UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Departamento de Ciência da Computação

VOLATILIDADE DE TESTES DE SOFTWARE: UMA ANÁLISE EMPÍRICA EM REPOSITÓRIOS OPEN-SOURCE

Pesquisa tecnológica

Aluno: Mateus Henrique Souza Silva
Orientador: André Hora

Belo Horizonte
Junho/2026

SUMÁRIO

1 INTRODUÇÃO

A Engenharia de Software é a disciplina que abrange e estuda todos os aspectos do desenvolvimento de sistemas, desde os estágios iniciais de especificação de requisitos até a manutenção do sistema após a sua implantação [Sommerville 2016]. O processo de produção de software é regido pelo Ciclo de Vida de Desenvolvimento de Software (SDLC), uma estrutura conceitual que organiza as etapas de planejamento, projeto, implementação, teste, implantação e manutenção [Ruparelia 2010]. Nesse contexto, a qualidade de software se manifesta como a capacidade de uma aplicação atender de forma consistente aos seus requisitos funcionais e não-funcionais. A verificação dessa qualidade se torna indispensável na fase de testes, onde o processo busca avaliar se o sistema funciona corretamente, com segurança e eficiência [IBM 2023].

No cenário do desenvolvimento ágil, os testes automatizados, divididos entre unidade, integração e sistema, são cruciais para a garantia da qualidade. No entanto, a dependência crescente dessas suítes expõe um desafio crítico: a manutenibilidade de testes. Ao mesmo tempo que o código de produção ui e é incrementado, os testes devem acompanhar essas mudanças. Essa necessidade de ajustes contínuos gera o problema central desta pesquisa: a **volatilidade de testes**.

A alta volatilidade de testes, caracterizada pela frequência excessiva de alterações, transforma um ativo de segurança em um passivo de alto custo. Testes voláteis são frequentemente frágeis, excessivamente acoplados ou mal escritos (*test smells*), exigindo mudanças mesmo quando os requisitos de negócio não se alteram. Essa instabilidade contraria o princípio ideal de desenvolvimento, resumido na filosofia de engenharia de software do Google: “o teste ideal é inalterável: após escrito, nunca precisa mudar a menos que os requisitos do sistema sob teste mudem” [Winters, Manshreck e Wright 2020].

Quando a volatilidade se torna um problema relevante, os testes geram falsos positivos, reduzindo a confiança da equipe e elevando drasticamente o custo de manutenção do projeto. O tempo gasto em corrigir e manter testes voláteis é tempo subtraído do desenvolvimento de novas funcionalidades, afetando a produtividade e o Tempo de Mercado (*Time to Market*).

1.1 Objetivos Gerais

Este projeto tem como principal objetivo o desenvolvimento de uma ferramenta de extração, análise e quantificação da volatilidade de testes de software em repositórios Git. Além disso, o estudo visa realizar uma avaliação empírica em repositórios *open-source*, categorizando a volatilidade em **positiva** (co-evolução com o código de produção) ou **negativa** (refatoração ou correções isoladas).

1.2 Ojetivos Específicos

Para alcançar o objetivo geral, foram definidos alguns objetivos específicos:

1. **Realizar a travessia de *commits*:** Implementar um algoritmo que percorra os *commits* de forma cronológica para identificar as mudanças realizadas;
2. **Implementar a rastreabilidade estrutural:** Elaborar e implementar um algoritmo de análise utilizando Árvore de Sintaxe Abstrata (AST) e heurísticas de escopo para identificar modificações em testes;
3. **Categorizar a natureza da volatilidade:** Identificar a correlação das alterações com o código de produção através dos *commits* relacionados.

2 REFERENCIAL TEÓRICO

2.1 Engenharia e Qualidade de Software

A Engenharia de Software é uma área da engenharia que estuda todos os aspectos do desenvolvimento de software, desde os estágios iniciais de especificação de requisitos até a manutenção do sistema depois que ele foi colocado em uso [Sommerville 2016]. A Engenharia de Software Moderna é definida como uma disciplina focada em fornecer práticas e princípios para o desenvolvimento produtivo de sistemas complexos [Valente 2020]. Durante o processo de produção, o sistema passa por vários estágios denominados Ciclo de Vida de Desenvolvimento de Software (SDLC).

O SDLC é uma estrutura conceitual que considera todas as etapas envolvidas no desenvolvimento, desde o estudo de viabilidade até a manutenção [Ruparelia 2010]. Nesse contexto, a qualidade de software se manifesta como a capacidade de uma aplicação atender aos requisitos funcionais e não-funcionais. A busca pela qualidade é central em todos os estágios, mas é na fase de testes que sua verificação se torna indispensável. Testes automatizados são a base para a sustentabilidade de projetos ágeis, permitindo refatorações seguras e entregas contínuas [Valente 2020].

2.2 Testes de Software

No cenário da produção de software, a identificação precoce de problemas é crucial. O teste de software é um processo fundamental, focado em avaliar se uma aplicação funciona corretamente, com segurança e eficiência [IBM 2023]. Em metodologias modernas, os testes deixam de ser uma fase final isolada para se tornarem uma atividade contínua integrada ao desenvolvimento [Valente 2020].

