Universidade de Brasília – UnB
Faculdade UnB Gama – FGA
Engenharia de Software

Uso de IA Generativa na Geração Automática de Casos de Teste Unitários

Autor: Lucas Gabriel Bezerra
Orientadora: Dra. Elaine Venson

Brasília, DF
2024

Lucas Gabriel Bezerra

Uso de IA Generativa na Geração Automática de Casos de Teste Unitários

Monografia submetida ao curso de graduação em Engenharia de Software da Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília – UnB

Faculdade UnB Gama – FGA

Orientador: Dra. Elaine Venson

Brasília, DF

2024

Agradecimentos

Gostaria de expressar minha sincera gratidão a todos que contribuíram para a realização deste trabalho.

Primeiramente, agradeço a Deus por me permitir concretizar um sonho da minha família.

Em seguida, um agradecimento especial à minha família e à minha namorada Elana, cujo apoio incondicional e encorajamento foram pilares constantes ao longo desta jornada. Aos meus pais, Jovelina e Vladilson, e à minha irmã, Luana, seus sacrifícios e compreensão foram cruciais para que eu pudesse superar os desafios enfrentados durante o processo.

Agradeço também aos professores e colegas da FGA, em especial à minha orientadora, Elaine Venson, pela orientação valiosa, paciência e apoio contínuo durante toda a elaboração deste trabalho.

A todos, meu mais profundo agradecimento.

Resumo

A área de Inteligência Artificial (IA) tem experimentado um crescimento significativo e uma crescente popularização nos últimos anos, assim como as aplicações de software, que se tornam cada vez mais complexas. Esse avanço torna a fase de teste do ciclo de desenvolvimento de software uma parte significativa dos custos e do tempo envolvidos, sendo uma das atividades mais demoradas, mesmo com a introdução de ferramentas de automação. Com o progresso das tecnologias de IA, especialmente com os modelos conhecidos como *Large Language Models* (LLMs), surge uma nova possibilidade: a automatização não apenas da execução dos testes, mas também de sua geração. O GPT (*Generative Pre-trained Transformer*), uma das implementações mais avançadas de LLM desenvolvidas pela OpenAI, é capaz de compreender e gerar texto de forma contextualizada. Nesse contexto, esta pesquisa utiliza o modelo GPT para gerar casos de teste unitários e compara-os com aqueles desenvolvidos manualmente por programadores avaliando a eficácia e a precisão dos testes gerados pela IA. O objetivo principal é analisar a capacidade das ferramentas LLM na geração de testes, buscando compreender se tais tecnologias poderiam ser uma alternativa viável para mitigar os custos e o esforço associados à criação de testes unitários no ciclo de desenvolvimento de software. A metodologia envolveu a criação de *scripts* e *prompts* personalizados para gerar e executar os testes, além de coletar métricas de cobertura de código. Os resultados revelaram um aumento expressivo na quantidade de testes unitários, mas com impacto mínimo na cobertura e uma taxa significativa de falhas nos testes gerados, evidenciando limitações na assertividade e eficácia do GPT. Isso indica que, o modelo ainda não é capaz de substituir completamente o trabalho humano nessa tarefa. Conclui-se que as ferramentas LLM têm potencial como apoio para desenvolvedores, mas ainda requerem avanços para uma automação mais confiável e eficaz da geração de testes unitários.

Palavras-chave: Inteligência Artificial; Teste de Software; GPT; *Large Language Model*.

Abstract

The field of Artificial Intelligence (AI) has experienced significant growth and increasing popularity in recent years, along with software applications that are becoming increasingly complex. This advancement has made the testing phase of the software development cycle a significant part of the costs and time involved, being one of the most time-consuming activities, even with the introduction of automation tools. With the progress of AI technologies, especially models known as Large Language Models (LLMs), a new possibility arises: the automation of not only test execution but also their generation. GPT (Generative Pre-trained Transformer), one of the most advanced LLM implementations developed by OpenAI, is capable of understanding and generating context-aware text. In this context, this research utilizes the GPT model to generate unit test cases and compares them to those manually developed by programmers, evaluating the effectiveness and accuracy of the tests generated by the AI. The main goal is to analyze the capacity of LLM tools in test generation, aiming to understand whether such technologies could be a viable alternative to mitigate the costs and effort associated with creating unit tests in the software development cycle. The methodology involved the creation of personalized scripts and prompts to generate and execute the tests, as well as to collect code coverage metrics. The results revealed a significant increase in the number of unit tests, but with minimal impact on coverage and a high rate of failures in the tests generated, highlighting limitations in GPT's accuracy and effectiveness. This indicates that the model is still not capable of fully replacing human work in this task. It is concluded that LLM tools have the potential to assist developers, but they still require advancements for more reliable and effective automation of unit test generation.

Key-words: Artificial Intelligence; Software Testing; ChatGPT; Large Language Model.

Lista de Ilustrações

Lista de Tabelas

Lista de abreviaturas e siglas

AM	Aprendizado de Máquina
GPT	*Generative Pre-trained Transformer*
GPU	Unidade de Processamento Gráfico
IA	Inteligência Artificial
IDE	*Integrated Development Environment*
LLM	*Large Language Model*
PLN	Programação de Linguagem Natural
RNA	Rede Neural Artificial
TCC	Trabalho de Conclusão de Curso

Sumário

1 Introdução

1.1 Considerações Iniciais

Este capítulo apresenta o contexto deste trabalho, o problema de pesquisa, o objetivo geral e os objetivos específicos, a metodologia de pesquisa empregada e a organização dos capítulos subsequentes.

1.2 Contexto

Nas últimas décadas, a área de desenvolvimento de software está em alta, com crescimento estimado até 2033 de aproximadamente 26%, segundo a U.S. Bureau of Labor Statistics (2023). A preocupação em torno da qualidade das aplicações tem acompanhado este crescimento, visto que as aplicações têm se tornado mais importantes e determinantes em diversas outras áreas. A maior preocupação com a qualidade, aumentou o enfoque em todas as fases do desenvolvimento, sendo a fase de teste de software uma das principais afetadas.

No início dos anos 80, a fase de teste de software representava cerca de 50% do tempo gasto e mais de 50% do custo de desenvolvimento (MYERS; SANDLER; BADGETT, 2011). A fim de melhorar e otimizar o teste das aplicações, diversas técnicas, ferramentas e estratégias foram desenvolvidas ao longo dos anos. Contudo, apesar dos avanços, uma pesquisa realizada por Lukashik (2022) indicou que, para 35% das empresas participantes, o teste manual é a atividade que mais consome tempo no ciclo de teste. Além disso, 36% dos testadores indicaram que entre 10% e 50% dos testes são realizados por testadores não dedicados (LUKASHIK, 2022).

Simultaneamente ao crescimento do desenvolvimento de aplicações, surgia em meados dos anos 50 a Inteligência Artificial (IA). Com o objetivo de criar uma máquina que pudesse resolver problemas outrora solucionados apenas por seres humanos, John McCarthy, Marvin Minsky, Claude Shannon e Nathaniel Rochester organizaram um seminário de dois meses em Dartmouth, com a participação de alguns pesquisadores entusiastas (RUSSELL; NORVIG; DAVIS, 2010). Este seminário foi o ponto de partida para a criação de uma comunidade e intensificação nos estudos sobre IA.

Apesar do entusiasmo inicial, criado a partir do seminário, o poder computacional da época era um grande limitante para o desenvolvimento da IA. Desta forma, durante as décadas seguintes o desenvolvimento foi contido. Até que em meados dos anos 80, a IA se torna uma indústria que passa a movimentar bilhões de dólares, graças à combinação de

fatores: evolução tecnológica, adoção do método científico nas pesquisas, a reinvenção do algoritmo de aprendizado por retroprogamação (modelo de rede neural) (TOTVS, 2024).

Atualmente, no século XXI, a evolução da IA atingiu outro nível, principalmente no que tange as áreas de Aprendizado de Máquina (AP) e Processamento de Linguagem Natural (PLN), devido ao crescimento da disponibilidade de dados advindo com a internet e o *Big Data*. A IA se tornou popular e tem sido inserida em atividades e campos relacionados ou não com tecnologia, como, por exemplo: a área automobilística com veículos autônomos, áreas administrativas com o planejamento logístico de transporte de cargas, e no dia-a-dia com assistentes virtuais inteligentes (Alexa, Cortana, Bixby) e as IA conversacionais como o ChatGPT[1] e o Bard[2] (JADHAV, 2024).

O avanço da Inteligência Artificial em todas as suas subáreas tem sido notável, especialmente devido ao crescente interesse na IA conversacional. Essa vertente, originada do ramo de Processamento de Linguagem Natural (PLN), apresenta ferramentas baseadas em modelos *Large Language Models* (LLM). O maior expoente, atualmente, deste tipo de ferramenta é o ChatGPT. A utilização da ferramenta, para melhora da produtividade na construção de programas, vem acontecendo gradualmente, isso tem gerado debates e pesquisas sobre em quais fases se aplicar ferramentas LLM para melhorar e agilizar o desenvolvimento de aplicações.

Considerando a crescente demanda por aplicativos nos mais diversos negócios e a relevância dos testes na garantia da qualidade, segurança e confiabilidade, observa-se a necessidade de otimizar o processo de teste das aplicações. A produtividade dos testes de software pode ser melhorada com ferramentas, o que já é feito na indústria no que se refere à automatização da execução dos testes. Recentemente, novas perspectivas se abrem com as ferramentas LLM e sua capacidade de gerar código e auxiliar no *debug*. Tais ferramentas permitem pensar na automatização não só da execução de testes, mas também na geração automática dos casos de teste.

1.3 Problema

O crescimento da atividade de desenvolvimento do software, assim como a importância de garantir a qualidade das aplicações no mundo moderno, gera a necessidade de minimizar o tempo e o esforço gastos na testagem das aplicações, permitindo uma melhor relação custo-benefício nos projetos. Por sua vez, a automatização das tarefas de teste por meio de IA tem se mostrado como uma oportunidade para reduzir o esforço despendido pelo desenvolvedor em tais atividades.

Neste contexto, a pergunta de pesquisa na qual se baseia este trabalho é: **As fer-**

[1] https://openai.com/blog/chatgpt
[2] https://bard.google.com/

ramentas de Inteligência Artificial, baseadas em algoritmos LLM, são capazes de gerar testes unitários comparáveis aos testes feitos por desenvolvedores?

1.4 Objetivo

1.4.1 Objetivo Geral

O objetivo geral deste trabalho é analisar a capacidade de geração de teste unitários a partir de ferramentas LLMs.

1.4.2 Objetivos Específicos

- Gerar casos de teste unitários para métodos, obtidos a partir de classes de um repositório de projeto de código aberto do GitHub, utilizando o modelo LLM GPT3.5 Turbo;
- Reparar os casos de teste gerados com erro e falha pelo modelo;
- Realizar análise comparativa, utilizando estatística descritiva, entre os casos de teste criados pelo modelo LLM e pelos desenvolvedores.

1.5 Metodologia

A metodologia deste trabalho é fundamentada em uma abordagem explicativa (GIL, 2017), direcionada à avaliação da capacidade de geração de testes unitários por ferramentas de Inteligência Artificial (IA) baseadas em algoritmos de Large Language Models (LLMs), com a utilização do modelo GPT versão 3.5.

