UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE SOFTWARE

VICTOR ANTHONY PEREIRA ALVES

AVALIAÇÃO DA QUALIDADE DE TESTES EM PYTHON GERADOS POR LLMS

QUIXADÁ

2025

VICTOR ANTHONY PEREIRA ALVES

AVALIAÇÃO DA QUALIDADE DE TESTES EM PYTHON GERADOS POR LLMS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Orientadora: Profa.Dra. Carla Ilane Moreira Bezerra.

QUIXADÁ

2025

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A482a Alves, Victor Anthony Pereira.
Avaliação da qualidade de testes python gerados por LLMs / Victor Anthony Pereira Alves. – 2025.
75 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Software, Quixadá, 2025.
Orientação: Profa. Dra. Carla Ilane Moreira Bezerra.

1. Large language models. 2. Código de teste python. 3. Test smells. I. Título.

CDD 005.1

VICTOR ANTHONY PEREIRA ALVES

AVALIAÇÃO DA QUALIDADE DE TESTES EM PYTHON GERADOS POR LLMS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Software do Campus de Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Software.

Aprovada em: 20/02/2025.

BANCA EXAMINADORA

Profa.Dra. Carla Ilane Moreira Bezerra (Orientadora)
Universidade Federal do Ceará (UFC)

Prof.Dr. Ivan do Carmo Machado
Universidade Federal da Bahia (UFBA)

Prof.Dr. Jeferson Kenedy Morais Vieira
Universidade Federal do Ceará (UFC)

Profa.Dra. Larissa Rocha
Universidade Estadual da Bahia (UNEB)

A todos que nunca baixaram a cabeça, mesmo diante das maiores tempestades. E à minha mãe, por acreditar e me ensinar que nenhuma tempestade é eterna.

AGRADECIMENTOS

A conclusão deste trabalho representa não apenas o fim de uma etapa acadêmica, mas também a soma de esforços, apoio e incentivo de muitas pessoas que, direta ou indiretamente, contribuíram para essa conquista.

Agradeço ao Criador, ao Rei dos Reis, que me agracia e me abençoa todos os dias e que me ensina cotidianamente que toda a minha vida possui um propósito.

Aos meus pais e meu irmão, meus maiores incentivadores em cada momento, que me ensinaram que a educação é o caminho para um futuro melhor e que, por uma oportunidade promissora, estariam ao meu lado para dar a volta ao mundo.

Aos meus avós Antonio, Raimunda e Valdenora (em memória), pelo exemplo, carinho, compreensão e aconchego. Vocês são luz na minha vida.

Aos meus familiares próximos, tios, padrinhos e primos pelo apoio cotidiano.

Aos meus amigos, pela parceria, pelas trocas de conhecimento e pelo apoio mútuo durante essa jornada. Sem vocês, essa trajetória teria sido muito mais difícil.

A todos aqueles que, mesmo distante, se fazem presente na minha vida todos os dias.

À minha orientadora, Dra. Carla Bezerra, pelos valiosos ensinamentos que contribuíram imensamente para o desenvolvimento deste trabalho e também abriram portas para o meu crescimento acadêmico. E por acreditar na minha capacidade.

Aos professores de diversas universidades que me apoiaram e continuam sendo fundamentais para o meu desenvolvimento acadêmico.

À Universidade Federal do Ceará (UFC), campus Quixadá, e ao corpo docente do curso de Engenharia de Software, pela excelência no ensino, pelas disciplinas ministradas, pelos projetos desafiadores, pela Iniciação Científica e pelas oportunidades de pesquisa e crescimento pessoal e profissional.

Aos membros da banca examinadora pelo tempo e pelas valiosas contribuições.

A todos que, de alguma forma, fizeram parte dessa caminhada, meu sincero agradecimento. Muito Obrigado.

"Do chão nós nos erguemos; Nas nossas naves vivemos; Nas estrelas sonhamos." (Becky Chambers.)

RESUMO

A geração manual de *scripts* de teste é um processo demorado, custoso e propenso a erros, ressaltando a importância de soluções automatizadas. Os *Large Language Models* (LLMs) têm demonstrado um potencial significativo nessa área, aproveitando seu vasto conhecimento para gerar código de teste de forma mais eficiente. Este estudo investiga a qualidade do código de teste em Python produzido por três LLMs: GPT-4o, Amazon Q e LLama 3.3. A confiabilidade estrutural das suítes de teste é avaliada em dois contextos distintos de *prompt*: *Text2Code* (T2C) e *Code2Code* (C2C). A análise envolve a identificação de erros e *test smells*, com um foco especial na correlação desses problemas com padrões inadequados de design. Os resultados indicam que a maioria das suítes de teste geradas pelos LLMs continha pelo menos um erro ou *test smell*. Os erros de asserção foram os mais prevalentes, representando 64% de todos os erros identificados, enquanto o *Lack of Cohesion of Test Cases* foi o *test smell* mais detectado (41%). O contexto do *prompt* teve um impacto significativo na qualidade dos testes, pois *prompts* textuais com instruções detalhadas geraram testes com menos erros, mas uma maior incidência de *test smells*. Entre os LLMs avaliados, o GPT-4o produziu a menor quantidade de erros em ambos os contextos (10% em C2C e 6% em T2C), enquanto o Amazon Q apresentou as maiores taxas de erro (19% em C2C e 28% em T2C). Em relação aos *test smells*, o Amazon Q teve menos detecções no contexto C2C (9%), enquanto o LLama 3.3 apresentou melhor desempenho no contexto T2C (10%). Além disso, observou-se uma forte relação entre erros específicos, como problemas de asserção e indentação, e *test smells* relacionados à coesão dos casos de teste. Esses resultados destacam oportunidades para aprimorar a qualidade dos testes gerados por LLMs e reforçam a necessidade de pesquisas futuras para explorar cenários de geração otimizados e estratégias mais eficazes de engenharia de *prompt*.

Palavras-chave: *large language models*; código de teste python; *test smells*; engenharia de *prompts*.

ABSTRACT

The manual generation of test scripts is a time-intensive, costly, and error-prone process, emphasizing the importance of automated solutions. Large Language Models (LLMs) have demonstrated significant potential in this area by leveraging extensive knowledge to generate test code more efficiently. This study examines the quality of Python test code produced by three LLMs: GPT-4o, Amazon Q, and LLama 3.3. The structural reliability of test suites is evaluated under two distinct prompt contexts: Text2Code (T2C) and Code2Code (C2C). The analysis involves identifying errors and test smells, with a particular focus on their correlation to inadequate design patterns. The findings indicate that most test suites generated by the LLMs contained at least one error or test smell. Assertion errors were the most prevalent, accounting for 64% of all identified errors, while Lack of Cohesion of Test Cases was the most frequently detected test smell (41%). Prompt context played a significant role in test quality, as textual prompts with detailed instructions tended to generate tests with fewer errors but a higher incidence of test smells. Among the evaluated LLMs, GPT-4o produced the fewest errors in both contexts (10% in C2C and 6% in T2C), whereas Amazon Q exhibited the highest error rates (19% in C2C and 28% in T2C). Regarding test smells, Amazon Q had fewer detections in the C2C context (9%), while LLama 3.3 showed better performance in the T2C context (10%). Additionally, a strong relationship was observed between specific errors, such as assertion and indentation issues, and test case cohesion smells. These results highlight opportunities to enhance the quality of LLM-generated tests and emphasize the need for further research into optimized generation scenarios and improved prompt engineering strategies.

Keywords: *large language models*; python test code; test smells; prompt engineering.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

AWS	*Amazon Web Services*
BERT	*Bidirectional Encoder Representations from Transformers*
C2C	*Code to Code*
GPT	*Generative Pre-trained Transformer*
IA	*Inteligência Artificial*
LLaMa	*Large Language Model Meta AI*
LLM	*Large Language Model*
PLN	*Processamento de Linguagem Natural*
T2C	*Text to Code*
T5	*Text-To-Text Transfer Transformer*

SUMÁRIO

1 INTRODUÇÃO

Teste de software é o processo que transforma os defeitos ocultos em defeitos identificáveis. Esta é uma fase crucial do ciclo de vida de desenvolvimento de software, pois revela os defeitos latentes em um produto de software (Kumar; Mishra, 2016). O teste unitário serve como uma faceta importante do teste de software porque permite que unidades individuais de código sejam testadas em isolamento (Gonzalez *et al.*, 2017). No entanto, o processo de escrita de *scripts* de testes unitários tende a ser negligenciado pelos desenvolvedores na maioria dos casos devido a sua dificuldade (Runeson, 2006; Beller *et al.*, 2015; Daka *et al.*, 2015).

A escrita manual de *scripts* de teste é um processo demorado, trabalhoso e caro (Li, 2022). Nesse sentido, um esforço cada vez maior tem sido dedicado à implementação de abordagens e ferramentas para automatizar a geração de testes unitários (Serra *et al.*, 2019). Neste contexto, diversos métodos para a geração automática de testes têm sido explorados e avaliados pelos pesquisadores (Shamshiri *et al.*, 2018; Tufano *et al.*, 2021). No entanto, Schäfer *et al.* (2024) destacam que a maioria dessas técnicas resultam em testes inutilizáveis e de baixa usabilidade.

Nesse contexto, ferramentas de Inteligência Artificial (IA) emergiram como auxiliares na programação e na geração de código, utilizando otimização baseada em pesquisa e técnicas orientadas por *feedback* para aprimorar seu desempenho (Hansson; Ellréus, 2023). Algumas dessas feramentas, como o Codex introduzido por Chen *et al.* (2021), são especializadas na geração de código a partir do Processamento de Linguagem Natural (PLN). O recente advento das ferramentas de geração de código baseadas em IA disponíveis gratuitamente apresenta várias oportunidades promissoras neste domínio (Becker *et al.*, 2023).

Estudos recentes, como os de Schäfer *et al.* (2024) e Haji *et al.* (2024), destacaram o potencial significativo dos *Large Language Models* (LLMs) no processo de geração de código de teste. Uma estratégia comumente adotada nesses estudos envolve o treinamento prévio de LLMs por meio da formulação de *prompts* em linguagem natural para guiar a geração de código de teste. Conforme destacado por Yu *et al.* (2023), essa abordagem capacita os LLMs a produzirem *scripts* de teste alinhados com os objetivos desejados, uma vez que esses modelos são capazes de compreender e gerar texto semelhante ao humano, permitindo a criação de códigos de teste adaptados a cenários específicos. Yetistiren *et al.* (2022) afirma que determinados LLMs podem gerar código válidos e corretos na maioria dos cenários. O estudo de Haji *et al.* (2024) observou que um dos principais fatores que impactam a usabilidade do código de teste é o contexto. Os

códigos de teste gerados por LLMs neste trabalho demonstraram uma eficácia ampliada quando incorporados em uma suíte de testes.

A Engenharia de *prompts*, o processo de elaboração de *prompts* eficazes para orientar os resultados do modelo, provou ser um fator essencial para otimizar o desempenho dos LLMs (Sahoo *et al.*, 2024; Marvin *et al.*, 2024). *Prompts* bem projetados podem melhorar significativamente a qualidade, a estrutura e a usabilidade do código de teste gerado (Cain, 2024). A literatura mostrou que os LLMs podem se adaptar a diferentes tipos de *prompts*, como descrições textuais (*Text2Code* ou T2C) e trechos de código pré-existentes (*Code2Code* ou C2C), produzindo níveis variados de qualidade, dependendo do contexto de geração (Lu *et al.*, 2021; Agarwal *et al.*, 2024). Essa adaptabilidade destaca o potencial transformador dos LLMs na automação de testes de software.

Apesar de seus recursos, os *scripts* de teste gerados pelos LLMs costumam apresentar problemas relacionados a erros e *test smells* (Siddiq *et al.*, 2024). Assim como o código de produção, o código de teste deve aderir às práticas de programação adequadas para garantir a confiabilidade e a capacidade de manutenção (Wang *et al.*, 2022b). Os *test smells*, que se referem a escolhas de *design* ou implementação abaixo do ideal no código de teste, podem reduzir a eficácia dos testes na detecção de falhas e na validação do comportamento do software (Tufano *et al.*, 2016; Tufano *et al.*, 2021). Por exemplo, o estudo de Alves *et al.* (2024) identificou um número significativo de *test smells* em testes gerados pelo GitHub Copilot (que utilizava, na época do estudo, o modelo GPT-4). Os problemas comuns incluíram *smells* de asserção, coesão deficiente entre casos de teste e tratamento inadequado de erros, destacando a necessidade de uma investigação mais aprofundada sobre o desempenho do LLM em diversos contextos.

Este estudo se baseia no trabalho de Alves *et al.* (2024), ampliando a análise da qualidade do código de teste gerado pelo LLM em três modelos importantes: GPT-4o, Amazon Q e LLama 3.3. Foi comparado o desempenho desses modelos em dois contextos de *prompts*, Text2Code e Code2Code, para avaliar sua capacidade de produzir *scripts* de teste Python de alta qualidade. Esta pesquisa se concentra na confiabilidade estrutural dos testes gerados, analisando erros e *test smells* para identificar padrões associados ao *design* não ideal. Ao investigar a relação entre esses dois contextos (erros e *test smells*), o objetivo deste estudo consiste em descobrir oportunidades para melhorar a geração de testes baseados em LLM e fornecer percepções práticas para pesquisadores e profissionais. Além de avaliar o desempenho dos modelos, este trabalho enfatiza a função da engenharia de *prompt* como um mecanismo para aprimorar a usabilidade e a

confiabilidade da geração de testes automatizados. À medida que os LLMs continuam a avançar, compreender suas limitações e otimizar seu uso na engenharia de testes são etapas essenciais para melhorar a qualidade do software e reduzir os custos de desenvolvimento.

1.1 Objetivos

1.1.1 Objetivo Geral

Conduzir um estudo empírico comparativo para avaliar a capacidade de diferentes LLMs na geração de códigos de teste de qualidade em diferentes contextos e *prompts*.

1.1.2 Objetivos Específicos

a) Investigar a ocorrência de erros no código de teste gerado por LLMs em resposta a dois contextos: *Text2Code* e *Code2Code*.
b) Avaliar a presença e a distribuição de *test smells* nos códigos de teste gerados por LLMs em cada contexto de *prompt*, identificando os padrões mais comuns.
c) Analisar as diferenças de desempenho entre os LLMs na geração de códigos de teste considerando as particularidades dos contextos *Text2Code* e *Code2Code*.
d) Identificar possíveis padrões e correlações entre a ocorrência de erros e *test smells* nos códigos de teste gerados por LLMs.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo serão apresentados os conceitos relevantes e necessários para compreender este trabalho. Na Seção 2.1, serão discutidas as definições de testes unitários em Python, os erros mais comuns nesses testes e a geração automática de testes unitários. Em seguida, na Seção 2.2, serão explorados os conceitos sobre LLMs com destaque para alguns modelos (Github Copilot, Amazon Q Developes e LLaMa), incluindo suas utilidades e funcionalidades. Além disso, a Seção 2.2 também contém conceitos de Engenharia de *prompts.* Por fim, a Seção 2.3 fornecerá uma visão geral dos conceitos básicos relacionados à qualidade de código de teste, com ênfase nos conceitos de *test smells* e suas ferramentas de detecção.

2.1 Teste de Unidade

Teste de unidade é um tipo de verificação que procura por defeitos e valida o funcionamento de componentes do software, como módulos, objetos e classes, que podem ser testados separadamente (Graham *et al.*, 2021). Para diminuir a quantidade de bugs levados para o ambiente de produção, é necessário testar o código constantemente. Idealmente, a cada alteração feita, todo o sistema deve ser testado por inteiro novamente (Aniche, 2014). Todos os testes devem ser integrados ao ciclo de desenvolvimento, eles devem ser executados em cada mudança de código, mesmo a menor delas (Khorikov, 2020).

Escrever testes de unidade é uma atividade comum em que os desenvolvedores escrevem casos de teste junto com o código de produção. Essa prática é últil para evitar possíveis erros de programação e detectar problemas logo no início do processo de desenvolvimento (Peng *et al.*, 2021). Estruturas de automação como JUnit[1] para Java popularizaram essa abordagem, permitindo a execução frequente e automática de suítes de testes unitários (Daka; Fraser, 2014).

2.1.1 Teste Unitário em Python

Na linguagem Python, as bibliotecas que popularizaram a escrita de testes unitários são o *pytest*[2] e o *unittest*[3], ambas oferecendo uma variedade de comandos que facilitam a construção de cenários de teste específicos para essa linguagem. A demanda por testes unitários em Python tem aumentado, impulsionada pelo crescente aumento de popularidade da linguagem

[1] https://junit.org/junit5/
[2] https://docs.pytest.org/en/8.2.x/
[3] https://docs.python.org/3/library/unittest.html

no desenvolvimento de programas (Trautsch; Grabowski, 2017).

O Código-fonte 1 exemplifica uma classe de testes unitários em Python utilizando a biblioteca *unittest*. A classe de teste é identificada como uma subclasse de *unittest.TestCase* e contém três casos de teste distintos, que validam o comportamento de três métodos de *Strings*. Os três testes são definidos por métodos cujos nomes começam com a palavra *test*, uma convenção que sinaliza ao executor quais métodos representam os testes. O ponto crucial de cada teste é a chamada para os métodos *assert*, que verificam os resultados esperados. O método *assertEqual()* verifica se o resultado obtido corresponde exatamente ao esperado, *assertTrue()* e *assertFalse()* avaliam uma condição como verdadeira ou falsa, respectivamente, e *assertRaises()* verifica se uma exceção específica é lançada.