Os testes dividem-se em manuais e automatizados. Enquanto os manuais são úteis para testes exploratórios, os automatizados usam *scripts* para validar comportamentos específicos de forma repetível. Os testes automatizados são geralmente estruturados em níveis:

- **Testes de unidade**: Validam o comportamento de componentes isolados, como funções ou métodos [Myers, Badgett e Sandler 2011]. São a base da pirâmide de testes devido à sua rapidez de execução.
- **Testes de integração**: Verificam se várias unidades funcionam corretamente em conjunto, focando na comunicação entre módulos [IBM 2023].
- **Testes de sistema**: Validam o comportamento do sistema como um todo, simulando cenários reais de uso sob a perspectiva do usuário final.

2.3 Volatilidade de Testes

A volatilidade de testes refere-se à frequência com que artefatos de teste sofrem modificações. Embora alguma evolução seja necessária para acompanhar novos requisitos, a volatilidade excessiva indica problemas como fragilidade ou alto acoplamento. Testes que mudam constantemente geram custos elevados de manutenção e reduzem a confiança da equipe na suíte de testes (falsos positivos), impactando diretamente a produtividade e o tempo de mercado [Winters, Manshreck e Wright 2020].

A degradação da suíte de testes tem impactos diretos na agilidade da equipe. [Athanasiou et al. 2014] analisou empiricamente e demonstrou que baixa qualidade e alta complexidade de manutenção de código de testes estão diretamente relacionadas a uma baixa velocidade na resolução de bugs, o que gera maior atraso na entrega de novas funcionalidades.

2.4 Mineração de repositórios

A Mineração de Repositórios de Software (MSR - *Mining Software Repositories*) é uma prática da Engenharia de Software focada na análise de dados gerados a partir do histórico de alterações realizadas durante todo o ciclo de vida da aplicação. Ferramentas de controle de versão (como o Git) armazenam diversos tipos diferentes de metadados sobre essas alterações, como histórico de *commits*, autores, alterações de código (*diffs*) e rastreamento de problemas [Hassan 2008].

A aplicação de técnicas de MSR permite transformar repositórios de código em bases de dados ricas em informações de pesquisa[Kagdi, Collard e Maletic 2009]. Aplicando a técnica, é possível obter informações quantitativas de como o software evolui na prática, independente de entrevistas ou métodos manuais. No contexto desse trabalho, a MSR é utilizada como base metodológica para a extração do histórico de alterações, possibilitando a análise desejada.

2.5 Árvores de Sintaxe Abstrata (AST)

Análises de código-fonte baseada em manipulação textual ou de linhas sofrem limitações estruturais, devido à limitação relacionada à semântica da linguagem. Como forma de contornar esse problema, compiladores e ferramentas de análise estática utilizam Árvores de Sintaxe Abstratas (AST - *Abstract Syntax Tree*), uma estrutura de dados baseada em árvores que representa a sintaxe do código-fonte, focando em declarações de funções, métodos e laços de repetição [Aho et al. 2006].

No cenário de evolução de software, o uso de AST permite o rastreamento em granularidade fina (*fine-grained analysis*). Comparar ASTs permite mapear com precisão quais funções, métodos ou laços sofrem alterações ao longo de vários *commits*, permitindo analisar as alterações sem influência de formatações[Falleri et al. 2014].

3 TRABALHOS RELACIONADOS

A manutenção e a evolução de testes de software têm sido objeto de diversos estudos na área de Engenharia de Software. Esta seção apresenta trabalhos que investigam a qualidade dos testes, o fenômeno da co-evolução entre produção e testes, e o impacto da instabilidade (flakiness) na confiabilidade das suítes.

3.1 Qualidade de Testes e Test Smells

A qualidade do código de teste é um fator determinante para garantir a sua manutenibilidade ao longo do tempo. A redução dessa qualidade é impulsionada pela presença de *test smells*, conceito definido por [**?**] para descrever más práticas de *design* ou antipadrões na construção de testes automatizados. O impacto negativo desse tipo de implementação é um gargalo amplamente reconhecido tanto na literatura acadêmica quanto nas práticas da indústria de software [Garousi e Küçük 2018].

Estudos empíricos, como o conduzido por Palomba et al. [Palomba et al. 2016], demonstram que a presença desses *smells* está fortemente correlacionada com uma maior chance da existência de defeitos e modificações frequentes no código de produção associado. Esses trabalhos são fundamentais para identificar as prováveis causas raízes da volatilidade, sugerindo que testes mal projetados são naturalmente mais frágeis e exigem maior retrabalho por parte dos desenvolvedores.

O presente trabalho complementa essa linha de pesquisa de maneira reversa: enquanto estudos como os de Palomba et al. [Palomba et al. 2016] e Garousi e Küçük [Garousi e Küçük 2018] focam na análise qualitativa e estática da estrutura do código (causa), esta monografia propõe uma abordagem quantitativa para observar as consequências desse fenômeno em larga escala, mensurando a volatilidade bruta (frequência de reescrita em *commits*) ao longo do histórico de desenvolvimento do projeto.