Classificado como quantitativo, o estudo busca obter as métricas a partir da análise dos testes gerados pelo GPT e dos testes tradicionais elaborados por desenvolvedores. Serão analisadas a quantidade e eficácia dos testes gerados, além da comparação com testes tradicionais em termos de detecção de falhas, cobertura e identificação de possíveis deficiências nos casos de teste.

Para atingir os objetivos propostos, a pesquisa foi dividida em cinco etapas, uma adaptação da diagramação geral de pesquisa de Gil 2017. Algumas etapas foram retiradas por não serem necessárias nesta pesquisa. Desta forma, as etapas planejadas são: formulação do problema, seleção de amostras, elaboração de instrumentos para utilização de ferramenta, execução dos instrumentos, coleta de dados, análise de cobertura e análise comparativa.

O processo metodológico abrange a elaboração de instrumentos (*scripts*) para a utilização do da API OpenAI, incluindo pré-processamento de métodos, geração e reparo

dos testes gerados, e análise comparativa entre os testes elaborados pelo GPT e pelos desenvolvedores. A avaliação foi realizada com base em métricas objetivas e qualitativas, a fim de comparar a qualidade entre os testes gerados por IA e os testes implementados por desenvolvedores.

1.6 Organização do Trabalho

Este trabalho é composto por sete capítulos, organizados da seguinte forma:

- **Capítulo 1 - Introdução:** apresentação do contexto do trabalho, assim como o problema, os objetivos e a metodologia utilizada na pesquisa.
- **Capítulo 2 - Referencial teórico:** apresenta conceitos importantes de Teste de Software e Inteligência Artificial para o entendimento e compreensão do trabalho no seu desenvolvimento.
- **Capítulo 3 - Metodologia:** especifica a classificação da pesquisa e o planejamento, abordando todos as etapas necessárias para alcançar os objetivos propostos.
- **Capítulo 4 - Preparação para Geração de Testes:** Preparação do projeto, dos *scripts* e dos*prompts* para a atividade de gerar testes com o modelo GPT.
- **Capítulo 5 - Execução e Coleta de Dados:** Execução dos *scripts* e dos testes gerados e coleta de métricas e indicadores.
- **Capítulo 6 - Análise de Resultados:** Análise de gráficos e tabelas criados a partir das métricas e indicadores coletados.
- **Capítulo 7 - Conclusão e Trabalhos Futuros:** Conclusão sobre o trabalho realizado e possíveis expansões deste trabalho.

2 Referencial Teórico

2.1 Considerações Iniciais

Neste capítulo serão apresentadas os conceitos relativos a Inteligência Artificial (IA) e Teste de Software, visando facilitar o entendimento acerca dos temas abordados nesta pesquisa. Este capítulo apresenta as definições de IA, processamento de linguagem natural, *Large Language Model* (LLM), e as ferramentas criadas a partir do LLM, além da definição de teste de software, as técnicas de testes e critérios destas técnicas.

2.2 Inteligência Artificial

A inteligência artificial (IA) surge após a Segunda Guerra Mundial, quando várias pessoas, de forma independente, começaram a explorar a ideia de criar máquinas inteligentes. O matemático inglês Alan Turing foi um dos pioneiros nesse campo, dando uma palestra sobre o assunto em 1947 e escrevendo o artigo *Computing Machinery and Intelligence* onde ele aborda a questão da possibilidade de máquinas pensarem, propondo o que mais tarde ficou conhecido como o "Teste de Turing"(TURING, 1950).

À medida que a década de 1950 avançava, um número crescente de pesquisadores mergulhava no campo emergente da Inteligência Artificial, e entre eles destacava-se John McCarthy, reconhecido como uma figura de grande influência na área (ZHANG; LU, 2021). McCarthy definiu a IA como uma disciplina da ciência e engenharia focada na construção de máquinas inteligentes, especialmente programas de computador inteligentes, sem necessariamente se limitar a métodos biologicamente observáveis (KHALIQ; FAROOQ; KHAN, 2022). Essa visão ampla permitiu o desenvolvimento de abordagens diversificadas e inovadoras no campo da IA.

Em essência, a Inteligência Artificial procura desenvolver sistemas que possam realizar tarefas que normalmente exigiriam inteligência humana, como raciocinar, aprender, solucionar problemas, entender a linguagem natural, reconhecer padrões e tomar decisões autônomas. Essa definição ampla e dinâmica destaca a versatilidade e o potencial da IA em uma variedade de aplicações, desde assistentes virtuais até veículos autônomos e diagnóstico médico avançado.

Conforme a abrangência do campo de IA cresceu, a área se organizou em sub-áreas, onde cada uma delas tem um foco de desenvolvimento e aplicação, além de utilizar de diferentes métodos e algoritmos para alcançar tal objetivo. Dentre as sub-áreas da IA, destacam-se: aprendizado de máquina, métodos computacionais que utilizam experiência

para aprimorar o desempenho ou realizar previsões precisas; e processamento de linguagem natural, vertente que se concentra na interação entre humanos e sistemas inteligentes por meio de linguagem natural (HOURANI; HAMMAD; LAFI, 2019).

O avanço da Inteligência Artificial e sua subsequente popularização no século XXI foram impulsionados por diversos fatores, sendo a evolução das Unidades de Processamento Gráfico (GPUs) um dos elementos-chave. As GPUs possibilitaram um aumento significativo na capacidade computacional, permitindo que os computadores realizassem operações aritméticas complexas e paralelizassem tarefas de forma mais eficiente. Isso viabilizou o desenvolvimento de algoritmos cada vez mais sofisticados, capacitando os sistemas de IA a lidarem com conjuntos de dados extensos e a executarem processamentos complexos em tempo hábil (ZHANG; LU, 2021). Além disso, outro ponto crucial para o crescimento da IA foi a notável expansão na capacidade de armazenamento de dados (ZHANG; LU, 2021). Esse aumento no armazenamento possibilitou o acesso a uma grande variedade de dados, incluindo imagens, textos, vídeos e outros tipos de informações em volumes abundantes. A combinação desses avanços tecnológicos permitiu a criação de modelos de IA mais robustos e poderosos, expandindo as aplicações em vários setores que acabou por alcançar o público, no dia a dia, através de ferramentas baseadas em IA. Entre essas tecnologias, destacam-se o Processamento de Linguagem Natural, o Aprendizado de MáquinaAprendizado Profundo (AP) e os Modelos de Linguagem Grande (LLMs).

2.2.1 Processamento de Linguagem Natural

Para compreender o Processamento de Linguagem Natural (PLN), é essencial entender o conceito de linguagem natural. De acordo com Lyons uma linguagem é considerada natural quando é desenvolvida sem planejamento prévio, mas é governada por regras, que são produtos de abstração, o que é o caso da língua portuguesa (LANGENDOEN, 1993). As linguagens não naturais, por outro lado, são produto da construção humana. Elas se assemelham, ontologicamente, com as linguagens artificiais, matemáticas, lógicas e de programação.

O Processamento de Linguagem Natural é um campo da IA que estuda teorias e métodos capazes de capacitar os sistemas computacionais a compreender, interpretar e gerar linguagem de maneira semelhante à humana (KANG et al., 2020).

O surgimento do PLN ocorre por volta da década de 50, entretanto, Alan Turing, já havia germinado o conceito de PLN na "Máquina de Turing", proposta em 1936. No período entre 1936 e o início dos anos 50, a pesquisa fundamental em PLN foi conduzida, incluindo o trabalho de Shannon, que aplicou modelos probabilísticos a processos de Markov discretos para automação de linguagem e medição da informação contida na linguagem humana usando conceitos de entropia (BORDIGNON, 2016). Kleene investigou autômatos finitos e expressões regulares nos anos 1950, enquanto Chomsky propôs uma

gramática livre de contexto para o PLN em 1956, levando ao desenvolvimento de técnicas baseadas em regras e probabilidade (BORDIGNON, 2016). Mas foi o surgimento da IA, nesta década, e a sua subsequente união ao processamento de linguagem natural, que possibilitou a evolução do campo nas décadas seguinte, até hoje.

O PLN pode ser visto a partir de dois focos de pesquisa: A geração de linguagem natural e a compreensão da linguagem natural. A geração de linguagem natural envolve a capacidade dos sistemas de PLN em criar textos naturais e compreensíveis para os humanos. Essa área abrange desde a geração automática de frases até a produção de textos completos, como a escrita automática de histórias. Por outro lado, a compreensão da linguagem natural refere-se à habilidade dos sistemas em interpretar e entender a linguagem escrita ou falada. Essa vertente engloba tarefas como análise sintática, semântica e pragmática, permitindo extração de significado, identificação de intenções e contextos, e compreensão para prover uma resposta de maneira adequada às interações com os usuários (KANG et al., 2020).

Recentemente, o uso do PLN tem se expandido para aplicações que vão além da compreensão de textos e conversações, adentrando áreas como o desenvolvimento de software, com a geração de código-fonte. A geração de código utilizando PLN representa um campo de pesquisa desafiador e inovador. O GitHub Copilot é um exemplo de ferramenta, que utiliza PLN, para fornecer assistência no ciclo de vida do desenvolvimento do software, gerando código e corrigindo erros em diversas linguagens de programação, por meio de instruções e também do contexto (GitHub. . . ,).

De maneira geral, a codificação de software requer habilidades técnicas específicas e conhecimento em linguagens de programação. O uso do PLN nesse contexto busca criar ferramentas e modelos que permitam aos desenvolvedores interagir com os sistemas por meio de linguagem natural, convertendo automaticamente instruções em código executável, ou até mesmo a geração de código a partir da junção de instruções e outros códigos já existentes. Diversas ferramentas para geração e correção de código-fonte tem sido criadas, isso representa um avanço significativo na automação de tarefas repetitivas e na melhoria da produtividade dos desenvolvedores. No entanto, é importante notar que, apesar dos avanços, esses sistemas continuam em constante aprimoramento, e há desafios a serem enfrentados, como a precisão na sugestão de código, a compreensão de contextos complexos e a adaptação a diferentes linguagens de programação e estilos de codificação (Victory, 2023).

2.2.2 Aprendizado de máquina

O Aprendizado de Máquina (AM) é um campo que utiliza técnicas estatísticas e computacionais para ensinar aos computadores a capacidade de aprender e tomar decisões baseadas em padrões identificados nos dados (DURELLI et al., 2019). Assim como os

humanos têm diferentes formas de aprendizado, as máquinas também possuem distintas abordagens para adquirir conhecimento a partir de conjuntos extensos de dados.

Ao longo dos anos, diferentes abordagens foram desenvolvidas para capacitar a máquina a "aprender"a partir de um conjunto grande de dados. Desta forma, foi criada uma classificação para os algoritmos, com base na quantidade de supervisão que recebem durante o treinamento, sendo elas (GéRON, 2021):

- Aprendizado Supervisionado: os algoritmos são treinados com conjuntos de dados rotulados, ou seja, os dados de entrada estão associadas a saídas desejadas (rótulos ou respostas conhecidas) . Isso inclui algoritmos como regressão linear, árvores de decisão, SVM (Support Vector Machines), redes neurais, entre outros.

- Aprendizado Não Supervisionado: ao contrário do aprendizado supervisionado, nesta técnica os algoritmos são treinados com um conjunto de dados não rotulados, buscando por padrões nos dados. Isso inclui os algoritmos de *clustering*, redução de dimensionalidade, análise de componentes principais.

- Aprendizado Semi-Supervisionado: esta técnica é a combinação entre o aprendizado supervisionado e não supervisionado, onde os algoritmos são treinados com um conjunto de dados composto por uma pequena quantidade de dados rotulados e uma grande quantidade de dados não rotulados.

