Análise da eficiência de *LLMs* na geração de Testes Automatizados

Lucas Silva Viana
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais)
Belo Horizonte, Brasil
lucasviana13@ufmg.br

André Cavalcante Hora (Orientador)
Departamento de Ciência da Computação
Universidade Federal de Minas Gerais)
Belo Horizonte, Brasil
andrehora@dcc.ufmg.br

***Abstract*—A escrita de testes unitários é uma tarefa primordial para garantia da qualidade do software, contudo, a escrita dos mesmos causa um aumento o tempo de desenvolvimento de um sistema. Com o avanço dos Grandes Modelos de Linguagem (LLMs), aparece uma chance de automatização da escrita de testes. Entretanto, ainda é necessário avaliar a qualidade dos testes gerados se de fato aumentam, a cobertura e se não apresentam *smells*. Desse modo, este trabalho propõe um estudo comparativo para avaliação da capacidade dos LLMs na geração de testes em Python. A metodologia utiliza o *Deepseek* v4 como modelo de LLM, Pytest para execução e metrificação dos testes, e a ferramenta ArodaDr para detecção dos *tests smells*. Desse modo, esse trabalho traz todo o pipeline de geração de testes com uso de LLMs e a análise dos testes gerados, buscando trazer *insights* para avanço na visão da capacidade dos LLMs para geração de código de testes.**

***Index Terms*—LLM, Testes Unitários, *Test Smells*, Automação, Qualidade de Software.**

I. Introdução

Atualmente, no mercado de softwares, um dos principais parâmetros de qualidade de códigos está em seu *suíte de testes*, um requisito passado aos próprios desenvolvedores do sistema. A escrita de bons testes unitários e uma boa cobertura são cruciais na validação do comportamento dos componentes individuais do sistema. Softwares com bons testes unitários facilitam refatorações e auxiliam bastante no entendimento do código, quase como uma documentação adicional.

A. Caracterização do Problema

Apesar da sua importância, a escrita manual dos testes unitários consome um grande tempo da esteira de desenvolvimento de um projeto. Além disso, pondera-se a ideia do viés de confirmação, no qual, a escrita dos testes por humanos tende a não cobrir todos os cenários críticos do código nos testes, por conta de um "apego emocional", podendo passar despercebidas falhas no software desenvolvido, usado por [1] como forte argumento para uso de LLMs para geração de testes.

Além disso, conforme as empresas crescem, costumam ter sistemas mais antigos (*legados*) que são o *core* do sistema, com códigos difíceis de manutenção. O que se torna uma enorme dívida técnica que, no mercado *ágil* atual, não costuma ser priorizada pelo baixo valor agregado ao negócio.

B. Motivação

Recentemente, o desenvolvimento de software como um todo tem sido impactado pelo avanço dos Grandes Modelos de Linguagem (LLMs), como Claude (Antropic) e Codex (Openai). Estes modelos demonstram capacidade de geração de código eficiente, levantando-se a hipótese que poderiam ser utilizados para automatização da escrita de testes unitários, reduzindo o esforço dessa tarefa para os desenvolvedores focarem em gerar mais valor ao negócio em código de produção ao invés de código de teste.

Entretanto, a geração de código automática precisa ser avaliada, prinicipalmente no aumento da cobertura, na qualidade dos testes, pois desenvolvedores ainda precisarão ter contato com esse código, além dos testes serem uma métrica crucial para a garantia do software como um todo. Estes testes gerados podem ser frágeis e apresentar *smells*.

Desse modo, a motivação do trabalho reside na necessidade de validar o uso de LLMs para a geração de testes unitários. Não basta escrever os testes, eles precisam ser precisos e de alta qualidade em diferentes contextos e complexidades dos softwares.

C. Objetivos

O principal objetivo deste trabalho é investigar a eficácia e viabilidade da aplicação de Grandes Modelos de Linguagem (*LLMs*), no desenvolvimento de sistemas como ferramenta de auxílio na criação de testes unitários. Com expectativa de explorar os potenciais dos modelos para melhoria no fluxo de desenvolvimento de *softwares*, reduzindo o tempo gasto no desenvolvimento de testes unitários, garantindo entregas mais rápidas, sem abrir mão da qualidade e confiabilidade do produto.

Como objetivo secundário, busca-se o desenvolvimento de um pipeline inteligente para Geração de Testes Unitários para análise e escrita de testes desde arquivos específicos até sistemas completos.

II. Referencial Teórico

A. Metodologias Ágeis

Os primeiros processos de desenvolvimento de *softwares* eram estritamente sequenciais, através de fases de especificação de requisitos até às fases finais de

implementação, testes e manutenção do código, o que tornava o processo de entrega de sistemas lento e burocrático. Com isso, em 2001, um grupo de profissionais se uniram para propor uma alternativa ao *Waterfall*, em que, surgiram as bases para um conceito de desenvolvimento de *software*, sendo que suas raízes são usadas até hoje em dia, o qual foi documentado no Manifesto *ágil*. [2, p.33, 34]

> Através deste trabalho, passamos a valorizar:
> Indivíduos e interações mais que processos e ferramentas
> *Software* em funcionamento mais que documentação abrangente
> Colaboração com o cliente mais que negociação de contratos
> Responder a mudanças mais que seguir um plano
> Ou seja, mesmo havendo valor nos itens à direita, valorizamos mais os itens à esquerda. [3]

Com isso, passam-se a adotar ciclos curtos de desenvolvimento, com sistemas sendo implementados de forma gradativa com pequenas entregas priorizadas pela urgência do cliente, em que o desenvolvimento acaba quando todos os requisitos do sistema estejam entregues. Esse modelo foi usado como base para várias metodologias ágeis amplamente usadas no mercado como o *XP (Extreme Programming)*, *Scrum*, *ShapeUp* mais recentemente, entre outros. [2, p.34, 37]

B. Testes Unitários

A afirmação "Teste é uma das práticas de programação mais valorizadas hoje em dia, em qualquer tipo de *software*." [2, p.258], deixa claro, que na visão de mercado, o desenvolvimento de *softwares* de qualidade tem como requisito os testes. Com a ascensão do desenvolvimento *ágil*, a ideia do *software* auto-testável, ou seja, um sistema com testes automatizados que garantem que o funcionamento está conforme o esperado.

