacional

Para garantir a reprodutibilidade dos experimentos, todas as execuções foram realizadas em ambientes computacionais dedicados. As especificações de cada ambiente são detalhadas a seguir.

Ambiente 1: Estação de Trabalho Local

- **Processador (CPU):** AMD Ryzen 7 5700X
- **Memória RAM:** 64 GB DDR4
- **Placa-mãe:** Gigabyte B550 AORUS Elite
- **Unidades de Processamento Gráfico (GPU):** 2 x NVIDIA GeForce RTX 3060 (12 GB GDDR6 VRAM cada)
- **Armazenamento:** 1 TB NVMe M.2 SSD Kingston

Ambiente 2: Estação de Trabalho Local

- **Processador (CPU):** 13th Gen Intel Core i5-13400
- **Memória RAM:** 64 GB
- **Placa-mãe:** Gigabyte H610M H DDR4
- **Unidades de Processamento Gráfico (GPU):** 1 x NVIDIA GeForce RTX 3060
- **Armazenamento:** 240 GB SSD (A-Data SU650)

Ambiente 3: Instância de Nuvem (Oracle Cloud Infrastructure)

- **Shape da Instância:** VM.GPU.A10.1
- **Processador (CPU):** 15 OCPUs (baseado em arquitetura AMD EPYC)
- **Memória RAM:** 240 GB
- **Unidade de Processamento Gráfico (GPU):** 1 x NVIDIA A10 (24 GB GDDR6 VRAM)
- **Armazenamento:** Armazenamento em Bloco (Block Storage) remoto
- **Rede:** Largura de banda de 24 Gbps

A execução dos experimentos em múltiplos ambientes computacionais decorreu da disponibilidade de infraestrutura, caracterizando-se como uma contingência prática e não como uma variável do desenho experimental. Para assegurar a integridade dos resultados, manteve-se rigorosa consistência metodológica: todos os ambientes executaram as mesmas versões dos modelos de linguagem sob configurações idênticas na ferramenta ChatUniTest, utilizando parâmetros determinísticos (temperatura e seed fixas).

As distinções de hardware (CPU/GPU e memória) impactaram preponderantemente a latência de inferência e o gerenciamento de recursos, riscos estes mitigados pelo ajuste dinâmico do tamanho de lote (*batch size*) para evitar interrupções (*timeouts*). Embora a literatura técnica reconheça que variações de arquitetura possam introduzir diferenças numéricas sutis, não foram observadas discrepâncias sistemáticas na semântica ou na qualidade dos testes gerados entre as execuções. Portanto, considera-se que a heterogeneidade dos ambientes influenciou apenas o tempo de processamento, sem comprometer a validade comparativa dos dados.

3.2 Objetos de Estudo

A seleção das ferramentas, modelos e projetos foi um passo fundamental para garantir que os resultados da pesquisa fossem relevantes e representativos do contexto industrial.

3.2.1 Ferramenta de Geração de Testes

A ferramenta selecionada para orquestrar os experimentos foi o *ChatUniTest*. A mesma automatiza o pré-processamento (como a análise do código e a construção de *prompts* com contexto focal), a geração (extração do teste) e o pós-processamento, que valida o teste gerado e, em caso de falha, aciona um novo ciclo de reparo baseado em LLM.

Conforme ilustrado na **Figura 4.2**, sua arquitetura implementa de forma robusta o ciclo iterativo de geração-validação-reparo (discutido na Seção 2.3.2).

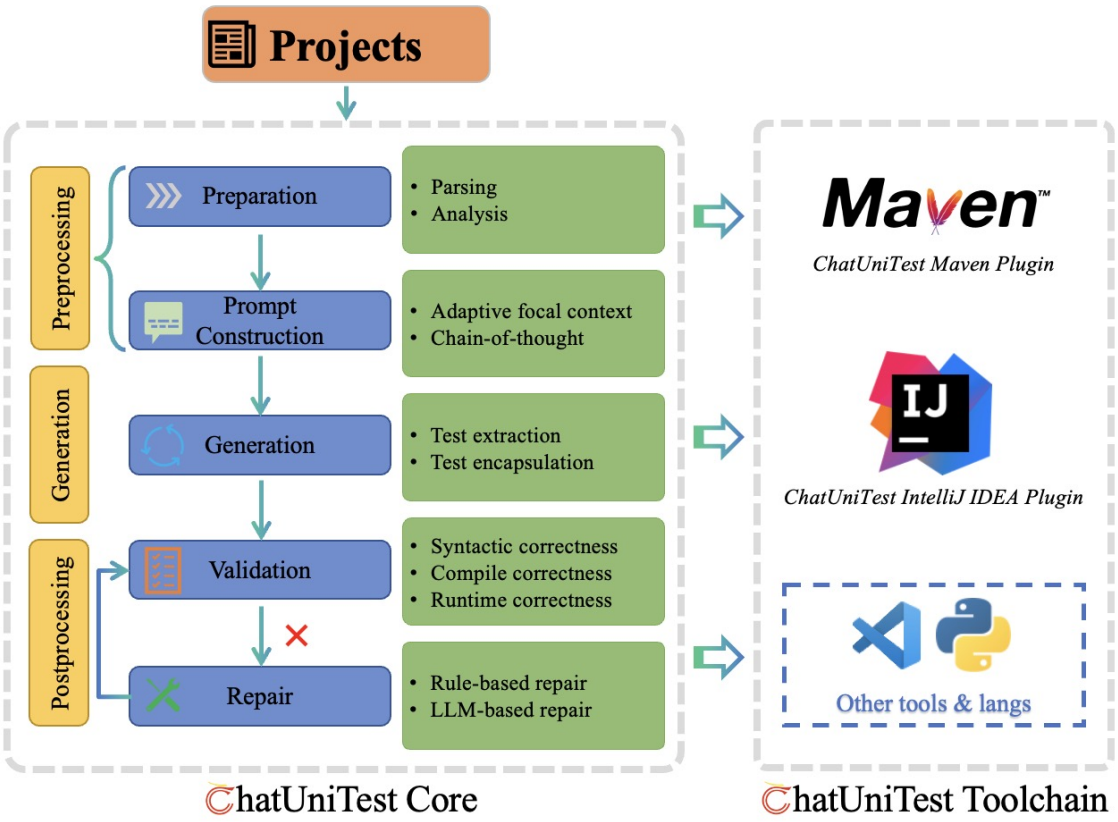


Figura 3.1: *Arquitetura e fluxo de processamento da ferramenta ChatUniTest, detalhando o ciclo de pré-processamento, geração e pós-processamento (validação e reparo).*

