Comparação Empírica entre EvoSuite e ChatUniTest na Geração de Testes de Unidade: Um Estudo de Replicação com Extensão

Djeymisson Rennan De Souza Moraes
Universidade Federal do Agreste de Pernambuco
Garanhuns, Brasil
djeymisson.moraes@ufape.edu.br

Thaís Alves Burity Rocha
Universidade Federal do Agreste de Pernambuco
Garanhuns, Brasil
thais.burity@ufape.edu.br

Resumo

Contexto: Testes de unidade são fundamentais para identificar defeitos e apoiar a evolução segura de sistemas de software. Entretanto, sua elaboração demanda tempo e esforço, tornando a geração automática uma alternativa para apoiar sua construção e manutenção. Objetivo: Comparar o ChatUniTest, baseado em grandes modelos de linguagem, ao EvoSuite, baseado em busca evolutiva, estendendo o estudo original de avaliação do ChatUniTest. Método: Avaliamos testes gerados pelas duas ferramentas para 400 métodos de cinco projetos Java, ampliando em 52% a amostra do estudo original. Além da cobertura de linhas, a avaliação contemplou produtividade, robustez, eficácia e qualidade estrutural. Resultados: O EvoSuite superou o ChatUniTest em produtividade, robustez, cobertura de linhas (em contraste ao estudo original) e escore de mutação. O ChatUniTest apresentou maior cobertura de *branches* e resultados mistos em qualidade estrutural, com menor complexidade ciclomática e menos *test smells*, porém maior quantidade de *code smells*. Conclusão: Os resultados indicam que ferramentas baseadas em LLM são promissoras para geração automática de testes, mas ainda apresentam limitações de robustez e eficácia. Ao ampliar a amostra e as dimensões de avaliação, este trabalho fornece evidências mais abrangentes sobre os potenciais e limitações dessas abordagens.

Palavras-chave

Geração Automática de Testes, Teste de Unidade, Grandes Modelos de Linguagem, EvoSuite, ChatUniTest, Estudo Empírico

1 Introdução

Testes de unidade são testes automatizados projetados para verificar o comportamento de uma unidade do sistema de forma isolada, como um método. Trata-se de um recurso amplamente utilizado em projetos de software e frequentemente associado à garantia da qualidade do código [10]. Além disso, contribuem para a detecção de falhas introduzidas durante a evolução do software, apoiando atividades de teste de regressão [12, 20]. Apesar disso, demandam tempo e esforço dos desenvolvedores [15, 24, 28], que em meio a prazos apertados, acabam por negligenciá-los, de maneira que nem sempre as suítes de testes evoluem em consonância com o código, tornando-se defasadas ou alcançando níveis de cobertura insatisfatórios.

Diversas abordagens para geração automática de testes de unidade têm sido propostas, incluindo busca evolutiva, execução simbólica e, mais recentemente, grandes modelos de linguagem (LLMs) [1, 29, 33]. Dentre elas, destacam-se o EvoSuite [6], baseado em algoritmos genéticos, e o ChatUniTest [4], que combina geração, validação e reparo automático de testes utilizando LLMs. Enquanto o EvoSuite frequentemente alcança elevada cobertura estrutural, estudos apontam limitações relacionadas à legibilidade, à relevância semântica dos cenários produzidos e à construção de objetos complexos [11, 25]. Em contrapartida, abordagens baseadas em LLM tendem a produzir testes semanticamente mais coerentes e legíveis [32], embora sua efetividade ainda demande investigação. Desde que o ChatUniTest foi proposto, diversas ferramentas baseadas em LLM surgiram, explorando diferentes estratégias de construção de contexto, validação e refinamento dos testes gerados. No entanto, a comparação entre as ferramentas é dificultada pela heterogeneidade dos protocolos experimentais, métricas e conjuntos de dados utilizados. Assim, replicações independentes e extensões de estudos anteriores são importantes para verificar a robustez dos resultados reportados e ampliar a compreensão sobre as capacidades e limitações dessas ferramentas.

O estudo original do ChatUniTest avaliou uma amostra de 264 métodos focais de quatro projetos Java, comparando a ferramenta ao EvoSuite e ao TestSpark [23] exclusivamente quanto à cobertura de linhas, além de conduzir uma pesquisa com nove desenvolvedores sobre a utilidade percebida. Com o objetivo de ampliar as evidências sobre os potenciais e limitações de abordagens baseadas em LLM, realizamos uma replicação com extensão envolvendo 400 métodos focais selecionados aleatoriamente de cinco projetos Java do Defects4J [14][1]. Inspirados no estudo de Silva et al. [27], avaliamos os testes gerados por ChatUniTest e EvoSuite em quatro dimensões: produtividade, robustez, eficácia e qualidade estrutural. A produtividade considera a quantidade de testes executáveis e o volume de código produzido; a robustez avalia falhas de compilação, falhas de execução e testes com cobertura nula; a eficácia é analisada por cobertura de linhas, cobertura de *branches* e escore de mutação; e a qualidade estrutural é medida por complexidade ciclomática, ocorrência de *code smells* e *test smells*.

Em relação ao estudo original, ampliamos a amostra em 52% e expandimos os critérios de avaliação. Além disso, optamos por não incluir o TestSpark por ser um *plugin* para o IntelliJ IDEA que permite selecionar uma dentre as estratégias já investigadas neste trabalho (LLM e busca evolutiva), o que significa que não acrescenta uma técnica distinta às já consideradas. Especificamente, investigamos quatro perguntas de pesquisa (PP):

- PP1: ChatUniTest gera mais testes executáveis do que EvoSuite?
- PP2: ChatUniTest gera menos testes defeituosos do que EvoSuite?
- PP3: ChatUniTest gera testes mais eficazes do que EvoSuite?
- PP4: ChatUniTest gera testes com melhor qualidade estrutural do que EvoSuite?

[1] https://github.com/rjust/defects4j

Em síntese, os resultados indicam que o EvoSuite superou o ChatUniTest em produtividade, robustez, cobertura de linhas (em contraste ao estudo original) e escore de mutação. O ChatUniTest alcançou maior cobertura de *branches* e melhor qualidade estrutural em termos de complexidade ciclomática e *test smells*, embora tenha produzido mais *code smells*.

A principal contribuição deste trabalho é fortalecer a avaliação empírica do ChatUniTest por meio de uma replicação independente que amplia tanto a amostra quanto as dimensões analisadas. Dessa forma, o estudo fornece evidências mais abrangentes sobre os potenciais e limitações de uma das principais abordagens baseadas em LLM para geração automática de testes de unidade.

O restante do artigo está organizado da seguinte forma: A Seção 2 apresenta as ferramentas avaliadas; a Seção 3 descreve a metodologia; a Seção 4 discute os resultados; a Seção 5 apresenta os trabalhos relacionados; a Seção 6 discute as ameaças à validade; e a Seção 7 apresenta as conclusões e os trabalhos futuros.

2 Geração Automática de Testes de Unidade

ChatUniTest e EvoSuite são ferramentas de geração automática de testes para Java que utilizam JUnit[2]. Como operam em diferentes níveis de granularidade, adotamos o método focal como unidade de análise, reorganizando os resultados do EvoSuite de acordo. Como exemplo, considere o método focal do Código 1.

```
/**
 * Subtracts a value from the value of this instance.
 * @param operand  the value to subtract, not null
 * @since 2.2
 */
public void subtract(final int operand) {
    this.value -= operand;
}
```

Código 1: Método focal `MutableInt#subtract`.