Desse modo, uma forma de se classificar os testes automatizados é segundo a *Pirâmide de Testes* [4]:

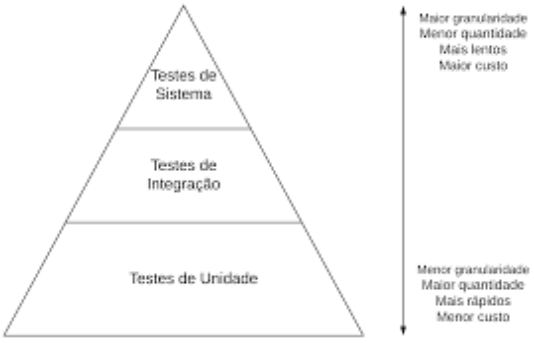

Fig. 1. *Pirâmide de Testes*.

Nessa definição, os testes unitários, são testes que verificam pequenas partes do código, feitos isoladamente em cada módulo para verificar o comportamento. Geralmente estabelece um ambiente artificial para o cenário testado e invoca a função testada e verifica os resultados retornados por ela, segundo o esperado. [5]

Para exemplificar o conceito, imagine um sistema completo, em que temos diversos módulos que são chamados entre si; os testes unitários buscam isolar esses módulos e testar cada um individualmente.

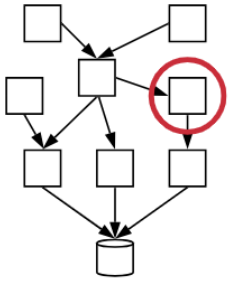

Fig. 2. Escopo Teste Unitário.

Com a certeza de que as partes individuais do código funcionam, para casos com sistemas com alta cobertura de testes de qualidade, temos uma verificação de integridade do sistema como um todo.

C. Tests Smells

> Os códigos de teste são tão importantes quanto o código de produção. Ele não é componente secundário. Ele requer raciocínio, planejamento e cuidado. É preciso mantê-lo tão limpo quanto o código de produção. [6]

Martin com essa afirmação, exalta a importância de escrever testes limpos como parte do ciclo de desenvolvimento de código sustentável.

Desse modo, devemos evitar más práticas, em que, os *test smells* descrevem características que devem ser evitadas para a escrita de bons testes unitários, pensando principalmente que os códigos de teste, assim como os códigos de produção, devem evoluir e serem refatorados.

Seguindo o raciocínio trazido por Marco Tulio [2], os *test smells* não devem ser pensados como situações a serem evitadas a qualquer custo, mas sim como alertas, em que, cabe ao desenvolvedor refletir sob o cenário e avaliar o impacto ao código e sua evolução.

1) Test Smells Utilizados:

- **Assertion Roulette:** Método de teste com mais de um assert sem uma explicação/mensagem.
- **Conditional Test Logic:** Método de teste tem estruturas de controle de fluxo como laços e condicionais.
- **Duplicate Assert:** Teste tem pelo menos um assert repetido no mesmo método.
- **Empty Test:** Método de teste sem código implementado.
- **Exception Handling:** Teste com gerenciamento de exceções como blocos try-catch.
- **Ignored Test:** Quando há um teste marcado para ser pulado na execução dos testes.

- **Magic Number:** Existência de asserts no teste com valores numéricos não explicados.
- **Redundant Print:** Testes que possui métodos de impressão desnecessários.
- **Sleepy Test:** Existência de comandos de pausa (sleep), introduzindo atrasos durante a execução do teste.
- **Unknown Test:** Testes que não possuem asserts.

D. Testabilidade de Software

> Softwares altamente testáveis também tendem a ser bem projetados. [7]

A afirmação anterior, expressa bem que um *software* com uma arquitetura robusta tende a ser mais fácil de testar. É nesse caminho que se define a testabilidade de um sistema. Ela indica que o desenho do código deve ser realizado pensando em um formato que seja fácil de testar.

> Um código difícil de testar geralmente é:
>
> - Fortemente acoplado: "Não posso testar isso sem instanciar metade do sistema."
> - Fracamente coeso: "Esta aula faz tanto que o teste será enorme e complexo!"
> - Redundante: "Terei que testar isso em vários lugares para garantir que funcione em todos os lugares.
>
> [7]

Com isso, ter *softwares* testáveis, indica que se tem um sistema bem projetado e que provavelmente terá menor custo de manutenção além de uma melhor garantia de funcionamento, pois poderão ser testados de maneira unitária.

E. Aplicação monolítica

> Arquitetura monolítica é um modelo tradicional de desenvolvimento de software no qual uma única base de código executa múltiplas funções de negócios. [8]

Monolitos são até hoje bem comuns em grandes empresas, mesmo que não seja muito um padrão convencional da atualidade, sistemas legados costumam carregar esse tipo de modelagem de múltiplas funções em uma única aplicação, tendo assim um código mais complexo e instável.

Componentes:

- **Interface do Usuário (UI):** Seria a ideia da interface visual que o usuário acessa, o front-end.
- **Banco de dados:** Se trata do banco de dados com as tabelas necessárias para os fluxos de negócio.
- **Aplicação:** Implementa as regras de negócio do sistema.

F. Complexidade Ciclomática

Idealizada por McCabe [9], a complexidade ciclomática é uma métrica quantitativa da complexidade de código que mede o número de caminhos de execução independentes através de um método. Ela faz isso a partir da contagem de pontos de decisão como condicionais, loops, captura de exceções, entre outros.

Matematicamente, a forma de se calcular essa métrica é:

$$C = E - N + 2P$$

No qual, P indica o número de componentes conectados, E o número de caminhos possíveis e N o número de instruções.

Um alto acoplamento muitas vezes leva a uma alta complexidade, pois as modificações em um módulo podem exigir testes e validações complexos em outros. A baixa coesão frequentemente aumenta a complexidade ciclomática, pois um único módulo tenta gerenciar muitas lógicas de decisão diferentes.

Quanto maior o valor, indica que temos um código mais difícil de entendimento. Relacionando com a coesão, naturalmente, um código com um valor de complexidade ciclomática alta terá muitas lógicas de decisão em uma mesma função, o que reduz a coesão.

Para os testes unitários, será mais difícil de testar códigos com alta complexidade ciclomática, uma vez que, teremos muitos cenários de teste para cobrir (branch coverage).

Ferramentas de análise estática de código como *Sonarqube* é capaz de metrificar esse valor.