Fonte: Adaptado de ZJU-ACES-ISE (2023)

O funcionamento da ferramenta ocorre em um fluxo sequencial de três estágios principais, detalhados a seguir para esclarecer o processamento dos dados:

- **Pré-processamento e Análise Estrutural:** Diferentemente de abordagens que enviam o código bruto ao modelo, o ChatUniTest inicia com uma etapa rigorosa de Parsing e Análise. A ferramenta analisa a Árvore de Sintaxe Abstrata (AST) do projeto para decompor o código em componentes lógicos. Nesta fase, são identificados:
  - **Classe Focal:** A classe alvo para a qual os testes serão gerados.
  - **Dependências:** A ferramenta rastreia importações e injeta dependências necessárias, extraindo assinaturas de métodos e campos de classes externas que interagem com

a classe focal. Isso é crucial para que o LLM saiba como criar mocks corretos sem precisar ler o projeto inteiro.

- **Engenharia de Prompt e Contexto Focal Adaptativo:** Após a análise, a ferramenta constrói o prompt utilizando a técnica de *Adaptive Focal Context*(Contexto Focal Adaptativo). Este mecanismo seleciona dinamicamente quais informações devem compor o contexto com base no limite de tokens do modelo e na relevância semântica. O prompt final é estruturado seguindo o padrão *Chain-of-Thought* (Cadeia de Pensamento), instruindo o modelo a raciocinar passo a passo sobre a lógica do teste antes de gerar o código. O prompt inclui:

  (i) a assinatura da classe focal,

  (ii) o código do método a ser testado,

  (iii) assinaturas de métodos dependentes e

  (iv) regras de geração (ex: usar JUnit 5, priorizar cobertura de branch).

- **Pós-processamento: Validação e Reparo:** Uma vez gerado o teste, a ferramenta entra na fase de Extração e Encapsulamento, isolando o bloco de código Java da resposta textual do LLM. Em seguida, inicia-se o ciclo de validação:

  - **Verificação Sintática:** Checa se o código é Java válido.
  - **Compilação:** Tenta compilar o teste gerado. Se houver erro (ex: symbol not found), a ferramenta captura o log do compilador.
  - **Mecanismo de Reparo:** Em caso de falha, o ChatUniTest aciona um módulo de reparo que pode ser Baseado em Regras (para erros simples de import) ou Baseado em LLM (reenviando o erro ao modelo para que ele reescreva o código corrigido).

A escolha desta ferramenta, em detrimento de outras abordagens, foi estratégica e se justifica pelos seguintes motivos:

- **Suporte a Múltiplos LLMs:** Sua arquitetura permite a integração facilitada com diversos modelos de linguagem, tanto locais quanto via API, o que é essencial para uma análise comparativa.
- **Especialização em Testes de Unidade:** É uma ferramenta projetada especificamente para o contexto de geração de testes em Java, automatizando tarefas como a identificação de classes focais e a invocação dos modelos.
- **Controle e Reprodutibilidade:** Permite configurar e registrar os parâmetros de geração (como *temperature* e *prompts*), contribuindo para a reprodutibilidade dos resultados.
- **Código Aberto e Extensibilidade:** Por ser um projeto de código aberto, oferece transparência sobre seu funcionamento e permite futuras adaptações, se necessário.

3.2.2 Modelos de Linguagem (LLMs)

Foram selecionados três modelos de linguagem de grande escala, com foco em geração de código, para execução local. A escolha baseou-se em sua relevância na comunidade de desenvolvimento, performance reportada em *benchmarks* de programação e disponibilidade para uso *offline*. Diferentemente de modelos proprietários massivos acessados via API (como GPT-4 ou Claude 3.5 Sonnet, que operam na nuvem com trilhões de parâmetros estimados), a escolha recaiu sobre modelos de pesos abertos e parâmetros reduzidos (14B a 32B).

Essa escolha delimita o escopo da pesquisa ao cenário de 'IA On-Premise' (execução local), uma tendência crescente na indústria para garantir a privacidade dos dados (o código não sai da empresa) e a previsibilidade de custos (sem pagamento por token). Embora sejam ordens de grandeza menores que os modelos de fronteira (*state-of-the-art*), modelos como o Qwen 2.5 e o DeepSeek V2 utilizam técnicas modernas de treinamento e destilação de conhecimento para oferecer alta densidade de inteligência, viabilizando sua execução em hardware convencional. Os modelos utilizados foram:

- **Qwen 2.5 Coder (14B)**
- **DeepSeek Coder V2 (16B)**
- **Qwen 2.5 Coder (32B)**

3.2.3 Projetos-Alvo

Os experimentos foram conduzidos sobre projetos reais e completos de uma empresa de software, escritos em Java. A seleção dos projetos-alvo buscou garantir a representatividade do ambiente industrial, incluindo sistemas com diferentes domínios de negócio, complexidade arquitetural e dependências externas.

A seleção dos objetos de estudo buscou cobrir diferentes níveis de acoplamento e responsabilidades no sistema distribuído. O corpus analisado totaliza 32.507 linhas de código (LOC) e 433 classes, distribuídas em microsserviços que variam de APIs de integração a núcleos de processamento de regras de negócio. A complexidade desses sistemas é evidenciada pela alta integração com bibliotecas externas: juntos, os projetos gerenciam 129 dependências diretas (via Maven), refletindo um ambiente industrial heterogêneo e interconectado.

A Tabela 3.1 detalha as métricas quantitativas de cada projeto, caracterizando a complexidade do cenário experimental.

Para garantir a isonomia na comparação e a validade interna do experimento, foi estabelecido um rigoroso controle de variáveis. Isso significa que todos os modelos foram submetidos estritamente aos mesmos inputs: o mesmo prompt de sistema (instruções padronizadas), a mesma estrutura de prompt de usuário (contendo o código da classe focal e suas dependências) e os mesmos parâmetros de configuração da ferramenta. Dessa forma, assegura-se que qualquer variação nas

Tabela 3.1: *Caracterização Quantitativa dos Projetos-Alvo*

Projeto	Linhas de Código (LOC)*	Nº de Classes	Dependências Externas
Projeto 1	7.235	75	29
Projeto 2	9.380	112	24
Projeto 3	8.855	136	35
Projeto 4	5.508	83	22
Projeto 5	1.529	27	19
TOTAL	**32.507**	**433**	**129**

Fonte: Elaborado pelo autor (2025).

métricas de desempenho seja decorrente exclusivamente das capacidades de raciocínio e geração de código de cada LLM, e não de diferenças na formulação do problema ou no contexto fornecido.