- Aprendizado por Reforço: esta técnica se difere das demais, nela os modelos aprendem a partir da interação com o ambiente. O sistema, também chamado de agente, observa o ambiente, seleciona e executa ações a fim de obter recompensas em troca. Desta forma, ele aprende a melhor estratégia para obter o maior número de recompensas ao longo do tempo. Isso inclui algoritmos comumente utilizados para jogos, como o *AlphaGo*[1] da empresa *DeepMind*, que foi capaz de vencer o campeão mundial de Go.

Essas estratégias, que podem ser mescladas, proporcionam diferentes abordagens para que as máquinas possam aprender e processar informações, viabilizando a resolução de uma ampla gama de problemas em diversos domínios. O aumento considerável na disponibilidade de dados e os avanços computacionais no século XXI têm despertado um interesse crescente nessa área, impulsionando pesquisadores e profissionais a explorar soluções que envolvem a aprendizagem a partir desses dados, por meio de algoritmos de Aprendizado de Máquina das mais variadas naturezas (DURELLI et al., 2019).

Dentre os algoritmos de aprendizado supervisionado, os algoritmos de rede neural artificial têm se destacado. Principalmente, por sua capacidade de resolver problemas

[1] https://deepmind.google/technologies/alphago/

complexos, muitas vezes associados às habilidades humanas, como a conversação, geração de texto.

2.2.2.1 Aprendizado Profundo

Diante do crescimento de interesse acerca do AM, novas técnicas, ainda mais avançadas, tem emergido, como o Aprendizado Profundo (AP), do inglês *Deep Learning*. O Aprendizado profundo se baseia em Redes Neurais Artificiais (RNA) com várias camadas treinadas para serem capazes de resolver problemas de alta complexidade (CHACKO; CHACKO, 2023). Estas, RNAs são estruturas compostas por camadas de neurônios interconectados, organizados em camadas de entrada, ocultas e de saída. Cada camada da rede realiza transformações nos dados, aprendendo representações cada vez mais sofisticadas à medida que os dados são propagados pela rede (CHACKO; CHACKO, 2023).

O Aprendizado Profundo, permitiu avanços significativos em diversos campos, principalmente no processamento de linguagem natural. Um ponto-chave para estes avanços está na capacidade de adaptar modelos pré-treinados para solucionar problemas para os quais não haviam sido treinados inicialmente. Esses modelos, como o GPT (*Generative Pre-trained Transformer*) da OpenAI[2] e o BARD[3], criado por pesquisadores da Google, são baseados em redes neurais profundas e podem ser treinados em grandes volumes de texto para entender e gerar linguagem natural (DSA, 2022).

O advento dos *Large Language Models* (LLMs) tem como centro os modelos baseados em transformação do aprendizado profundo. O termo LLMs não é tão popular, mas aplicações, como as assistentes IA, que utilizam destes modelos estão adentrando na sociedade na última década (DSA, 2022).

2.2.3 Large Language Model

Os *Large Language Models* (LLMs) são modelos de linguagem de grande escala que impulsionaram a capacidade das máquinas no processamento de linguagem natural. Originados do Aprendizado Profundo, esses modelos são uma técnica avançada, que tem como núcleo os modelos baseados na arquitetura *transformer* e fundamentados nas redes neurais (CHACKO; CHACKO, 2023).

Esses modelos são treinados com grandes conjuntos de dados textuais para aprender padrões e relações entre elementos na linguagem. Desta forma os LLMs são capazes de dado um texto de entrada, também chamado de *prompt*, predizer o que seguirá, chamado de *completion*. Isso pode englobar tanto texto natural quanto também linguagens de programação, conforme o conjunto de dados utilizados para o treinamento do modelo

[2] https://openai.com/
[3] https://bard.google.com/

(SCHäFER et al., 2023). Os modelos são capazes de "compreender"a semântica da linguagem em questão com base na probabilidade do relacionamento de uma palavra com outras em determinado contexto.

Um dos diferenciais do LLM está no fato de que ele está numa categoria de modelos de grande escala que são pré-treinados com conjuntos massivos de dados, sendo estes dados não rotulados externamente, mas sim pelo próprio modelo (YUAN et al., 2023). Isso permite que o modelo aprenda representações ricas e generalizadas da linguagem a partir de dados de texto brutos. Além disso, os modelos LLMs tem sido otimizados a partir da mescla de técnicas, aplicando ao modelo a técnica de Aprendizado por Reforço (YUAN et al., 2023). Um exemplo é o ChatGPT, uma ferramenta proprietária da OpenAI, que visando optimizar o GPT, aplicou o aprendizado por reforço a partir do feedback humano, o que possibilitou mensurar a qualidade das respostas geradas pela ferramenta, melhorando a capacidade do modelo de fornecer respostas mais precisas e contextualmente adequadas (PHAN, 2023).

Atualmente, os LLMs têm a capacidade de realizar uma variedade de tarefas, como tradução de idiomas, análise de sentimentos, interações humanas por meio de *chatbots*, até análise e geração de código no contexto do desenvolvimento de software (PEREIRA, 2023).

2.2.4 Ferramentas LLM

A disseminação dos LLMs foi impulsionada pelo surgimento de várias ferramentas baseadas nesses modelos, com destaque para o BERT (*Bidirectional Encoder Representations from Transformers*) e o GPT (*Generative Pre-Trained Transformer*) (YUAN et al., 2023). O grande expoente de popularidade do GPT é o ChatGPT, um *chatbot* desenvolvida pela OpenAI e disponibilizada ao público de maneira gratuita no ano de 2022, que aprimorou o GPT com a utilização de técnicas de Aprendizado por Reforço (PHAN, 2023). Esse aprimoramento por Aprendizado por Reforço, possibilitou uma melhor generalização e maior coerência nas interações.

Os modelos de linguagem, como o GPT, recebem um *prompt* como entrada de texto para iniciar a geração ou compreensão de texto. O *prompt* tem limitações, já que o GPT opera com base em *tokens*, unidades básicas de informação que podem ser palavras, caracteres ou partes de palavras. Há uma capacidade máxima de processamento de tokens em uma sequência tanto no *prompt* quanto na resposta do modelo. Ultrapassar esse limite pode comprometer a qualidade ou a integridade da resposta e até mesmo não processar ou gerar o texto adequadamente (OpenAI. . . ,). No caso do GPT versão 3.5 este limite é de 4096 *tokens*.

Além de suas aplicações em diálogos interativos, o GPT apresenta potencial para

aplicações em áreas da engenharia de software. Estudos recentes, como o de Yuan et al. (2023), exploram a capacidade do ChatGPT de gerar testes unitários. A perspectiva de Schäfer et al. (2023), que se alinham com os estudos de Yuan et al. (2023), sugere que as propriedades atuais de LLMs indicam a possibilidade de gerar testes naturais, ou seja, semelhantes aos gerados por desenvolvedores humanos, semelhanças que vão desde nomes de variáveis até asserções.

2.3 Teste de Software

Conforme definido por Myers (2011), o teste de software envolve a execução sistemática de um sistema ou de suas partes para identificar falhas, verificar se os requisitos estão sendo atendidos e assegurar que o software funcione corretamente em diferentes condições. O objetivo é validar e verificar que o sistema realiza as funções esperadas e que cumpre com os critérios de desempenho, segurança e usabilidade estabelecidos. A qualidade de software, por sua vez, refere-se à medida em que o software atende aos requisitos especificados e às expectativas dos usuários, incluindo aspectos como confiabilidade, eficiência e robustez. Portanto, o teste de software não só busca identificar e corrigir defeitos, mas também assegurar que o software entregue seja de alta qualidade e apto para seu propósito pretendido.

2.3.1 Caso de teste

Uma das atividades para a realização de testes é a elaboração de casos de teste. Um caso de teste é a definição de condições específicas, nas quais ao executar a aplicação, se espera que ela falhe. Uma característica importante dos casos de teste é que o testador deve saber a saída esperada ao se executar o que está sendo testado.

Idealmente, um sistema deve ter todas as possibilidades cobertas pelos casos de teste, porém isso é irreal, principalmente ao considerarmos a complexidade atual das aplicações (VERGILIO; MALDONADO; JINO, 2006). A atividade de criar casos de teste que alcançassem todas as possibilidades da aplicação seria exaustivo e aumentaria consideravelmente o custo e tempo de desenvolvimento. Ademais, a realização da atividade de teste não garante a ausência de falhas, apenas verifica a existência delas, dessa forma, testar exaustivamente não iria garantir um software livre falhas.

Para que um sistema seja considerado suficientemente testado é preciso estabelecer estratégias para adequação dos testes e criação de casos de teste da maneira sistemática e efetiva. A partir disto, surgem as técnicas de teste e os critérios de teste.

2.3.2 Técnicas de teste

As técnicas de teste são abordagens ou estratégias específicas utilizadas para projetar, planejar e executar testes de software a fim de identificar defeitos, avaliar a qualidade e garantir o adequado funcionamento de um sistema. As principais técnicas de teste são a técnica funcional e a técnica estrutural.

2.3.2.1 Técnica Funcional

A técnica de teste funcional é uma abordagem usada para projetar casos de teste, nos quais o programa é considerado uma caixa-preta, ou seja, nessa técnica, os detalhes de implementação não são considerados, e os testes são realizados com o ponto de vista do usuário. Dessa forma, o teste funcional se concentra em verificar se o software atende aos requisitos funcionais estabelecidos, simulando as interações e comportamentos esperados pelos usuários (DELAMARO; MALDONADO; JINO, 2007).

Esta abordagem avalia as funcionalidades do software de forma independente de sua estrutura interna, garantindo que as entradas corretas gerem as saídas esperadas e que todas as funcionalidades sejam adequadamente testadas. Os casos de teste são elaborados com base nos requisitos, especificações e cenários de uso, permitindo validar se o software atende às necessidades e expectativas dos usuários finais.

Teoricamente, o teste caixa-preta poderia identificar todos os defeitos executando o programa com todas as entradas possíveis, o que seria um teste exaustivo. No entanto, a quantidade de combinações possíveis de entradas pode ser infinita, tornando esse tipo de teste inviável em termos de tempo e recursos.Além disso, o teste funcional se baseia em exceções externas, ignorando o comportamento interno do programa, limitando sua capacidade de identificar falhas estruturais e não garantindo a cobertura de partes críticas e essenciais do código (DELAMARO; MALDONADO; JINO, 2007).

2.3.2.2 Técnica Estrutural

A técnica de teste estrutural, também conhecida como teste caixa-branca, estabelece os requisitos com base na implementação do programa, ou seja, o código-fonte é de onde os casos de teste são originados. Ela se preocupa em como a aplicação se comporta interna e estruturalmente, sendo necessário ter conhecimento do código-fonte e sua estrutura lógica, por esse motivo é chamada de técnica estrutural (MYERS; SANDLER; BADGETT, 2011).

Esta técnica apresenta limitações e desvantagens devido à complexidade da atividade de teste de programa, tornando a automação do processo de validação de software um desafio. Sua principal limitação é a inexistência de um procedimento de teste de propósito geral que possa comprovar completamente a correção de um programa, além da

indecisão sobre a computação de funções idênticas entre programas distintos. Essa abordagem também enfrenta dificuldades em verificar a executabilidade de caminhos específicos e pode apresentar problemas como a falta de casos de teste para partes do programa sem caminhos correspondentes. Além disso, há o risco de coincidência de correção, onde um programa pode produzir resultados corretos para determinadas entradas, mas não para outras (DELAMARO; MALDONADO; JINO, 2007).