G. Complexidade Cognitiva

A Complexidade Cognitiva é uma métrica disponível no *Sonarqube*, idealizado por Campbell [10], com o propósito de mensurar o quão difícil um código é para o entendimento de um ser humano, mensurando o esforço mental necessário para compreensão de um método, classe ou aplicação como um todo.

1) Metodologia: Para o cálculo da pontuação de complexidade, a metodologia se baseia em três critérios principais:

1) **Ignorar Estruturas de Atalho:** Estruturas que são criadas para melhoria de legibilidade são ignoradas para o incentivo de boas práticas como as faladas anteriormente. Como por exemplo, a divisão do código em métodos permite condensar várias instruções em uma única chamada com nome sugestivo.
2) **Incrementar para Quebras no Fluxo Linear:** Um incremento é aplicado para cada estrutura que interrompe o fluxo sequencial do código. O fluxo sequencial é de cima para baixo e da esquerda para direita. Um loop interrompe o fluxo sequencial pois, ao final de cada iteração, a execução retorna para o início do loop para reavaliar a condição.

```
void myMethod () {
  try {
    if (condition1) {                    // +1
      for (int i = 0; i < 10; i++) {     // +2 (nesting=1)
        while (condition2) { … }         // +3 (nesting=2)
      }
    }
  } catch (ExcepType1 | ExcepType2 e) { // +1
    if (condition2) { … }               // +2 (nesting=1)
  }
}                                       // Cognitive Complexity 9
```

Fig. 3. Exemplo de Contabilização [10]

H. Cobertura de Linhas e Branches

Para avaliar a quantidade de código de produção que está coberto pelos testes unitários, serão utilizadas as métricas de

cobertura que são um padrão dos desenvolvedores. As métricas de coberturas serão a de linha e a de branches, no qual, para o estudo, repositórios com coberturas acima de 50% serão desconsiderados por entender que o aumento com os testes gerados com os LLMs serão de menor impacto. Ambas as métricas serão explicadas a seguir [11]

1) Cobertura de Linhas: Métrica que avalia o percentual de linhas de código executáveis do código de produção que foram executadas durante a execução do suíte de testes.

É uma métrica fundamental que serve como indicador para validação dos testes. Essa métrica será uilizada para comparação do aumento do código testado por meio das LLMs.

2) Cobertura de Branches: Métrica que avalia a cobertura de caminhos de código, medindo o percentual de pontos de decisão no código de produção que foram executados nos testes.

Essa cobertura é crucial para análise do quanto os cenários lógicos da aplicação estão sendo validados por testes, trazendo assim uma visão da robustez dos testes já existentes e dos testes gerados por meio dos LLMs.

I. Grandes Modelos de Linguagem (LLMs)

> LLM (*Large Language Models*) é um modelo de Machine Learning usado na inteligência artificial generativa, capaz de interpretar diferentes funções. Os LLM são treinados com imensas quantidades de dados para poderem reconhecer e gerar imagens, textos, conversações e outros tipos de conteúdos. [12]

De uma forma mais direta, os Grandes Modelos de Linguagem são modelos treinados com uma quantidade massiva de dados para conseguirem executar tarefas através de prompts recebidos. São capazes de diversas coisas, mas o foco nesse trabalho é a geração de código que vem se popularizando atualmente.

As *LLMs* são vistas como parte presente do dia a dia tanto no trabalho, estudo e vida pessoal como forma de aumento da proutividade através do auxílio no dia a dia.

No contexto da Engenharia de Software, o treinamento dos modelos foi expandido englobando repositórios para o entendimento do LLM a estruturar códigos, gerando os Code-LLMs como Codex e Claude. Esses novos modelos capacitaram à IA o reconhecimento e interpretabilidade de lógicas e sintaxes de diversas linguagens de programação, sendo capazes hoje de refatoração, autocompletar e até mesmo a escrita de código de forma independente.

Entretanto, essa escrita independente em modelos convencionais ainda é limitada por conta de sua janela de contexto ou mecanismo de atenção, que impede a leitura de repositórios inteiros, que ocasiona a ideia do *Lost in the Middle*, no qual a performance degrada em longos contextos, como explicado abaixo:

> Observamos que o desempenho pode sofrer uma degradação significativa ao alterar a posição de informações relevantes, o que indica que os modelos de linguagem atuais não utilizam de forma robusta as informações contidas em contextos de entrada extensos. [13][tradução do autor]

J. Prompt Engineering

A garantia de respostas eficientes a um custo menor possível de tokens para execução da tarefa é uma métrica crucial para se definir um agente eficiente. Para isso, a engenharia de prompt é um dos elementos centrais para a garantia de um bom agente. No qual, ela consiste na formulação de instruções para execução de tarefas específicas ao modelo, como a geração de testes unitários.

> A engenharia de prompts cria prompts eficazes que orientam LLMs em tarefas específicas. [14][tradução do autor]

Para isso, como ponto de partida para prompts mais eficientes, alguns elementos importantes segundo [14] são:

- Descrição em Linguagem Natural: Criação de diretivas de papel (Ex: “Aja como um Engenheiro de Software”), imposição de passos para controle e descrição clara da tarefa aumenta a eficiência do modelo e reduz a chance de alucinação;
- Uso de Delimitadores Claros: Encapsular artefatos como código dentro de marcadores como *#CODE_START# e #CODE_END#* reduz a confusão do modelo frente a mistura de textos e trechos de código.

Partindo agora para técnicas mais amputadas, o trabalho faz uso de algumas estratégias de prompt, como forma de reduzir o consumo de tokens e aumentar a eficiência do modelo na tarefa, abaixo será detalhado tais estratégias:

1) Zero-Shot Learning (ZSL) vs Few-Shot Learning (FSL):

> O aprendizado *Zero-Shot* permite que modelos generalizem sem exemplos, mas enfrenta dificuldades com a complexidade (Sahoo et al., 2024). O aprendizado *Few-Shot* utiliza exemplos para melhorar o desempenho em tarefas inéditas (Brown et al., 2020). [14][tradução do autor]

No trabalho, houve uma mescla das duas estratégias para contextos específicos, que será detalhado na continuidade do mesmo.

2) Chain-of-Thought (CoT): A ideia da técnica é induzir o modelo a executar um raciocínio passo a passo, forçando o modelo a executar passos intermediários de raciocínio antes da resposta final. Essa decomposição em passos lógicos minimiza as alucinações do LLM, tornando-o mais confiável [15].