3.3 Procedimento Experimental e Coleta de Dados

O processo experimental foi dividido em etapas sequenciais e controladas, executadas para cada um dos LLMs selecionados:

1. **Configuração:** Preparação do ambiente com versões fixas das dependências dos projetos e ferramentas para garantir consistência.

2. **Geração de Testes:** Utilização do ChatUniTest para instruir cada LLM a gerar um conjunto de testes de unidade para uma seleção de classes dos projetos-alvo. O processo experimental foi dividido em etapas sequenciais e controladas para cada LLM:

   (a) **Configuração:** Preparação do ambiente com versões fixas das dependências e ferramentas para garantir consistência.

   (b) **Geração de Testes:** Utilização do ChatUniTest para instruir cada LLM a gerar testes para o projeto. O processo inicia com um **prompt de sistema** (system prompt) padronizado, um recurso da própria ferramenta que define o papel do LLM e as diretrizes da tarefa. O prompt padrão utilizado é apresentado em sua forma original abaixo:

   > "Please help me generate a whole JUnit test for a focal method in a focal class. I will provide the following information of the focal method:
   > 1. Required dependencies to import.
   > 2. The focal class signature.
   > 3. Source code of the focal method.
   > 4. Signatures of r methods and fields in the class.
   > I will provide the following brief information if the focal method has dependencies:
   > 1. Signatures of dependent classes.
   > 2. Signatures of dependent methods and fields in the dependent classes.

> I need you to create a whole unit test using JUnit 5, ensuring optimal branch and line coverage. Compile without errors, and use reflection to invoke private methods. No additional explanations required."
>
> A partir desta instrução de sistema, a ferramenta compunha um **prompt de usuário** (user prompt) mais detalhado, contendo os trechos de código-fonte da classe focal e suas dependências, e o enviava para o modelo de linguagem para a geração do teste de unidade correspondente.

3. **Coleta de Dados Brutos:** Armazenamento automático de todos os arquivos de teste gerados, juntamente com os *logs* de execução, para análise posterior.

3.4 Métricas de Avaliação

Para obter uma visão abrangente sobre a eficácia de cada modelo, a avaliação dos testes gerados foi realizada com base em um conjunto de métricas quantitativas e qualitativas, detalhadas a seguir.

3.4.1 Métricas Quantitativas

A análise quantitativa focou em indicadores objetivos de performance e cobertura, utilizando ferramentas consolidadas no mercado.

1. **Taxa de Compilação e Execução:** Proporção de testes que puderam ser compilados e executados com sucesso (passaram) em relação ao total de testes gerados. Este é o primeiro indicador de viabilidade de um teste.

2. **Cobertura de Código:** Medida por meio da ferramenta de análise de cobertura integrada à IDE IntelliJ IDEA, analisando a cobertura de linha (*line coverage*) e de ramo (*branch coverage*). Indica o percentual de código-fonte que foi exercitado pelos casos de teste gerados.

3. **Pontuação de Mutação:** Obtida com a ferramenta *PITEST*, avalia a capacidade dos testes em detectar defeitos introduzidos artificialmente no código-fonte, conhecidos como *mutantes*. Uma pontuação alta sugere que os testes são robustos e capazes de encontrar falhas reais.

4. **Detecção de Falhas Reais:** Verificação da capacidade dos testes em identificar comportamentos incorretos ou exceções não tratadas em versões do código com defeitos previamente conhecidos.

3.4.2 Métricas de Qualidade

Para além dos dados numéricos, foi realizada uma análise qualitativa por inspeção manual de uma amostra dos testes gerados. O objetivo desta análise é avaliar a utilidade prática e a manuteni-

bilidade dos testes, aspectos fundamentais para sua adoção em um ambiente industrial. Os critérios avaliados foram:

- **Legibilidade e Manutenibilidade:** Análise da clareza, organização e simplicidade do código de teste. Um teste legível é mais fácil de ser compreendido, mantido e depurado por outros desenvolvedores no futuro.
- **Qualidade das Asserções (*Asserts*):** Verificação da relevância das validações realizadas. A análise diferencia asserções triviais (ex: verificar se um objeto não é nulo quando ele nunca poderia ser) de asserções significativas, que validam regras de negócio, estados específicos ou valores de retorno críticos.
- **Relevância Semântica dos Cenários:** Avaliação se os testes gerados representam cenários de uso realistas e relevantes para a funcionalidade-alvo. Testes com alta relevância semântica são mais propensos a encontrar bugs que impactariam o usuário final.

Capítulo 4

Resultados e Discussão

Esta seção apresenta os resultados obtidos nos experimentos de geração automática de testes de unidade utilizando Large Language Models (LLMs), conforme a metodologia descrita no Capítulo 3. A análise está dividida em três frentes: uma avaliação quantitativa, focada nas métricas de sucesso e cobertura de código, uma análise comparativa aprofundada por meio de um estudo de caso e, por fim, uma análise qualitativa que investiga a relevância e as causas-raiz dos erros de geração. O objetivo é discutir as implicações destes resultados à luz dos objetivos da pesquisa, identificando os pontos fortes, as limitações e as oportunidades de melhoria na aplicação de LLMs em um contexto industrial.

4.1 Análise Quantitativa Geral

Nesta seção, são apresentados os resultados quantitativos da geração de testes. Foram executados experimentos com múltiplos modelos (da família Qwen e o DeepSeek Coder) nos cinco projetos-alvo, executados em diferentes ambientes. Os resultados estão consolidados em duas tabelas: a Tabela 4.1 apresenta as métricas de geração e a taxa de sucesso (testes que compilam e passam na execução inicial), enquanto a Tabela 4.2 detalha as métricas de cobertura de código obtidas.

A análise da Tabela 4.1 revela dois achados centrais. Primeiro, destaca-se o não-determinismo da geração de testes: o projeto 2, executado com o mesmo modelo qwen2.5-coder:14b em ambientes distintos, produziu resultados diferentes, com taxas de sucesso de 84,4% (ambiente 2) e 93,8% (ambiente 3). O mesmo fenômeno de variabilidade é observado no Projeto 4 (52,6% vs. 59,1%).