O EvoSuite gerou dois testes para o método em questão e quatro verificações (Código 2). O primeiro teste cria duas instâncias, modifica uma delas e invoca o método focal. Em vez de verificar diretamente o resultado da operação, valida propriedades do estado dos objetos, como a preservação do argumento e a desigualdade entre as instâncias. O segundo teste exercita diferentes formas de invocação do método, incluindo uma sobrecarga. Embora pareça pouco provável de ser elaborado manualmente, esse teste contribui com a cobertura estrutural do código.

```
@Test(timeout = 4000)
public void test00()  throws Throwable  {
 MutableInt mutableInt0 = new MutableInt();
 MutableInt mutableInt1 = new MutableInt(mutableInt0);
 mutableInt1.decrementAndGet();
 mutableInt0.subtract((Number) mutableInt1);
 assertEquals((-1), (int)mutableInt1.getValue());
 assertFalse(mutableInt0.equals((Object)mutableInt1));
}
@Test(timeout = 4000)
public void test01()  throws Throwable  {
 MutableInt mutableInt0 = new MutableInt();
 mutableInt0.subtract(2067);
 assertEquals((-2067.0), mutableInt0.doubleValue(), 0.01);
 mutableInt0.subtract((Number) mutableInt0);
 assertEquals(0, (int)mutableInt0.toInteger());
}
```

Código 2: Testes do EvoSuite para `MutableInt#subtract`.

[2]https://junit.org/

Por sua vez, o ChatUniTest gera um teste (Código 3) que exercita cinco cenários, incluindo subtrações com valores positivos, negativos e zero. Para inspecionar o estado interno do objeto, a ferramenta utiliza reflexão para acessar diretamente o atributo privado `value`. O ChatUniTest ainda gerou uma versão alternativa mais modular, que separa as verificações em testes independentes e usa métodos auxiliares. Como os comportamentos avaliados são essencialmente os mesmos, optamos por omitir.

```
@Test
public void testSubtract() throws NoSuchFieldException,
      IllegalAccessException {
MutableInt mutableInt = new MutableInt(10);
 // Using reflection to access the private field 'value'
 Field valueField = MutableInt.class.getDeclaredField("value");
 valueField.setAccessible(true);
 // Initial value check
 assertEquals(10, valueField.getInt(mutableInt));
 mutableInt.subtract(5); // Subtracting 5
 assertEquals(5, valueField.getInt(mutableInt));
 mutableInt.subtract(0); // Subtracting 0 (should remain unchanged)
 assertEquals(5, valueField.getInt(mutableInt));
 // Subtracting negative value (should increase the value)
 mutableInt.subtract(-3);
 assertEquals(8, valueField.getInt(mutableInt));
 mutableInt.subtract(10); // Subtracting to go negative
 assertEquals(-2, valueField.getInt(mutableInt));
}
```

Código 3: Teste do ChatUniTest para `MutableInt#subtract`.

O exemplo evidencia diferenças de estratégia entre as ferramentas. Enquanto o EvoSuite prioriza a exploração automática de estados e a maximização da cobertura estrutural, o ChatUniTest tende a produzir cenários mais próximos daqueles concebidos manualmente por desenvolvedores, tornando a intenção dos testes mais evidente. Por outro lado, a ferramenta recorre à reflexão para acessar diretamente o estado interno do objeto, aumentando o acoplamento entre o teste e a implementação da classe. Assim, alterações estruturais internas, como a renomeação de `value`, podem exigir ajustes no teste mesmo quando o comportamento externo do método focal não muda. Outra diferença é que o EvoSuite adiciona limites de tempo para evitar problemas decorrentes da exploração de estados arbitrários do programa, como laços infinitos, recursões excessivas e execuções anormalmente longas.

2.1 EvoSuite

O EvoSuite é uma ferramenta amplamente conhecida para geração automática de testes de unidade em Java. A partir da classe compilada e do *classpath* do projeto, a ferramenta modela a geração de testes como um problema de otimização e emprega algoritmos genéticos para evoluir populações de suítes de teste. Ao longo do processo, operadores de mutação e recombinação geram novas suítes, enquanto funções de aptidão (*fitness functions*) orientam a busca para maximizar a cobertura de *branches* [6].

A ferramenta tende a apresentar bons resultados em classes simples e coesas, nas quais consegue construir automaticamente estados válidos para alcançar o código-alvo. No entanto, sua eficácia diminui em cenários que exigem sequências específicas de chamadas, objetos complexos, dependências externas ou protocolos de uso implícitos [2, 18]. Também são reportadas limitações envolvendo interação com o ambiente externo, concorrência, inicialização complexa de classes e reflexão [7].

Além disso, embora a cobertura de *branches* seja amplamente utilizada como objetivo de busca, ela é uma *proxy* imperfeita para detecção de *bugs* [26]. Outra limitação é a baixa legibilidade dos testes gerados, que pode dificultar sua manutenção [8, 9], embora isso seja menos relevante em cenários voltados à validação automática ou detecção de conflitos semânticos [5].

2.2 ChatUniTest

O ChatUniTest é uma ferramenta para geração automática de testes de unidade em Java baseada em LLM, originalmente utilizando o GPT-3.5-turbo-0613 [4]. Ela se diferencia da interação direta com o modelo de linguagem por combinar análise estrutural do código, seleção de contexto e ciclos automáticos de validação e reparo.

Inicialmente, o ChatUniTest analisa a AST (*Abstract Syntax Tree*) para identificar o método focal e extrair informações relevantes sobre classes, métodos e dependências. Em seguida, usa essas informações para construir um *prompt* contendo apenas os elementos necessários para gerar os testes. Após a geração, os testes são validados quanto a problemas sintáticos, de compilação e execução. A ferramenta aplica estratégias de reparo baseadas em regras, como reorganizar código truncado por limite de *tokens* ou acrescentar *imports*. Se necessário, a ferramenta recorre ao LLM para corrigir os testes, considerando o limite de 5 ciclos de validação e reparo. Essa arquitetura busca aumentar a geração de testes válidos e executáveis, embora dependa da interpretação do modelo sobre o comportamento esperado do método focal.

3 Metodologia

Esta seção descreve o desenho experimental adotado para comparar o ChatUniTest e o EvoSuite na geração automática de testes.

3.1 Perguntas de Pesquisa

Detalhamos a seguir as perguntas de pesquisa que guiaram o estudo.

3.1.1 PP1: ChatUniTest Gera Mais Testes Executáveis do que EvoSuite? A pergunta investiga a produtividade das ferramentas. Para respondê-la, contabilizamos a quantidade de testes que compilam e executam com sucesso, o total de linhas de código (LOC) geradas e a média de testes por método focal. Consideramos como casos de teste os métodos anotados com `@Test`, enquanto a métrica LOC inclui também métodos auxiliares presentes nas classes de teste.

3.1.2 PP2: ChatUniTest Gera Menos Testes Defeituosos do que EvoSuite? A pergunta investiga a robustez das ferramentas, considerando como defeituosos os testes que apresentam falha de compilação, falha de execução ou cobertura nula sobre o método focal. Para respondê-la, contabilizamos a quantidade absoluta de testes defeituosos e métricas relativas calculadas em relação ao total de testes gerados e ao conjunto de métodos focais.

3.1.3 PP3: ChatUniTest Gera Testes Mais Eficazes do que EvoSuite? A pergunta avalia a eficácia dos testes gerados por meio das seguintes métricas:

- Cobertura de linhas: Proporção de linhas dos métodos focais executadas pelos testes;
- Cobertura de *branches*: Proporção de caminhos de decisão dos métodos focais exercitados pelos testes;
- Escore de mutação: Proporção de mutantes dos métodos focais detectados pelos testes.