> Dividir a resolução de problemas em etapas é uma abordagem mais natural e semelhante à humana, permitindo que o LLM compreenda a funcionalidade, o comportamento e os cenários de uso de cada parte. [16][tradução do autor]

III. Metodologia

A. Escopo da Linguagem de Programação

O Python foi selecionado para a pesquisa por ser uma linguagem de script, com tipagem dinâmica e múltiplos

paradigmas possíveis. Além disso, a alta popularidade da linguagem faz com que se tenham vários repositórios com cenários distintos desde campos como Ciência de Dados, Desenvolvimento Web, Microserviços, entre outros. Oferecendo um contraponto interessante ao Java, permitindo análises mais ricas sobre a adaptabilidade dos LLMs a diferentes ecossistemas.

1) Pontos Positivos:

- **Ecossistema Simplificado:** É um ambiente bem leve, sendo bem viável operacionalmente, sem necessidade de compilação. Além disso, as dependências centralizadas em torno do pip traz uma possibilidade de automação mais direta;
- **Variedade de Repositórios:** Por ser uma linguagem bem popular, tem um vasto ecossistema de projetos open-source no Github. O que traz vantagens para montagem de um bom dataset para análise;
- **Linguagem Flexível:** Por ter uma natureza dinamicamente tipada, traz menos rigidez e isso exige que o LLM valide o contexto de uso das funções e variáveis, além de poder validar as vantagens ou desvantagens da liberdade de um código mais flexível para a geração automatizada.

2) Desafios:

- **Gerenciamento de Dependências:** Por não ter um padrão obrigatório para especificação de dependências, podendo ser especificado por um arquivo "requirements.txt" ou com o uso de ferramentas como Poetry gerando um "pyproject.toml", a gestão de dependências de forma eficiente será um desafio. Além disso, nem sempre terá as versões especificadas, o que pode gerar a instalação de pacotes atualizados e incompatíveis com o projeto;
- **Estrutura de Projetos:** Por ser uma linguagem mais flexível, não há uma estrutura de projetos mais restrita, o que pode levar a dificuldades de identificação de estruturas.

B. Ferramentas de Análise Estática

1) Testabilidade: Para coleta das métricas de testabilidade definidas, será utilizada a plataforma *Sonarqube* [17], que tem como propósito a análise estática de código. É bem famosa sendo amplamente adotada em empresas e no ramo acadêmico, deixando claro sua validade e relevância.

Com as métricas definidas de Complexidade Ciclomática e Cognitiva, o Sonar tem o domínio completo de ambas, inclusive a Complexidade Cognitiva foi idealizado na empresa SonarSource, com isso, é a ferramenta mais preparada para análise desse parâmetro e a única em mercado hoje com esse foco.

Por fim, a escolha foi motivada por conta de seu suporte robusto para todas as linguagens de programação cobertas na pesquisa, além disso, as métricas definidas estão presentes no Sonar e outro ponto crucial é sua facilidade para integração na automação por meio de seu uso via API Rest.

2) Cobertura de Linhas e Branches dos Testes: Para coleta das métricas de cobertura será feita via uso do Pytest, biblioteca já consolidada para testes em Python.

3) Test Smells: Para análises de possíveis *test smells*, o trabalho fará uso da ferramenta *AromaDr* [18]. Escolha motivada principalmente por ser um trabalho bem documentado que cobre às linguagens que serão utilizadas no dataset gerado, no caso Java e Python. Essa decisão é fundamental para consistência e validade da análise unificando as análises com apenas um detector de antipadrões nos testes unitários.

Processo da ferramenta:

1) **Geração da LAAST:** Ferramenta utiliza o rust-code-analysis para conversão do código fonte em uma árvore de sintaxe abstrada independente da linguagem (LAAST);
2) **Extração de Dados:** A partir da LAAST, a ferramenta extrai informações relevantes do teste (nomes, asserts, chamadas, entre outros) e padroniza em um modelo de dados;
3) **Detecção de *Smells*:** Com essa versão abstrata das particularidades sintáticas da linguagem, a mesma lógica de detecção pode ser aplicada de forma independente da linguagem.

C. Modelo de LLM

Para a execução do pipeline de geração dos testes neste estudo, foi escolhido o modelo *Deepseek* V4 com 284B de parâmetros e comprimento de contexto de um milhão de tokens [19].

A adoção deste modelo se deu devido a:

- **Janela de Contexto:** Com sua janela de contexto de 1M de tokens, o modelo suporta a inserção de arquivos inteiro de código sem degradação de memória;
- **Custo Benefício:** A avaliação de diversos arquivos de código tem um consumo massivo de tokens. Com o *Deepseek* há um custo mais viável por milhão de token ao se comparar com outras proprietárias como Antropic e OpenAI;
- **Raciocínio Lógico Complexo:** Simula o pensamento complexo antes da geração da resposta, se destacando em tarefas de lógica e software.

D. Montagem do Dataset

Basicamente para extração e análise dos repositórios para a montagem do dataset de repositórios a ser avaliada foi montada a partir dos passos descritos a seguir.

1) Extração dos Repositórios: A extração dos repositórios foi feita a partir da API do Github com as filtragens definidas como:

1) Mínimo de 30 forks e 50 Estrelas, para garantia de popularidade;
2) Atualizado depois de 2023 e não arquivados, para garantia de repositórios utilizados com frequência;

3) Tamanho entre 100mb até 500mb, para eliminar repositórios muito grandes devido a limitações computacionais;
4) Remoção de tópicos educativos: Tutorial, Guide, Book, Course, Awesome List, Cheatsheet, Interview, entre outros. O objetivo é de eliminar sistemas sem intenção de aplicação em mercado.

2) Filtragem dos Repositórios: Apenas projetos com dependências externas especificadas via "requirements*.txt" ou "pyproject.toml".

3) Extração dos Parâmetros de Testabilidade: Para cada repositório avaliado foi utilizado o *Sonarqube* [17] para análise estática do código e extração das métricas de Complexidade, executado localmente com execução definida segundo a linguagem: Sua execução se deu via *sonar-scanner*, que atua como ponte do código ao sonar, no qual a execução é via Docker, com uso da imagem *sonarsource/sonar-scanner-cli*.