Segundo, a taxa de sucesso não foi uniformemente alta, mostrando-se altamente dependente da complexidade do projeto-alvo. Enquanto projetos como o Projeto 5 atingiram 100% de sucesso (tanto com o modelo 14B quanto com o 32B), o Projeto 3 representou um desafio significativo para o qwen2.5-coder:14b, que alcançou taxas de sucesso de apenas 20,8% (ambiente 2) e 26,0% (ambiente 3). Isso indica que, embora os LLMs possam gerar código sintaticamente correto, sua capacidade de produzir testes funcionais (que compilam e passam) em projetos complexos e com muitas dependências ainda é inconstante.

Tabela 4.1: *Resultados de Geração e Taxa de Sucesso por Modelo e Ambiente*

Projeto	LLM Utilizado	Máquina	Testes Gerados	Testes Passam	Taxa Sucesso (%)
Projeto 1	qwen2.5-coder:14b	Ambiente 3	49	38	77,6
Projeto 2	qwen2.5-coder:14b	Ambiente 2	263	222	84,4
Projeto 2	qwen2.5-coder:14b	Ambiente 3	226	212	93,8
Projeto 3	qwen2.5-coder:14b	Ambiente 2	461	96	20,8
Projeto 3	qwen2.5-coder:14b	Ambiente 3	504	131	26,0
Projeto 4	qwen2.5-coder:14b	Ambiente 2	428	225	52,6
Projeto 4	qwen2.5-coder:14b	Ambiente 3	530	313	59,1
Projeto 4	deepseek-coder:v2	Ambiente 3	481	277	57,6
Projeto 5	qwen2.5-coder:14b	Ambiente 3	38	38	100,0
Projeto 5	Qwen-Coder-32	Ambiente 1	14	14	100,0

Fonte: Elaborado pelo autor com base nos resultados experimentais (2025).

A Tabela 4.1 detalha todos os cenários experimentais executados. Para facilitar a visualização do impacto da complexidade do projeto na eficácia dos modelos, a Figura 4.1 consolida os resultados, apresentando a taxa de sucesso máxima alcançada para cada projeto. Observa-se claramente que, enquanto projetos de menor complexidade (como o Projeto 5) atingem o teto de desempenho, o aumento da complexidade estrutural (Projetos 3 e 4) impõe uma penalidade severa à capacidade de geração do modelo.

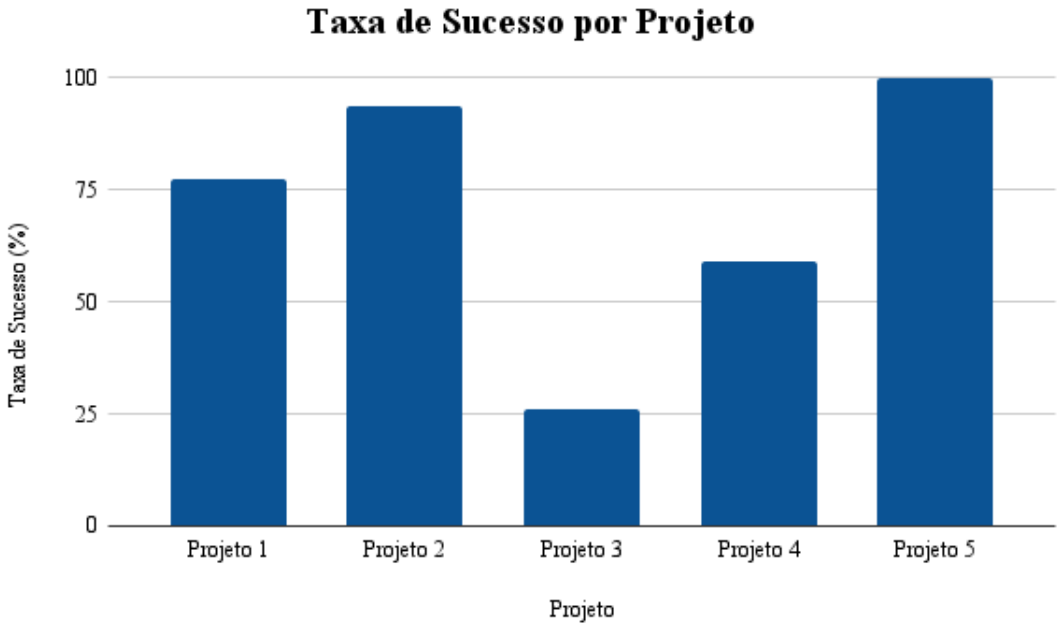


Figura 4.1: *Comparativo da melhor taxa de sucesso obtida por projeto. A tendência de queda evidencia a dificuldade dos LLMs em lidar com contextos de alta complexidade.*

Fonte: Elaborado pelo autor (2025).

Ao analisar as métricas de cobertura na Tabela 4.2, o cenário se mostra mais complexo e expõe uma limitação fundamental da tecnologia atual. Embora as coberturas de classe e linha tenham

atingido níveis moderados, a cobertura de *branch* (ramo) permanece como o principal desafio.

Tabela 4.2: *Resultados de Cobertura de Código por Modelo e Ambiente de Execução*

Projeto	LLM Utilizado	Máquina	Cob. Classe (%)	Cob. Método (%)	Cob. Linha (%)	Cob. Branch (%)
Projeto 1	qwen2.5-coder:14b	Ambiente 3	48	23	22	3
Projeto 2	qwen2.5-coder:14b	Ambiente 2	52	37	25	16
Projeto 2	qwen2.5-coder:14b	Ambiente 3	34	20	16	12
Projeto 3	qwen2.5-coder:14b	Ambiente 2	54	49	39	5
Projeto 3	qwen2.5-coder:14b	Ambiente 3	54	50	39	5
Projeto 4	qwen2.5-coder:14b	Ambiente 2	59	40	37	13
Projeto 4	qwen2.5-coder:14b	Ambiente 3	58	44	39	14
Projeto 4	deepseek-coder:v2	Ambiente 3	64	45	42	16
Projeto 5	qwen2.5-coder:14b	Ambiente 3	56	28	25	3
Projeto 5	Qwen-Coder-32	Ambiente 1	43	20	16	1

Fonte: Elaborado pelo autor com base nos resultados experimentais (2025).