3.2 Co-evolução de Testes e Produção

A relação temporal entre alterações no código de produção e no código de teste é conhecida como co-evolução. Frequentemente, o desenvolvimento de testes não ocorre em sincronia perfeita com o desenvolvimento de funcionalidades, havendo fases de "crescimento de testes"seguidas por estagnação [Zaidman et al. 2011, Pinto, Sinha e Orso 2012]. Este estudo relaciona-se com este trabalho ao destacar que a volatilidade pode ocorrer em picos, dependendo da cultura de desenvolvimento do projeto.

3.3 Testes Instáveis (*Flaky Tests*)

Um subconjunto crítico da volatilidade são os testes instáveis ou *flaky tests*, testes que apresentam resultados diferentes (passam ou falham) sem alterações no código. [Luo et al. 2014] realizaram um estudo extensivo sobre as causas comuns de instabilidade, analisando o histórico de *commits* de projetos da Apache Foundation. Embora o foco de Luo seja na instabilidade de execução (comportamento), seu trabalho dialoga com esta monografia, pois a tentativa de corrigir testes instáveis gera, invariavelmente, uma alta volatilidade no arquivo de teste (muitos *commits* para tentar "acalmar"o teste).

3.4 Evolução Contínua e Manutenção de Testes

Diversos estudos têm se dedicado a entender o esforço associado à manutenção do código de testes ao longo da evolução de um projeto. Casos de teste são constantemente adicionados, modificados e deletados, exigindo um esforço de manutenção contínuo [Pinto, Sinha e Orso 2012, Beller, Gousios e Zaidman 2017]. O estudo focou especificamente em como os testes evoluem.

Além disso, [Beller, Gousios e Zaidman 2017] analisaram repositórios *open-source* e sistemas de CI (Continuous Integration), revelando que a manutenção de testes, geralmente é o maior gargalo de entregas, com falhas de *build* ocasionadas pela fragilidade da suíte e não por erros do código de produção. O presente trabalho visa ampliar essa visão, não apenas analisando o esforço de manutenção, mas também categorizando as alterações entre positiva e negativa e cruzando ecossistemas distintos (Python, JS e TS).

3.5 O Desafio da Granularidade na Mineração

A precisão no estudo de evolução de software é diretamente dependente do nível de granularidade implementado pela ferramenta de mineração. Conforme a análise das abordagens de MSR levantadas por [Kagdi, Collard e Maletic 2009], os métodos existentes de análise de evolução de software se limitam ao nível dos arquivos ou a comparações baseadas em linhas de código bruto (*line-based diffs*). Porém, avaliar o esforço de manutenção de suítes de testes com base nessas abordagens pode esconder muita informação importante, tendo em vista

que um único arquivo de testes pode possuir centenas ou milhares de cenários independentes, mascarando o verdadeiro foco de volatilidade.

Pesquisas como a de Fleuri destacam a importância de se mapear o código por meio de Árvores de Sintaxe Abstrata (AST) para rastrear mudanças estruturais com precisão, evitando ruídos de formatação [Fluri et al. 2007]. Nesse contexto, a ferramenta desenvolvida neste trabalho destaca-se da literatura tradicional de *flaky tests* e co-evolução ao incorporar nativamente o *parsing* da AST das três linguagens alvo. Ao isolar a análise no nível do método de teste individual, a abordagem proposta consegue revelar com precisão os "*hotspots*"do sistema.

3.6 Considerações Finais

Enquanto os trabalhos citados focam em aspectos qualitativos (*smells*), temporais (co-evolução) ou comportamentais (instabilidade), esta monografia propõe uma abordagem prática e quantitativa. O diferencial deste trabalho reside no desenvolvimento de uma ferramenta automatizada capaz de mensurar a volatilidade bruta através da mineração de repositórios em múltiplos ecossistemas (Python, JavaScript/TypeScript), oferecendo uma métrica direta de esforço de manutenção.

4 METODOLOGIA

Para atingir os objetivos propostos, este trabalho adotou uma abordagem quantitativa baseada na Mineração de Repositórios de Software (MSR). Foi desenvolvida uma ferramenta automatizada para extrair e analisar dados históricos de repositórios hospedados no GitHub.

4.1 Ferramentas Utilizadas

A ferramenta de extração foi implementada na linguagem Python, utilizando a biblioteca **PyDriller** [Spadini, Aniche e Bacchelli 2018]. O PyDriller é um *framework* robusto para mineração de repositórios Git, escolhido por facilitar a navegação entre *commits*, a extração de modificações em arquivos e a análise de *diffs*, abstraindo a complexidade dos comandos Git de baixo nível. O código fonte do *framework* e sua documentação estão disponíveis publicamente [Spadini, Aniche e Bacchelli 2018]. E para o mapeamentos dos métodos em uma AST, foi utilizado o módulo **ast** nativo do Python.

Além disso, para a busca de repositórios em larga escala e baseado em alguns critérios de filtragem, foi utilizada a ferramenta Github Search [Dabic, Aghajani e Bavota 2021]. Essa ferramenta permite a listagem repositórios Git através de filtros simples, como número de estrelas, quantidade de *commits*, data de criação, entre outros.