2.3.3 Critérios de teste

Os critérios de teste são abordagens utilizadas para determinar quais partes de um sistema devem ser testadas e como os casos de teste devem ser projetados. Esses critérios ajudam a orientar o processo de teste, definindo as metas e os requisitos que devem ser abordados durante a atividade de teste.

Os critérios de teste ajudam a garantir uma abordagem sistemática e abrangente, proporcionando uma base para a elaboração dos casos de teste e a avaliação da cobertura dos testes.

Por exemplo, criação de casos de teste para a técnica funcional é baseada apenas nas especificações da aplicação, como os requisitos, e para a geração de casos de teste dois critérios são comumente utilizados: particionamento e análise de valor limite. Enquanto, para a estratégia estrutural, caixa branca, onde a implementação é fundamental, a criação dos casos de teste é baseada no código-fonte e tem como ponto principal cobrir todas as estruturas condicionais e estruturais. E os dois principais critérios adotados para isso são: controle de fluxo e fluxo de dados (DURELLI et al., 2019).

2.3.3.1 Controle de Fluxo

O critério de teste de controle de fluxo é um dos critérios utilizados para determinar os casos de teste da técnica de teste estrutural. Este critério visa garantir que todas as decisões e ramificações lógicas do sistema sejam adequadamente testadas. Ele busca identificar e testar todos os caminhos lógicos, como fluxos de controle condicionais e laços de repetição, para garantir que todas as possibilidades sejam consideradas. Dessa forma, os casos de teste são projetados para exercitar diferentes combinações de condições e fluxos de controle, incluindo a execução de ramos verdadeiros e falsos, para verificar se o programa se comporta corretamente em todas as situações (DELAMARO; MALDONADO; JINO, 2007).

O controle de fluxo pode ser classificado como:

- Todos-Nós: requer que cada comando do programa seja executado pelo menos uma vez durante a execução do teste, garantindo que a cobertura do teste abranja todos os vértices do Grafo de Fluxo de Controle (GFC).

- Todas-Arestas: requer que durante o teste, cada desvio de fluxo de controle do programa seja percorrido pelo menos uma vez.
- Todos-Caminhos: requer que todos os caminhos do programa sejam executados.

2.3.3.2 Fluxo de dados

Os critérios baseados em fluxo de dados utilizam a análise de fluxo de dados como uma fonte de informação para derivar os requisitos de teste. Esses critérios têm em comum a necessidade de testar as interações que envolvem a definição de variáveis e suas referências posteriores.

Eles são especialmente adequados para identificar defeitos computacionais, uma vez que as dependências de dados são identificadas e segmentos funcionais específicos são exigidos como requisitos de teste.

Existem duas famílias de critérios baseados em fluxo de dados amplamente conhecidas: Rapps e Weyuker, Potenciais-Usos.

2.3.4 Cobertura de código

A cobertura de código é uma métrica usada para identificar partes do código que o conjunto de teste não alcança. Ela fornece uma medida quantitativa de quão bem os testes exercitam o código durante a execução, ajudando a identificar áreas do código que não estão sendo testadas adequadamente (Atlassian,).

Para considerar a cobertura dos testes de uma linha de código existem diversos critérios a serem definidos pelo desenvolvedor, conforme a eficiência e a abrangência que mais se encaixa na necessidade do programa em questão. Vale ressaltar que quanto mais complexo o critério maior o custo de tempo para desenvolver a suíte de teste. De acordo com Aniche (2022), os principais critérios de cobertura de código incluem:

- Cobertura de linha: este critério mede a porcentagem de linhas de código executadas durante a execução dos testes. Cada linha de código é marcada como coberta se for executada pelo menos uma vez durante os testes, o que não garante que todos os possíveis caminhos dentro de uma linha sejam executados.
- Cobertura de ramo: este critério mede a porcentagem de ramos, gerados pelas condicionais, que são exercidas pelo conjunto de testes, tanto no ramo verdadeiro quanto falso.
- Cobertura de condição: semelhante à cobertura de ramo, a cobertura de condição concentra-se na execução de todas as condições individuais no código, verificando se todas as expressões *booleanas* são avaliadas como verdadeiras e falsas.

- Cobertura de caminho: esse critério garante que todos os caminhos, incluindo declarações condicionais e loops, sejam exercitados pelos testes.

Cada critério de cobertura varre o código com um rigor diferente, o que torna comum a combinação de diferentes critérios em determinadas ocasiões, a fim de obter uma visão mais completa da cobertura.

3 Metodologia

3.1 Considerações Iniciais

Neste capítulo será abordada a classificação da pesquisa, considerando seu objetivo e sua abordagem. Além disso, será apresentado o plano metodológico que será utilizado durante o desenvolvimento desta pesquisa.

3.2 Classificação da Pesquisa

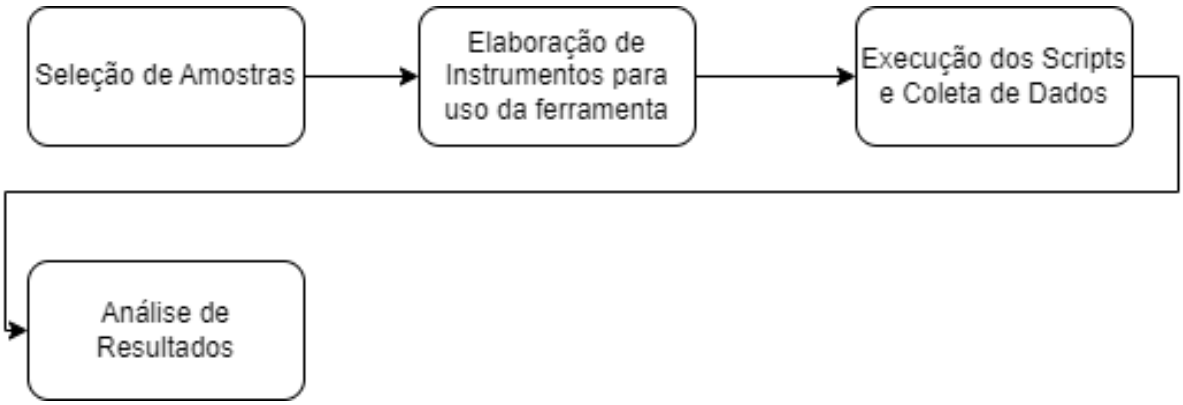
Com base nos objetivos definidos, este estudo pode ser caracterizado como uma pesquisa explicativa, conforme a classificação proposta por Gil (2017). Este trabalho engloba uma aplicação prática do GPT-3.5 Turbo para a geração de testes, seguida por uma avaliação e comparação entre os testes produzidos pela ferramenta de IA e os testes convencionais elaborados por desenvolvedores. A análise comparativa desses testes é conduzida por meio da aplicação de métricas provenientes da cobertura de código. A natureza explicativa deste estudo reside na busca por compreender profundamente as relações entre a geração automatizada de testes pela IA e os métodos tradicionais de criação de testes, a fim de elucidar as vantagens, limitações e impactos dessas abordagens no contexto do desenvolvimento de software.

Em relação à abordagem, este trabalho é classificado como quantitativo, dado as métricas obtidas a partir da cobertura de código e da execução dos casos de teste, o que envolve a coleta e análise de dados objetivos, como a quantidade de testes gerados, a eficácia dos testes gerados em capturar falhas, entre outros indicadores numéricos. A comparação entre os testes gerados pela ferramenta e pelos desenvolvedores também pode ser analisada quantitativamente.

3.3 Planejamento

A Figura 1 apresenta um diagrama de quatro etapas com o fluxo deste trabalho.

Figura 1 – Diagrama de planejamento da pesquisa.



Fonte: Autor (2023)

3.3.1 Seleção de Amostras

Para execução deste trabalho foi selecionada uma amostra de 200 classes, que contabilizou pelo menos 200 métodos, a partir do código-fonte de um projeto de código aberto hospedado no GitHub[1]. Esses números foram baseados em uma estimativa de viabilidade econômica, levando em conta limitações orçamentárias de $10 dólares, visto que a ferramenta LLM utilizada tem um custo de aproximadamente R$ 0,0049 a cada mil *tokens* de entrada e R$ 0,0098 a cada mil *tokens* de saída (PRICING, 2022).

A seleção destes métodos foi feita com base nos seguintes critérios:

- Repositório ativo: o repositório deve ter *commits* ou *pull requests* realizados no segundo semestre do ano de 2023, isso permite utilizar métodos de códigos de projetos que estão ativos no ciclo de vida do software.
- Repositório colaborativo: o repositório não deve ser pessoal, portanto os métodos serão selecionados de repositórios com pelo menos 3 colaboradores.
- Popularidade do repositório: o repositório deve ter mais de 500 estrelas ou mais de 500 *forks*.
- Linguagem de programação: a linguagem utilizada nos métodos selecionados deve ser Java, visto que o GPT na versão 3.5 foi treinado com dados até setembro de 2022 e Java é uma linguagem consolidada e amplamente conhecida até essa data, tornando-se uma escolha apropriada para garantir a compatibilidade com a ferramenta.
- Cobertura do método: os métodos selecionados devem ter ao menos 1 caso de teste que os exercitem.

[1] https://github.com/

Estes critérios foram baseados nos resultados descobertos pelos estudos de Kalliamvakou et al. (2014).

3.3.2 Elaboração de Instrumentos para Utilização de Ferramenta

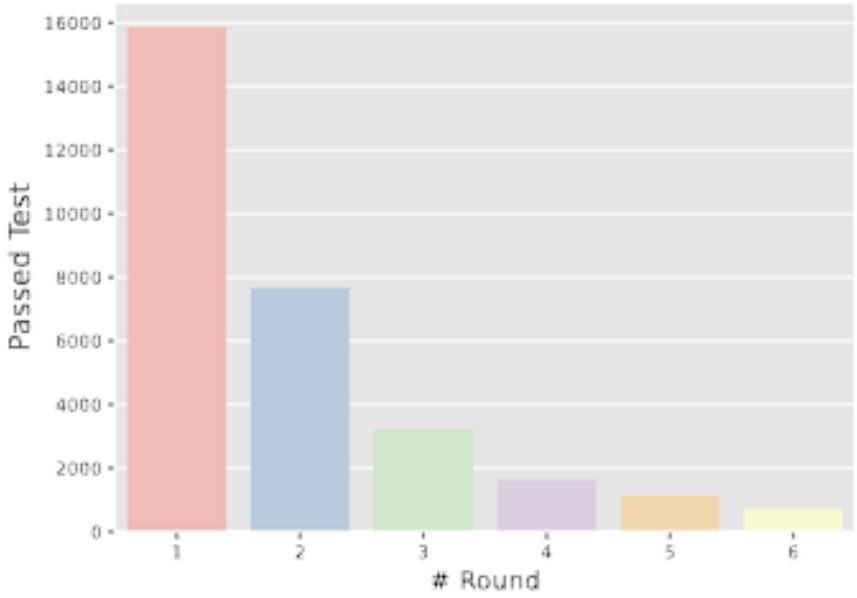
Para a realização deste trabalho foram realizados os mesmos passos em cada repositório e para cada método selecionado.

Primeiramente, foi criado um *script* para o pré-processamento dos métodos selecionados. Isso incluiu, a extração do método e a verificação da existência de casos de testes que testam o método.