Valores mais elevados indicam maior capacidade dos testes de explorar e validar o comportamento do código. Como métricas de cobertura estrutural não garantem necessariamente a detecção de defeitos, também avaliamos o escore de mutação, frequentemente considerado uma das métricas mais relevantes para estimar a qualidade dos testes [13]. A comparação entre as ferramentas considera média, mediana e desvio padrão.

3.1.4 PP4: ChatUniTest Gera Testes com Melhor Qualidade Estrutural do que EvoSuite? A pergunta investiga a qualidade estrutural dos testes por meio de três métricas: Complexidade ciclomática, ocorrência de *test smells* e ocorrência de *code smells*. Para respondê-la, comparamos média, mediana e desvio padrão por método focal.

A complexidade ciclomática estima a quantidade de caminhos de execução possíveis em um teste. Valores menores indicam estruturas mais simples e fáceis de manter. Neste estudo, calculamos a complexidade de cada teste e usamos a média por método focal como unidade de análise. Os *test smells* são indicadores de problemas de projeto e implementação de testes que podem comprometer legibilidade, manutenibilidade e capacidade de detectar defeitos [22]. Menores quantidades são desejáveis. Como o estudo analisa testes por métodos individuais, excluímos *smells* de nível de suíte ou específicos de ecossistemas. A Tabela 1 apresenta os *test smells* considerados, detectados pela ferramenta TsDetect[3]. De forma complementar, os *code smells* são indicadores de problemas gerais de qualidade de código. Aqui, utilizamos os *smells* detectados pela ferramenta PMD[4] e contabilizamos a ocorrência nos testes por método focal.

3.2 Construção da Amostra

Construímos uma amostra de métodos focais extraídos de projetos Java do GitHub, para os quais geramos testes utilizando as ferramentas ChatUniTest e EvoSuite. As subseções resumem o processo de 3 etapas da construção da amostra.

3.2.1 Seleção dos Projetos. Utilizamos o repositório de *bugs* Defects4J como fonte de projetos Java com ampla suíte de testes. Dos 17 projetos disponíveis, excluímos `commons-cli` e `commons-csv` por terem sido usados no estudo original. Entre os 15 projetos restantes, realizamos seleções aleatórias sucessivas até obter 5 projetos compatíveis com Java 11 e suporte ao Maven, requisitos necessários para execução do ChatUniTest. A tabela 2 apresenta os projetos selecionados, considerando a versão usada e a quantidade de classes e métodos de produção.

3.2.2 Seleção dos Métodos Focais. Visando compor uma amostra ampla e equilibrada, definimos como meta selecionar 80 métodos focais por projeto, totalizando uma amostra de 400 métodos para os quais fosse possível gerar ao menos um teste com ambas as ferramentas. Para isso, utilizamos o *parser* Java do ChatUniTest para extrair os métodos dos projetos e armazená-los em arquivos JSON, posteriormente processados por um *script* Python desenvolvido para automatizar a execução e organização do estudo.

[3] https://github.com/TestSmells/TestSmellDetector
[4] https://pmd.github.io/

Tabela 1: *Test smells* avaliados.

Test Smell	Descrição
Assertion Roulette	Múltiplas asserções sem propósito claro
Conditional Test Logic	Uso de condicionais ou repetições
Duplicate Assert	Asserções com os mesmos parâmetros
Empty Test	Sem comandos executáveis
Exception Handling	Tratamento explícito de exceções
Ignored Test	Teste ignorado
Magic Number Test	Literais sem significado claro
Eager Test	Invocação de múltiplos métodos de produção
Mystery Guest	Dependência de recursos externos
Redundant Print	Impressão no console
Redundant Assertion	Asserção com valores esperado e obtido equivalentes
Resource Optimism	Uso de arquivos sem garantia de existência
Sensitive Equality	Uso de `toString()` em comparações
Sleepy Test	Chamada a `Thread.sleep()`
Unknown Test	Sem asserções nem exceções esperadas

Tabela 2: Projetos Java selecionados.

Nome	Versão	#Classes	#Métodos
jackson-core	2.15.4	138	2.455
commons-lang	3.17.0	338	3.801
gson	2.13.2	100	669
joda-time	2.13.0	243	3.793
jsoup	1.18.1	173	1.550

Sorteamos os métodos que atendiam aos seguintes critérios: (i) Eram públicos; (ii) possuíam implementação concreta; (iii) não eram construtores, *getters* ou *setters*; e (iv) continham lógica executável, excluindo métodos vazios, compostos apenas por comentários ou por instruções de lançamento de exceção.

Dos 400 métodos sorteados, o ChatUniTest gerou testes para 386 métodos (96,5%) e o EvoSuite, para 360 métodos (90%). No total, restaram 343 métodos cobertos por testes executáveis gerados por ambas as ferramentas (85,7%). Para atingir a meta de 80 métodos por projeto, realizamos uma seleção complementar de 57 métodos, adotando critérios adicionais que são requisitos para o uso das ferramentas de testes e de qualidade, que só identificamos ao analisar os casos de falha. Além dos critérios anteriores, exigimos elegibilidade para mutação, excluímos classes com escopo de pacote, classes base estruturais e pacotes *internal* (por exemplo, `com.google.gson.internal`), que frequentemente concentram classes utilitárias de uso estritamente interno à biblioteca, apresentando construtores complexos que costumam causar falhas ou atingir o *timeout* na geração dos testes pelo EvoSuite. Também restringimos a seleção a métodos com três parâmetros, no máximo.

3.2.3 Geração de Testes. Para executar o ChatUniTest (versão 2.1.1), configuramos os projetos com as dependências e os *plugins* Maven requeridos pela ferramenta. Mantivemos sua configuração padrão, substituindo apenas o modelo de linguagem original pelo GPT-4o-mini e mantendo os parâmetros de geração definidos pela ferramenta: Temperatura de 0,5, *top-p* de 1,0, *frequency penalty* de 0, *presence penalty* de 0 e limite máximo de 1.024 *tokens* na resposta. A ideia foi utilizar uma versão mais recente do modelo mantendo custos operacionais viáveis. O ChatUniTest requer Java 11 e utiliza JUnit 5.

No caso do EvoSuite, utilizamos a configuração padrão com *timeout* de 60 segundos para geração dos testes. Como a ferramenta opera em nível de classe, desenvolvemos um *script* para filtrar os testes associados a cada método focal. O EvoSuite requer Java 8 e usa JUnit 4.

3.3 Avaliação de Métricas

A execução dos testes e a extração das métricas foram automatizadas por um *script* Python [3], responsável por processar os artefatos gerados pelo ChatUniTest e EvoSuite, executar os testes e consolidar os resultados por pergunta de pesquisa. O *script* também contabilizou a quantidade de métodos focais testados, testes gerados e linhas de código produzidas.

Realizamos a compilação e execução dos testes por meio do *maven-surefire-plugin*. Para lidar com falhas de compilação, execução e *flaky tests*, implementamos um mecanismo automatizado de isolamento dos testes defeituosos. A partir dos *logs*, a cada execução o script identificava o método de teste responsável pela falha, o desabilitava e repetia a execução até que apenas testes executáveis permanecessem ativos. A quantidade de testes desabilitados correspondeu ao número de testes falhos. Também contabilizamos os testes executáveis que não alcançaram cobertura sobre o método focal.

Obtivemos as métricas de cobertura de linhas e cobertura de *branches* através do JaCoCo[5] (v0.8.11). Calculamos o escore de mutação com o Pitest[6] (v1.15.0). Como o Pitest opera em nível de classe, utilizamos a biblioteca *javalang* para identificar os limites dos métodos focais na AST e associá-los aos mutantes reportados.

Por fim, calculamos a complexidade ciclomática com o Lizard[7], detectamos *test smells* com o TsDetect e detectamos *code smells* com o PMD, contabilizando-se as violações reportadas para os testes associados a cada método focal.