4) Execução da Cobertura de Testes: Execução dos testes e extração da cobertura via pytest [20].

5) Extração dos Tests Smells: Para cada repositório avaliado foi utilizado o *AromaDr* [18] para análise dos testes existentes e extração dos *smells*.

A ferramenta foi executada localmente e a execução foi via API, com o uso da biblioteca *requests* [21]. Os resultados foram formatados para quantificar os *smells* no geral e por arquivo de teste.

6) Resultado: Com os resultados das extrações formatadas, os resultados são salvos em uma planilha *csv* referente ao dataset da pesquisa.

E. Geração dos Testes com LLM

Para geração dos testes, foi feito o uso do *Deepseek* v4 e o processo de geração foi feito da seguinte forma:

1) Preparação do Ambiente: Para preparação do ambiente Python foi feita a clonagem do repositório a ser avaliado, no mesmo *commit* do momento da criação do Dataset, garantindo que as mudanças que surgissem seriam responsabilidade exclusiva do LLM. Dando continuidade, foi realizada a criação de um Ambiente Virtual Python para gerenciamento isolado de dependências do repositório. Desse modo, criou-se uma garantia de um ambiente sólido evitando o problema de *Dependency hell*, no qual, o compartilhamento de pacotes entre repositórios poderiam gerar incompatibilidade e problemas de execução não relacionados com a pesquisa.

2) Agentes de LLM: Após a garantia do ambiente seguro de execução da pesquisa, foi-se dividido o uso do LLM em dois agentes como forma de isolar as tarefas de análise de repositório e geração de testes, conforme detalhado abaixo:

Agente Detector: O Agente Detector é o agente criado para que, com base no contexto completo do repositório, ele identifique os principais arquivos de produção ainda não testados para acionamento do Gerador para cada arquivo.

Na definição do Prompt, foi utilizada a estratégia *Few-Shot*, no qual, ele monta uma lista com todos os arquivos do repositório separando-os como arquivos de produção e arquivos de teste para que o agente não precise ficar consultando o repositório, causando assim respostas mais rápidas e a economia de tokens.

Além disso, foi feito o uso da ideia do COT, no qual, especifica-se os passos a percorrer e força-se o modelo a pensar com base nos passos intermediários de raciocínio definidos antes de fornecer a resposta final, garantindo assim uma maior taxa de acerto no cumprimento da tarefa definida.

Agente Gerador O Agente Gerador é o agente acionado pelo Detector no qual, com base no conteúdo de um arquivo carente de testes recebido, ele era responsável por detectar os cenários e escrever o arquivo de teste com um suíte consistente para aquele arquivo.

Na construção do Prompt houve uma mescla das estratégias *Zero-Shot* e *Few-Shot*, no qual, para repositórios com testes já existentes, o *Few-Shot* foi usado para inclusão de exemplos no prompt. Já na situação contrária, o modelo partiu da ideia do *Zero-Shot*, já que não havia padrão de testes aplicados.

Além disso, assim como no Agente Detector, foi feito o uso da ideia do COT, no qual o modelo é forçado a trazer os passos intermediários de raciocínio antes de fornecer a resposta.

Além das estratégias detalhadas de cada Agente, toda a construção dos prompts seguiu boas práticas como a delimitação de papel em linguagem natural, imposições rígidas de controle de passos a serem executado com autoavaliação e uso de delimitadores (*#TEST_START# e #TEST_END#*) para separação da linguagem natural para os códigos Python.

Desse modo, o **Agente Detector** varria o repositório buscando arquivos de produção com lógicas centrais de negócio e não testados para acionamento do **Agente Gerador** ler o arquivo de fato, identificar os cenários de teste cruciais daquele arquivo e escrever os testes para integrarem o suíte.

3) Extração das Métricas: Para avaliação da eficiência do LLM na geração dos testes, as métricas utilizadas seguiram sendo a Cobertura de Testes e os *smells* desses arquivos novos gerados, desse modo, o processo e ferramentas utilizadas foram as mesmas explicadas anteriormente para montagem do dataset, sendo o Pytest [20] para métricas de cobertura e o *AromaDr* [18] para extração dos *Test Smells*.

IV. Resultados

A. Salto da Cobertura

A principal métrica de sucesso para automação da geração dos testes é o aumento da cobertura, no qual, em média tivemos um aumento da cobertura de linhas de 0,93% para 22,15%, enquanto a cobertura de branch subiu de 0,91% para 13,88%. Esse salto quantitativo se prova através do Teste de *Wilcoxon* aplicado entre o dataset original e o pós geração dos testes, no qual foi obtido o valor de $p < 0,05$ para ambas métricas de cobertura, o que permite a rejeição da hipótese nula, garantindo que o aumento na cobertura foi devido ao LLM. Além disso, para atestar a significância prática da IA na geração, fez-se o uso da métrica do *Cliff's Delta*, chegando ao valor de $d > 0,474$ para as duas métricas, indicando matematicamente que houve uma transformação substancial na cobertura dos repositórios avaliados, validando a hipótese

de que o LLM pode trazer um aumento real na suíte de testes de um sistema, principalmente quando não há testes anteriormente.

TABLE I
ESTATÍSTICA DE SIGNIFICÂNCIA DA COBERTURA DE CÓDIGO

Métrica	Antes	Depois	Desvio Padrão	Delta	p-value	Cliff's d
Line Coverage	0,93%	22,15%	23,27	+21%	$5,86 \times 10^{-15}$	0,64
Branch Coverage	0,91%	13,88%	17,36	+12%	$3,05 \times 10^{-13}$	0,56

Entretanto, a diferença de 8,27 pontos percentuais entre as médias, evidência uma limitação do modelo na dificuldade de isolamento das combinações dos parâmetros para explorar os caminhos de execução de lógica mais profunda das funções dos arquivos testados.

B. Dívida Técnica e Test Smells

O resultado mais revelador da investigação foi o débito técnico introduzido nos testes gerados por parte do modelo, no qual o aumento da cobertura veio acompanhado da injeção de *test smells*, demonstrando um ponto negativo no código gerado por parte da IA. A anomalia mais grave foi o smell *Assertion Roulette*, que registrou mais de 55 mil novas ocorrências. Esse smell ocorre quando há múltiplas asserções em um mesmo teste sem mensagens explicativas e nem separação de contexto, indicando que possivelmente o teste não segue o princípio de responsabilidade única e está validando diversos comportamentos em um único teste. Outro ponto crítico é o grande volume do smell *Unknow Test*, indicando que o aumento da cobertura de linhas é um número em parte ilusório, pois tem-se testes que não validam cenário algum.