4.2 Processo de Análise

O algoritmo desenvolvido opera em quatro etapas sequenciais:

1. **Coleta e Clonagem**: A ferramenta recebe um arquivo .txt com uma lista de URLs, divididas uma por linha. Para otimizar o tempo de execução, utiliza-se processamento paralelo (*multiprocessing*) para clonar repositórios em ambientes isolados temporários.

   Dada a magnitude dos dados e a complexidade computacional envolvida na mineração de repositórios, que envolve a clonagem, a travessia de milhares de *commits* e a análise de diferenças de código (*diffs*), a performance foi um requisito não-funcional crítico. A implementação padrão do interpretador Python (CPython) possui uma limitação conhecida como Global Interpreter Lock (GIL), que restringe a execução de código a um único núcleo de Central Processing Unit (CPU) por vez, tornando o uso de *threads* ineficiente para tarefas intensivas de processamento (*CPU-bound*).

   Para superar essa limitação e maximizar a utilização do hardware disponível, a ferramenta utilizou a biblioteca nativa *multiprocessing*. Esta biblioteca permite a criação de processos independentes, cada um com seu próprio espaço de memória e instância do interpretador, possibilitando o paralelismo verdadeiro em arquiteturas *multi-core*.

   A arquitetura da solução baseou-se no padrão de Pool de Processos (multiprocessing.Pool). O algoritmo detecta o número de núcleos lógicos disponíveis na máquina hospedeira (*cpu_count*) e instancia um número equivalente de processos trabalhadores (*workers*). A lista de repositórios a serem analisados é então distribuída dinamicamente entre esses trabalhadores. Essa abordagem garantiu que múltiplos repositórios fossem clonados e analisados simultaneamente, reduzindo o tempo total de execução de forma linear em relação ao número de núcleos disponíveis.

2. **Filtragem e Identificação**: Durante a varredura do histórico, aplica-se uma heurística de nomenclatura para identificar arquivos de teste. No contexto deste estudo foram aplicados os padrões dos *frameworks* de testes para cada uma das linguagens suportadas:

   - **Python**: Arquivos que seguem o padrão `text_*.py` ou `*_test.py`
   - **JavaScript**: Arquivos que seguem o padrão `*.spec.js` ou `*.text.js`
   - **TypeScript**: Arquivos que seguem o padrão `*.spec.ts` ou `*.text.ts`

   A heurística de identificação de arquivos de teste baseou-se nas convenções de nomenclatura predominantes na indústria para cada ecossistema analisado. A Tabela 1 resume os padrões adotados pela ferramenta, correlacionando-os com os *frameworks* de teste.

   Além da identificação dos arquivos, os métodos são mapeados de forma fina e individual através do uso das ASTs, permitindo a identificação e contabilização das alterações com maior precisão.

Tabela 1: Padrões de Nomenclatura e Frameworks de Teste por Linguagem

Linguagem	**Padrão de Arquivo**	**Framework**
Python	`test_*.py` `*_test.py`	`unittest`, `pytest`
JavaScript	`*.test.js` `*.spec.js`	Jest, Mocha Jasmine, Mocha (BDD)
TypeScript	`*.spec.ts` `*.test.ts`	Jest, Jasmine Jest, Vitest

3. **Classificação Heurística e Mapeamento via AST**: O ponto principal do trabalho consiste em analisar a suíte de testes a nível de métodos, mapeando o esforço empregado em cada um individualmente. Esta etapa foi dividida em dois principais pontos:

    - **Classificação da Natureza da Volatilidade (Positiva/Negativa)**: A mensagem de *commit* é extraída e analisada através de Expressões Regulares (Regex) para encontrar palavras-chave que indicam correções no código de produção (*fix*, *bug*, *resolve*, *patch* e *issue*). Caso seja encontrada alguma dessas palavras, todas as modificações de teste daquele *commit* são classificadas como **Volatilidade Positiva**. Caso contrário, entende-se que as modificações se tratam de refatorações, atualizações de dependências ou evolução isolada da suíte, sendo classificada como **Volatilidade Negativa**.
    - **Rastreamento Granular via Árvore de Sintaxe Abstrata (AST)**: Para cada *commit*, a ferramenta analisa o *diff*, submete o código fonte alterado para o *parser* de AST e identifica os métodos alterados no *commit*. Por fim, é feita a verificação se as linhas presentes no *diff* estão dentro do escopo de algum método de teste identificado anteriormente. Se identificada a interseção, a volatilidade do teste é incrementada.

    Vale ressaltar que a mineração de repositórios é sensível a ruídos inerentes à prática do desenvolvimento. Conforme o estudo conduzido por Herzig e Zeller [Herzig e Zeller 2013], desenvolvedores frequentemente realizam "*tangled code changes*", ou seja, misturam alterações de refatoração com correções de bugs em um mesmo *commit*. Portanto, a heurística aplicada neste trabalho busca analisar o intuito principal da mensagem de *commit*.

4. **Persistência**: Os dados consolidados são salvos em formato .csv na pasta "***results***", contendo informações sobre cada método de teste avaliado, como "Total de modificações", "Volatilidade Positiva", "Volatilidade Negativa".