4 Resultados

Os resultados da avaliação dos testes de unidade gerados pelas ferramentas ChatUniTest e EvoSuite para a amostra de 400 métodos focais são apresentados a seguir, organizados por pergunta de pesquisa.

4.1 PP1: Produtividade

Em relação à quantidade de testes executáveis, o EvoSuite produziu 1.853 testes (média de ≈ 5 por método focal), enquanto o ChatUniTest gerou 1.470 testes (média de ≈ 4 por método focal). Entretanto,

[5] https://www.eclemma.org/jacoco/
[6] https://github.com/hcoles/pitest
[7] https://github.com/terryyin/lizard

ao considerar todos os métodos presentes nas classes de teste, incluindo métodos auxiliares, o ChatUniTest totalizou 2.354 métodos, contra 1.853 do EvoSuite. Esse resultado sugere estratégias distintas de organização do código: Enquanto o EvoSuite concentra a lógica diretamente nos métodos de teste, o ChatUniTest utiliza com maior frequência métodos auxiliares para encapsular código reutilizável, como inicialização e configuração de objetos.

Numa perspectiva complementar, além de ter produzido cerca de 26% mais testes executáveis, o EvoSuite gerou testes mais extensos, totalizando mais que o triplo de linhas de código geradas pelo ChatUniTest, o que corresponde a $\approx$ 57 linhas por teste, contra $\approx$ 23 linhas por teste. Essa diferença pode estar relacionada à maior utilização de métodos auxiliares pelo ChatUniTest, que favorece o reúso de código entre os testes.

Os testes gerados para `org.jsoup.nodes.Element#isBlock` ilustram esse comportamento e são discutidos em detalhe no material suplementar online[8]. O EvoSuite produziu cenários mais extensos, envolvendo múltiplas operações sobre objetos relacionados e diversas verificações de estado além do método focal. Já o ChatUniTest gerou testes mais diretos, concentrados nos principais cenários de uso de `isBlock()`, ainda que recorrendo a *mocking* e reflexão para isolar dependências.

> Resposta à PP1: Não. O EvoSuite gerou uma quantidade maior de testes executáveis. Entretanto, os testes gerados pelo ChatUniTest apresentaram menor volume de código e maior uso de métodos auxiliares, resultando em classes menos verbosas.

4.2 PP2: Robustez

Conforme ilustrado pelas Figuras 1 e 2, o EvoSuite apresentou maior robustez em comparação ao ChatUniTest, tanto na análise geral da amostra quanto na análise por projeto. Considerando como teste cada método anotado com `@Test`, o EvoSuite registrou apenas 8 falhas de compilação ou execução (0,43% do total) e 31 testes com cobertura zero (1,67%). Já o ChatUniTest apresentou 118 testes com erro de compilação ou execução (8,03%) e 105 testes com cobertura zero (7,14%). Os piores resultados do ChatUniTest concentraram-se em dois projetos. No *gson*, 11% dos testes apresentaram falhas de compilação ou execução. Já no *jsoup*, 26,52% dos testes gerados não alcançaram cobertura sobre o método focal, percentual substancialmente superior ao observado nos demais projetos.

A diferença observada é particularmente relevante porque o ChatUniTest incorpora um ciclo automático de validação e reparo, submetendo os testes a um processo iterativo de correção. Para compreender melhor esse mecanismo, analisamos dez arquivos de *records* distribuídos entre os cinco projetos. Os registros mostram que a ferramenta corrige parte das falhas de compilação e execução, geralmente após uma ou duas rodadas de reparo. Entretanto, esse mecanismo nem sempre é acionado. Observamos casos em que apenas alguns testes de uma classe apresentavam falhas, enquanto os demais eram gerados corretamente. Nessas situações, a ferramenta não submetia os testes problemáticos ao mecanismo de reparo. Além disso, problemas envolvendo hierarquias complexas, classes abstratas, classes internas e dependências mais elaboradas persistiam mesmo após múltiplas tentativas de correção.

[8]Disponível em: https://github.com/Djeymisson/chatunitest-analysis-results/blob/main/appendix/element_isblock.md.

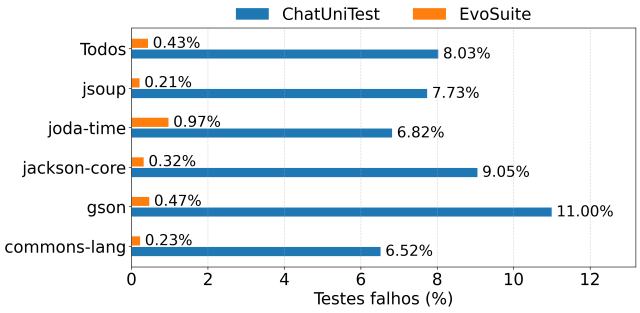

Figura 1: Testes falhos gerados por projeto.

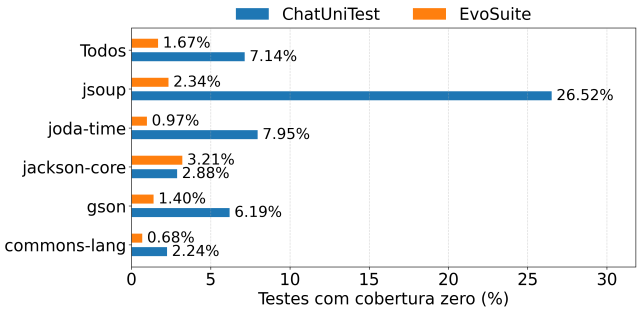

Figura 2: Testes com cobertura zero gerados por projeto.

Ainda assim, a principal limitação observada não está relacionada aos testes que falham durante a validação, mas àqueles que compilam e executam corretamente sem exercitar o método focal. Para compreender esse fenômeno, realizamos uma inspeção manual de 60 métodos focais associados a testes com cobertura zero gerados pelo ChatUniTest. A análise revelou três padrões. No primeiro, o modelo recria manualmente classes ou comportamentos de produção dentro do próprio teste, em vez de utilizar a implementação real do sistema (*alucinação estrutural*). No segundo, a ferramenta cria *mocks* inadequados que substituem a própria classe sob teste, fazendo com que as verificações incidam apenas sobre retornos artificialmente configurados (*mocking indevido da classe focal*). Por fim, observamos casos de uso excessivo de *reflection* para manipular estados internos ou invocar métodos auxiliares sem percorrer os fluxos normais de execução da aplicação. Em todos esses casos, os testes gerados compilam e executam sem erros, mas deixam de verificar o comportamento do código de produção, resultando em cobertura nula sobre o método focal.

De maneira complementar, analisamos manualmente os 118 testes com falha de compilação ou execução gerados pelo ChatUniTest. De acordo com a Tabela 3, as três categorias mais frequentes compreendem 84,8% das ocorrências. O projeto *Gson*, que apresentou o maior percentual de testes falhos, ilustra bem essas limitações.

Gson em muitas situações lida com valores nulos, vazios ou inválidos sem lançar exceções. Apesar disso, o ChatUniTest gerou testes admitindo que certas entradas produziriam exceção. Em métodos invocados via reflexão, o ChatUniTest falhou ao verificar a exceção original em vez de *InvocationTargetException*, que é a

Tabela 3: Falhas em testes gerados pelo ChatUniTest.