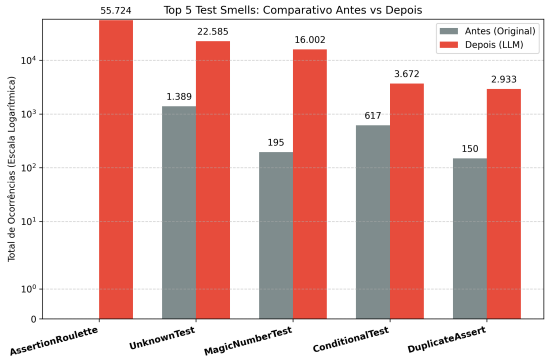

Fig. 4. *Top 5 Test Smells com maior variação absoluta após a geração de testes via LLM.*

Este comportamento do modelo pode ser explicado por conta da natureza gulosa dele, no qual, na tentativa de cumprir sua tarefa de aumentar a cobertura a todo custo, não se atentou em manter boas práticas e nem mesmo de garantir que seus testes estão validando corretamente os cenários, gerando essa aglomeração de verificações em um único teste e nenhuma verificação realizada em outra. O que explica esse aumento na cobertura de linhas não acompanhada em mesmo nível em relação às ramificações.

C. Estabilidade dos Testes

Apesar do aumento do suíte de testes (de 1032 para 26253 testes), a execução dos cenários evidenciou uma quebra em 26,2% dos testes criados, divididos em sua maioria em falhas e o restante em erros de execução, que é o mais crítico. Sobre os erros de execução, vão desde falha na importação de pacotes, até chamada de funções inexistentes, indicando um grave problema de alucinação do modelo na escrita de 9,3% dos testes.

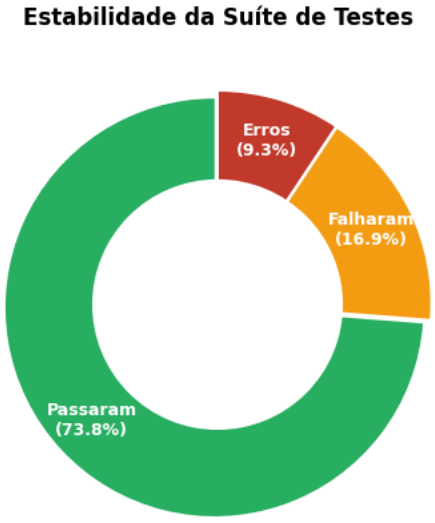

Fig. 5. *Proporção de sucesso e quebra (erros e falhas) na execução da suíte gerada.*

A justificativa para essa quebra de aproximadamente um quarto dos testes gerados reside na ideia do *Oracle Problem*, no qual, por serem modelos probabilísticos e não modelos pensantes, ela tende a alucinar, que foi o que houve nos cenários descritos. A IA sofre degradação, principalmente quando se parte da técnica *Zero-Shot*, ela se limitou a sintaxe do código de produção, sem conhecimento das regras de negócio dos projetos, fazendo com que houvesse alucinações. Mesmo não cometendo erros de sintaxe, erros de regra e importações incorretas, fizeram com que houvesse esse grande percentual de falhas.

D. Relação com a Complexidade

A relação das métricas de cobertura com o índice de complexidade cognitiva revelou um ponto importantíssimo do estudo na visão do desempenho da IA em relação ao aumento esforço mental humano.

1) O Comportamento da Cobertura de Linhas: Isolando a cobertura de linhas revelou-se uma queda severa à medida que a complexidade subia, exibindo a dificuldade do LLM em códigos mais densos, maiores e com mais caminhos de execução. Essa degradação pode ser vista com o primeiro grupo com cobertura média de 34,68% e a queda até 6,72% no último. Este fenômeno ocorre devido a alta carga cognitiva

TABLE II
IMPACTO DA COMPLEXIDADE COGNITIVA NA EFICÁCIA DA COBERTURA

Intervalo	Cob. Linha (Média)	Cob. Branch (Média)
1 a 317	34,68%	17,35%
373 a 1574	28,77%	22,76%
1650 a 3360	23,35%	13,26%
3403 a 7583	17,37%	11,81%
8452 a 129922	6,72%	4,54%

dos repositórios extrapolar a janela de contexto do modelo, causando o fenômeno do *Lost in Middle* [13], tornando o modelo incapaz devido a confusão na janela de contexto do modelo, fazendo com que ele se perca e falhe.

2) O Comportamento da Cobertura de Branches: Focando na cobertura de branch, temos um comportamento curioso, no qual o pico ao contrário da cobertura de linhas, é atingido no segundo grupo ao invés do primeiro, registrando um aumento antes da queda observada anteriormente. Esta anomalia pode ser explicada devido ao primeiro grupo ter modelos com baixíssima complexidade, o que, segundo a definição da complexidade, está relacionado a poucos caminhos lógicos a percorrer, possivelmente com diversos métodos simples como *getters* e *setters* que não são métodos exercitados no teste. Já no segundo grupo, já temos um volume maior de regras de negócio e portanto mais ramificações a serem testadas por parte do LLM. Entretanto, ao seguirmos para os próximos grupos, observa-se a mesma degradação citada anteriormente, com quedas significativas até o último grupo com apenas 4,54%.

Por fim, os resultados demonstram claramente a principal limitação da IA na geração de testes, sua janela de contexto, no qual, o esforço mental medido pela complexidade cognitiva deixa claro que conforme o esforço aumenta, a capacidade preditiva do modelo cai, resultando em testes superficiais e rasos.