Esta métrica se manteve consistentemente baixa em quase todos os projetos e modelos, atingindo um máximo de 16% (observado no Projeto 2 com qwen:14b e no Projeto 4 com deepseek-coder:v2). Esta baixa performance indica uma limitação importante dos modelos: embora consigam compreender e testar o fluxo principal de um método, eles demonstram dificuldade em inferir e gerar cenários de teste que explorem caminhos alternativos e casos de borda (por exemplo, diferentes blocos de *if/else* ou laços de repetição).

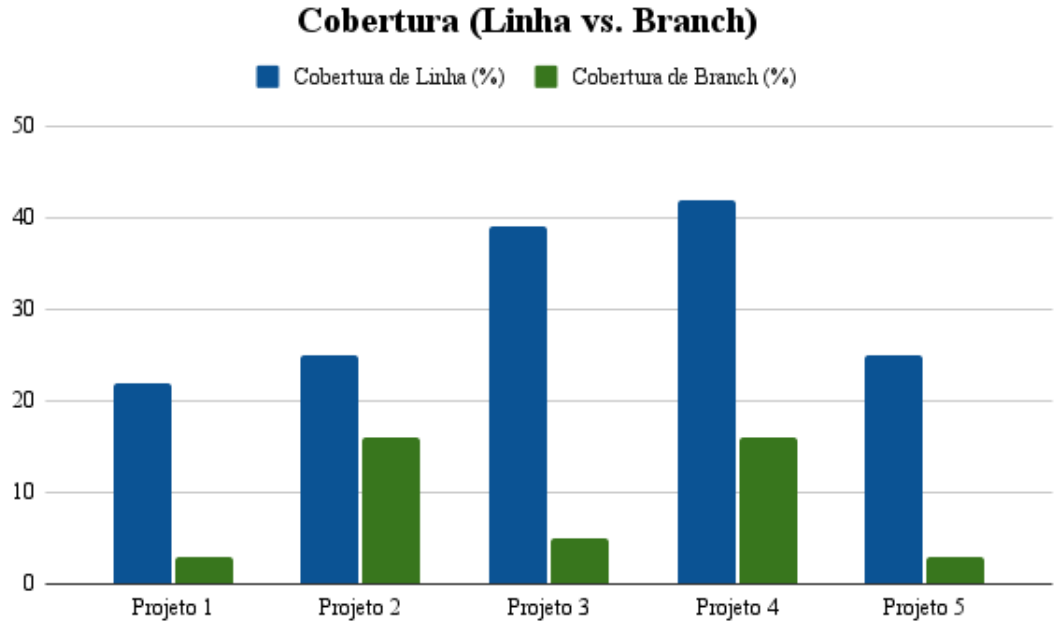


Figura 4.2: *Discrepância entre Cobertura de Linha e Cobertura de Branch. A estagnação da cobertura de branch (independente da complexidade do projeto) aponta para uma limitação estrutural dos modelos atuais em gerar testes para lógica condicional.*

Fonte: Elaborado pelo autor (2025).

A Figura 4.2 ilustra graficamente o contraste entre a cobertura de linha e a cobertura de branch (ramo) para os melhores casos de cada projeto. O gráfico torna evidente o 'abismo' de cobertura mencionado anteriormente: mesmo nos cenários onde o modelo consegue exercitar um volume razoável de linhas (barras azuis), ele falha sistematicamente em cobrir os desvios condicionais (barras verdes), o que inviabilizou a aplicação subsequente da análise de mutação.

Embora a análise de mutação, utilizando a ferramenta PITEST, tenha sido planejada como uma das métricas quantitativas, sua aplicação se mostrou inviável na prática. Conforme demonstrado na Tabela 4.2, a cobertura de *branch* alcançada pelos testes gerados foi consistentemente baixa. Uma vez que a análise de mutação depende da execução das linhas de código alteradas ("mutantes"), e muitos destes residem em desvios condicionais não cobertos, a aplicação da métrica resultaria em uma pontuação baixa e não representativa da qualidade das asserções. Portanto, concluiu-se que a baixa cobertura de *branch* é um gargalo mais fundamental que precede a análise de mutação, tornando-a um passo subsequente e dependente de uma cobertura mais robusta.

4.2 Análise Comparativa: Estudo de Caso do Projeto 5

Para aprofundar a análise, foi realizado um estudo de caso no projeto 5, comparando o desempenho do modelo `qwen2.5-coder` com 14 bilhões de parâmetros (14B) com uma versão maior, de 32 bilhões de parâmetros (32B). O objetivo foi investigar se um modelo com maior capacidade computacional se traduziria em testes de maior qualidade.

Os resultados, apresentados nas Tabelas 4.3 e 4.4, revelam uma dinâmica complexa e não linear. O modelo 32B demonstrou uma melhoria expressiva no `TransactionService`, praticamente dobrando a cobertura. No entanto, para o `CacheService`, seu desempenho foi consideravelmente inferior ao do modelo 14B. Curiosamente, a cobertura de branch no `TransactionService` foi maior com o modelo 14B (60%) do que com o 32B (40%). Essa variação evidencia um ponto crucial: a geração de testes por LLMs é intrinsecamente não-determinística. Um modelo maior não é inerentemente superior em todas as tarefas.

Tabela 4.3: *Análise Geral de Cobertura para o Pacote `service` (14B vs. 32B)*

Modelo	Pacote	Cob. de Método	Cob. de Linha	Cob. de Branch
qwen2.5-coder:14b	service	35%	21%	10%
qwen2.5-coder:32b	service	40%	23%	5%

Fonte: Elaborado pelo autor com base nos resultados experimentais (2025).

A análise qualitativa aprofunda essa percepção, revelando estratégias de teste distintas:

- **Complementaridade:** Observou-se que o modelo maior (32B) não englobou necessariamente todo o sucesso do modelo menor. Houve uma alternância de foco:
  - O modelo 32B conseguiu gerar testes para o método `getSubscriptionWithinPeriodForTenantAndService`, que havia sido completamente ignorado pela versão 14B.