Causa	Quant.	Quant.%
Interpretação incorreta do comportamento do sistema	50	42,4
Uso incorreto da API de *Reflection*	32	27,1
Configuração inadequada do ambiente de teste	18	15,3
Falhas sem causa identificável	12	10,2
Inferência de comportamentos inexistentes	5	4,2
Limitações do código sob teste	1	0,8
Total	118	100

exceção propagada em contexto de reflexão. Além disso, o ChatUniTest nem sempre compreendeu máquinas de estado implícitas, gerando sequências inválidas de chamadas de método. Por exemplo, em um leitor de JSON posicionado no início do documento, um teste incorreto invocaria `endObject()` antes de `beginObject()`, violando a ordem de operações exigida pela API. Em vários usos de *mocks* com *Mockito*, o ChatUniTest criava objetos simulados, mas omitia o *stubbing* necessário. Por exemplo, um método que depende de `repository.buscar(id)` requer uma configuração como `when(repository.buscar(id)).thenReturn(objeto)`. Sem ela, o *mock* pode retornar valores padrão e causar falhas antes da execução do método focal. Em outros casos, o ChatUniTest gerava entradas inválidas para o método focal, como a invocação de `salvar(null)` em métodos que exigiam objetos válidos. Assim, a falha estava associada à geração da entrada, e não ao comportamento do método focal.

Os casos remanescentes correspondem a falhas sem causa claramente identificável, nas quais as asserções utilizam valores incompatíveis com o comportamento observado do método sob teste. Por exemplo, uma asserção que compara se um método retorna 5 como sendo o dobro de 2 ao invés de 4. Também identificamos casos de alucinação, em que a ferramenta inferiu validações inexistentes, inventou interfaces ou interpretou incorretamente regras de negócio, evidenciando limitações na compreensão semântica do sistema sob teste. Por fim, observamos um caso em que foi gerado um teste para verificar o lançamento de uma exceção declarada no código, mas que não poderia ocorrer em tempo de execução, resultando em uma expectativa incompatível com o comportamento real do método.

Já o EvoSuite gerou um número reduzido de falhas, cuja natureza difere daquelas observadas no ChatUniTest. A análise manual dos 8 testes falhos revelou limitações associadas à exploração de combinações incomuns de entradas e estados de objetos. Como consequência, a ferramenta construiu estruturas artificiais com dependências circulares e recursões infinitas, resultando em `StackOverflowError` nos projetos *jackson-core* e *gson*. Além disso, a geração de valores extremos e combinações atípicas produziu exceções de tipagem em *joda-time* e falhas de validação relacionadas a caminhos de arquivos inválidos em *jsoup*. Por fim, observamos um *flaky test* em *commons-lang*, cuja asserção dependia de valores voláteis do ambiente de execução, como o resultado de `hashCode()`, tornando o teste não determinístico.

> Resposta à PP2: Não. O ChatUniTest gerou uma quantidade superior de testes com falha de compilação ou execução, além de um número maior de testes com cobertura zero. Além disso, as falhas são de naturezas distintas: No ChatUniTest, predominam falhas relacionadas à compreensão do comportamento do sistema e a configuração do ambiente de testes; no EvoSuite, predominam falhas decorrentes da exploração de cenários artificiais para maximização da cobertura.

4.3 PP3: Eficácia

Conforme ilustrado pela Figura 3 e contrariando o estudo original, o EvoSuite superou o ChatUniTest em cobertura média de linhas, havendo diferença estatística entre os valores segundo o teste de *Wilcoxon Signed-Rank* ($p = 0,011$), apesar do tamanho do efeito tenha sido desprezível ($\delta = -0,068$), sugerindo uma diferença muito pequena na prática.

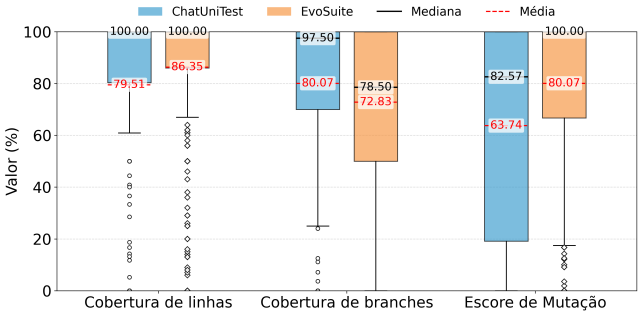

Figura 3: Eficácia dos testes gerados por ferramenta.

A análise por projeto (vide apêndice [3]) mostra que a principal diferença ocorreu no projeto *jsoup*, no qual o ChatUniTest obteve cobertura média inferior a 60%. Esse resultado está associado à elevada incidência de testes com cobertura zero, discutida na Subseção 4.2. Nos demais projetos, a cobertura média da ferramenta permaneceu superior a 80%.

De maneira complementar, os resultados de cobertura detalhados por projeto são apresentados na Tabela 4. Em geral, ambas as ferramentas alcançaram mediana de cobertura próxima ou igual a 100%, indicando que a diferença observada decorre principalmente de um subconjunto reduzido de casos com baixo desempenho (*jsoup*).

Tabela 4: Cobertura estrutural.

Métrica	Projeto	ChatUniTest / EvoSuite		
		Média	Mediana	DP
Linhas	*commons-lang*	86,64% / 87,85%	100% / 100%	0,29 / 0,24
	gson	82,16% / 81,88%	100% / 100%	0,32 / 0,31
	jackson-core	86,45% / 82,95%	100% / 100%	0,30 / 0,32
	joda-time	86,16% / 88,10%	100% / 100%	0,31 / 0,25
	jsoup	56,13% / 90,96%	87,30% / 100%	0,47 / 0,25
Branches	*commons-lang*	88,44% / 78,55%	100% / 82,50%	0,23 / 0,21
	gson	77,82% / 61,11%	80% / 61%	0,23 / 0,29
	jackson-core	73,78% / 72,63%	91,67% / 81%	0,34 / 0,30
	joda-time	88,22% / 75,43%	100% / 79%	0,20 / 0,26
	jsoup	67,35% / 78,52%	75% / 100%	0,32 / 0,27

Em relação à cobertura de *branches*, o resultado foi inverso: o ChatUniTest superou o EvoSuite, com diferença estatisticamente significativa ($p = 8,72 \times 10^{-4}$) e tamanho de efeito pequeno ($\delta = 0,190$). Ainda assim, os resultados por projeto não foram homogêneos (Tabela 4). O ChatUniTest apresentou melhor desempenho (mediana) em *commons-lang*, *gson*, *jackson-core* e *joda-time*, enquanto o EvoSuite obteve vantagem em *jsoup*.

Por outro lado, além de obter maior cobertura média de linhas, o EvoSuite também apresentou escore de mutação superior (Figura 3), com diferença estatisticamente significativa segundo o teste de *Wilcoxon Signed-Rank* ($p = 2,74 \times 10^{-9}$) e tamanho de efeito pequeno ($\delta = -0,212$). Com base na Tabela 5, nota-se que o EvoSuite superou o ChatUniTest em quatro dos cinco projetos, com exceção de *gson*. A maior diferença ocorreu em *jsoup*, projeto no qual o ChatUniTest também apresentou sua menor cobertura média de linhas.

Tabela 5: Escore de mutação.

Projeto	ChatUniTest / EvoSuite		
	Média	Mediana	DP
commons-lang	68,5% / 88,1%	83,3% / 100%	0,38 / 0,23
gson	63,18% / 59,6%	78,9% / 63,4%	0,40 / 0,37
jackson-core	70,8% / 83,6%	100% / 100%	0,30 / 0,31
joda-time	71,7% / 80,8%	100% / 100%	0,41 / 0,30
jsoup	44,5% / 88,3%	39,6% / 100%	0,40 / 0,28

Em conjunto, esses resultados indicam que a maior cobertura de *branches* alcançada pelo ChatUniTest não se traduz, necessariamente, em maior capacidade de detecção de defeitos. Embora a ferramenta explore mais caminhos de execução, o EvoSuite exercita uma porção maior do código executável e demonstra maior efetividade em revelar alterações artificiais de comportamento. Além disso, a maior variabilidade observada em projetos como *jsoup* sugere que o desempenho do ChatUniTest é mais sensível às características do sistema analisado.