Código-fonte 1 – Exemplo de testes unitários em Python

```
import unittest
class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

Fonte: https://docs.python.org/3/library/unittest.html#basic-example

2.1.2 Erros de Testes Unitários em Python

Assim como o código de produção, o código de teste unitário em Python pode apresentar erros na sua estrutura que são detectados pelo próprio interpretador e que impedem o código de ser executado com sucesso. A linguagem Python fornece uma documentação extensa que cataloga e mapeia os possíveis erros que podem ser detectados nos códigos, o material está

disponível na documentação de Erros[4] e Exceções[5]. Alguns estudos, como o de Haji *et al.* (2024) e Alves *et al.* (2024), detectaram esses erros nos códigos de teste python para considerá-los como válidos e inválidos. Nas Subsubseções 2.1.2.1, 2.1.2.2, 2.1.2.3 e 2.1.2.5 estão mapeados alguns dos erros mais frequentes que estão documentados pelo Python e um exemplo em código de como eles são aplicáveis em testes.

2.1.2.1 AssertionError

Essa exceção é gerada quando uma instrução *assert* falha. Na biblioteca *unittest*, os *assertions* são mais diversos, incluindo *assertEquals*, *assertTrue*, *assertIn*, entre outros. A ocorrência dessa exceção indica que a expectativa do teste não está alinhada com o comportamento do código de produção. O Código-fonte 2 ilustra um exemplo em que essa exceção é levantada. O método da Linha 1 verifica se um *AttributeError* é gerado onde as mensagens de erro devem devem corresponder à seguinte *string*: *"unknown image attribute fake_attribute"* (que está implementada no código de produção). No entando, no teste, a string validada é: *"image does not have attribute"*. Isso levantará um erro de expectativa, uma vez que o teste não espera o mesmo do método de produção. Esse tipo de exceção é comumente encontrada em testes unitários, uma vez que é o domínio de código que mais utiliza asserções na sua estrutura.

Código-fonte 2 – Exemplo de teste unitário contendo erro de asserção em Python

```
1 def test_handle_bad_attribute () :
2 """ Verify that accessing a nonexistent attribute
3 raises an AttributeError ."""
4     with open (os.path.join(os.path.dirname(__file__), "grand_canyon.jpg"), "
    rb"
5     ) as image_file :
6         image = Image (image_file)
7
8     with pytest.raises(AttributeError,match="image does not have attribute"):
9         image.fake_attribute
```

Fonte: Haji *et al.* (2024)

[4] https://docs.python.org/3/tutorial/errors.html
[5] https://docs.python.org/3/library/exceptions.html

2.1.2.2 *SyntaxError*

O *SyntaxError* é gerado quando o código viola as regras sintáticas da linguagem, ou seja, quando não segue o padrão de interpretação do Python. Um exemplo comum é o esquecimento de fechar parênteses após uma chamada de função ou expressão, o que resulta em um erro. Além disso, a ausência de dois pontos (:) após a declaração de estruturas como condicionais, laços ou definições de funções pode gerar erros de sintaxe. Outro erro frequente é deixar uma string sem a citação de fechamento, seja com aspas simples ou duplas. No Código-fonte 3, esse erro pode ser observado na última linha, onde o parêntese da chamada à função *assertEqual* não foi fechado corretamente, resultando em uma falha na análise do código.

Código-fonte 3 – Exemplo de teste unitário com erro de sintaxe em Python

```
1  def test_captures_stdout_stderr(self) :
2  """ Test capturing stdout and stderr from print """
3      message = " This should be captured ..."
4
5      buf = io.StringIO ()
6      with stdout_err_redirector(buf) :
7      print(message)
8      print(message , file=sys.stderr)
9
10     self . assertEqual(buf.getvalue() , message + '\n'+ message + '\
```

Fonte: Haji *et al.* (2024)

2.1.2.3 *IndentarionError*

O *IndentationError* é um erro bastante comum em Python. Como se trata de uma linguagem interpretada, seu interpretador processa o código linha por linha. Para que o código seja executável, é essencial que ele esteja devidamente formatado, respeitando corretamente os espaços e recuos. Esse alinhamento correto é conhecido como indentação. Caso a indentação não seja aplicada corretamente, o interpretador levantará um erro de indentação, impedindo a execução do código. O Código-fonte 4 ilustra um erro de indentação na classe de teste declarada na Linha 3. O erro de indentação ocorre porque a linha *self.assertEqual(result, 4)* está com um nível de indentação a mais do que deveria. Além disso, a classe *TestExample* está no mesmo nível de indentação do método *def test_addition(self)*. Isso resultará em um erro de *IndentationError* ao tentar executar o código.

Código-fonte 4 – Exemplo de teste unitário com erro de indentação em Python

```
1  import unittest
2
3  class TestExample(unittest.TestCase):
4
5  def test_addition(self):
6      result = 2 + 2
7          self.assertEqual(result, 4)
8
9  if __name__ == "__main__":
10     unittest.main()
```

Fonte: Criado pelo Autor.

A estrutura correta do código para evitar o *IndentationError* seria a do Código-fonte 5, em que todos os elementos estão no nível correto de indentação:

Código-fonte 5 – Exemplo de teste unitário com a indentação correta em Python

```
1  import unittest
2
3  class TestExample(unittest.TestCase):
4
5      def test_addition(self):
6          result = 2 + 2
7          self.assertEqual(result, 4)
8
9  if __name__ == "__main__":
10     unittest.main()
```

Fonte: Criado pelo Autor.

2.1.2.4 KeyError

O erro *KeyError* é levantado quando uma chave de mapeamento não é encontrada entre as chaves existentes no conjunto de dados. Em Python, o mapeamento mais comum é o dicionário. Quando tentamos acessar um item no dicionário usando uma chave que não está presente no conjunto de chaves, o Python gera o erro *KeyError*. O método de teste apresentado no Código-fonte 6 falha porque o valor da chave *CRON_VAR* não existe em *self.crontab.env*. Outro erro de busca ocorre quando um valor de índice fora do intervalo é usado em um dicionário ou array.

Código-fonte 6 – Exemplo de teste unitário com erro de chave em Python

```
import unittest

def test_06_env_access(self):
    """ Test that we can access env variables """
    self.assertEqual(self.crontab.env['PERSONAL_VAR'], 'bar')
    self.assertEqual(self.crontab.env['CRON_VAR'], 'fork')
    self.assertEqual(self.crontab[0].env['CRON_VAR'], 'fork')
    self.assertEqual(self.crontab[1].env['CRON_VAR'], 'spoon')
    self.assertEqual(self.crontab[2].env['CRON_VAR'], 'knife')
    self.assertEqual(self.crontab[3].env['CRON_VAR'], 'knife')
    self.assertEqual(self.crontab[3].env['SECONDARY'], 'fork')
```

Fonte: Haji *et al.* (2024)

2.1.2.5 NameError

Em Python, um *NameError* é gerado quando o programa tenta acessar ou usar uma variável que não foi definida ou atribuída a um valor. Isso pode acontecer se a variável for escrita incorretamente ou se for acessada antes de ser definida. Esse erro pode ser gerado tanto com variáveis, quanto com outro tipo de nomes, como declarações de funções, classes ou bibliotecas que não existem. Por exemplo, o método de teste do Código-fonte 7 contém o uso de uma classe chamada *CronRange*. Isso resultou em um erro *NameError*, indicando que *CronRange* não está definida. Dentro do projeto, realmente existe uma classe chamada *CronRange*. No entanto, ela não está importada e, portanto, não existe no namespace. Esse tipo de erro pode acontecer também caso as variáveis não sejam importadas corretamente.

Código-fonte 7 – Exemplo de teste unitário com erro de nome em Python

```
    def test_18_range_cmp(self):
    """ Compare ranges """
    self.assertEqual(CronRange('*/6'), CronRange('*/6'))
    self.assertNotEqual(CronRange('*/6'), CronRange('*/7'))
    self.assertNotEqual(CronRange('*/6'), CronRange('*/6-7'))
```

Fonte: Haji *et al.* (2024)

2.1.3 Geração Automática de Teste Unitário

A geração automática de testes possibilita a produção e execução de um grande número de entradas que exercitam extensivamente a unidade em teste (Xie; Notkin, 2006). Esses testes podem ser integrados aos códigos do software e, assim como os testes escritos manualmente, oferecem suporte aos desenvolvedores durante a manutenção do software (Shamshiri *et al.*, 2018). Consequentemente, um amplo espectro de técnicas para a geração automática de testes unitários tem sido objeto de investigação ao longo dos anos, visando reduzir os esforços manuais na elaboração de testes (McMinn, 2004; Yuan *et al.*, 2024).

Dentre as técnicas mais tradicionais, destacam-se a geração de testes baseados em modelos, especificações, análise estática e teste baseado em pesquisa (McMinn, 2004; Maragathavalli, 2011). Algumas dessas abordagens operam a partir de especificações do software, capturando modelos comportamentais, diagramas de atividade, gráficos de estado e diagramas de sequência (Wang *et al.*, 2022a). Outras abordagens utilizam algoritmos evolutivos, que são métodos computacionais utilizados para treinar um conjuntos de casos de teste e avaliar a qualidade desses conjuntos em termos de cobertura de código, detecção de falhas, ou outros critérios de qualidade de teste (Fraser; Arcuri, 2013).

Estudos anteriores sobre sistemas de código aberto mostraram que as ferramentas de geração de testes são bastante eficazes na detecção de falhas (Almasi *et al.*, 2017). A maioria das ferramentas, como o Evosuite[6] e Randoop[7], oferece suporte à linguagem Java e gera testes no formato JUnit (Fraser; Arcuri, 2011). No entanto, atualmente, diversos ecossistemas de programação também contam com ferramentas similares. Por exemplo, o PYNGUIN[8] é uma ferramenta extensível para geração de testes em Python (Lukasczyk; Fraser, 2022), enquanto o JSEFT[9] utiliza cobertura de função e algoritmos de abstração de estado para gerar testes automatizados em JavaScript (Mirshokraie *et al.*, 2015), entre várias outras disponíveis para uso.

Entretanto, a aplicação das ferramentas de geração automática de testes unitários na indústria é limitada, em parte devido ao tempo que os desenvolvedores precisam dedicar para analisar a saída dessas ferramentas (Fraser *et al.*, 2015). No entanto, a IA tem desempenhado um papel importante na superação desses desafios, especialmente quando se trata da geração de casos de teste baseados em PLN. A geração automática de código por IA tem o potencial

[6] https://www.evosuite.org/
[7] https://randoop.github.io/randoop/
[8] https://www.pynguin.eu/
[9] https://github.com/saltlab/JSeft

de reduzir significativamente o tempo e os custos associados à codificação manual e à busca por ferramentas que automatizam esse processo (Hansson; Ellréus, 2023). Por meio dessa abordagem, as ferramentas de IA podem ser treinadas para gerar código a partir da linguagem humana, permitindo a criação de *scripts* de teste com base em cenários descritos pelo próprio desenvolvedor. Tudo o que é necessário é especificar as entradas, saídas e arquitetura do teste (Yuan *et al.*, 2024).

2.2 *Large Language Models*

LLMs são modelos de linguagem baseados em IA que foram treinados em grandes quantidades de texto para aprender padrões de linguagem e gerar texto de forma semelhante ao humano (Yang *et al.*, 2024). Os primeiros modelos de linguagem têm como objetivo principal modelar e gerar dados de texto, enquanto os modelos de linguagem mais recentes, por exemplo, o *Generative Pre-trained Transformer 4* (GPT-4) se concentram na resolução de tarefas complexas (Zhao *et al.*, 2023). De acordo com Yao *et al.* (2024), um LLM deve ter quatro características principais : (i) compreensão profunda do contexto da linguagem natural; (ii) capacidade de gerar texto semelhante ao humano; (iii) consciência contextual, especialmente em domínios intensivos em conhecimento; (iv) forte capacidade de seguir instruções que é útil para a resolução de problemas e tomada de decisões.

Recentemente, a OpenAI[10] se destacou como a empresa pioneira no desenvolvimento de modelos de linguagens avançados. No último relatório técnico referente ao desenvolvimento do GPT-4 levantado pela empresa, os resultados destacam que o modelo exibe desempenho de nível humano em vários *benchmarks* profissionais e acadêmicos (OpenAI, 2024). Esta tecnologia, assim como outros LLMs, passa por um extenso treinamento com grandes quantidades de dados para conseguir compreender e modelar as saídas baseadas em aprendizado de máquina (Roumeliotis; Tselikas, 2023). Dessa forma, a arquitetura dos LLMs é dividida em dois grupos:

- ***Encoder-only*:** Esses modelos são projetados com arquiteturas que permitem apenas a codificação de texto. Incluem modelos como *Bidirectional Encoder Representations from Transformers* (BERT) e *Text-To-Text Transfer Transformer* (T5) (Fan *et al.*, 2023; Yang *et al.*, 2024).
- ***Decoder-only*:** Esses modelos são projetados especificamente para gerar texto de forma autônoma a partir da decodificação de texto, sem a necessidade de entrada de contexto

[10] https://openai.com/

adicional durante a geração. Incluem modelos como o GPT, GPT-3 e GPT-4 (Fan *et al.*, 2023; Yang *et al.*, 2024).

A Figura 1 apresenta a arquitetura geral *Encoder-Decoder* dos LLMs, conforme proposta por Vaswani *et al.* (2017) e denominada "*The Transformer Model*". Essa arquitetura é utilizada em modelos de PLN para compreensão e geração de texto.

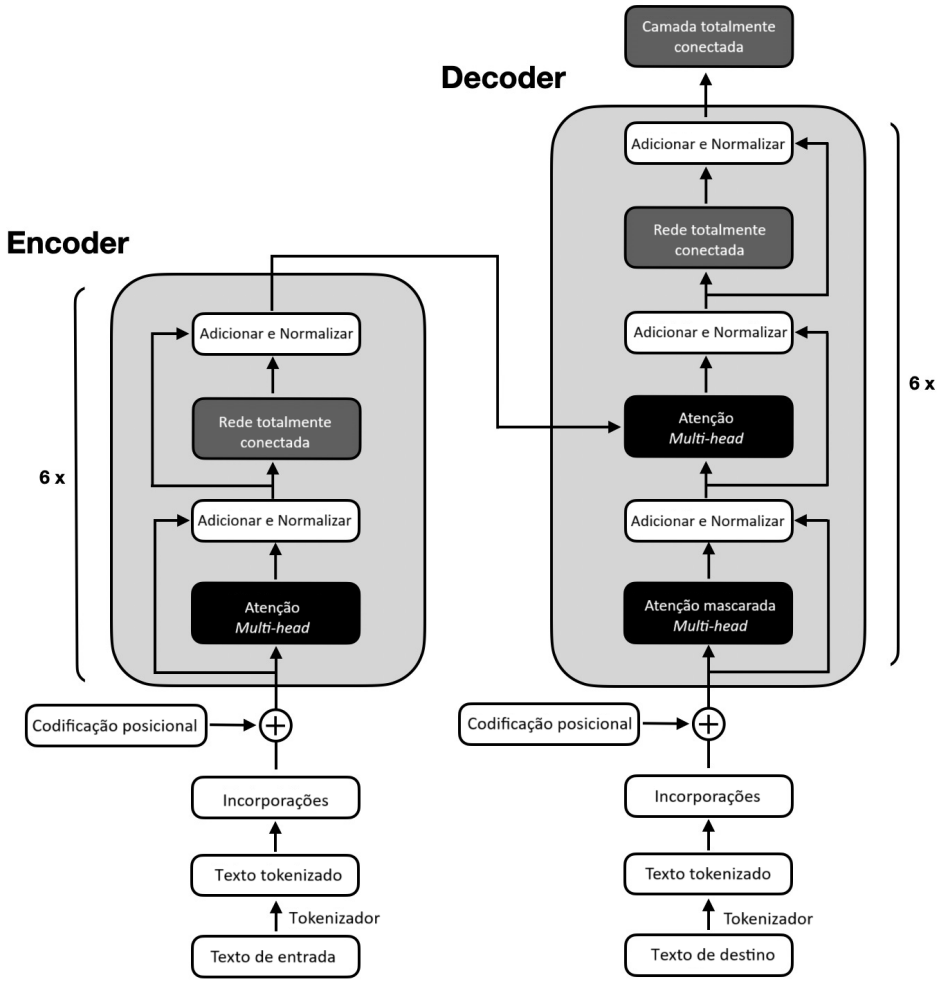
À esquerda da Figura 1, está representado o processo de codificação de textos (*Encoder*), abordagem adotada pelo modelo BERT. O *Encoder* tem como principal objetivo compreender o contexto de uma palavra considerando tanto as palavras anteriores quanto as posteriores. O texto de entrada passa por um processamento no *Encoder*, gerando uma representação interna que captura informações contextuais de maneira eficaz. À direita da figura, encontra-se a decodificação de texto (*Decoder*), abordagem utilizada pelo modelo GPT. Diferente do *Encoder*, o *Decoder* foca na geração de texto, prevendo o próximo *token* da sequência de saída com base nas palavras geradas até o momento. Esse mecanismo permite que o modelo construa sentenças coerentes e contextualmente relevantes.

A Figura 1 também evidencia a interação entre o *Encoder* e o *Decoder*, ilustrando como esses componentes trabalham em conjunto para aprimorar tanto a compreensão quanto a geração de texto. No processo de tradução automática, por exemplo, o *Encoder* transforma a sentença de entrada em uma representação intermediária rica em contexto, que é então utilizada pelo *Decoder* para gerar a tradução correspondente na língua de destino. Essa comunicação entre os módulos permite a adaptação à diferentes tarefas de PLN. Enquanto o *Encoder* captura relações semânticas complexas no texto de entrada, o *Decoder* emprega essas informações para gerar saídas coerentes e contextualmente apropriadas. Dessa forma, a arquitetura *Encoder-Decoder* demonstra como diferentes partes dos LLMs são otimizadas para tarefas específicas, garantindo eficiência desde a interpretação até a produção textual.

Além disso, os LLMs podem ser treinados com parâmetros incorporados à sua arquitetura para processar os dados de entrada e gerar saídas mais precisas. Alguns modelos utilizam uma vasta quantidade de parâmetros, que podem variar de 6 bilhões (6B) a 70 bilhões (70B). Isso implica que, em geral, quanto maior o número de parâmetros, maior a capacidade do modelo de capturar nuances e detalhes, resultando em respostas mais precisas e de maior qualidade, essas descobertas foram registradas por Yeom *et al.* (2024) ao promover e avaliar o modelo de linguagem da Meta[11] chamado LLaMa (*Large Language Model Meta AI*).

[11] https://about.meta.com/br/

Figura 1 – Arquitetura *Transformer* utilizada pelos LLMs



Fonte: Vaswani *et al.* (2017).

Essas múltiplas habilidades dos LLMs estão sendo aproveitados para automatizar atividades de engenharia de software, como traduzir código entre linguagens de programação, gerar documentação para código e gerar de testes unitários (Ross *et al.*, 2023). Fan *et al.* (2023) descrevem como "Propriedades Emergentes"as habilidades dos LLMs que têm sido cada vez mais buscadas e estudadas na área da Engenharia de Software. Estas propriedades incluem codificação, reparo, refatoração, melhoria de desempenho, documentação e análise. O Quadro 1 mapeia os dois LLMs mais utilizados para gerar códigos de acordo com Yetiştiren *et al.* (2023) e suas principais características: Amazon CodeWhisperer e GitHub Copilot. Para este trabalho, também foi adicionado o modelo de linguagem LLaMa e as características foram destacadas nos parâmetros propostos pelos autores. As subseções 2.2.1, 2.2.2 e 2.2.3 fornecem uma descrição detalhada de suas funcionalidades.

Quadro 1 – Características dos modelos de linguagens mais utilizados para gerar códigos

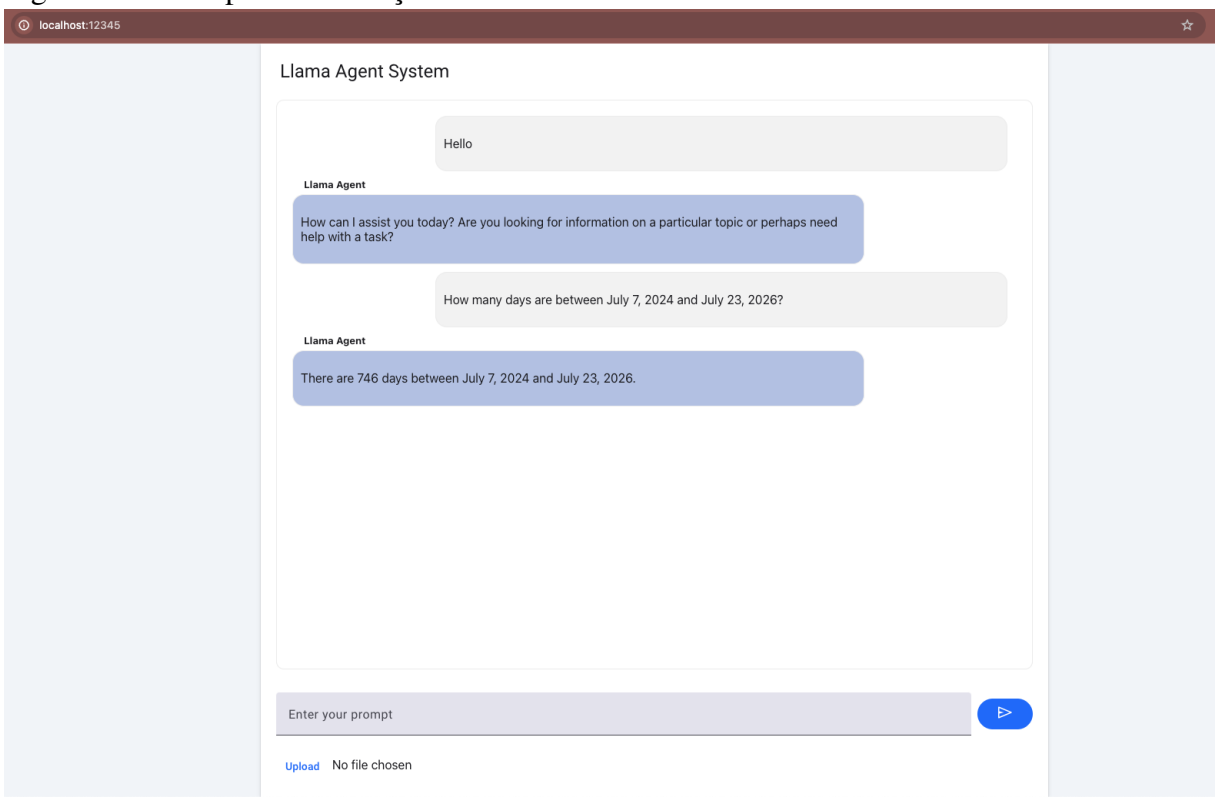
Funcionalidades	**LLaMa**	**Amazon CodeWhisperer**	**GitHub Copilot**
Suporte para IDE	Sem suporte para IDEs	JetBrains, Visual Studio Code, AWS Cloud9, AWS Lambda console	IntelliJ IDEA, Android Studio, AppCode, CLion, Code With Me Guest, DataGrip,DataSpell, GoLand, JetBrains Client, MPS, PhpStorm, PyCharm, Rider, RubyMine, WebStorm
Lançamento	Fevereiro de 2023	Junho de 2022	Outubro de 2021
Desenvolvedor	Meta	AWS	OpenAI - GitHub
Fornece fontes e referências das sugestões	Não	Sim	Não
Explicação das sugestões	Sim	Não	Não
Fornece múltiplas sugestões	Sim	Sim	Sim
Fonte de dados de treinamento	Repositórios	“Vastas quantidades de código disponíveis publicamente”	“...treinado em todas as linguagens que aparecem em repositórios públicos" (adaptado)
Linguagens de programação que fornece suporte	N/A	C#, Java, JavaScript, Python, TypeScript	Python, C, C++, C#, Go, Java, JavaScript, PHP, Ruby, Scala, TypeScript
Polivalente (exceto programação)	Sim	Não	Não
Pode ser usado offline	Não	Não	Não
Pode acessar arquivos locais	Não	Sim	Sim

Fonte: Adaptado de Yetiştiren *et al.* (2023)

2.2.1 LLaMa

O LLaMa é um modelo de linguagem desenvolvido pela Meta, focado em criar uma alternativa eficiente e acessível para pesquisadores e desenvolvedores que trabalham com IA. Conforme demonstrado no estudo de Yeom *et al.* (2024), esse LLM busca alcançar desempenho competitivo em tarefas de PLN ao utilizar menos recursos computacionais em comparação com modelos maiores, como o GPT. A Meta lançou o LLaMa com a intenção de fomentar a inovação na IA, oferecendo um modelo que combina alta performance com eficiência, e que pode ser facilmente acessado e adaptado pela comunidade acadêmica e por empresas (Meta, 2023). Seu lançamento inicial foi em fevereiro de 2023 com o LLaMa 1, que incluía modelos de 7B a 65B parâmetros. Em julho de 2023, a Meta lançou o LLaMa 2, aprimorando a eficiência e o desempenho, com versões de 7B, 13B e 70B parâmetros. A versão mais atual, LLaMa 3, traz ainda mais refinamentos, consolidando-se como uma ferramenta para pesquisa e aplicações comerciais em IA. A Figura 2 demonstra um exemplo de utilização do modelo extraído dos anúncios da própria Meta.

Figura 2 – Exemplo de utilização do LLaMa

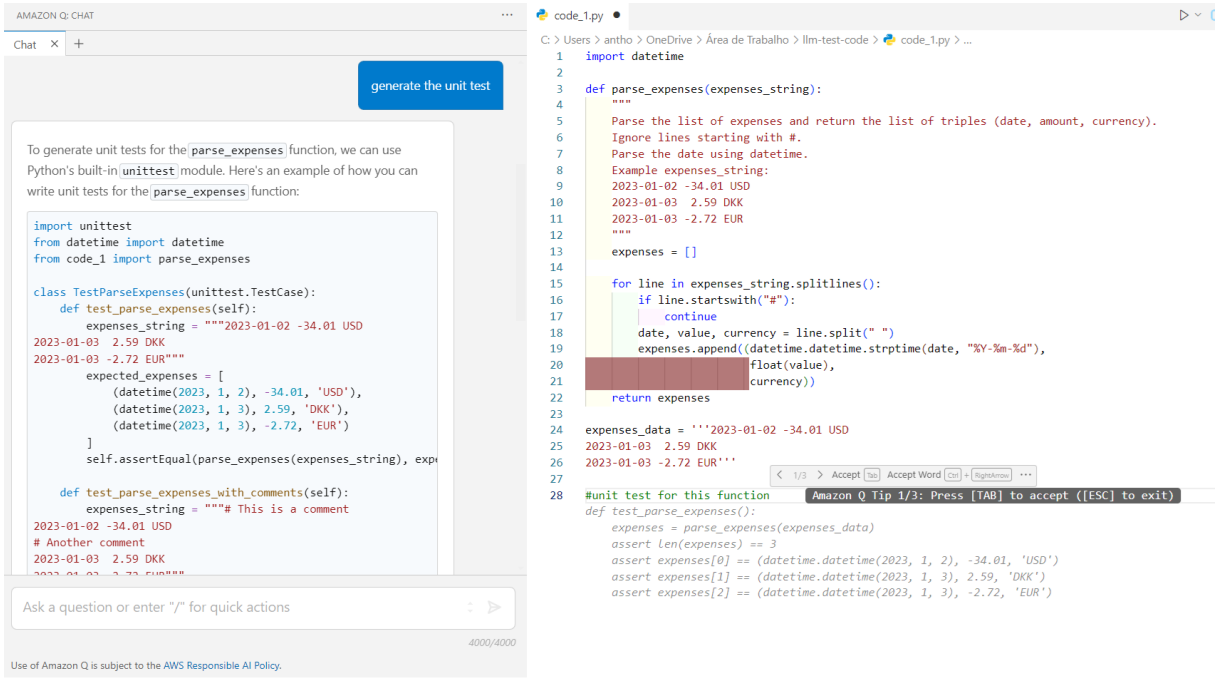


Fonte: https://github.com/meta-llama/llama-agentic-system

2.2.2 Amazon Q Developer

Incialmente chamado de Amazon CodeWhisperer, o Q Developer é um LLM desenvolvido pela *Amazon Web Services* (AWS) e foi anunciado em 2022 como um modelo de aprendizado de máquina treinado em fontes de dados da Amazon e em códigos-fonte abertos (Bays, 2022). Esse modelo pode ser integrado a IDEs compatíveis, permitindo que os usuários forneçam instruções em linguagem natural para que o Q Developer responda de maneira adequada. Além disso, quando os desenvolvedores escrevem um comentário no editor de código de sua IDE conforme exibido na Figura 3, o LLM analisa automaticamente o comentário e identifica os serviços mais apropriados. Em seguida, ele gera e insere um trecho de código diretamente no editor (Yetiştiren *et al.*, 2023). Embora a arquitetura do Q Developer seja especificamente adaptada e integrada aos diversos serviços da AWS, estudos recentes indicam que, apesar de ser uma ferramenta útil, ele pode não ser tão eficaz quanto modelos baseados em GPT-x na geração de respostas válidas, por mais que se destaque em outros âmbitos, como sugestões de código e completamento automático (Huang *et al.*, 2024; Kapitsaki, 2024).

Figura 3 – Exemplo de utilização do Amazon CodeWhisperer



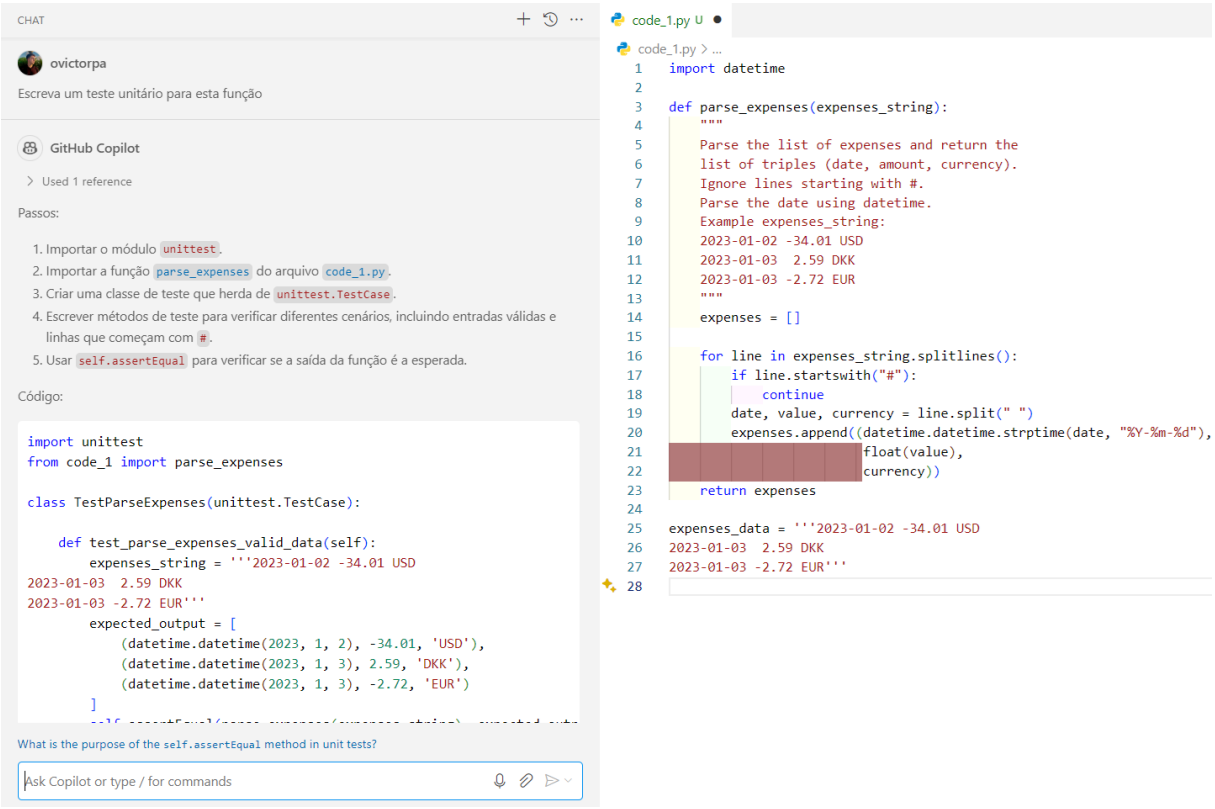
Fonte: Criado pelo autor.

2.2.3 GitHub Copilot

O GitHub Copilot[12], lançado em 2021, é um LLM de codificação assistida desenvolvido pela OpenAI em colaboração com o GitHub, utilizando o modelo GPT-4o. Conhecido popularmente como *"AI pair programmer"*, sua principal funcionalidade é a geração automática de código. Isso significa que, ao abrir um arquivo de código de produção em uma IDE com suporte ao Copilot, os desenvolvedores podem usar a ferramenta para criar automaticamente testes unitários (Figura 4), agilizando o processo de desenvolvimento. Dessa forma, essas funcionalidades do Copilot têm motivado estudos recentes para avaliar sua capacidade em resolver bugs, refatorar código e realizar outras atividades relacionadas à programação mediante interações em linguagem natural (Imai, 2022; Dakhel *et al.*, 2023; Wermelinger, 2023). Embora esses estudos enfatizem a capacidade do Copilot em gerar código de maneira eficiente, outros ressaltam suas limitações, afirmando que ainda há necessidade de intervenção humana e ajustes manuais para maximizar sua eficácia e garantir a qualidade dos resultados produzidos (Mastropaolo *et al.*, 2023).

[12] https://github.com/features/copilot

Figura 4 – Exemplo de utilização do GitHub Copilot



Fonte: Adaptado de https://github.com/features/copilot

2.2.4 Engenharia de prompts

Um *prompt* é um conjunto de instruções fornecidas a um LLM que orienta e personaliza o modelo, aprimorando ou refinando suas capacidades (Liu *et al.*, 2023). A Engenharia de *prompts* é um conjunto de procedimentos de programação de instruções para personalizar saídas e interações de LLMs, sendo uma habilidade obrigatória para a eficácia da comunicação e da interação com essas ferramentas (Marvin *et al.*, 2024). Esse processo envolve projetar estrategicamente instruções específicas para tarefas, orientando a saída do modelo sem alterar os parâmetros (Sahoo *et al.*, 2024). Desse modo, essa engenharia surge como um mecanismo chave para acessar as capacidades transformadoras do modelo de linguagem no processo de aprendizagem e na formulação da resposta (Cain, 2024).

No contexto de Engenharia de Software, a eficiência na modelagem de *prompts* é essencial para guiar e contextualizar o modelo acerca da estrutura e organização das diversas tarefas relacionadas ao código-fonte (Pornprasit; Tantithamthavorn, 2024). Nesse sentido, estudos recentes sobre automação de processos com LLMs, como o de Agarwal *et al.* (2024) e o de Lu *et al.* (2021), caracterizaram tipos diferentes de contextos envolvendo *prompts* que podem ser utilizados para a geração de saídas por LLMs, dentre elas:

- **Código para Código (*Code2Code*):** contexto em que o *prompt* é um trecho de código e a saída também é um trecho de código. Utilizado em tarefas de produção, manutenção e tradução de código.
- **Texto para Código (*Text2Code*):** contexto em que o *prompt* é um texto em linguagem natural e a saída é um trecho de código. Utilizado em tarefas de geração e refatoração de código.
- **Código para Texto (*Code2Text*):** contexto em que o *prompt* é um trecho de código e a saída é um texto em linguagem natural. Utilizado em tarefas de resumo e documentação de código.

Esses contextos de *prompts* destacam a importância de projetá-los para fornecer valor além do simples texto ou geração de código (White *et al.*, 2023). Em dois estudos aprofundados sobre geração de respostas em um contexto de *prompts*, Reeves *et al.* (2023) e Liu *et al.* (2024) destacaram que a eficácia dos *prompts* pode variar significativamente dependendo do contexto e do objetivo em questão. Eles afirmam que a forma como as instruções são modeladas é crucial para a obtenção de respostas mais precisas e corretas. Essa modelagem envolve a escolha cuidadosa das palavras, a estruturação lógica das instruções e a adaptação do *prompt* ao tipo de tarefa desejada, de modo a maximizar a performance do modelo e garantir resultados mais alinhados com as expectativas do usuário.

2.3 *Test Smells*

Test smells são problemas que podem surgir nos códigos de teste e que afetam negativamente a eficácia, dificultam a compreensão do código de teste e tornam a manutenção mais difícil (Kim, 2020; Santana *et al.*, 2020). Normalmente são identificados como escolhas de design inadequadas na implementação dos casos de testes, podendo ter um impacto significativo na eficácia desses testes (Palomba *et al.*, 2018). Esses *smells* geralmente são introduzidos quando o código de teste é confirmado no repositório pela primeira vez e tende a permanecer no sistema por um longo tempo, dificultando sua manutenibilidade (Tufano *et al.*, 2021). Os *test smells* se manifestam de diversas formas, como falta de estruturação no código de teste e lógica excessivamente complexa. Dessa forma, isso pode resultar em testes longos, duplicação de código e aumento no risco de erros (Breugelmans; Rompaey, 2008; Kim *et al.*, 2021). De acordo com Spadini *et al.* (2018), a presença de *smells* em testes unitários acarreta em códigos de produção mais suscetíveis a problemas não identificados, resultando em um software menos

confiável e com maior risco de defeitos.

Deursen *et al.* (2001) catalogaram 11 tipos de *test smells* em um estudo sobre refatoração de código de teste: *Mystery Guest, Resource Optimism, Test Run War, General Fixture, Eager Test, Lazy Test, Assertion Roulette, Indirect Testing, For Testers Only, Sensitive Equality,* e *Test Code Duplication.* Recentemente, Peruma *et al.* (2019) propuseram mais 12 tipos de *test smells* além daqueles já presentes no estudo anterior: *Conditional Test Logic, Constructor Initialisation, Default Test, Duplicate Assert, EmptyTest, Exception Handling, Ignored Test, Magic Number Test, Redundant Assertion, Redundant Print, Sleepy Test* e *Unknown Test*. Dessa forma, esses problemas foram catalogados de acordo com a sua origem natureza.

A maioria das pesquisas desenvolvidas sobre *test smells* estão focadas em projetos Java (Bavota *et al.*, 2012; Spadini *et al.*, 2018; Tufano *et al.*, 2021). Entretanto, com o crescente uso de Python em softwares de aprendizado de máquina, surgiu a necessidade de investigar essas métricas para testes específicos nessa linguagem (Wang *et al.*, 2022b). O Código-fonte 8 apresentado por Fushihara *et al.* (2023) demonstra um exemplo de teste em Python contendo *test smells*. Alguns *smells* observáveis são: *Assertion Roulette*, devido à falta de clareza nas várias asserções (*expected* e *assertEquals*) dentro do loop *for*; *Magic Number Test*, pelo uso direto de valores numéricos; e *Conditional Test Logic*, com lógica condicional (*for, if, else*) complexa que pode esconder erros e comprometer a cobertura do caso de teste.

Código-fonte 8 – Exemplo de caso de teste em Python com *test smells*