Tabela 4.4: *Análise Comparativa de Cobertura por Classe de Serviço (14B vs. 32B)*

Modelo	Classe	Cob. de Método	Cob. de Linha	Cob. de Branch
qwen2.5-coder:14b	CacheService	40%	20%	5%
qwen2.5-coder:32b	CacheService	10%	3%	0%
qwen2.5-coder:14b	SubscriptionService	38%	20%	0%
qwen2.5-coder:32b	SubscriptionService	46%	21%	0%
qwen2.5-coder:14b	TransactionService	28%	26%	60%
qwen2.5-coder:32b	TransactionService	57%	56%	40%

Fonte: Elaborado pelo autor com base nos resultados experimentais (2025).

  - Por outro lado, o modelo 14B foi superior ao explorar os cenários e ramificações do método `findAllSubscriptionsWithinPeriod`, onde o modelo maior teve desempenho inferior.

  Isso demonstra que os modelos possuem "pontos cegos" distintos: onde um falha, o outro pode obter sucesso, sugerindo que o uso combinado de ambos traria um resultado superior ao uso isolado de qualquer um deles

- **Abordagens Técnicas:** O modelo 32B adotou uma abordagem mais sofisticada, utilizando reflexão para testar um método privado diretamente, o que é considerado uma melhor prática.
- **Pontos Cegos Comuns:** Métodos importantes e com regras de negócio complexas, como `createSubscription`, foram ignorados por ambos os modelos.

Em síntese, o aumento da capacidade do modelo (de 14B para 32B) resultou em uma troca de foco (*trade-off*) em vez de uma melhoria universal e linear. O modelo maior não apenas "somou" inteligência, mas sim deslocou sua atenção para outras partes do código, ganhando em alguns métodos complexos mas perdendo em outros que o modelo menor já cobria bem. Esse fenômeno reforça a natureza não determinística dos LLMs e sugere que, em um cenário industrial crítico, a estratégia de orquestrar múltiplos modelos pode ser mais segura do que confiar cegamente apenas no maior modelo disponível.

4.3 Análise da Qualidade e Padrões de Falha

Uma análise aprofundada dos testes que falharam é crucial para compreender as limitações práticas dos LLMs. A investigação revelou que os erros derivam de uma compreensão incompleta do contexto global do projeto. Os principais padrões de erro foram:

1. **Falta de Contexto sobre Construtores e Lógica Interna:** O LLM frequentemente gerava objetos para asserções sem compreender a lógica interna de seus construtores, levando a falhas em comparações `equals()`.

2. **Incompatibilidade de Tipos Genéricos:** Em testes com *mocks* de bibliotecas como *Redisson*, o LLM demonstrou dificuldade em lidar com tipos genéricos, gerando erros de compilação pois o *Mockito* exige correspondência exata.

3. **Alucinação de Métodos e Assinaturas Incorretas:** O modelo por vezes "inventava" métodos que não existiam na classe-alvo, preenchendo lacunas de conhecimento com padrões que não se aplicavam ao código.

4. **Geração Incorreta do Tipo de Teste (Unidade vs. Integração):** Foi observado o uso inadequado da anotação `@SpringBootTest`, que configura um teste de integração ao invés de um teste de unidade. Isso indica que a IA interpretou incorretamente o escopo solicitado.

5. **Erros Comuns de Codificação:** Problemas como omissão de *`imports`*, quebra de encapsulamento, configuração incompleta de *mocks* e uso de métodos de bibliotecas com versões incompatíveis foram recorrentes.

Essa análise aprofundada dos erros reforça a ideia de que, embora os LLMs sejam ferramentas poderosas, elas apresentam limitações similares às de desenvolvedores iniciantes, exigindo supervisão humana. Os testes gerados demandam um retrabalho significativo, o que representa uma das principais barreiras para sua adoção em larga escala.

Capítulo 5

Conclusões e Trabalhos Futuros

Este Trabalho de Conclusão de Curso teve como objetivo geral analisar a viabilidade e a eficácia da geração automática de testes de unidade utilizando Modelos de Linguagem de Grande Escala (LLMs) aplicada a projetos de software de larga escala. Através de uma metodologia experimental que combinou análise quantitativa e qualitativa, a pesquisa buscou identificar os benefícios práticos, desafios e limitações para a adoção desta tecnologia em um ambiente industrial.

5.1 Cumprimento dos Objetivos Específicos

A análise dos resultados permite afirmar que os objetivos propostos foram integralmente atingidos:

- **Investigar o estado da arte:** Atingido no Capítulo 2, onde foi realizado um levantamento bibliográfico abrangente, cobrindo desde os benchmarks fundamentais (HumanEval) até estudos recentes de 2025 (como Silva et al. e Pan et al.), estabelecendo as bases teóricas sobre engenharia de contexto e limitações dos modelos atuais.
- **Executar experimentos com múltiplos modelos e projetos:**Concretizado no Capítulo 3, através da configuração de um ambiente experimental que avaliou três LLMs (Qwen 14B, 32B e DeepSeek V2) em cinco projetos reais de diferentes complexidades, totalizando mais de 32.000 linhas de código analisadas.
- **Avaliar quantitativamente os testes gerados:** Realizado no Capítulo 4 (Seção 4.1), onde foram mensuradas as taxas de compilação, sucesso e cobertura de código. Identificou-se que a cobertura de linha atingiu níveis moderados, enquanto a cobertura de branch se revelou o principal gargalo técnico.
- **Analisar qualitativamente a utilidade e complexidade:** Alcançado na Seção 4.3, onde a inspeção manual revelou padrões de erro críticos, como alucinação de métodos e configurações incorretas de mocks, demonstrando que a utilidade dos testes depende de supervisão humana.

- **Identificar desafios para adoção industrial:** Consolidado ao longo da discussão dos resultados, onde se concluiu que a instabilidade dos testes e o custo do retrabalho ("alucinações") representam as barreiras primárias para a integração total em pipelines de CI/CD sem revisão humana.

5.2 Síntese das Conclusões

Com base no cumprimento dos objetivos, a pesquisa consolida os seguintes achados principais:

- **A eficácia é dependente da complexidade do projeto.** Os resultados quantitativos (Tabela 4.1) revelaram um cenário de eficácia mista. Enquanto projetos de menor complexidade atingiram taxas de sucesso de 100%, projetos com maior acoplamento e dependências, como o "Projeto 3", apresentaram taxas de sucesso notavelmente baixas (20,8%). Isso permite concluir que a aplicação de LLMs é altamente recomendada e segura para microsserviços utilitários, DTOs e componentes com baixo acoplamento. Para essas camadas do sistema, a ferramenta atua como um acelerador robusto, validando a hipótese de que a IA pode assumir a carga de testes triviais com alta eficácia.

