UNIVERSIDADE FEDERAL DE SERGIPE

CENTRO DE CIÊNCIAS EXATAS E SUAS TECNOLOGIAS

DEPARTAMENTO DE COMPUTAÇÃO

BRENNO DE FARO VIEIRA

DAVI SOUZA FONTES SANTOS

JOSÉ FREIRE FALCÃO NETO

HUMBERTO DA CONCEIÇÃO JÚNIOR

NEWTON SOUZA SANTANA JÚNIOR

ATIVIDADE 2 -

Uso de LLMs (Transformers) para Apoiar Atividades de Teste de Software

“Using Large Language Models to Generate JUnit Tests: An Empirical Study”

São Cristóvão – Sergipe

2025

Sumário

1. Introdução

A criação de testes unitários configura-se como uma atividade fundamental no contexto do desenvolvimento de software, devendo estar intrinsecamente integrada ao processo de construção de sistemas. Considerando essa necessidade e os avanços recentes nas tecnologias baseadas em Modelos de Linguagem de Grande Escala (LLMs), este trabalho propõe a análise do artigo “Using Large Language Models to Generate JUnit Tests: An Empirical Study” (Siddiq et al., 2023), disponível em https://arxiv.org/abs/2305.00418.

Abaixo está o link do repositório github do projeto e vídeo tutorial respectivamente:

- https://github.com/brenofaro/Teste_Software_LLM_2025_vieira_brenno
- https://drive.google.com/file/d/1Zfn0MC5Uob16hmPKgTaqyoYMthS7GMv6/view?usp=sharing

2. Respostas

2.1. Descrição do problema

O problema de teste de software abordado centra-se na dificuldade dos desenvolvedores em gerar testes de unidade manualmente, especialmente para linguagens fortemente tipadas como Java.
Um dado importante que comprova a baixa adesão dos desenvolvedores à criação dos testes é um estudo que revelou que apenas 17% dos projetos no GitHub continham arquivos de teste.
Torna-se visível o apreço e ao mesmo tempo o desafio de usar Large Language Models (LLMs) para gerar testes de unidade em Java, destacando que esses modelos costumam ter desempenho inferior em linguagens fortemente tipadas.

2.2. Discussão da solução Proposta

A solução proposta foi usar Large Language Models (LLMs), baseados em arquitetura Transformer, na geração de testes de unidade em Java sem o ajuste fino para especializar o modelo. O estudo considerou 3 modelos: **Codex, GPT-3.5-Turbo e StarCoder**. Foram avaliados se podem gerar testes a partir de prompts contendo código, imports e documentação, seguindo um processo de:

1. **Criação de contexto e prompt:** definição do método a ser testado e instruções para gerar 10 casos de teste.

2. **Geração dos testes pelos LLMs**, seguida de pós-processamento com heurísticas para corrigir erros comuns e aumentar as taxas de compilação.

A avaliação considerou compilação, correção, cobertura de código e Test Smells, comparando com a ferramenta **EvoSuite**. Os modelos, de forma mais descrita, foram:

Codex: Utilizou a API da OpenAI e foi configurado de 2.000 a 4.000 tokens. A temperatura de resposta do modelo foi definida como zero a fim de produzir saídas mais determinísticas.

◦ **GPT-3.5-Turbo**: Também acessível via API da OpenAI. Foi instruído a gerar até 2.000 tokens, dedicando o restante dos seus 4.096 tokens de limite de entrada para o contexto. A temperatura foi definida como zero, e a função do sistema foi configurada como "Você é um assistente de codificação". Você gera apenas código-fonte".

◦ **StarCoder**: Utilizou o modelo StarCoderBase da biblioteca HuggingFace. Tinha uma janela de contexto de 8.000 tokens somando entrada e saída, com a saída limitada a 2.000 tokens para alinhar com os outros modelos de LLM. Os parâmetros de inferência foram os mesmos do modelo Codex.

Após a geração, os testes passaram por um **pós-processamento com heurísticas** para corrigir erros comuns ao código. Foi observado que os LLMs podiam gerar classes de teste incompletas, incluir explicações em linguagem natural, repetir o código da classe sob teste ou o prompt, alterar ou remover declarações de pacote ou gerar testes incompletos devido ao limite de tokens.

2.3. Descrição da Solução e Avaliação

Considerando dois benchmarks: HumanEval (160 problemas de programação com soluções de referência) e SF110 (47 projetos Java reais, 194 classes e 411 métodos testáveis).

A geração ocorreu em duas etapas: criação do prompt incluindo código da classe e imports do JUnit5 e produção dos testes, limitada pelo número de tokens suportado por cada modelo de LLM. Foram medidas quatro métricas principais:

- Compilação: Sem heurísticas, a taxa era muito baixa, especialmente no SF110 (2,7% a 12,7%). Com heurísticas automáticas, houve aumento médio de 41% no HumanEval, 100% no Codex 2K e 81% no SF110, mas persistiram erros semânticos como símbolos desconhecidos e pacotes incorretos.