Posteriormente, a fim de facilitar a geração de casos de teste, por parte do GPT-3.5 Turbo, para os métodos selecionados, foi criada uma estrutura básica de *prompt*, assim como um *script* para aplicar às centenas de métodos de maneira automatizada na ferramenta e validar a reposta gerada pela ferramenta. Esta validação foi feita com uma tentativa de compilação do teste gerado e sua execução. No caso de o teste falhar para compilar ou encontrar algum erro durante a execução, ele foi reparado ainda com o uso do *script*, que foi responsável por passar o teste pela ferramenta com um novo *prompt* com uma estrutura própria para o reparo do erro.

O processo de reparo foi realizado também utilizando o GPT-3.5 Turbo, a fim de minimizar o número de testes gerados pelo GPT-3.5 Turbo que não puderam ser executados e posteriormente analisados. Isso foi feito com base em um estudo similar que comprova que a geração de reparos reduzem os testes com erro de compilação e sintaxe em mais de 10% (XIE et al., 2023). Tendo isso em vista, o número de tentativas de reparo foi de três vezes, número definido com base no estudo realizado por Xie et al. (2023) que indica uma baixa taxa de resolução após a terceira tentativa. Isto pode ser visto na Figura 2.

Figura 2 – Distribuição de testes executados com sucesso em cada tentativa.



Fonte: Xie et al. (2023)

3.3.3 Execução dos Scripts e Coleta de Dados

Após a elaboração dos *scripts* e *prompts*, esses foram executados para gerar os casos de teste, os quais foram então executados para coletar métricas de cobertura e realizar análises comparativas.

Esta etapa compreendeu uma sequência de procedimentos: primeiro, a geração de testes, utilizando os instrumentos e o modelo definido; em seguida, a execução de conjuntos de testes, tanto os criados pelo modelo quanto os elaborados pelos desenvolvedores; posteriormente, a análise dos resultados dos testes realizados; e por fim, a avaliação da cobertura de código obtida pelos testes aplicados.

3.3.4 Análise de Resultados

Após todas as etapas, foi realizada uma análise de resultados entre os casos de teste criados por desenvolvedores humanos, presentes nos repositórios dos quais os métodos foram extraídos, e os casos de testes gerados a partir do GPT na versão 3.5. Esse comparativo teve como base os dados gerados a partir das métricas obtidas pelos scripts e a análise de cobertura que foi obtida durante a aplicação das etapas da pesquisa.

O comparativo foi realizado por meio da aplicação da estatística descritiva, que organiza e sintetiza os dados coletados, oferecendo uma melhor representação dos resultados obtidos. Para melhor compreensão e visualização, os dados foram estruturados por meio de tabelas e gráficos. Além do mais, foram utilizadas medidas descritivas, como média,

desvio padrão, variância, e valores mínimo e máximo, para resumir os dados numéricos e destacar aspectos relevantes dos conjuntos de dados analisados (capcs, 2019).

4 Preparação para Geração de Testes

4.1 Seleção de Amostras

A etapa de seleção de amostras foi dividida em 2 partes: a seleção de um projeto de código aberto e a escolha das amostras de métodos para a geração de testes unitários.

4.1.1 Seleção do Projeto

Para a realização deste trabalho foi necessário selecionar um projeto de código aberto a fim de obter a quantidade de amostras proposta. Conforme os critérios estabelecidos na subseção 3.3.1, o seguinte projeto foi selecionado:

- ZAP Proxy: O ZAP (Zed Attack Proxy) é uma ferramenta de segurança gratuita, desenvolvida para identificar automaticamente vulnerabilidades de segurança em aplicativos web durante as fases de desenvolvimento e teste. A escolha do ZAP Proxy se baseou em sua conformidade com todos os critérios estabelecidos, assim como em sua complexidade e relevância.

  O ZAP Proxy [1] é um projeto com considerável relevância e complexidade. Segundo a análise realizada pelo SonarCloud [2] no ZAP Proxy, o projeto possui mais de 170 mil linhas de código distribuídas em mais de 1.300 arquivos. A complexidade ciclomática do projeto, que mede o número de caminhos independentes através do código, é de 31.064 caminhos independentes. Esse número indica que o projeto possui uma quantidade significativa de condicionais e loops, o que torna o código muito complexo de entender, testar e manter. Além disso, a complexidade cognitiva, que avalia a dificuldade de entender e manter o código com base em sua estrutura e lógica, é de 23.916 (SOFTWARE, 2019).

  Essas métricas destacam a complexidade e a robustez do projeto ZAP Proxy, justificando a sua escolha para a execução do trabalho.

4.1.2 Seleção dos Métodos

A partir do projeto selecionado, foi realizada a escolha das amostras de métodos a serem submetidos à IA para a geração de casos de teste. Para proporcionar uma melhor contextualização do funcionamento do código dos métodos e considerar o orçamento disponível para a utilização da API do OpenAI com o modelo GPT-3.5 Turbo, optou-se

[1] https://github.com/zaproxy/zaproxy
[2] https://sonarcloud.io/projects

por uma seleção mais ampla baseada em classes. Foram selecionadas aleatoriamente 200 classes do projeto utilizando um script, o que significa que o número total de métodos pode variar conforme as classes selecionadas, mas será de no mínimo 200 métodos.

4.2 Elaboração de Instrumentos para Uso do Modelo

4.2.1 Scripts para Geração de Testes

A execução deste trabalho requer a submissão de diversas classes e seus respectivos métodos ao modelo LLM, GPT3.5 Turbo, juntamente com instruções e dicas, para criar um contexto que auxilie o modelo a tomar decisões embasadas no conhecimento obtido. Para isso, foram desenvolvidos *scripts* com o objetivo de analisar o projeto, extrair dados como nomes de classes, métodos, suítes de testes já existentes e realizar a seleção aleatória das amostras. Além disso, um *script* adicional foi criado para gerar os testes por meio da API da OpenAI e para executar as suítes de testes, permitindo a obtenção de métricas relativas aos testes gerados pela IA.

4.2.2 Modelos de prompt

Além dos *scripts*, foram desenvolvidos modelos de *prompt* para instruir a IA nas possíveis decisões a serem tomadas. Esses modelos foram inspirados pelo trabalho de Xie 2023 e seguem uma estrutura generalizada, mas com adaptações específicas ao projeto selecionado. Algumas dessas particularidades são ajustadas dinamicamente durante a execução dos *scripts*, enquanto outras são personalizadas com base no conhecimento do usuário sobre o projeto.

Os modelos se separam em modelo de geração de testes e modelo de reparo de testes. A seguir são apresentados os modelo de geração e reparo, para o projeto ZAP Proxy, e suas respectivas explicações:

- Modelo de geração

```
Generate unit test cases for the following method(s) in JUnit 5:
- Method name(s): METHOD_NAME
- Class name: CLASS_NAME
- Class path: CLASS_PATH
- Test path: TEST_PATH
- The class is written in Java.
- Create the unit tests using the following libraries: org.hamcrest and
                                                    org.junit.jupiter.api.Test.
```

```
Follow these instructions carefully:
1. The test class name should be TEST_NAME.
2. Use mocks, stubs, dummies, fakes, or spies when necessary to isolate the
unit under test from its dependencies, control the test environment, and
avoid unwanted side effects.
3. Implement the test logic for each method, including setup, execution,
and verification of results.
4. Use assertions to verify that the method behaves as expected.
5. Include the correct package declaration at the top of the file.
6. Ensure that all necessary imports for classes used in the tests are
included.
7. Each test case should be clearly named and include specific logic to test
different scenarios and edge cases.
8. Avoid generating test methods with placeholder comments.
9. Make sure each test has assertion.
10. Respond with a java test suite for the class

Make sure the tests are complete and avoid compile errors.
Class Code:
```

- Modelo de reparo

```
Instructions for correction the erro of test suite:
1. Review the provided test suite written in Java, using JUnit 5.
2. Identify the cause of the error based on the error output.
3. Make sure all imports are done correctly and the package is declared
correctly
4. If erro is in construct of another class, create mocks, stubs, dummies,
fakes, or spies and add missing dependencies to ensure the
tests run correctly.
5. If some test cases in the suite fail, just remove them from the suite
6. Ensure that the test suite code avoids using wildcard imports
(static or non-static).
7. Return only the corrected and updated test suite code.
8. If any method is deprecated, add the annotation
@SuppressWarnings("deprecation")
9. Respond with a java test suite for the class

Below is the code of test suite. When executing the test suite, the
following error output occurred.
```

```
Test suite:
TEST_CODE

Error output when running the test suite:
ERROR_MESSAGE
```

O modelo de geração está escrito na língua inglesa devido ao melhor suporte e treinamento do modelo LLM para essa língua, o que gera resultados mais precisos e eficazes. O modelo de geração é utilizado como base para o *prompt* de criação da suíte de testes, portanto ele especifica os métodos alvo e detalhes da classe, e instrui a utilização das bibliotecas para contextualização da IA. Ademais, o modelo de geração tem 10 instruções diretas e detalhadas para assegurar que cada aspecto do processo de teste seja abordado.

Primeiramente, a classe de teste gerada deve ter um nome específico. O modelo também recomenda o uso de estratégias de dublês de teste, como *mocks*, *stubs*, *dummies*, *fakes* ou *spies*, quando necessário para isolar a unidade em teste, controlar o ambiente de teste e evitar efeitos colaterais indesejados.

O modelo de geração instrui que a lógica de teste para cada método seja implementada utilizando assertivas e inclua a configuração inicial, execução do método e verificação dos resultados, instruindo assim a IA a gerar casos de teste completos e funcionais. Além disso, para evitar erros de compilação, o modelo instrui a realizar as importações necessárias e a responder com uma classe Java contendo uma suíte de testes.

O modelo de reparo é similar ao modelo de geração, com instruções que seguem a mesma lógica, mas se diferencia por enviar ao GPT as instruções de reparo, o teste gerado pela ferramenta e o erro obtido durante a execução do teste gerado, propiciando a contextualização necessária para que o GPT possa tentar realizar correções no código de teste.

Os *scripts* desenvolvidos para a análise do projeto e a geração de testes, bem como os modelos de *prompt* utilizados e *scripts* de suporte, estão hospedados no repositório GitHub[3]. Neste repositório, está disponível o código-fonte completo, instruções detalhadas sobre como utilizar os *scripts* e informações sobre os modelos de *prompt* para criar e ajustar os testes.

[3] https://github.com/lucasgbezerra/TCC

5 Execução e Coleta de Dados

5.1 Definição dos parâmetros do modelo LLM

Para a geração dos testes, foi utilizada a API da OpenAI com o modelo GPT-3.5 Turbo, na versão gpt-3.5-turbo-0125. A API permite a utilização de parâmetros para controlar diversos aspectos da geração de texto pelo modelo, como a aleatoriedade das respostas, a probabilidade cumulativa dos *tokens* escolhidos pela IA, entre outros.