> Resposta à PP3: Não. Embora o ChatUniTest tenha alcançado maior cobertura média de *branches*, o EvoSuite obteve melhores resultados gerais de cobertura de linhas e escore de mutação. Isso sugere que a maior exploração de fluxos lógicos promovida pelo ChatUniTest não se converteu em maior efetividade na detecção de defeitos potenciais.

4.4 PP4: Qualidade Estrutural

O ChatUniTest apresentou menor complexidade ciclomática (média = 1,09; mediana = 1,00; DP = 0,32) do que o EvoSuite (média = 1,21; mediana = 1,00; DP = 0,34), havendo pouca variação das médias entre os projetos para ambas as ferramentas (vide apêndice [3]). O teste de *Wilcoxon Signed-Rank* indicou diferença estatisticamente significativa ($p = 1,45 \times 10^{-9}$), embora com tamanho de efeito pequeno ($\delta = -0,189$). Apesar disso, o ChatUniTest também apresentou os maiores valores em casos isolados, atingindo média 5,0 para os testes de um método focal, enquanto o máximo observado para o EvoSuite foi 2,0.

Em termos de *test smells*, o EvoSuite apresentou aproximadamente o dobro de ocorrências do ChatUniTest (6.634 contra 3.312). A Figura 4 mostra a distribuição dos principais tipos identificados. Em ambas as ferramentas, predominaram os *smells Magic Number Test* e *Exception Handling*, indicando uso frequente de valores literais e tratamento explícito de exceções nos testes gerados, que podem reduzir a expressividade e aumentar a complexidade estrutural dos testes.

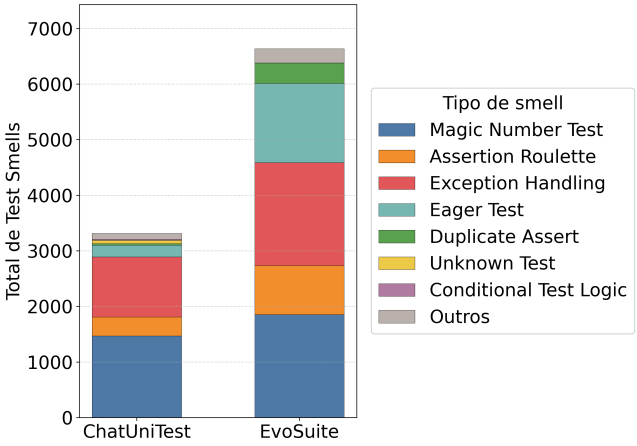

Figura 4: Tipos de *test smells* por ferramenta.

A maior diferença entre as ferramentas ocorreu na categoria *Eager Test*, com 1.426 ocorrências no EvoSuite e 210 no ChatUniTest. Como este estudo adota o método focal como unidade de análise, esse resultado sugere que os testes gerados pelo EvoSuite tendem a exercitar mais comportamentos além do método focal, reduzindo sua coesão. Também foram observadas diferenças expressivas em *Assertion Roulette* (879 contra 340 ocorrências) e *Duplicate Assert* (368 contra 27), indicando maior incidência de múltiplas asserções sem propósito explícito e verificações redundantes.

Por outro lado, os *smells Unknown Test* (59 ocorrências) e *Conditional Test Logic* (21 ocorrências) foram identificados exclusivamente nos testes gerados pelo ChatUniTest, indicando casos em que os testes não possuíam asserções explícitas ou incorporavam lógica condicional desnecessária.

Diferentemente do observado para os *test smells*, a incidência de *code smells* detectada pelo PMD foi maior nos testes do ChatUniTest. Foram registradas 13.163 ocorrências (média de 32,91 por arquivo) contra 9.298 no EvoSuite (média de 23,25). O teste de Wilcoxon indicou diferença estatisticamente significativa ($p = 1,167 \times 10^{-14}$), com tamanho de efeito médio ($\delta = 0,36$), favorecendo o EvoSuite.

A distribuição de *code smells* por ferramenta é apresentada na Figura 5. Observa-se que a maior parte das ocorrências está concentrada em *UnnecessaryImport*, responsável por 10.656 ocorrências no ChatUniTest e 5.188 no EvoSuite. As demais categorias ocorreram em frequências substancialmente menores. Esse resultado sugere que a diferença global de *code smells* entre as ferramentas é fortemente influenciada por importações não utilizadas, um problema relativamente simples de corrigir automaticamente. Ainda assim, o ChatUniTest também apresentou ocorrências associadas a convenções de nomenclatura, uso de reflexão e métodos vazios sem documentação, indicando que parte das violações não se restringe

a questões superficiais de formatação. Já o EvoSuite concentrou mais violações associadas à instanciação desnecessária de *wrappers*, chamadas sem efeito e blocos de exceção vazios.

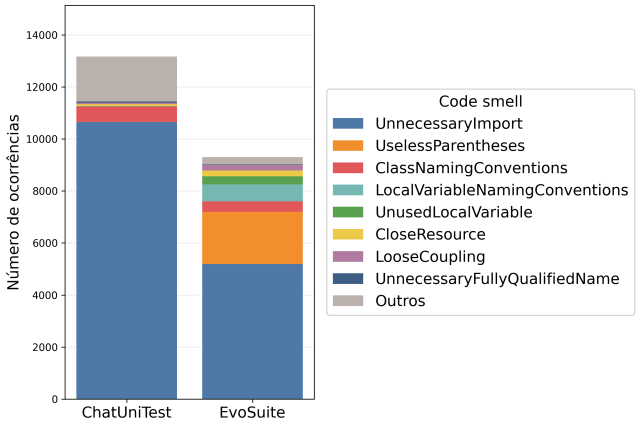

Figura 5: Tipos de *code smells* por ferramenta.

> Resposta à PP4: Parcialmente. O ChatUniTest apresentou menor complexidade ciclomática e menor incidência de *test smells*, sugerindo testes mais simples e coesos. Por outro lado, apresentou maior quantidade de *code smells*, embora a diferença tenha sido fortemente influenciada por ocorrências de *UnnecessaryImport*.

5 Trabalhos Relacionados

Diversas ferramentas baseadas em LLM têm sido propostas para geração de testes de unidade em Java. Dentre elas, este trabalho concentra-se nas abordagens que evoluem a estratégia introduzida pelo ChatUniTest, incorporando novos mecanismos de extração de contexto, geração, reparo ou refinamento dos testes. Como essas ferramentas compartilham uma arquitetura semelhante, compreender suas contribuições requer analisar também como essas ferramentas foram avaliadas. A Tabela 6 resume os principais aspectos dos estudos de avaliação, incluindo linguagem, conjunto de dados da avaliação, modelos utilizados, ferramentas comparadas e métricas consideradas.

O ChatTester [32] utiliza o ChatGPT para inferir a intenção do método focal em linguagem natural e emprega essa descrição como contexto para gerar os testes. Em caso de erros de compilação, novas interações com o ChatGPT são realizadas para corrigi-los. Como o EvoSuite normalmente produz múltiplos testes por método, os autores selecionaram aleatoriamente um deles para comparação.

O ASTER [21] incorpora suporte sistemático à geração de testes para aplicações com dependências externas, identificando campos, tipos e chamadas que devem ser simulados por meio de *mocks* e *stubs*. Após gerar e validar os testes, a ferramenta utiliza a cobertura obtida para solicitar novos testes direcionados às linhas ainda não cobertas.