- Correção: Um teste era correto se todos os métodos passassem. No HumanEval, o StarCoder obteve até 81% de testes corretos e o GPT-3.5-Turbo 52%, mas 92,3% tinham pelo menos um método válido. No SF110, nenhum modelo passou da casa dos 52%.

- Cobertura de código: No HumanEval, a cobertura de linha ficou entre 67% e 87,7%, e de branch entre 69,3% e 92,8%, abaixo dos testes manuais e do Evosuite que foram até 96,1%. No SF110, a cobertura foi inferior a 2%, muito menor que a do

Evosuite, 27,5% linha.

- Test Smells: Foram detectados ocasiões recorrentes como Magic Number Test, Lazy Test, Assertion Roulette, Eager Test entre outros.

Além disso, a pesquisa avaliou o impacto do contexto do prompt como a presença do JavaDoc, exemplos, implementação completa e etc. Os resultados variaram. Alguns cenários melhoraram a compilação ou correção para certos modelos, mas nenhum superou consistentemente os prompts originais, nem alcançou desempenho próximo ao Evosuite ou aos testes manuais.

3. Acesso aos códigos e artefatos usados no artigo

O artigo disponibilizou, por meio de um repositório público no Zenodo, o acesso a todo o código-fonte e aos resultados obtidos durante a pesquisa. O material completo pode ser consultado no seguinte link: https://zenodo.org/records/10530787.

3.1. Sobre os modelos LLMs usados

- **StarCoder (Decoder Only)**

- **Versão**: 1
- **Descrição**: Modelo de código aberto, disponível para download na plataforma Hugging Face:
- **Link**: https://huggingface.co/bigcode/starcoder

- **Gpt 3.5 Turbo (Decoder Only)**

- **Versão**: 3.5
- **Descrição**: Modelo proprietário da OpenAI, acessível por meio de API conforme a documentação oficial. Também disponível para uso direto no Playground da OpenAI.
- **Links**:
  API: https://platform.openai.com/docs/models/gpt-3.5-turbo
  Playground: https://platform.openai.com/chat/edit?models=gpt-3.5-turbo

- **Codex (Decoder Only)**

- **Versão**: code-davinci-002
- **Descrição**: Modelo open source da openai acessível atualmente via github no link:
- **Link**: https://github.com/openai/codex
- **Link documentação:** https://platform.openai.com/docs/models/davinci-002

3.2. Sobre os datasets usados

- **HumanEval** (Versão Java):

- **Descrição**: O conjunto de dados HumanEval foi originalmente desenvolvido pela OpenAI para avaliação de modelos de geração de código em Python, o artigo utilizou uma versão adaptada para a linguagem de programação java, essa versão está disponível junto com outros artefatos no link abaixo.
- **Link**: https://zenodo.org/records/10530787

- **EvoSuite** SF110 Benchmark Dataset:

- **Descrição**: Conjunto de dados amplamente utilizado para avaliação de técnicas de geração automática de testes em Java. Está associado ao benchmark SF110 e pode ser acessado em:
- **Link**: https://www.evosuite.org/experimental-data/sf110/

3.3. Exemplo de Avaliação da Solução

Na métrica de compilação, para o dataset HumanEval, Codex gerou testes com taxa de compilação original de 37,5%, que aumentou para 100% após aplicação de heurísticas de correção automática, como remover código extra e ajustar package. Na cobertura de código, para HumanEval, Codex alcançou cerca de 87,7% de cobertura de linhas e 92,8% de branch coverage.

3.4. Discussão sobre o potencial de uso e suas limitações

A dificuldade de escrever testes unitários manualmente, especialmente em linguagens fortemente tipadas como Java, é algo comum na prática profissional. O próprio artigo mostra que apenas 17% dos projetos no GitHub possuem testes, o que indica que há uma resistência ou dificuldade real nesse processo. Nesse cenário, o uso de LLMs pode automatizar parte dessa tarefa repetitiva, funcionando como uma ferramenta facilitadora para escrever testes de maneira mais rápida.

O processo descrito no estudo — em que se define um contexto, o método a ser testado e um prompt solicitando casos de teste — poderia ser facilmente integrado ao fluxo de trabalho, principalmente durante práticas como Test-Driven Development (TDD) ou revisão de código. Mesmo que os testes gerados pelos LLMs não sejam perfeitos ou completos, eles já trazem uma base inicial que pode ser ajustada manualmente com menos esforço do que criar tudo do zero.

Em contrapartida, o artigo também evidencia limitações importantes que precisam ser consideradas no uso prático. As principais delas são:

- **Erros de compilação:** os testes gerados frequentemente apresentam erros de sintaxe ou uso de elementos inexistentes, o que impede a execução direta sem correções, exigindo um gasto de tempo para encontrar e solucionar os problemas.

- **Ineficiência em testes complexos:** em projetos complexos, os LLMs não conseguem gerar testes com boa cobertura de linhas e ramos, logo, eles se tornam

obsoletos em cenários reais onde os códigos apresentam um alto nível de complexidade.