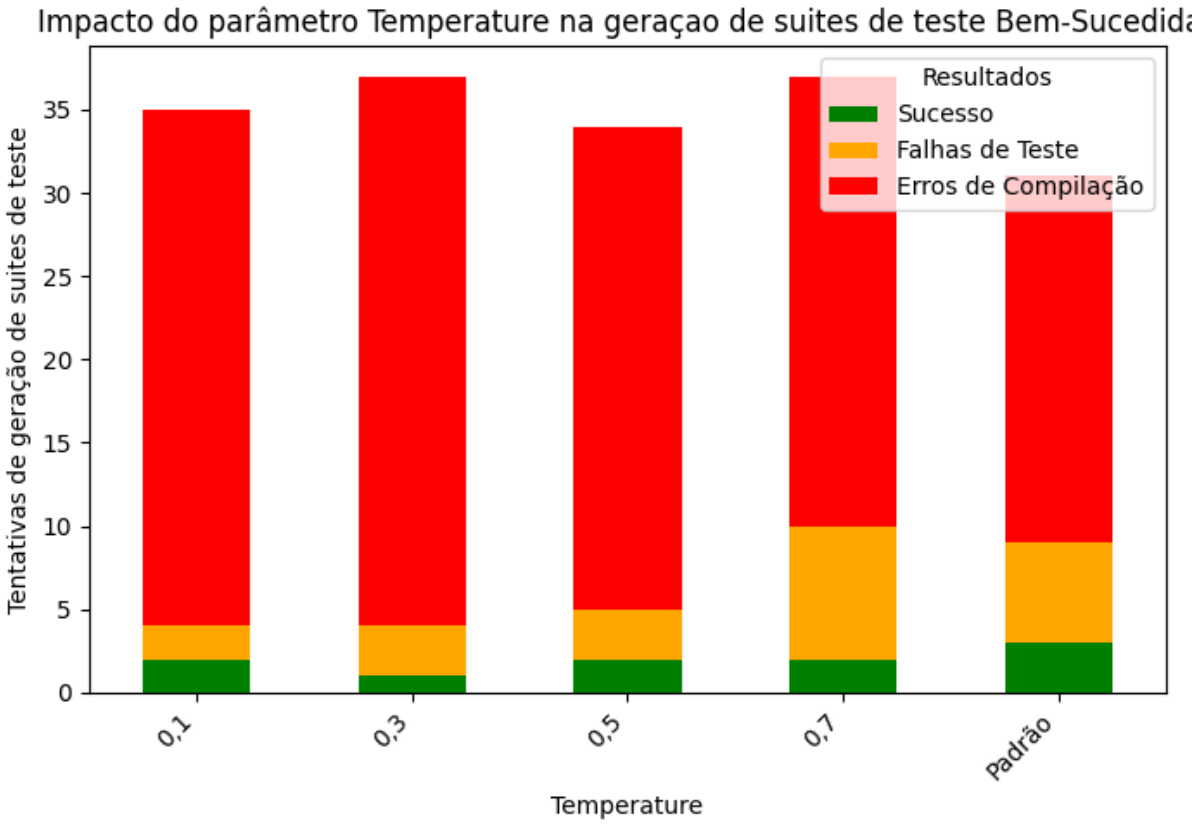
Com o objetivo de realizar uma escolha fundamentada dos parâmetros, foi conduzido um teste de variáveis com um pequeno conjunto amostral de 10 classes, totalizando 156 métodos, conforme detalhado na Tabela 1.

Tabela 1 – Resultados das Suítes de Teste com parâmetro *Temperature* Padrão

Nome Classe	**Geração**	**Reparo 1**	**Reparo 2**	**Reparo 3**
HttpResponseBody	Sucesso	-	-	-
SiteNodeStringComparator	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
ApiResponseElement	Falhas de teste	Falhas de teste	Falhas de teste	Erros de compilação
ProxyThread	Sucesso	-	-	-
Spider	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
PopupFlagCustomPageIndicatorMenu	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
ExtensionLoader	Erros de compilação	Falhas de teste	Falhas de teste	Falhas de teste
EncodingUtils	Sucesso	-	-	-
HttpPanelParamTableModel	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
UrlPatternScanFilter	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação

O processo de geração dos casos de teste envolveu a alteração do parâmetro *temperature* do GPT, que é importante para a criatividade na geração dos testes, pois controla a aleatoriedade das respostas. Foram realizadas tentativas utilizando o parâmetro padrão da API, bem como variando-o de 0,1 até 0,7, com um incremento de 0,2. Os resultados da execução das suítes de testes geradas com cada variação do parâmetro estão ilustrados na Figura 3.

Figura 3 – Impacto do parâmetro Temperature na geração de suítes de teste Bem-Sucedidas



Fonte: Autor (2024)

Com base na observação do gráfico da Figura 3 e da Tabela 1 é possível observar que a alteração do parâmetro não teve grande influência no resultado final. Todavia, é notável que a proporção de suítes de testes bem sucedidas em relação à quantidade de tentativas é maior quando o parâmetro *temperature* está no seu valor padrão da API. Além do mais, ao considerar os resultados em que os testes falharam — ou seja, suítes geradas nas quais nem todos os casos de teste passaram — a proporção de sucesso é ainda mais significativa, com aproximadamente 30% das tentativas apresentando resultados satisfatórios.

Portanto, é possível concluir que utilizar o parâmetro definido por padrão pela API do OpenAI para utilização do modelo GPT3.5 Turbo é o mais indicado para a realização do trabalho.

5.2 Execução do Script no Projeto

Essa etapa tem como objetivo a execução dos *scripts* responsáveis por analisar o projeto e obter dados necessários para a atividade de geração de casos de testes unitários e submeter esses dados, juntamente com as classes e métodos identificados, à API da OpenAI que utilizando o modelo GPT-3.5 Turbo. Após a submissão e geração dos testes

pela IA, as suítes de teste geradas são executadas, e os resultados dessas execuções são devidamente armazenados para análise posterior.

Os resultados incluem a Tabela 2 que se refere a um resumo da tabela gerada com os dados obtidos através da execução dos *scripts* e pode ser encontrada completa no repositório Github[1].

Nome Classe	**Casos de Teste**	**Casos com falha**	**Geração**	**Reparo 1**	**Reparo 2**	**Reparo 3**
StructuralNodeModifier	1	0	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
MethodScanFilter	0	0	Falhas nos casos de testes	Erros de compilação	Erros de compilação	Erros de compilação
ZapNumberSpinner	9	0	Sucesso			
SummaryAndConfigPanel	0	0	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
SocksProxy	7	0	Erros de compilação	Erros de compilação	Sucesso	
NodeJSAPIGenerator	0	0	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
OptionsProxiesTableModel	8	3	Erros de compilação	Erros de compilação	Erros de compilação	Falhas nos casos de testes
ZapTextField	0	0	Erros de compilação	Erros de compilação	Erros de compilação	Erros de compilação
MultipartFormParameter	6	0	Sucesso			

Tabela 2 – Resultados de Geração e Execução de Testes Unitários

5.3 Coleta de Indicadores Preliminares

Para a análise comparativa foi utilizado o SonarCloud, uma ferramenta de análise estática de código, que a partir do código fonte gera indicadores sobre o projeto. O projeto ZAP Proxy disponibiliza o SonarCloud que apresenta os indicadores coletados resumidos na Tabela 3.

[1] https://github.com/lucasgbezerra/TCC

Tabela 3 – Indicadores Preliminares

Descrição	**Valor**
Número de classes	1.620
Número de funções	15.559
Número total de testes unitários	2.784
Teste unitários bem-sucedidos (%)	100%
Cobertura de testes (%)	17,6 %
Cobertura de linha (%)	17,8 %
Cobertura de condição (%)	16,9 %
Complexidade Cognitiva	23.916
Complexidade Ciclomática	31.064

5.4 Coleta de Indicadores Pós-Geração de testes

Após a execução dos *scripts* e a consequente geração dos casos de teste unitário, os testes o SonarQube foi executado localmente no projeto e gerou os indicadores apresentados na Tabela 4. A tabela apresenta variações em todos os indicadores, vale ressaltar que foi submetido ao modelo LLM, apenas, um conjunto de 200 classes aleatórias.

Tabela 4 – Indicadores Pós-Geração de Casos de Teste

Descrição	**Valor**
Número de classes	1.620
Número de funções	15.559
Número total de testes unitários	2.944
Teste unitários bem-sucedidos (%)	95,9%
Cobertura de testes (%)	17,8 %
Cobertura de linha (%)	18,2 %
Cobertura de condição (%)	16,8 %
Complexidade Cognitiva	23.916
Complexidade Ciclomática	31.064

6 Análise de Resultados

Esta seção apresenta a análise dos resultados obtidos a partir da execução dos testes e das métricas de desempenho geradas pelo SonarQube. A análise é dividida em duas partes principais: a primeira examina os indicadores do repositório antes da intervenção da IA, enquanto a segunda avalia os resultados considerando os testes unitários gerados pelo modelo LLM. Por fim, foi realizada a consolidação da análise dos resultados, a partir da comparação entre as duas análises.

6.1 Análise preliminar

O ZAP Proxy, inicialmente, revela algumas métricas importantes sobre a qualidade e a extensão dos testes realizados, como visto na Tabela 3. Com 1.620 classes e um total de 2.784 testes unitários, o projeto possui uma base de código extensa e um número considerável de testes. No entanto, a cobertura de testes é relativamente baixa, com apenas 17,6% do código sendo coberto pelos testes, o que indica que muitos aspectos do código não estão sendo testados de forma adequada. A cobertura de linha é semelhante, em torno de 17,8%, indicando que a proporção de linhas de código executadas durante os testes está alinhada com a cobertura geral dos testes.

Além disso, a cobertura de condição está em 16,9%, o que sugere que apenas uma fração das condições lógicas presentes no código está sendo testada. A grande quantidade de funções no projeto pode tornar desafiadora a criação de um número suficiente de testes para melhorar os indicadores de cobertura, devido ao tempo e custo envolvidos nessa tarefa.

6.2 Análise Pós-Intervenção de IA

A análise inicial feita sobre os dados da Tabela 3 a Tabela e 4 já apresentam alterações positivas e negativas para a utilização de casos de teste unitários gerados pelo modelo LLM. Pode ser observado, um aumento no número de testes e na cobertura de testes e cobertura por linha em relação as duas tabelas. Entretanto, ocorreu uma diminuição de cobertura de condição, o que pode indicar que os testes gerados pelo modelo GPT3.5 Turbo, não tem o foco no critério de testes de cobertura de condições, mas sim na cobertura do máximo de linhas.

De igual modo, a Tabela 4 revela uma redução na porcentagem de testes unitários bem-sucedidos, que caiu de 100% antes da geração dos novos casos de teste para 95,9%

após a inclusão dos testes gerados pela IA. Essa diminuição indica que a IA produziu casos de teste que falharam, o que pode ser atribuído tanto a testes mal formulados pelo modelo quanto a funções do projeto que podem não estar funcionando adequadamente.

Tabela 5 – Ocorrência de Resultado por Fase da Suíte de Testes

Resultados	**Geração**	**Reparo 1**	**Reparo 2**	**Reparo 3**
Erro de compilação	168	147	137	132
Sucesso	16	12	8	3
Erro de configuração de teste	0	0	1	1
Falha de caso de teste	14	23	24	26

A redução dos testes unitários bem-sucedidos pode ser mais bem compreendida ao analisar a Tabela 2, que apresenta um resumo dos dados obtidos pelos *scripts*. As colunas da Tabela 2 representam cada submissão ao GPT, e a cada nova submissão são retiradas do próximo ciclo de submissão as suítes de testes bem-sucedidas, ou seja, apenas os testes com erros e falhas no ciclo anterior são reparados. Com base nesses dados, foi gerada a Tabela 5, que detalha a ocorrência de diferentes resultados de testes ao longo das fases de geração e reparo. Observa-se que, em cada fase, um número menor de suítes de testes apresenta erros de compilação, caindo de 168 suítes na fase de geração para 132 na última fase de reparo. Essa redução de aproximadamente 17% reflete a capacidade do modelo LLM de corrigir erros de compilação, embora de forma moderada.