```
1  import unittest
2  class TestFizzBuzz(unittest.TestCase):
3      def test_fizz_buzz(self):
4          for value in range(1,31):
5              if value in [3,6,9,12,18,21,24,27]:
6                  expected = "Fizz"
7              elif value in [5,10,20,25]:
8                  expected = "Buzz"
9              elif value in [15,30]:
10                 expected = "Fizz Buzz"
11             else:
12                 expected = str(value)
13             actual = fizz_buzz(value)
14             self.assertEqual(expected, actual)
```

Fonte: Fushihara *et al.* (2023)

2.3.1 Ferramentas de detecção de test smells

Aljedaani *et al.* (2021) realizaram um mapeamento sistemático sobre as ferramentas de detecção de *test smells* e destacaram que seu uso é essencial para a avaliação da qualidade do código de teste. Estudos recentes desenvolveram ferramentas específicas responsáveis por realizar a detecção de *test smells* e metrificar a qualidade dos testes em Python. Algumas dessas ferramentas de detecção já foram diretamente utilizadas na avaliação da qualidade estrutural de teste unitários, como no estudo de Alves *et al.* (2024). As ferramentas de detecção mais presentes atualmente na literatura são: Pynose[13] e TEMPY[14]:

- **Pynose:** Este plugin integrável à IDE PyCharm detecta *test smells* exibindo avisos diretamente na interface, permitindo que os desenvolvedores monitorem em tempo real se há problemas no código sendo desenvolvido (Wang *et al.*, 2022b).
- **TEMPY:** Projeto de código aberto desenvolvido em Python, executável via IDE, que disponibiliza uma interface simples para selecionar um projeto e gerar um relatório em formato HTML com todos os *test smells* identificados. Essa ferramenta é útil para uma análise detalhada da qualidade dos códigos de teste em projetos Python (Fernandes *et al.*, 2022).

Alguns tipos de *test smells* são amplamente encontrados em diversas linguagens de programação e podem comprometer a qualidade dos testes unitários. Exemplos comuns incluem *Conditional Test Logic*, onde a lógica condicional dentro dos testes pode dificultar a interpretação dos resultados; *Assertion Roulette*, caracterizado pelo uso excessivo de múltiplas asserções sem mensagens explicativas, tornando difícil identificar a causa de falhas; *Empty Test*, que representa testes sem qualquer implementação válida; *Verbose Test*, quando a complexidade e o tamanho excessivo dos testes prejudicam a legibilidade e manutenção; *Programming Paradigms Blend*, que mistura diferentes paradigmas de programação de maneira inconsistente; e *Magic Number Test*, onde valores numéricos arbitrários são usados diretamente, dificultando a compreensão e manutenção do código. Esses *test smells* são detectados por diversas ferramentas automatizadas.

Além desses, existem *test smells* específicos do ecossistema Python devido às características próprias da linguagem na escrita de testes unitários. Entre eles, *Sleepy Test* refere-se a testes que utilizam atrasos desnecessários (*sleep*), impactando a eficiência da execução; *Test Maverick* ocorre quando um teste não pertence a nenhuma estrutura organizacional definida,

[13] https://pypi.org/project/pynose/
[14] https://github.com/danieldavidf/TEMPY

dificultando sua rastreabilidade; *General Fixture* acontece quando um conjunto de dados de teste contém informações irrelevantes para determinados testes, aumentando o custo de manutenção; e *Redundant Print*, caracterizado pelo uso desnecessário de comandos de saída (*print*), poluindo os logs de execução (Wang *et al.*, 2022b; Fernandes *et al.*, 2022).

Enquanto alguns desses *test smells* são identificados por várias ferramentas de detecção, outros são reconhecidos apenas por ferramentas específicas. O Quadro 2 apresenta os diferentes tipos de *test smells* que as duas ferramentas descritas anteriormente são capazes de detectar, juntamente com suas respectivas descrições.

Quadro 2 – Principais *test smells* detectados pelas ferramentas citadas

Test smell	**Descrição**	**Ferramenta que detecta**
Conditional Test Logic	Testes com estruturas de controle (if, for, while...)	Pynose, TEMPY
Assertion Roulette	Vários assertions sem qualquer explicação ou mensagem	Pynose
Magic Number Test	Existência de valores numéricos literais em um teste	Pynose
Exception Handling	Presença da instrução try/except ou da instrução raise	Pynose TEMPY
Sleepy Test	Invoca função time.sleep() no teste	Pynose TEMPY
Redundant Print	Invoca função print() no teste	Pynose TEMPY
Unknow Test	Testes sem nenhum assertion	TEMPY
Non-Functional Statement	Escopo vazio em um método de teste	TEMPY
Verbose Test	Método de teste muito longo	TEMPY
Duplicate Assert	Assertions duplicados em um mesmo teste	Pynose
Test Maverick	Se o conjunto de testes tiver um acessório com setUp, mas um caso de teste nesse conjunto não usar essa configuração	Pynose
Empty Test	Um caso de teste não contém uma única instrução executável	Pynose
Redundant Assertion	Presença de assert cujo resultado nunca muda (ou seja, assert 1 == 1)	Pynose
General Fixture	Nem todos os campos instanciados no método setUp() de um conjunto de testes são utilizados por todos os casos de teste nesse conjunto de testes	Pynose
Lack of Cohesion of Test Cases	Os conjuntos de testes em um caso de teste não são coesos de acordo com a métrica de similaridade de cosseno entre pares	Pynose
Suboptimal Assert	Presença de um dos *suboptimal asserts*	Pynose
Programming Paradigms Blend	Mistura de paradigmas no mesmo arquivo de teste	TEMPY

Fonte: Criado pelo autor.

3 TRABALHOS RELACIONADOS

Ao analisar a literatura existente, foram identificados estudos relevantes que abordam diversas percepções e investigações sobre geração e avaliação da qualidade de código gerados por LLMs. Nesta seção, são apresentados esses estudos, destacando suas contribuições e a forma como se relacionam com o presente trabalho.

3.1 *Assessing the quality of GitHub copilot's code generation*

Yetistiren *et al.* (2022) realizaram uma avaliação da qualidade da geração de código pelo GitHub Copilot, avaliando a correção, validade e eficiência das soluções geradas. O objetivo da pesquisa foi entender o quão eficaz o Copilot é na produção de códigos válidos, corretos e de alta qualidade. Para isso, os pesquisadores definiram três questões de pesquisa: (QP1) qual é a qualidade dos códigos gerados; (QP2) qual é o efeito do uso de *docstrings* na qualidade dos códigos gerados; (QP3) qual é o efeito do uso de parâmetros de funções apropriados na qualidade dos códigos gerados.

A metodologia desta pesquisa consistiu em utilizar o conjunto de dados HumanEval da OpenAI, criado para avaliar a eficácia de LLMs em resolver problemas de programação. Foram gerados códigos de duas maneiras: (i) através da geração automática, onde o Copilot sugeria trechos de código à medida que os programadores escreviam; e (ii) utilizando *prompts* em linguagem natural, onde o Copilot sugeria trechos de código com base nos comandos fornecidos. Todos os códigos gerados estavam na linguagem Python, devido ao maior suporte oferecido pelo modelo de linguagem.

Os resultados deste estudo indicaram que o GitHub Copilot conseguiu gerar códigos válidos na maioria dos casos em relação aos problemas do conjunto de dados HumanEval e demonstrou alta eficiência ao comparar suas soluções com as escritas por humanos. No entanto, apenas um quarto das soluções geradas foram consideradas corretas. O uso de *docstrings* e de parâmetros de entrada impactou diretamente a qualidade e a validade dos códigos gerados, destacando a importância da documentação na geração de códigos por LLMs. Os pesquisadores concluíram que, embora o Copilot seja uma ferramenta promissora para a geração automática de código, ainda há necessidade de avaliações contínuas para explorar ao máximo sua capacidade de gerar código de qualidade.

O estudo de Yetistiren *et al.* (2022) se relaciona diretamente com o presente trabalho

ao analisar as melhores técnicas para avaliar a qualidade dos códigos gerados pelo GitHub Copilot. No entanto, enquanto os autores da pesquisa citada se concentraram na qualidade dos códigos de produção, o foco do presente trabalho será avaliar a qualidade dos códigos de teste gerados.

3.2 *Using GitHub Copilot for Test Generation in Python: An Empirical Study*

Haji *et al.* (2024) conduziram um estudo empírico sobre a geração automática de testes unitários pelo GitHub Copilot. O objetivo principal deste trabalho foi analisar a capacidade do Copilot de gerar testes unitários válidos dentro e fora de um contexto de suíte de testes. Para este estudo, os autores elaboraram quatro questões de pesquisa referentes à usabilidade dos testes gerados dentro de um conjunto de testes existente (QP1) e fora de um conjunto de testes existente (QP2). Adicionalmente, eles analisaram a influência da documentação e dos comentários de código na geração dos testes. Para isso, investigaram a usabilidade dos testes gerados utilizando os comentários e documentações presentes nos métodos de teste, tanto dentro (QP3) quanto fora (QP4) de uma suíte de teste.

Neste trabalho, os autores utilizaram sete projetos de código aberto e empregaram os métodos de teste já existentes como base para o Copilot gerar o código. Para isso, removeram todo o corpo dos testes, deixando apenas o método e os possíveis comentários e documentações existentes. Em seguida, submeteram esses métodos de teste ao LLM para que os testes fossem gerados. Para a análise dos dados, os testes foram classificados em duas categorias: aqueles que faziam parte de suítes de teste e aqueles que incluíam comentários ou documentações. Essa abordagem permitiu aos autores avaliar a influência dos comentários e documentações na qualidade e usabilidade dos testes gerados pelo Copilot, tanto dentro quanto fora do contexto de uma suíte de teste.

Como resultados deste trabalho, os autores destacaram que o GitHub Copilot conseguiu gerar testes válidos em 45,28% dos casos quando os testes possuíam contexto, e apenas em 7,55% dos casos quando não havia contexto. Isso revelou uma diferença significativa na capacidade do LLM de gerar uma quantidade substancialmente maior de testes válidos dentro do contexto de uma suíte de teste. Esses resultados também foram observados em testes que incluíam comentários e documentação; no entanto, notou-se que a quantidade de testes válidos gerados fora de um contexto foi maior. Alguns dos erros mais recorrentes incluíram erros de sintaxe, referências a atributos inexistentes e assertivas incompatíveis.

Embora este trabalho relacionado avalie a capacidade do GitHub Copilot de gerar testes unitários, não aborda a qualidade dos códigos de teste considerados válidos. O presente trabalho também utilizará a metodologia de Haji *et al.* (2024) para a seleção de projetos de código aberto na geração de testes unitários, levando em consideração os aspectos que permitem ao Copilot gerar mais testes válidos. No entanto, de forma complementar, este estudo irá além, analisando a qualidade dos códigos de teste válidos gerados.

3.3 *GitHub Copilot AI pair programmer: Asset or Liability?*

Dakhel *et al.* (2023) exploram afundo a denominação de *"AI pair programmer"* dada ao Github Copilot investigando a qualidade do código que ele gera. A justificativa é investigar qual nível de qualidade o Copilot adicionará aos projetos de software se for usado como um programador de pares de IA. Um dos objetivos específicos deste estudo também é realizar uma investigação comparativa dos códigos do Copilot com códigos escritos por humanos, a fim de determinar se ele pode ser usado no lugar de um desenvolvedor em tarefas de programação em pares de projetos de software sem impactar a qualidade do código.

Primeiro, os autores avaliaram as capacidades do Copilot na resolução de problemas algorítmicos fundamentais de pesquisa e classificação em Python. Também estudaram a correção e reprodutibilidade das soluções do Copilot para esses problemas. Em segundo lugar, os autores compararam as soluções do Copilot com soluções humanas na resolução de tarefas de programação, para avaliar até que ponto este pode imitar o trabalho de um par de programadores humanos. Para isso, usaram um conjunto de dados de diferentes tarefas de programação contendo até 4000 soluções fornecidas por humanos (corretas e com erros).

Os resultados deste estudo mostraram que o Copilot é capaz de gerar soluções corretas para alguns problemas fundamentais no projeto de algoritmos. No entanto, a qualidade do código gerado depende muito da concisão e profundidade do *prompt* fornecido pelo desenvolvedor. Além disso, os resultados também indicaram que o Copilot ainda precisa de mais desenvolvimento na compreensão completa dos enunciados em linguagem natural para poder preencher a posição de um programador par. Na conclusão, os pesquisadores explicaram que ocasionalmente o Copilot pode não conseguir gerar código que atenda a todos os critérios descritos no *prompt*, mas o código gerado pode ser incorporado com alterações pequenas a moderadas no prompt fornecido ou no código.

O presente trabalho também se propõe a avaliar os códigos gerados pelo GitHub

Copilot, utilizando uma metodologia semelhante à empregada por Dakhel *et al.* (2023) na geração automática de código. No entanto, ao contrário dos autores mencionados, nosso foco está em avaliar a qualidade das soluções propostas pelo Copilot especificamente a nível de código de teste. Embora a programação em pares não seja um conceito explorado nesta pesquisa, os resultados obtidos podem fornecer descobertas valiosos para esse aspecto do desenvolvimento.

3.4 *Guiding ChatGPT for Better Code Generation: An Empirical Study*

Liu *et al.* (2024) realizaram um estudo empírico sobre o uso do ChatGPT para a geração de código, com foco na Engenharia de *Prompts* para guiar o modelo na geração de códigos de qualidade. Os autores exploraram diferentes contextos de prompts e utilizaram o *dataset* CodeXGLUE como referência para a geração dos códigos. No estudo, foram extraídos os tipos distintos de contextos para a geração de código: *Code2Code*, para tradução entre diferentes linguagens de programação, e *Text2Code*, para a geração de código a partir de textos em linguagem natural. No caso de *Text2Code*, foram empregados textos detalhados e resumidos, a fim de analisar a capacidade do modelo em lidar com esses dois tipos de *prompt*.

Para avaliar a qualidade dos códigos gerados em cada contexto, os autores utilizaram uma combinação de métricas específicas. A validade e a similaridade dos códigos foram medidas por meio de métricas como BLEU e CodeBLEU, que avaliam a correspondência entre o código gerado e o código de referência presente no *dataset*. Para analisar a estrutura do código, os autores recorreram a métricas adicionais de qualidade, como a detecção de *Code Smells*, que identificam padrões indesejáveis na implementação do código. Esses fatores foram analisados de forma independente para cada contexto e tipo de *prompt* submetido ao ChatGPT. A análise foi conduzida por meio de uma abordagem quantitativa, onde os resultados de cada métrica foram comparados e interpretados separadamente, proporcionando uma visão detalhada da performance do modelo em diferentes cenários de geração de código.

Os resultados do estudo demonstram que a construção e otimização de *prompts* para o ChatGPT podem melhorar significativamente o desempenho na geração de código, tanto em contextos de *Text2Code* quanto de *Code2Code*. A pesquisa revelou que os *prompts* projetados superaram o desempenho de LLMs de última geração, com uma análise de 100 amostras geradas que confirmou a relevância e correção do código produzido. Além disso, as métricas de desempenho, como BLEU e CodeBLEU, mostraram estabilidade em múltiplas execuções, indicando que instruções específicas nos *prompts* limitam a aleatoriedade da geração.

O presente trabalho se relaciona com o de Liu *et al.* (2024), uma vez que também realiza um estudo empírico da geração de código por LLMs em contextos *Text2Code* e *Code2Code*, como também utilizará a métrica CodeBLEU como base para analisar os resultados. No entanto, o presente estudo irá além dessa análise de código de produção e focará em códigos de teste, além de realizar também um estudo comparativo com diversas LLMs.

3.5 *Detecting Test Smells in Python Test Code Generated by LLM: An Empirical Study with GitHub Copilot*

Alves *et al.* (2024) avaliaram a qualidade dos códigos de teste gerados pelo GitHub Copilot em Python, focando na detecção de *test smells*. Os autores destacaram que embora os LLMs sejam cada vez mais utilizados para geração automática de testes unitários, há uma lacuna na avaliação da qualidade desses testes. Para isso, foram analisados os *test smells* presentes nos testes gerados pelo Copilot, verificando a frequência com que ocorrem e a percepção dos profissionais de software sobre esses problemas.

A metodologia adotada envolveu a geração de 194 casos de teste a partir de 30 arquivos de teste em Python, utilizando o Copilot. Esses testes foram analisados com ferramentas de detecção de *test smells* em Python (Pynose, TEMPY e pytest-smell). Além disso, uma avaliação qualitativa foi conduzida com profissionais de desenvolvimento e garantia de qualidade de software, que forneceram percepções sobre os problemas detectados nos testes gerados.

Os resultados indicam que 47,4% dos testes gerados apresentaram pelo menos um *test smell*, sendo o mais comum a falta de documentação nas asserções (Assertion Roulette). Outros problemas identificados incluem uso excessivo de valores numéricos fixos (*Magic Number Test*) e testes sem asserções (*Unknown Test*). A avaliação dos profissionais reforçou essas descobertas e apontou questões adicionais, como baixa legibilidade e falta de uso dos recursos avançados das bibliotecas *pytes* e *unittest*. Esses achados sugerem que, embora o GitHub Copilot gere testes funcionais, ainda há desafios relacionados à qualidade e manutenção desses códigos.