- **Alta incidência de test smells:** é comum a presença de test smells, como uso de valores fixos sem explicação (magic number), múltiplas asserções no mesmo teste sem mensagens claras (assertion roulette) e testes redundantes (lazy tests) nos testes gerados pelos modelos, o que compromete a manutenibilidade e a legibilidade dos testes.

- **Dependência de ajustes manuais:** embora os modelos gerem testes automaticamente, os resultados ainda exigem revisão e edição manual para garantir correção, clareza e aderência às boas práticas de testes.

Diante disso, o principal potencial dos LLMs está em seu uso como uma ferramenta complementar, sendo úteis para gerar rapidamente testes simples ou esboços iniciais de teste, mas que ainda exigem validação e refinamento manual — especialmente ao lidar com códigos mais complexos.

3.5. Como melhorar os resultados

Os resultados podem ser melhorados por meio de um sistema de feedback automático para o modelo. Para isso, o modelo será integrado a alguma ferramenta externa, como um compilador ou o EvoSuite, que enviará uma resposta sobre cada teste gerado. Dessa forma, o LLM utilizará essa resposta como prompt, tendo mais dicas de como melhorar o código gerado.

Como já foi dito anteriormente, fornecer melhores contextos e documentações leva a resultados superiores. Portanto, essa integração permitirá que ferramentas já utilizadas no mercado melhorem o contexto do prompt, auxiliando o LLM na geração de testes de maior qualidade.

Essa abordagem também dispensa a supervisão humana excessiva na geração de testes e, dependendo dos resultados obtidos em testes automatizados, pode se tornar uma solução eficaz para a geração automática de testes de código. Isso ocorre porque o sistema pode utilizar parâmetros definidos pelo programador para repetir o ciclo de feedbacks até que os critérios de qualidade sejam atingidos.

3.6. Problema do Stackoverflow

A pergunta no Stack Overflow escolhida é: “Generate automatically a jUnit class from a java class”, que solicita essa geração automática da classe JUnit, mas o autor já espera preencher manualmente os testes. A resposta aceita menciona: “Just select a class and go for: New → JUnit test case. Eclipse will ask you to checkbox all methods you want to test…” Ou seja, oferece apenas um esqueleto, não gera automaticamente asserts ou casos de teste completos, e nisso os LLMs estudados no artigo podem enriquecer a solução. Eles não só criam o stub da classe de teste, mas podem gerar testes JUnit completos com

asserts, se baseando no código do método ou na JavaDoc contida na classe, conforme analisado no estudo.

- URL do problema no Stackoverflow:

  https://stackoverflow.com/questions/32764453/generate-automatically-a-junit-class-from-a-java-class/32764647?utm_source=chatgpt.com

4. Referências

SIDDİQ, Mohammed Latif; SANTOS, Joanna C. S.; HASAN, Ridwanul; et al. *Using Large Language Models to Generate JUnit Tests: An Empirical Study*. arXiv preprint arXiv:2305.00418, 2024. Disponível em: https://arxiv.org/abs/2305.00418. Acesso em: 5 ago. 2025.

Anonymous. *Using Large Language Models to Generate JUnit Tests: An Empirical Study*. Zenodo, jan. 2024. DOI: 10.5281/zenodo.10530787. Disponível em: https://zenodo.org/records/10530787 . Acesso em: 05 ago. 2025.

ATHIWRATKUN, Ben; GOUDA, Sanjay Krishna; WANG, Zijian; LI, Xiaopeng; TIAN, Yuchen; MING, Ming; et al. *Multi-lingual Evaluation of Code Generation Models*. arXiv preprint arXiv:2210.14868, 2022. Disponível em: https://arxiv.org/abs/2210.14868. Acesso em: 5 ago. 2025.

BIGCODE. *StarCoder*. Hugging Face, 2023. Disponível em: https://huggingface.co/bigcode/starcoder. Acesso em: 5 ago. 2025.

OPENAI. *GPT-3.5 Turbo*. OpenAI Platform, 2023. Disponível em: https://platform.openai.com/docs/models/gpt-3.5-turbo. Acesso em: 5 ago. 2025.

OPENAI. *Playground GPT-3.5 Turbo*. OpenAI, 2023. Disponível em: https://platform.openai.com/chat/edit?models=gpt-3.5-turbo. Acesso em: 5 ago. 2025.

OPENAI. *Codex*. GitHub, 2021. Disponível em: https://github.com/openai/codex. Acesso em: 5 ago. 2025.

OPENAI. *HumanEval (Java version)*. GitHub, 2021. Disponível em: https://github.com/openai/human-eval. Acesso em: 5 ago. 2025.

FRASER, G.; ARCURI, A. *EvoSuite SF1100 Benchmark Dataset*. EvoSuite Project. Disponível em: https://www.evosuite.org/experimental-data/sf110/. Acesso em: 5 ago. 2025.