5 RESULTADOS E DISCUSSÃO

5.1 Caracterização da Amostra

Para garantir a robustez estatística e permitir uma análise comparativa profunda, o estudo avaliou um conjunto de 90 repositórios de software (83 com sucesso) *open-source* amplamente utilizados. A amostra abrange três ecossistemas distintos (Python, JavaScript, TypeScript) e diferentes domínios de aplicação, desde *frameworks* web até bibliotecas de computação científica.

Ao final da análise realizada, foram mapeados **583.643** métodos e mais de **1.8M** de modificações, o que permitiu extrair diversos dados e métricas.

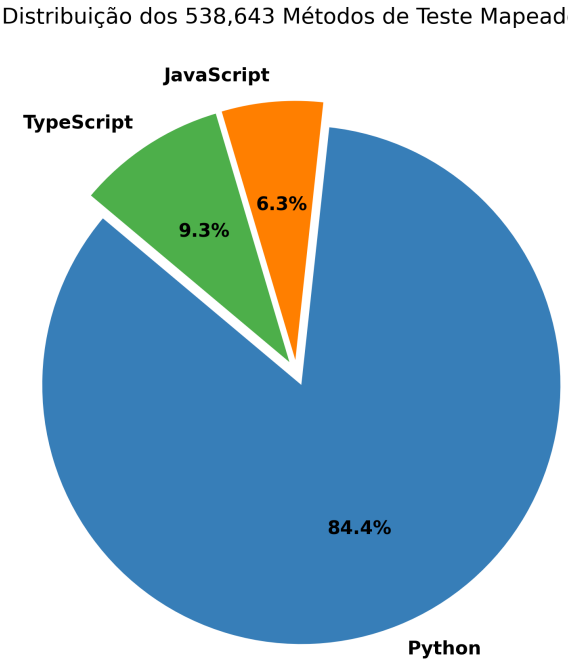

Figura 1: Distribuição de testes no dataset

Como pode ser observado na Figura 1, a grande discrepância no volume de testes do ecossistema Python (representando 84,4% da amostra) chama a atenção, porém pode ser explicada por fatores inerentes à arquitetura dos repositórios. Diferente da cultura do ecossistema JavaScript/TypeScript, que favorece a fragmentação de bibliotecas em micro-pacotes no NPM, o Python concentra repositórios caracterizados como monolitos massivos (ex: Pandas e Django) que, devido ao seu tamanho e complexidade, aumentam proporcionalmente a quantidade de métodos necessários para cobrir todos os cenários de forma satisfatória. Aliado a isso, o uso extensivo de bibliotecas Python para cálculos em Ciência de Dados gera uma explosão de cenários de teste, aumentando exponencialmente o tamanho das suítes analisadas.

5.2 Natureza da Volatilidade: Correção vs. Refatoração

Compreender o volume absoluto de modificações em testes de software é apenas uma parte do problema. O ponto principal é compreender a motivação por trás dessas modificações. A ferramenta desenvolvida neste trabalho permitiu categorizar o esforço de manutenção em duas categorias distintas: **Volatilidade Positiva** (testes alterados em conjunto com resoluções de falhas e defeitos em produção) e **Volatilidade Negativa** (esforço direcionado à refatoração, evolução arquitetural ou correções isoladas da própria suíte de testes).

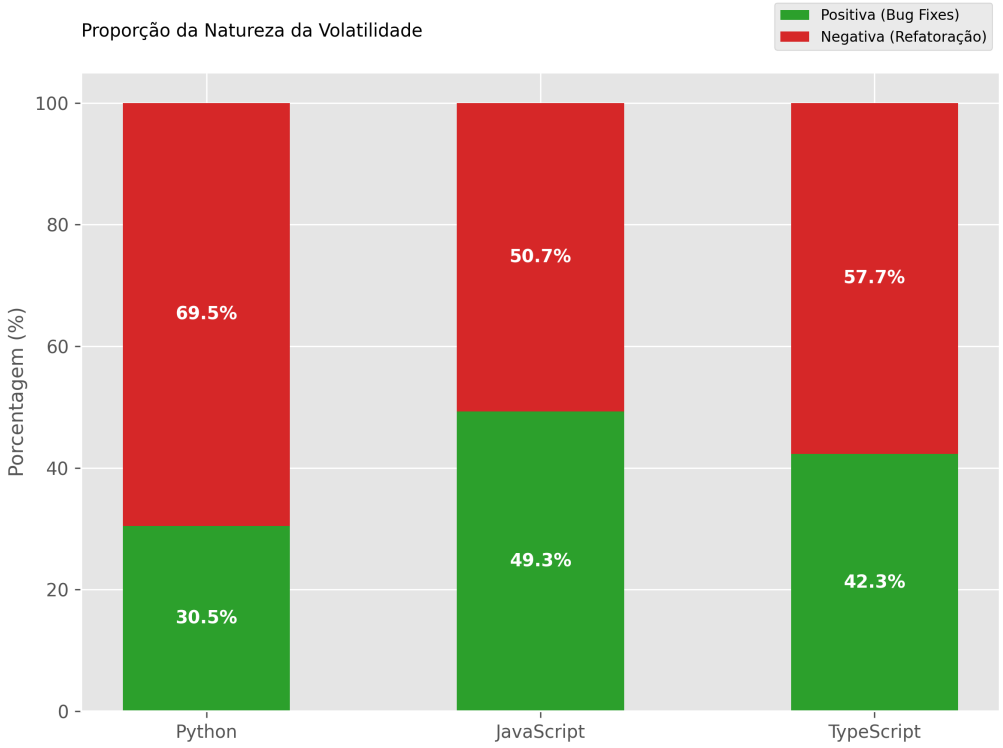

Figura 2: Proporção da Natureza da Volatilidade por Ecossistema

Como visto na Figura 2, a análise da distribuição da natureza por ecossistema revela que a manutenção de testes é um fenômeno que tem comportamentos diferentes dependendo da plataforma escolhida, gerando os seguintes pontos de reflexão:

- **O Perfil Estrutural do Python**: O Python apresentou a maior discrepância entre as categorias, com 69,5% de todo o esforço de manutenção voltado para a Volatilidade Negativa. Essa diferença pode ser explicada porque os grandes projetos *open-source* de Python analisados possuem bases de dados massivas e complexas. Portanto, o esforço da equipe de desenvolvimento nesses projetos tende a ser mais preventivo do que corretivo. Os testes mudam não porque a lógica de negócios quebrou, mas porque a própria estrutura de testes precisa ser constantemente otimizada, modularizada ou adaptada para novos *frameworks* de testes.

- **O Perfil Reativo da Web (JavaScript e TypeScript)**: Já nas linguagens voltadas para o desenvolvimento web demonstraram um equilíbrio considerável, onde a Volatilidade Positiva chega a bater as marcas de 49,3% em JavaScript (JS) e 42,3% em TypeScript (TS). Esse dado mostra a fragilidade do ecossistema NPM e do desenvolvimento para navegadores. Nessas linguagens, quase metade de todo o esforço de reescrita de testes ocorre de forma reativa, ou seja, um bug real afeta a produção e os testes precisam ser ajustados imediatamente em conjunto com o código defeituoso.

O alto índice de volatilidade positiva encontrado nas linguagens focadas em desenvolvimento Web (49% em JavaScript e 42% em TypeScript) comprova empiricamente o fenômeno da co-evolução discutida por Zaidman [Zaidman et al. 2011]. Os dados demonstram que os desenvolvedores Frontend precisam acompanhar a evolução do código de produção de forma eficiente, em resposta às alterações realizadas.

Em resumo, podemos analisar que a volatilidade demonstrou uma diferença nas propostas de cada linguagem. Enquanto o Python, mais focado em *backend*, processamento de dados e operações matemáticas, sofre mais com refatorações internas, enquanto o ecossistema NPM, focado na web, tem um perfil mais relativo a alterações de bibliotecas, interfaces, entre outros.

5.3 A Desigualdade da Manutenção de Testes

Na Engenharia de Software, assume-se frequentemente de forma equivocada que o esforço de manutenção de código é distribuído de forma homogênea. Para investigar a veracidade desta afirmação no contexto de testes, este estudo aplicou duas métricas de distribuição estatística sobre a quantidade de modificações mapeadas: o Princípio de Pareto (Regra 80/20) e o Coeficiente de Gini.

Enquanto trabalhos clássicos como o de [Pinto, Sinha e Orso 2012] já quebraram a ideia de que testes são artefatos imutáveis, devido ao fato de que eles precisam evoluir constantemente junto com o código de produção, este trabalho busca quantificar como essa evolução acontece. Os dados de Pareto e Gini (Figura 3 e 4) mostram que essa evolução ocorre, porém de forma não homogênea em toda a suíte de testes, sendo concentrada em alguns "*hostpots*"que concentram a grande maioria das modificações.

A Figura 3 ilustra a porcentagem média da suíte de testes que foi responsável por concentrar 80% de todo o esforço histórico de manutenção em cada linguagem.

Os dados revelam que a manutenção de testes no mundo real afasta-se parcialmente do "Pareto Ideal"(a linha teórica dos 20%). Nos três ecossistemas analisados, a métrica se manteve entre 50% e 57%. Analisando de forma mais geral, esse é um dado significativo: ele comprova que aproximadamente metade de todos os testes escritos em um projeto praticamente nunca sofrem alterações significativas após sua concepção. Em contrapartida, a outra metade atua como "*hotspots*", sendo responsável pela maioria (80%) dos custos de reescrita e correção ao longo de anos de desenvolvimento.