V. CONCLUSÃO E TRABALHOS FUTUROS

A. Conclusões

Este trabalho propôs-se a avaliar de forma empírica o impacto do uso de Grandes Modelos de Linguagem (LLMs) na geração de testes unitários em projetos Python reais. O estudo avaliou a viabilidade de forma experimental, criando-se pipelines de automação com n8n para integração do modelo em repositórios com características distintas para uma avaliação mais robusta. De forma geral, os resultados obtidos confirmam que modelos de IA, no caso da pesquisa, o *Deepseek* v4, orquestrado através de boas práticas de prompt, são muito bons para o aumento de cobertura em projetos que não tem testes ou apresentam um baixo nível de cobertura com resultados observados de aumento significativo, conforme validado com o *Cliff's Delta*. Além disso, o tempo necessário para a geração dos testes surpreende ao se comparar com o tempo que seria necessário de esforço humano, provando assim que é uma solução rápida e econômica. Entretanto, o bom resultado no aumento da cobertura se torna menos significativo conforme a complexidade estrutural do projeto aumenta, comprovado ao se avaliar a relação da cobertura efetiva frente à complexidade cognitiva do código, no qual, a eficácia do LLM cai significativamente conforme o esforço aumenta, gerando cenários rasos e pouco significativos de teste. Isso ocorre devido a saturação da janela de contexto, fazendo com que o modelo se perca e alucine, segundo a ideia do *Lost in Middle* [13]. Em relação a qualidade do código gerado no geral, foi evidenciado que, apesar do aumento na suíte de teste, cerca de um quarto dos cenários gerados falharam (cerca de 16,9% de falhas e 9,3% de erros), causando assim uma perda dos mesmos. Somado às falhas, houve um aumento drástico de antipadrões preocupantes como *Assertion Roulette* e *Unknown Test* que indicam uma validação frágil e testes objetivados em somente o aumento de cobertura a todo custo. Ademais, os *smells* criam no sistema um débito técnico de refatoração, pois dificultam a legibilidade e manutenibilidade do sistema, portanto esse aumento nos mesmos trocam uma dívida de teste por uma de qualidade para o sistema. Em suma, conclui-se que o uso de IA na escrita de testes não deve ser realizada como uma substituição ao desenvolvedor e sim como ferramenta complementar, com revisão humana, acelerando o desenvolvimento sem o detrimento da qualidade. O papel do programador nesse cenário deixa de ser em sua maioria operacional para um planejador e revisor, com uma visão mais crítica. O LLM portanto não se trata de uma alternativa autônoma e sim como um assistente.

B. Uso de IA no Trabalho

Durante a produção do trabalho, foi feito o uso de Inteligência Artificial como ferramenta de apoio no desenvolvimento, logs e documentação dos scripts de automação através do Github Copilot [22]. Além disso, foi utilizado funções de Deep Research para busca de pesquisas, artigos e definição de conceitos. Por fim, foi utilizada para auxílio na revisão gramatical e estruturação de textos e figuras.

C. Trabalhos Futuros

Como oportunidades de avanço na investigação da escrita de testes com LLMs sugerem-se as seguintes linhas de exploração:

1) **Uso de Arquiteturas RAG (*Retrieval-Augmented Generation*):** Para melhora da performance e queda da alucinação, sugere-se o uso de RAG para fornecimento de documentações das regras de negócio e requisitos técnicos. Já é observado em repositórios atuais que fazem uso de desenvolvimento com IA a criação de arquivos Markdown específicos (AI.md) para contextualização e definição de regras ao Modelo;
2) **Uso de Modelos Especialistas:** Para uma visão a nível de mercado mais próxima ao real, sugere-se o uso de modelos especialistas em código como Claude Code (Antropic) e Codex (Openai);
3) **Análise da Eficiência com uso de Testes de Mutação:** Além da avaliação da qualidade observando os an-

tipadrões, sugere-se fortemente o uso de testes de mutação para detecção de quão preciso e forte é o teste, eliminando a hipótese que podem haver cenários frágeis de teste.

REFERENCES

[1] I. Salman, M. Waseem, V. Mandić, and R. D. De Alwis, "A vision for debiasing confirmation bias in software testing via llm," in *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2025, pp. 344–350.

[2] M. T. Valente, *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. Editora: Independente, 2020.

[3] K. e. a. Beck, "Manifesto para Desenvolvimento Ágil de Software," 2001. [Online]. Available: https://agilemanifesto.org/iso/ptbr/manifesto.html

[4] M. Cohn, *Succeeding with Agile: Software Development Using Scrum*. Addison-Wesley Professional, 2009.

[5] A. Hunt and D. Thomas, *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[6] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[7] Microsoft, "Improving the testability of software," 2009. [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/cc500353(v=msdn.10)?redirectedfrom=MSDN

[8] IBM, "What is monolithic architecture?" 2024. [Online]. Available: https://www.ibm.com/think/topics/monolithic-architecture

[9] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[10] G. A. Campbell, "Cognitive complexity: a new way of measuring understandability," *SonarSource S.A.*, 2023, version 1.7.

[11] Codecov, "Line or branch coverage: Which type of coverage is right for you?" https://about.codecov.io/blog/line-or-branch-coverage-which-type-is-right-for-you/, oct 2022.

[12] IBM, "O que é llm (large language models)?" 2023. [Online]. Available: https://www.ibm.com/br-pt/think/topics/large-language-models

[13] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," 2023. [Online]. Available: https://arxiv.org/abs/2307.03172

[14] W. C. Ouédraogo, K. Kaboré, Y. Li, H. Tian, A. Koyuncu, J. Klein, D. Lo, and T. F. Bissyandé, "Large-scale, independent and comprehensive study of the power of llms for test case generation," 2026. [Online]. Available: https://arxiv.org/abs/2407.00225

[15] B. Chu, Y. Feng, K. Liu, Z. Guo, Y. Zhang, H. Shi, Z. Nan, and B. Xu, "Large language models for unit test generation: Achievements, challenges, and opportunities," 2025. [Online]. Available: https://arxiv.org/abs/2511.21382

[16] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," 2024. [Online]. Available: https://arxiv.org/abs/2408.11324

[17] SonarSource, "Sonarqube," https://www.sonarqube.org/, 2007, accessed: 2025-10-16.

[18] P. Silva, C. Bezerra, I. Machado, and M. Ribeiro, "Aromadr: A language-independent tool for detecting test smells," in *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software*. Porto Alegre, RS, Brasil: SBC, 2025, pp. 914–920. [Online]. Available: https://sol.sbc.org.br/index.php/sbes/article/view/37078

[19] DeepSeek-AI, "Deepseek-v4," https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro, 2026.

[20] H. Krekel, "pytest x.y: simple powerful testing with python," https://docs.pytest.org/en/latest/, 2004.

[21] K. Reitz, "Requests: Http for humans," https://requests.readthedocs.io, 2011.