Por outro lado, o modelo enfrenta dificuldades crescentes em transformar erros e falhas em suítes de testes bem-sucedidas. Um indicativo negativo, visto que o objetivo dos reparos é corrigir erros para que o teste possa ser executado com sucesso. Ademais, a quantidade de testes com falhas aumenta a cada fase de reparo, sugerindo que o modelo tende a gerar mais testes mal formulados conforme tenta corrigir os erros iniciais.

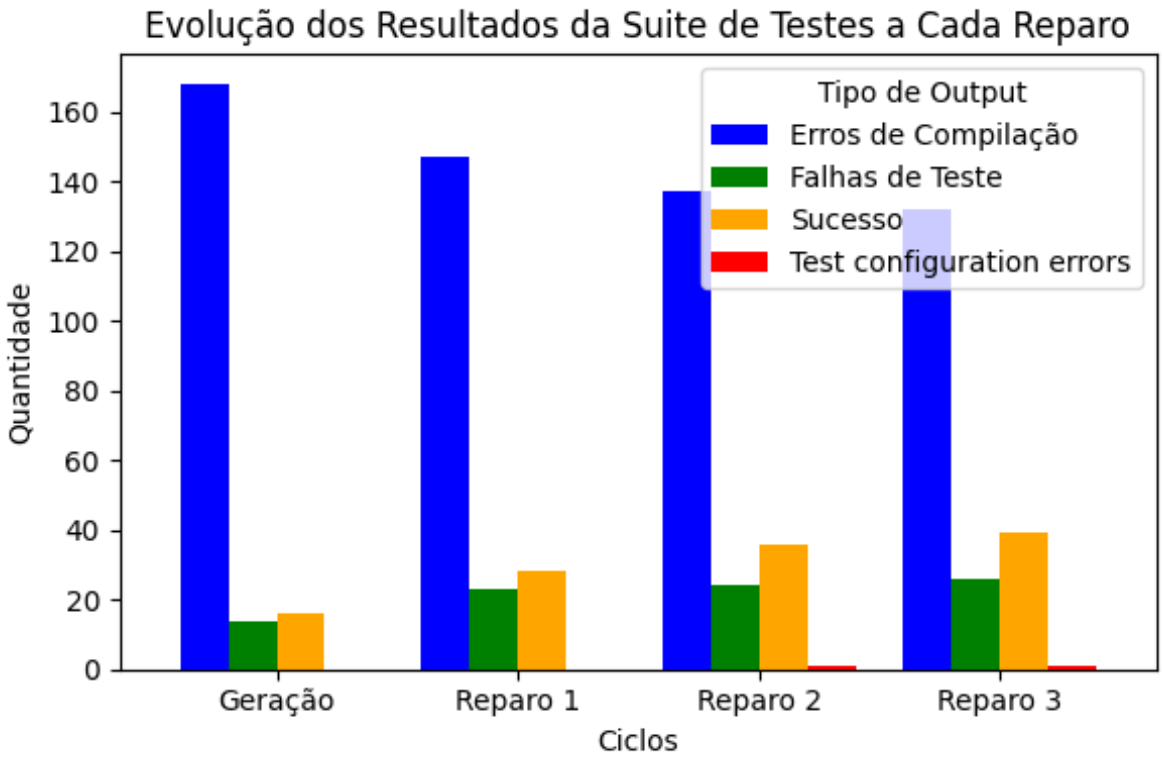


Figura 4 – Evolução dos Resultados da Suíte de Testes a Cada Reparos

Fonte: Autor (2024)

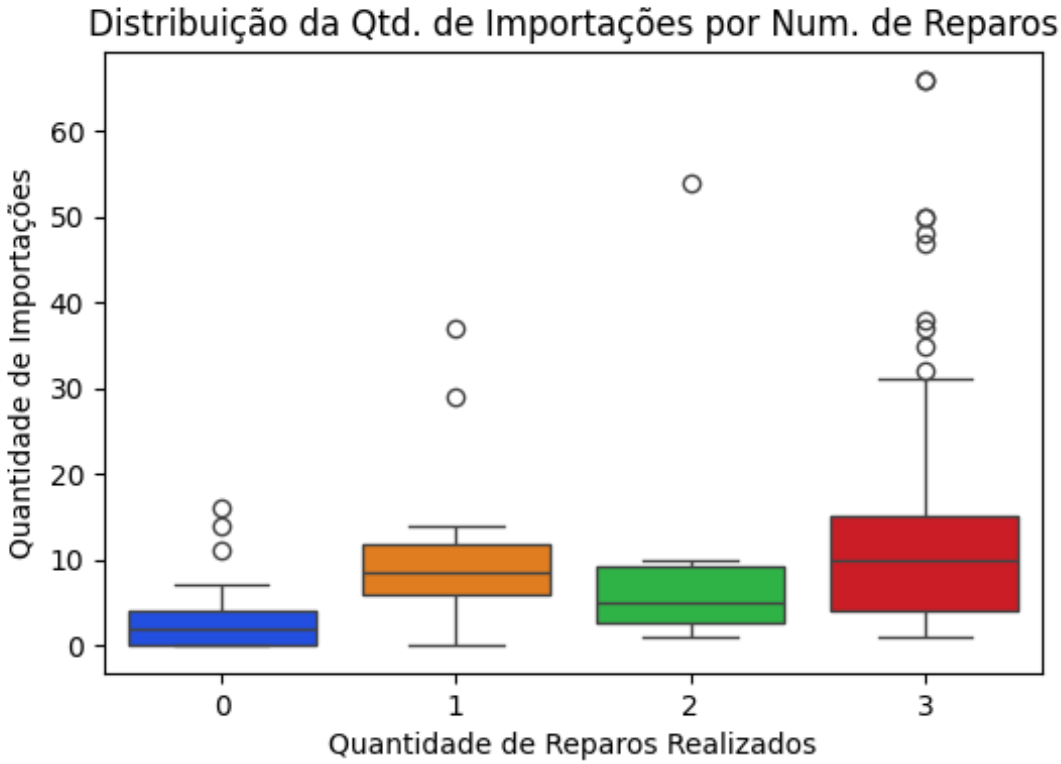
Sob uma perspectiva mais otimista, é possível considerar que uma suíte de testes que contenha alguns casos que falham, enquanto outros passam, pode ser vista como um sucesso parcial do modelo GPT-3.5 Turbo. Isso ocorre porque o modelo consegue testar alguns métodos e aumentar a cobertura da classe, ainda que não de forma completa. Como ilustrado na Figura 4, a quantidade de suítes de testes bem-sucedidas e aquelas com falhas nos casos de teste tende a aumentar a cada fase de reparo, enquanto as suítes com erros de compilação diminuem. Esse padrão sugere que, durante os reparos, o modelo consegue ampliar a cobertura dos testes, mesmo que a porcentagem de testes unitários bem-sucedidos diminua.

A alta taxa de testes gerados com erro de compilação pode ser atribuída a vários fatores, alguns são tratados no *prompt* como: a ausência de dependências, o alto acoplamento, que dificulta o modelo ao exigir a análise de várias classes e métodos para entender o contexto completo, e a presença de métodos depreciados. Embora existam muitas possibilidades para esses resultados, a baixa taxa de reparo sugere que os erros são complexos, o que reforça a ideia de que o alto acoplamento pode ser um dos principais fatores que contribuem para a dificuldade em gerar testes válidos.

Além disso, os elevados indicadores de complexidade cognitiva e ciclomática, que

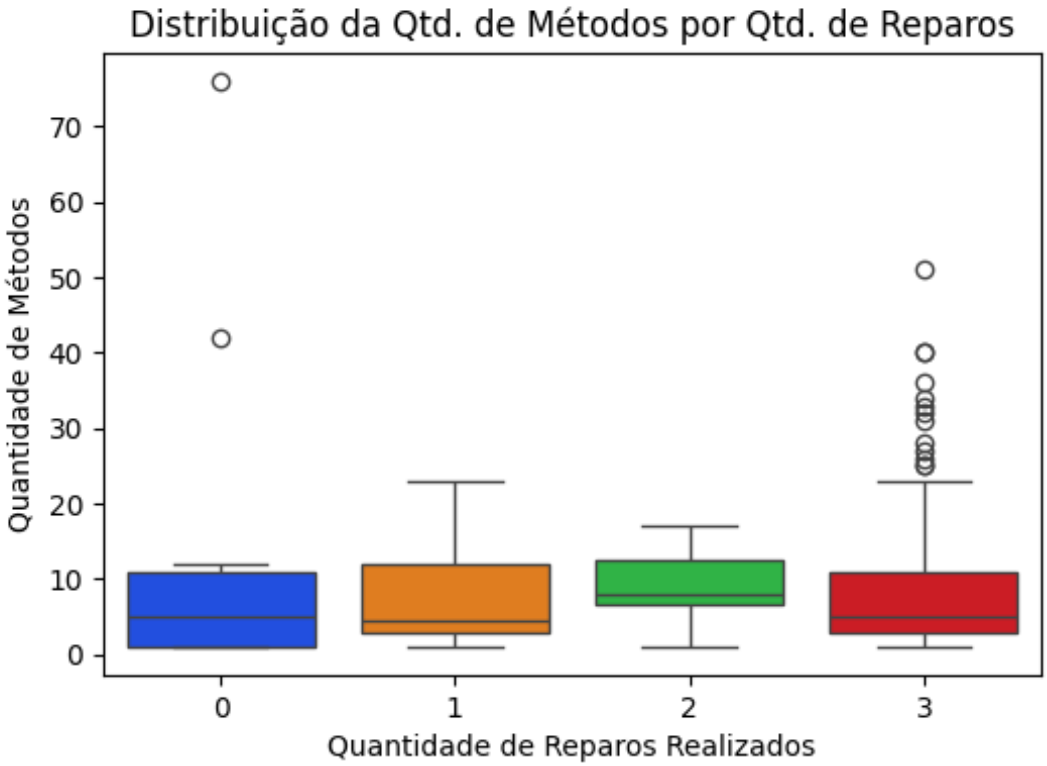
são de 23.916 e 31.064, respectivamente, conforme apresentado na Tabela 3 , podem estar impactando negativamente a eficácia do reparo dos casos de teste. A alta complexidade cognitiva sugere que o código do projeto contém muitos níveis de aninhamento, uma quantidade excessiva de blocos condicionais e funções com múltiplas atribuições, o que dificulta a compreensão do fluxo lógico. Por sua vez, a alta complexidade ciclomática indica uma quantidade significativa de ramificações resultantes de blocos condicionais, refletindo a complexidade das lógicas e decisões presentes no código. Essas características tornam o código mais difícil de entender e, consequentemente, de testar, o que pode dificultar a geração de testes pelo GPT-3.5 Turbo.

Figura 5 – Distribuição do Número de Importações em Relação aos Reparos



Fonte: Autor (2024)

Figura 6 – Distribuição do do Número de Métodos em Relação aos Reparos



Fonte: Autor (2024)

O diagrama de caixa da Figura 5 apresenta a seguinte tendência: classes com mais importações, ou dependências, necessitam de mais reparos. O terceiro reparo se destaca, com uma quantidade significativamente maior de dependências nas classes e vários pontos fora do padrão, (*outliers*), acima do limite superior. Isso indica uma grande dispersão na quantidade de importações das classes do terceiro reparo. Um comportamento similar pode ser encontrada na Figura 6, que apresenta a distribuição do número de métodos em relação aos reparos. Esses padrões sugerem que, com o avanço dos reparos dos testes gerados, o GPT não consegue reparar as classes com mais dependências e métodos, reflexo da dificuldade de lidar com uma maior complexidade.