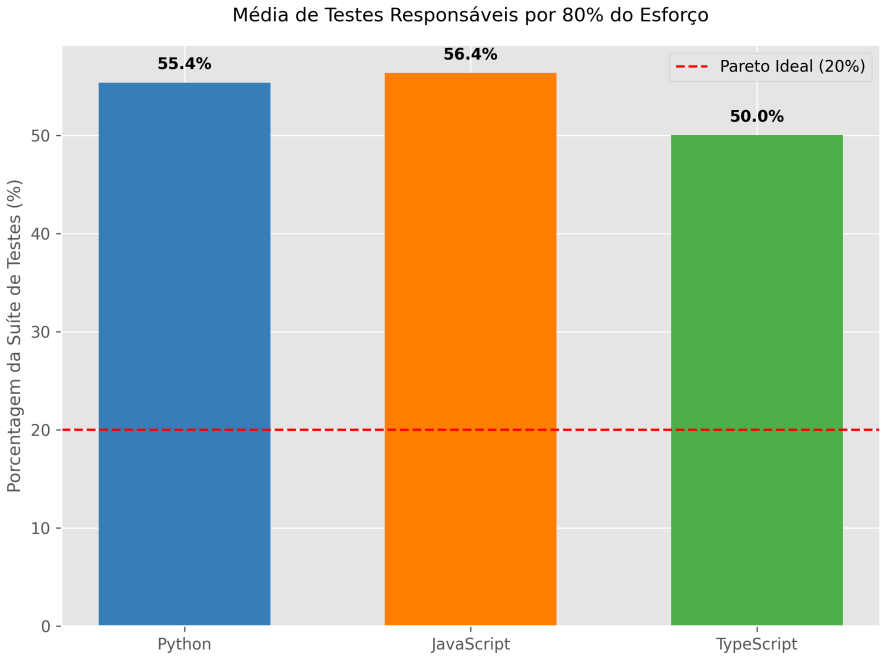

Figura 3: Prova empírica do Princípio de Pareto aplicado a Testes de Software

Para refinar matematicamente essa constatação de desigualdade, a distribuição do esforço em cada repositório foi submetida ao cálculo do Coeficiente de Gini, cujo resultado é apresentado no modelo da Figura 4.

O gráfico mostra uma visão não trivial sobre sistemas de tipagem forte. Ao pensar nesses sistemas, é intuitivo imaginar que eles possuam uma suíte de testes estável, por serem feitos em uma linguagem baseada em contratos e interfaces. Porém, o Typescript foi a linguagem com a maior centralização de testes encontrada.

Isso significa que o TypeScript concentra a volatilidade de maneira extremamente agressiva em arquivos muito específicos. Uma análise qualitativa sugere que esses "*hotspots*"em TS geralmente consistem em instâncias de validadores de tipo e *mocks* de interfaces. Quando o domínio da aplicação muda, a tipagem estrita impede que o projeto compile até que todos os contratos de teste vinculados àquela interface sejam reescritos e estejam consistentes, isso centraliza e muito a manutenção para arquivos específicos. Já os repositórios em Python e JavaScript, com suas tipagens dinâmicas e maior flexibilidade, demonstram uma desigualdade mais branda e um custo de manutenção melhor distribuído pelas suítes.

6 CONCLUSÃO

A manutenção de testes automatizados é um desafio crítico que afeta a velocidade da entrega e os custos da Engenharia de Software. Este trabalho demonstrou a implementação e aplicação de uma ferramenta que permite analisar de forma automática o esforço necessário para manter esses testes. A travessia de *commits* em repositórios Git, *parsing* de Árvores de

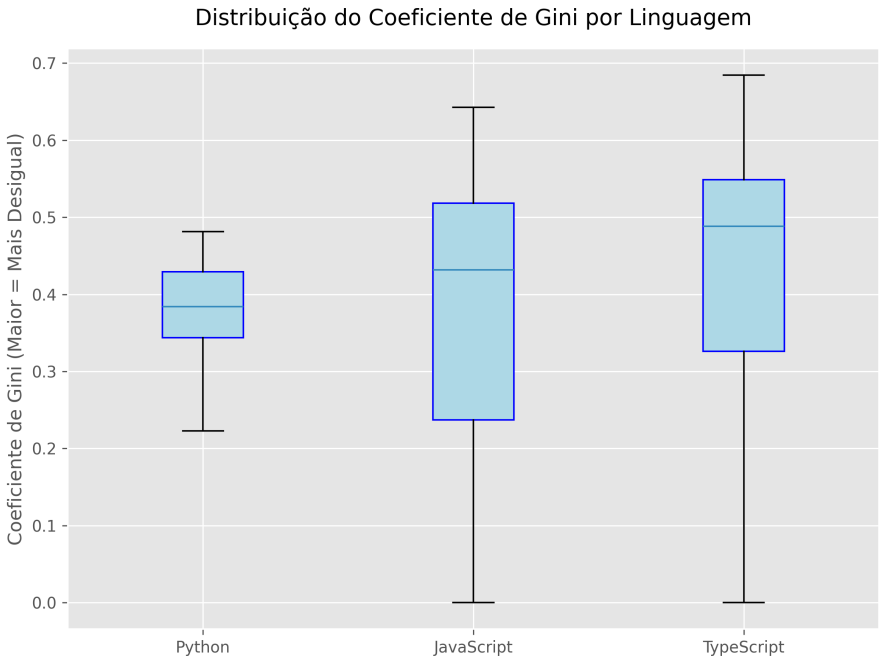

Figura 4: Distribuição do Coeficiente de Gini por Linguagem

Sintaxe Abstrata (AST) e heurísiticas de mensagens de *commit* permitiram extrair e categorizar a volatilidade em granularidade fina de mais de 583 mil testes em 83 grandes repositórios *open-source*.

Os resultados obtidos permitiram ter uma visão mais realista da evolução e manutenção de testes. Uma das constatações realizadas é que apenas 50% da suíte, concentra mais de 80% de todo o esforço empregado para tal finalidade, o que nos permite comprovar a desigualdade nesse cenário. Além disso, pudemos observar que o paradigma envolvido no ecossistema escolhido tem grande peso na volatilidade analisada.