O presente estudo apresenta uma extensão do trabalho de Alves *et al.* (2024), uma vez que avaliará essa qualidade dos testes para vários LLMs e em vários contextos de *prompts*, além de também não focar apenas em *test smells* e expandir a análise para os erros nos testes inválidos.

3.6 Análise Comparativa

O Quadro 3 apresenta uma análise do presente estudo em relação aos quatro trabalhos relacionados detalhados nas seções anteriores, isso visa destacar como o presente trabalho se posiciona no contexto da literatura existente. Para isso, foram utilizados 4 critérios principais: (i) se o trabalho explora técnicas de geração de código por LLMs; (ii) se o estudo realiza alguma medição da qualidade dos códigos utilizando métricas de qualidade; (iii) se o trabalho explora técnicas de geração utilizando contextos de *prompts*;(iv) se o trabalho foca em código de teste. Dessa forma, é possível observar que todos os quatro trabalhos relacionados exploram técnicas de geração de código. No entanto, apenas o estudo de Haji *et al.* (2024) e Alves *et al.* (2024) foca na geração de código de teste. Entre os trabalhos analisados, os estudos de Yetistiren *et al.* (2022), Dakhel *et al.* (2023), Alves *et al.* (2024) e Liu *et al.* (2024) avaliam a qualidade dos códigos gerados, enquanto os demais se limitam à avaliação da validade e às técnicas de geração. Considerando essas variáveis, o diferencial do presente trabalho é combinar a geração de código com a avaliação da qualidade, focando exclusivamente em códigos de teste e explorando diversos contextos de *prompts*.

Quadro 3 – Comparativo entre os trabalhos relacionados com o trabalho proposto.

Trabalhos	**Explora a geração de código por LLMs**	**Avalia a qualidade dos códigos gerados por LLMs**	**Explora diversos contextos (prompts)**	**Foca em códigos de teste**
Este trabalho	**x**	**x**	**x**	**x**
Yetistiren *et al.* (2022)	x	x		
Haji *et al.* (2024)	x		x	x
Liu *et al.* (2024)	x	x	x	
Dakhel *et al.* (2023)	x	x		
Alves *et al.* (2024)	x	x		x

Fonte: Criado pelo autor

.

4 PROCEDIMENTOS METODOLÓGICOS

Neste capítulo serão apresentados os passos necessários para a realização deste trabalho. O estudo foi planejado para investigar a qualidade dos testes unitários gerados pelos LLMs, com foco na detecção de problemas estruturais e padrões de baixa qualidade nos códigos de teste. Para isso, foram estabelecidas quatro questões de pesquisa que orientam a investigação e delimitam os aspectos mais relevantes para análise. O Quadro 4 apresenta essas questões, detalhando os objetivos específicos que cada uma pretende alcançar.

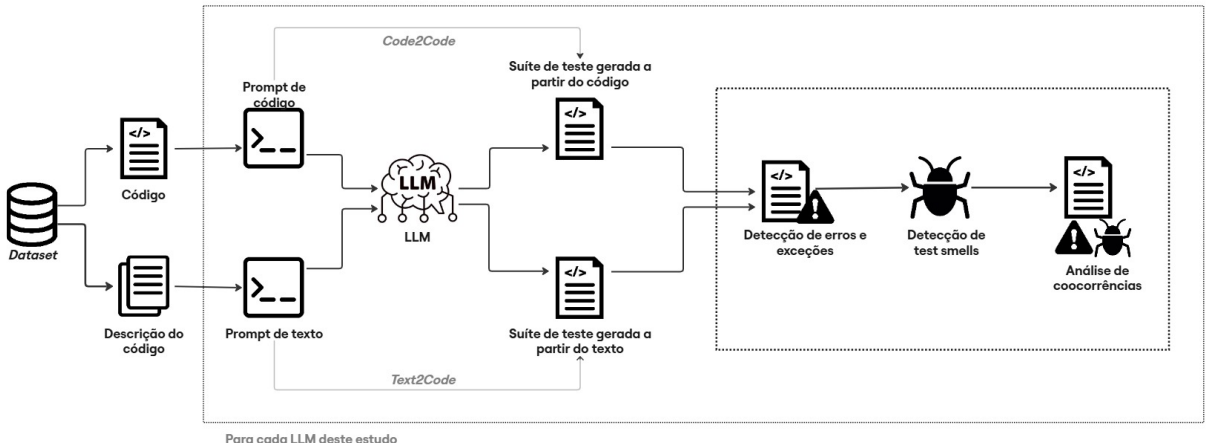
Quadro 4 – Questões de Pesquisa

ID	Questão de Pesquisa	Objetivo
QP1	Quais são os tipos e a frequência de erros relatados nos conjuntos de testes gerados pelos LLMs durante a execução?	Essa questão busca identificar os erros que ocorrem durante a execução dos conjuntos de testes gerados pelos modelos em Python. Os erros podem incluir problemas de sintaxe, exceções de tempo de execução ou falhas na lógica de teste.
QP2	Qual é a distribuição e quais são os tipos específicos de *test smells* identificados nos conjuntos de testes gerados pelos LLMs?	A segunda questão investiga a presença de *test smells* no código. A análise inclui a identificação dos tipos mais comuns e sua distribuição entre os diferentes conjuntos de testes.
QP3	Como o tipo de *prompt* influencia a ocorrência de erros e de *test smells* em conjuntos de testes gerados por LLMs?	Nesta pergunta, o foco está no impacto do tipo de *prompt* usado (T2C ou C2C) sobre a qualidade dos testes gerados. A pergunta avalia se os diferentes estilos de entrada resultam em variações significativas na frequência de erros e nos resultados dos testes.
QP4	Existe uma correlação entre a presença de erros e a detecção de *test smell* no mesmo conjunto de testes gerado por LLMs?	Esta questão explora a correlação entre a presença de erros de execução e os *test smells* no mesmo conjunto de testes. O objetivo é verificar se há um padrão no qual os testes estruturalmente problemáticos (com *test smells*) têm maior probabilidade de apresentar erros.

Fonte: Criado pelo autor.

Para responder às questões de pesquisa, o estudo foi estruturado em seis etapas. A primeira etapa consistiu na seleção do *dataset* com amostras de código-fonte que serviriam de base para a geração dos testes. Em seguida, foram preparados os *prompts* de entrada para orientar os LLMs na produção dos códigos de teste, garantindo um formato padronizado. A terceira etapa envolveu a geração dos testes unitários, aplicando os modelos de linguagem previamente configurados. Após a obtenção dos testes, a quarta etapa focou na análise dos erros presentes nos códigos gerados, identificando falhas que poderiam comprometer sua funcionalidade. A quinta etapa consistiu na detecção de *test smells*, utilizando ferramentas para identificar padrões problemáticos na estrutura dos testes. Por fim, na sexta etapa, foi realizada a análise das coocorrências entre erros e *test smells*, buscando entender a relação entre esses fatores e suas implicações na qualidade dos testes. A Figura 5 ilustra a sequência das etapas e os principais procedimentos adotados neste estudo.

Figura 5 – Fluxo dos procedimentos metodológicos



Fonte: Criado pelo autor.

4.1 Preparação dos *prompts* de entrada com base no *dataset HumanEval*

Primeiramente, foi realizada a extração dos códigos e descrições contidos no *dataset HumanEval*, os quais foram armazenados separadamente para facilitar o processo subsequente. O *HumanEval* foi desenvolvido pela OpenAI para treinar e avaliar a capacidade dos LLMs de gerar código funcional a partir de códigos e descrições em linguagem natural. O *dataset* consiste em uma coleção de problemas de programação, cada um com uma descrição textual, testes, exemplos de entrada e saída. A partir desses elementos, neste estudo foram elaborados dois tipos distintos de *prompts*: um focado em código e outro em texto.

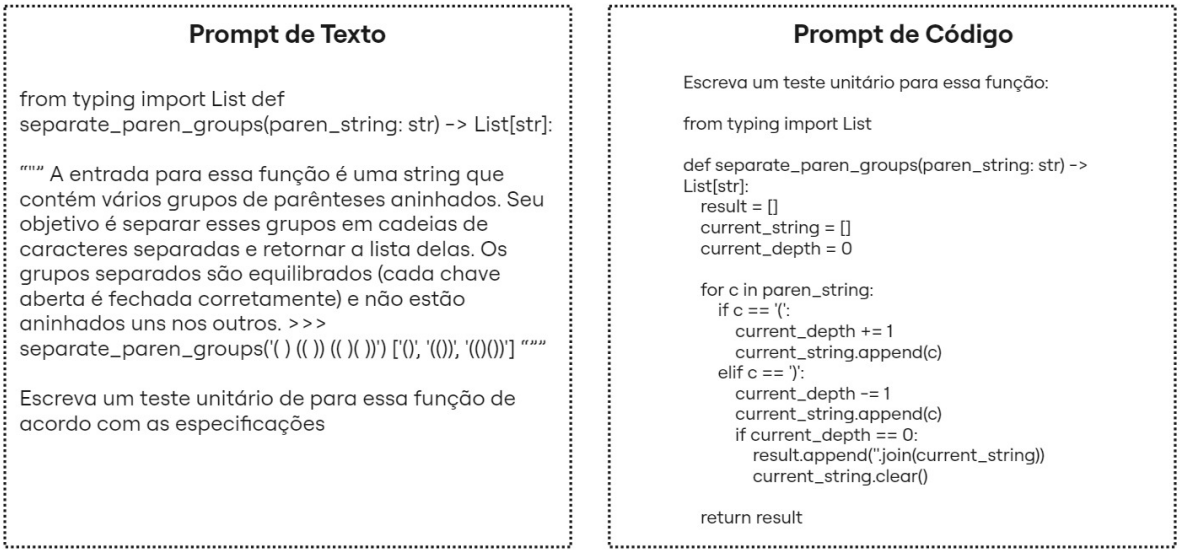
1. ***Prompt* de Código:** Este tipo de *prompt* foi composto majoritariamente pelo código extraído do *HumanEval*. Apenas uma instrução em linguagem natural foi incluída para orientar a geração dos testes, garantindo que o foco principal estivesse na análise do código.
2. ***Prompt* de Texto:** Além da descrição do código fornecida pelo *HumanEval*, o *prompt* textual incluiu uma instrução que guiou a geração do teste. Isso visou explorar a capacidade dos modelos de compreender e trabalhar com descrições textuais mais elaboradas.

A Figura 6 ilustra exemplos de *prompts* criados a partir dos códigos e descrições contidos no *HumanEval*..

4.2 Geração dos códigos de teste utilizando os LLMs

Os *prompts* elaborados no passo anterior, que incluíam tanto descrições em linguagem natural quanto blocos de código extraídos do *dataset HumanEval*, foram submetidos aos

Figura 6 – Exemplo de *prompts* que deverão ser submetidos para os LLMs



Fonte: Criado pelo autor.

LLMs para que estes gerassem testes automatizados a partir das informações fornecidas. Esses testes foram essenciais para avaliar a eficácia e a precisão dos modelos na tarefa de geração de código de teste com base em diferentes tipos de *prompts*. Para este estudo, três LLMss foram utilizados, conforme detalhado na Seção 2.2: LLaMa (desenvolvido pela Meta), versão 3.3 70B; Amazon Q Developer (desenvolvido pela AWS); e GitHub Copilot (desenvolvido pela OpenAI), versão GPT-4o.

4.3 Detecção de erros nos testes

Depois que os conjuntos de testes foram gerados, eles foram executados para verificar se havia erros de interpretação. Durante esse processo, todos os erros relatados pelo Python foram identificados e mapeados. Nessa etapa, a distribuição da detecção de erros foi contada e os tipos de erros foram classificados em relação a cada contexto de *prompt*, considerando os resultados obtidos para cada um dos três LLMs analisados. Para contar os erros e as exceções, inicialmente foi somada todas as detecções identificadas no conjunto geral. Em seguida, foi calculado o total específico para cada tipo de erro identificado em cada LLM, o que permitiu uma avaliação segmentada dos resultados para facilitar a futura análise dos dados extraídos, descrita na Equação 4.1.

$$T_{\text{erros}} = \sum_{i=1}^{m} \text{Erro}_i \tag{4.1}$$

No total, foram contabilizadas 151 detecções de erros nos conjuntos de testes. Depois de contar todos, foi calculada a distribuição percentual dos erros por tipo, conforme descrito na Equação 4.2. O objetivo desse cálculo é determinar a distribuição de erros por tipo com base no total de erros detectados em cada LLM (calculado na Equação 4.1) em relação ao número total de erros identificados em todos os LLMs.

$$Distrib_{\text{erros}} = \frac{T_{\text{erros}}}{\sum_{\text{LLMs}} T_{\text{erros}}} \tag{4.2}$$

4.4 Aplicação de ferramentas de detecção de *test smells* nos testes gerados

Depois que as suítes de teste foram geradas e os erros identificados, foi realizada a detecção de *smells* no código. Para essa etapa, foram selecionadas previamente duas ferramentas específicas para identificar inconsistências e violações de qualidade nos testes de unidade: Pynose(Wang *et al.*, 2022b) e TEMPY(Fernandes *et al.*, 2022). Cada uma adota suas próprias técnicas e abordagens para detectar e categorizar os *test smells*. Para contar os *smells*, primeiro foi somada todas as detecções. Em seguida, para cada LLM, foi aplicado o cálculo da soma individual de detecções por tipo (consulte Equação 4.3), o que permite uma análise detalhada e segmentada dos resultados.

$$T_{\text{smells}} = \sum_{i=1}^{n} \text{Smell}_i \tag{4.3}$$

No total, foram contabilizadas 512 detecções de *smells* nos conjuntos de testes. Depois de contar todas as identificações, foi calculada a distribuição percentual das identificações por tipo (consulte Equação 4.4). O objetivo dessa equação é determinar a distribuição das ocorrências de *test smells* por tipo com base no total de ocorrências detectadas em cada LLM (consulte 4.3) em relação ao número total de ocorrências de *smells* identificadas em todos os LLMs.

$$Distrib_{\text{smells}} = \frac{T_{\text{smells}}}{\sum_{\text{LLMs}} T_{\text{smells}}} \tag{4.4}$$

4.5 Análise de coocorrência

A etapa final do estudo consistiu em analisar a correlação entre os erros detectados e os *test smells* identificados pelas ferramentas. Para isso, foram considerados apenas os códigos que apresentavam simultaneamente erros e *smells* no mesmo conjunto de testes. O número de *test smells* detectados em cada conjunto de testes com um erro foi contado, e foi estabelecida uma relação entre os tipos de *smells* e identificados e os tipos de erros relatados pelo interpretador Python. Com essa abordagem, foi possível avaliar a coocorrência de erros e *test smells*, analisando a frequência e a distribuição de cada tipo nos conjuntos de testes afetados. Para contar as coocorrências, consideramos a soma (consulte a Equação 4.6):

$$k = \sum_{\text{LLMs}} T_{\text{smells}} \tag{4.5}$$

$$Qtd_{\text{co-occurrence}}(S,E) = \sum_{i=1}^{k} \nVdash(S \cap E) \tag{4.6}$$

Em que k é o número total de casos avaliados (número total de *test smells* detectados) e $1(S \cap E)$ é uma função indicadora que retorna 1 se S (*smell*) e E (erro) coincidirem no mesmo caso e 0 caso contrário. Por fim, um total de 512 casos foi analisado, resultando em 265 coocorrências.

5 AVALIAÇÃO DA QUALIDADE DOS TESTES GERADOS POR LLMS

Neste capítulo, serão apresentados os resultados deste estudo, juntamente com as respostas às quatro questões de pesquisa levantadas anteriormente. Durante a análise dos resultados, cada LLM será referido pelo nome de seu modelo original. Portanto, os resultados obtidos com o GitHub Copilot serão associados diretamente ao modelo GPT-4o. Ao final de cada seção de resultados, haverá uma subseção dedicada a responder diretamente à questão de pesquisa relacionada ao resultado apresentado. Foi criado um repositório[1] com todos os dados deste estudo, incluindo testes gerados, arquivos oriundo das ferramentas e *prompts* submetidos. Além disso, todas as planilhas[2] com os dados deste estudo estão disponíveis online.

5.1 Tipos e frequência de erros

Para analisar os erros, todos os conjuntos de testes gerados foram executados juntamente com seus respectivos códigos de produção, extraídos anteriormente do *HumanEval*. Os testes que foram executados com êxito foram classificados como válidos. Aqueles que falharam ou resultaram em exceções levantadas pelo interpretador Python foram registrados como casos de erro. No total, foram identificados 151 erros nos conjuntos de testes analisados. É importante observar que algumas suítes tinham vários erros, e cada ocorrência foi contada separadamente para fins de análise, refletindo a diversidade e a frequência das falhas detectadas.

5.1.1 Resultados gerais da incidência de erros nos testes gerados pelos LLMs

A Tabela 1 mostra a distribuição dos tipos de erros detectados nos conjuntos de testes para os contextos C2C e T2C (coluna C), analisados em cada um dos três LLMs. Em geral, os erros de asserção, denominados *AssertionError* (AE) foram os mais predominantes, representando 64,9% do total. Esse resultado mostra que muitos testes foram gerados com expectativas incorretas em relação ao comportamento do código de produção. O segundo tipo de erro mais frequente foi *IndentationError* (IE), que representou 11,3% das detecções, indicando que várias partes dos testes gerados não seguiram os padrões de indentação exigidos pelo Python, resultando em falhas de execução. O *SyntaxError* (SE) e *KeyError* (KE) tiveram a mesma taxa de 4,6%, sugerindo que algumas suítes tinham erros de sintaxe ou casos de "alucinações",

[1] https://github.com/ovictorpa/llms-research

[2] https://docs.google.com/spreadsheets/d/1X4r1PdtMSGULZfCI1qN5LgARSimqlscUBKBBix0wCxI/edit?usp=sharing

em que os LLMs tentavam acessar variáveis ou funções inexistentes. Outros erros também foram detectados em menor escala: *AttributeError* (ATE) com 4%, *NameError* (NE) com 3,3%, *TypeError* (TE) com 2,6%, *OverflowError* (OE) com 2%, *ZeroDivisionError* (ZE) com 1,3% e, com menor frequência, *IndexError* (IE) e *ValueError* (VE), ambos com 0,7%. A última coluna da Tabela 1 apresenta o percentual total de erros detectados por cada LLM em cada contexto (Equação 4.2), enquanto a última linha expõe a distribuição do total de erros detectados por tipo.

Tabela 1 – Distribuição dos erros nas suítes de testes geradas pelos LLMs

LLM	C	AE	IE	SE	KE	ATE	NE	TE	OE	ZE	IE	VE	TOTAL
GPT-4o	C2C	0,112	0,000	0,286	0,143	0,000	0,000	0,000	0,333	0,000	0,000	1,000	0,106
GPT-4o	T2C	0,082	0,000	0,000	0,000	0,000	0,000	0,000	0,333	0,500	0,000	0,000	0,066
Amazon Q	C2C	0,194	0,235	0,143	0,143	0,000	0,600	0,000	0,333	0,000	0,000	0,000	0,192
Amazon Q	T2C	0,184	0,765	0,286	0,286	0,667	0,400	0,500	0,000	0,000	0,000	0,000	0,285
LLama	C2C	0,276	0,000	0,000	0,143	0,167	0,000	0,500	0,000	0,500	1,000	0,000	0,219
LLama	T2C	0,153	0,000	0,286	0,286	0,167	0,000	0,000	0,000	0,000	0,000	0,000	0,132
% Distribuição		**0,649**	**0,113**	**0,046**	**0,046**	**0,040**	**0,033**	**0,026**	**0,020**	**0,013**	**0,007**	**0,007**	**1,00**

Fonte: Criado pelo Autor.

Ao analisar os desempenhos individuais dos LLMs estudados (conforme apresentado na Tabela 1), observa-se que o modelo GPT-4o apresentou a menor incidência de erros, com uma taxa de 10,6% no contexto C2C e 6,6% no contexto T2C. Notavelmente, GPT-4o também teve a menor diversidade de tipos de erros, com a maior parte das falhas concentradas em dois tipos: *AssertionError* e *SyntaxError*. A partir desses resultados, é possível identificar uma maior precisão e consistência no comportamento do modelo, sugerindo que, embora cometa erros, o modelo tende a ser mais previsível e eficiente na execução dos testes.

Por outro lado, o modelo Amazon Q Developer apresentou uma taxa de erros significativamente mais alta, atingindo 19,2% de detecções no contexto C2C e 28,5% no T2C. Este modelo se destacou por ser o único a registrar casos de *IndentationError*, refletindo problemas com a estruturação do código gerado. Em vários testes, as funções de teste foram declaradas com uma indentação inadequada, o que prejudicou sua execução e gerou falhas. Dessa forma, infere-se que o Amazon Q Developer pode ter dificuldades com a formatação correta de código, especialmente em cenários onde a indentação é crucial para o funcionamento do teste.

Por fim, o modelo LLaMA apresentou um desempenho intermediário, com 21,9% de erros no contexto C2C e 13,2% no T2C. Esses resultados colocam o LLaMA em uma posição equilibrada entre os dois extremos, refletindo uma distribuição mais moderada de erros em comparação com GPT-4o e Amazon Q Developer. A taxa intermediária de falhas sugere que o LLaMA poderia apresentar um equilíbrio entre qualidade e quantidade de erros, o que pode ser

interpretado como uma abordagem mais robusta, porém ainda passível de melhorias.

5.1.2 Exemplo de incidências de erros nos testes gerados pelos LLMs

O Código-fonte 9 apresenta um exemplo de teste com *IndentationError* gerado pelo Amazon Q Developer no contexto T2C. Esse tipo de erro foi exclusivo dos testes gerados por esse modelo. No exemplo da suíte de testes gerada, observa-se uma falha na definição da função de teste *test_example_case(self)* na linha 4, onde o modelo declarou a função, mas não inseriu nenhum conteúdo no seu corpo. Isso resultou em um erro de indentação no Python, já que o código não apresentou continuidade após os dois pontos (:) que indicam o início do método. Esse tipo de erro foi identificado em diversos testes gerados por esse modelo em ambos os contextos.

Código-fonte 9 – Exemplo de suíte de teste gerada contendo erro de identação