- **O principal gargalo é a baixa cobertura de *branch* (ramo).** Esta foi a descoberta quantitativa mais significativa da pesquisa. Conforme demonstrado na Tabela 4.2, a cobertura de *branch* manteve-se consistentemente baixa em todos os projetos, atingindo um máximo de 16%. Este dado é crucial, pois evidencia que os LLMs, em seu estado atual, demonstram grande dificuldade em inferir e gerar cenários de teste que explorem caminhos alternativos, casos de borda e diferentes blocos condicionais (como `if/else`).

- **LLMs atuam como programadores iniciantes que exigem supervisão.** A análise qualitativa dos erros (Seção 4.3) corroborou a baixa compreensão do contexto global do projeto. Os erros mais comuns incluíram a "alucinação"de métodos inexistentes, incompatibilidade de tipos genéricos em *mocks*, e a geração incorreta de testes de integração (`@SpringBootTest`) em vez de unidade. Isso reforça a conclusão de que os testes gerados exigem um **retrabalho significativo**, o que representa hoje a maior barreira para a adoção em larga escala.

- **Modelos maiores não são universalmente melhores; são complementares.** O estudo de caso (Seção 4.2) comparando o modelo Qwen de 14B e 32B revelou que o aumento da capacidade do modelo resultou em uma “troca de foco” em vez de uma melhoria linear. O modelo maior foi capaz de testar métodos complexos ignorados pelo menor, mas, ao mesmo tempo, ignorou cenários que o modelo menor havia coberto bem. Isso sugere que os modelos possuem “pontos cegos” distintos e que uma estratégia de orquestração de múltiplos modelos pode ser mais robusta do que a confiança em um único modelo, mesmo que maior.

5.3 Ameaças à Validade

Conforme diretrizes de engenharia de software experimental, discutem-se a seguir as ameaças à validade que podem influenciar a interpretação dos resultados deste estudo, categorizadas em validade interna, externa e de construto.

5.3.1 Validade Interna

A validade interna diz respeito a fatores incontroláveis que podem ter afetado os resultados experimentais. Uma ameaça primária é o não-determinismo intrínseco dos LLMs. Mesmo com parâmetros de temperatura controlados na ferramenta ChatUniTest, modelos generativos podem produzir saídas distintas para o mesmo prompt em execuções diferentes. Para mitigar essa ameaça, os experimentos foram conduzidos múltiplas vezes e, na análise qualitativa, buscou-se identificar padrões de erro recorrentes (como alucinações) em vez de focar em falhas isoladas. Outra ameaça reside na dependência da ferramenta de orquestração (ChatUniTest). Falhas na extração do contexto focal ou na análise estática prévia realizada pela ferramenta podem ter induzido os modelos ao erro, independentemente da capacidade de raciocínio da IA. Buscou-se atenuar esse risco garantindo que todas as dependências dos projetos estivessem corretamente configuradas no Maven antes das execuções.

5.3.2 Validade Externa

A validade externa refere-se à capacidade de generalização dos resultados. O estudo limitou-se à linguagem Java e ao framework Spring Boot. Conforme apontado na literatura (PAN et al., 2024), Java apresenta desafios específicos de verbosidade e tipagem que não necessariamente se aplicam a linguagens como Python ou JavaScript. Portanto, as taxas de sucesso e os tipos de erro reportados aqui não devem ser generalizados automaticamente para outros ecossistemas tecnológicos. Adicionalmente, os objetos de estudo são projetos proprietários de uma única empresa. Embora tenham sido selecionados para representar domínios variados, eles podem compartilhar estilos arquiteturais e convenções de código específicos da organização, o que pode ter influenciado positiva ou negativamente o desempenho dos modelos.

5.3.3 Validade de Construto

A validade de construto avalia se as métricas utilizadas refletem adequadamente o fenômeno estudado. A principal métrica de qualidade utilizada foi a cobertura de código (linha e branch). É consenso na literatura que alta cobertura não implica necessariamente em alta eficácia na detecção de falhas, pois é possível cobrir linhas com asserções triviais. Para mitigar essa ameaça, o desenho experimental previa o uso de Análise de Mutação (Mutation Testing) como uma métrica de qualidade superior ("padrão-ouro"). Contudo, conforme discutido nos resultados, a baixa cobertura

de branch alcançada pelos modelos inviabilizou a aplicação confiável dessa técnica, restando à análise qualitativa manual (Seção 4.3) o papel de validar a utilidade real dos testes gerados.

5.4 Trabalhos Futuros

Com base nas conclusões e limitações identificadas, são propostos os seguintes trabalhos futuros:

- **Engenharia de Prompt para Cobertura de *Branch***: Investigar técnicas de *prompt engineering* focadas em instruir explicitamente os LLMs a identificar e gerar testes para todos os caminhos lógicos, desvios condicionais e casos de borda do método-alvo.
- **Melhoria de Contexto (*Context-Awareness*):** Avaliar o impacto de fornecer um contexto de projeto mais amplo aos LLMs, como a análise de dependências inter-classes ou o uso de técnicas de *Retrieval-Augmented Generation* (RAG) para alimentar o modelo com informações de todo o repositório.
- **Orquestração de Múltiplos Modelos:** Desenvolver uma ferramenta que execute a geração de testes com múltiplos LLMs em paralelo e mescle os testes bem-sucedidos, explorando a natureza complementar e os "pontos cegos" distintos observados nesta pesquisa.
- **Análise de Custo-Benefício (Retrabalho):** Realizar um estudo empírico focado em medir o tempo gasto por desenvolvedores para corrigir os testes gerados pelos LLMs versus o tempo de criação manual, calculando o real custo-benefício da adoção da ferramenta.