6.3 Consolidação da Análise de Resultados

A partir da análise dos gráficos e das tabelas apresentadas neste capítulo, observa-se que, apesar da possibilidade de geração de casos de testes unitários utilizando o modelo LLM GPT-3.5 Turbo, existem limitações significativas. Um projeto complexo como o ZAP Proxy é um desafio para o modelo, devido à limitação de *tokens* imposta pela API e às restrições orçamentárias do trabalho. Isso é refletido na quantidade de 132 suítes de testes, que representa mais de 60% das suítes geradas, que apresentaram erros de compilação

mesmo após três tentativas de reparo. Não obstante, o modelo enfrentou dificuldades em lidar com altos níveis de acoplamento e as altas complexidades ciclomática e cognitiva.

É importante destacar que, apesar dessas fortes limitações, o uso de uma amostra de 200 classes, que representa 12,34% do total de classes do projeto, resultou na geração de suítes de testes para 75 dessas classes. Isso permitiu um aumento de 0,2% na cobertura de testes, demonstrando que, mesmo com desafios significativos, a aplicação do modelo teve um impacto positivo mínimo na cobertura de testes do projeto.

Desta forma, o modelo demonstrou a capacidade de gerar testes unitários, mas apresentou dificuldades em termos de assertividade e eficácia. Embora tenha gerado mais suítes de testes do que as existentes no projeto, o aumento na cobertura foi irrisório, com uma queda considerável na porcentagem de testes bem-sucedidos. Isso sugere que, apesar do potencial do GPT, a geração automatizada de testes ainda enfrenta limitações em um cenário de projeto.

Esses dados indicam um futuro promissor para a aplicação de modelos de linguagem na automação de testes; no entanto, a alta taxa de erros nos testes gerados e o grande número de testes unitários gerados com baixo impacto nos índices de cobertura demonstram a ineficácia da completa automatização dessa atividade. Portanto, a geração de testes assistida por IA pode ser uma opção mais viável no momento.

7 Conclusão e Trabalhos Futuros

Neste trabalho investigou-se a capacidade de utilizar modelos LLM, como o GPT3.5 Turbo, para a geração de casos de testes unitários. Para a conhecer a capacidade do GPT3.5 Turbo estabeleceram-se três objetivos específicos.

O primeiro objetivo foi de gerar casos de teste unitários para mais de 200 métodos, utilizando o modelo LLM GPT-3.5 Turbo, extraídos por meio de scripts de automação do repositório aberto no GitHub. Após a geração, os testes foram executados e corrigidos com o uso de *script* e o uso da submissão para a API do modelo, a fim de reparar erros e falhas identificados pelo modelo. Por fim, foi realizada uma análise dos resultados e métricas geradas usando estatística descritiva para avaliar e comparar a eficácia dos casos de teste criados pelo modelo LLM com aqueles desenvolvidos manualmente pelos desenvolvedores.

Portanto, enquanto o modelo demonstrou capacidade para gerar testes unitários, sua eficácia e assertividade foram limitadas, com um aumento irrisório na cobertura e uma queda significativa na taxa de testes bem-sucedidos. Esses resultados sugerem que, apesar do potencial oferecido pelo GPT-3.5 Turbo, a automação completa da geração de testes ainda não é viável em contextos de projetos complexos. Assim, a geração de testes assistida por IA se apresenta como uma alternativa no momento, permitindo que os desenvolvedores integrem ferramentas automatizadas no processo de geração de teste.

Em trabalhos futuros, pode-se explorar o uso de ferramentas que proporcionem uma contextualização mais abrangente do projeto e enfrentem menos limitações de API. Isso permitiria uma análise completa de todas as classes do projeto, incluindo o armazenamento de informações sobre dependências e acoplamentos. Ademais, a utilização de um modelo pré-treinado, que possa ser ajustado com base no código fonte específico do projeto, poderia aprimorar a geração de testes unitários, alcançando resultados mais promissores.

Referências

ANICHE, M. *Effective software testing: a developer's guide*. Shelter Island: Manning, 2022. ISBN 9781633439931. Citado na página 23.

Atlassian. *What is Code Coverage?* Disponível em: <https://www.atlassian.com/continuous-delivery/software-testing/code-coverage>. Citado na página 23.

BORDIGNON, A. C. d. A. A systematic literature review on natural language processing in business process identification and modeling. 2016. Disponível em: <https://lume.ufrgs.br/handle/10183/150923>. Citado 2 vezes nas páginas 15 e 16.

capcs. *Estatística Descritiva*. 2019. Disponível em: <http://www.capcs.uerj.br/estatistica-descritiva/>. Citado na página 29.

CHACKO, N.; CHACKO, V. Paradigm shift presented by Large Language Models (LLM) in Deep Learning. v. 40, 2023. Citado na página 18.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*. 1ª edição. ed. [S.l.]: Elsevier, 2007. ISBN 978-85-352-2634-8. Citado 2 vezes nas páginas 21 e 22.

DSA, E. *Capítulo 79 - Conhecendo o Modelo GPT-3 (Generative Pre-trained Transformer)*. 2022. Disponível em: <https://www.deeplearningbook.com.br/conhecendo-o-modelo-gpt-3-generative-pre-trained-transformer/>. Citado na página 18.

DURELLI, V. H. S. et al. Machine Learning Applied to Software Testing: A Systematic Mapping Study. *IEEE Transactions on Reliability*, v. 68, n. 3, p. 1189–1212, set. 2019. ISSN 1558-1721. Conference Name: IEEE Transactions on Reliability. Citado 3 vezes nas páginas 16, 17 e 22.

GIL, A. C. *Como elaborar projetos de pesquisa*. [S.l.]: Editora Atlas Ltda, 2017. ISBN 9788597012613. Citado 2 vezes nas páginas 12 e 25.

GitHub Copilot · Your AI pair programmer. Disponível em: <https://github.com/features/copilot>. Citado na página 16.

GéRON, A. *Mãos à obra: aprendizado de máquina com scikit-Learn & tensorFlow*. [S.l.]: Alta Books, 2021. ISBN 9788550815480. Citado na página 17.

HOURANI, H.; HAMMAD, A.; LAFI, M. The Impact of Artificial Intelligence on Software Testing. In: *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*. Amman, Jordan: IEEE, 2019. p. 565–570. ISBN 978-1-5386-7942-5. Disponível em: <https://ieeexplore.ieee.org/document/8717439/>. Citado na página 15.

INGRAM, J. *Why Software Developer Is the No. 1 Job of 2023*. 2023. Disponível em: <https://money.usnews.com/careers/articles/why-software-developer-is-the-no-1-job-of-2023>. Citado na página 10.

JADHAV, S. *Revolutionizing Software Engineering with LLMs*. 2024. Disponível em: <https://www.turing.com/blog/software-engineering-with-llms>. Citado na página 11.

KALLIAMVAKOU, E. et al. The promises and perils of mining GitHub. In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. New York, NY, USA: Association for Computing Machinery, 2014. (MSR 2014), p. 92–101. ISBN 9781450328630. Disponível em: <https://doi.org/10.1145/2597073.2597074>. Citado na página 27.

KANG, Y. et al. Natural language processing (NLP) in management research: A literature review. *Journal of Management Analytics*, v. 7, n. 2, p. 139–172, abr. 2020. ISSN 2327-0012. Disponível em: <https://doi.org/10.1080/23270012.2020.1756939>. Citado 2 vezes nas páginas 15 e 16.

KHALIQ, Z.; FAROOQ, S. U.; KHAN, D. A. *Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect*. arXiv, 2022. ArXiv:2201.05371 [cs]. Disponível em: <http://arxiv.org/abs/2201.05371>. Citado na página 14.

LANGENDOEN, D. Review of: Natural Language and Universal Grammar, by John Lyons. v. 69, p. 825–828, dez. 1993. Citado na página 15.

LUKASHIK, A. *Software Test Automation Statistics and Metrics for 2023*. 2022. Disponível em: <https://dogq.io/blog/test-automation-statistics-for-making-the-right-decisions/>. Citado na página 10.

MYERS, G.; SANDLER, C.; BADGETT, T. *The art of software testing*. 3rd. ed. [S.l.]: John Wiley & Sons, Incorporated, 2011. ISBN 978-1-118-13313-2. Citado 3 vezes nas páginas 10, 20 e 21.

OpenAI Platform. Disponível em: <https://platform.openai.com>. Citado na página 19.

PEREIRA, T. *O Que São Large Language Models (LLMs)?* 2023. Disponível em: <https://blog.dsacademy.com.br/o-que-sao-large-language-models-llms/>. Citado na página 19.

PHAN, T. L. L. *How ChatGPT is fine-tuned using Reinforcement Learning*. 2023. Disponível em: <https://dida.do/blog/chatgpt-reinforcement-learning>. Citado na página 19.

PRICING. 2022. Disponível em: <https://openai.com/pricing>. Citado na página 26.

RUSSELL, S. J.; NORVIG, P.; DAVIS, E. *Artificial intelligence: a modern approach*. 3rd ed. ed. Upper Saddle River: Prentice Hall, 2010. (Prentice Hall series in artificial intelligence). ISBN 9780136042594. Citado na página 10.

SCHäFER, M. et al. *An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation*. arXiv, 2023. ArXiv:2302.06527 [cs]. Disponível em: <http://arxiv.org/abs/2302.06527>. Citado 2 vezes nas páginas 19 e 20.

SOFTWARE, A. *Complexidade Cognitiva*. 2019. Disponível em: <https://artesoftware.com.br/2019/02/10/complexidade-cognitiva/>. Citado na página 30.

TOTVS, E. *Conheça 13 exemplos de aplicações da inteligência artificial nas empresas*. 2024. Disponível em: <https://www.totvs.com/blog/inovacoes/aplicacoes-da-inteligencia-artificial/>. Citado na página 11.

TURING, A. M. Computing Machinery and Intelligence. *Mind*, v. 59, n. 236, p. 433–460, 1950. ISSN 0026-4423. Disponível em: <https://www.jstor.org/stable/2251299>. Citado na página 14.

VERGILIO, S. R.; MALDONADO, J. C.; JINO, M. Infeasible Paths in the Context of Data Flow Based Testing Criteria: Identification, Classification and Prediction. jun. 2006. Citado na página 20.

Victory. *Using AI to Write Code: Pros and Cons*. 2023. Disponível em: <https://victorycto.com/using-ai-to-write-code-pros-and-cons/>. Citado na página 16.

XIE, Z. et al. *ChatUniTest: a ChatGPT-based automated unit test generation tool*. arXiv, 2023. ArXiv:2305.04764 [cs]. Disponível em: <http://arxiv.org/abs/2305.04764>. Citado 3 vezes nas páginas 27, 28 e 31.

YUAN, Z. et al. *No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation*. arXiv, 2023. ArXiv:2305.04207 [cs]. Disponível em: <http://arxiv.org/abs/2305.04207>. Citado 2 vezes nas páginas 19 e 20.

ZHANG, C.; LU, Y. Study on artificial intelligence: The state of the art and future prospects. *Journal of Industrial Information Integration*, v. 23, p. 100224, set. 2021. ISSN 2452-414X. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2452414X21000248>. Citado 2 vezes nas páginas 14 e 15.