```
1  import unittest
2
3  class TestMeanAbsoluteDeviation(unittest.TestCase):
4      def test_example_case(self):
5
6      def test_empty_list(self):
7          with self.assertRaises(ValueError):
8              mean_absolute_deviation([])
9
10     def test_single_element(self):
11         self.assertAlmostEqual(mean_absolute_deviation([5.0]), 0.0)
```

Fonte: Gerado pelo modelo de linguagem Amazon Q Developer para este estudo

O Código-fonte 10, gerado pelo modelo LLaMa no contexto C2C, apresentou um *AttributeError*. Na terceira função de teste (linha 15), o modelo incluiu uma validação para tratamento da exceção *TypeError*, no entanto, esse tratamento não estava presente no código de produção que o modelo estava analisando. Como resultado, ao executar o teste, um *AttributeError* é gerado, pois a função *count_distinct_characters* manipula *strings*, e não valores inteiros. Este caso é um exemplo clássico de "alucinação", pois o modelo pressupôs que o código de produção realizava um tratamento que, na realidade, não existe.

Código-fonte 10 – Exemplo de suíte de teste gerado contendo erro de atributo

```
1  import unittest
2
3  class TestDistinctCharactersFunction(unittest.TestCase):
```

```
4
5       def test_multiple_character_string(self):
6           # Test with a multiple characters string
7           self.assertEqual(count_distinct_characters("hello"), 4)
8
9       def test_duplicate_characters_string(self):
10          # Test with a string containing duplicate characters
11          self.assertEqual(count_distinct_characters("aaaabbbccc"), 3)
12
13      def test_non_string_input(self):
14          # Test that type error is raised when input is not a string
15          with self.assertRaises(TypeError):
16              count_distinct_characters(123)
```

Fonte: Gerado pelo modelo de linguagem LlaMa para este estudo

O Código-fonte 11 apresenta um caso de "alucinação"semelhante ao anterior, mas desta vez gerou um *AssertionError*, detectado na geração T2C do GitHub Copilot. Nesse contexto, o modelo validou o tratamento de exceção *ValueError* (linha 11), que não existe na função *string_xor*. A diferença é que, neste caso, o modelo utilizou o mesmo tipo de atributo aceito pela função, com o objetivo de validar regras específicas do código. Isso resultou em um erro de asserção, indicando que o tratamento de exceção não é acionado no código de produção.

Os cenários apresentados no Código-fonte 10 e Código-fonte 11 evidenciam a semelhança na geração de "alucinações", mesmo sendo originadas por LLMs diferentes. A partir desses resultados, é possível inferir que essas "alucinações"podem levar à ocorrência de diferentes tipos de erros no código, dependendo do contexto em que o teste é realizado.

Código-fonte 11 – Exemplo de suíte de teste gerado contendo erro de asserção

```
1  import unittest
2
3  class TestStringXor(unittest.TestCase):
4      def test_basic_functionality(self):
5          self.assertEqual(string_xor('010', '110'), '100')
6
7      def test_empty_strings(self):
8          self.assertEqual(string_xor('', ''), '')
9
10     def test_different_lengths(self):
11         with self.assertRaises(ValueError):
12             string_xor('01', '001')
```

Fonte: Gerado pelo modelo GPT-4o (Github Copilot) para este estudo

5.1.3 Resposta da QP_1: Quais são os tipos e a frequência de erros relatados nos conjuntos de testes gerados pelos LLMs durante a execução?

Um total de 151 ocorrências de erros foi identificado nos conjuntos de testes gerados pelos LLMs. Entre esses erros, os mais frequentes foram *AssertionError* (64,9%), *IndentationError* (11,3%) e *SyntaxError* (4,6%). Ao analisar os resultados por modelo, GPT-4o apresentou a menor quantidade de erros, enquanto Amazon Q Developer registrou a maior incidência. Notavelmente, alguns tipos de erros, como *IndentationError* e *NameError*, foram exclusivos dos testes gerados pelo Amazon Q Developer. O modelo LLaMA, por outro lado, embora tenha exibido uma taxa intermediária de erros no geral, foi responsável pelo maior número de ocorrências de *AssertionError*. Esses resultados sugerem que determinados modelos podem ser mais confiáveis do que outros na geração de testes válidos. No entanto, a prevalência de erros específicos, como os erros de asserção, pode impactar a precisão e a confiabilidade dos testes gerados, uma vez que esse tipo de falha foi detectado nos testes dos três modelos analisados.

5.2 Tipos e Distribuição de *Test Smells*

Cada conjunto de testes foi analisado usando ferramentas especializadas para detectar *test smells*. Todas as detecções feitas pelas duas ferramentas foram contadas, considerando cada odor identificado individualmente, mesmo quando vários *smells* foram encontrados no mesmo conjunto de testes. Ao consolidar os resultados das duas ferramentas, foram registradas 512 detecções de *test smells*, que serviram de base para a análise deste estudo.

5.2.1 Resultados gerais da incidência de test smells nos testes gerados pelos LLMs

Os resultados da distribuição dos odores de teste são mostrados na Tabela 2. A análise revelou que o *smell* identificado com mais frequência foi *Lack of Cohesion of Test Cases* (LC), responsável por 41,2% de todas as detecções. Isso indica que um número significativo de conjuntos de testes continha casos que não tinham coesão entre si, comprometendo a qualidade estrutural dos testes. Em seguida, o *smell Assertion Roulette* (AR) foi detectado em 24,2% dos casos, o que sugere que muitos testes tinham várias asserções sem documentação ou explicações claras, dificultando a compreensão e a manutenção. O *Programming Paradigms Blend* (PP)

foi o terceiro *test smell* mais frequente, com 15,8% das detecções, indicando que alguns testes usavam campos inicializados fora da classe de teste (no escopo global), o que é considerado uma prática inadequada. Outro *smell* relevante foi o *Suboptimal Assertion* (SA), documentado por Wang *et al.* (2022b) na ferramenta Pynose, que obteve detecções recorrentes. Esse *test smell* reflete asserções que não eram específicas, robustas ou relevantes o suficiente para validar o comportamento esperado do código de produção. Além disso, a Tabela 2 destaca outros *test smells* menos predominantes: *Non-Functional Statement* (NF) e *Unknown Test* (UT) apareceram em 3,7% das detecções, *Conditional Test Logic* (CT) em 2,7%, *Duplicated Assert* (DA) em 1,4% e *Exception Handling* (EH) foi identificado apenas uma vez. Esses resultados reforçam a diversidade e a complexidade dos problemas de design encontrados nos conjuntos de testes gerados pelos LLMs.

Tabela 2 – Distribuição dos *test smells* nas suítes de testes geradas pelos LLMs

LLM	C	LC	AR	PP	SA	NF	CT	UT	DA	EH	TOTAL
GPT-4o	C2C	0,171	0,185	0,012	0,056	0,000	0,000	0,000	0,000	0,000	0,121
GPT-4o	T2C	0,180	0,137	0,432	0,917	0,000	0,000	0,000	0,000	0,000	0,240
Amazon Q	C2C	0,104	0,097	0,099	0,000	0,000	0,000	0,316	0,143	1,000	0,098
Amazon Q	T2C	0,180	0,315	0,444	0,028	1,000	0,143	0,684	0,857	0,000	0,301
LLama	C2C	0,185	0,202	0,012	0,000	0,000	0,357	0,000	0,000	0,000	0,137
LLama	T2C	0,180	0,065	0,000	0,000	0,000	0,500	0,000	0,000	0,000	0,104
% Distribuição		**0,412**	**0,242**	**0,158**	**0,070**	**0,037**	**0,027**	**0,037**	**0,014**	**0,002**	**1,000**

Fonte: Criado pelo Autor.

Na análise por LLM, o Amazon Q Developer apresentou a maior recorrência e diversidade de *test smells*, destacando-se como o modelo com a maior quantidade e distribuição de detecções. Notadamente, alguns *smells* foram exclusivos do código gerado por esse modelo, como *Unknown Test, Non-Functional Statement, Duplicated Assert* e *Exception Handling*, o que demonstra problemas específicos na qualidade dos testes produzidos. Por outro lado, o LLama apresentou o melhor desempenho em termos de qualidade estrutural, com o menor número e variedade de *smells* detectados. Apenas três tipos foram identificados em suas suítes de teste: *Lack of Cohesion of Test Cases, Assertion Roulete* e *Conditional Test Logic*, demonstrando maior consistência na geração de testes com menos violações de qualidade. O GPT-4o ficou entre esses dois extremos, com um volume moderado de detecções e uma baixa diversidade de *smells*. Apenas quatro tipos foram identificados, semelhante ao LLama. No entanto, houve uma concentração significativa do *Suboptimal Assertion*, que teve uma alta frequência de detecções em um dos contextos analisados. Esse padrão sugere que, embora o GPT-4o produza menos variedade de *test smell*, ele pode estar propenso a problemas específicos relacionados à qualidade

das asserções geradas.

5.2.2 Exemplo de incidências de test smells testes gerados pelos LLMs

No conjunto de testes apresentados no Código-fonte 12 gerado pelo GPT-4o no contexto T2C, o *Assertion Roulette* está presente porque os métodos de teste contêm múltiplas asserções sem mensagens explicativas, tornando difícil identificar qual delas falhou em caso de erro. O *Lack of Cohesion of Test Case* ocorre porque os testes avaliam diferentes cenários do comportamento da função, mas não estão claramente agrupados em casos altamente coesos; por exemplo, um único método poderia estar testando tanto números pequenos quanto grandes, reduzindo a clareza da intenção do teste. Já o *Programming Paradigms Blend* pode ser observado na mistura de paradigmas de programação funcional e imperativa dentro do código de teste e da implementação, já que a função *truncate_number* é escrita de forma funcional, mas os testes seguem uma abordagem orientada a objetos com *unittest.TestCase*, o que pode gerar inconsistências no estilo de código.

Código-fonte 12 – Exemplo de suíte de teste gerado contendo AR, LC e PP