Referências Bibliográficas

ABDULLIN, A.; DERAKHSHANFAR, P.; PANICHELLA, A. *Test Wars: A Comparative Study of SBST, Symbolic Execution, and LLM-Based Approaches to Unit Test Generation*. 2025. Disponível em: https://arxiv.org/abs/2501.10200. Acesso em 15 de novembro de 2025. 5, 6, 11

AI, M. Automated unit test improvement using large language models at meta. 2024. Disponível em: https://arxiv.org/abs/2402.09171. Acesso em 15 de novembro de 2025. 7, 11

ALVES, V. A.; BEZERRA, C.; MACHADO, I. An empirical study on test smells in github copilot generated tests. 2024. Disponível em: https://sol.sbc.org.br/index.php/cbsoft_estendido/article/view/30247. Acesso em 25 de novembro de 2025. 10

ANAND, S. et al. An orchestrated survey of methodologies for automated software test case generation. 2013. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121213000563. Acesso em 15 de novembro de 2025. 1

CHEN, M. et al. Evaluating large language models trained on code. 2021. Disponível em: https://arxiv.org/abs/2107.03374. Acesso em 25 de novembro de 2025. 6

FRASER, G.; ARCURI, A. A large-scale evaluation of automated unit test generation using evosuite. 2014. Disponível em: https://dl.acm.org/doi/10.1145/2685612. Acesso em 25 de novembro de 2025. 5

GU, S.; NASHID, N.; MESBAH, A. Llm test generation via iterative hybrid program analysis. 2025. Disponível em: https://arxiv.org/html/2503.13580v2. Acesso em 25 de novembro de 2025. 8

GU, S.; ZHANG, Q.; FANG, C.; OTHERS. Testart: Improving llm-based unit testing via co-evolution of automated generation and repair iteration. *arXiv preprint arXiv:2408.03095*, 2024. Disponível em: https://arxiv.org/abs/2408.03095. Acesso em 15 de novembro de 2025. 4, 9, 11

HONG, H.; BAIK, J. Retrieval-augmented generation for code: A survey. 2025. Disponível em: https://arxiv.org/abs/2506.11591. Acesso em 25 de novembro de 2025. 7

International Organization for Standardization. *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models*. 2011. 4

JUST, R. et al. Are mutants a valid substitute for real faults in software testing? 2014. 6

KOCHHAR, P. S.; LO, D.; LAWALL, J.; NAGAPPAN, N. Code coverage and postrelease defects: A large-scale study on open source projects. *IEEE Transactions on Reliability*, v. 66, n. 4, p. 1213–1228, 2017. 4, 5

KONSTANTINOU, M.; DEGIOVANNI, R.; PAPADAKIS, M. Do llms generate test oracles that capture actual or expected behavior? 2024. Disponível em: https://arxiv.org/abs/2410.21136. Acesso em 25 de novembro de 2025. 7

LEMIEUX, C.; INALA, J. P.; LAHIRI, S. K.; SEN, S. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. 2023. Disponível em: https://doi.org/10.1109/ICSE48619.2023.00085. Acesso em 25 de novembro de 2025. 9

OSMANI, A. Context engineering guide for llm-based software development. 2024. Disponível em: https://www.promptingguide.ai/guides/context-engineering-guide. Acesso em 25 de novembro de 2025. 8

PAN, R.; KIM, M.; KRISHNA, R.; PAVULURI, R.; SINHA, S. Aster: Natural and multi-language unit test generation with llms. 2025. Disponível em: https://arxiv.org/abs/2409.03093. Acesso em 25 de novembro de 2025. 6

PAPADAKIS, M. et al. Chapter six - mutation testing advances: An analysis and survey. Elsevier, 2019. Disponível em: https://www.sciencedirect.com/science/article/pii/S0065245818300305. Acesso em 25 de novembro de 2025. 1

PENG, S.; KALLIAMVAKOU, E.; CIHON, P.; DEMIRER, M. *The Impact of AI on Developer Productivity: Evidence from GitHub Copilot*. 2023. Disponível em: https://arxiv.org/abs/2302.06590. Acesso em 15 de novembro de 2025. 2

PRESSMAN, R. S. *Engenharia de Software: Uma Abordagem Profissional*. 7. ed. : AMGH Editora, 2010. 1

SCHÄFER, M.; NADI, S.; EGHBALI, A.; TIP, F. *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*. 2023. Disponível em: https://arxiv.org/abs/2302.06527. Acesso em 15 de novembro de 2025. 4, 7, 11

SIDDIQ, M. L. et al. Using large language models to generate junit tests: An empirical study. 2024. Disponível em: https://arxiv.org/abs/2305.00418. Acesso em 25 de novembro de 2025. 7

SILVA E. C. O., C. R. S. . S. L. F. Llms as test generators: A comparative benchmarking study. 2025. Disponível em: https://doi.org/10.5753/sbes.2025.9618. Acesso em 25 de novembro de 2025. 6

SOMMERVILLE, I. *Engenharia de Software*. 10. ed. : Pearson, 2019. 4

WANG, G.; XU, Q.; BRIAND, L. C.; LIU, K. On mutation-guided unit test generation. 2024. Disponível em: https://arxiv.org/html/2506.02954v2. Acesso em 25 de novembro de 2025. 9

ZHANG, Q. et al. *A Survey on Large Language Models for Software Engineering*. 2024. Disponível em: https://arxiv.org/abs/2312.15223. Acesso em 15 de novembro de 2025. 2

ZJU-ACES-ISE. *ChatUniTest: Automated Unit Test Generation with LLMs*. 2023. Disponível em: https://github.com/ZJU-ACES-ISE/ChatUniTest. Acesso em 15 de novembro de 2025. 8

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DA PARAÍBA
Campus Campina Grande - Código INEP: 25137409
R. Tranqüilino Coelho Lemos, 671, Dinamérica, CEP 58432-300, Campina Grande (PB)
CNPJ: 10.783.898/0003-37 - Telefone: (83) 2102.6200

Documento Digitalizado Ostensivo (Público)

Versão Final do TCC

Assunto:	Versão Final do TCC
Assinado por:	Maria Melo
Tipo do Documento:	Dissertação
Situação:	Finalizado
Nível de Acesso:	Ostensivo (Público)
Tipo do Conferência:	Cópia Simples

Documento assinado eletronicamente por:

- **Maria Eduarda Pereira de Souza Melo**, **DISCENTE (202111250025) DE BACHARELADO EM ENGENHARIA DE COMPUTAÇÃO - CAMPINA GRANDE**, em 18/02/2026 11:39:20.

Este documento foi armazenado no SUAP em 18/02/2026. Para comprovar sua integridade, faça a leitura do QRCode ao lado ou acesse https://suap.ifpb.edu.br/verificar-documento-externo/ e forneça os dados abaixo:

Código Verificador: 1769140
Código de Autenticação: d68c455fc6