Adicionalmente, o Coeficiente de Gini permitiu verificar que linguagens de tipagem estática podem ter o efeito inverso do que é esperado delas. Apesar de garantirem maior segurança entre os contratos, elas fazem com que o esforço de manutenção fique mais centralizado em poucos "*hotspots*", gerando gargalo.

7 TRABALHOS FUTUROS

Considerando as possibilidades abertas por este estudo, propõe-se para trabalhos futuros:

- **Expansão Multiparadigma**: Estender os *parsers* de AST e heurísticas de identificação de testes da ferramenta para englobar diferentes linguagens de tipagem forte com compilação corporativa pesada (como Java e C#).
- **Avaliação Qualitativa Mista**: Realizar entrevistas estruturadas (*surveys*) com os autores

e mantenedores dos métodos classificados como "*hotspots*"na mineração (ex: contribuidores do Playwright ou Pandas). Cruzar os dados quantitativos gerados pela ferramenta com o contexto humano permitiria compreender o peso da cultura organizacional na decisão de conviver com testes altamente voláteis.

REFERÊNCIAS

[Aho et al. 2006]AHO, A. V. et al. *Compilers: Principles, Techniques, and Tools*. 2nd. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[Athanasiou et al. 2014]ATHANASIOU, D. et al. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, IEEE, v. 40, n. 11, p. 1100–1125, 2014.

[Beller, Gousios e Zaidman 2017]BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. Oops, my tests broke the build: An explorative analysis of travis ci with github. In: *Proceedings of the 14th International Conference on Mining Software Repositories*. [S.l.: s.n.], 2017. p. 356–367.

[Dabic, Aghajani e Bavota 2021]DABIC, O.; AGHAJANI, E.; BAVOTA, G. Sampling projects in github for MSR studies. In: *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021*. [S.l.]: IEEE, 2021. p. 560–564.

[Falleri et al. 2014]FALLERI, J.-R. et al. Fine-grained and accurate source code differencing. In: ACM. *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. [S.l.], 2014. p. 313–324.

[Fluri et al. 2007]FLURI, B. et al. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, IEEE, v. 33, n. 11, p. 725–743, 2007.

[Garousi e Küçük 2018]GAROUSI, V.; KÜÇÜK, B. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*, Elsevier, v. 138, p. 52–81, 2018.

[Hassan 2008]HASSAN, A. E. The road ahead for mining software repositories. In: IEEE. *2008 Frontiers of Software Maintenance*. [S.l.], 2008. p. 48–57.

[Herzig e Zeller 2013]HERZIG, K.; ZELLER, A. The impact of tangled code changes. In: IEEE. *Proceedings of the 10th Working Conference on Mining Software Repositories*. [S.l.], 2013. p. 121–130.

[IBM 2023]IBM. *O que é teste de software? Guia para iniciantes*. 2023. `https://www.ibm.com/br-pt/topics/software-testing`. Acessado em: 09 out. 2025.

[Kagdi, Collard e Maletic 2009]KAGDI, H.; COLLARD, M. L.; MALETIC, J. I. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of software maintenance and evolution: Research and practice*, Wiley Online Library, v. 19, n. 2, p. 77–131, 2009.

[Luo et al. 2014]LUO, Q. et al. An empirical analysis of flaky tests. In: ACM. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. [S.l.], 2014. p. 643–653.

[Myers, Badgett e Sandler 2011]MYERS, G. J.; BADGETT, T.; SANDLER, C. *The Art of Software Testing*. 3. ed. New Jersey: John Wiley & Sons, 2011.

[Palomba et al. 2016]PALOMBA, F. et al. An empirical study of test smells and their relationship with software quality. In: IEEE. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.], 2016. p. 1–10.

[Pinto, Sinha e Orso 2012]PINTO, G.; SINHA, S.; ORSO, A. Understanding myths and realities of test-suite evolution. In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. [S.l.: s.n.], 2012. p. 1–11.

[Ruparelia 2010]RUPARELIA, N. B. Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 35, n. 3, p. 8–13, 2010.

[Sommerville 2016]SOMMERVILLE, I. *Software Engineering*. 10. ed. Boston: Pearson Education, 2016.

[Spadini, Aniche e Bacchelli 2018]SPADINI, D.; ANICHE, M.; BACCHELLI, A. Pydriller: Python framework for mining software repositories. In: *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. [S.l.: s.n.], 2018.

[Valente 2020]VALENTE, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade*. Belo Horizonte: Independente, 2020. Disponível em: <https://engsoftmoderna.info/>.

[Winters, Manshreck e Wright 2020]WINTERS, T.; MANSHRECK, T.; WRIGHT, H. *Software Engineering at Google: Lessons Learned from Programming Over Time*. Sebastopol, CA: O'Reilly Media, 2020.

[Zaidman et al. 2011]ZAIDMAN, A. et al. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Softw. Engg.*, Kluwer Academic Publishers, USA, v. 16, n. 3, p. 325–364, jun. 2011. ISSN 1382-3256. Disponível em: <https://doi.org/10.1007/s10664-010-9143-7>.