O HITS [30] divide métodos complexos em fatias (*slices*) e gera testes para cada uma delas, buscando aumentar a cobertura de linhas e *branches*. A ferramenta utiliza GPT-3.5-turbo-0125.

O KTester [17] busca aprimorar o projeto dos testes incorporando conhecimento sobre o projeto obtido por análise estática, como padrões de uso e exemplos de invocação de métodos. A ferramenta separa o projeto dos casos de teste de sua codificação por meio de uma representação intermediária em JSON e fornece diretrizes relacionadas ao fluxo de controle, comportamento funcional e tratamento de exceções. Os testes são posteriormente validados e corrigidos por novas interações com o GPT-4o-mini.

A ferramenta TELPA [31] busca aumentar a cobertura de *branches* difíceis combinando análise estática e LLMs. Para isso, identifica cenários reais de uso do método, recupera a construção de objetos complexos, analisa dependências das condições dos *branches* e utiliza testes malsucedidos como contraexemplos para orientar novas gerações. Originalmente proposta para Python, a ferramenta foi posteriormente adaptada para Java.

O IntUT [19] busca aumentar a cobertura dos testes gerados por meio de uma etapa intermediária de construção da intenção de teste. Para tanto, realiza análise estática do método focal para identificar as condições necessárias para alcançar cada *branch*, as chamadas que devem ser simuladas por meio de *mocks* e os resultados esperados, incorporando essas informações ao *prompt* juntamente com o contexto do código e exemplos de testes semelhantes recuperados automaticamente.

O YATE [16] combina análise estática, reparo automático de testes e refinamento iterativo orientado por cobertura. A ferramenta corrige erros de compilação, execução e asserções antes de solicitar novos testes para cobrir *branches* ainda não exercitadas. Entre os modelos avaliados, o GPT-4o-mini apresentou o melhor desempenho.

Em conjunto, esses trabalhos mostram uma evolução das estratégias de geração de testes com LLMs, passando da simples geração a partir do método focal para abordagens que incorporam intenção de teste, fatiamento, suporte a *mocks*, análise de dependências, reparo automático e realimentação por cobertura. Entretanto, as mudanças introduzidas por essas ferramentas também alteram significativamente o processo de geração, dificultando isolar se os ganhos observados decorrem das novas estratégias propostas ou de diferenças nos protocolos experimentais empregados. Além disso, as avaliações permanecem concentradas em cobertura estrutural, compilação, execução e percepção de desenvolvedores. Apenas o YATE considera escore de mutação, enquanto aspectos relacionados à qualidade estrutural dos testes, como complexidade e incidência de *test smells*, continuam pouco explorados. Nesse contexto, uma avaliação mais abrangente do ChatUniTest torna-se importante não apenas para compreender suas limitações, mas também para servir de referência na interpretação dos avanços propostos por trabalhos posteriores.

6 Ameaças à Validade

Por se tratar de uma replicação com extensão, buscamos reproduzir o protocolo experimental do estudo original, mas substituímos o GPT-3.5-turbo-0613 pelo GPT-4o-mini, avaliando o ChatUniTest em um cenário tecnológico mais atual. Entretanto, essa alteração impede atribuir eventuais diferenças em relação ao estudo original exclusivamente à amostra utilizada. Diferenças nos ambientes de execução (Java 8/JUnit 4 versus Java 11/JUnit 5) também podem

Tabela 6: Trabalhos relacionados.

Trabalho	Dados de avaliação	*Baselines*	Dimensões avaliadas	Principal resultado
ChatTester	100 métodos + 3 projetos	ChatGPT, EvoSuite	Compilação, execução, cobertura de linhas, cobertura de *branches*, legibilidade e usabilidade	Semelhante ao EvoSuite em cobertura
ASTER	9.977 métodos	EvoSuite	Cobertura de linhas, cobertura de *branches*, cobertura de métodos, naturalidade e percepção de desenvolvedores	Superior em aplicações com dependências externas
HITS	120 métodos	EvoSuite, ChatUniTest, ChatTester e SymPrompt	Execução, cobertura de linhas, cobertura de *branches* e contribuição do fatiamento	Maior cobertura média; EvoSuite superior em alguns projetos
KTester	110 métodos	ChatUniTest, ChatTester, HITS e UTGen	Compilação, execução, cobertura de linhas, cobertura de *branches*, tempo de geração, quantidade de testes, corretude, legibilidade e manutenibilidade	Maior cobertura; UTGen superior em execução
TELPA	4 projetos Defects4J	EvoSuite	Cobertura de linhas e cobertura de *branches*	Superior ao EvoSuite em *branches* difíceis
IntUT	57 métodos	Configurações do próprio IntUT	Compilação, execução, cobertura de linhas, cobertura de *branches*, proximidade aos testes humanos	Intenção de teste aumenta a cobertura
YATE	3.657 métodos	HITS, SymPrompt, TestSpark, CoverUp e LLM-Plain	Compilação, execução, cobertura de linhas, cobertura de *branches*, escore de mutação e custo computacional	Melhor cobertura e escore de mutação

ter afetado os resultados, particularmente métricas baseadas na execução dos testes, como cobertura e escore de mutação. Para *code smells*, detectados via análise estática, o impacto é indireto, dado que a versão do JUnit pode causar diferenças estruturais nos testes. Não padronizamos os ambientes para preservar os requisitos de cada ferramenta. Assim, os resultados representam as ferramentas em seus respectivos ambientes, e não exclusivamente as estratégias de geração de testes empregadas.

Os projetos selecionados estavam disponíveis publicamente antes da data de corte de conhecimento do GPT-4o-mini (outubro de 2023), não sendo possível descartar que tenham sido incluídos nos dados de treinamento do modelo, potencialmente favorecendo o desempenho do ChatUniTest. Esse é um desafio recorrente em estudos empíricos envolvendo LLMs e projetos *open source*. Uma possível estratégia de mitigação seria incluir projetos privados ou disponibilizados após a data de corte, embora esta última alternativa restrinja a seleção à projetos mais recentes e, potencialmente, com menor volume de dados para análise. Nosso estudo utilizou projetos do Defects4J visando a possibilidade futura de investigar o potencial dos testes gerados na detecção de *bugs*.

As métricas adotadas não contemplam aspectos como legibilidade, manutenibilidade e utilidade percebida pelos desenvolvedores. Além disso, a classificação manual das falhas está sujeita à subjetividade.

A generalização dos resultados é limitada aos 400 métodos focais extraídos de cinco projetos Java de código aberto. Além disso, o desempenho de abordagens baseadas em LLM depende do modelo, do contexto fornecido a este e das estratégias de geração, validação e reparo, podendo variar em outras configurações ou versões.

Como o EvoSuite gera testes em nível de classe, foi necessário reorganizá-los por método focal antes da extração das métricas. Para garantir integridade, o processo foi automatizado e os artefatos foram inspecionados. Por fim, preservamos a estratégia do estudo original de realizar uma única execução do ChatUniTest por método focal, que engloba até cinco tentativas de reparo. Dada a natureza não determinística da ferramenta, a estratégia pode não capturar toda a variabilidade dos resultados.

7 Conclusão

Este trabalho apresentou uma replicação com extensão da avaliação do ChatUniTest, comparando-o ao EvoSuite em uma amostra de 400 métodos focais de cinco projetos Java de código aberto. Em relação ao estudo original, ampliamos a amostra e as dimensões de qualidade avaliadas. Os resultados indicam desempenho superior do EvoSuite em produtividade, robustez e efetividade, com mais testes executáveis, menos falhas, maior cobertura de linhas (ao contrário do estudo original) e maior escore de mutação. Embora o ChatUniTest tenha obtido maior cobertura média de *branches*, essa vantagem não se refletiu em maior capacidade de detecção de defeitos, evidenciando que métricas de cobertura, isoladamente, são insuficientes para caracterizar a qualidade dos testes.