```
import unittest

class TestTruncateNumber(unittest.TestCase):
    def test_basic_functionality(self):
        self.assertEqual(truncate_number(3.5), 0.5)
        self.assertEqual(truncate_number(10.75), 0.75)
        self.assertEqual(truncate_number(0.99), 0.99)

    def test_whole_numbers(self):
        self.assertEqual(truncate_number(4.0), 0.0)
        self.assertEqual(truncate_number(100.0), 0.0)

    def test_small_numbers(self):
        self.assertEqual(truncate_number(0.001), 0.001)
        self.assertEqual(truncate_number(0.0001), 0.0001)

    def test_large_numbers(self):
        self.assertEqual(truncate_number(123456.789), 0.789)
        self.assertEqual(truncate_number(987654321.123), 0.123)

if __name__ == '__main__':
    unittest.main()
```

Fonte: Gerado pelo modelo GPT-4o (Github Copilot) para este estudo

O Código-fonte 13 apresenta um dos testes mais simples gerados pelo modelo GPT-4o no contexto T2C. Apesar de sua estrutura minimalista, ele contém um *Suboptimal Assertion* na asserção *self.assertEqual(is_happy("a"), False)*. Esse problema ocorre porque a biblioteca *unittest* oferece um método específico para validar valores booleanos (*assertFalse*), tornando o uso de *assertEqual* redundante e menos expressivo. Esse tipo de estrutura foi recorrente nos testes gerados pelo GPT-4o.

Código-fonte 13 – Exemplo de suíte de teste gerado contendo Suboptimal Assertion

```
import unittest

class TestIsHappy(unittest.TestCase):
    def test_example_1(self):
        self.assertEqual(is_happy("a"), False)

if __name__ == '__main__':
    unittest.main()
```

Fonte: Gerado pelo modelo GPT-4o (Github Copilot) para este estudo

O Código-fonte 14, gerado pelo Amazon Q Developer no contexto T2C, apresenta uma das suítes de teste com o maior número de *test smells* detectados. A diferença estrutural entre os três casos de teste é evidente na análise visual do código, o que levou as ferramentas a identificarem múltiplos *smells* durante a detecção.

O *smell Non-Functional Statement* aparece no comentário dentro do método de teste chamado *test_prime_fib_properties*, que explica a verificação de Fibonacci sem contribuir para a execução do teste. O *Assertion Roulette* ocorre em *test_prime_fib_examples* e *test_prime_fib_properties*, pois há múltiplas asserções sem mensagens claras, dificultando a identificação de falhas. O *Lack of Cohesion of Test Cases* está presente porque *test_prime_fib_properties* mistura verificações de primalidade e pertencimento à sequência de Fibonacci, tornando o teste menos focado. O *Conditional Test Logic* ocorre na asserção composta em *test_prime_fib_properties*, onde o operador lógico or introduz múltiplas possibilidades de sucesso, tornando o teste menos previsível. O *Suboptimal Assertion* está em *test_prime_fib_properties*, pois assertTrue(condition) poderia ser substituído por assertEqual ou assertIn para maior clareza. Por fim, o *Programming Paradigms Blend* está na mistura de estilos imperativo e funcional, pois o código de teste segue um paradigma orientado a objetos *(unittest.TestCase)*, enquanto a lógica matemática utilizada

nas verificações reflete um estilo mais funcional.

Código-fonte 14 – Exemplo de suíte de teste gerado contendo múltiplos test smells

```
import unittest

class TestPrimeFibFunctions(unittest.TestCase):

    def test_prime_fib_examples(self):
        self.assertEqual(prime_fib(1), 2)
        self.assertEqual(prime_fib(2), 3)
        self.assertEqual(prime_fib(3), 5)
        self.assertEqual(prime_fib(4), 13)
        self.assertEqual(prime_fib(5), 89)

    def test_prime_fib_properties(self):
        for i in range(1, 6):
            result = prime_fib(i)
            self.assertTrue(is_prime(result), f"{result} should be prime")
            # Check if it's a Fibonacci number (using the property that a
    number is Fibonacci
            # if and only if one of 5n^2+4 or 5n^2-4 is a perfect square)
            n = result
            self.assertTrue(
                math.isqrt(5*n*n + 4)**2 == (5*n*n + 4) or
                math.isqrt(5*n*n - 4)**2 == (5*n*n - 4),
                f"{result} should be a Fibonacci number"
            )

    def test_is_prime_type_error(self):
        with self.assertRaises(TypeError):
            is_prime(3.14)

if __name__ == '__main__':
    unittest.main()
```

Fonte: Gerado pelo modelo Amazon Q Developer para este estudo

O exemplo a seguir é uma evidência de como em determinados contextos os LLMs inserem lógicas complexas em testes que deveriam ser simples. Código-fonte 15, gerado pelo LLama no contexto C2C, evidencia como o *Conditional Test Logic* pode impactar negativamente a complexidade e a legibilidade do código. A atribuição das variáveis *numbers* e *any_triple_sum_to_zero* envolve uma lógica intrincada, combinando múltiplos *loops* aninhados com condições *if*. Esse padrão de codificação aumenta a complexidade cognitiva do teste,

dificultando sua compreensão e manutenção. Além disso, a presença excessiva de estruturas condicionais dentro da lógica de teste compromete a clareza da intenção do código, tornando mais difícil identificar falhas e interpretar os resultados dos testes.

Código-fonte 15 – Exemplo de suíte de teste gerado contendo lógica complexa

```
import unittest

class TestTriplesSumToZeroFunction(unittest.TestCase):

    def test_large_input(self):
        # Test with a large input
        numbers = [i for i in range(-100, 101)]
        any_triple_sum_to_zero = any((x + y + z == 0) for x in numbers for y
    in numbers for z in numbers if x != y and x != z and y != z)
        self.assertTrue(any_triple_sum_to_zero)