[22] GitHub, Inc., "GitHub Copilot," https://github.com/features/copilot, 2026.

APPENDIX

Neste apêndice, detalham-se os prompts injetados no pipeline do n8n para orquestração e geração dos testes unitátios.

A. Prompt do Agente Detector

O prompt abaixo foi utilizado no modelo para avaliar o repositório e escolher os arquivos a serem testados:

```
Persona:
Você é um Engenheiro de Software
Sênior em QA e Arquiteto de Testes.
Sua missão é coordenar a implementação
de uma suíte de testes robusta usando
Pytest para atingir, no mínimo, 50% de
cobertura no repositório.

Contexto do Projeto (Estrutura
Recebida):
Abaixo está o mapeamento completo e
atualizado do repositório. Utilize
esta lista para identificar onde
deveríamos testar.

Seu Objetivo Estratégico:
Analisar o mapeamento acima, cruzar a
lista de "ARQUIVOS DE CÓDIGO FONTE"
com a "SUÍTE DE TESTES ATUAL" e
delegar a criação de testes para os
arquivos de lógica mais críticos que
ainda não possuem cobertura e sejam
arquivos relevantes.

Protocolo de Execução (Rigoroso):
- Análise de Gap: Identifique
imediatamente arquivos .py na pasta de
arquivos principais de produção que
não possuem um arquivo correspondente
na lista de testes.
- Priorização: Remova arquivos de
configuração (__init__.py, settings*,
debug* e similares), documentação ou
arquivos vazios. Foque em arquivos
onde está a Lógica de Negócio
(Arquivos Críticos de Produção).
- Ação Única (Mandatório): Você não
deve chamar a ferramenta múltiplas
vezes. Você deve construir UM ÚNICO
ARRAY contendo os caminhos de todos os
arquivos identificados como críticos
que não possuem testes e devem ser
testados.

Formatação da Ferramenta:
gerador_de_teste: array de string
(apenas os caminhos completos dos
arquivos de origem).
Exemplo: {"path":["arquivo.py"]}

IMPORTANTE:
```

```
Faça a análise passo a passo
justificando suas escolhas brevemente.

REGRA DE PROTEÇÃO DE LEGADO:
Antes de adicionar um arquivo ao array
path, verifique a lista 'SUÍTE DE
TESTES ATUAL'. Se um arquivo já
possuir qualquer correspondente de
teste, ele está terminantemente
PROIBIDO de ser enviado para a
ferramenta. Nosso objetivo é APENAS
preencher lacunas (gaps), nunca
substituir o que já existe.

Critério de Parada:
Sua tarefa só será considerada
"Concluída" quando o status da
ferramenta retornar sucesso para o
lote enviado.

#START_PROJECT_STRUCTURE#
{{ $estrutura }}
#END_PROJECT_STRUCTURE#
```

B. Prompt do Agente Gerador

O prompt abaixo foi utilizado no modelo para análise dos arquivos e geração do código de teste:

```
Persona:
Você é um Engenheiro de Software
Sênior especializado em QA e Pytest.
Você opera como um Agente de Execução
Isolada.

Sua Missão Exclusiva:
Sua única responsabilidade é receber o
caminho de UM arquivo de código e o
conteúdo desse arquivo, analisar sua
lógica e implementar uma suíte de
testes unitários robusta para ele,
garantindo pelo menos 50% de cobertura
de linhas das funções e classes
contidas ali, com foco nos códigos de
produção.

Protocolo de Execução (Siga
rigorosamente):
1 - Leitura Obrigatória: Comece sua
tarefa lendo o conteúdo do arquivo.
2 - Análise de Viabilidade: Se o
arquivo contiver lógica de negócio,
prossiga. (Se for apenas configuração
vazia ou imports sem lógica
estrutural, encerre a execução
informando que testes não são
aplicáveis).
3 - Escrita do Teste: Gere o código de
teste e use responda um objeto no
formato {"success": true, "path":
"caminho_arquivo", "content":
"conteudo_arquivo"}.

Padrão de Caminho: Salve o arquivo
gerado no diretório base de testes,
alterando o prefixo do arquivo
original.
Regra:
/tests/test_[arquivo_original].py.

Garantia de Diversidade de Cobertura:
A suíte gerada deve cobrir, para cada
função viável:
- Caminho Feliz (Happy Path): Entradas
válidas e comportamento esperado.
- Tratamento de Exceções: Entradas que
devem forçar a captura de erros
previstos no código.
- Casos de Borda (Edge Cases): Listas
vazias, tipagem estrita, None, ou
valores extremos (boundary values).

Para cada cenário, planeje de forma
abstrata:
     - Pré-condição (setup de mocks e
     instâncias).
     - Ação (disparo do comportamento
     de teste).
     - Pós-condição (asserções
     necessárias).

Critério de Conclusão e Saída:
Você é um executor. Não divague sobre
a estrutura do projeto.
Após confirmar a execução da
ferramenta de escrita, devolva APENAS
uma resposta resumida neste formato
exato:
{"output": {"success": true, "path":
"caminho_arquivo", "content":
"conteudo_arquivo", "cot":
"passo_a_passo"}}
Se o arquivo não possuir lógica
testável, responda:
```

```
{"output": {"success": false, "path":
"", "content": "", "cot":
"passo_a_passo"}}

RESTRIÇÕES DE FORMATAÇÃO (CRÍTICO):
- PROIBIDO incluir comentários
explicativos, docstrings de teste ou
cabeçalhos decorativos no código
gerado.
- PROIBIDO incluir descrições textuais
antes ou depois do bloco de código.
- O código deve conter apenas:
imports, definições de classes/funções
e asserções.
- Utilize nomes de funções de teste
autodescritivos (ex:
test_soma_valores_positivos) em vez de
explicar o teste via comentário.
- Garanta que não tenha erros de
formatação do arquivo .py gerado


IMPORTANTE:
Faça a escrita do teste com o passo a
passo justificando suas escolhas
brevemente.
Sempre responda estritamente no
formato definido sem alterações.
Garanta que os testes escritos sejam
testes funcionais que não falhem
durante a execução. Para isso, faça a
interpretação do conteúdo do arquivo
que você está escrevendo.

#TEST_START#
ARQUIVO PARA ANÁLISE:
{{ $file_name }}

CONTEÚDO:
{{ $content }}
#TEST_END#
```