Por outro lado, o ChatUniTest gerou testes menos complexos e com menos *test smells*, mas apresentou limitações de robustez. A análise manual mostrou que parte dos testes compila e executa sem exercitar adequadamente o método focal, devido principalmente à interpretação incorreta do comportamento esperado, uso inadequado de *mocks*, abuso de *reflection* e alucinações estruturais.

Em conjunto, os resultados mostram que o EvoSuite permanece como uma alternativa mais robusta para geração automática de testes, enquanto o ChatUniTest demonstra potencial para produzir testes mais simples e legíveis, mas ainda carece de mecanismos mais eficazes de compreensão do contexto, validação e reparo dos testes gerados.

Como trabalhos futuros, pretende-se investigar a capacidade dos testes gerados de detectar *bugs* utilizando o catálogo do Defects4J, ampliar a avaliação para suítes completas de testes, e estender a comparação para as ferramentas discutidas na Seção 5.

Por fim, reforçamos a necessidade de protocolos de avaliação mais abrangentes e padronizados para ferramentas de geração de testes baseadas em LLM. Embora diversos trabalhos proponham novas estratégias de geração, as diferenças entre os critérios de avaliação dificultam a comparação objetiva entre elas e a identificação dos fatores responsáveis pelos ganhos observados.

Disponibilidade de Artefatos

Visando transparência e possibilidade de replicação e reprodução do estudo, disponibilizamos os artefatos usados no apêndice [3].

Referências

[1] Shaukat Ali, Lionel C Briand, Hadi Hemmati, and Rambod K Panesar-Walawege. 2010. A systematic review of the application and empirical investigation of search-based test-case generation. *IEEE Transactions on Software Engineering* 36, 6 (2010), 742–762.

[2] Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272.

[3] Apêndice Online. 2026. https://github.com/Djeymisson/chatunitest-analysis-results

[4] Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024. ChatUniTest: A Framework for LLM-Based Test Generation. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion '24)*. ACM, 572–576.

[5] Léuson Da Silva, Paulo Borba, Toni Maciel, Wardah Mahmood, Thorsten Berger, João Moisakis, Aldiberg Gomes, and Vinícius Leite. 2024. Detecting semantic conflicts with unit tests. *Journal of Systems and Software* 214 (2024), 112070. doi:10.1016/j.jss.2024.112070

[6] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 416–419. doi:10.1145/2025113.2025179

[7] Gordon Fraser and Andrea Arcuri. 2014. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology* 24, 2 (2014).

[8] Gordon Fraser, Matt Staats, Phil McMinn, Andrea Arcuri, and Frank Padberg. 2015. Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 23.

[9] Giovanni Grano, Fabio Palomba, and Harald C. Gall. 2019. An Empirical Study on Test Smells in Automatically Generated Code. In *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 128–138.

[10] Lucas Gren and Vard Antinyan. 2017. On the Relation Between Unit Testing and Code Quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 52–56. doi:10.1109/SEAA.2017.36

[11] Mark Harman. 2007. The Current State and Future of Search Based Software Engineering. In *Future of Software Engineering (FOSE '07)*. IEEE Computer Society, 342–357. doi:10.1109/FOSE.2007.29

[12] Mary Jean Harrold. 1999. Testing evolving software. *Journal of systems and software* 47, 2-3 (1999), 173–181.

[13] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678. doi:10.1109/TSE.2010.62

[14] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[15] Claus Klammer and Albin Kern. 2015. Writing unit tests: It's now or never!. In *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 1–4.

[16] Michael Konstantinou, Renzo Degiovanni, Jie M. Zhang, Mark Harman, and Mike Papadakis. 2025. YATE: The Role of Test Repair in LLM-Based Unit Test Generation. arXiv:2507.18316 [cs.SE] https://arxiv.org/abs/2507.18316

[17] Anji Li, Mingwei Liu, Zhenxi Chen, Zheng Pei, Zike Li, Dekun Dai, Yanlin Wang, and Zibin Zheng. 2026. Knowledge Matters: Injecting Project and Testing Knowledge into LLM-based Unit Test Generation. arXiv:2511.14224 [cs.SE] doi:10.1145/3744916.3787769

[18] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-Based Seed Object Synthesis for Search-Based Unit Testing. In *Proceedings of the 29th ACM Joint Meeting on ESEC/FSE (ESEC/FSE '21)*. 1068–1080. doi:10.1145/3468264.3468619

[19] Zifan Nan, Zhaoqiang Guo, Kui Liu, and Xin Xia. 2025. Test Intention Guided LLM-Based Unit Test Generation. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1026–1038. doi:10.1109/ICSE55347.2025.00243

[20] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. 2016. Will my tests tell me if I break this code?. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery*. ACM, 23–29.

[21] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. 2025. ASTER: Natural and Multi-Language Unit Test Generation with LLMs . In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE Computer Society, Los Alamitos, CA, USA, 413–424. doi:10.1109/ICSE-SEIP66354.2025.00042

[22] Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. tsDetect: an open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1650–1654. doi:10.1145/3368089.3417921

[23] Arkadii Sapozhnikov, Mitchell Olsthoorn, Annibale Panichella, Vladimir Kovalenko, and Pouria Derakhshanfar. 2024. TestSpark: IntelliJ IDEA's Ultimate Test Generation Companion. 5 pages. doi:10.1145/3639478.3640024

[24] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. 2024. An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. *IEEE Transactions on Software Engineering* 50, 1 (2024), 85–105. doi:10.1109/TSE.2023.3334955

[25] Novi Setiani, Ridi Ferdiana, and Rudy Hartanto. 2020. Test Case Understandability Model. *IEEE Access* 8 (2020), 169036–169046. doi:10.1109/ACCESS.2020.3022876

[26] Sina Shamshiri, René Just, José M. Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*. 201–211.

[27] Esdras Silva, Roberta Coelho, and Lyrene Silva. 2025. LLMs as Test Generators: A Comparative Benchmarking Study. In *Anais do XXXIX Simpósio Brasileiro de Engenharia de Software* (Recife/PE). SBC, Porto Alegre, RS, Brasil, 25–36. doi:10.5753/sbes.2025.9618

[28] Philipp Straubinger and Gordon Fraser. 2023. A Survey on What Developers Think About Testing. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 25–36.

[29] Murat Tasarsu. 2026. Test case generation using large language models: a systematic literature review. *Cluster Computing* 29, 4 (2026). doi:10.1007/s10586-026-06021-z

[30] Zejun Wang, Kaibo Liu, Ge Li, and Zhi Jin. 2024. HITS: High-coverage LLM-based Unit Test Generation via Method Slicing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 1258–1268. doi:10.1145/3691620.3695501

[31] Chen Yang, Junjie Chen, Bin Lin, Ziqi Wang, and Jianyi Zhou. 2026. Advancing Code Coverage: Incorporating Program Analysis with Large Language Models. *ACM Trans. Softw. Eng. Methodol.* 35, 5, Article 118 (April 2026), 31 pages. doi:10.1145/3748505

[32] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. 2024. Evaluating and Improving ChatGPT for Unit Test Generation. *Proc. ACM Softw. Eng.* 1, FSE, Article 76 (July 2024), 24 pages. doi:10.1145/3660783

[33] Junwei Zhang, Xing Hu, Cuiyun Gao, Xin Xia, and Shanping Li. 2026. Enhancing Automated Unit Test Generation with Large Language Models: A Systematic Literature Review. *ACM Trans. Softw. Eng. Methodol.* (March 2026). doi:10.1145/3802827 Just Accepted.