if __name__ == '__main__':
    unittest.main()
```

Fonte: Gerado pelo modelo LLama para este estudo

5.2.3 *Resposta da QP_2: Qual é a distribuição e quais são os tipos específicos de test smells identificados nos conjuntos de testes gerados pelos LLMs?*

Identificou-se um total de 512 detecções de *test smells* nas suítes de teste geradas pelos LLMs. *Lack of Cohesion of Test Cases* foi o *smell* mais frequente (41,2%), seguido por *Assertion Roulette* (24,2%) e *Programming Paradigms Blend* (15,8%). Outros *smells* menos prevalentes incluíram *Non-Functional Statement, Unknown Test, Conditional Test Logic, Duplicated Assert* e *Exception Handling*. Entre os LLMs, o Amazon Q Developer apresentou a maior ocorrência e diversidade de *test smells*, com vários problemas exclusivos de seu código gerado, como *Unknown Test* e *Non-Functional Statement*. Por outro lado, o LLama apresentou a melhor qualidade estrutural, com apenas três tipos de *smells* detectados, o que sugere testes mais coesos e bem estruturados. O GPT-4o apresentou desempenho moderado, com quatro tipos de *test smells* identificados, mas uma concentração notável em *Suboptimal Assertion*, indicando problemas frequentes com a especificidade e a robustez da asserção.

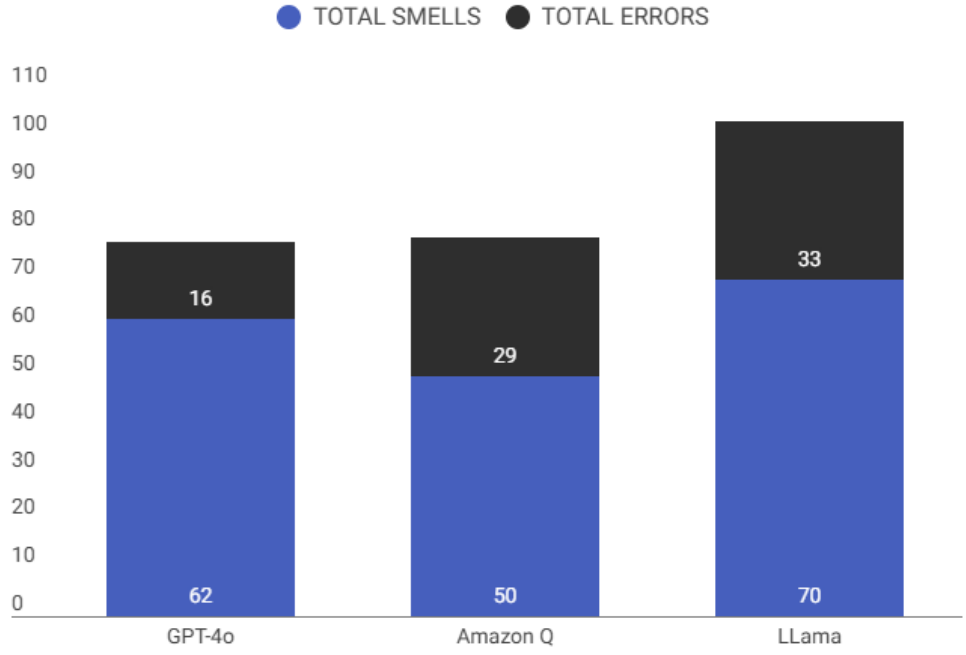
5.3 Influência do contexto de *prompts* na qualidade dos testes

Com base na análise das Tabelas 1 e 2, foi possível observar o impacto dos diferentes tipos de *prompt* na ocorrência de erros e de *test smells* nos conjuntos de testes gerados por cada LLM. Com base nesses dados, foi realizada uma análise detalhada, separando o número total de detecções para cada contexto de *prompt* (*Text2Code* e *Code2Code*), o que permitiu uma compreensão mais profunda da influência do contexto na qualidade das gerações.

5.3.1 Análise do contexto Code2Code

Ao analisar os dados das suítes de teste geradas no contexto C2C (Figura 7), observa-se uma variação moderada na detecção de *test smells*. O modelo Amazon Q teve 50 detecções, enquanto o LLaMA liderou com 70, e o GPT-4o ficou em uma posição intermediária com 62 detecções. Quanto à ocorrência de erros, o GPT-4o se destacou por ter o menor número nesse contexto, enquanto os modelos LLaMA (33) e Amazon Q (29) tiveram incidência semelhante. Esses resultados indicam que, apesar de terem gerado os testes com base no código de produção fornecido no *prompt*, os modelos Amazon Q e LLaMA apresentaram alta ocorrência de erros nos testes gerados, o que demonstra fragilidades na qualidade de suas gerações nesse cenário.

Figura 7 – Quantidade de erros e *test smells* no contexto C2C

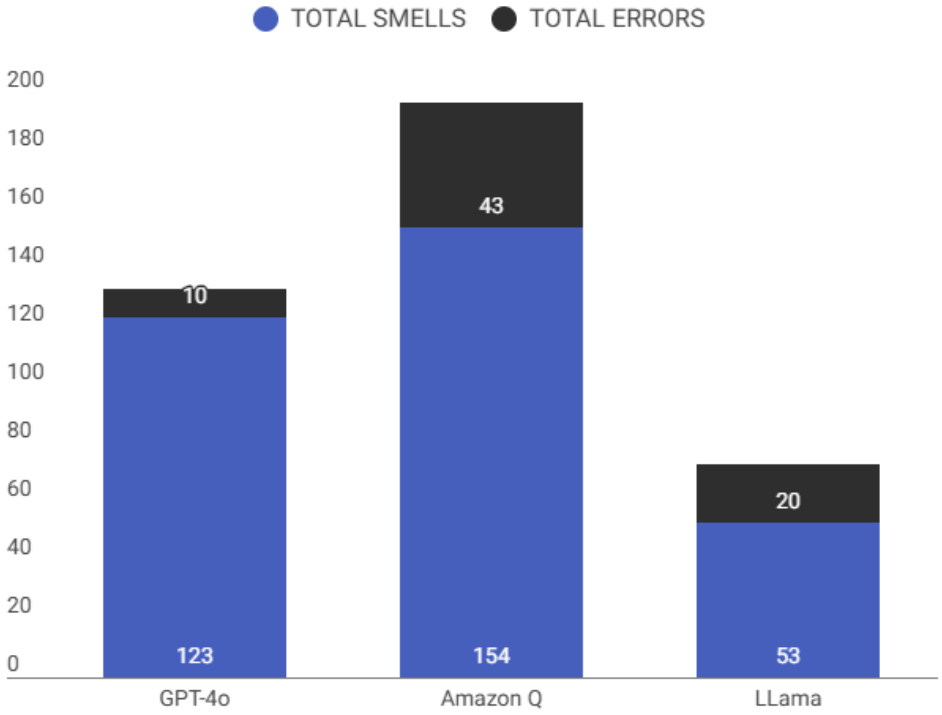


Fonte: Criado pelo Autor.

5.3.2 Análise do contexto Text2Code

No contexto T2C da Figura 8, a incidência de *test smells* no GPT-4o dobrou em comparação com o contexto C2C, chegando a 123 detecções, embora a ocorrência de erros tenha sido menor. O Amazon Q teve um aumento ainda mais acentuado, com o número de *test smells* triplicando (154 detecções) e a incidência de erros subindo para 43. Por outro lado, o LLaMA apresentou uma melhoria significativa em comparação com o contexto C2C, com uma redução tanto no número de *test smells* quanto no número de erros. Esses resultados indicam que o LLaMA demonstrou maior capacidade de gerar testes de qualidade a partir de descrições textuais, superando o Amazon Q. Além disso, o Amazon Q enfrentou dificuldades críticas na interpretação do texto para gerar código, com alguns dos problemas relacionados a erros no código produzido. No caso do GPT-4o, o principal desafio no contexto do T2C foi a alta incidência de *smells* na estrutura dos códigos gerados. Entretanto, o número de erros permaneceu baixo, indicando uma estrutura funcional, mas pouco otimizada.

Figura 8 – Quantidade de erros e *test smells* no contexto T2C



Fonte: Criado pelo Autor.

5.3.3 Resposta da QP_3: Como o tipo de promp influencia a ocorrência de erros e de test smells em conjuntos de testes gerados por LLMs?

Os resultados dessa análise mostram a individualidade de cada LLM em relação aos dois contextos de prompt analisados. Não foi possível identificar um padrão consistente entre os três modelos em termos da qualidade dos conjuntos de testes gerados, pois cada LLM apresentou comportamentos diferentes. O GPT-4o se destacou como o modelo mais estável entre os dois contextos, com uma baixa incidência de erros em ambos os *prompts*. No entanto, no contexto T2C, houve um aumento significativo no número de *testt smells*, o que sugere que o modelo estrutura melhor os testes por ter acesso direto ao código de produção no *prompt*. O Amazon Q enfrentou grandes dificuldades no contexto T2C, com uma alta incidência de erros e *test smells*. Isso indica que o modelo é mais confiável para gerar testes quando pode visualizar o código diretamente, como no contexto C2C. Por outro lado, o LLaMA teve um desempenho melhor ao gerar testes no contexto T2C, com uma redução considerável de erros e de *test smells* em comparação com o contexto C2C. Esse comportamento indica que o modelo é mais capaz de interpretar descrições textuais e gerar testes de qualidade a partir delas.

5.4 Correlação entre Erros e *Test Smells*

Para correlacionar a ocorrência de erros com *test smells*, contabilizamos todas as detecções de *smells* e identificamos aquelas diretamente associadas a erros. Essa análise revelou que, das 512 detecções de *test smells*, 265 foram encontradas em conjuntos de testes que exibiam algum tipo de erro. A Figura 9 demonstra visualmente as relações entre os *smells* detectados e seus erros associados. Esses resultados representam as detecções agregadas em todos os três LLMs analisados neste estudo.

Lack of Cohesion of Test Cases foi o *test smell* mais frequente, com 131 ocorrências, e está fortemente associado ao *AssertionError*, que totalizou 138 ocorrências. Essa relação indica que suítes de teste com baixa coesão frequentemente falham em validar corretamente o comportamento esperado do código de produção. Além disso, esse *test smell* também apresentou conexões com erros como *IndentationError* e *SyntaxError*, evidenciando problemas estruturais e de formatação em testes com casos não coesos. *Assertion Roulette*, identificado 40 vezes, esteve majoritariamente ligado ao *AssertionError*, ressaltando que testes com múltiplas asserções sem explicações claras aumentaram consideravelmente a probabilidade de expectativas incorretas. Por

outro lado, o *Programming Paradigms Blend*, com 50 detecções, mostrou uma ampla dispersão de conexões, incluindo erros como *IndentationError* e *TypeError*. Isso sugere que a mistura de paradigmas nos testes pode comprometer sua estrutura e execução lógica.

Unknown Test, com 17 ocorrências, destacou-se por sua forte associação com *IndentationError*. Essa conexão indica que muitos testes classificados como desconhecidos apresentaram sérios problemas de formatação, relacionados ao fato de o Amazon Q Developer não estruturar o código adequadamente (como discutido na RQ_3). Outros *test smells*, como *Non-Functional Statement* (14 ocorrências), possuem múltiplas conexões com erros como *KeyError* e *NameError*. Esses resultados indicam que testes mal definidos ou não funcionais frequentemente utilizam elementos inexistentes ou inválidos, causando falhas críticas. Além disso, *Duplicate Assert* e *Conditional Test Logic*, embora menos frequentes, estão associados a *AssertionError*, evidenciando que problemas com lógica condicional ou asserções duplicadas impactam diretamente a eficácia dos testes.

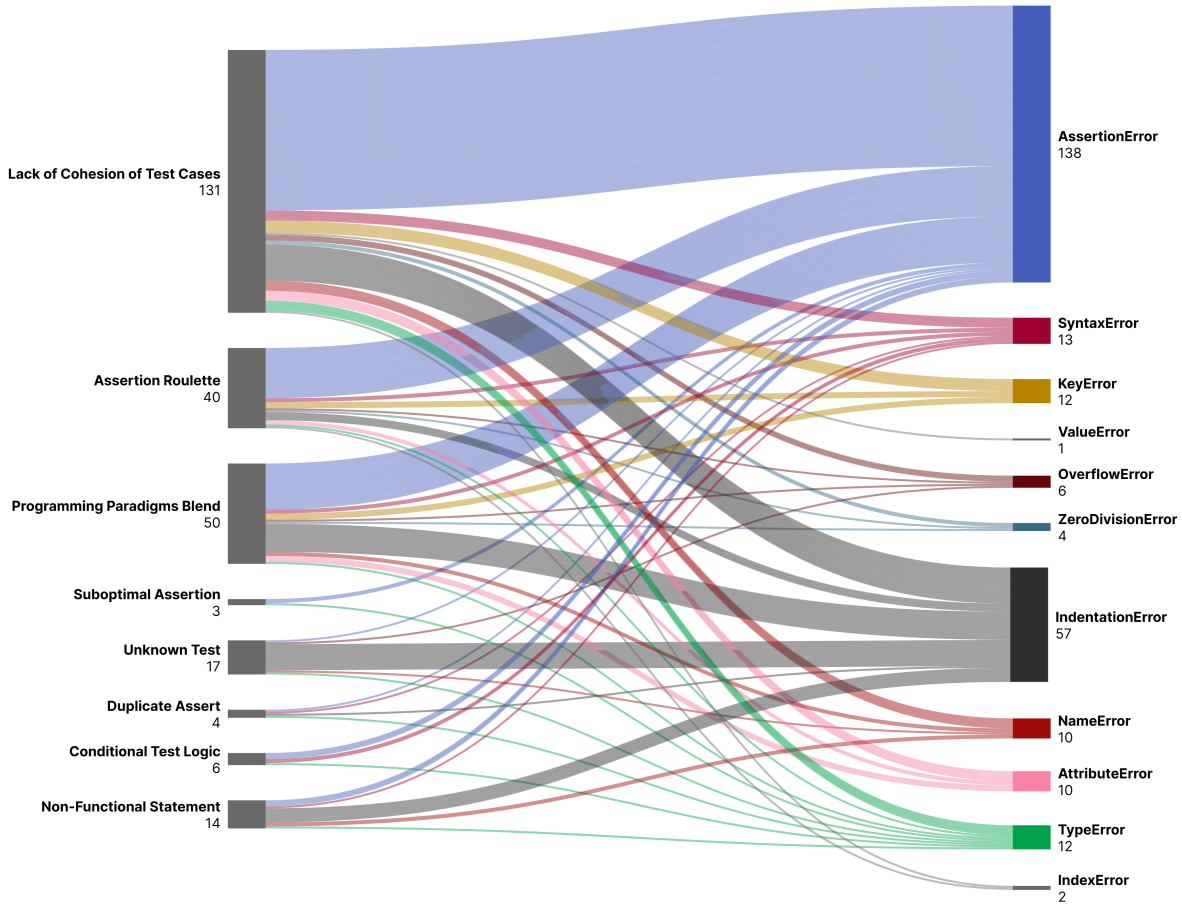
Os erros menos recorrentes, como *OverflowError*, *ZeroDivisionError*, *IndexError* e *ValueError*, aparecem com menor frequência, mas mantêm conexões com diversos *test smells*. Isso sugere que falhas específicas podem ser causadas por uma combinação de problemas estruturais e lógicos. Por fim, é importante destacar que todos os *test smells* detectados estão direta ou indiretamente ligados ao *AssertionError*, reforçando que a validação incorreta é um problema crítico nas suítes de teste geradas por LLMs.

5.4.1 Resposta da QP_4: Existe uma correlação entre a presença de erros e a detecção de test smell no mesmo conjunto de testes gerado por LLMs?

A análise revelou que 265 das 512 detecções de *test smells* ocorreram em testes com erros, evidenciando uma forte relação entre problemas estruturais e falhas nos testes gerados por LLMs. *Lack of Cohesion of Test Cases* foi o *test smell* mais frequente, apresentando uma forte associação com *AssertionError*, o que indica que testes com baixa coesão frequentemente falham na validação do código. Além disso, erros como *IndentationError* e *SyntaxError* foram comuns em testes não coesos, apontando falhas estruturais. *Assertion Roulette* também aumentou a incidência de *AssertionError*, demonstrando que múltiplas asserções sem contexto levam a expectativas incorretas.

A mistura de paradigmas nos testes gerou uma variedade de erros, afetando tanto a estrutura quanto a lógica dos testes. Testes mal definidos ou com lógica condicional problemática

Figura 9 – Coocorrências de erros e *test smells* por tipo



Fonte: Criado pelo Autor.

também contribuíram para falhas críticas. De modo geral, a predominância de *AssertionError* na maioria dos *test smells* reforça que a validação inadequada é um problema central nos testes gerados por LLMs, comprometendo sua confiabilidade.

5.5 Discussão e Implicações

Os resultados deste estudo têm implicações diretas para a indústria de testes de software, especialmente para equipes que adotam metodologias ágeis e práticas como o Desenvolvimento Orientado a Testes (TDD). A análise da qualidade dos testes gerados por LLMs nos contextos *Code2Code* (C2C) e *Text2Code* (T2C) revela desafios específicos para diferentes perfis de desenvolvedores e abordagens de escrita de testes.

No cenário em que os desenvolvedores escrevem testes diretamente a partir do código (C2C), como nas abordagens tradicionais de TDD, a geração automática de testes pode acelerar o ciclo de desenvolvimento. No entanto, os resultados indicam que os LLMs ainda

apresentam limitações na produção de testes coesos e bem estruturados, com alta incidência de *test smells*, como *Lack of Cohesion of Test Cases* e *Assertion Roulette*. Isso significa que a revisão manual ainda é essencial para garantir a qualidade dos testes gerados, uma vez que erros como *AssertionError* e *IndentationError* foram recorrentes, exigindo refinamento antes de sua adoção em um ambiente real.

Por outro lado, desenvolvedores que escrevem testes com base em descrições textuais (T2C), como equipes que seguem o Desenvolvimento Orientado a Comportamento (BDD) ou documentam requisitos antes da implementação, podem enfrentar desafios distintos. Os achados indicam que os LLMs tendem a produzir testes mais propensos a erros estruturais e com alta incidência de *test smells*, especialmente *Suboptimal Assertions*. Para equipes que priorizam documentação em linguagem natural, o uso de LLMs pode acelerar a criação de testes, mas exige estratégias para mitigar a baixa especificidade das asserções geradas, garantindo verificações mais robustas.

Além disso, foram observadas diferenças no desempenho dos LLMs nos diferentes cenários analisados. O GPT-4o apresentou a menor taxa de erros e a maior consistência na geração de testes, tornando-se uma opção viável para equipes que buscam maior confiabilidade na automação de testes. No entanto, exibiu alta incidência de *Suboptimal Assertions*, o que pode impactar equipes que utilizam TDD ou BDD, pois exigiria ajustes nas verificações de cada teste gerado. O Amazon Q Developer, por sua vez, apresentou uma alta recorrência de erros estruturais, como *IndentationError* e *NameError*, além de diversos *test smells* exclusivos. Isso sugere que sua aplicação pode ser mais limitada em cenários que exigem testes bem estruturados. Já o LLama teve um desempenho intermediário, com um número moderado de erros, mas se destacou negativamente por apresentar a maior quantidade de *AssertionError*. Por outro lado, esse modelo teve um bom desempenho na geração de testes a partir de texto (T2C), o que pode torná-lo uma escolha viável para desenvolvedores que trabalham com documentação ou BDD.

A adoção de LLMs para a geração automática de testes precisa ser acompanhada por boas práticas, como revisão manual, engenharia de prompts mais refinada e integração com ferramentas de análise estática. Empresas que utilizam metodologias ágeis podem se beneficiar dessas tecnologias para acelerar a escrita de testes, mas devem considerar os riscos associados à confiabilidade dos testes gerados, garantindo que a automação contribua, de fato, para a melhoria contínua da qualidade do software.

5.6 Ameaças à Validade

- **Validade da Conclusão:** Embora os testes tenham sido gerados a partir do conjunto de dados HumanEval, a diversidade de cenários pode não ser ampla o suficiente para capturar todas as possíveis falhas e test smells que os LLMs podem produzir. A detecção de erros e test smells isoladamente pode não ser suficiente para determinar com precisão a qualidade dos testes gerados. Além disso, a análise desses problemas depende de ferramentas automatizadas e da linguagem utilizada, as quais possuem limitações e podem não identificar todas as inconsistências presentes nos testes.
- **Validade Interna:** A variabilidade na geração de código pelos LLMs pode influenciar a reprodutibilidade dos testes, especialmente se os modelos forem atualizados no futuro. Além disso, possíveis diferenças nos parâmetros utilizados pelos LLMs podem ter impactado a qualidade dos testes gerados, dificultando a garantia de que os resultados sejam completamente atribuíveis aos tipos de prompts analisados.
- **Validade de Construção:** Foram utilizadas ferramentas específicas para a detecção de test smells (Pynose e TEMPY), mas elas podem não cobrir todos os smells presentes em testes automatizados. Além disso, a categorização dos erros foi baseada nas mensagens do interpretador Python, o que pode levar a diferentes interpretações sobre a criticidade dos problemas encontrados.
- **Validade Externa:** Os resultados podem não ser generalizáveis para outros domínios de software, pois o estudo foi conduzido exclusivamente com código e testes em Python. Além disso, apenas três LLMs foram analisados (GPT-4o, Amazon Q Developer e LLama 3.3), o que pode limitar a aplicabilidade dos achados a outros modelos de geração de código. Os códigos de produção presentes no HumanEval abrangem problemas específicos de programação, frequentemente utilizados para treinar LLMs. Assim, os resultados obtidos podem não ser generalizáveis para outros domínios de código fora desse escopo, limitando a aplicabilidade dos achados a contextos mais complexos ou diversificados.

6 CONCLUSÕES E TRABALHOS FUTUROS

A qualidade estrutural dos códigos de teste gerados por LLMs é um tópico pouco abordado e que deve ser estudado de forma aprofundada (Alves *et al.*, 2024). Este trabalho apresenta um estudo comparativo sobre a qualidade dos testes gerados por três LLMs (GPT-4o, Amazon Q Developer e LLama 3.3). O conjunto de dados *HumanEval* foi utilizado para gerar testes a partir de *prompts* nos contextos *Text2Code* e *Code2Code*. Após a geração, as suítes de teste foram executadas em conjunto com o código de produção para verificar a ocorrência de erros. Em seguida, ferramentas de detecção de *test smells* foram aplicadas para avaliar a qualidade estrutural dos testes gerados. Além disso, foi analisada a correlação entre a presença de erros e *test smells* dentro da mesma suíte de testes.

Os resultados destacam a influência do tipo de *prompt* e do modelo de LLM na qualidade das suítes de teste geradas. A análise revelou que os erros mais frequentes foram *AssertionError, IndentationError* e *SyntaxError*, com variações significativas entre os modelos avaliados. O GPT-4o apresentou a menor taxa de erros, enquanto o Amazon Q Developer foi o modelo mais propenso a gerar testes com falhas estruturais, incluindo *IndentationError* e *NameError*, sendo este último exclusivo desse modelo. Além disso, os *test smells* mais prevalentes foram *Lack of Cohesion of Test Cases* e *Assertion Roulette*, indicando que muitas suítes geradas por LLMs apresentam baixa coesão entre os testes e múltiplas asserções sem contextualização adequada. A correlação entre erros e *test smells* demonstrou que problemas estruturais impactam diretamente a confiabilidade dos testes automatizados.

Apesar dos avanços na geração automática de testes, os achados indicam que os LLMs ainda enfrentam desafios para produzir suítes de alta qualidade, principalmente devido à recorrência de erros e violações de boas práticas de teste.

6.1 Principais Contribuições

- Investigação detalhada da qualidade dos testes gerados por diferentes LLMs, considerando dois contextos distintos de *prompting (Text2Code e Code2Code)*.
- Identificação dos erros mais comuns em testes gerados automaticamente e suas associações com *test smells*, fornecendo evidências sobre os principais desafios estruturais desses testes.
- Comparação do desempenho de três modelos de LLM, destacando suas limitações e tendências na geração de testes automatizados.

- Análise da relação entre *test smells* e falhas nos testes, demonstrando como problemas estruturais impactam diretamente a confiabilidade da validação automatizada.
- Direcionamento de futuras pesquisas e práticas na área de geração automática de testes, sugerindo melhorias no uso de LLMs para esse propósito.

6.2 Trabalhos futuros

- Avaliação de modelos adicionais, incluindo variações do GPT, Claude, Gemini e outros LLMs especializados em código, para verificar se os padrões de erros e *test smells* se mantêm ou variam entre diferentes arquiteturas.
- Utilização de datasets mais diversificados além do *HumanEval*, incluindo repositórios de código aberto (e.g., CodeXGLUE, BigCode) para testar a aplicabilidade dos resultados em cenários mais complexos e realistas.
- Estudo sobre a evolução dos testes gerados por LLMs ao longo do ciclo de vida do software, avaliando sua manutenção e eficácia conforme o código de produção sofre modificações.
- Investigação de como LLMs podem ser incorporados em ferramentas de CI/CD e metodologias ágeis (e.g., TDD e BDD), otimizando a geração e execução automatizada de testes dentro do fluxo de desenvolvimento.
- Proposta de métodos automatizados para corrigir falhas nos testes gerados, utilizando LLMs para refatoração e aprimoramento dos casos de teste detectados como problemáticos.

REFERÊNCIAS

AGARWAL, M.; SHEN, Y.; WANG, B.; KIM, Y.; CHEN, J. **Structured Code Representations Enable Data-Efficient Adaptation of Code Language Models**. ArXiv, 2024. Disponível em: https://arxiv.org/abs/2401.10716. Acesso em: 06 nov. 2024.

ALJEDAANI, W.; PERUMA, A.; ALJOHANI, A.; ALOTAIBI, M.; MKAOUER, M. W.; OUNI, A.; NEWMAN, C. D.; GHALLAB, A.; LUDI, S. **Test Smell Detection Tools:** a systematic mapping study. Proceedings of the 25th International Conference on Evaluation and Assessment in Software Engineering, p. 170–180, 2021. Disponível em: https://doi.org/10.1145/3463274.3463335. Acesso em: 11 jan. 2025.

ALMASI, M. M.; HEMMATI, H.; FRASER, G.; ARCURI, A.; BENEFELDS, J. **An Industrial Evaluation of Unit Test Generation:** finding real faults in a financial application. 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP), p. 263–272, 2017. Disponível em: https://ieeexplore.ieee.org/document/7965450. Acesso em: 27 jan. 2025.

ALVES, V.; SANTOS, C.; BEZERRA, C.; MACHADO, I. **Detecting Test Smells in Python Test Code Generated by LLM:** an empirical study with github copilot. Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software, p. 581–587, 2024. ISSN 0000-0000. Disponível em: https://sol.sbc.org.br/index.php/sbes/article/view/30398. Acesso em: 19 jan. 2025.

ANICHE, M. **Test-Driven Development:** teste e design no mundo real com .net. Casa do Código, 2014. Disponível em: https://books.google.com.br/books?id=6G2CCwAAQBAJ. Acesso em: 01 fev. 2025.

BAVOTA, G.; QUSEF, A.; OLIVETO, R.; LUCIA, A. D.; BINKLEY, D. In: **An empirical analysis of the distribution of unit test smells and their impact on software maintenance**. 28th IEEE International Conference on Software Maintenance (ICSM), 2012. p. 56–65. Disponível em: https://ieeexplore.ieee.org/document/6405253. Acesso em: 11 jan. 2025.

BAYS, J. **AWS Announces Amazon CodeWhisperer (Preview)**. 2022. Disponível em: https://aws.amazon.com/about-aws/whats-new/2022/06/aws-announces-amazon-codewhisperer-preview/. Acesso em: 05 dez. 2024.

BECKER, B. A.; DENNY, P.; FINNIE-ANSLEY, J.; LUXTON-REILLY, A.; PRATHER, J.; SANTOS, E. A. **Programming Is Hard - Or at Least It Used to Be:** educational opportunities and challenges of ai code generation. Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, p. 500–506, 2023. Disponível em: https://doi.org/10.1145/3545945.3569759. Acesso em: 01 fev. 2025.

BELLER, M.; GOUSIOS, G.; PANICHELLA, A.; ZAIDMAN, A. In: **When, how, and why developers (do not) test in their IDEs**. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015. (ESEC/FSE 2015), p. 179–190. ISBN 9781450336758. Disponível em: https://doi.org/10.1145/2786805.2786843. Acesso em: 19 nov. 2024.

BREUGELMANS, M.; ROMPAEY, B. V. **TestQ:** exploring structural and maintenance characteristics of unit test suites. WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques., 2008. Disponível em: https://api.semanticscholar.org/CorpusID:9839569. Acesso em: 31 jan. 2025.

CAIN, W. **Prompting Change:** exploring prompt engineering in large language model ai and its potential to transform education. TechTrends, v. 68, p. 47–57, 2024. Disponível em: https://doi.org/10.1007/s11528-023-00896-0. Acesso em: 15 jan. 2025.

CHEN, M.; TWOREK, J.; JUN, H.; YUAN, Q.; PINTO, H. P. de O.; KAPLAN, J.; EDWARDS, H.; BURDA, Y.; JOSEPH, N.; BROCKMAN, G.; RAY, A.; PURI, R.; KRUEGER, G.; PETROV, M.; KHLAAF, H.; SASTRY, G.; MISHKIN, P.; CHAN, B.; GRAY, S.; RYDER, N.; PAVLOV, M.; POWER, A.; KAISER, L.; BAVARIAN, M.; WINTER, C.; TILLET, P.; SUCH, F. P.; CUMMINGS, D.; PLAPPERT, M.; CHANTZIS, F.; BARNES, E.; HERBERT-VOSS, A.; GUSS, W. H.; NICHOL, A.; PAINO, A.; TEZAK, N.; TANG, J.; BABUSCHKIN, I.; BALAJI, S.; JAIN, S.; SAUNDERS, W.; HESSE, C.; CARR, A. N.; LEIKE, J.; ACHIAM, J.; MISRA, V.; MORIKAWA, E.; RADFORD, A.; KNIGHT, M.; BRUNDAGE, M.; MURATI, M.; MAYER, K.; WELINDER, P.; MCGREW, B.; AMODEI, D.; MCCANDLISH, S.; SUTSKEVER, I.; ZAREMBA, W. In: **Evaluating Large Language Models Trained on Code**. CoRR, 2021. abs/2107.03374. Disponível em: https://arxiv.org/abs/2107.03374. Acesso em: 19 nov. 2024.

DAKA, E.; CAMPOS, J.; FRASER, G.; DORN, J.; WEIMER, W. In: **Modeling readability to improve unit tests**. Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015. (ESEC/FSE 2015), p. 107–118. ISBN 9781450336758. Disponível em: https://doi.org/10.1145/2786805.2786838. Acesso em: 01 dez. 2024.

DAKA, E.; FRASER, G. In: **A Survey on Unit Testing Practices and Problems**. IEEE 25th International Symposium on Software Reliability Engineering, 2014. p. 201–211. Disponível em: https://ieeexplore.ieee.org/document/6982627. Acesso em: 13 nov. 2024.

DAKHEL, A. M.; MAJDINASAB, V.; NIKANJAM, A.; KHOMH, F.; DESMARAIS, M. C.; JIANG, Z. M. J. **GitHub Copilot AI pair programmer:** asset or liability? Journal of Systems and Software, v. 203, p. 111734, 2023. ISSN 0164-1212. Disponível em: https://www.sciencedirect.com/science/article/pii/S0164121223001292. Acesso em: 06 dez. 2024.

DEURSEN, A. van; MOONEN, L.; BERGH, A. van den; KOK, G. **Refactoring Test Code**. Proceedings 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), may 2001. Disponível em: https://dl.acm.org/doi/10.5555/869201. Acesso em: 01 fev. 2025.

FAN, A.; GOKKAYA, B.; HARMAN, M.; LYUBARSKIY, M.; SENGUPTA, S.; YOO, S.; ZHANG, J. M. **Large Language Models for Software Engineering:** survey and open problems. IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE), p. 31–53, may 2023. Disponível em: https://doi.ieeecomputersociety.org/10.1109/ICSE-FoSE59343.2023.00008. Acesso em: 01 fev. 2025.

FERNANDES, D.; MACHADO, I.; MACIEL, R. **TEMPY:** test smell detector for python. Proceedings of the XXXVI Brazilian Symposium on Software Engineering, p. 214–219, 2022. Disponível em: https://doi.org/10.1145/3555228.3555280. Acesso em: 13 dez. 2024.

FRASER, G.; ARCURI, A. **EvoSuite:** automatic test suite generation for object-oriented software. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, p. 416–419, 2011. Disponível em: https://doi.org/10.1145/2025113.2025179. Acesso em: 01 fev. 2025.

FRASER, G.; ARCURI, A. **Whole Test Suite Generation**. IEEE Transactions on Software Engineering, v. 39, n. 2, p. 276–291, 2013. Disponível em: https://ieeexplore.ieee.org/document/6152257. Acesso em: 27 jan. 2025.

FRASER, G.; STAATS, M.; MCMINN, P.; ARCURI, A.; PADBERG, F. **Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study**. ACM Trans. Softw. Eng. Methodol., v. 24, n. 4, sep 2015. ISSN 1049-331X. Disponível em: https://doi.org/10.1145/2699688. Acesso em: 28 dez. 2024.

FUSHIHARA, Y.; AMAN, H.; AMASAKI, S.; YOKOGAWA, T.; KAWAHARA, M. **A Trend Analysis of Test Smells in Python Test Code Over Commit History**. 2023 49th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), p. 310–314, 2023. Disponível em: https://ieeexplore.ieee.org/document/10371608/. Acesso em: 31 jan. 2025.

GONZALEZ, D.; SANTOS, J. C.; POPOVICH, A.; MIRAKHORLI, M.; NAGAPPAN, M. **A Large-Scale Study on the Usage of Testing Patterns That Address Maintainability Attributes:** patterns for ease of modification, diagnoses, and comprehension. IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), p. 391–401, 2017. Disponível em: https://ieeexplore.ieee.org/document/7962388/. Acesso em: 13 nov. 2024.

GRAHAM, D.; BLACK, R.; VEENENDAAL, E. van. **Foundations of Software Testing ISTQB Certification, 4th edition**. Cengage Learning, 2021. ISBN 9780357884157. Disponível em: https://books.google.com.br/books?id=mOwxEAAAQBAJ. Acesso em: 01 fev. 2025.

HAJI, K. E.; BRANDT, C.; ZAIDMAN, A. **Using GitHub Copilot for Test Generation in Python:** an empirical study. Association for Computing Machinery, New York, NY, USA, p. 45–55, 2024. Disponível em: https://doi.org/10.1145/3644032.3644443.

HANSSON, E.; ELLRéUS, O. **Code Correctness and Quality in the Era of AI Code Generationt**: Examining chatgpt and github copilot. Tese (Doutorado), 2023. Disponível em: https://urn.kb.se/resolve?urn=urn:nbn:se:lnu:diva-121545. Acesso em: 19 nov. 2024.

HUANG, Y.; LI, Y.; WU, W.; ZHANG, J.; LYU, M. R. **Your Code Secret Belongs to Me:** neural code completion tools can memorize hard-coded credentials. Association for Computing Machinery, New York, NY, USA, v. 1, n. FSE, jul 2024. Disponível em: https://doi.org/10.1145/3660818. Acesso em: 05 dez. 2024.

IMAI, S. **Is GitHub copilot a substitute for human pair-programming?** an empirical study. Association for Computing Machinery, New York, NY, USA, p. 319–321, 2022. Disponível em: https://doi.org/10.1145/3510454.3522684. Acesso em: 06 dez. 2024.

KAPITSAKI, G. M. **Generative AI for Code Generation:** software reuse implications. Springer Nature Switzerland, Cham, p. 37–47, 2024. Disponível em: https://link.springer.com/chapter/10.1007/978-3-031-66459-5_3. Acesso em: 09 dez. 2024.

KHORIKOV, V. **Unit Testing Principles, Practices, and Patterns:**: Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in c#. Manning, 2020. ISBN 9781617296277. Disponível em: https://books.google.com.br/books?id=CbvZyAEACAAJ. Acesso em: 29 jan. 2025.

KIM, D. J. **An Empirical Study on the Evolution of Test Smell**. 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion),

p. 149–151, 2020. Disponível em: https://ieeexplore.ieee.org/document/9270304. Acesso em: 28 dez. 2024.

KIM, D. J.; CHEN, T.-H. P.; YANG, J. **The secret life of test smells:** an empirical study on test smell evolution and maintenance. Kluwer Academic Publishers, USA, v. 26, n. 5, set. 2021. ISSN 1382-3256. Disponível em: https://doi.org/10.1007/s10664-021-09969-1. Acesso em: 19 nov. 2024.

KUMAR, D.; MISHRA, K. **The Impacts of Test Automation on Software's Cost, Quality and Time to Market**. Proceedings of International Conference on Communication, Computing and Virtualization (ICCCV), v. 79, p. 8–15, 2016. ISSN 1877-0509. Acesso em: 19 jan. 2025.

LI, C. **Mobile GUI test script generation from natural language descriptions using pre-trained model**. Association for Computing Machinery, New York, NY, USA, p. 112–113, 2022. Disponível em: https://dl.acm.org/doi/10.1145/3524613.3527809. Acesso em: 19 jan. 2025.

LIU, C.; BAO, X.; ZHANG, H.; ZHANG, N.; HU, H.; ZHANG, X.; YAN, M. **Guiding ChatGPT for Better Code Generation:** an empirical study. 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), p. 102–113, 2024. Disponível em: https://ieeexplore.ieee.org/document/10589825/. Acesso em: 05 dez. 2024.

LIU, P.; YUAN, W.; FU, J.; JIANG, Z.; HAYASHI, H.; NEUBIG, G. **Pre-train, Prompt, and Predict:** a systematic survey of prompting methods in natural language processing. Association for Computing Machinery, New York, NY, USA, v. 55, n. 9, jan 2023. ISSN 0360-0300. Disponível em: https://doi.org/10.1145/3560815. Acesso em: 09 dez. 2024.

LU, S.; GUO, D.; REN, S.; HUANG, J.; SVYATKOVSKIY, A.; BLANCO, A.; CLEMENT, C.; DRAIN, D.; JIANG, D.; TANG, D.; LI, G.; ZHOU, L.; SHOU, L.; ZHOU, L.; TUFANO, M.; GONG, M.; ZHOU, M.; DUAN, N.; SUNDARESAN, N.; DENG, S. K.; FU, S.; LIU, S. **CodeXGLUE:** a machine learning benchmark dataset for code understanding and generation. arXiv, 2021. Disponível em: https://arxiv.org/abs/2102.04664. Acesso em: 06 nov. 2024.

LUKASCZYK, S.; FRASER, G. **Pynguin:** automated unit test generation for python. Association for Computing Machinery, New York, NY, USA, p. 168–172, 2022. Disponível em: https://doi.org/10.1145/3510454.3516829. Acesso em: 27 dez. 2024.

MARAGATHAVALLI, P. **Search-based software test data generation using evolutionary computation**. ArXiv, abs/1103.0125, 2011. Disponível em: https://api.semanticscholar.org/CorpusID:25209645. Acesso em: 27 jan. 2025.

MARVIN, G.; HELLEN, N.; JJINGO, D.; NAKATUMBA-NABENDE, J. **Engenharia de Prompt em Grandes Modelos de Linguagem**. Springer, Cingapura, 2024. Disponível em: https://doi.org/10.1007/978-981-99-7962-2_30. Acesso em: 15 jan. 2025.

MASTROPAOLO, A.; PASCARELLA, L.; GUGLIELMI, E.; CINISELLI, M.; SCALABRINO, S.; OLIVETO, R.; BAVOTA, G. **On the Robustness of Code Generation Techniques:** an empirical study on github copilot. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), p. 2149–2160, 2023. Disponível em: https://www.computer.org/csdl/proceedings-article/icse/2023/570100c149/1OM4Phg3PMc. Acesso em: 02 dez. 2024.

MCMINN, P. **Search-based software test data generation:** a survey: Research articles. John Wiley and Sons Ltd., GBR, v. 14, n. 2, p. 105–156, jun 2004. ISSN 0960-0833. Disponível em: https://dl.acm.org/doi/abs/10.5555/1077276.1077279. Acesso em: 29 jan. 2025.

Meta. **LLaMa 3**. 2023. Disponível em: https://ai.meta.com/blog/meta-llama-3/. Acesso em: 28 dez. 2024.

MIRSHOKRAIE, S.; MESBAH, A.; PATTABIRAMAN, K. **JSEFT:** automated javascript unit test generation. 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), p. 1–10, 2015. Disponível em: https://ieeexplore.ieee.org/document/7102595. Acesso em: 27 dez. 2024.

OpenAI. **GPT-4 Technical Report**. 2024. Disponível em: https://cdn.openai.com/papers/gpt-4.pdf. Acesso em: 10 dez. 2024.

PALOMBA, F.; ZAIDMAN, A.; LUCIA, A. D. **Automatic Test Smell Detection Using Information Retrieval Techniques**. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), p. 311–322, 2018. Disponível em: https://ieeexplore.ieee.org/document/8530039. Acesso em: 13 dez. 2024.

PENG, Z.; LIN, X.; SIMON, M.; NIU, N. **Unit and regression tests of scientific software: A study on SWMM**. Journal of Computational Science, v. 53, p. 101347, 2021. ISSN 1877-7503. Disponível em: https://www.sciencedirect.com/science/article/pii/S1877750321000442. Acesso em: 29 jan. 2025.

PERUMA, A.; ALMALKI, K.; NEWMAN, C. D.; MKAOUER, M. W.; OUNI, A.; PALOMBA, F. **On the distribution of test smells in open source Android applications:** an exploratory study. Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, USA, p. 193–202, 2019. Disponível em: https://dl.acm.org/doi/10.5555/3370272.3370293. Acesso em: 02 dez. 2024.

PORNPRASIT, C.; TANTITHAMTHAVORN, C. **Fine-tuning and prompt engineering for large language models-based code review automation**. Information and Software Technology, v. 175, p. 107523, 2024. ISSN 0950-5849. Disponível em: https://www.sciencedirect.com/science/article/pii/S0950584924001289. Acesso em: 17 nov. 2024.

REEVES, B.; SARSA, S.; PRATHER, J.; DENNY, P.; BECKER, B. A.; HELLAS, A.; KIMMEL, B.; POWELL, G.; LEINONEN, J. **Evaluating the Performance of Code Generation Models for Solving Parsons Problems With Small Prompt Variations**. Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, New York, NY, USA, p. 299–305, 2023. Disponível em: https://doi.org/10.1145/3587102.3588805. Acesso em: 19 dez. 2024.

ROSS, S. I.; MARTINEZ, F.; HOUDE, S.; MULLER, M.; WEISZ, J. D. **The Programmer's Assistant:** conversational interaction with a large language model for software development. Proceedings of the 28th International Conference on Intelligent User Interfaces, New York, NY, USA, p. 491–514, 2023. Disponível em: https://doi.org/10.1145/3581641.3584037. Acesso em: 01 fev. 2025.

ROUMELIOTIS, K. I.; TSELIKAS, N. D. **ChatGPT and Open-AI Models:** a preliminary review. Future Internet, v. 15, n. 6, 2023. ISSN 1999-5903. Disponível em: https://www.mdpi.com/1999-5903/15/6/192. Acesso em: 06 dez. 2024.

RUNESON, P. **A survey of unit testing practices**. IEEE Software, v. 23, n. 4, p. 22–29, 2006. Disponível em: https://ieeexplore.ieee.org/document/1657935. Acesso em: 01 dez. 2024.

SAHOO, P.; SINGH, A. K.; SAHA, S.; JAIN, V.; MONDAL, S.; CHADHA, A. **A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications**. arXiv, 2024. Disponível em: https://arxiv.org/abs/2402.07927. Acesso em: 06 nov. 2024.

SANTANA, R.; MARTINS, L.; ROCHA, L.; VIRGÍNIO, T.; CRUZ, A.; COSTA, H.; MACHADO, I. **RAIDE:** a tool for assertion roulette and duplicate assert identification and refactoring. Proceedings of the XXXIV Brazilian Symposium on Software Engineering, p. 374–379, 2020. Disponível em: https://sol.sbc.org.br/index.php/sbes/article/view/17031. Acesso em: 28 dez. 2024.

SCHäFER, M.; NADI, S.; EGHBALI, A.; TIP, F. **An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation**. IEEE Transactions on Software Engineering, v. 50, n. 1, p. 85–105, 2024. Disponível em: https://ieeexplore.ieee.org/document/10329992. Acesso em: 19 nov. 2024.

SERRA, D.; GRANO, G.; PALOMBA, F.; FERRUCCI, F.; GALL, H. C.; BACCHELLI, A. **On the Effectiveness of Manual and Automatic Unit Test Generation:** ten years later. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), p. 121–125, 2019. Disponível em: https://ieeexplore.ieee.org/document/8816768. Acesso em: 13 nov. 2024.

SHAMSHIRI, S.; ROJAS, J. M.; GALEOTTI, J. P.; WALKINSHAW, N.; FRASER, G. **How Do Automatically Generated Unit Tests Influence Software Maintenance?** 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST), p. 250–261, 2018. Disponível em: https://ieeexplore.ieee.org/document/8367053. Acesso em: 19 nov. 2024.

SIDDIQ, M. L.; SANTOS, J. C. D. S.; TANVIR, R. H.; ULFAT, N.; RIFAT, F. A.; LOPES, V. C. **Using Large Language Models to Generate JUnit Tests:** an empirical study. Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering, New York, NY, USA, p. 313–322, 2024. Disponível em: https://doi.org/10.1145/3661167.3661216. Acesso em: 13 nov. 2024.

SPADINI, D.; PALOMBA, F.; ZAIDMAN, A.; BRUNTINK, M.; BACCHELLI, A. **On the Relation of Test Smells to Software Code** quality. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), p. 1–12, 2018. Disponível em: https://ieeexplore.ieee.org/document/8529832. Acesso em: 10 jan. 2025.

TRAUTSCH, F.; GRABOWSKI, J. **Are There Any Unit Tests?** an empirical study on unit testing in open source python projects. 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), p. 207–218, 2017. Disponível em: https://ieeexplore.ieee.org/document/7927976. Acesso em: 31 jan. 2025.

TUFANO, M.; DRAIN, D.; SVYATKOVSKIY, A.; DENG, S. K.; SUNDARESAN, N. **Unit Test Case Generation with Transformers and Focal Context**. arXiv, 2021. Disponível em: https://arxiv.org/abs/2009.05617. Acesso em: 19 nov. 2024.

TUFANO, M.; PALOMBA, F.; BAVOTA, G.; PENTA, M. D.; OLIVETO, R.; LUCIA, A. D.; POSHYVANYK, D. **An empirical investigation into the nature of test smells**. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, p. 4–15, 2016. Disponível em: https://doi.org/10.1145/2970276.2970340. Acesso em: 13 nov. 2024.

VASWANI, A.; SHAZEER, N.; PARMAR, N.; USZKOREIT, J.; JONES, L.; GOMEZ, A. N.; KAISER, L. u.; POLOSUKHIN, I. **Advances in Neural Information Processing Systems**. Curran Associates, Inc., v. 30, 2017. Disponível em: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. Acesso em: 11 jan. 2025.

WANG, C.; PASTORE, F.; GOKNIL, A.; BRIAND, L. C. **Automatic Generation of Acceptance Test Cases From Use Case Specifications: An NLP-Based Approach**. IEEE Transactions on Software Engineering, Los Alamitos, CA, USA, v. 48, n. 02, p. 585–616, feb 2022. ISSN 1939-3520. Disponível em: https://www.computer.org/csdl/journal/ts/2022/02/09103626/1kersE97zri. Acesso em: 29 jan. 2025.

WANG, T.; GOLUBEV, Y.; SMIRNOV, O.; LI, J.; BRYKSIN, T.; AHMED, I. **PyNose:** a test smell detector for python. IEEE Press, p. 593–605, 2022. Disponível em: https://doi.org/10.1109/ASE51524.2021.9678615. Acesso em: 19 nov. 2024.

WERMELINGER, M. **Using GitHub Copilot to Solve Simple Programming Problems**. Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1, New York, NY, USA, p. 172–178, 2023. Disponível em: https://doi.org/10.1145/3545945.3569830. Acesso em: 13 dez. 2024.

WHITE, J.; FU, Q.; HAYS, S.; SANDBORN, M.; OLEA, C.; GILBERT, H.; ELNASHAR, A.; SPENCER-SMITH, J.; SCHMIDT, D. C. **A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT**. ArXiv, abs/2302.11382, 2023. Disponível em: https://api.semanticscholar.org/CorpusID:257079092. Acesso em: 19 nov. 2024.

XIE, T.; NOTKIN, D. **Tool-assisted unit test generation and selection based on operational abstractions**. Automated Software Engineering Journal, v. 13, n. 3, p. 345–371, jul. 2006. Disponível em: https://link.springer.com/article/10.1007/s10851-006-8530-6. Acesso em: 27 dez. 2024.

YANG, J.; JIN, H.; TANG, R.; HAN, X.; FENG, Q.; JIANG, H.; ZHONG, S.; YIN, B.; HU, X. **Harnessing the Power of LLMs in Practice:** a survey on chatgpt and beyond. Association for Computing Machinery, New York, NY, USA, feb 2024. ISSN 1556-4681. Just Accepted. Disponível em: https://doi.org/10.1145/3649506. Acesso em: 04/12/2024.

YAO, Y.; DUAN, J.; XU, K.; CAI, Y.; SUN, Z.; ZHANG, Y. **A Survey on Large Language Model (LLM) Security and Privacy:** the good, the bad, and the ugly. High-Confidence Computing, v. 4, n. 2, p. 100211, 2024. ISSN 2667-2952. Disponível em: https://www.sciencedirect.com/science/article/pii/S266729522400014X. Acesso em: 28 dez. 2024.

YEOM, J. H.; LEE, H.; BYUN, H. *et al.* **Tc-llama 2:** fine-tuning llm for technology and commercialization applications. Journal of Big Data, v. 11, n. 100, 2024. Disponível em: https://doi.org/10.1186/s40537-024-00963-0. Acesso em: 28 dez. 2024.

YETISTIREN, B.; OZSOY, I.; TUZUN, E. **Assessing the quality of GitHub copilot's code generation**. Association for Computing Machinery, New York, NY, USA, p. 62–71, 2022. Disponível em: https://doi.org/10.1145/3558489.3559072. Acesso em: 19 jan. 2025.

YETIŞTIREN, B.; ÖZSOY, I.; AYERDEM, M.; TüZüN, E. **Evaluating the Code Quality of AI-Assisted Code Generation Tools:** an empirical study on github copilot, amazon codewhisperer, and chatgpt. ArXiv, 2023. Disponível em: https://arxiv.org/abs/2304.10778. Acesso em: 19 nov. 2024.

YU, S.; FANG, C.; LING, Y.; WU, C.; CHEN, Z. **LLM for Test Script Generation and Migration:** challenges, capabilities, and opportunities. 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS), p. 206–217, 2023. Disponível em: https://api.semanticscholar.org/CorpusID:262459296. Acesso em: 19 jan. 2025.

YUAN, Z.; LIU, M.; DING, S.; WANG, K.; CHEN, Y.; PENG, X.; LOU, Y. **Evaluating and Improving ChatGPT for Unit Test Generation**. Association for Computing Machinery, New York, NY, USA, v. 1, n. FSE, jul 2024. Disponível em: https://doi.org/10.1145/3660783. Acesso em: 27 jan. 2025.

ZHAO, W. X.; ZHOU, K.; LI, J.; TANG, T.; WANG, X.; HOU, Y.; MIN, Y.; ZHANG, B.; ZHANG, J.; DONG, Z.; DU, Y.; YANG, C.; CHEN, Y.; CHEN, Z.; JIANG, J.; REN, R.; LI, Y.; TANG, X.; LIU, Z.; LIU, P.; NIE, J.-Y.; WEN, J.-R. **A Survey of Large Language Models**. ArXiv, 2023. Disponível em: http://arxiv.org/abs/2303.18223. Acesso em: 28 dez. 2024.